- attachEvent 事件监听方式
- 对不同浏览器的兼容性处理
- 删除事件（解绑事件）
- 传统方式删除事件
- 监听方式删除事件 removeEventListener
- 监听方式删除事件 detachEvent
- 对不同浏览器的兼容性处理
- DOM 事件流
- 代码验证
- 事件对象
- 常见事件对象的属性和方法
- this 和 target
- 阻止默认行为
- 阻止冒泡（重点）
- 事件委托
- 常用的鼠标事件
- 禁止鼠标右键菜单（contextmenu）
- 禁止鼠标选中（selectstart 开始选中）
- 鼠标事件对象
- 常用的键盘事件
- 键盘事件对象
- 操作元素
- 改变元素内容
- **innerText**
- **innerHTML**
- 修改或添加属性内容
- 表单元素的属性操作
- 修改或添加CSS样式属性
- 使用 className 修改元素类名的方式修改CSS样式属性
- 对自定义属性值的操作
- 获取自定义属性值 getAttribute()
- 添加和修改自定义属性值
- 移除自定义属性值
- H5自定义属性新规范
- 节点操作
- 节点概述
- 节点的使用
- 创建节点并添加节点
- 删除节点
- 复制节点
- 三种创建元素方式的区别
- DOM 重点核心（复习）
- 创建
- 增
- 删
- 改
- 查
- 属性操作
- 事件操作

BOM

- 什么是 BOM
- BOM 的构成
- 页面加载事件
- 调整窗口大小事件
- 定时器

ES基础

注释

在 JavaScript 中有2种注释：

1. 单行注释：`// 默认快捷键 ctrl + /`
2. 多行注释：`/* 默认快捷键 shift + alt + a */`

输入输出语句

方法	说明	归属
alert(msg)	浏览器弹出警示框	浏览器
console.log(msg)	浏览器控制台打印输出信息	浏览器
prompt(info)	浏览器弹出输入框，用户可以输入	浏览器

变量

变量是程序在内存中开辟的一块用于存放数据的空间。

声明变量

在 JavaScript 中变量可以使用`var`进行声明，也可以省略`var`直接进行使用。

```
// 单行
var test;
var test01 = 1;
test02 = 1;
var test03 = 'i am str';
test04 = 'i am str';
// 多行
var test = 18,
    test02 = 'i am str';
```

注意：

- 仅声明变量，但是未给变量赋值，控制台中输出为`undefined`.
- 未声明变量，在控制台中调用输出将报错。

变量的命名规范

- 由字母(A-Za-z)、数字(0-9)、下划线()、美元符号($)组成，如：usrAge，num01，_name
- 严格区分大小写。var app;和var App;是两个变量
- 不能以数字开头。18age是错误的
- 不能是关键字、保留字。例如：var、for、while
- 变量名必须有意义。MMD BBD nl → age
- 遵守驼峰命名法。首字母小写，后面单词的首字母需要大写。myFirstName
- 推荐翻译网站：有道 爱词霸

数据类型

JavaScript 中的数据类型类似于 Python，都是一种弱类型。他们在声明变量时，不用声明数据类型。而是在程序运行的过程中，数据类型会被自动确认。

在运行代码时，变量的数据类型是由 JS引擎根据 = 右边变量值的数据类型来判断的，运行完毕后，变量就确定了数据类型。

数据类型的分类：

1. 简单数据类型：Number，String，Boolean，Undefined，Null
2. 复杂数据类型：Object

简单数据类型

简单数据类型	说明	默认值
Number	数字型，包含整型和浮点型	0
Boolean	布尔值型，true / false	false
String	字符串型 字符串需要带引号	“”
Undefined	声明了变量但是没有负值此时变量就为Undefined	undefined
Null	声明变量为空值	null

关于数字型

1. 数字型直接赋值是使用的十进制
2. 在需要的数字前加 `0` 是使用八进制 `var a = 010`
3. 在需要的数字前加 `0x` 是表示十六进制 `var a = 0x10`

在控制台中输出的都是十进制

4. 数字型中的最大值 `Number.MAX_VALUE`
5. 数字型中的最小值 `Number.MIN_VALUE`
6. 数字型的无穷大 `Number.MAX_VALUE * 2` 比任何值都大
7. 数字型的无穷小 `Number.MIN_VALUE * 2` 比任何值都小
8. NaN，Not a Number，代表一个非数值，当的不出我们想要的结果并且不是个数字时用其表示，如 `‘test’ - 100`

isNaN()

这个方法用来判断非数字，如果是数字返回 `false` 否则返回 `true`

字符串型

字符串类型使用单引号和双引号都可以，但是**推荐用单引号**

当使用到引号嵌套时，要使用不同的引号，外部是单引号内部就用双引号，外部是双引号内部就用单引号

转义字符

转义符	说明
\n	换行
\\	斜杠\
\'	单引号
\”	双引号
\t	tab 缩进
\b	空格，b 是 blank 的意思

字符段长度 length

```
var a = 'i am str';
console.log(a.length);
```

字符串拼接

```
console.log('沙漠'+'骆驼');
// out: 沙漠骆驼
console.log('永远'+18);
// out: 永远18
console.log('永远'+18+'岁');
// out: 永远18岁
```

总结：数值相加，字符相连

布尔型

```
var flag = true,
    flag1 = false;
console.log(flag + 1); // true 参与运算当 1 来看
console.log(flag1 + 1); // false 参与运算当 0 来看
```

undefined 和 null

```
var str;
console.log(str); // 输出 undefined
var v1 = undefined;
console.log(v1 + 'test'); // 输出 undefinedtest
console.log(v1 + 1); // 输出 NaN

var space = null;
console.log(space + 'test'); // 输出 nulltest
console.log(space + 1); // 输出 1 ,因为什么都没有+1还是1
```

数据类型检测 typeof

```
var num = 10;
var str = 'brokyz';
console.log(typeof num);
console.log(typeof str)
```

数据类型的转换

转换为字符串

方式	说明	案例
toString()	转字符串	var num = 1; alert(num.toString());
String() 强制转换	转字符串	var num = 1; alter(String(num))
加号拼接字符串	和字符串拼接的结果都是字符串	var num = 1; alter(num + “)

转换为数字型

方式	说明	案例
parseInt(string) 函数	将string类型转换为整数数值型	parseInt(‘78’)
parseFloat(string) 函数	将string类型转换为浮点数数值型	parseFloat(‘3,14’)
Number() 强制转换函数	将string类型转换为数值型	Number(‘12’)
js 隐式转换 （- * /）	利用算数运算隐式转换 (注意不能用+)	‘12’ - 0

转换为布尔

```
console.log(Boolean('')); // false
console.log(Boolean(0)); // false
console.log(Boolean(NaN)); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false

// 只有上面五种情况为 false 其余情况都为 true

console.log(Boolean('123')); // true
console.log(Boolean('你好')); //true
```

运算符

算数运算符

运算符	描述	实例
+	加	
-	减	
*	乘	
/	除	
%	取余	5 % 3 = 2; 3 % 5 = 3;

浮点数运算时会有问题

```
console.log(0.1 + 0.2); // 0.30000000000000000004
```

由于浮点数运算时会转换成二进制，用二进制进行算术运算，这时就会存在一些误差

所以要尽量避免小数参与运算

前置递增运算符

`++num` 类似于 `num = num + 1` 但是更加简单

前置的运算顺序是，先自加后返回值

后置递增运算符

`num++` 类似于 `num = num + 1` 但是更加简单

前置自增和后置自增如果单独使用，效果一样

但是后置自增的运算顺序是。先返回原值，后自加

```
var num = 10;
console.log(++num + 10) // 这里返回21，运算后num变为11

var num = 10;
console.log(num++ + 10) // 这里返回20，运算后num变为11
```

比较运算符

比较运算返回的是布尔类型，一般的符号一般默认会转换数据类型进行比较

运算符	说明	案例	结果
<	小于		
>	大于		
>=	大于等于		
<=	小于等于		
==	等于		
!=	不等于		
=== !==	全等 要求值和数据类型都一致		

逻辑运算符

逻辑运算符	说明	案例
&&	“逻辑与”，简称“与” and	
||	”逻辑或“，简称”或“ or	
!	”逻辑非“，简称“非” not	

短路运算（逻辑中断）

原理：当有多个表达式（值）时，左边的表达式值可以确定结果时，就不再继续运算右边的表达式的值；

- 逻辑与

  如果第一个表达式的值为真，则返回表达式2

  如果第一个表达式的值为假，则返回表达式1

```
console.log(123 && 456); // 结果456
console.log(0 && 456); // 结果0
console.log(0 && 1 + 2 && 456 * 7) //结果0 ，在0之后的停止执行
```

- 逻辑或

  如果第一个表达式的值为真，则返回表达式1

  如果第一个表达式的值为假，则返回表达式2

```
console.log(123 && 456); // 结果123
console.log(123 && 1 + 2 && 456 * 7); // 结果123, 后面的不运行，已经中断了
console.log(0 && 1 + 2 && 456 * 7) //结果3 ，在1+2之后的停止执行
```

赋值运算符

`num += 2` 等同于 `num = num +2`

赋值运算符	说明	案例
=		
+=、-=		
*=、/=、%=		

运算优先级

优先级	运算符	顺序
1	小括号	()
2	一元运算符	++ -- !
3	算数运算符	先乘除后加减
4	关系运算符	> >= < <=
5	相等运算符	== ！ = === ！ ===
6	逻辑运算符	先&&后||
7	赋值运算符	=
8	逗号运算符	,

流程控制

控制代码按照一定的结构顺序来执行

有三种结构：顺序就够、分支结构、循环结构

if 语句

```
if (条件表达式){

}
```

if else 语句

```
if(条件){

}else{

}
```

else if 语句

```
if(条件){

}else if(条件2){

}else if(条件3){

}else{

}
```

三元表达式

由三元运算符组成的式子

```
条件表达式 ? 表达式1 : 表达式2
// 如果条件表达式为真，则返回表达式1的值，否则返回表达式2的值
```

switch 语句

```
switch(表达式){
    case value1:
        语句1;
         break;
    case value2:
           语句2;
        break;
    default:
        最后的语句;
}
```

当表达式必须全等case时才可以匹配成功

switch 和 else if 区别

- 一般情况两个语句可以相互替换
- switch 通常处理 case 比较确定值的情况；而 else if 更加灵活，可以用于范围判断；
- switch 语句进行条件判断后直接执行到程序的条件语句，效率高；而 else if 得依次判断前面的条件。
- 当分支比较少时，else if 语句执行效率更高。
- 分支比较多时，switch 执行效率更高。

循环

for

```
for(i=0;i<10;i++){

}
```

while

```
while(条件){

}
```

do while

```
do{

}while(条件)
```

continue

跳过本次循环，并执行下一次循环

break

跳过所有循环

数组

数组是**一组数据的集合**，其中每个数据被称为**元素**，在数组中可以**存放任意类型的元素**。数组是一种将一组数据存储在单个变量名下的优雅方式。

数组的创建方式

- 使用 new 创建数组

```
var arr = new Array(); // 创建一个空数组
```

- 利用数组字面量来创建数组

```
var arr = [];
var arr = ['brokyz',1,2,true]; // 数组里面的数据称为数组元素
// 数组声明了并赋值，称为数组的初始化
```

获取数组中的元素

索引（下标）：用来访问数组元素的符号（数组下标从 0 开始）

```
var arr = ['brokyz',1,2,true];
console.log(arr[0]) // 输出 brokyz
console.log(arr[4]) // 输出 undefined 因为数组中没有第五个元素
```

遍历数组

把数组中元素从头到尾访问一遍

```
// arr.length 获取数组长度
for (i = 0; i < arr.length; i++){
    console.log(arr[i]);
}
```

新增数组元素

- 通过修改 length 长度来修改

```
        // 通过改变 length 新增数组元素
        var arr = [1, 2, 3];
        console.log(arr.length);
        arr.length = 4;
        console.log(arr.length);
        console.log(arr);
```

- 通过修改索引号追加数组

```
        // 通过修改索引号新增数组元素
        var arr = [1, 2, 3];
        arr[3] = 4;
        console.log(arr);
```

- 不要直接给数组赋值 `arr = 'brokyz'` 这样数组就直接被替换了

函数

当一段代码需要重复被调用的时候，那么就需要用函数对代码进行封装

函数不调用是，自己不执行

函数中可以调用另一个函数，允许互相调用

声明函数

- 利用 function 关键字进行声明

```
function sayHi(name){
    console.log('Hi, ' + name);
}
```

- 通过 var 声明函数

```
var sayHi = function(name){
    console.log('Hi, ' + name);
}
```

- 声明函数中的参数为形参

调用函数

```
sayHi(brokyz);
```

- 调用函数时的参数为实参

形参和实参的匹配问题

在其他语言，如 Java 中，形参和实参要求必须匹配；

但是在 JS 中，实参并不需要匹配实参的数量；调用时，实参仅仅根据自身个数对形参的前面几个进行匹配；如果调用时，实参的个数小于形参的个数，那么少的那个形参被定义为 undefined 结果为 NaN

函数返回值

函数使用 return 返回值

```
function sayHi(name){
    return 'Hi, ' + name;
}
```

- 在函数中，执行完 return 函数就停止执行

```
function sayHi(name){
    return  'Hi', name; // 返回结果时最后一个值
}
```

- return 只能返回一个值
- 但需要返回多个值时，可以用数组的形式返回
- 函数中如果存在 return 那么返回的就是指定的值，如果不存在 return 那么返回的是 undefined

arguments 的使用

当函数中没有设置形参时，但是调用时传入了实参，这时可以在函数中通过 arguments 接受；

也就是说 arguments 中存取了所有传入的实参；

arguments 是一种伪数组，不是真正意义上的数组，它具有数组的 length 属性，按照索引的方式进行存储，但是他没有数组的一些方法，比如 pop()、push()

作用域

作用域就是代码变量在哪个范围内起作用

在 es6 之前有：全局作用域 局部作用域

全局作用域在整个 script 标签或者单独的 js 文件中起作用

注意：在函数内部，没有声明直接赋值的变量属于全局作用域，如 `num = 10`

局部作用域（函数作用域）在函数内部起作用

全局变量和局部变量的区别

1. 全局变量只有在浏览器关闭的时候才会销毁，比较占用内存资源
2. 局部变量当我们程序执行完毕就会销毁，比较节约内存资源

块级作用域

在 es6 之前没有块级作用域，在 es6 新增了块级作用域

用 {} 包含的就是块级作用域

由于 es6 之前没有块级作用域，所以在{}中写的变量，在{}外面也可以使用

作用域链

当函数中又声明了一个函数时，就出现了作用域链的问题

作用域链：内部函数访问非自身变量，采取的是链式查找的方式来决定选取哪个值，这种结构叫作用域链

```
var num = 10;

function fn(){ // 外部函数
    var num = 20;
    function fun(){ // 内部函数
        console.log(num)
    }
}
```

当出现函数嵌套时，内部函数优先去外部函数查找需要使用的变量，如果存在，就使用外部函数中的变量，如果不存在就去外部函数之外寻找所使用的变量，如果存在就使用外部函数之外的变量

预解析

js 引擎运行 js 代码分为两步：

1. 预解析：js 引擎会把 js 里面的 var 和 function 提升到当前作用域的最前面
2. 执行代码：代码按照预解析之后的顺序，从上往下执行

预解析分为：

- 变量预解析（变量提升）：把所有的变量声明提升到当前作用域最前面，但是不提升赋值操作
- 函数预解析（函数提升）：把所有的函数声明提升到当前作用域最前面，不调用函数

例子：

```
// 源代码
var num = 10;
fun();
function fun() {
    console.log(num);
    var num = 20;
}

// js引擎预解析之后
```

```
var num;
function fun() {
    var num;
    console.log(num); // 由于作用域链，输出的是上面那个只定义的num
    num = 20;
}
num = 10;
fun();
// 输出为 undefined
```

对象

对象就是一个**具体事物**，在 JS 中对象是一组无序的属性和方法的集合，所有的事物都是对象，例如字符串、数值、数组、函数等

对象是由**属性**和**方法**组成的。

- 属性：事物的特性，在对象中用属性来表示
- 方法：事物的行为，在对象中用方法来表示

为什么需要对象

当我们保存一个值的时候，可以使用变量保存多个值，也可以使用数组保存多个值。但是如果我们要保存一个事物的完整信息时候，用对象存储就非常的方便

对象的创建

1. 利用字面量创建的对象

对象的字面量为 {}

```
var obj = {
    uname: 'brokyz',
    agr: 18,
    sex: 'male',
    sayHi: function(){
        console.log('hi');
    }
}
```

2. 利用 new Object 创建对象

```
var obj = new Object();
obj.uname = 'brokyz';
obj.age = 18;
obj.sex = 'male';
obj.sayHi = function(){
    console.log('hi');
}
```

3. 使用构造函数创建对象

前两个方法只能创建一个对象，我们可以利用构造函数重复相同的构造

构造函数的名字首字母要大写

构造函数不需要 return 就可以返回结果

```
function Apple(name){
    this.name = name;
    this.sayHi = function(){
        console.log('hi');
    }
}
var iphone = new Apple('iphone12');
console.log(iphone.name);
console.log(iphone['name']);
```

调用对象

```
console.log(obj.uname); // 调用对象的属性
obj.sayHi() // 调用对象的方法
```

构造函数和对象的区别

- 构造函数：抽取了对象的公共部分，封装到了函数里它泛指了一大类
- 创建对象：特指某一个，通过 new 关键字创建对象的过程我们也称为对象实例化

new 关键字的执行逻辑

1. new 构造函数可以在内存中创建一个空的对象
2. this 就会指向刚才创建的空对象
3. 执行构造函数里的代码，给空对象添加属性和方法
4. 返回这个对象

遍历对象

使用 for...in 遍历对象

```
for(var k in obj){
    console.log(k); // 得到的是属性名和方法名
    console.log(obj[k]) // 得到的是属性值
}
```

内置对象

在 JS 中除了我们自己定义创建的对象以外，还存在一些 JS 自带的对象，这为我们提供了一些常用的功能，我们直接拿来使用即可。

Math 对象

Math 对象不是一个构造函数，不需要我们使用 new 去创建，我们直接只用里面的属性和方法即可。

这里介绍几个常用的属性和方法，具体属性和方法可以查看 MDN文档

圆周率 Math.PI

这个是就是我们的圆周率 Π

```
console.log(Math.PI)
```

最大值和最小值

求最大值和最小值。

如果给定的参数中至少有一个参数无法被转换成数字，则会返回 NaN。

如果没有参数，则结果为 -Infinity。

```
console.log(Math.max(1, 3, 2));
console.log(Math.min(1, 3, 2));
```

向下取整和向上取整

```
Math.floor() // 向下取整
Math.ceil() // 向上取整
```

四舍五入取整

```
Math.round() // 四舍五入，注意 当为-3.5时是，-3
```

绝对值

```
Math.abs()    // 绝对值
// 存在隐士转换，会把字符型能转换成数字的转换为数字型
```

随机数（重点）

`Math.random()` 函数返回一个浮点，伪随机数在范围 [0,1)，我们可以对其进行缩放，得到我们想要的范围。

```
// 取到 [min,max) 范围
Math.random() * (max - min) + min;
// 取到 [min,max) 的整数
Math.floor(Math.random() * (max - min)) + min;
// 取到 [min,max] 的整数
Math.floor(Math.random() * (max - min + 1)) + min;
```

