JS基础

- prompt()：函数获取用户输入的内容，返回值类型总是一个字符串
- console.time()：开始计时器
- console.timeEnd()：结束计时器
- confirm()：用来弹出一个确认框 确认返回true 取消返回false

输出

- alert()：窗口弹出一个警告框
- console.log()：控制台输出一个日志
- document.write()：网页中输出一个内容

编写位置

- 直接将js代码编写在script标签中
- 将js代码编写在外部的js文件中，然后通过script标签进行引入，script标签要么用来引入，要么用来写代码

```
<script src="app.js"></script>
```

- 将js代码编写在标签的指定属性中

```
<button onmouseenter="alert('你点我干嘛！');">点我一下</button>
```

- 将js代码编写在href属性后边的，以javascript: 开头

```
<a href="javascript:alert('hello');">我是一个超链接</a>
```

变量/常量

```
// 声明和赋值同时进行
let age = 33;
//const 用来声明一个常量，常量只能进行一次赋值无法修改
const b = 44;
```

标识符

- 在程序中所有的可以自主命名的内容都可以认为是标识符（比如：变量名、函数名、类名...）
- 标识命名规范：
  - 标识符中可以含有字母、数字、_、$，但是不能以数字开头
  - 标识符不能是JS中的关键字和保留字，也不建议使用js中内置函数名和变量名
    - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Lexical_grammar

- 标识符应该要遵循驼峰命名法

基本数据类型（不可变）

- 字符串（string）字符串
  - \ 用于转义
  - 模板字符串（老版本的浏览器不支持）
    - 使用反单引号 ` （键盘左上角）来创建模板字符串
    - 特点：
      - 可以换行，并保留字符串中的格式
      - 在模板字符串中可以直接嵌入

```
let name = '石天凯';
str = `我喜欢${name}`;
```

- 数值（number）
  - 所有数值包含整数和浮点数（小数）都属于number类型
  - 特殊数字：
    - Infinity ： 正无穷
    - NaN：Not a Number 非法数字
    - bigint ： 大整数

```
let num10 = 100n;
```

- 布尔值（boolean）
- null
  - 表示一个空的对象，一个不存在的东西
  - null类型只有一个值，就是 null
  - 使用typeof检查一个null时，返回 'object'
- undefined
  - 表示未定义，当我们定义一个变量但是不赋值时它就是undefined
    - undefined类型的值只有一个，就是undefined
  - 使用typeof检查一个undefined时，返回'undefined'

判断JS所有数据类型

- typeof：返回的是数据类型的小写字符串形式 typeof 变量名 / typeof(变量名)
  - 数字——number
  - 字符串——string
  - 布尔——boolean
  - undefined——undefined
  - null——object

    - 数组——object
    - 函数——function
    - 对象——object
- instanceof：检查一个对象的原型链上是否有某个类存在 A instanceof B
    - 数组——Array arr instanceof Array
    - 对象——Object
- ===：判断null和undefined
    - null和undefined都是数据类型，且只有一个值

类型转换

转换为string

- 显式
    - 调用toString()方法（不适用于 null 和 undefined，会报错）

```
a = a.toString();
```

    - 调用String()函数（null和undefined直接转换为 'null'和'undefined'）

```
a = String(a);
```

- 隐式
    - 任意值加上一个空串

```
a = a + '';//任何值和字符串做加法时都会转换为字符串，再拼串
```

转换为number

- 显式
    - 调用Number()函数
        - 字符串
            - 字符串是合法的数字——对应数字
            - 字符串不是合法的数字——NaN
            - 字符串是空串或空格串——0
        - 布尔值
            - true——1
            - false——0
        - null——0
        - undefined——NaN
    - parseInt()： 将一个字符串解析为一个整数
    - parseFloat()：将一个字符串解析为小数
- 隐式

- 任意值 -0，*1等方式（字符串+0不可以）

```
a = a-0;//除了字符串的加法，其余类型的值进行算术运算时，会转换为数值再运算
```

- 一元的+任意值

```
a = +a;//非数值类型的值进行正负运算时，将其转换为数值再运算
```

转换为boolean

- 显式
  - 调用Boolean()函数
    - 0、NaN、null、undefined、false、''——false
- 隐式
  - 任意值取两次反

```
a = !!a;//非布尔值会转换为布尔值然后取反
```

进制转换

```
object.toString(radix)              //object转换对象(number型)，radix要转换为的进制
parseInt(object,radix)              //解析一个字符串并返回指定基数的十进制整数，radix是接收的进制数

//十进制转其他进制
var x=110;
alert(x.toString(2));
alert(x.toString(8));
alert(x.toString(16));
//其他转十进制
var x='110';
alert(parseInt(x,2));  //6     =>以2进制解析110
alert(parseInt(x,8));  //72     =>以8进制解析110
alert(parseInt(x,16)); //272     =>以16进制解析110
//其他转其他
//先用parseInt转成十进制再用toString转到目标进制
alert(parseInt('ff',16).toString(2));
```

对象数据类型转基本数据类型

- 计算、比较、判等——全部转基本
  - 判等的时候，判等如果都是对象，判地址，如果有一个不是对象类型，转基本

```
function fn(){
    console.log(111)
}
//数组转基本  去掉中括号 中间留下什么 就带引号      [1,2,3]   '1,2,3'
console.log([1,2,3] + 100)  // '1,2,3100'
//对象转基本  固定的 '[object  Object]'
console.log({name:'zly'} + 100) //'[object  Object]100'
//函数转基本  固定的 函数整体加字符串
console.log(fn + 100)//  'function fn(){console.log(111)}100'
```

运算符

- 什么是运算符 什么是表达式
  - 参与运算的符号是运算符
  - 由变量或者常量和运算符组成的式子就是表达式 表达式都是有值的

算术运算符

- +
  - 任何值和字符串做加法时都会转换为字符串，再拼串
  - 除了字符串的加法，其余类型的值进行算术运算时，会转换为数值再运算
  - NaN除了和字符串相加，和任何值做任何运算结果都是NaN
- -
- *
- /
- ** 幂
- % 模，两个数相除取余数

一元运算符

- +正：不会对数值产生任何影响
- -负：对数值进行符号位取反
  - 非数值类型的值进行正负运算时，将其转换为数值再运算

自增自减运算符

- 自增
  - a++(后++)
    - a++会使变量立即自增1，console.log(a++)是变量自增前的值（原值）
  - ++a(前++)
    - ++a会使变量立即自增1，console.log(++a)是变量自增后的值（新值）
- 自减
  - a--(后--)
    - a--会使变量立即自减1，console.log(a--)是变量自减前的值（原值）
  - --a(前--)

        - --a会使变量立即自减1，console.log(--a)是变量自减后的值（新值）

赋值运算符

- =
- +=：a += x 等价于 a = a + x
- -=
- *=
- /=
- **=
- %=

逻辑运算符

- !（逻辑非）
    - 非布尔值会转换为布尔值然后取反
- &&（逻辑与）
    - 找false，有false就返回false
    - 如果第一个值是false，则不看第二个值
    - 非布尔值运算时，将其转换为布尔值，然后运算，最终返回原值
- ||（逻辑或）
    - 找true，有true就返回true
    - 如果第一个值是true，则不看第二个值
    - 非布尔值运算时，将其转换为布尔值，然后运算，最终返回原值

关系运算符

- 比较两个值之间大小等于的关系——关系成立，返回true，否则返回false
    - 非数值类型的值比较时，转换为数值再比较
    - 字符串时，逐位比较字符串的字符编码

相等运算符

- ==：相等
    - 对值进行自动的类型转换（Number）
    - null和undefined做相等比较时，返回true
- ===：全等
    - 不会自动类型转换，类型不同时直接返回false
    - null和undefined做全等比较时，返回false
- !=：不相等
    - 对值进行自动的类型转换（Number）
- !==：不全等
    - 不会自动类型转换，类型不同时直接返回true
- NaN不和任何值相等，检查一个值是否是NaN时，函数 isNaN()
- 比较对象是比的是内存地址

条件运算符

- 三元运算符
- 条件表达式？语句1：语句2
  - 如果判断结果为true，则执行语句1
  - 如果判断结果为false，则执行语句2

运算符优先级

- 运算符的优先级根据运算符优先级表来决定
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence
  - 位置越上的优先级越高，优先级越高越优先计算
  - 优先级一样，从左向右的顺序计算
  - 通过()来改变优先级