Date 日期对象

Date 是一个构造函数，我们只能通过 Date 构造函数来实例化对象。

具体使用产看 MDN文档

```
// 如果没有参数，返回系统的当前时间
var date = new Date();
```

```
// 如果有参数，则返回参数的格式化时间，支持一下参数写法
var birthday = new Date('December 17, 1995 03:24:00');
var birthday = new Date('1995-12-17T03:24:00');
var birthday = new Date(1995, 11, 17); // 从0开始数，这里是1995年 12月 17日
var birthday = new Date(1995, 11, 17, 3, 24, 0);

console.log(Date.now())    // 输出自Unix诞生到现在的毫秒数
console.log(Date.parse('04 Dec 1995 00:12:00 GMT')) // 输出自Unix诞生到指定时间的毫秒数
console.log(date.getFullYear()) // 获得 date 的年
console.log(date.getMonth()+1)    // 获得 date 的月。由于月从0算起，所以加一
console.log(date.getDate())    // 获得 date 的日。
console.log(date.getHours()) // 获得 date 的时。
console.log(date.getMinutes())    // 获得 date 的分。
console.log(date.getSeconds())    // 获得 date 的秒。
console.log(date.getDay()) // 获得 date 时当前周的第几天，从0其，0是第一天。
// 同理如果get换为set就是设置
date.setMonth(0) // 设置date月份为1月
```

Array 数组对象

JavaScript 的 `Array` 对象是用于构造数组的全局对象，数组是类似于列表的高阶对象。

详情使用方法请查看 MDN文档。

数组的两种创建方式

```
// 通过字面量创建数组
var arr = [1,2,3];
// 通过构造函数创建
var arr = new Array(3);    // 创建了一个长度为3的空数组
var arr = new Array(1,2,3) // 创建了一个数组为[1,2,3],等价于第一种方法；这种写法必须是两个元素以上才行。
```

检测数组的两种方式

```
// instanceof 运算符用来检测是否为数组
console.log([1,2] instanceof Array);    // 同理可以用于检测其他类型
// Array.isArray() 方法
console.log(Array.isArray(arr))
```

添加元素

```
var arr = [1,2,3];
// 使用push()进行尾插入
arr.push(4, 5);    // 将4，5尾插入数组arr，返回新数组的长度，原数组会发生变化
// 使用unshift()进行首插入
arr.unshift(-1.0)    // 将-1,0首插入数组arr，返回新数组的长度，原数组会发生变化
```

删除数组元素

```
var arr = [1,2,3,4,5,6];
// 使用pop()进行尾删除，每次只能删除一个元素，返回的是删除的元素，原数组发生变化
arr.pop();
// 使用shift()进行首删除，每次只能删除一个元素，返回的是删除的元素，原数组发生变化
arr.shift();
```

数组排序

```
const array1 = ['one', 'two', 'three'];
// reverse()翻转数组，不改变原数组
const reversed = array1.reverse();    //out:"reversed:" Array ["three", "two",
"one"]
// sort()对数组进行排序 冒泡方法，不改变原数组
var arr = [1, 30, 'March', 'Jan', 'Feb', 'Dec',4, 21, 100000];
console.log(arr.sort()); // [1, 100000, 21, 30, 4, "Dec", "Feb", "Jan", "March"]
这种不带参数只根据第一个字符进行排序
// 升序排列
console.log(arr.sort(function(a,b){
    return a-b;
}));
// 降序排列
console.log(arr.sort(function(a,b){
    return b-a;
}));
```

数组索引的获取

```
var arr = [1,2,3,4,5,1,2];
// indexOf()从前往后查找
console.log(arr.indexOf(2));    // 返回数组中指定元素的第一个索引
// lastIndexOf()从后往前查找
console.log(arr.lastIndexOf(2));
```

数组去重（重点）

自写函数

```
function unique(arr){
    var newArr = [];
    for (var i = 0; i < arr.length; i++) {
        if (newArr.indexOf(arr[i]) === -1) {
            newArr.push(arr[i]);
        }
    }
    return newArr;
}
```

利用`Set`对象

```
var arr = [1,2,3,1,2,3,1,2,3];
var set = new Set(arr);
console.log(set);
```

数组转字符串

```
var arr = [1,2,3];
// toString()转换字符串，以逗号分隔
console.log(arr.toString());
// join()转换为字符串，默认为逗号，可以指定分隔符
console.log(arr.join('&'));
```

指定返回对象

```
const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];
// 使用slice()得到数组，不会改变原数组
console.log(animals.slice(2));
// expected output: Array ["camel", "duck", "elephant"]
console.log(animals.slice(2, 4));
// expected output: Array ["camel", "duck"]
console.log(animals.slice(-2));
// expected output: Array ["duck", "elephant"]
console.log(animals.slice(2, -1));
// expected output: Array ["camel", "duck"]
// 注意是左闭右开
```

在指定位置删除和添加元素

```
const months = ['Jan', 'March', 'April', 'June'];
// splice(start,del,element) star:从第几个元素开始 del：向后删除几个元素，element:在这个位置前面添加element。 改变原数组
months.splice(1, 0, 'Feb'); // 从index为1的元素March开始，删除0个元素，在其前面插入Feb
console.log(months);
// expected output: Array ["Jan", "Feb", "March", "April", "June"]
months.splice(4, 1, 'May'); // 从index为4的元素开始，删除1个元素，在前面插入May
// replaces 1 element at index 4
console.log(months);
// expected output: Array ["Jan", "Feb", "March", "April", "May"]
months.splice(4, 1); // 从index为4的元素开始，删除1个元素.
// expected output: Array ["Jan", "Feb", "March", "April"]
```

合并数组

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2,array2); // 不改变原数组
console.log(array3);
// > Array ["a", "b", "c", "d", "e", "f", "d", "e", "f"]
```

String 字符串对象

`String` 全局对象是一个用于字符串或一个字符序列的构造函数。

基本包装类型

只有对象才会有属性和方法 这是复杂数据类型才会有的。

而字符串是简单数据类型，但是可以**通过基本包装类型。将其转换为复杂数据类型。**

基本包装类型就是，把简单数据类型包装称为复杂数据类型，这样基本数据类型就有了属性和方法。

```
var str = 'andy';
console.log(str.length);

// 上面两个代码的内部执行逻辑为
// 1. 把简单数据类型转换为复杂数据类型
var temp = new String('andy');
// 2. 把临时变量的值给 str, 这样str就成为了一个对象
str = temp;
// 3. 销毁临时变量
temp = null;
```

字符串的不可变性

```
// 在内存中开辟了一个内存空间用来存储andy，让声明的str指向andy的内存地址
var str = 'andy';
// 在内存中又新开辟了一个内存空间来存储red，这是改变str指向red的内存地址，但是andy依然存在于内存中，其内存地址依然存在，并不会进行销毁
str = 'red';
// 上面过程就是字符串的不可变性，所以我们尽量避免大量的字符串拼接操作
```

根据字符返回位置

字符串的所有方法，都不会改变字符串本身（字符串是不可变的），操作完成会返回一个新的字符串。

```
var str = 'brokyzbrokyz';
console.log(str.indexOf('k'));    // 3,只输出第一个
console.log(str.indexOf('k',4))    // 9, 指定从3后开始查找
```

查找所有位置

```
var str = 'brokyzbrokyzbrokyz';
var index = str.indexOf('b');
var num = 0;
while(index !== -1) {
    num++;
    console.log(index);
    index = str.indexOf('o',index+1)
}
console.log(num);
```

根据位置返回字符

```
var str = 'andy';
// charAt() 根据位置返回索引号
console.log(str.charAt(3));
// 遍历字符
for(var i = 0; i < str.length; i++) {
    console.log(str.charAt(i));
}
// charCodeAt(index) 返回相应索引号的字符的ASCLL值 常用于判断用户按下了哪个按键
console.log(str.charCodeAt(3))

// H5新增
console.log(str[3])
```

拼接和截取字符串

方法名	说明
concat(str1,str2,str3...)	concat() 方法用于连接两个或多个字符串。拼接字符串，等效于+，+更常用
substr(start,length)	从start开始，向后去length个个数
slice(start,end)	从start开始，截取到end位置，end取不到
substring(start,end)	从start开始，截取到end位置，end取不到，基本和slice相同，但是不支持负号

替换字符串

```
// 使用replace('被替换的字符','替换的字符') 只会替换第一个字符
var str = 'brokyzbrokyz';
console.log(str.replace('z','s'));
// 替换所有
while (str.indexOf('z') !=== -1) {
    str = str.replace('z','s');
}
```

字符串转换为数组

```
var str = '1,2,3,4,5';
// 使用split('分隔符') 转换为数组
console.log(str.split(','));
```

大小写转换

```
// toLowerCase() 将字符串转换为小写
console.log('中文简体 zh-CN || zh-Hans'.toLowerCase());    // 中文简体 zh-cn || zh-hans
// toupperCase() 将字符串转换为大写
console.log('The quick brown fox jumps over the lazy dog.'.toUpperCase());
// expected output: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."
```

JS 简单数据类型和复杂数据类型

简单数据类型又叫做基本数据类型或者**值类型**，复杂数据类型又叫做**引用类型。**

- 值类型：简单数据类型，在存储时变量中存储的时值本身，因此叫做值类型。
  - string，number，boolean，undefined，null
  - 简单数据类型null，返回的时一个空对象 object。如果有个变量我们以后打算存储为对象，但是现在不需要，这个时候就可以赋值为null。
- 引用类型：复杂数据类型，在存储时变量中存储的仅仅是地址（引用），因此叫做引用类型。
  - 通过 new 关键字创建的对象（内置对象、自定义对象等），如 Object、Array、Date等。

堆和栈

1. 栈（操作系统）：由操作系统自动分配释放存放函数的参数值、局部变量的值等。其操作方式类似于数据结构中的栈。

简单数据类型存在在栈里

2. 堆（操作系统）：存储复杂类型（对象），一般由程序员分配释放，若程序员不释放，由垃圾回收机制回收。

复杂数据类型存放到堆里面

注意：JS 中没有堆栈的概念，通过堆栈的方式们可以让大家更容易理解代码的一些执行方式，便于将来学习其他语言。

简单数据类型的内存分配

值类型变量的数据直接存放在变量（栈空间）中

比如`var age = 18;`

会在栈中开辟一个内存空间，里面存放了18这个值，然后让 age指向18的内存地址。

复杂数据类型的内存分配

复杂数据类型的存储和栈堆都有关系。

比如 `var arr = [1,2,3];`

会在堆里面开辟一个空间，存放了[1,2,3]，然后在栈里面会存放堆的内存地址，之后 arr 会指向栈里面的内存地址，然后由栈中的内存地址指向堆里面存放的数据。

简单数据类型传参

简单类型传参同一变量不会开辟新的内存地址。

比如`var a = 1; a = 2;`

会在栈中开辟一个内存空间用来存放1这个值，然后由a指向栈中1对应的内存地址。当让a变为2时，并不会开辟新的内存空间来存放2，而是a指向栈中的内存地址不发生改变，其内存地址存放的值由1改为2。

但是如果是string，就得考虑字符串的不可变性了。

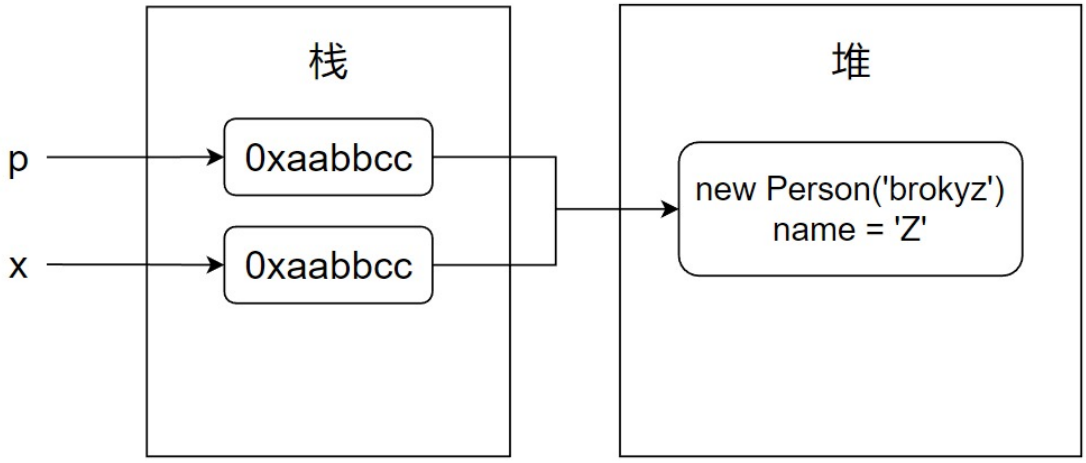
复杂数据类型传参

有如下代码：

```
function Person(name) {
    this.name = name;
}

funtion f1(x) {
    console.log(x.name);
    x.name = "Z";
    console.log(x.name);
}

var p = new Person("brokyz");
console.log(p.name);
f1(p);
console.log(p.name);
```



1. 首先通过`new`关键字给变量`p`创建了一个`Person`对象，初始化`name = 'brokyz'`。这时在堆中开辟了一个内存空间，将创建的`Person`对象存在堆中，在栈中开辟了一个内存空间，这个内存空间中存放了堆中`Person`的内存地址的16进制的值，然后由变量`p`指向栈中存放`Person`的内存地址值。这样一来变量`p`先是指向了栈中存放的堆内存地址值，然后这个值指向了堆中的`Person`对象。
2. 当我们调用函数`f1(p)`时，这时将实参`p`传递给形参`x`。这时在栈中重新开辟了一个内存地址，存放了和实参`p`一样的值，然后形参`x`指向这个栈中新开辟的值。也就是说，此时`p`和`x`指向的是栈中不同的内存地址，但是内存地址的存放的值都是指向同一个`Person`的堆内存地址。在函数中通

过`x.name = 'z'`修改了`Person`的属性值，是`p`和`x`同时指向的`Person`，所以这时候输出`p.name`时，其属性也发生了改变。

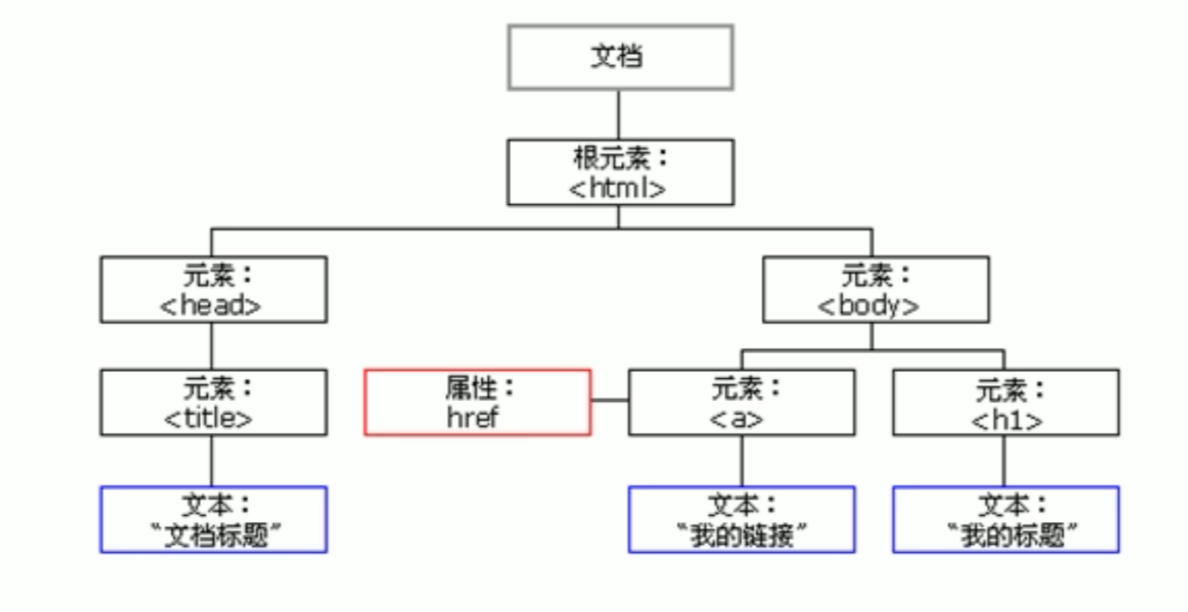
DOM

什么是DOM?

文档对象模型（Document Object Model），是 W3C 组织推荐的处理可拓展标记语言（HTML或者 XML）的编程接口。

W3C 已经定义了一系列的 DOM 接口，通过这些接口可以改变网页的内容、结构和样式。

DOM 树



- 文档：一个页面就是一个文档，DOM 中使用 document 表示
- 元素：页面中的所有标签都是元素，DOM中使用 element 表示
- 节点：网页中的所有内容都是节点（标签、属性、文本、注释等），DOM 中使用 node 表示

DOM 把以上内容都看成是对象

获取元素

代码	归属	描述
document.getElementById(‘nav’)	DOM 老方法 不推荐	通过元素的 id 属性值获取元素，只返回第一个元素
document.getElementByTagName(‘div’)	DOM 老方法 不推荐	通过元素的标签名获取元素，将选择到的所有对象存入伪数组并返回
document.querySelector(‘.nav li’)	H5 新增方法 推荐	通过选择器的方法获取元素，只返回第一个对象
document.querySelectorAll(‘.nav li’)	H5 新增方法	通过选择器的方法获取元素，将选择

代码	推荐	描述
document.getElementByClassName('nav')	H5 新增方法	通过元素的类名获取元素，将选择到的所有对象存入伪数组并返回，不兼容ie
document.body		获取整个文档的body部分
document.documentElement		获取整个文档，即html部分

根据元素 ID 获取 getElementById()

- 此方法为 DOM 老 API，兼容性好，但是不推荐开发时使用。
- 此方法只返回获取到的第一个元素对象。
- 语法：

```
// 获取 id 为 time 的第一个元素
var timer = document.getElementById('time');
// 返回获取到的整个元素标签
console.log(timer);
// 返回获取到的元素类型
console.log(typeof timer);
// 在控制台中显示指定 JavaScript 对象的属性，并通过类似文件树样式的交互列表显示，更好的查看属性和方法
console.dir(timer);
```

通过标签名获取 getElementByTagName()

- 此方法为 DOM 老 API，兼容性好，但是不推荐开发时使用。
- 此方法将获取到的对象存放于伪数组，以伪数组的形式返回所有获取的元素。
- 如果获取到的元素只有一个，依然返回伪数组；如果没有获取到元素，返回的是一个空的伪数组、
- 语法：

```
// 获取页面中所有 li
var lis = document.getElementsByTagName('li');
// 输出伪数组
console.log(lis);
// 输出伪数组中的第2个对象
console.log(lis[1]);
// 遍历
for (i = 0; i < lis.length; i++){
    console.log(lis[i]);
}
```

根据类名来获取 getElementByClassName()

- 此方法为 HTML5 新增的方法，仅仅兼容ie9+，平时不常用。
- 此方法将获取到的对象存放于伪数组，以伪数组的形式返回所有获取的元素。
- 如果获取到的元素只有一个，依然返回伪数组；如果没有获取到元素，返回的是一个空的伪数组。
- 语法：

```
var box = document.getElementsByClassName('box');
console.log(box);
```

根据选择器获取 querySelector()

- 此方法为 HTML5 新增的方法，仅仅兼容ie9+，推荐使用。
- 此方法只返回获取到的第一个元素对象。