代码块

- {}为代码进行分组
  - 要么都执行，要么都不执行
  - 代码块中let声明的变量，在代码块外部无法访问
  - 代码块中var声明的变量没有块作用域，在代码块的外部也可以访问

流程控制语句

if-else语句

- ```
  if(条件表达式){
      语句..
  }else if(条件表达式){
         语句..
  }else if(条件表达式){
        语句..
  }else(条件表达式){
       语句..
   }
  ```
- 条件表达式的结果是非布尔值时，转换为布尔值判断

switch语句
  ```

- ```
  switch(条件表达式){
          case 表达式:
          语句..
          break;          //如果没有break会继续执行下面的case
          case 表达式:
          语句..
          break;
          default:
          语句..
          break;
          *}
  ```

- 比较case后的表达式和switch后的表达式是否全等，所有的比较都不成立，自default处开始执行代码
- 全等的条件比较多时，适合用switch

循环语句

- while：先判断后执行

  ```
  // 循环的三个要件：
  //初始化表达式，创建一个变量控制循环的执行
  let i = 0;
  // 条件表达式，设置循环的执行条件
  while(i < 10){
      console.log(i);
      // 更新表达式，修改初始化变量
      i++;
  }
  ```

  ```
  let i = 0;
  while (true){
      console.log(i);
      i++;
      if(i === 5){
          break; // break也可以用来结束循环语句
      }
  }
  ```

- do-while：先执行后判断（确保循环至少一次）

  ```
  let i = 5;
  do{
       console.log(i);
       i++;
    }while (i < 5)
  ```

- for
  ```

```
for(1初始化表达式; 2条件表达式; 4更新表达式){
    3语句...
    }
for(let i=0; i<5; i++){
    console.log(i);
    }
```

  - 先执行初始化表达式，初始化一个变量（只会执行一次）
  - 执行条件表达式，判断循环是否执行
    - 如果为false，则循环结束
    - 如果为true，则执行循环体（语句）
  - 执行更新表达式对变量进行更新
  - 重复执行条件表达式，判断循环是否执行
- 死循环

```
for(;;){
    }
while (true){
    }
```

- break
  - 用来结束switch和循环语句
  - 只会影响到离他最近的循环
- continue
  - 用来跳过当次循环，会继续进行下次循环
  - 只会影响到离他最近的循环

对象（Object）

- 一种复合数据类型，可以存储其他数据，本质是存储数据的容器，存储的数据称为属性
- 创建空对象
  - 常规方式

```
let obj = new Object();       //new可以省略 Object是一个构造函数
```

  - 字面量形式：使用{}来创建一个对象，创建时直接向对象中添加属性
    - 前端中对象属性名本质是字符串，符合命名规则的属性引号可省略，不符合的属性引号必须写

```
{
    属性名:属性值,
    属性名:属性值 //最后一个不加逗号
        }

let obj2 = {
            name:'孙悟空',
            brother:{
                name:'猪八戒'
            }
        }
```

- 添加属性
  - 对象.属性名 = 属性值;

```
obj.name = '孙悟空';
obj.age = 18;
obj.gender = '男';
obj.test = Object();  //对象的属性值可以是任意类型的值，包括对象和函数
```

  - 对象['属性名'] = 属性值;

```
obj['gender'] = '男';
```

- 读取属性：
  - 对象.属性名（没有的属性时，返回undefined）

```
obj.gender;
```

  - 对象['属性名']

```
obj['gender'] = '男';
console.log(obj['gender']);
let a = 'gender';     //用变量存储字符串
console.log(obj[a]);
```

- 删除属性
  - delete 对象.属性名

```
delete obj.gender;
```

  - delete 对象['属性名']
- 改变量 / 改对象
  - 改变量对其他变量不会有影响（赋值运算符、自增自减等）
    - 变量 = xxx
    - 变量 += xxx

        - 变量++
        - 对象 = xxx
    - 改对象的属性，会影响到其他 **指向同一对象** 的变量
        - 对象.属性 = xxx
        - 对象['属性'] = xxx
    - 比较两个对象时，无论是相等还是全等，比较都是对象的内存地址
    - 声明常量是禁止修改变量（变量对应的内存地址），不会影响我们修改对象（内存地址里存的属性值）
- 属性的枚举
    - '属性名' in 对象：检查对象中是否含有某个属性

      ```
      'name' in obj;     //有返回true， 没有返回false
      ```

    - for-in：用来对对象中的属性进行枚举

      ```
      for(let 变量 in 对象){
                      语句...
                  }

      for(let n in obj){   //对象中有几个属性就会执行几次
                    console.log(n,'=',obj[n]);  //n代表属性名 obj[n]代表属性值
                }
      ```

    - hasOwnProperty()： 检查一个属性是否存在于对象自身中（有可能在这个对象的原型中）

      ```
      p1.hasOwnProperty('test');
      p1.__proto__.hasOwnProperty('test');    //有返回true， 没有返回false
      ```

- 使用typeof检查一个对象时，会返回'object'
- 对象分类
    - 内建对象
        - 由ES标准所规定的对象
        - Object Function String Boolean Number Array Math Date JSON...
    - 宿主对象
        - 由JS运行环境所提供的对象 DOM 和 BOM
        - Window Console Document ...
    - 自定义对象
        - 由开发人员自己定义对象

垃圾回收（垃圾收集GC)

- 一个对象，没有任何的变量对其进行引用，这个对象就是垃圾对象，垃圾对象大量存在会占用大量内存空间导致程序速度变慢。
- 这些垃圾对象我们不需要处理，因为在JS中拥有自动的垃圾收集机制，将垃圾对象从内存中清除，我们要将不再使用的变量设置为null。（obj3 = null;）

函数（function）

- 也是一个对象，存储JS代码，在需要时对其进行调用
- 函数和方法最主要的是三要素功能、参数、返回值
- 创建函数
  - 函数声明

```
function 函数名([形参1=xx,形参2=xxx,....形参N]){  //[]里的内容代表可以没有
                    语句...
                }
```

  - 函数表达式

```
let 函数名 = function([形参1,形参2,....形参N]){
                    语句...
                };            //分号结尾
```

- 匿名函数

```
//没有名字的函数，调用不了，需要直接加()调用
//IIFE(Immediately Invoked Function Expression)
//立即执行函数，会在函数创建后立刻调用，并且只会调用一次
(function(){})();
(function(){}());
```

- 调用函数（执行函数中存储的代码）

```
函数名([实参1,实参2,....实参N]);
```

- 参数
  - 形参：相当于在函数中声明了变量，但是没有赋值
  - 实参：调用函数时，实参将会赋值给对应的形参
    - 可以传递任意类型的值作为参数（包括对象和函数）
    - 可以传递任意数量的值作为参数
      - 等于形参数量 --> 一一对应
      - 小于形参数量 --> 没有的就是undefined
      - 大于形参数量 --> 多了的不用
- 返回值
  - 可以是任意类型的值，包括对象和函数
  - 不设置return 或 return 后没有任何值，相当于 return undefined
  - return后，后面所有代码都不会执行，函数立即结束
- 使用typeof检查一个函数时，会返回'function'

This

在调用函数时，浏览器每次都会向函数中传递一个隐含的参数——this（this指向一个对象）

- 以函数的形式调用一个函数时，this是window
- 以方法的形式调用一个函数时，this是调用方法的对象

```
name = '沙和尚';
let obj3 = {
      name:'孙悟空',
      sayHello:function () {
          alert(this.name);//不加this输出沙和尚，因为name指的是一个变量，向上一层找全局变量
      }
  };
obj3.sayHello();      //输出孙悟空，因为name指的是obj3的name属性
```

- 以构造函数的形式调用一个函数时，this是新建的对象/实例

```
function Dog(name, age) {
    this.name = name;
    this.age = age;    //this就是dog
}
let dog = new Dog('啸天', 10);
```