```
var box = document.querySelector('.box');
console.log(box);
```

根据选择器获取 querySelectorAll()

- 此方法为 HTML5 新增的方法，仅仅兼容ie9+，推荐使用。
- 此方法将获取到的对象存放于伪数组，以伪数组的形式返回所有获取的元素。
- 如果获取到的元素只有一个，依然返回伪数组；如果没有获取到元素，返回的是一个空的伪数组

```
var box =  document.querySelectorAll('.box');
console.log(box);
```

获取 body 和 html

- 在文档中元素的获取中，有专门获取 body 和 html 的方法
- 语法：

```
// 获取body
var body = document.body;
console.log(body);
console.dir(body);
// 获取整个html
var htmlEle = document.documentElement;
console.log(htmlEle);
```

事件操作

网页中的每个元素都可以产生某些可以触发 JS 的事件，例如，我们可以在用户点击某按钮时产生一个事件，然后去执行某些操作

事件有三部分组成：事件源、事件类型、事件处理程序，我们将其称为事件三要素

1. 事件源：事件的触发对象（谁触发了事件？）
2. 事件类型：事件如何被触发 什么事件 比如鼠标点击（onclick）还是鼠标经过 还是键盘按下
3. 事件处理程序：事件被触发要执行的操作

注册事件

给元素添加事件，称为**注册事件**或者**绑定事件**。

注册事件有两种方式：**传统方式**和**方法监听注册方式**。

1. 传统注册方式
    - 使用 on 开头的事件 onclick

    - `<button onclick="alert('hi')"></button>`
    - btn.onclock = function(){}
    - 特点：注册事件的**唯一性**

- 同一个元素同一个事件只能设置一个处理函数，最后注册的处理函数将会覆盖前面注册的处理函数。

2. 监听注册方式
    - w3c 标准 推荐方式
    - addEventListener() 他是一个方法
    - IE9之前的IE版本不支持此方法，可以使用attachEvent() 代替
    - 特点：同一个元素同一个事件可以注册多个监听器

传统方式注册事件

事件	描述
onclick	点击事件
onfocus	获得焦点事件
onblur	失去焦点事件
onmouseover	鼠标经过
onmouseout	鼠标离开
onmousemove	鼠标移动
onmouseup	鼠标弹起
onmousedown	鼠标按下

语法：

```
// 获取按钮元素
var btn = document.querySelector('button');
// 给按钮元素添加点击事件
btn.onclick = function(){
    // 事件处理程序，但被点击时执行
    console.log('按钮被点击');
}
```

addEventListener 事件监听方式

```
eventTarget.addEventListener(type, listener, useCapture)
```

eventTarget.addEventListener() 方法将指定的监听器注册到 evenTarget (目标对象) 上，当该对象触发指定事件时，就会执行事件处理函数。

该方法有三个参数：

- type：事件类型字符串，比如 click、mouseover，注意这里并没有on
- listener：事件处理函数，事件发生时，会调用该监听函数
- useCapture：事件流，默认为false。flase为冒泡阶段，true为捕获阶段。

代码实例：

```
// 给元素btn添加两个监听鼠标点击事件
btn.addEvenListener('click',function(){
    alert(22);
})
btn.addEvenListener('click',function(){
    alert(33);
})
```

attachEvent 事件监听方式

```
eventTarget.attachEvent(type, callback)
```

该方法将指定的监听器注册到指定的对象上，当对象被触发，就会执行指定的回调函数。

该方法只支持ie9之前的ie浏览器，并不被其他浏览器支持。

- type：事件类型字符串，比如 onclick、onmouseover，这里要带有 on
- callback：事件处理函数，当目标触发事件时回调函数被调用

代码实例：

```
btn.attachEvent('onclick', function(){
    alert(11)
})
```

对不同浏览器的兼容性处理

这里写一个兼容性函数，来适配不同浏览器的兼容性问题（PS：IE就是个毒瘤）

兼容性要首先照顾大多数浏览器，再处理特殊浏览器

```
function addEventListener(element, eventName, fn){
    // 判断当前浏览器是否支持 addEventListener
    if(element.addEvenListener){
        element.addEventListener(eventName, fn);
    }else if(element.attachEvent){
        element.attachEvent('on' + eventName, fn);
    }else{
        // 相当于 element.onclick = fn
        element['on' + eventName] = fn;
    }
}
```

删除事件（解绑事件）

传统方式删除事件

直接给指定的对象的相应的事件赋值为null，即可删除该事件。

```
eventTarget.onclick = null;
```

监听方式删除事件 removeEventListener

```
eventTarget.removeEventListener(type, listener, useCapture)
```

- type：事件类型字符串，比如 click、mouseover，注意这里并没有on
- listener：需要解绑的事件处理函数
- useCapture：事件流，默认为false。flase为冒泡阶段，true为捕获阶段。

代码实例：

```
btn.addEventListener('click', fn);
function fn() {
    alert(22);
    btn.removeEventListener('click', fn);
}
```

注意：如果要删除监听事件，那么添加监听事件时一定要用具体的函数名，不能使用匿名函数。使用匿名函数时，再删除事件时候，无法指定事件触发函数。

监听方式删除事件 detachEvent

此方法对应 attackEvent

```
eventTarget.detachEvent(type, callback)
```

- type：事件类型字符串，比如 onclick、onmouseover，这里要带有 on
- callback：要删除的事件处理函数

代码实例：

```
btn.attachEvent('onclick', fn);
function fn() {
    alert(33);
    btn.detachEvent('onclick', fn);
}
```

对不同浏览器的兼容性处理

```
function removeEventListener(element, eventName, fn){
    // 判断当前浏览器是否支持 removeEventListener
    if(element.removeEvenListener){
        element.removeEventListener(eventName, fn);
    }else if(element.detachEvent){
        element.detachEvent('on' + eventName, fn);
    }else{
        // 相当于 element.onclick = fn
        element['on' + eventName] = null;
    }
}
```

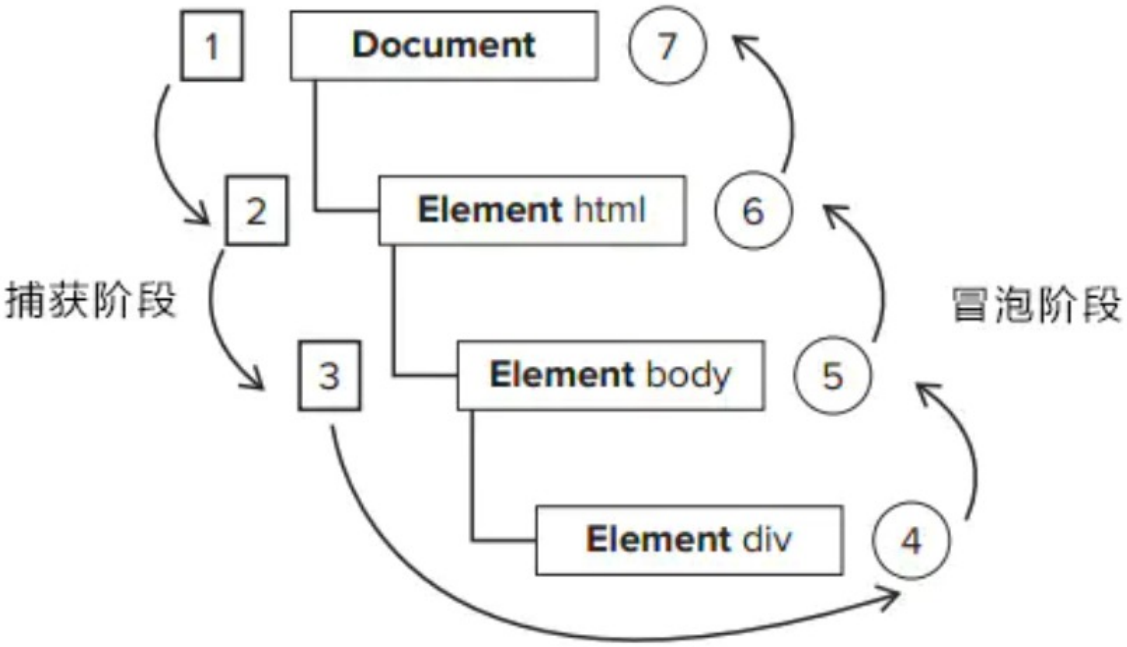
DOM 事件流

事件流描述的是**从页面接收事件的顺序。**

事件发生时会在元素节点之间按照特定的**顺序传播**，这个传播过程称为 DOM 事件流。

DOM 事件流分为3个阶段：

1. 捕获阶段
2. 当前目标阶段
3. 冒泡阶段



事件冒泡：IE最早提出，事件开始时由最具体的元素接收，然后逐级向上传播到 DOM 最顶层节点的过程。

事件捕获：网景最早提出，由 DOM 最顶层节点开始，然后逐级向下传播到最具体的元素接收的过程。

代码验证

1. JS 代码只能执行捕获或者冒泡的其中一个阶段。
2. onclick 和 attachEvent 只能得到冒泡阶段。
3. addEventListener中的第三个参数如果是 true，表示再事件捕获阶段调用事件处理程序；如果是 false（不写默认时false）则表示再事件冒泡阶段调用事件处理程序。

演示地址

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>事件流</title>
    <style>
        .father {
            width: 300px;
```

```
            height: 300px;
            margin: 10px auto;
            background-color: antiquewhite;
            overflow: hidden;
        }

        .son {
            width: 200px;
            height: 200px;
            margin: 50px auto;
            background-color: aliceblue;
        }
    </style>
</head>

<body>
    <div class="father">father
        <div class="son">
            son 冒泡阶段 触发事件处理程序
        </div>
    </div>
    <div class="father">father2
        <div class="son">
            son2 捕获阶段 触发事件处理程序
        </div>
    </div>
    <script>
        var son = document.querySelectorAll('.son');
        // 指定事件在冒泡阶段触发事件处理程序
        son[0].addEventListener('click', function () {
            alert('son');
        })
        son[0].parentNode.addEventListener('click', function () {
            alert('fatehr');
        })

        // 指定事件在捕获阶段触发事件处理程序
        son[1].addEventListener('click', function () {
            alert('son');
        }, true)
        son[1].parentNode.addEventListener('click', function () {
            alert('fatehr');
        }, true)
    </script>
</body>

</html>
```

事件对象

事件对象只有有了事件才会存在，他是系统给我们自动创建的，不需要我们传递参数。

事件对象是我们事件的一系列的相关数据的集合，跟事件相关的，比如鼠标点击里面就包含了鼠标的相关信息，如鼠标坐标等。如果是键盘事件，那么里面就包含的键盘事件的信息，比如判断用户按下了哪个键。

事件对象可以由我们自己命名，比如 event、evt、e。

对 ie678 有兼容性问题，ie678 不可自己对事件对象命名，只能使用`window.event`

代码：

```
div.onclick = function(event) {
    console.log(event);
}
```

兼容性处理：

```
div.onclick = function(e) {
    e = e || window.event;
    console.log(e);
}
```

常见事件对象的属性和方法

属性	描述
e.target	返回触发事件的对象 不兼容ie678
this	返回的是绑定事件的对象
e.srcElement	返回触发事件的对象 兼容ie678
e.currentTarget	和this相似 但是不兼容ie678
e.type	返回事件的类型，比如 click、mouseover 不带on
e.cancelBubbel	阻止冒泡 ie678不兼容
e.returnValue	阻止默认事件（默认行为）非标准 比如不让链接跳转 ie678不兼容
e.preventDefault()	阻止默认事件（默认行为）标准
e.stopPropagation()	阻止冒泡 标准

this 和 target

如果我们有一个 ul>li，给 ul 绑定一个点击事件。

那么当我们点击 li 的时候：

e.target 返回的是触发事件的对象，也就是 li，e.target 不兼容ie678

this 返回的是绑定事件的对象，也就是 ul

对于这两个方法我们还有两个额外的相似的方法：

e.srcElement 对应 e.target，但是前者兼容了 ie678

e.currentTarget 对应 this，但是前者不兼容ie678

阻止默认行为

有些标签默认就带有一些行为，比如 a 链接跳转等，我们可以使用如下方法阻止其默认行为。

阻止行为的方法有两种：

- e.preventDefault()：标准的阻止默认事件方法，推荐使用，但是不兼容ie678
- e.returnValue：这是一个属性，兼容ie678。
- return false：这种方法也可以阻止默认事件，而且没有兼容性问题。需要注意的是，用这种方法后，return之后的代码将不会执行。

```
a.onclick = function(e) {
    e.preventDefault();
    e.returnValue;
    return false;
}
```

阻止冒泡（重点）

我们已经知道 dom 事件流包括：捕获，目标，冒泡三个阶段。

事件开始冒泡时，事件将会由具体的目标元素向上逐级传递到dom最顶层节点。

阻止冒泡就是阻止事件向上传播。

阻止冒泡的方法有两种：

- e.stopPropagation()：标准阻止冒泡的方法，推荐使用，但是不兼容ie678
- e.cancelBubbel：这是一个属性，用于阻止冒泡，兼容ie678

```
div.onclick = function(e) {
    e.stopPropagation();
    e.cancelBubbel = True;
}
```

事件委托

事件委托也称为事件代理，在jQuery里面也叫事件委派。

其思想就是，当一个父节点中的所有子节点都需要点击事件时，我们不是给每个子节点单独设置事件监听器，而是将事件监听器设置到父亲节点上，然后利用冒泡原理来影响每个子节点，这样点击子节点就可以触发父节点的监听器。

这样我们只需要操作一次DOM，提高了程序的性能。

```
<body>
    <ul>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
    </ul>
    <script>
        var ul = document.querySelector('ul');
        ul.addEventListener('click', function(e) {
            e.target.style.backgroundColor = 'pink';
        })
    </script>
</body>
```

常用的鼠标事件

禁止鼠标右键菜单（contextmenu）

contextmenu 主要控制应该何时显示上下文菜单，主要用于程序员取消默认的上下文菜单。

```
document.addEventListener('contextmenu', function(e) {
    e.preventDefault();
})
```

禁止鼠标选中（selectstart 开始选中）

```
document.addEventListener('selectstart', function(e) {
    e.preventDefault();
})
```

鼠标事件对象

event 对象代表了事件的状态，跟事件相关的一些列信息的集合。其中就包裹键盘事件信息和鼠标事件信息。鼠标事件对象是 MouseEvent；键盘事件对象是 KeyboardEvent。

鼠标事件对象	描述
e.clientX	返回鼠标相对于浏览器窗口可视区的 X 坐标
e.clientY	返回鼠标相对于浏览器窗口可视区的 Y 坐标
e.pageX	返回鼠标相对于文档页面的 X 坐标 IE9+支持
e.pageY	返回鼠标相对于文档页面的 Y 坐标 IE9+支持
e.screenX	返回鼠标相对于电脑屏幕的 X 坐标
e.screenY	返回鼠标相对于电脑屏幕·的 Y 坐标

常用的键盘事件

键盘事件	描述
onkeyup	某个键盘按键被松开时触发
onkeydown	某个键盘按键被按下时触发
onkeypress	某个键盘按键被按下时触发 但是这个方法不识别功能键，比如 ctrl shift等

当三个事件同时存在时，执行顺序永远为：keydown --> keypress --> keyup

键盘事件对象

事件对象	描述
key	记录用户按下的键盘按键，除了ie678，还有些其他浏览器不兼容
keyCode	记录用户按下键的 ASCLL 值

操作元素

JavaScript 的 DOM 操作可以改变网页内容、结构和样式，我们可以利用 DOM 操作元素来改变元素里面的内容、属性等。

改变元素内容

innerText

- innerText 只获取元素中的文本内容
- innerText 会删除空格和换行，只保留文本

语法：

```
var div = document.querySelector('div');
div.innerText = 'brokyz'
```

innerHTML

- innerHTML 会保留空格和换行
- innerHTML 修改时支持 HTML 标签，在内容中添加标签将会被识别
- 推荐使用 innerHTML

```
var div = document.querySelector('div');
div.innerHTML = '<strong>brokyz</strong>'
```

修改或添加属性内容

- 当获取到元素时，我们对元素中的属性进行添加和更改。
- **本方法对 html 中已经有定义的属性进行操作。**
- 修改属性时会直接覆盖掉原来的属性值。

语法：

```
// 获取a标签
var a = document.querySelector('a');
// 更改或添加a的属性
a.href = 'https://bilibili.com';
a.id = 'bilibili';
// 修改类名属性值
a.className = 'iama';
```

表单元素的属性操作

利用 DOM 可以操作如下表单元素属性：

```
type、value、checked、selected、disabled
```

修改或添加CSS样式属性

- 我们可以通过 JS 修改元素的大小、颜色、位置等 CSS 样式。
- JS 里面的样式采用驼峰命名法，比如 fontSize、backgroundColor
- JS 修改 style 样式操作，产生的是行内样式，修改后的 CSS 权重更高

语法：

```
var div = document.querySelector('div');
this.style.backgroundColor = 'antiquewhite';
this.style.width = '400px';
```

使用 className 修改元素类名的方式修改CSS样式属性

- 使用 className 修改样式属性会直接覆盖掉原来的 class 值

```
<!DOCTYPE html>
<head>
    <style>
        .text {
            width: 400px;
            height: 400px;
            background-color: antiquewhite;
        }

        .change {
            background-color: aliceblue;
        }
    </style>
</head>

<body>
    <div class="text">文本</div>

    <script>
        var div = document.querySelector('div');
        div.onclick = function(){
            // 此方法会直接覆盖掉原来class属性中的text
            this.className = 'change';
```

```
            // 如果不覆盖原来的class属性，只是追加的话可以这样使用
            this.className += ' change';
        }
    </script>
</body>
</html>
```

对自定义属性值的操作

- 在标签中，有些属性是内置的，比如 a标签的`href`、`target`等，这些都可以直接获取元素a，然后使用`a.href`对属性进行操作。
- 在标签中，也有些属性是我们自定义的，比如我们自己给a标签定义一个`index = '1'`，这时我们就不能使用`a.index`对自定义属性进行操作。
- 以下我们将讲解对自定义属性的操作方法：

获取自定义属性值 getAttribute()

- `getAttribute()`既可以获取自定义的属性值，也可以获取自带的属性值。

语法：

```
var div = document.querySelector('div');
// 获取内置属性值
console.log(div.getAttribute('id'));
// 获取自定义属性值
console.log(div.getAttribute('index'));
```

添加和修改自定义属性值

- `div.setAttribute('自定义属性','要修改的值');`既可以修改自定义的属性值，也可以修改自带的属性值。
- 但是用`div.setAttribute('自定义属性','要修改的值')`修改元素的类名时，只需要指定`class`即可，不需要自带方法那样使用`className`

语法：

```
var div = document.querySelector('div');
div.setAttribute('index','2');
div.setAttribute('class','box');
```

- 当使用`setAttribute`设置 class 时就是使用 class 而不是使用 className

移除自定义属性值

- `div.removeAttribute()`既可以删除自定义的属性值，也可以删除自带的属性值。

语法：

```
div.removeAttribute('index');
```

H5自定义属性新规范

由于自定义属性，无法使用自带的方法`div.index`调用，所以html5新增了对自定义属性的调用支持

- h5标准要求在自定义属性前加`data-`,当调用的时候使用`dataset`调用
- 当自定义属性中有多个 - 连接时，调用时需要使用驼峰命名法调用
- 此方法存在兼容性问题，使用 dataset 获取时，仅对ie11+支持

语法：

```
<body>
    <div data-listname="123" data-list-name="456"></div>
    <script>
        var div = document.querySelector('div');
        console.log(div.dataset.listname);
        console.log(div.dataset.listName);
    </script>
</body>
```