- 通过call和apply调用函数时，第一个参数是谁this就是谁
  - call()
    - 当我们调用call()方法时，实际上和直接调用函数的效果类似。
    - call可以用来指定函数的this，它的第一个参数传谁函数的this就是谁
    - call()传递实参时，直接从第二个参数开始一一传递
  - apply()
    - apply()的作用和call一样
    - apply需要将实参保存到一个类数组对象中同一传递

```
function fn(a, b) {
    console.log(a, b);
}
let obj = {fn};
fn.call('hello');
fn.apply(window);
obj.fn.call(window, 123, 456);
obj.fn.apply(window, [123, 456]);
```

  - bind()
    - 也是一个函数的方法，调用方法时bind会返回一个新的函数对象，这个新的函数对象的功能和原来函数的功能是一样的，不同点在于新函数的this是设置死的

```
let obj = {name:'孙悟空'};
function fn2() {
    console.log(this);
}
let fn3 = fn2.bind(obj);
fn3();
```

- 箭头函数中的this由它外层作用域决定
- 事件的回调函数中，this就是绑定事件的对象

方法（method）

- 一个对象的属性是函数，称这个函数是这个对象的方法，调用这个函数称为调用xx对象的xx方法

```
let obj = {
            test1:function () {
                alert('我是obj中函数1');
            },
            test2(){                //:function可以省略不写
                alert('我是obj中函数2');
            }
        }
obj.test1();      //调用obj对象的test1方法
```

工厂方法创建对象

```
function createPerson(name, age, gender, address){
    let obj = {};
    // 向对象中添加属性
    obj.name = name;
    obj.age = age;
    obj.gender = gender;
    obj.address = address;
    obj.sayHello = function () {
        alert(`大家好，我是${this.name}`);
    };
    //将对象作为返回值返回
    return obj;
}
let per = createPerson('孙悟空', 18, '男', '花果山');
let per2 = createPerson('白骨精', 16, '女', '盘丝洞');
let per3 = createPerson('蜘蛛精', 14, '女', '北七家');
per.sayHello();
per2.sayHello();
per3.sayHello();
```

构造函数（constructor）

- 专门用来创建对象的函数

- 构造函数的定义方式和普通函数没有区别，唯一的不同点在于构造函数的名字首字母要大写
- 构造函数和普通函数的区别在于调用方式
  - 构造函数如果直接调用，它就是一个普通函数
  - 构造函数如果使用new关键字调用，它就是一个构造函数
- 一个构造函数也当做是一个类（class），通过该构造函数所创建的对象称为该类实例，通过同一个类创建的对象，称为同类对象——不是真正的类
- 构造函数的执行流程：
  - 创建一个新的对象
  - 将这个新的对象设置为函数中的this（per/per2/dog）
  - 执行函数中的代码
  - 将新建的对象返回

```
function Person(name, age) {
     this.name = name;
     this.age = age;
 }
function Dog(name, age) {
    this.name = name;
    this.age = age;
}
let per = new Person('孙悟空', 18);
let per2 = new Person('猪八戒', 28);
let dog = new Dog('啸天', 10);
```

- 对象 instanceof 类/构造函数：检查一个对象的原型链上是否有某个类存在

```
per instanceof Person;     //是返回true 否返回false
per instanceof Object;     //Object是所有对象的祖先 一定为true
```

- 包装类
  - JS中，给我们提供了包装类
  - String、Number、Boolean、通过这三个包装类，可以直接创建String类型的、Number类型的和Boolean类型对象
  - 包装类并不是提供给我们创建对象用的，而是JS自己留着用
    - 当我们调用一个基本数据类型的属性或方法时，浏览器会通过包装类临时将其转换为对象，然后调用对象的属性或方法

```
let num = 10;
// 以下这些写法，在实际开发中 绝对不要出现
let num = new Number(10);
let num2 = new Number(10);
let str = new String('Hello');
let bool = new Boolean(true);
```

作用域（scope）

- 变量的作用范围
- 全局作用域
  - 页面加载时创建，页面关闭时销毁
  - 直接写在script标签中的变量/函数，都位于全局作用域中
  - 全局作用域中定义的变量是全局变量，定义的函数是全局函数
    - 全局变量和全局函数可以在任意位置被访问到
    - 全局作用域中存在有一个全局对象（global object） window，window对象代表浏览器窗口
      - 全局作用域中，var声明的变量（全局变量），都会作为window对象的属性保存
        - let声明的变量，不会存储在window对象中
      - 全局作用域中，function声明的函数（全局函数），都会作为window对象的方法保存
- 局部作用域
  - 块作用域
    - 代码块中let声明的变量，在代码块外部无法访问
  - 函数作用域
    - 函数调用时创建，调用结束时销毁
    - 函数每次调用都会产生一个新的函数作用域，每个作用域间相互独立
    - 在函数作用域中声明的变量是局部变量，声明函数是局部函数
      - 只能在函数内部访问，外部无法访问
      - **注意：在函数中声明变量时如果不使用var或let，则变量会变成全局变量**
    - 在函数作用域中可以访问全局作用域的变量
    - 在全局作用域中无法访问函数作用域的变量
    - 变量和函数的提升，在函数作用域中同样适用
- 作用域链
  - 访问一个变量时，JS会先在当前作用域中寻找
    - 如果有，直接使用
    - 如果没有，去当前作用域的上一层作用域寻找
      - 如果有，直接使用
      - 如果没有，去当前作用域的上一层作用域寻找，以此类推……
      - 直到找到全局作用域，如果没有找到则报错 xxx is not defined
    - JS中的作用域叫做词法作用域，函数作用域由它定义的位置来决定的，和调用位置无关。函数的定义在哪，它的上一层就是哪。

变量/函数的提升

- 变量的提升

  使用var声明的变量，会在所有的代码执行前被创建（预解析阶段），但是不会赋值，赋值会在赋值语句执行时才进行，可以在变量声明前就对其进行访问

- 函数的提升

  使用function开头的函数，会在所有代码执行前被创建（预解析阶段），可以在函数声明前就对其进行调用

- 被提升后，代码按顺序执行，不会执行被提升过的代码

- 先提升函数

debug

- 浏览器

  刷新浏览器——Scope——Global中可以看到代码执行情况

双击要监视的对象——右击——add selected text to watches——watch中可以看到代码执行情况

原型（prototype）

- 每个函数对象中都有一个属性叫做prototype（显式原型），该属性指向的是一个对象——原型对象
  - 如果函数作为普通函数调用，则原型对象（prototype）没有任何作用
  - 如果函数作为构造函数调用，那么通过该构造函数所创建的对象（实例）中都会有一个隐含的属性（__proto__）指向函数的原型对象（prototype）
    - 即：实例的隐式原型指向类的显式原型

     image-20220324200925164
  - 原型对象，就相当于一个公共的区域，可以被所有的该类实例共享
    - 可以将一些实例共有的方法/属性存储在原型对象中，这样只需要设置一次，即可让所有的对象（实例）都访问到该方法/属性
- 原型链
  - 当我们访问一个对象中的属性时，JS会先在对象本身中寻找
    - 找到了则直接使用
    - 没有找到则去对象的原型(__proto__)中寻找
      - 找到了，则使用
      - 如果没有找到，则去原型的原型中寻找，以此类推……
      - 直到找到Object的原型，如果依然没有，则返回undefined，不会报错
  - 作用域链——访问变量时生效，原型链——访问对象属性时生效
- Object的原型，是所有对象的原型，它的原型为null
- 结论：

  定义一个类时
  - 如果属性（或方法）是对象独有的，就直接在构造函数中通过this添加

    ```
    function 类名(){
     this.xxx = yyy;
    }
    ```
  - 如果方法（或属性）是公共的，每一个对象的值都是相同，可以通过原型来添加
    - 类.prototype.属性名 = 值;

```
function MyClass() {
        }
// 向原型中添加一个属性
MyClass.prototype.name = '原型中的name属性';
let mc = new MyClass();

// 访问mc中的隐含属性
console.log(mc.name);
```

- 方法的重写：如果原型中的方法不能满足我们的需求，可以选择创建一个同名的方法将其覆盖

```
function MyClass() {
}
let mc = new MyClass();
mc.toString = function(){  //toString是Object原型中的方法
    return 'HELLO';
};
```