节点操作

节点操作可以**利用层级关系获取元素**，为我们开发提供了一种更加方便的获取元素的方法。

节点概述

网页中的所有内容都是节点（标签、属性、文本、注释等），在 DOM 中，节点使用 node 来表示。

HTML DOM树中的所有节点均可以通过 JavaScript 进行访问，所有 HTML 元素（节点）均可被修改，也可以创建或删除。

一般地，节点至少拥有node Type（节点类型）、nodeName（节点名称）和 nodeValue（节点值）这三个基本属性。

- 元素节点 nodeType 为 1
- 属性节点 nodeType 为 2
- 文本节点 nodeType 为 3 （文本节点包含文字、空格、换行等）

实际开发中，节点操作主要操作的是元素节点

节点的使用

代码	使用频率	描述
ul.parentNode	常用	获取父亲节点，如果找不到返回空
ul.childNodes	不常用	得到所有子节点，包含 元素节点 文本节点等等
ul.children	常用	获取所有的子**元素**节点 实际开发常用
ul.firstChild	不常用	获取第一个子节点，不管是文本节点和元素节点都可以被拿到
ul.lastChild	不常用	获取最后一个子节点，不管是文本节点和元素节点都可以被拿到
ul.firstElementChild	常用	获取第一个子元素节点，仅支持ie9+

代码	使用频率	描述
ul.firstElementChild	常用	获取第一个元素节点，仅支持ie9+
ul.lastElementChild	常用	获取最后一个元素节点，仅支持ie9+
li.nextSibling	不常用	返回下一个兄弟节点 包含 元素节点 文本节点等等
ul.nextElementSibling	常用	返回下一个元素兄弟节点，仅支持ie9+
ul.previousSibling	不常用	返回上一个兄弟节点 包含 元素节点 文本节点等等
ul.previousElementSibling	常用	返回上一个元素兄弟节点，仅支持ie9+

```
// 1.父节点 parentNode
var erweima = document.querySelector('.erweima');
console.log(erweima.parentNode); // 如果找不到父节点就返回为空

// 2.子节点 childNodes （集合） 得到所有子节点，包含 元素节点 文本节点等等
var ul = document.querySelector('ul');
console.log(ul.childNodes);
// 注意：返回值里面包含了所有的子节点
// 如果只想获得里面的元素节点，则需要专门处理。所以我们一般不提倡使用childNodes
for (let i = 0; i < ul.childNodes.length; i++) {
    const element = ul.childNodes[i];
    if (element.nodeType == 1){
        console.log(element);
    }
}

// children 获取所有的子元素节点 实际开发常用
console.log(ul.children);

// firstchild 获取第一个子节点，不管是文本节点和元素节点都可以被拿到
console.log(ul.firstChild);
// lastchild 获取最后一个子节点，不管是文本节点和元素节点都可以被拿到
console.log(ul.lastChild);

// firstElementChild 获取第一个元素节点
console.log(ul.firstElementChild);
// lastElementChild 获取最后一个元素节点
console.log(ul.lastElementChild);
// 注意：这几个返回第一个和最后一个存在兼容性问题，不支持ie9
// 实际开发中，兼顾兼容性
console.log(ul.children[0]);

// 3.兄弟节点
// nextSibling 返回下一个兄弟节点 包含 元素节点 文本节点等等
console.log(ul.nextSibling);
// nextElement 返回下一个元素兄弟节点
console.log(ul.nextElementSibling);
// prevousSibling 返回上一个兄弟节点 包含 元素节点 文本节点等等
console.log(ul.previousSibling);
// prevousElementSibling 返回上一个兄弟节点 包含 元素节点 文本节点等等
console.log(ul.previousElementSibling);
```

创建节点并添加节点

代码	描述
ul.appendChild()	在孩子的最后面添加元素
ul.insertBefore(要添加的元素,指定的孩子元素)	在指定孩子元素的前面添加要添加的元素

语法：

```
var li = document.createElement('li');
var ul = document.querySelector('ul');
// 在ul的孩子后面追加元素
ul.appendChild(li);
// 在ul的指定孩子前面插入元素
ul.insertBefore(li,ul.children[0])
```

删除节点

```
var ul = document.querySelector('ul');
// 删除父节点里面的孩子
ul.removeChild(ul.children[0]); // 删除ul中的第一个孩子
```

复制节点

```
var ul = document.querySelector('ul');
//复制节点
var cloned = ul.children[0].cloneNode(true); //括号为空或者false,是浅拷贝， 则只复制标签不复制内容，如果为true，复制内容，深拷贝
ul.appendChild(cloned);
```

三种创建元素方式的区别

- document.write()
  - `document.write('<div>123<div>')`==文档执行完毕，他会导致页面全部重绘==
- innerHTML = ‘’
  - `div.innerHTML += '<div>123<div>'`使用拼接的方式创建多个元素时，耗时非常大
  - 当将要创建的元素插入数组中，再通过innerHTML插入时，效率大大提成，为最优
- document.createElement()
  - 使用这种方法创建，并使用节点操作添加到页面中时，方法效率高，但是没有innerHTML结合数组的形式效率高

实验： 测试地址

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>三种创建元素的区别</title>
</head>

<body>
    <button>文档流执行完毕，测试document.write</button>
    <button>测试innerHTML执行效率</button>
    <button>测试innerHTML结合数组执行效率</button>
    <button>测试createElement执行效率</button>
    <div class="test"></div>
    <script>
        // document.write 当所有文档流执行完毕时，执行这段代码会导致页面重绘
        var btn = document.querySelectorAll('button');
        btn[0].onclick = function () {
            document.write('<div>当所有文档流执行完毕时，执行这段代码会导致页面重绘
document.write</div>');
        }


        // 测试innerHTML执行效率
        var test = document.querySelector('.test');
        btn[1].onclick = function () {
            if (test.children[0]) {
                test.removeChild(test.children[0]);
            }
            var div = document.createElement('div');
            var d1 = +new Date();
            for (var i = 0; i < 2000; i++) {
                div.innerHTML += "<div style='width:100px;border:1px solid
blue;'></div>"
            }
            var d2 = +new Date();
            console.log(d2 - d1);
            test.appendChild(div);

        }

        // 测试innerHTML加数组效率
        btn[2].onclick = function () {
            if (test.children[0]) {
                test.removeChild(test.children[0]);
            }

            var div = document.createElement('div');
            var d1 = +new Date();
            var arr = [];
            for (var i = 0; i < 2000; i++) {
                arr.push("<div style='width:100px;border:1px solid blue;'>
</div>");
            }
            div.innerHTML = arr.join('');
            test.appendChild(div);
            var d2 = +new Date();
            console.log(d2 - d1);
```

```
        }

        // createElement效率测试
        btn[3].onclick = function () {
            if (test.children[0]) {
                test.removeChild(test.children[0]);
            }
            var div = document.createElement('div');
            var d1 = +new Date();
            // 注意创建要添加的元素一定要写在循环内，因为如果写在外面，只创建一次，append添加之后就没有了。
            for (var i = 0; i < 2000; i++) {
                var divs = document.createElement('div');
                divs.style.border = '1px solid blue';
                divs.style.width = '100px';
                div.appendChild(divs);
            }
            var d2 = +new Date();
            test.appendChild(div)
            console.log(d2 - d1);
        }
    </script>
</body>
</html>
```

DOM 重点核心（复习）

DOM 就是文档对象模型（Document Object Model），是w3c组织推荐的处理可拓展标记语言（HTML或者XML）的标准编程接口。

W3C已经定义了一系列的 DOM 接口，通过这些接口 DOM 接口可以改变网页的内容、结构和样式。

1. 对于 JavaScript，为了能够使 JavaScript 操作 HTML，JavaScript就有了一套自己的 DOM 编程接口。
2. 对于 HTML，DOM 使得 HTML 形成了一棵 DOM 树。其中包含了文档（整个页面）、元素（页面中所有的标签）、节点（页面中所有的内容，文档是节点，元素是节点，属性也是节点）。
3. 我们获取来的 DOM 元素是一个对象（object），所以称为文档对象模型。

关于 DOM 操作，我们主要针对于元素的操作。主要有创建、增、删、改、查、属性操作、事件操作。

创建

1. document.write
2. innerHTML
3. createElement

增

1. appendChild
2. insertBefore

删

1. removeChild

改

主要是修改 DOM 元素的属性、内容、表单值等

1. 修改元素属性：src、href、title等
2. 修改普通元素内容：innerHTML、innerText
3. 修改表单元素：value、type、disable等
4. 修改元素样式：style、className

查

主要获取查询 DOM 元素

1. DOM 提供的 API 方法：getElementById、getElementByTagName 古老用法，不推荐
2. H5 提供的新方法：querySelector、querySelectorAll 推荐
3. 利用节点操作获取元素：父（parentNode）、子（children）、兄（previousElementSibling、nextElementSibling）推荐

属性操作

主要针对于自定义属性

1. setAttribute：设置dom的属性值
2. getAttribute：得到dom的属性值
3. removeAttribute：移除属性

事件操作

给元素注册事件、采取 `事件源.事件类型 = 事件处理程序`

1. onclick：鼠标点击左键触发
2. onmouseover：鼠标经过触发
3. onmouseout：鼠标离开触发
4. onfocus：获得鼠标焦点触发
5. onblur：失去鼠标焦点触发
6. onmousemove：鼠标移动触发
7. onmouseup：鼠标弹起触发
8. onmousedown：鼠标按下触发

BOM

什么是 BOM

BOM （Browser Object Model）即**浏览器对象模型**，它提供了独立于内容而与**浏览器窗口进行交互的对象**，其核心对象是 window。

BOM 由一系列相关的对象构成，并且每个对象都提供了很多方法与属性。

BOM 缺乏标准，JavaScript 语法的标准化组织是 ECMA，DOM 的标准化组织是 W3C，BOM 最初是 Netspace 浏览器标准的一部分。

DOM	BOM
文档对象模型	浏览器对象模型
DOM 就是把 [文档] 当作一个 [对象] 来看待	把 [浏览器] 当做一个 [对象] 来看待
DOM 的顶级对象是 document	BOM 的顶级对象是 window
DOM 主要学习的是操作页面元素	BOM 学习的是浏览器窗口交互的一些对象
DOM 是 W3C 标准规范	DOM 是浏览器厂商在各自浏览器上定义的，兼容性比较差

BOM 的构成

window 对象是浏览器的顶级对象，它具有双重角色。

1. 它是 JS 访问浏览器的一个接口。
2. 它是一个全局对象。定义在全局作用域中的变量、函数都会变成 window 对象的属性和方法。

在调用的时候可以省略 window，前面学习的对话框都属于 window 对象方法，如 alert()、prompt() 等。

注意：window下的一个特殊属性 window.name

页面加载事件

```
window.onload = function() {}
或者
window.addEventListener("load",function(){})
```

window.onload 是窗口（页面）加载事件，当文档内容完全加载完成会触发该事件（包括图像、脚本文件、CSS文件等）才调用处理函数。

注意：

1. 有了 window.onload 就可以把 JS 代码写到页面元素的上方，因为 onload 是等页面内容全部加载完毕，再去执行处理函数。
2. window.onload 传统注册事件方式只能写一次，如果有多个，会以最后一个 window.onload 为准。
3. 如果使用 addEventListener 则没有限制，写多少个都行。

```
document.addEventListener("DOMcontentLoaded",function(){})
```

DOMContentLoaded 事件触发时，仅当 DOM 加载完成，不包括样式表，图片，flash等等。

ie9以上才支持。

如果页面有很多图图片，从用户访问到 onload 触发可能需要较长的时间，交互效果就不能实现，必然影响用户体验，此时用DOMContentLoaded 事件比较合适。

调整窗口大小事件

```
window.onresize = function(){}
window.addEventListener("resize",function(){})
```

window.onresize 是调整窗口大小加载事件，当触发时就调用的处理函数。

注意：

1. 只要窗口大小发生像素变化，就会触发这个事件。
2. 我们经常利用这个事件完成响应式布局。window.innerWidth 当前屏幕的宽度。

```
// 浏览器窗口的当前宽度
window.innerWidth
```

定时器

window 对象给我们提供了 2 种非常好的方法 - **定时器。**

setTimeout()

```
window.setTimeout(调用函数, [延迟的毫秒数])
// 在调用的时候 window 可以省略
setTimeout(function() {
    console.log('三秒过去了');
}, 3000)
// 延迟时间单位是毫秒，如果不写默认为0
function callback(){
    console.log('三秒过去了');
}
setTimeout(callback, 3000);
setTimeout('callback()', 3000); // 不提倡这种写法
// 页面中可能有多个定时器，我们经常给定时器加标识符（名字）
var timer1 = setTimeout(callback, 3000);
var timer2 = setTimeout(callback, 5000);
```

setTimeout() 方法用于设置一个定时器，该定时器在定时器到期后执行调用函数。

注意：

1. window 可以省略。
2. 这个调用函数可以**直接写函数，或者写函数名**或者采取字符串 `'函数名()'` 三种形式。
3. 延迟的毫秒数省略默认是0，如果写，必须是毫秒。
4. 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。

setTimeout() 这个函数我们也叫做**回调函数 callback**

普通函数是按照代码顺序直接调用。

而这个函数，**需要等待**时间，时间到了才去调用这个函数，因此称为回调函数。

简单理解，回调，就是回头调用。上一件事情干完，再回头去调用这个函数。

之前我们学的 element.onclick = function() {} 或者 element.addEventListener("click",fn) 里面的函数也是回调函数。

停止 setTimeout() 定时器

```
window.clearTimeout(timeoutID)
// 可以省略window
clearTimeout()
```

setInterval() 定时器

```
window.setInterval(回调函数，间隔毫秒数)
```

setInterval() 方法重复调用一个函数，每隔这个时间，就去调用一次回调函数。

注意：

1. window 可以省略。
2. 这个调用函数可以**直接写函数，或者写函数名**或者采取字符串 `'函数名()'` 三种形式。
3. 延迟的毫秒数省略默认是0，如果写，必须是毫秒。
4. 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。

停止 clearInterval() 定时器

```
window.clearInterval(timeoutID)
// 可以省略window
clearInterval()
```

this

在我们使用停止定时器时，不能在定时器中使用 this 停止定时器。

因为定时器的时 window 对象的函数，this 指向当前对象，所以指向的是 windows。

所以 this 指向的是调用函数的**对象。**

JS 执行机制

JS 老版本是单线程的

JavaScript 语言的一大特点就是**单线程**，也就是说，**同一个时间只能做一件事。**这是因为 JavaScript 这门脚本语言的诞生的使命所致——JavaScript 是为处理页面中用户的交互，以及操作 DOM 而诞生的。比如我们对某个 DOM 元素进行添加和删除操作，不能同时进行。应该先进行添加，之后再删除。

比如以下代码：

```
console.log(1);
setTimeout(function(){
    console.log(3);
},1000);
console.log(2);
```

这段代码需要等定时器执行完，才会向下执行代码，如果定时器设置时间较长，那么就会等待很久。

现在新版 Js 支持同步和异步

为了解决这个问题，利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许JavaScript 脚本创建多个线程。于是，JS中出现了**同步**和**异步**。

同步

前一个任务结束后再执行后一个任务，程序的执行顺序与任务的排列顺序是一致的、同步的。比如做饭的同步做法：我们要烧水煮饭，等水开了（10分钟之后），再去切菜，炒菜。

异步

在做一件事情时，因为这件事情需要花费很长时间，在做这件事情的同时，你还可以去处理其他的事情。比如做饭的异步做法，在我们烧水的同时，利用这10分钟，去切菜，炒菜。

同步和异步的本质区别：这条流水线上各个流程的执行顺序不同。

JS 执行机制的本质

有如下代码：

```
console.log(1);
setTimeout(function(){
    console.log(3);
},0);
console.log(2);
```

以上代码的输出结果为 1，2，3。

在 JS 中有两种任务：

1. 同步任务：所有的同步任务都在主线程上执行，形成一个**执行栈。**
2. 异步任务：异步任务是通过**回调函数**来实现的，在执行栈执行同步任务时，其中的回调函数会被放到任务队列（消息队列）里面等待主线程执行结束再执行。

   一般来说异步任务有如下三种类型：

   - 普通事件，如 click、resize等
   - 资源加载，如 load、error等
   - 定时器，包括 setInterval、setTimeout等

主线程执行栈	**任务队列（消息队列）**
console.log(1);	function(){ console.log(3); }
setTimeout(fn, 0);	
console.log(2);	

当执行栈执行到`setTimeout(fn, 0);`时，定时器本身是**同步任务**，但是定时器里面的**回调函数**是异步任务，被放到任务队列中，等待执行栈执行完毕，再去执行任务队列里面的任务。

在执行栈和任务队列之间，又有一个**异步进程处理**的步骤，这一步骤将决定，回调函数在何时被加入任务队列。

以如下代码举例：

```
console.log(1);
document.onclick = function() {
    console.log('click');
}
console.log(2);
setTimeout(function() {
    console.log(3);
},3000)
```

1. 首先执行栈会按照顺序执行同步任务。
2. 当执行到包含异步任务的同步任务时，这个同步任务就会被提交到异步进程中进行处理。
3. 在异步进程中，其中待处理的任务只有满足了条件时，才会被放到任务队列中。比如点击事件，只有被点击时，其中的异步进程才会被送到任务队列。
4. 当执行栈中的同步任务运行完毕，执行栈会将任务队列中的异步任务按顺序拿到执行栈进行执行。
5. 然后运行完成后，执行栈并没有停止，它还会不断地去获得任务，执行任务，这种机制叫做循环机制。比如这时我们再次点击页面，异步进程会再次将异步任务放到任务队列，由于执行栈会重复获取任务，执行栈又会将任务队列中的任务拿到执行栈进行执行。

重点：由于主线程不断的重复获取任务、执行任务、再获取任务、再执行，所以这种机制被称为**事件循环**

location 对象

window 对象给我们提供了一个 location 属性，用于**获取设置窗体的 url**，并且可以用于解析 url。因为这个属性返回的是一个对象，所以我们将这个属性也成为 location 对象。

URL

统一资源定位符（Uniform Resource Locator，URL）是互联网上标准资源的地址。互联网上每个文件都有一个唯一的 URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

URL 的一般格式为：

```
protocol://host[:port]/path/[?query]#fragment
http://www.brokyz.tk/index.html?name=andy&age=18#link
```

组成	说明
protocol	通信协议 常用的 http、ftp、maito等
host	主机（域名）
port	端口号 可选，省略时使用方案默认端口，如http默认为80
path	路径 有 零或多个/符号隔开的字符串，一般用来表示主机上的一个目录或文件地址
query	参数 以键值对的形式，通过&符号分隔
fragment	片段 #后面内容 常见于链接 锚点

location 对象的属性

属性	返回值
location.href	获取或者设置 整个 URL，如果不打印设置url会实现页面跳转
location.host	返回主机（域名）
location.port	返回端口号 如果未写则返回空字符串
location.pathname	返回路径
location.search	返回参数
location.hash	返回片段 #后面内容 常用于链接 锚点

```
// 5秒后页面跳转
var t = 5;
setTimeout(function() {
    if (t == 0){
        location.href = 'https://baidu.com';
    }else{
        t--;
    }
})
```

获取 URL 参数

```
<body>
    <form action="获取URL参数.html">
        <input type="text" name="uname">
        <input type="submit" value="提交">
    </form>
    <div></div>
    <script>
        console.log(location.search)
        var params = location.search.substring(1);
        console.log(params);
        var arr = params.split('=');
        console.log(arr);
        var div = document.querySelector('div');
        if(arr[1]){
            div.innerHTML = '这里是' + arr[1];
        }
    </script>
</body>
```

- substring(1)：输入第1位之后的字符
- split('=')：通过=拆分字符，拆分返回拆分后的数组

location 常用方法

location方法	返回值
location.assign(url)	跟 href 一样，可以跳转页面（也称为重定向页面）
location.replace(url)	跳转替换当前页面，因为不记录历史，所以不能后退页面
location.reload()	重新加载页面，相当于刷新按钮或者 f5 如果参数为true 强制刷新 ctrl+f5（无视浏览器缓存）

navigator 对象

navgator 包含了有关浏览器的相关信息，他有很多属性，我们最常用的时 userAgent，该属性可以返回由客户机发送服务器的 user-agent 头部的值。

下面前端代码可以判断用户哪个终端打开页面，实现跳转。

```
//
if
((navigator.userAgent.match(/(phone|pad|pod|iPhone|iPod|ios|iPad|Android|Mobile|
BlackBerry|IEMobile|MQQBrowser|JUC|Fennec|wOSBrowser|BrowserNG|WebOS|Symbian|Win
dows Phone)/i))) {
      window.location.href = "../H5/index.html"; //手机
}
```

history 对象

window 对象给我们提供了一个 hostory 对象，与浏览器历史记录进行交互。该对象包含用户（在浏览器窗口中）访问过的 URL。

history对象方法	作用
history.back()	可以后退功能
history.forward()	前进功能
history.go(参数)	前进后退功能 参数如果是1 前进1 个页面，2是前进2，如果时-1 后退1 个页面

PC网页特效

元素偏移量 offset 系列

offset 翻译过来就是偏移量，我们使用 offset 系列相关属性可以**动态的**得到该原告诉的位置（偏移）、大小等。

- 获得元素距离带有定位父元素的位置
- 获得元素自身的大小（宽度高度）
- 注意：返回的数值都不带单位

offset 系列常用属性：

offset系列属性	作用
element.offsetParent	返回作为该元素带有定位的父级元素 如果父级没有定位则返回body
element.offsetTop	返回元素相对带有定位父元素上方的偏移
emement.offsetLeft	返回元素相对带有定位父元素左边框的偏移
element.offsetWidth	返回自身包括padding、边框、内容区的宽度，返回数值不带单位
element.offsetHeight	返回自身包括padding、边框、内容区的高度、返回数值不带单位

offset 和 style 的区别

offset	style
offset 可以得到任意样式表中的样式值	style 只能获得行内样式表中的样式值
offset 系列获得的数值是没有单位的	style.width 获得的是带有单位的字符串
offsetWidth 包含 padding、border、width	style.width 获得不包含 padding、border 的值
offsetWidth 等属性是只读属性，只能获取不能赋值	style.width 是可读写属性，可以获取也可以赋值
所以，我们想要获取元素大小位置，用offset更合适	**所以，我们想要给元素赋值，用style更合适**

获取鼠标在盒子内的距离

我们可以通过事件对象的属性 `e.pageX`、`e.pageY` 来获取我们鼠标在浏览器中的坐标。

然后，我们通过得到的鼠标距离浏览器的距离，减去，盒子距离浏览器的距离，就是我们鼠标在盒子内的距离。

元素可视区 client 系列

client 翻译是客户端的意思，我们可以使用 client 系列的相关属性来获取元素可视区的相关信息。通过 client 系列的相关属性可以动态的得到该元素的边框大小、元素大小等。

client 系列属性	作用
element.clientTop	返回元素上边框的大小
element.clientLeft	返回元素左边框的大小
element.Width	返回自身包括padding、内容区的宽度，不含边框，返回数值不带单位
element.clientHeight	返回自身包括padding、内容区的高度，不含边框，返回数值不带单位

立即执行函数

立即执行函数不需要调用，立马能够自己执行。

主要作用：创建一个独立的作用域，里面所有的变量都是局部变量，避免了命名冲突的问题。

```
(function() {})()
// 或者
(function(){}())
```

元素滚动 scroll 系列

使用 scroll 系列的相关属性可以动态的得到该元素的大小、滚动距离等。

scroll 属性	作用
element.scrollTop	返回被卷上去的上侧距离，返回数值不带单位
element.scrollLeft	返回被卷上去的左侧距离，返回数值不带单位
element.scrollWidth	返回自身实际宽度，不含边框，但是包含padding，返回数值不带单位
element.scrollHeight	返回自身实际高度，不含边框，但是包含padding，返回数值不带单位
window.pageYOffset	获得页面被卷去的头部距离，不支持ie9以下，修改此值不会影响页面
window.pageXOffset	获得页面被卷去的左侧距离，不支持ie9以下，修改此值不会影响页面

当盒子内内容超出后，scroll返回的是超出之后的大小，而之前的client返回的则是固定指定的大小。

页面被卷去头部兼容性解决方案

被卷去的头部，存在兼容性问题：

1. 声明了 DTD，使用 `document.documentElement.scrollTop`
2. 未声明 DTD，使用`document.body.scrollTop`
3. 新方法 `window.pageYOffset`和`window.pageXOffset`，IE9 才开始支持，修改这个值，并不会影响页面位置。

兼容性函数：

```
function getScroll() {
    return {
        left: window.pageXOffset || document.documentElement.scrollLeft ||
document.body.scrollLeft || 0,
        top: window.pageYOffset || document.documentElement.scrollTop ||
document.body.scrollTop || 0,
    };
}
getScroll().left;
getScroll().top;
```

三大系列总结

三大系列对比	作用
element.offsetWidth	返回自身包括padding、边框、内容区的宽度，返回值不带px
element.clientWidth	返回自身包括padding、内容区域的宽度，不含边框，返回值不带px

三大系列对比	作用
element.scrollWidth	返回自身的实际宽度（比如字太多被滚动撑开），不含边框，返回数值不带单位

主要用法：

1. offset系列经常用于获得元素位置：offsetTop、offsetLeft
2. client系列经常用于获得元素大小：clientWidth、clientHeight
3. scroll系列经常用于获取滚动距离：scrollTop、scrollLeft

mouseover 和 mouseenter 区别

mouseenter 鼠标事件

- 当鼠标移动到元素上时就会触发 mouseenter 事件
- 类似 mouseover，他们两者之间的差别是
- mouseover 鼠标经过自身盒子会触发，经过子盒子还会触发。mouseenter 只会经过自身盒子触发
- 出现这个现象的原因是因为，mouseover会冒泡，mouseenter不会冒泡

动画函数封装

核心原理：通过定时器 setInterval() 不断移动盒子的位置

实现步骤

1. 获得盒子当前的位置
2. 让盒子在当前位置加上1个距离
3. 利用定时器不断重复这个操作
4. 添加结束定时器的条件
5. 注意此元素需要添加定位，才能使用`element.style.left`

```
        // 1. 获得盒子当前位置
        var div = document.querySelector('div');
        // 2. 让盒子在当前位置上加上1个距离
        div.style.left = div.offsetLeft + 1 + 'px';
        // 3. 用定时器不断的重复这个操作
        var timer = setInterval(function () {
            if (div.offsetLeft >= 400) {
                // 4. 停止动画的本质是停止定时器
                clearInterval(timer);
            }
            div.style.left = div.offsetLeft + 1 + 'px';
        }, 1)
```

动画函数的封装

```
        function animate(obj, target) {
            var timer = setInterval(function () {
                if (obj.offsetLeft >= target) {
                    clearInterval(timer);
                }
                obj.style.left = obj.offsetLeft + 1 + 'px';
            }, 1)
        }

        animate(div,300);
```

动画封装函数优化

问题：

1. 每一个对象调用函数都会声明`var timer`开辟一个内存空间，这样调用对象一多影响性能
2. 每一个对象调用函数时，函数里面的定时器都叫`timer`，可能会引起歧义
3. 当同一个对象多次调用定时器时，动画会加快，因为定时器同时运行出现叠加

解决办法：

- 将定时器看成当前对象的属性`obj.timer`，这样就避免声明新对象，也避免了定时器名字相同引起的歧义。
- 在调用函数的时候，首先清除一下定时器，这时候在运行代码，就保证了一直是一个定时器运行。

```
        function animate(obj, target) {
            clearInterval(obj.timer);
            obj.timer = setInterval(function () {
                if (obj.offsetLeft >= target) {
                    clearInterval(obj.timer);
                }
                obj.style.left = obj.offsetLeft + 1 + 'px';
            }, 1)
        }
        animate(div,300);
```

缓动动画

缓动动画就是让元素运动速度有所变化，最常见的就是让速度慢慢停下来。

原理：

1. 让盒子每次移动的距离慢慢变小，速度就会慢慢落下来。
2. 核心算法：`（目标值 - 现在的位置）/ 10`作为每次的移动距离。
3. 停止的条件是：让当前的盒子位置等于目标的位置就停止计时器。

```
function animateAvg(obj, target, callback) {
            clearInterval(obj.timer);
            obj.timer = setInterval(function () {
                // 把步长值改为整数，来防止小数计算，使得目标到不了指定位置而无法停止
                // 当step是正数向上取整（取大），当是负数向下取整（取小）
                var step = (target - obj.offsetLeft) / 10;
                step = step > 0 ? Math.ceil(step) : Math.floor(step);
                if (obj.offsetLeft == target) {
```

```
                    clearInterval(obj.timer);
                    // 回调函数写到定时器结束的时候调用
                    if (callback) {
                        callback();
                    }
                }
                obj.style.left = obj.offsetLeft + step + 'px';
                // console.log('stop');
            }, 20)
        }
```

轮播图

HTML部分

- `slideshow`：整个轮播图的框架
- `prev`：上一页按钮
- `next`：下一页按钮
- `promo`：轮播图促销图片
- `circle`：促销图片对应的按钮
- `promo_now`：当前促销图片对应的按钮

```
<body>
    <div class="slideshow">
        <a src="" alt="" class="prev iconfont"><</a>
        <a src="" alt="" class="next iconfont">></a>
        <ul class="promo">
            <li>
                <img src="../../res/taobao-promo.png" alt="" class="promo_img"
/>
            </li>
            <li>
                <img src="../../res/taobao-promo1.png" alt="" class="promo_img"
/>
            </li>
            <li>
                <img src="../../res/taobao-promo2.png" alt="" class="promo_img"
/>
            </li>
            <li>
                <img src="../../res/taobao-promo3.png" alt="" class="promo_img"
/>
            </li>
        </ul>
        <ol class="circle">
        </ol>
    </div>
</body>
```

CSS部分

- 注意: 后续需要添加动画的元素, 需要加上绝对定位.

```
<style>
    * {
        margin: 0;
        padding: 0;
    }

    li {
        list-style: none;
    }

    a {
        text-decoration: none;
    }

    .slideshow {
        position: relative;
        width: 600px;
        height: 300px;
        margin: 200px auto;
        background-color: aliceblue;
        border-radius: 10px;
        overflow: hidden;
    }

    .prev,
    .next {
        display: none;
        position: absolute;
        top: 50%;
        width: 20px;
        height: 30px;
        margin-top: -15px;
        display: inline-block;
        color: white;
        background-color: rgba(0, 0, 0, .3);
        text-align: center;
        line-height: 30px;
        z-index: 2;
    }

    .prev:hover,
    .next:hover {
        cursor: pointer;
    }

    .prev {
        left: 0;
        border-radius: 0 15px 15px 0;
    }

    .next {
```

```
        right: 0;
        border-radius: 15px 0 0 15px;
    }

    .promo {
        position: absolute;
    }

    .promo li {
        float: left;
    }

    .promo li .promo_img {
        width: 600px;
        height: 300px;
        cursor: pointer;
    }

    .circle {
        position: absolute;
        right: 5%;
        bottom: 5px;
        height: 15px;
        /* width: 50px; */
        /* background-color: rgba(255, 255, 255, .7); */
        border-radius: 10px;
    }

    .circle li {
        float: left;
        margin-top: 2px;
        margin-left: 5px;
        width: 10px;
        height: 10px;
        background-color: rgba(255, 255, 255, 1);
        border-radius: 10px;
    }

    .circle li:hover {
        background-color: coral;
        cursor: pointer;
    }

    .circle .promo_now {
        background-color: coral;
    }
</style>
```

JS部分

1. 获取所需要操作的元素.
    - `slideshow`: 轮播图整体框架.
    - `prev`: 轮播图上一页按钮.
    - `next`: 轮播图下一页按钮.

- `promo`: 轮播图促销海报主体.
- `circle`: 轮播图促销海报对应的按钮.
- `imgWidth`: 轮播图一张促销海报的宽度.

2. 鼠标经过轮播图显示上一页和下一页的图标, 鼠标离开轮播图隐藏上一页和下一页的图标.
   - 给轮播图整体框架添加鼠标进入和离开事件.
3. 动态设定轮播图促销海报主体的宽度.
   - 由于后面为了优化第一张和最后一张的过渡动画, 所以我们这里动态调整的时候应该比`length`多一张.
   - 促销海报的总数 * 促销海报单张的宽度 = 轮播图促销海报主体的宽度
4. 动态生成轮播图对应的显示按钮.
   - 通过循环遍历, 动态生成按钮. 这样按钮数和促销海报数就可一一对应.
   - 创建按钮的时候, 给每个按钮添加上点击事件, 让点击哪个按钮哪个按钮就以类名标记为当前按钮`promo_now`. 在添加前要使用排他思想, 清除所有按钮的`promo_now`标记.
5. 复制在轮播图中复制并尾插入轮播图的第一张图片， 为了优化第一页和最后一页之间的切换动画.
   - 克隆促销海报的第一张海报, 将其复制到所有海报的最后.
   - 这样一来, 最后一张切换到第一张时, 我们可以先让其滚动到我们克隆的最后一张图片上. 等动画结束, 我们让轮播图迅速切换到第一张图片位置. 这样人眼无法发觉, 利用了一个小障眼法, 优化了动画效果.
   - 第一张切换到最后一张同理, 先让其无动画迅速切换到最后一个克隆的轮播图位置. 之后再向前翻页, 切换到实际的最后一张海报. 优化动画效果.
6. 实现轮播图下一页的逻辑.
   - 获取当前按钮`promo_now`, 以及其编号`index`, 以此来判断当前位于第几个轮播图.
   - 计算下一张轮播图所需要的移动坐标`target`.
   - 需要获取的信息完毕后, 清除当前轮播图的标记`promo_now`.
   - 使用封装好的动画函数移动轮播图, 当轮播图移动到最后一张时候, 让其施展障眼法瞬间移动到第一张.
   - 给移动到的下一张轮播图添加当前轮播图标记`promo_now`.
7. 实现轮播图上一页的逻辑.
   - 获取当前按钮`promo_now`, 以及其编号`index`, 以此来判断当前位于第几个轮播图.
   - 点击后判断, 当为第一张轮播图时, 施展障眼法瞬间移动到最后一张轮播图.
   - 计算上一张轮播图所需要的移动坐标`target`.
   - 需要获取的信息完毕后, 清除当前轮播图的标记`promo_now`.
   - 使用封装好的动画函数移动轮播图.
   - 给移动到的上一张轮播图添加当前轮播图标记`promo_now`.
8. 播图点击按钮移动到相对应的轮播图。
   - 获取当前点击按钮的`index`.
   - 计算移动到当前点击按钮位置轮播图所需要移动到的位置`target`.
   - 通过动画函数, 移动到计算好的位置.
9. 实现轮播图定时自动切换下一页.
   - 设置定时函数, 指定时间间隔调用以此, 下一页按钮.
   - 当鼠标进入轮播图时, 定时器.
   - 当鼠标离开时重新加载定时器.

```
<script>
    // 1. 获取所需要操作的元素。
    var slideshow = document.querySelector(".slideshow");
    var prev = document.querySelector(".prev");
```

```
    var next = document.querySelector(".next");
    var ul = slideshow.querySelector(".promo");
    var ol = slideshow.querySelector(".circle");
    var imgWidth = slideshow.offsetWidth;
    // 2. 鼠标经过轮播图显示上一页下一页
    slideshow.addEventListener("mouseenter", function () {
        prev.style.display = "block";
        next.style.display = "block";
        // 9. 实现轮播图定时自动切换下一页。
        clearInterval(promo_auto);
    })
    // 2. 鼠标离开轮播图隐藏上一页下一页
    slideshow.addEventListener("mouseleave", function () {
        prev.style.display = "none";
        next.style.display = "none";
        // 9. 实现轮播图定时自动切换下一页。
        promo_auto = setInterval(function () {
            next.click();
        }, 3000);
    })
    // 3. 动态设定轮播图本质主体的宽度。
    ul.style.width = (ul.children.length + 1) * imgWidth + "px";

    // 4. 动态生成轮播图对应的显示按钮。
    for (var i = 0; i < ul.children.length; i++) {
        var li = document.createElement("li");
        li.setAttribute("index", i);
        ol.appendChild(li);
        li.addEventListener("click", function () {
            for (var i = 0; i < ol.children.length; i++) {
                ol.children[i].className = "";
            }
            this.className = "promo_now";
            // 8. 实现轮播图点击按钮移动到相对应的轮播图。
            var promo_index = this.getAttribute("index");
            target = promo_index * imgWidth;
            animate(ul, -target);
        })
    }
    // 设置第一个按钮为当前按钮
    ol.children[0].className = "promo_now";
    // 5. 复制在轮播图中复制并尾插入轮播图的第一张图片，为了优化第一页和最后一页之间的切换动画。
    var ul0 = ul.children[0].cloneNode(true);
    ul.appendChild(ul0);
    // 6. 实现轮播图下一页的逻辑。
    next.addEventListener("click", function () {
        var promo_now = document.querySelector(".promo_now");
        var promo_index = promo_now.getAttribute("index");
        target = (promo_index - 0 + 1) * imgWidth;
        promo_now.className = "";
        animate(ul, -target, function () {
            if ((promo_index - 0 + 1) == ol.children.length) {
                ul.style.left = "0px";
            }
```

```
        });
        if ((promo_index - 0 + 1) == ol.children.length) {
            ol.children[0].className = "promo_now";
        } else {
            ol.children[promo_index - 0 + 1].className = "promo_now";
        }
    })
    // 7. 实现轮播图上一页的逻辑。
    prev.addEventListener("click", function () {
        var promo_now = document.querySelector(".promo_now");
        var promo_index = promo_now.getAttribute("index");
        if (promo_index == 0) {
            ul.style.left = -(ul.children.length - 1) * imgWidth + "px";
            promo_index = (ul.children.length - 1);
        }
        target = (promo_index - 1) * imgWidth;
        promo_now.className = "";
        animate(ul, -target);
        console.log("target:" + target);
        console.log("promo_index:" + (promo_index - 1));
        ol.children[promo_index - 1].className = "promo_now";
    })

    // 9. 实现轮播图定时自动切换下一页。
    var promo_auto = setInterval(function () {
        next.click();
    }, 3000);

</script>
```

```
function animate(obj, target, callback) {
    clearInterval(obj.timer);
    obj.timer = setInterval(function () {
        // 把步长值改为整数，来防止小数计算，使得目标到不了指定位置而无法停止
        // 当step是正数向上取整（取大），当是负数向下取整（取小）
        var step = (target - obj.offsetLeft) / 10;
        step = step > 0 ? Math.ceil(step) : Math.floor(step);
        if (obj.offsetLeft == target) {
            clearInterval(obj.timer);
            // 回调函数写到定时器结束的时候调用
            // if (callback) {
            //     callback();
            // }
            // 高级写法
            callback && callback();
        }
        obj.style.left = obj.offsetLeft + step + 'px';
        // console.log('stop');
    }, 10)
}
```

JS高级

构造函数和原型

ES5创建对象的方法

- 在ES6之前，如果想创建对象有三种方式：
  - 通过字面量`{}`创建对象
  - 利用`new Object`创建对象
  - 使用构造函数创建对象