- 实例的隐式原型指向类（构造函数）的显示原型

```
function MyClass() {
}
let mc = new MyClass();

MyClass.prototype           //显式原型
mc.__proto__                //隐式原型
mc.__proto__.__proto__      //隐式原型

// 获取Object的原型的原型 Object.prototype得到的是对象 所以后面用__proto__
console.log(Object.prototype.__proto__);
```

数组（Array）

- 数组也是一个对象，用来存储一组有序的数据，数组中存储的数据，称为元素（element）
- 数组元素根据存储的顺序保存在数组之中，数组中的每一个元素都有一个唯一的索引（index），索引就是一组以0开始的整数
    - 使用typeof检查一个数组时，返回 'object'
    - 检查arr是否是一个数组
        - Array.isArray(arr)：静态方法
        - arr instanceof Array：检查一个对象的原型链上是否有某个类存在
- 创建数组对象
    - 创建数组

```
let 数组 = new Array();
let 数组 = [];      //字面量形式：使用[ ]来创建一个数组
```

    - 创建一个指定大小的数组

```
let 数组 = new Array(长度);
```

- 创建数组同时，向数组中添加元素

```
let 数组 = new Array(元素1, 元素2, 元素3, ...元素N);
let 数组 = [元素1, 元素2, 元素3, ...元素N];
```

- 二维数组：一个数组中的元素还是数组

```
let arr = [[1,2,3],[4,5,6]];
```

- 向数组中添加元素： 数组[索引] = 值

```
arr[0] = 10;
arr[arr.length] = 10;      // 数组[数组.length] = 值; 向数组的最后的位置添加元素
```

- 读取数组中的元素：数组[索引]
  - 若数组中没有的元素，不会报错而是返回undefined

```
arr[0];
arr[arr.length-1];      //获取最后一个元素

arr = ['ma', 'qing', 'wei'];
let [a, b, c] = arr;
console.log(a);
```

- 获取数组中元素的数量：length，值是数组的最大索引+1
  - length属性可以被修改
    - 将length改小，则后边的元素会被删除
    - 将length改大，后边会多余空的元素

```
arr.length
```

- 遍历

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

for(let i=arr.length - 1; i>=0; i-=2){
    console.log(arr[i]);      //倒叙取值 每隔两个取一次
}

for(let i in arr){
    console.log(arr[i]);
}
```

```
for(let v of arr){
    console.log(v);      //v是值
}
// for-of 遍历数组、Set、Map、字符串、伪数组
```

forEach遍历

- 专门用来对数组进行遍历
- 该方法需要一个函数作为参数
  - 这种由我们定义，但是不由我们调用的，被称为回调函数（callback）
- forEach()的回调会被调用多次，
  - 数组有几个元素，函数就调用几次
  - forEach()通过回调函数，将元素传递出来
- 遍历到的元素信息将会以参数的形式传递进行回调函数
  - 通过定义形参来获取元素的信息
  - forEach()的回调函数共有三个参数：
    - 当前被遍历的元素
    - 当前遍历的元素的索引
    - 当前正在被遍历的数组对象

```
let arr = ['孙悟空', '沙和尚', '猪八戒', '唐僧'];
arr.forEach(function(element, index, array){
    console.log(index, element);
});

this.attrInfo.attrValueList.forEach((item) => {
  this.$set(item, "flag", false);
});

this.spuImageList.forEach((item) => {
  item.name = item.imgName;
  item.url = item.imgUrl;
});
```

map遍历加工

- 功能： 加工数组，根据已有的数据创建新的数组，新数组当中每一项和老数组当中每一项对应有关系
- 参数： 回调函数（参数：item,index,arr） 每个数组项都会执行这个回调函数，返回的是一个新的项，放在新数组当中
- 返回值： 把每一项都返回的新项组成的新数组返回
- forEach和map（加工返回一个新的数组）区别

```
//map会返回一个数组，改变数组需要重新赋值+return
arrWords = arrWords.map((word)=>{
      if(minWords.indexOf(word)!=-1){
        return '';
      }
```

```
      else{
        return word;
      }
    })
//forEach不会返回一个数组，改变数组需要用arr[index]
arrWords.forEach((word,index)=>{
    if(minWords.indexOf(word)!=-1){
        arrWords[index] = '';
    }
})

this.spu.spuImageList = this.spuImageList.map((item) => {
  return {
    imgName: item.name,
    imgUrl: (item.response && item.response.data) || item.url,
  };
});
```

filter过滤

- 功能：从原数组当中过滤出一个新的数组
- 参数：回调函数（同步的回调）
  - 功能：对遍历的每一项执行回调函数
  - 回调函数的参数：当前项 、当前项的索引、当前遍历的数组
  - 返回值：返回的是一个布尔值，（布尔值，条件表达式），根据这个布尔值的真假来决定当前遍历的这项要不要收集到新数组当中
- 返回值：返回的是新的数组

```
//非箭头函数
let arr = persons.filter(function(item,index){
    return item.name.indexOf(keyword) !== -1  //indexOf()查询元素在字符串中的位置
})
//箭头函数
let arr = persons.filter(item => item.name.indexOf(keyword) !== -1)

this.attrInfo.attrValueList = this.attrInfo.attrValueList.filter(
  (item) => {
    delete item.flag;
    return true;
  }
);
```

reduce统计

- 功能： 统计数组当中的符合条件的结果（数字或者其它类型）
- 参数：
  - 回调函数
    - 参数：prev(上一次统计的结果)、item当前项、index当前项的索引、arr当前遍历的数组
  - 统计的初始值

- 返回值：返回统计后的结果

```
arr.reduce((prev,item) => {
        //这个方法也是暗含遍历，会拿数组的每一项执行回调函数
        //第一次执行回调的时候，prev的值就是你给的初始值
        //第一次执行完回调函数后会返回prev值，返回给了第二次执行时候的初始值
        //最后一次执行返回的prev值因为没有下一次了，这个值直接作为整个reduce的返回值
        if(item.isOver){
          prev += 1
        }
        return prev
      },0)

let result = arr.reduce((a, b) => {
      return a + b;          //数组里面所有的值相加求和（两两相加），若需要别的计算，更改运算符即可
    });
```

every

- forEach map filter some every reduce 都暗含遍历
- 检测数组所有元素是否都符合条件，全部满足返回true，有一个不满足，返回false，剩下的不再检测

```
computed: {
  // 计算出未被选择的销售属性
  unSelectedSaleAttr() {
    //数组的过滤方法，可以从已有的数组当中过滤出用户需要的元素，并未返回一个新的数据
    let result = this.baseSaleAttrList.filter((item) => {
      // every数组的方法，会返回一个布尔值
      return this.spu.spuSaleAttrList.every((item1) => {
        return item.name != item1.saleAttrName;
      });
    });
    return result;
  },
}
```

some

- 只要有一个符合指定条件，返回true，剩下的不再检测，全部不通过返回false

```
let isRepeat = this.attrInfo.attrValueList.some((item) => {
  if (item !== row) {
    return item.valueName == row.valueName;
  }
});
if (isRepeat) {
  this.$message({ type: "info", message: "请输入不重复的数据" });
  return;
}
```

去重

```
const res = [...new Set(arr)]; // 去重
```

删除

```
arr.pop();        //删除并返回数组的最后一个元素
arr.shift();      //删除并返回数组的第一个元素
```

添加

```
arr.push('玉兔精', '蝎子精'); //向数组的最后添加一个或多个元素，并返回新的长度
arr = [...arr, '玉兔精'];     //另一种写法
arr.unshift('二郎神');        //向数组的前边添加一个或多个元素，并返回新的长度
```

slice截取

- 非破坏性的方法
- 截取数组
- 参数：
  - 截取的起始位置索引 (包含起始位置)
  - 截取的结束位置索引 (不包含结束位置)
- 该方法不会影响到原来的数组，将结果作为返回值返回
- 可以省略第二个参数，如果不写则一直截取到最后
- 索引可以传负值，-1表示倒数第一个，-2表示倒数第二个.....

```
result = arr.slice(1, 3);
```

```
let arr2 = ['孙悟空', '猪八戒', '沙和尚'];
// 对arr2进行浅复制（拷贝），只复制对象本身
let arr3 = arr2.slice(0);      //修改其中一个不会影响到其他

let arr3 = arr2;         //修改其中一个会影响到其他
```

splice删除/替换/添加

- 破坏性的方法
- 删除，替换，添加数组中的元素
- 参数：
  - 删除元素起始位置索引
  - 删除的数量
  - 可以传递多个参数，这些参数将会作为新元素添加到数组中
  - 参数只有两个：代表删除 第一个起始位置，第二个删几个
  - 参数如果是多个：增/改 第二个参数是0代表增 如果不是0代表改

- 该方法会对原数组产生影响
- 返回值：被删除的元素

```
arr = ['孙悟空', '猪八戒', "沙和尚", '唐僧'];
result = arr.splice(0, 2);  //删除指定的元素
result = arr.splice(0, 2, '牛魔王', '铁扇'); //替换指定元素
result = arr.splice(0, 2, '牛魔王', '铁扇', '小红');  //替换指定元素
result = arr.splice(2, 0, '牛魔王', '二郎神');  //向指定位置插入新元素
```

冒泡排序

```
let arr = [9,1,3,4,5,7,8,6,2];
for(let j=0; j<arr.length-1; j++){
    // 遍历数组，获取到所有的元素
    for(let i=0; i<arr.length - 1 - j; i++){
        //当前数 arr[i]   后边数 arr[i+1]
        if(arr[i] > arr[i+1]){
            // 前边的数字大于后边的数字，需要交换两个元素的位置
            let temp = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = temp;
        }
    }
}
```

concat连接

- 非破坏性的方法
- 用来连接两个个或多个数组
- 不会影响原数组，将元素拼接到一个新数组中返回

```
let result = arr.concat(arr2, ['牛魔王', '二郎神'], '哈哈');
```

indexOf查询位置

- 非破坏性的方法
- 查询元素在数组中第一次出现的位置
- 参数：
  - 要查询的元素
  - 查询的起始位置
- 返回值：
  - 返回元素第一次在数组中出现的索引，找不到返回-1

```
result = arr.indexOf('孙悟空');
```

lastIndexOf查询位置

- 非破坏性的方法
- 查询元素在数组中最后一次出现的位置
- 参数：
  - 要查询的元素
  - 查询的起始位置
- 返回值：
  - 返回元素最后一次出现的索引，找不到返回-1

```
result = arr.lastIndexOf('孙悟空');
```

join连接

- 非破坏性的方法
- 将数组中的所有元素连接为一个字符串
- 参数：
  - 指定一个连接符（字符串）作为参数
  - 如果不指定，则默认使用“,”

```
arr = ['a', 'b', 'c', 'd'];
result = arr.join('@-@');
```

reverse反转

- 破坏性的方法
- 用来对数组进行反转

```
arr.reverse();
```

sort排序

- 破坏性的方法
- 用来对数组进行排序
- 使用sort()对元素进行排序的时候，比较的是元素的字符编码而不是数字的大小
- 可以在sort()通过回调函数来指定排序规则

```
//升序
arr.sort(function(a, b){
    return a - b;
});
//简化
arr = arr.sort();
//降序
arr.sort(function(a, b){
    return b - a;
});
//简化
```

```
arr = arr.sort().reverse();
//乱序:
function (a, b) {
    return Math.random() - Math.random();
}
```

arguments（伪数组）

- 除了this外，函数中还有一个隐含的参数叫做arguments， arguments 是一个类数组对象（伪数组），类数组对象和数组的操作方式基本一致，只是不能调用数组的方法
- 在函数执行时，所有的实参都会存储在arguments对象中，通过arguments，即使不定义形参也可以使用实参

```
// 创建一个函数，可以求任意个数字的和
function sum() {
    // 创建一个变量，存储结果
    let result = 0;
    // 遍历arguments
    for(let i=0; i<arguments.length; i++){
        result += arguments[i];
    }
    return result;
}
```

- ...args（剩余参数）
  - 剩余参数会获取到所有的没有形参对应的实参
  - 和arguments的区别:
    - 剩余参数就是一个数组，可以调用数组的方法
    - args只会保存没有形参对应的参数
    - 剩余参数必须是参数列表中的最后一个

```
function fn(a, b, ...args) {
    console.log(args);
}
fn(10, 20, 30, 40, 50);
```

递归

- 核心就是对问题的分解，将一个大问题拆分一个一个的小问题
- 函数自己调用自己，它的作用和循环基本上一致
- 递归两个要点:
  - 基线条件，递归停止条件
  - 递归条件，如何对问题进行拆分

```
function jieCheng(num) {
    // 1.基线条件，递归停止的条件
    if(num === 1){
        return 1;
    }
    // 2.递归条件，如何对问题进行拆分
    // 6! = 5! * 6
    // num! = (num - 1) * num
    return num * jieCheng(num-1);
}
```

```
/*一对兔子，年龄超过两个月后，每个月生一对兔子
创建一个函数，求n个月后有多少对兔子
创建一个函数计算斐波那契数列：1 1 2 3 5 8 13 21 34 55 ...
*/
/*创建函数，用来获取斐波那契数列的第 num 个数字*/
function fib(num) {
    // 基线条件
    if(num < 3){
        return 1;
    }
    //num-1  num-2
    return fib(num-1) + fib(num-2);
}
```

```
//快速排序
let nums = [5,1,3,4,9,7,8,6,2];
/*定义一个函数用来对一个数组进行快速排序
*/
function quickSort(arr) {
    // 设置基线条件，什么时候不需要排序
    if(arr.length < 2){
        return arr;
    }
    // arr是要排序的数组
    // 获取一个基准值 随机获取一个index
    let basisIndex = Math.floor(Math.random()*arr.length);
    let basis = arr[basisIndex];
    // 创建两个数组
    let left = []; // 放左值
    let right = []; // 放右值
    //将数组中的值和基准值进行比较
    for(let i=0; i<arr.length; i++){
        if(i === basisIndex){
            continue;
        }
        if(arr[i] < basis){
            // 将小于基准值的值，放入到left中
            left.push(arr[i]);
        }else{
            // 将大于或等于基准值的值，放入到right中
            right.push(arr[i]);
```

```
        }
    }
    return quickSort(left).concat(basis, quickSort(right));
}
console.log(quickSort(nums));
```

Math

- Math不是一个构造函数，不能用来创建对象
- Math中包含了一些数学相关的常量或方法
- Math我们是一种 工具类，不是构造函数
- 属性：
  - Math.PI 表示圆周率
- （类/静态）方法：
  - Math.abs()：用来求一个数的绝对值
  - Math.ceil()：用来对一个数进行向上取整
  - Math.floor()：用来对一个数进行向下取整
  - Math.round()：用来对一个数进行四舍五入取整
  - Math.max()、Math.min()：获取多个值中的较大或较小值
  - Math.pow(x, y)：求x的y次幂
  - Math.sqrt()：对一个数进行开方
  - Math.random()：用来生成一个 0-1之间的随机数
    - 生成一个 0-x之间的随机数：Math.round(Math.random() * x)
    - 生成一个 x-y之间的随机数：Math.round(Math.random() * (y-x))+x
- eval() 函数会将传入的字符串当做 JavaScript 代码进行执行，如果传入的字符串是表达式则返回表达式求值结果，否则返回 undefined

```
eval("400+5")
```

Date

- JS中所有的日期相关的数据都通过Date来表示
- 创建Date对象

```
// 创建一个表示当前日期的对象
let d = new Date();
// 创建一个表示指定日期的对象
// 日期字符串的格式 '月/日/四位年 时:分:秒'
let d = new Date('12/30/2019 14:33:58');
```

- 对象的方法
  - d.getDay()： 获取当前日期对象是周几

    - 返回值是 0-6：0表示周日、1表示周一

```
let weekName = ['周日','周一','周二','周三','周四','周五','周六']
let day = d.getDay()
console.log(weekName[day])
```

  - d.getDate()：获取当前日
  - d.getMonth()：获取当前是几月
    - 返回值 0-11：0 表示1月、1表示 2月
  - d.getFullYear()：获取四位年份
  - d.getTime()：获取当前日期对象的时间戳，自1970年1月1日0时0分0秒，到现在时间所经历的毫秒
    - 在计算机底层所有的时间都是以时间戳的形式存储
  - Date.now()（类方法）：获取当前的时间戳

字符串

- 字符串本质上就是一个字符数组
  - 数组的破坏性方法字符串都没有，因为字符串类型不可变

    可以通过str.split()方法将字符串拆成数组，调用数组的方法，再通过数组的join()方法拼接成字符串