```
// 1. 通过字面量{}创建对象
var obj = {
    uname: 'brokyz',
    agr: 18,
    sex: 'male',
    sayHi: function(){
        console.log('hi');
    }
}
// 2. 利用new Object创建对象
var obj = new Object();
obj.uname = 'brokyz';
obj.age = 18;
obj.sex = 'male';
obj.sayHi = function(){
    console.log('hi');
}
// 3. ES5使用构造函数创建对象
function Apple(name){
    this.name = name;
    this.sayHi = function(){
        console.log('hi');
    }
}
var iphone = new Apple('iphone12');
console.log(iphone.name);
console.log(iphone['name']);
```

new在执行时会做四件事

1. 在内存中创建一个新的空对象。
2. 让`this`指向这个新的对象。
3. 执行构造函数里面的代码，给这个新的对象添加属性和方法。
4. 返回这个新的对象（所以构造函数里面不需要`return`。

实例成员和静态成员

1. 实例成员：在构造函数内部通过`this`来添加的成员。只能通过实例化的对象来访问。
2. 静态成员：在构造函数本身上添加的成员。只能通过构造函数来访问。

```
function Apple(name){
    // 实例成员
    this.name = name;
    this.sayHi = function(){
        console.log('hi');
    }
}
// 实例成员只能通过实例化对象来访问
console.log(new Apple('brokyz').name);
// 静态成员需要在构造函数本身上添加
Apple.age = 15;
// 只能通过构造函数来访问
console.log(Apple.age);
```

构造函数的资源浪费问题

```
function Apple(name){
    this.name = name;
    this.sayHi = function(){
        console.log('hi');
    }
}
var apple1 = new Apple('iphone12');
var apple2 = new Apple('iphone13');
console.log(apple1.sayHi === apple2.sayHi);  // false
```

- 构造函数中的方法`sayHi`属于复杂数据类型，创建实例时会在内存中单独开辟空间来存放。
- 实例化对象之间并不能公用相同的方法。所以实例化对象会分别在内存中开辟空间来存放自己的方法。
- 比如`apple1`在内存中有一个自己的`sayHi`方法。当继续实例化`apple2`时，构造函数会为其在内存开辟一个新的`sayHi`方法。

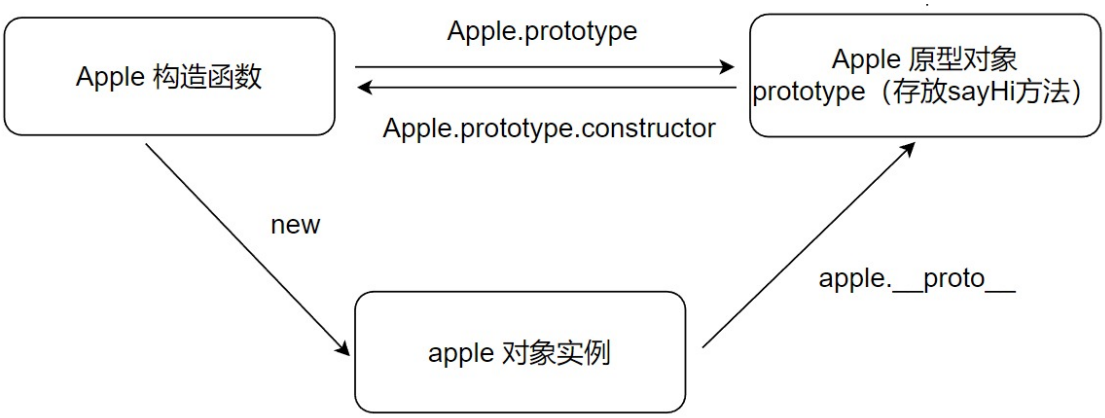
构造函数原型对象prototype

- 解决了构造函数的资源浪费问题。
- 构造函数通过原型分配的函数时所有对象**共享的。**
- 在JS中，**每一个构造函数都有一个**`prototype`**属性**，指向另一个对象。注意这个`prototype`就是一个对象，这个对象的所有属性和方法，都会被构造函数所拥有。
- **我们可以把不变的方法，直接定义在**`prototype`**上，这样所有对象的实例就可以共享这些方法。**这样一来所有的实例化对象，都不用被独自开辟方法，可以共同去`ptototype`中寻找使用方法。

```
function Apple(name){
    this.name = name;
}
Apple.prototype.sayHi = function(){
    console.log('hi');
}
var apple1 = new Apple('iphone12');
var apple2 = new Apple('iphone13');
console.log(apple1.sayHi === apple2.sayHi); // true
console.log(apple1._proto_ === Apple.prototype); // true
```

> 为什么方法定义在构造函数的prototype上,而实例化对象可以使用

- 在实例化对象身上还有`__proto__`属性指向构造函数的`prototype`原型对象，我们之所以可以使用构造函数`prototype`原型对象的属性和方法，就时因为对象有`_proto_`原型的存在。
- `__proto__`**对象原型**和**原型对象**`prototype`时等价的。



constructor构造函数

- 对象原型(`_proto`)和构造函数(`prototype`)原型对象里面都有一个`constructor`属性，我们将其称为构造函数，因为他会指回构造函数本身。
- `constructor`主要用于记录该对象引用哪个构造函数，它可以让原型对象重新指向原来的构造函数。

```
function Apple(name){
    this.name = name;
}
Apple.prototype.sayHi = function(){
    console.log('hi');
}
var apple1 = new Apple('iphone12');
var apple2 = new Apple('iphone13');
console.log(Apple.prototype);   // 里面存在constructor属性
console.log(apple1.__proto__);  // 里面存在constructor属性
console.log(Apple.prototype.constructor);   // 指向构造函数本身Apple(name){...}
console.log(apple1.__proto__.constructor);  // 指向构造函数本身Apple(name){...}
```

- 我们可以在原型对象中直接赋值添加方法**会覆盖掉**`constructor`，这时我们需要手动指会构造函数。

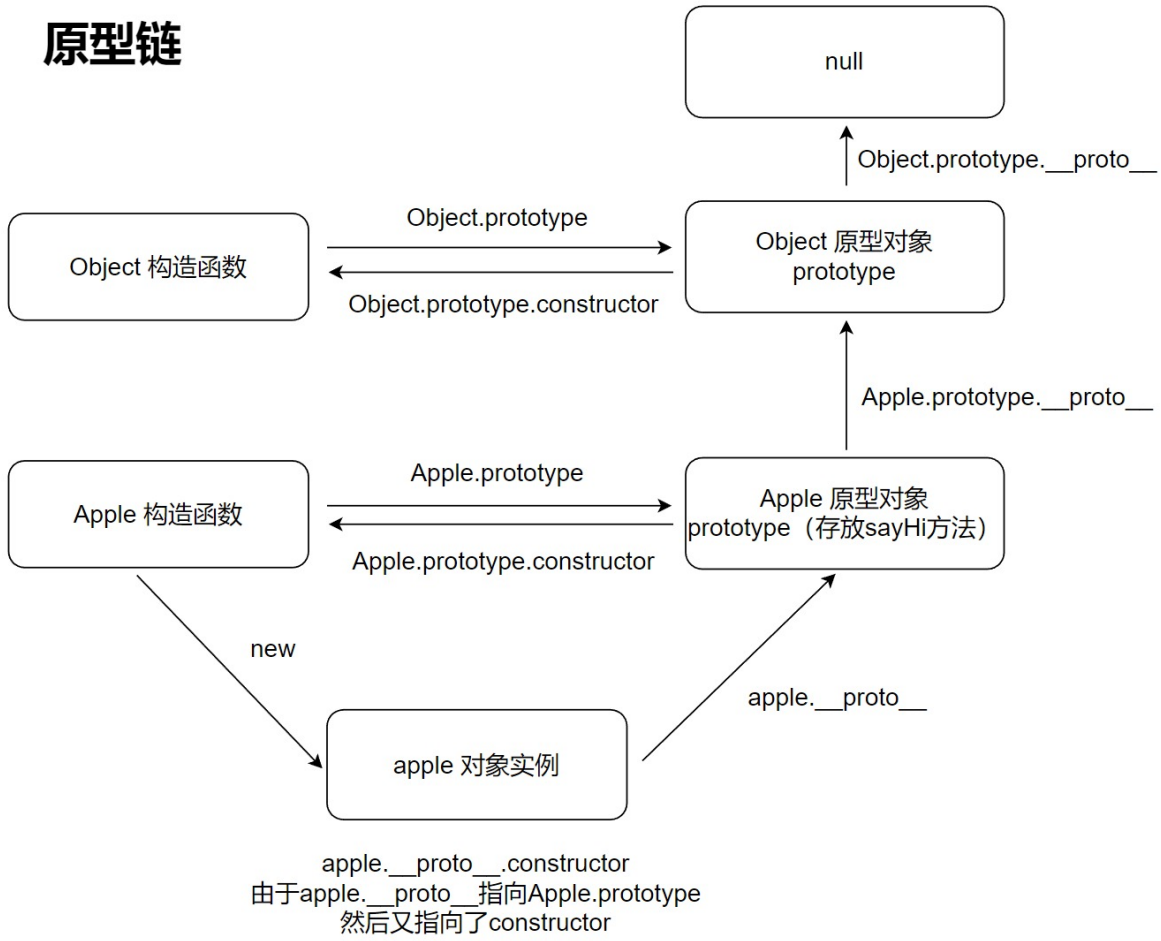
```
function Apple(name){
    this.name = name;
}
Apple.prototype = {
    // 手动指回
    constructor: Apple,
    sayHi: function() {
        console.log('hi');
    },
    open: function() {
        console.log('open');
    }
}
```

```
var apple1 = new Apple('iphone12');
var apple2 = new Apple('iphone13');

console.log(Apple.prototype);   // 如果没有手动指回，里面的constructor被覆盖掉，所以不存在constructor
console.log(apple1.__proto__);  // 如果没有手动指回，里面的constructor被覆盖掉，所以不存在constructor
console.log(Apple.prototype.constructor);
console.log(apple1.__proto__.constructor);
```

总结：如果我们修改了原来的原型对象，修改时如果以对象的形式来进行赋值的话，必须手动利`constructor`指向回原来的构造函数。

原型链



JS成员查找机制

1. 当访问一个对象的属性（包括方法）时，首先查找这个**对象自身**有没有该属性。
2. 如果没有就查找它的原型（也就是`__proto__`指向的`prototype`**原型对象**）。
3. 如果还有没就查找原型对象的原型（Object的原型对象）。
4. 依此类推一直找到Object为止（null）。

原型对象中this的指向

- 构造函数和原型对象中，`this`都指向实例对象。

```
function Apple(name){
    this.name = name;
}
let that;
Apple.prototype.sayHi = function(){
    that = this;
    console.log('hi');
}
var apple1 = new Apple('iphone12');
apple1.sayHi();
console.log(that === apple1); // true
```

原型对象扩展内置对象

- 我们可以通过原型对象，对原来的内置对象进行拓展自定义的方法。比如给数组增加自定义求偶数和的功能。

```
// 因为数组对象中并没有内置求和方法，所以我们可以通过原型对象进行添加
Array.prototype.sum = function() {
    var sum = 0;
    for (let i = 0; i < this.length; i++) {
        sum += this[i];
    }
    return sum;
    let arr = [1, 2, 3];
    console.log(arr.sum()); // 返回求和结果6
}
```

注意：

1. 通过`Array.prototype.sum = function()`方法添加的时在原型对象中进行追加。不会覆盖掉原来的原型对象。
2. 如果使用`Array.prototype = {}`方法进行添加时，这里时重新定义整个原型对象，会覆盖掉原来所有的方法和属性如排序和反转数组。

所以对内置对象添加新的方法只能通过第一种方式添加，第二种方式会覆盖掉本来的原型对象中的方法

继承

ES6之前并没有给我们提供extends继承。我们可以通过**构造函数+原型对象**模拟实现继承，被称为**组合继承。**

call()

调用这个函数，并且修改函数运行时的this指向

```
fun.call(thisArg, arg1, arg2, ...)
```

- thisArg：当前调用函数this的指向对象

- arg1, arg2：传递的其他参数

```
function fn(x, y) {
    console.log('fun');
    console.log('this');
    console.log(x + y);
}
var o = {
    name: 'andy'
};
fn();
// 1. call() 可以调用函数
fn.call();
// 2. call() 可以改变这个函数的this指向 此时这个函数的this 就指向了O这个对象
fn.call(o, 1, 1);
```

借用父构造函数继承属性

```
function Father(uname,age){
    this.uname = uname;
    this.age = age;
}

function Son(uname, age, score){
    // 调用Father将里面的Father中的this指向为Son中的this，本质指向了实例对象
    Father.call(this, uname, age);
    this.score = score;
}
var son = new Son('brokyz', 18);
console.log(son);
```

- 在Son中通过call()调用了Father，将Father中的this指向了Son中的this，也就是实例对象son，将相应参数传入调用的Father，这样就可以利用Father生成相应的属性，实现了函数的继承。
- 注意这种方法是继承属性，不是继承方法。方法的继承需要借用原型对象继承。

借用原型对象继承方法

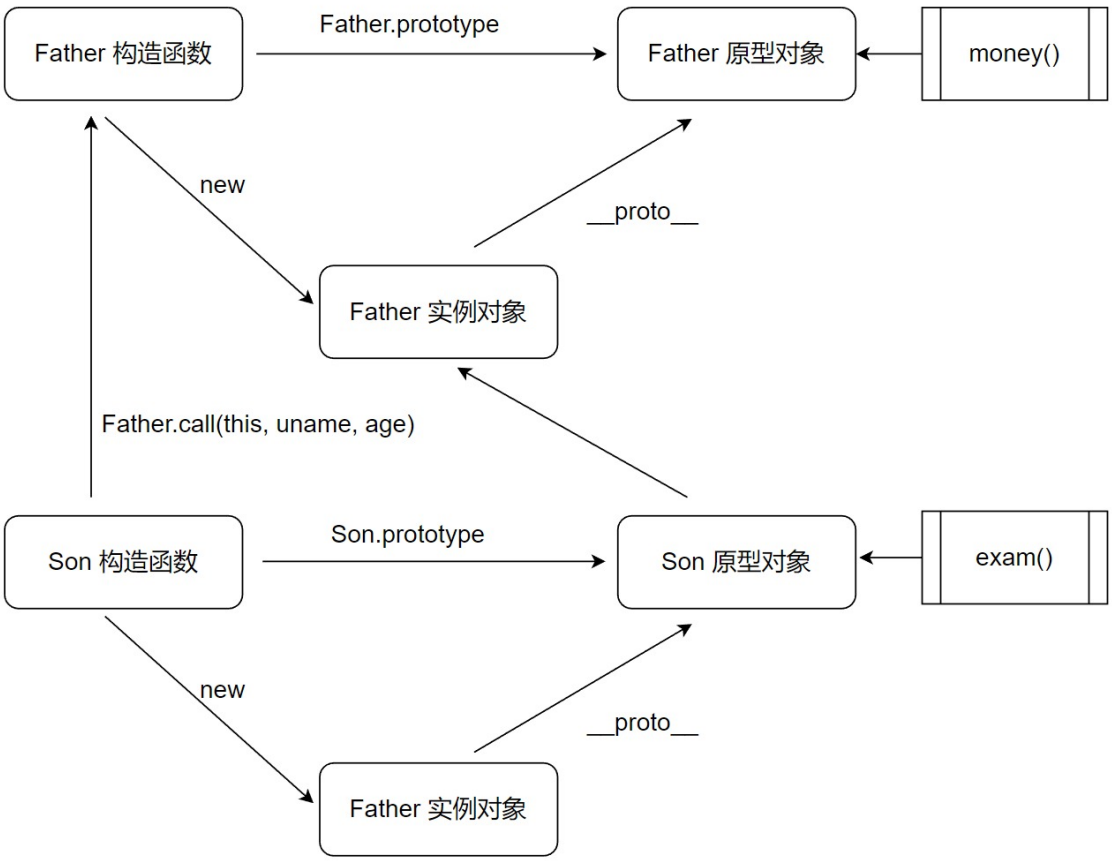
```
function Father(uname, age) {
    this.uname = uname;
    this.age = age;
}

function Son(uname, age, score) {
    // 调用Father将里面的Father中的this指向为Son中的this，本质指向了实例对象
    Father.call(this, uname, age);
    this.score = score;
}
// Son.prototype = Father.prototype;
// 浅拷贝这样直接赋值会有问题，如果修改了子原型对象，父原型对象也会跟着修改
// 让Son的原型对象指向Father的一个实例对象，Father的实例可以使用其原型对象中的方法，这样就实现了方法的继承
Son.prototype = new Father();
// 如果利用对象的形式修改了原型对象，别忘了利用constructor指回原来的原型对象
// 因为这里是用Father实例对象覆盖了Son的原型对象
```

```
Son.prototype.consturctor = Son;
// 子构造函数专门的方法
Son.prototype.exam = function () {
    console.log('exam');
};
Father.prototype.money = function() {
    console.log('money');
}
var son = new Son('brokyz', 18, 100);
console.log(son);
son.money()
```

原理

ES5继承原理



类和对象（ES6新增）

创建类

语法：

```
class Star {
    // class body
}
```

创建实例：

```
var xx = new Star();
```

构造函数

- 当生成实例时将会自动执行构造函数。
- 如果省略不写时，会自动生成构造函数。

语法：

```
class Star {
    constructor(name){
        this.name = name;
    }
}
```

方法

- 共有的方法可以直接写道类中。

语法：

```
class Star {
    constructor(name){
        this.name = name;
    }

    sayHi(){
        console.log("Hi!")
    }
}
```

继承

- 子类可以继承父类的方法。

语法：

```
class Father {
}
class Son extends Father {
}
```

super

- super允许子类调用父类的方法和构造函数

```
class Father {
    constructor(x,y){
        this.x = x;
        this.y = y;
    }
    sum(){
        console.log(this.x + this.y);
    }
}
```

```
class Son extends Father {
    constructor(x,y){
        // 使用父类的构造方法
        super(x,y);
    }
}
new Son(1, 2).sum();
```

就近原则重写父类方法

- 当子类重写了父类的方法时，调用时调用的时子类的方法。
- 如果想使用父类方法需要用到super

```
class Father {
    say(){
        console.log("father");
    }
}
class Son extends Father {
    say(){
        // 调用父类的say()
        console.log(super.say() + "son");
    }
}
let son = new Son();
son.say();  // 调用的是子类的方法
```

> 子类可以对父类的方法进行拓展

- 子类中必须在构造函数中使用super，必须放在this前面，才可以使用父类的方法

```
class Father {
    constructor(x,y){
        this.x = x;
        this.y = y;
    }
    sum(){
        console.log(this.x + this.y);
    }
}

class Son extends Father {
    constructor(x,y){
        // 使用父类的方法时需要将参数传递给父类，因为父类中的this是父类中的
        super(x,y); // 必须写道this前面
        this.x = x;
        this.y = y;
    }

    subtract(x,y){
        console.log(this.x - this.y);
    }
}

let son = new Son(5, 3);
```

```
son.sum();
son.substact();
```

类的注意点

- ES6中的类没有变量提升，必须先声明再使用。
- 在类中使用共有属性和方法必须要加this，调用本类中的方法也是需要加this
- 当获取对象时，也是需要用this的
- this的指向问题，谁调用指向谁