```
let str = 'hello hello';
let result = str.split('');
result.reverse();
str = result.join('');
//简写
let str = 'hello hello';
str = str.split('').reverse().join('');
```

- 属性：
  - str.length：获取字符串的长度
- 方法：
  - str.concat()：用来检查两个或多个字符串连接为一个字符串
  - str.charAt()：根据索引获取字符串中的字符串
  - str.charCodeAt()：根据索引获取指定字符的字符编码
  - String.fromCharCode()（类方法）：根据编码返回字符
  - str.indexOf()：查询子串在字符串中第一次出现的位置（同数组）
  - str.lastIndexOf()：查询子串在字符串中最后一次出现的位置（同数组）
  - str.slice()：用来从一个字符串中截取一个子串（同数组）
  - str.split()：用来将一个字符串拆分为一个数组
    - 参数：需要一个分隔符作为参数，根据分隔符将字符串拆分成数组
    - str.split(正则)：根据正则表达式来将一个字符串拆分位一个（g不起作用）
  - str.toLowerCase()：将字符串转换为小写
  - str.toUpperCase()：将字符串转换为大写

- str.trim()：去除字符串的前后空格
- str.trimEnd()：去除字符串后的空格
- str.trimStart()：去除字符串前的空格
- str.endsWith()：检查一个字符串是否以指定内容结尾
- str.startsWith()：检查一个字符串是否以指定内容开头
- str.search(正则)：根据正则表达式搜索字符串是否含有指定的内容（g不起作用）
  - 返回值：返回字符串第一次出现位置的索引，如果没有找到则返回-1
- str.replace()：使用一个新的内容，替换字符串中指定内容
  - 参数：
    - 正则表达式
    - 新的内容（支持回调函数）
  - 使用replace()进行替换时，默认只会替换第一个符合条件的内容，如果希望替换所有的符合条件的内容，可以使用匹配模式 g
  - 匹配模式：
    - i：忽略大小写
    - g：全局匹配

```
result = str.replace(/1[3-9][0-9]{9}/g,'哈哈哈');
```

- str.match(正则)：将字符串中符合正则表达式的内容提取出来
  - 参数：正则表达式
  - 返回值：将符合条件的内容保存到一个数组中返回

解构赋值

- 将数组中元素赋值给变量
  - let [变量1,变量2,...变量n,] = 数组;

```
let arr = [1, 2, 3, 4, 5, 6, 7];
let [a, b, ...c] = arr;     //把1给a 2给b 剩下的全部给c
[arr[1], arr[2]] = [arr[2], arr[1]];    //实现替换
```

- 将数组中的元素拆分作为参数传递
  - fn(...数组);

```
function sum(a, b, c) {
    return a + b + c;
}
let arr2 = [567, 765, 345];
console.log(sum(...arr2));
```

- 对对象的解构
  - let {变量1, 变量2} = 对象;

```
let obj = {name:'孙悟空', age:18, gender:'男'};
let {name,age} = obj;       //变量名与对象中的属性名一样
```

- ...运算符
  - 扩展运算符，用于打包和拆包，要么是数组要么是对象
  - 数组的打包和拆包

```
//拆包 数组可以直接拆包
let arr = [1,2,3,4]
console.log(...arr)         ----1 2 3 4
console.log([1,2,3,4,...arr])          ----[1,2,3,4,1,2,3,4]
//打包 只在函数形参当中会出现（为了传递不定参数），只有这一种情况是打包，并且打包只能打包数组
function add(a,b,...arr){
    console.log(arr)        ----[3,4,5,6,7,8,9,10]
    console.log(arr instanceof Array)         ----true
}
add(1,2,3,4,5,6,7,8,9,10)
```

  - 对象只能拆包，而且不能直接拆包

```
//拆包对象只能是在一个新的对象当中去拆老的对象
let obj = {
 name:'zly',
 age:33
}
console.log({...obj})     ----{name:'zly',age:33}
```

箭头函数

- 语法：
  - ([参数列表]) => 返回值
  - 参数 => 返回值
  - ([参数列表]) => { 语句... }
  - 注意：
    - 箭头函数中的this在函数创建时确定，它由外层函数（作用域）中的this来决定，外层的this是谁，它的this就是谁

```
function sum(a, b){
    return a+b;
}
//转换为箭头函数
let sum = (a, b) => a+b;

let arr = ['孙悟空', '猪八戒', '沙和尚'];
arr.forEach(element=>console.log(element));    //只有个参数

let arr2 = [3,4,1,2,5];
```

```
arr2.sort((a,b)=>a-b);       //数据升序排列

let sum2 = (a,b) => {        //函数体中多行给代码
    console.log('hello');
    console.log('你好');
    return a+b;
};

let fn = ()=>({name:'孙悟空'});     //没有参数写个空()，返回一个对象时，{}外加上加上()
```

```
let fn2 = ()=>alert(this);
let obj1 = {
    fn2:fn2,
    test:function () {
        function fn(){
            alert(this);
        }
        fn();
    }
};
obj1.fn2();  //fn2的this是window
obj1.test(); //test的this是window，因为这个this是在fn中打印，由fn的调用方式决定，fn是函数形式调用
//将test函数转换为箭头函数
let obj2 = {
    fn2:fn2,
    test:function () {
        let fn = ()=> alert(this);
        fn();
    }
};
obj2.fn2();  //fn2的this是window
obj2.test(); //test的this是obj，因为fn在test方法中定义，箭头函数中的this由外层函数中的this来决定
```

闭包

- 闭包就是能访问到外部函数中变量的内部函数
- 用途：主要用来藏起一些不希望被别人看到的东西
- 构成要件：
    - 闭包必须有函数的嵌套
    - 内部函数必须要访问外部函数的变量
    - 必须将内部函数作为返回值返回
- 生命周期：
    - 当外部函数调用时，闭包便产生了，外部函数每调用一次就会产生一个闭包
    - 当内部函数被垃圾回收时，闭包销毁

```
function outer(){
    // 定义一个变量，记录函数执行的次数
    let times = 0;
    // 创建一个函数，函数每次调用时，都会打印函数被调用的次数
```

```
    function inner() {
        times++;
        alert(times);
    }
    // 将inner设置为函数的返回值
    return inner;
}
let fn = outer();
fn = null;

//用匿名函数(function(){})();简化
let fn4 = (function () {
    let times = 0;
    return function () {
        alert(++times);
    };
})();
```

DOM

DOM（Document Object Model）

- 文档对象模型
  - 文档（Document）：文档指整个网页
  - 对象（Object）：DOM将网页中的所有的东西都转换为了对象
  - 模型（Model）：模型用来体现节点之间的关系
- 网页由三个部分组成：
  - 结构（HTML）
  - 表现（CSS）
  - 行为（JavaScript）
- DOM中为我们提供了大量的对象，使我们可以通过JS来完成对网页操控
- 节点（Node）
  - 网页中的所有部分都可以称为使一个节点，虽然都是节点，但是有着不同的类型
    - 文档节点：表示整个网页
    - 元素节点：各种标签都属于元素节点
    - 属性节点：标签中的属性称为属性节点
    - 文本节点：标签中的文字
- 在浏览器中，document已经为我们提供好了，可以直接使用，document是整个DOM树中最顶级的对象，通过document可以获取到其他的任意对象。
  - document——初始包含块——html——body
  - 初始包含快：不是元素，和浏览器首屏宽高一致的块，不是节点，提供设置宽高的参照物

事件概述

- 事件（event）：就是用户和网页的交互瞬间，例如：点击鼠标，点击按钮，鼠标移动，键盘按下 ...
  - https://developer.mozilla.org/en-US/docs/Web/Events

    - 鼠标事件（一般绑定给document）
        - mouseenter：鼠标移入
        - mouseleave：鼠标移出
        - mousedown：鼠标按下
        - mousemove：鼠标移动
        - mouseup：鼠标松开
        - contextmenu：鼠标右键菜单的事件
            - 点击右键弹出菜单，是contextmenu事件的默认行为，通过取消它的默认行为，来禁用菜单
        - wheel：鼠标滚轮的事件，获取滚轮滚动的方向
            - event.deltaY：滚轮的垂直滚动方向（大于0向下，小于0向上）
            - event.deltaX：滚轮的水平滚动方向
    - 键盘事件（键盘事件只能绑定给可以获取焦点的元素或document）
        - keydown：按下任意键
            - event.key：获取当前是哪个按键被按下
            - event.ctrlKey：用来检查ctrl是否按下，如果按下了 返回true，否则返回false
            - event.shiftKey
            - event.altKey
        - keyup：除了shift、fn、CapsLock以外的任意键
        - keypress：释放任意键
- 我们通过为事件设置响应函数来完成和用户的交互
    - 在元素中设置事件的响应