```
let that;
class Star {
    constructor(name){
        that = this;
        this.name = name;
        this.sayHi();
        this.btn = document.querySelector('button');
        this.btn.onclick = this.sayHi; //这里调用的是函数名所以不加括号。
        // 这里btn调用了函数，所以函数中的this为btn，因此this.name无法输出name
        // 可以给一个全局变量赋值，然后调用全局变量来解决
    }

    sayHi(){
        console.log("Hi!" + this.name);
        console.log("Hi!" + that.name);
    }
}
```

类的本质

- 在ES6之前通过**构造函数+原型对象**实现面向对象编程
- 在ES6通过**类**实现面向对象编程

类的本质其实还是一个函数，我们也可以简单的认为类就是构造函数的另外一种写法。

类和构造函数都有以下特点：

1. 构造函数有原型对象prototype
2. 构造函数原型对象prototype里面有constructor指向构造函数（类）本身
3. 构造函数可以通过原型对象添加方法
4. 构造函数创建的实例对象有`__proto__`原型指向构造函数的原型对象

```
class Star {
    constructor(uname) {
        this.uname = uname;
    }
    eat() {
        console.log('eat');
    }
}
console.log(typeof Star);
// 1. 类有原型对象
console.log(Star.prototype);
// 2. 类的原型对象里面有constructor
```

```
console.log(Star.prototype.constructor);
// 3. 类可以通过原型对象添加方法
Star.prototype.sing = function() {
    console.log('sing');
}
var star = new Star('brokyz');
console.log(star);
console.log(star.__proto__ === Star.prototype); // true
```

ES5中的新增方法

数组方法

迭代（遍历）方法：forEach()、map()、filter()、some()、every()

forEach()

```
array.forEach(function(currentValue, index, arr))
```

- currentValue：数组当前项的值
- index：数组当前项索引
- arr：数组对象本身

```
var arr = [1, 2, 3];
arr.forEach(function(value, index, arr) {
    console.log('数组元素:' + value);
    console.log('数组索引:' + index);
    console.log('数组本身:' + arr);
})
```

filter()

```
array.filter(function(currentValue, index, arr))
```

- filter()方法创建一个新的数组，新数组中的元素是同过检查指定数组中符合条件的所有元素，**主要用于筛选数组**
- **注意它直接返回一个新数组**
- currentValue：数组当前项的值
- index：数组当前项的索引
- arr：数组对象本身

```
var arr = [12,66,4,88];
var newArr = arr.filter(function (value, index, arr) {
    return value >= 20;
})
console.log(newArr);
```

some()

```
array.some(function(currentValue, index, arr))
```

- some()方法用于检测数组中的元素是否满足指定条件。就是查找数组中是否有满足条件的元素。
- **注意它的返回值是布尔值，如果查找这个元素，就返回true，如果查不到就返回false。**
- 如果找到第一个满足条件的元素，则终止循环不在继续查找。
- currentValue：数组房前项的值
- index：数组当前项的索引
- arr：数组对象本身

```
var arr = [12, 66, 4, 88];
var flag = arr.some(function (value, index, arr) {
    return value < 2;
});
console.log(flag);
```

some和forEach的区别

- 在forEach()里面，遇到了return会跳出本次循环但是继续下一次迭代（filter和forEach相同）

```
var arr = ['pink', 'green', 'red', 'blue'];
arr.forEach(function (value) {
    if (value == 'green') {
        console.log('找到了元素');
        return true;    // 在forEach里面，遇到了return会跳出本次循环但是继续下一次迭代
    }
    console.log(22);
})
```

- 在some()里面，遇到了return true就是终止遍历 迭代效率更高

```
var arr = ['pink', 'green', 'red', 'blue'];
arr.some(function (value) {
    if (value == 'green') {
        console.log('找到了元素');
        return true;    // 在some里面，遇到了return true就是终止遍历 迭代效率更高
    }
    console.log(11);
});
```

trim()去除字符串空格

- trim()可以去除两侧空格

```
var str = '  an  dy  ';
console.log(str);
var str1 = str.trim();
console.log(str1);
```

对象方法

Object.keys()

遍历对象中的属性和方法

```
Object.keys(obj)
```

Object.defineProperty()

定义对象中新属性和修改原有的属性。

```
Obeject.defineProperty(obj, prop, desriptor)
```

- obj：必须。目标对象
- prop：必须。需定义或修改的属性的名字
- descriptor：必须。目标属性所拥有的特性
  - value：设置属性的值 默认为undefined
  - writable：值是否可以重写。true|flase 默认为false
  - enumerable：目标属性是否可以被枚举。true|false 默认为false
  - configurable：目标属性是否可以被删除或是否可以再次修改特性。true|false 默认为false

```
var obj = {
    id: 1,
    pname: '小米',
    price: 1999
};
// 1. 以前的对象添加和修改属性的方式
// obj.num = 1000;
// obj.price = 99;
// 2. Object.defineProperty() 定义新属性或修改原有的属性
Object.defineProperty(obj, 'num', {
    value: 1000
});
Object.defineProperty(obj, 'price', {
    value: 9.9
});
Object.defineProperty(obj, 'id', {
    writable: false
});
Object.defineProperty(obj, 'address', {
    value: 'beijing',
    writable: false,
    enumerable: false,
    configurable: false // 如果为false则不允许删除属性，不允许再修改特性。
});
delete obj.address;
obj.id = 2;
console.log(obj);
console.log(Object.keys(obj)); // ['id', 'pname', 'price']
// 通过defineProperty添加的属性才会默认带有enumerable: false
```

函数进阶

函数的定义与调用

1. 函数声明方式function关键字（命名函数）

```
function fn(){}
```

2. 函数表达式（匿名函数）

```
var fn = function(){}
```

3. new Function() 不常用效率低

```
var fn = new Function('a','b', 'console.log(a+b)');
fn(1,2);
console.dir(fn);
console.log(fn intanceof Object);
```

- Function里面参数都必须是字符串格式
- 第三种方式执行效率低，也不方便书写，因此较少使用
- 所有函数都是Function实例（对象）
- 函数也属于对象

函数的调用方式

1. 普通函数

```
function fn() {
    console.log('test');
}
fn();
```

2. 对象的方法

```
var o = {
    sayHi: function() {
        console.log('test');
    }
}
o.sayHi();
```

3. 构造函数

```
function Star() {

}
new Star();
```

4. 绑定事件函数

```
btn.onclick = function() {
    // 点击了按钮可以调用这个函数
}
```

5. 定时器函数

```
setInterval(function(){
    // 定时器自动1秒执行一次
}, 1000)
```

6. 立即执行函数

```
(function(){
    console.log('test');
})(); // 立即执行函数会自动调用
```

函数内部的this指向问题

调用方式	this指向
普通函数调用	window
构造函数调用	实例对象，原型对象里面的方法也指向实例对象
对象方法调用	该方法所属对象
事件绑定方法	绑定事件对象
定时器函数	window
立即执行函数	window

1. 普通函数 this指向调用者

```
function fn() {
    console.log(this);
}
fn(); // 指向window，因为这是window.fn()的缩写
```

2. 对象方法 this指向调用者，一般为window

```
var o = {
    sayHi: function() {
        console.log(this);
    }
}
o.sayHi(); //指向对象o
```

3. 构造函数和原型对象 this指向调用它的实例对象

```
var that1;
var that2;
function Star() {
    that1 = this;
}
Star.prototype.sing = function() {
    that2 = this;
}
var brokyz = new Star();
brokyz.sing();
console.log(that1);
console.log(that2);
```

4. 绑定事件函数 this指向调用者

```
btn.onclick = function() {
    console.log(this);
}
```

5. 定时器中的this也是执行window

```
setInterval(function(){
    console.log(this);
}, 1000)
// 指向window因为是 window.setInterval()的缩写
```

6. 立即执行函数 中的this指向window，类似于普通函数，只不过不需要我们手动调用

```
(function(){
    console.log(this);
})(); // 立即执行函数会自动调用
```

改变函数内部this指向

JS为我们专门提供了一些函数方法来帮助我们更优雅的处理函数内部this的指向问题，常用的有bind()、call()、apply()三种方法

call()

调用这个函数，并且修改函数运行时的this指向

```
fun.call(thisArg, arg1, arg2, ...)
```

- thisArg：当前调用函数this的指向对象
- arg1, arg2：传递的其他参数

```
function fn(x, y) {
    console.log('fun');
    console.log('this');
    console.log(x + y);
}
var o = {
    name: 'andy'
};
fn();
// 1. call() 可以调用函数
fn.call();
// 2. call() 可以改变这个函数的this指向 此时这个函数的this 就指向了0这个对象
fn.call(o, 1, 1);
```

apply()

aplly()方法调用一个函数。简单理解为调用函数的方式，但是它可以改变函数的this指向。

```
fun.apply(thisArg, [argsArray])
```

- thisArg：在fun函数运行时指定的this值
- argsArray：传递的值，必须包含在数组里面
- 返回值就是函数的返回值，因为它就是调用函数

```
var o = {
    name: 'brokyz'
}
function fn(arr) {
    console.log(this);
    console.log(arr);
}
fn.apply(o, ['pink']);
// 1. 调用函数 第二个可以改变函数内部的this指向
// 2. 但是他的参数必须是数组(伪数组)
// 3. apply的主要应用 比如我们可以利用aplly借助于数学内置对象求最大值
var arr = [1,66,3,99,4];
var max = Math.max.apply(Math, arr)
```

bind()

bind()方法不会调用函数。但是能够改变函数内部的this指向

```
fun.bind(thisArg, arg1, arg2, ...)
```

- thisArg：在fun函数运行时指定的this值
- arg1，arg2：传递的其他参数
- 返回由指定的this值和初始化参数改造的**原函数拷贝**。也就是说返回的时改完的新新函数

```
var o = {
    name: 'brokyz'
}
function fn() {
    console.log(this);
}
var f = fn.bind(0);
f();
// 1. 不会调用原来的函数
// 2. 可以改变原来的函数内部的this指向
// 3. 返回的是原函数改变this之后的新函数
```

例子：当点击按钮之后就禁用这个按钮，3秒后自动开启

```
var btn = document.querySeletor('button');
btn.onlcick = function() {
    this.disabled = true;
    setTimeout(function(){
        this.disabled = false; // this指向的是window，所以要用bind
    }.bind(this), 3000)
}
```

三者的总结

相同点：

都可以改变函数内部的this指向。

区别：

1. call和apply会调用函数，并且改变函数内部this的指向。
2. call和apply传递的参数不一样，call传递参数arg1，arg2，...的形式，apply必须数组形式[arg]
3. bind不会调用函数，可以改版函数内部this指向

使用场景：

1. call经常用作继承。
2. apply经常和数组有关系的操作。比如借助数学对象实现数组求最大值。
3. bind不调用函数，但是还想改变this的指向。比如改变定时器内部this的指向。

严格模式

JS除了提供正常模式外，还提供了**严格模式（strict mode）**。ES5的严格模式是采用具有限制性JS变体的一种方式，即在严格的条件下运行JS代码。

严格模式在IE10以上的版本的浏览器中才会被支持，旧版本浏览器中会被忽略。

严格模式对正常的JavaScript语义做了一些更改：

1. 消除了JS语法的一些不合理、不严谨之处，减少了一些怪异的行为。
2. 消除代码运行的一些不安全之处，保证代码运行的安全。
3. 提高编译器效率，增加运行速度。
4. 禁用了在EMCAScript的未来版本中可能会定义的一些语法，为未来新版本的JS做好铺垫。比如一些保留字如：class,enum,export,extends,super不能作为变量名。

开启严格模式

1. 给整个脚本添加严格模式

```
<script>
    'use strict';
</script>
```

或:

```
<script>
    (function(){
        'use strict';
    })();
</script>
```

2. 为某个函数开启严格模式

```
<script>
    function fn(){
        'use strict';
    }
    function f(){
        'use strict';
    }
</script>
```

严格模式的变化

变量规范

1. 在正常模式中，如果一个变量没有声明就赋值，默认是全局变量。严格模式禁止这种用法，变量都必须先用var命令声明，然后再使用。
2. 严禁删除已经生命好的变量。

```
'use strict';
// 1. 变量必须先声明
// num = 10;
var num = 10;
console.log(num);
// 2. 我们不能随意删除已经声明好的变量
// delete num;
```

严格模式下的this指向

1. 在正常模式下，全局作用域中的this指向window对象
2. 严格模式下，全局作用域中的this是undefined

```
'use strict';
function fn(){
    console.log(this)
}
fn();
```

3. 正常模式下，构造函数不加new也可以调用，this指向全局对象
4. 严格模式下，构造函数如果不加new调用，this会报错
5. 如果是通过new创建构造函数，那么this依然指向实例对象

```
'use strict';
function Star(){
    this.sex = '男'; // 在严格模式下这个this指向undefined
}
Star();
```

6. 严格模式下，定时器的this还是指向window
7. 严格模式下，事件、对象还是指向调用者

函数的变化

1. 函数不允许有重名的参数。

```
function fn(a,a){
    console.log(a+a);
}
fn(1,2); // 输出为4，以为传入完成时a为2，计算的是2+2=4。严格模式不允许这样。
```

2. 函数必须声明在顶层。新版本的JS会引入"块级作用域"（ES6已经引入）。为了与新版本接轨。不允许在非函数的代码块内声明函数。比如不允许在if和for中声明函数。但是依然可以在函数中声明函数来完成函数的嵌套。

高阶函数

高阶函数是对其他函数进行操作的函数，它**接受函数作为参数**或**将函数作为返回值输出**。满足其一就是高阶函数。

```
// 将函数作为参数，可以为函数作为回调函数使用。
function fn(a,b,callback){
    console.log(a+b);
    callback&&callback();
}
fn(1,2,function(){alert('hi')})

// 将函数作为参数返回
function fn() {
    return function(){}
}
fn();
```

闭包

闭包指有权访问另一个函数作用域中变量的**函数。**------JavaScript高级程序设计

简单理解就是，有个作用域可以访问另外一个函数内部的局部变量。

闭包的主要作用：延伸了变量的作用范围

```
function fn() {
    var num = 10;
    function fun() {
        console.log(num);
    }
    fun();
}
fn();
```

用fn外面的作用域访问fn内部的局部变量

```
function fn() {
    var num = 10;
    return function() {
        console.log(num);
    }
}
var f = fn();
f();
```

- 所以闭包不及让函数内部的作用域访问局部变量，也可以使用全局作用域来访问函数的局部变量。

闭包应用-点击li输出当前li的索引号：

```
<body>
    <ul class="nav">
        <li>榴莲</li>
        <li>小汉堡</li>
        <li>臭豆腐</li>
        <li>腐乳</li>
    </ul>
    <script>
        // 闭包应用-点击li输出当前li的索引号
        // 1. 我们可以利用动态添加属性的方式
        var lis = document.querySelector('.nav').querySelectorAll('li');
        // for (var i = 0; i < lis.length; i++) {
        //     lis[i].index = i;
        //     lis[i].onclick = function () {
        //         console.log(this.index);
        //     };
        // }
        // 2.利用闭包的方式得到当前小li的索引号
        for (var i = 0; i < lis.length; i++) {
            // 利用for循环创建了4个立即执行函数
            (function (i) {
                // console.log(i);
                lis[i].onclick = function () {
                    console.log(i); // 调用了立即执行函数的i，使用了闭包
                };
```

```
            })(i);
        }
    </script>
</body>
```

闭包应用-3秒钟之后，打印所有li元素的内容：

```
<body>
    <ul class="nav">
        <li>榴莲</li>
        <li>小汉堡</li>
        <li>臭豆腐</li>
        <li>腐乳</li>
    </ul>
    <script>
        // 闭包应用-3秒钟之后，打印所有li元素的内容：
        var lis = document.querySelector('.nav').querySelectorAll('li');
        for (var i = 0; i < lis.length; i++) {
            // 回调函数只有触发了才会执行，这是异步任务，会先进入任务队列中等待执行
            // setTimeout(function () {
            //     console.log(lis[i]);
            // }, 2000);
            (function (i) {
                setTimeout(function () {
                    console.log(lis[i].innerHTML);
                }, 2000);
            })(i);
        }
    </script>
</body>
```

闭包应用-计算打车价格

```
// 闭包应用-计算打车价格
// 打车起步价13（3公里内），之后每多一公里增加5元，用户输入公里数就可以计算打车价格
// 如果有拥堵状况，总价格多收取10块钱拥堵费
var car = (function () {
    var start = 13; // 起步价
    var total = 0;    // 总价
    return {
        price: function (n) {
            if (n <= 3) {
                total = start;
            } else {
                total = 13 + (n - 3) * 5;
            }
            return total;
        },
        yd: function (flag) {
            return flag ? total + 10 : total;
        }, // 拥堵之后的费用
    };
})();
console.log(car.price(1));
```

```
console.log(car.yd(true));
console.log(car.price(5));
console.log(car.yd(true));
```

- 由于函数price和yd使用了其他函数中的局部变量start和total，所以产生了闭包，而包含这个变量的函数就被称为闭包函数

递归函数

如果**一个函数在内部可以调用其本身**，那么这个函数就是**递归函数**

递归函数的作用和循环效果一样

由于递归很容易发生“栈溢出”错误（stack overflow），所以**必须要加退出条件return**

```
// 递归函数：函数内部自己调用自己，这个函数就是递归函数
var num = 1;
function fn(){
    console.log('我要打印6句话');
    if (num == 6) {
        return; // 递归里面必须加退出条件
    }
    num++;
    fn();
}
fn();
```

利用递归求1-n阶乘

```
function fn(n){
    if(n == 1) {
        return 1;
    }
    return n * fn(n-1);
}
fn(3);
// return 3 * fn(2)
// return 3 * 2 * fn(1)
// return 3 * 2 * 1
```

利用递归函数求斐波那契数列

```
function fn(n) {
    if (n == 1 || n == 2){
        return 1;
    }
    return fn(n-1) + fn(n-2);
}
fn(4);
```

根据id号返回相应的数据

```
var data = [{
    id: 1,
```

```
    name: '家电',
    goods: [{
        id: 11,
        gname: '冰箱',
        goods: [{
            id: 111,
            gname: '海尔'
        }]
    }, {
        id: 12,
        gname: '洗衣机'
    }]
}, {
    id: 2,
    name: '服饰'
}];
// 1. 利用 forEach去遍历里面的每一个对象
function getID(json, id) {
    var o = {};
    json.forEach(function (item) {
        if (item.id == id) {
            // console.log(item);
            o = item;
            // 2. 我们想要得到里层的数据 11 12 可以利用递归函数完成
            // 里面应该有goods数组并且数组的长度不为0
        } else if (item.goods && item.goods.length > 0) {
            o = getID(item.goods, id);
        }
    });
    return o;
}
console.log(getID(data, 11));
console.log(getID(data, 111));
```

浅拷贝 Object.assign()

浅拷贝只拷贝一层，更深层次对象级别的只拷贝引用

可以通过Object.assign()来实现浅拷贝

```
Object.assign(obj2,obj1)
// 将obj1浅拷贝给obj2
```

```
var obj = {
    id: 1,
    name: 'andy',
    msg: {
        age: 18
    }
};
var o = {};
// for(var k in obj) {
//     // k是属性名字 obj[k]是属性值
//     o[k] = obj[k];
```