```
<button onmouseenter="alert('哈哈! ');" id="btn">我是一个按钮</button>
<button id="btn">我是一个按钮</button>
```

    - 在script标签中，通过js代码来设置事件的响应

```
let btn = document.getElementById('btn');
// 可以通过为指定对象的事件属性设置响应函数的形式来处理事件
// 在事件发生时触发的函数称为事件的响应函数
btn.onclick = function () {
    alert('Hello');
};
```

- 在事件的响应函数中，this就是绑定事件的对象

```
let checkedAllBox = document.getElementById('checkedAllBox');
checkedAllBox.onclick = function () {
    for(let i=0; i<items.length; i++){
        items[i].checked = this.checked;   //this是checkedAllBox
    }
};
```

文档的加载

- 网页的加载是按照自上向下的顺序一行一行加载的，如果将script标签写在前边，这样js执行时页面还没加载完毕，会出现获取不到DOM对象的情况。
- 解决方式
  - 将js代码写在body的最下方
  - 将js代码编写到window.onload的回调函数

```
<script>
    window.onload = function(){
        let btn = document.getElementById('btn');
        btn.onclick = function () {
            alert('嘻嘻! ');
        };
    };
</script>
```

  - 如果是外部文件，可以在引入时在script标签添加defer属性使其延迟加载

```
<script defer src="app.js"></script>
```

DOM查询

- 查询指在网页中获取指定的节点
- 通过document对象查询
  - document.getElementById()：根据id获取一个元素节点对象
  - document.getElementsByClassName()：根据class属性值获取一组元素
  - document.getElementsByTagName()：根据标签名来获取一组元素节点对象
    - document.getElementsByTagName('*')：获取页面中的所有元素获取页面中的所有元素
  - document.getElementsByName()：根据name属性值获取一组元素节点对象（表单）
  - document.querySelector()：根据选择器字符串获取符合条件的第一个元素
    - 选择器的字符串作为参数，根据选择器去页面中获取元素，只会返回符合条件的第一个元素

```
let div = document.querySelector('div[title]');  //有title的div，[]：属性选择器
```

  - document.querySelectorAll()：根据选择器字符串获取符合条件的所有元素，不论一个还是多个，都返回列表

```
let items = document.querySelectorAll('.item');
```

    - getElementsByTagName和querySelectorAll区别

```
let allLi = document.getElementsByTagName('li');//可以实时的更新其中元素
let allLi2 = document.querySelectorAll('li');//不会实时更新，获取几个就是几个
```

  - document.documentElement：获取页面的根元素 （html）

    - document.body：获取页面的body元素
- 通过element进行查询
    - element.getElementsByTagName()：根据标签名获取元素中的指定的后代元素
    - element.childNodes：获取当前元素的所有子节点（包含空白文本）
    - element.children：获取当前元素的所有子元素
    - element.firstChild：获取第一个子节点
    - element.firstElementChild：获取第一个子元素
    - element.lastChild：获取最后一个子节点
    - element.lastElementChild：获取最后一个子元素
    - element.parentNode：获取当前元素的父元素
    - element.previousSibling：获取前一个兄弟节点
    - element.previousElementSibling：获取前一个兄弟元素
    - element.nextSibling：获取后一个兄弟节点
    - element.nextElementSibling：获取后一个兄弟元素
- 元素中的属性：
    - 读取元素的属性：元素.属性名
        - element.name
        - element.value
        - element.id
        - element.className
        - element.classList[x]：对象所有class的列表，可以通过索引来获取具体的class
    - 设置元素的属性：元素.属性名 = 属性值
        - element.name = xx
        - element.value = xxx
        - element.id = xxx
        - element.className = xx
        - element.classList：当修改的样式过多时，操作元素类，修改元素的class属性
            - add()：向元素添加一个或多个类 box1.classList.add('b2');
            - remove()： 移除元素中的一个类 box1.classList.remove('b1');
            - replace()： 使用一个新的class替换原有class box1.classList.replace('b1', 'b2');
            - toggle()： 切换一个元素的class box1.classList.toggle('b2');
                - 如果元素拥有该类，则删除
                - 如果元素没有该类，则添加
            - contains()：检查一个元素是否含有某个class box1.classList.contains('b2') - 如果包含，返回true，否则返回false
- 其他属性
    - innerHTML：内部的HTML代码，带标签（自结束标签没有）
    - innerText：内部的文本内容，不带标签
- 读取一个标签内部的文本：
    - span.innerHTML
    - span.innerText
    - span.firstChild.nodeValue
    - span.textContent：只支持高级浏览器

DOM增删改

- 创建元素
  - document.createElement(标签名)：创建一个新的元素
  - document.createTextNode(文本内容)：创建一个新的文本节点（一般直接用innerHTML）
- 插入元素
  - 父节点.appendChild(子节点)：向父节点中插入一个子节点（位置在父节点最后）
    - 元素.insertAdjacentElement('位置', 元素)：向元素的指定位置插入子元素
    - 元素.insertAdjacentHTML('位置', 'HTML代码')：向元素的指定位置插入HTML代码
    - 元素.insertAdjacentText('位置', '文本内容')：向元素的指定位置插入文本内容

      ```
      - 位置需要传递一个字符串作为参数：
            - 'beforebegin' ：开始标签前，成为当前元素的前一个兄弟元素
            - 'afterbegin' ：开始标签后，成为当前元素的第一个子元素
            - 'beforeend' ：结束标签前，成为当前元素的最后一个子元素
            - 'afterend' ：结束标签后，成为当前元素的后一个兄弟元素
      ```

    - 父节点.replaceChild(新节点, |旧节点)：使用新节点替换|旧节点
    - 父节点.insertBefore(新节点, |旧节点)：将新节点插入到|旧节点的前边
    - 添加元素时，可以直接通过修改元素innerHTML属性来达到目的，但是修改innerHTML时，是直接为innerHTML进行重新赋值，这样将会导致其他元素被替换，所以使用innerHTML一定要慎重

      ```
      let list = document.getElementById('list');   // 获取list
      list.innerHTML += '<li>白骨精</li>';
      ```

- 删除元素
  - 父节点.removeChild(子节点)：删除一个子节点
    - 子节点.parentNode.removeChild(子节点)
  - 子节点.remove()：删除当前节点，老版本的浏览器不支持
- 复制节点
  - 节点.cloneNode()：对节点进行浅复制（只复制节点本身）
  - 节点.cloneNode(true)：对节点进行深复制（复制节点本身及所有的后代节点）
    - 复制完记得修改id等信息

默认行为

- 默认行为指当事件触发时元素默认会做的事情
  - 比如：点击超链接后页面会发生跳转，点击表单的提交按钮后页面发生跳转 ...
- 有时默认行为会影响到正常功能，需要将其取消，只需要在事件的响应函数中return false即可取消

```
link.onclick = function(){
            ...
            return false;     //只适用于这种绑定方式
        };
```