```
// }
// console.log(o)
// o.msg.age = 20;
// console.log(obj)
Object.assign(o, obj);
console.log(o);
// 由于浅拷贝更深层次对象只拷贝引用，所以o修改msg.age时obj也会同时跟着修改
```

深拷贝

拷贝所有层

```
var obj = {
    id: 1,
    name: 'andy',
    msg: {
        age: 18
    }
};
var o = {};
function deepCopy(newobj, oldobj) {
    // 判断我们的属性值属于哪种数据类型
    for (var k in oldobj) {
        // 1. 获取属性值
        var item = oldobj[k];
        // 2. 判断这个值是否是数组
        if (item instanceof Array) {
            newobj[k] = [];
            deepCopy(newobj[k], item);
        } else if (item instanceof Object) {
            // 3. 判断这个值是否是对象
            newobj[k] = {};
            deepCopy(newobj[k], item);
        } else {
            // 4. 属于简单数据类型
            newobj[k] = item;
        }
    }
}
deepCopy(o, obj);
console.log(o);
```

正则表达式

正则表达式（Regular Expression是用于匹配字符串中字符组合的模式。在JavaScript中，正则表达式也是对象。

正则表达式通常被用来检索、替换那些符合某个模式（规则）的文本，例如表单验证：用户名表单只能输入英文字符、数字或者下划线，昵称输入框中可以输入中文（**匹配**）。此外，正则表达式还常用于过滤页面内容的一些敏感词（**替换**），或从字符串中获取我们想要的特定部分（**提取**）等。

正则表达式的特点：

1. 灵活性、逻辑性和功能性非常强。
2. 可以迅速的用极简单的方式达到字符串的复杂控制。

3. 对于刚接触的人来说，比较晦涩难懂。
4. 实际开发中，一般都是直接复制写好的正则表达式，但是要求会使用正则表达式并且根据实际情况修改正则表达式。

创建正则表达式

在JS中可以通过两种方式来创建一个正则表达式。

1. 通过调用RegExp对象的构造函数创建

```
var 变量名 = new RegExp(/表达式/);
```

2. 利用字面量来创建

```
var 变量名 = /正则表达式/;
```

测试正则表达式 test

test()正则对象方法，用于检测正则表达式是否符合规范，该对象会返回true或flase，其参数是测试字符串。

```
regexObj.test(str)
```

1. regexObj是写的正则表达式
2. str是我们要测试的文本
3. 就是检测str文本是否符合我们写的正则表达式规范

```
var reg = /123/;
reg.test(123);  //true
reg.test('abc'); //false
```

边界符

`^`：表示以规定字符开头。

`$`：表示以规定字符结尾。

```
// 边界符 ^ $
var rg = /abc/;
// 只要包含abc这个字符串返回的都是true
rg.test('123adc123');   //true
// ^开头字符，规范了以规定字符开头
var reg = /^abc/;
rg.test('123adc123');   //false
rg.test('adc123');  //true
// $结尾符号，规范了以规定字符结尾
var reg = /^abc$/;
rg.test('123adc123');   //false
rg.test('adc123');  //false
rg.test('adc'); //true
```

字符类

`[]`：表示有一系列字符可供选择，只要匹配其中一个就可以了。

`-`：范围符号，表示一个范围选择。如`[a-z]`。

`[^]`：中括号内部的`^`表示取反符号，表示不能包含括号里的东西。

`()`：表示里面的字符要作为整体出现，而不是只出现一个。

```
var rg = /[abc]/;
// 只要包含a或者b或者c都返回true
console.log(re.test('andy'));   //true
console.log(re.test('baby'));   //true
console.log(re.test('color'));  //true
console.log(re.test('red'));    //false
var rg1 = /^[abc]$/;
console.log(re1.test('aa'));    //false
console.log(re1.test('a')); //true
var rg2 = /^[a-z]$/;    //表示任意一个小写英文字符
console.log(re2.test('f')); //true
console.log(re2.test('F')); //false
// 可以进行字符组合
var rg3 = /^[a-zA-Z]$/;
var rg4 = /^[^a-zA-Z]$/ //不能包含括号内的字符

var rg5 = /^(abc){3}$/;
console.log(re5.test('abc')); // false
console.log(re5.test('abcabcabc')); //true
```

量词符

用来规定某个模式出现的次数

`*`：重复0次或者更多次。

`+`：重复1次或者更多次。

`?`：重复0次或1次。

`{n}`：重复n次。

`{n,}`：重复n次或更多次。

`{n,m}`：重复n到m次。

```
var rg = /^[a]{3,6}$/;
console.log(re.test('aa')); //flase
console.log(re.test('aaa')); //true
console.log(re.test('aaaaaa')); //true
console.log(re.test('aaaaaaa')); //false
```

预定义类

预定类	说明
\d	匹配0-9之间的任一数字，相当于[0-9]
\D	匹配所有0-9之外的字符，相当于 `[^0-9]`
\w	匹配任意的字母、数字和下划线，相当于[A-Za-z0-9_]
\W	除了所有字母、数字和下划线的字符、相当于 `[^A-Za-z0-9_]`
\s	匹配空格（包括换行符、制表符、空格符等），相当于[\t\r\n\v\f]
\S	匹配非空格字符，相当于 `[^\t\r\n\v\f]`

正则表达式参数

```
/表达式/[switch]
```

switch（也称为修饰词）规定了正则表达式按照什么模式来进行匹配。有三种模式：

1. g：全局匹配
2. i：忽略大小写
3. gi：全局匹配和忽略大小写

```
var re = /test/g;
```

正则表达式的替换

replace()方法可以实现替换字符串的操作，用来替换的参数可以是一个字符串或是一个正则表达式。

可以使用正则表达式来匹配要替换的字符。

```
stringObject.replace(regexp/substr,replacement)
```

1. 第一个参数：被替换的字符串或者正则表达式
2. 第二个字符串：替换为字符串
3. 返回值：替换完的新字符串

```
// 替换replace
var str = 'andy和red';
var newStr = str.replace('andy','baby');
console.log(newStr);
var newStr1 = str.replace('/andy/','baby');
console.log(newStr);
```

替换敏感词

```
<body>
    <textarea name="words" id="" cols="30" rows="10"></textarea>
    <button>替换敏感词</button>
    <script>
        var text = document.querySelector('textarea');
        var btn = document.querySelector('button');
        btn.addEventListener('click', function (){
            console.log(text.value);
            text.value = text.value.replace(/激情/g,'**');
        });
    </script>
</body>
```

实例

用户名验证

```
// 规定用户名为6-16位，由字符、数字、_和-组成
var reg = /^[a-zA-Z0-9_-]{6,16}$/;
console.log(reg.test('brokyz_0223'));  //true
console.log(reg.test('brokyz_0223.!'));  //false
```

html

```
<body>
    <input id="user_name" type="text"><span id="user_text" style="color:#ccc">请输入用户名</span>
    <script>
        var userName = document.querySelector('#user_name');
        var userText = document.querySelector('#user_text');
        userName.addEventListener('blur', function () {
            if (/^[a-zA-Z0-9_-]{6,16}$/.test(this.value)) {
                userText.innerHTML = '√'
                userText.style.color = 'green';
            } else {
                userText.innerHTML = '×';
                userText.style.color = 'red';
            }
        });

    </script>
</body>
```

ES6

ES6简介

- ES的全程是EMCAScript，它是由EMCA国际标准化组织，指定的一项**脚本语言标准化规范**
- ES6就是JavaScript的一个发行版本, 于2015.06发行, 全称为ECMAScript. 一般来说JS我们刚开始学习的基础版本为ES5, ES6再ES5的基础上新增了一些新特性.
- ES6在ES5的基础上对其进行了升级, 解决了ES5的一些不足. 比如新增了类的概念.
- ES6在企业中应用很普遍.

新增声明方法

let

- let的新增是为了解决es5中使用var声明变量时存在的各种问题。

使用var的问题：

```
// 1. 使用var时，会存在声明提升的问题
console.log(test);
var test = 'test';
// 这里会把声明部分进行提升，将会输出test

// 2. 变量覆盖问题
var test = 'test1';
var test = 'test2';
console.log(test);
// 此处重复声明test变量，将会替换掉test变量的值

// 3. 块级作用域的问题
for (true){
    var test = 'test';
}
for (var i = 0; i < 2; i++){

}
console.log(test);  // test
console.log(i);  // 2
```

let对上述问题的改正：

```
// 1. 不会声明提升
console.log(test);
let test = 'test';
// 这里会报错，不会提前声明

// 2. 不可重复声明
let test = 'test1';
let test = 'test2';
console.log(test);
// 此处报错，不能重复声明已经存在的变量

// 3. 修复块级作用域，let声明只在所在的作用域有效
for (true){
    let test = 'test';
}
for (let i = 0; i < 2; i++){

}
console.log(test);
console.log(i);
// 此时输出将会输出test is not defined
// 4. 使用let声明的变量由暂时性死区的特性
```

```
//  ES6 规定，如果区块中存在 let 和 const 命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”，也就是说使用let声明的变量都是先声明再使用，不存在变量提升问题。
var num = 10;
if (true) {
    console.log(num);
    let num;
}
// 报错原因：在块作用域内，let声明的变量被提升，但变量只是创建被提升，初始化并没有被提升(初始化就是给变量先赋值成undefined)，在初始化之前使用变量，就会形成一个暂时性死区。
```

for循环计数器适合用let：

```
for (var i = 0; i < 10; i++) {
  setTimeout(function(){
    console.log(i);
  })
}
// 输出十个 10
for (let j = 0; j < 10; j++) {
  setTimeout(function(){
    console.log(j);
  })
}
// 输出 0123456789
/*
变量 i 是用 var 声明的，在全局范围内有效，所以全局中只有一个变量 i，每次循环时，setTimeout 定时器里面的 i 指的是全局变量 i ，而循环里的十个 setTimeout 是在循环结束后才执行，所以此时的 i 都是 10。

变量 j 是用 let 声明的，当前的 j 只在本轮循环中有效，每次循环的 j 其实都是一个新的变量，所以 setTimeout 定时器里面的 j 其实是不同的变量，即最后输出 12345。（若每次循环的变量 j 都是重新声明的，如何知道前一个循环的值？这是因为 JavaScript 引擎内部会记住前一个循环的值）。
*/
```

面试题：

```
var arr = [];
for (var i = 0; i < 2; i++) {
    arr[i] = function () {
        console.log(i);
    }
}
arr[0]();   //2
arr[1]();   //2
// 由于var声明的i并没有自己的块级作用域，所以等for循环结束之后，函数中的console.log(i)实际上调用的是循环之后的i值2

let arr = [];
for (let i = 0; i < 2; i++) {
    arr[i] = function () {
        console.log(i);
    }
```

```
}
arr[0]();   //0
arr[1]();   //1
// 由于let声明的i存在自己的块级作用域，所以本质想循环中的i不会公用，会分别在自己的块级空间中存在一个i，当调用了函数时，函数会首先从自己的块级作用域中向上进行查找i，所以输出的时自己所在的块级作用域中的i的值
```

const

- 新增const用来声明**常量**，一旦声明，常量的值就不可改变。
- 通常来讲**常量名使用大写。**
- 一旦对变量进行声明就必须初始化。
- 和let一样，也具有块级作用域

```
const SUM = 30;
const TEST; // 没有进行初始化将会报错
const SUM = 40; // 可以重新声明来改变值
```

- const不可更改常量的本质时，被const声明之后的内存地址不允许改变

```
const arr = [100, 200];
arr[0] = 'a';   //此操作不会改变arr的内存地址
arr[1] = 'b';   //此操作不会改变arr的内存地址
arr = ['a', 'b'];   //此操作会改变arr的内存地址，因此不被允许
```

- 所以对于基本数据类型不可更改，对于复杂数据类型数据本身不能更改，但是数据内部的值可以更改。

解构赋值

ES6中允许从数组中提取值，按照对应位置，对变量赋值。对象也可以实现解构。

数组解构

- 数组解构允许我们按照一一对应的关系从数组中提取值，然后赋值给变量
- 如果解构不成功，那么变量的值位undefined

```
let [a, b, c] = [1, 2, 3];
console.log(a);
console.log(b);
console.log(c);
```

对象解构

- 对象解构允许我们使用变量的名字匹配对象的属性，匹配成功将对象属性的值赋值给变量

```
let person = {name: 'borkyz', age: 20, sex: '男'};
let {name, sex} = person;
let {age} = person;
console.log(name);
console.log(sex);
console.log(age);
```

- 对象解构还允许我们将提取出来的值赋值给另外的变量

```
let person = {name: 'borkyz', age: 20, sex: '男'};
let {name: myName, sex: mySex} = person;
console.log(myName);
console.log(mySex);
```

箭头函数

- 箭头函数是用来简化函数定义语法的。
- 如果传入参数只有一个那么小括号可以省略/
- 在箭头函数中，如果函数体中只有一句代码，并且代码执行结果就是函数的返回值，函数体大括号就可以省略。

```
(arg1, arg2, ...) => {code}
```

```
let sum = (a, b) => {
    console.log(a);
    console.log(b);
    console.log(a+b);
    return a+b;
}
// 传入参数是一个那么可以省略小括号
let put = a => {
    console.log(a);
}

// 一行代码并且返回执行结果就可以省略大括号
let plus = (a, b) => a + b;
```

箭头函数的this指向

箭头函数不绑定this，因此箭头函数没有自己的this关键字，如果在箭头函数中调用this，那么this应该指向箭头函数定义位置中的this。

```
function fn () {
    console.log(this);
    return () => {
        console.log(this);
    }
}

const obj = {name: 'brokyz'};
const resFn = fn.call(obj);
// 输出结果
```

面试题

```
var obj = {
    age: 20,
    say: () => {
        alert(this.age);
    }
}

obj.say(); // undefined
// 箭头函数定义在了obj对象中，由于obj是对象，所以obj没有产生作用域，所以这个箭头函数实际上是被定义在了全局作用域下。因此this指向window，window中没有age所以弹出了undefined
```

剩余参数

- 剩余参数语法允许我们将一个不定数量的参数表示为一个数组。
- `arguments`将会以伪数组的形式存储我们传入函数的所有参数。
- 在箭头函数中无法使用`arguments`。

```
function fn(first, ...args) {
    console.log(first); // 10
    console.log(args);  // [20, 3]
    console.log(arguments); // [10, 20, 30]伪数组
}
fn(10, 20, 30);
// 箭头函数中无法使用 arguments
const sum = (...args) => {
    let total = 0;
    args.forEach(item => total += item);
    return total;
};
console.log(sum(10, 20, 30));   // 60
```

剩余参数和解构配合使用

```
let student = ['wangwu', 'zhangsan', 'lisi'];
let [st1, ...st2] = student;
console.log(st1);   // wangwu
console.log(st2);   // ['zhangsan', 'lisi']
```

ES6内置的对象扩展

Array 的扩展方法

扩展运算符（展开语法）

扩展运算符可以将数组或者对象转为用逗号分隔的参数序列。

```
let arr = [1, 2, 3];
console.log(...arr); // 1 2 3
// 注意：...arr实际上值是1, 2, 3带有逗号，但是输出时,被当作参数了
console.log(1, 2, 3); // 1 2 3
```

应用：合并数组

```
// 方法1
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let arr3 = [...arr1, ...arr2];
console.log(arr3);  // [1, 2, 3, 4, 5, 6]

// 方法2
let arr4 = arr1.push(...arr2);
```

应用：将伪数组转换为真正的数组

```
let lis = document.querySelectorAll('li');
arrLis = [...lis];
console.log(lis);   // NodeList(4) [li, li, li, li]
console.log(arrLis);    // (4) [li, li, li, li]
// 测试数组的方法
arrLis.push('a');   // 未报错
console.log(arrLis);    // (5) [li, li, li, li, 'a']
```

构造函数方法：Array.from()

- `Array.from()`方法可以将类数组或者可遍历的对象转换为真正的数组。
- `Array.from()`可以接受第二个函数参数来对数组进行加工。

```
// 模仿伪数组
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    'length': 3
}
let arr = Array.from(arrayLike);    // ['a', 'b', 'c']
console.log(arr);
// 可以传入函数对数组进行加工
let arrayLike2 = {
    '0': 1,
    '1': 2,
    '2': 3,
    'length': 3
}
let arr2 = Array.from(arrayLike2, item => item * 2);    // [1, 4, 6]
console.log(arr2);
```

实例方法：find()

- 用于查找第一个符合条件的数组成员，如果没有找到返回 undefined。

```
let arr = [{
    id: 1,
    name: '张三'
}, {
    id: 2,
    name: '李四'
}];
let target = arr.find((item, index) => item.id == 2);
console.log(target);    // {id: 2, name: '李四'}
```

实例方法：findIndex()

- 用于找出第一个符合条件的数组成员的位置，如果没有找到返回-1。

```
let arr = [1, 5, 10, 15];
let index = arr.findIndex((vlaue, index) => vlaue > 9);
console.log(index); // 2
```

实例方法：includes()

- 表示某个数组是否包含给定的值，返回布尔值。

```
[1, 2, 3].includes(2); // true
[1, 2, 3].includes(4); // false
```

String 的扩展方法

模板字符串

- 模板字符串可以**解析变量**。
- 模板字符串可以**换行**。
- 模板字符串可以**调用函数**。

```
let name = 'brokyz';
let sayHello = `hello, my name is ${name}`; // hello, my name is brokyz
// 模板字符串可以换行
let sayHi = () => 'Hi!';
let html = `
    <div>
        <span>${sayHi()}</span>
    </div>
`;
console.log(html);
```

实例方法：startsWith() 和 endsWith()

- `startsWith()` 表示参数字符串是否在原字符产的头部，返回布尔值。
- `endsWith()` 表示参数字符串是否在原字符串的尾部，返回布尔值。

```
let str = 'Hello world!';
str.startsWith('Hello') // true
str.startsWith('!') // true
```

实例方法：repeat()

- `repeat()` 方法表示将原字符串重复n次，返回一个新字符串。

```
'x'.repeat(3)   // 'xxx'
'hello'.repeat(2)   // 'hellohello'
```

Set 数据结构

- ES6 提供了新的数据解构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。
- Set本身是一个构造函数，用来生成Set数据解构。

```
const s = new Set();
```

- Set 函数可以接受一个数组作为参数，用来初始化。

```
const set = new Set([1, 2, 3, 4, 4]);
console.log(set);   // Set(4) {1, 2, 3, 4}
console.log(set.size)
```

数组去重

```
const set = new Set([1, 2, 3, 4, 4]);
const arr = [...set];
console.log(arr);   // [1, 2, 3, 4]
```

实例方法

- `add(value)`：添加某个值，返回 Set 解构本身。
- `delete(value)`：删除某个值，返回一个布尔值，表示删除成功。
- has(value)：返回一个布尔值，表示该值是否为 Set 成员。
- `clear()`：清除所有成员，没有返回值。

```
const set = new Set();
s.add(1).add(2).add(3); // 向set结构中添加值
s.delete(2);    // 删除set结构中的2值
s.has(1);   // 表示set结构中是否有1这个值，返回布尔值
s.clear()   // 清除所有成员，没有返回值
```

遍历

- Set 结构的实例与数组一样，也拥有 forEach 方法，用于对每个成员执行某种操作，没有返回值。

```
s.forEach(value => console.log(value));
```

```
const set = new Set(['a', 'b', 'c']);
set.forEach(value => {
    console.log(value);
})
```