```
baidu.addEventListener('click', function (event) {
    // 取消默认行为
    event.preventDefault();
    alert(123);
});
baidu.onclick = function (event) {
    event.preventDefault();
    alert(123);
};
```

操作CSS

- 操作内联样式
  - 属性：style
  - 读取样式：元素.style.样式名
  - 设置样式：元素.style.样式名 = 样式值
  - 注意：
    - 通过style属性所读取和设置的样式都是内联样式
    - 所以通过它所设置的样式通常会立即生效
    - 如果样式名不符合标识符的规范，需要对样式名就行修改：
      - 去掉-，-后的字母大写
      - background-color ==> backgroundColor
      - border-left-width ==> borderLeftWidth
- 获取当前的生效的样式
  - getComputedStyle()
    - 参数：
      - 要获取样式的元素
      - 要获取的伪类（没有可以不写）
    - 返回值：一个对象，对象中包含了当前元素所有生效的样式
    - 注意：该方法获取的样式全都是只读的，无法修改
- 其他的样式相关的属性：
  - clientWidth、clientHeight：获取内容区和内边距的总大小
  - offsetWidth、offsetHeight：获取内容区、内边距和边框的总大小
  - offsetParent
    - 获取当前元素的定位父元素
    - 离当前元素最近的开启了定位的祖先元素，如果所有的祖先都没有开启定位则返回body
  - offsetLeft、offsetTop：当前元素距离其定位父元素的距离

- scrollHeight、scrollWidth：获取元素滚动区域的大小

  注意：

  - 以上属性都是只读属性，无法修改
  - 以上属性所获取的值都是不带单位的值，可以直接参与运算
- scrollTop、scrollLeft：获取（设置）垂直和水平滚动条滚动的距离
- 判断滚动条滚动到底：
  - 垂直：scrollHeight - scrollTop === clientHeight
  - 水平：scrollWidth - scrollLeft === clientWidth
- 禁用系统滚动条

```
html,body{
  height: 100%;
  overflow: hidden;
}
```

事件

事件对象

- 当事件的回调函数被调用时，浏览器每次都会传递一个对象作为参数（实参），这个对象就是事件对象。
- 是这一次事件触发后相关的所有一切信息被封装成的一对象
- 浏览器调用回调要传递事件对象，是为了防止用户在函数内部用到这次事件相关的信息
- 事件对象中存储了事件相关的一切信息：
  - 事件触发时，哪个鼠标按键被按下、哪个键盘上的按键被按下、鼠标滚轮滚动的方向..、
- 要获取事件对象，只需在事件的回调函数中定义一个形参即可

```
document.onmousemove = function (event) {   //onmousemove类事件一般绑定给document
    ...
    };
document.onmousemove = null; //取消事件
```

事件的冒泡（bubble）

- 冒泡指事件的向上传导，子元素上事件触发时，会同时导致其祖先元素上的同类事件也被触发
- 冒泡的存在简化了代码的编写
- 冒泡的发生只和html结构有关，和元素的位置无关
- 但是有时我们不希望冒泡的存在，可以通过事件对象来取消冒泡：
  - 通过cancelBubble属性来取消冒泡：event.cancelBubble = true;
  - 通过stopPropagation()方法来取消冒泡：event.stopPropagation();

事件的绑定

- addEventListener()：为元素设置响应函数
  - 参数：

        - 要绑定的事件，需要一个事件的字符串作为参数（不要on）
        - 事件的回调函数
        - 是否在捕获阶段触发事件，需要一个布尔值
            - true：会发生事件的捕获
            - false：不会（默认值）
- 通过该方式所绑定的事件不会互相干扰，可以为同一个事件绑定多个响应函数，事件触发时，函数会按照绑定的顺序执行
    - removeEventListener()：移除一个事件的响应函数 - 移除时的参数必须和设置时的一模一样

```
function clickHandler(event) {
        alert(1);
    }
btn01.addEventListener('click', clickHandler);
btn01.removeEventListener('click',clickHandler);
```

事件的传播

- 事件的捕获
    - 事件从最外层元素（window）向目标元素进行事件的捕获
    - 默认情况下，捕获阶段不会触发事件
    - 如果希望在捕获时触发事件，可以将addEventListener()的第三个参数设置为true
- 目标元素（触发事件的元素）
    - 事件捕获到目标元素，捕获停止
- 事件的冒泡
    - 从目标元素开始，向外层元素进行事件的冒泡
    - 默认情况下，事件是在冒泡阶段触发的

事件的委派（事件的委托）

- 当需要为多个元素绑定相同的响应函数时，可以统一将事件绑定给它们共同的祖先元素，只需要绑定一次即可让所有的元素都具有该事件，即使元素是新增的也会具有该事件。
- 在事件对象中有一个属性叫做target，它表示的是触发事件的对象

```
// 将事件绑定给ul
ul.addEventListener('click',function (event) {
    // 判断触发事件的对象是否是超链接
    // 在事件对象中有一个属性叫做target，它表示的是触发事件的对象
    // 判断事件是否由超链接触发
    if(event.target.tagName.toUpperCase() === 'A'){
        event.preventDefault();
        alert('我是ul上的单击事件! ');
    }
});
```

定时调用

- setTimeout()（延时调用）

    - 指定时间后调用函数
    - 参数：
        - 回调函数，要调用的函数
        - 时间（毫秒）
- clearTimeout()
    - 关闭定时器（延时调用）

```
setTimeout(function test() {
    num++;
    h1.innerHTML = num;
    if(num === 10){
        return;
    }
    setTimeout(test, 1000);
}, 1000);
```

- setInterval()（定时调用）
    - 每隔指定时间调用一次
    - 参数：
        - 回调函数，要调用的函数
        - 时间（毫秒）
    - 返回值：
        - 返回一个定时器的id
- clearInterval()
    - 关闭定时器
    - 参数：
        - 定时器的id

```
let timer = setInterval(function () {
    num++;
    h1.innerHTML = num;
    if(num === 10){
        clearInterval(timer);
    }
}, 1000);
```

BOM（浏览器对象模型）

- BOM中为我们提供了一组对象，用来完成对浏览器的各种操作
- BOM对象：
    - Window：浏览器窗口
    - History：代表的是浏览器的历史记录
        - 由于隐私的原因，History无法访问具体的历史记录，只能用来控制浏览器向前向后翻页
        - history.length：当前访问的页面的数量

        - history.forward()：切换到前边访问的网址
        - history.back()：相当于浏览器的回退按钮
        - history.go()：跳转到指定的历史记录
    - Location：代表的是浏览器的地址栏
        - 直接读取location，则可以获取到地址栏的信息
        - 修改location的值，则浏览器会自动跳转到新的地址，通过这种方式跳转页面，会留下历史记录，可以通过回退按钮回退
        - history.assign()：用来跳转地址，和直接修改location是一样的
        - history.replace()：跳转地址，它不会产生历史记录，无法通过回退按钮回退
        - history.reload()：重新加载网页，相当于网页的刷新按钮
    - Navigator：代表浏览器的信息
        - navigator.userAgent：返回的是一个字符串，用来表示浏览器的信息
    - Screen：代表的是设备屏幕信息
- BOM对象都是window对象的属性，所以可以直接访问

正则表达式

- 定义一个字符串的规则
- 创建一个正则表达式对象
    - new RegExp('正则表达式', '匹配模式');
    - 字面量 语法：/正则/匹配模式
- test() 用来检查一个字符串是否符合正则表达式

```
new RegExp('ab', 'i');    //i：忽略大小写
RegExp.test('Abc');
/ab/i.test('Abc');
```

- 量词
    - {m}： 正好出现m次
    - {m,n}： 出现m到n次
    - {m,}： 至少出现m次
    - +：至少1次，等价于 {1,}
    - ?：0-1次，等价于 {0,1}
    - *：0-多次，等价于 {0,}
- 规则
    - |： 或
    - []： 中的内容都表示或 只表示一个
    - [a-z]： 任意的小写字母
    - [A-Z]： 任意的大写字母
    - [a-zA-Z]： 任意字母
    - [0-9]： 任意数字
    - [^]： 除了
    - ^： 字符串的开头

- $：字符串的结尾
  - 如果一个正则表达式以^ 开头，以$结尾，则要求字符串和正则表达式要完全匹配
- .：任意字符
- \.：表示 .，（正则表达式中使用 \ 作为转义字符）
- \w：任意单词字符，相当于[A-Za-z0-9_]
- *\W：除了单词字符，相当于[^A-Za-z0-9]*
- \d：任意数字，相当于[0-9]
- \D：除了数字，相当于[^0-9]
- \s：空格
- \S：除了空格

```
//手机正则
let phoneReg = /^1[3-9][0-9]{9}$/;
//邮箱正则
let emailReg = /^\w+(\.\w+)*@[a-z0-9-]+(\.[a-z]{2,5}){!,2}$/i
```