# **注意事项**

* 1. 所有start参数都包含本身，所有end参数都不包含本身
  2. NaN === NaN 返回false，Object.is（NaN，NaN）返回true
  3. 伪数组：具有length属性、按索引的方式进行存储、没有真正数组的一些方法（pop、push等）
     1. 伪数组的length属性在调用时会重新计算，因此若把它作为循环判据，会多次计算影响效率，可将其保存为变量
  4. 阻止链接跳转：在链接的地址处填javascript：void（0）； 或 javascript：；
  5. js代码中可不加分号，在一行开头为括号或方括号时必须在行首加上分号。在js文件开头加分号可以防止合并压缩报错
  6. 正则表达式的（.\*？）防止一个属性从第一个双引号的左引号匹配到最后一个双引号的右引号
  7. ECMAScript标准的缺陷：没有模块系统、标准库较少、没有标准接口、缺乏管理系统
  8. 使用扩展运算符出现对象的属性冲突时，以后面的为主
  9. 对超出数组最大长度或对象中不存在的属性进行读取时，返回undefined

1. new T1(r => r()).then(v => v) instanceof T1 // true
2. new T2(r => r()).then(v => v) instanceof T2 // false
   1. 较高不确定性的随机数：window.crypto.getRandomValues（）
   2. 若有多个全局执行上下文，使用instanceof进行类型检测时可能有问题

# **综述**

* 1. 浏览器包含渲染引擎和JS引擎，前者解析HTML和CSS，也称内核，后者处理JS代码。
  2. JS由ECMAScript（JavaScript语法）、DOM（页面文档对象模型）、BOM（浏览器对象模型）组成

## JavaScript执行机制

* + 1. js是单线程的
    2. 处理机制：
       1. 同步异步：同步即前一个任务结束后再执行下一个任务。异步即可以同时进行多个任务
       2. 同步任务放入主线程执行栈、异步任务（回调函数）放入任务队列（消息队列）
       3. 执行机制：
          1. 先执行执行栈中的同步任务、当异步任务有运行结果后，将事件（回调函数）放入任务队列、执行栈中的同步任务执行完毕后读取任务队列中的事件进入执行栈，开始执行
          2. 事件绑定的回调函数在事件触发后才写入任务队列、定时器在时间到后才写入任务队列
       4. 事件循环（事件轮询）：主线程不断地重复获得异步任务（事件）、执行异步任务（事件）、再获取（事件）任务、再执行
    3. 进程与线程：
       1. JS是单线程运行的，但使用H5中的Web Workers可以多线程运行。浏览器都是多线程的。
       2. firefox和老IE是单进程的，chrome和新IE是多进程的。
       3. 定时器的回调函数是在主线程执行的，alert会暂停当前主线程的执行，并暂停计时
    4. 浏览器内核：支撑浏览器运行的最核心的程序。内核由很多模块组成：
       1. 主线程模块：js引擎模块；html，css文档解析模块；DOM/CSS模块；布局和渲染模块；。。。
       2. 分线程模块：定时器模块；事件响应模块；网络请求模块
  1. js引入html：
     1. ①内嵌式：直接将JS代码放入<script></script>标签内。
     2. ②外部式【推荐】：<script src=”JS文件路径”></script>
     3. ③行内式：写在元素内部，例如<input type=”button” onclick=”JS语句”>
     4. ④动态加载：通过DOM创建script标签，此方式创建的script元素相当于添加了async属性，考虑兼容性，可将async属性设置为false
     5. 动态加载的资源对浏览器预加载器不可见，可以通过link标签预加载
     6. script元素可写在head标签中 或 写在body标签的最后（推荐）

## 字符编码

* + 1. 字符串使用两种Unicode编码混合的策略：UCS-2和UTF-16。
    2. 一个码元由16个比特【2字节】组成；码点就是对应字符的编码，码点值等于Unicode编码值【十六进制】
    3. UCS-2：适用于Unicode编码在U+0000~U+FFFF的字符，一个码点对应一个码元
    4. UTF-16：对于Unicode编码超过U+FFFF的字符，采用该策略。是UCS-2的父集，对于UCS-2不能表示的字符，每个字符
       1. 会用另一个码元去选择一个增补平面，这种每个字符使用两个16位码元的策略称为代理对。

## 严格模式

* + 1. 在脚本开头加上 “use strict”； 开启严格模式，或在函数开头加上该指令，使函数在严格模式下执行，在严格模式下：
       1. 使用未声明变量时报错；delete未声明变量时抛出异常；用0作为前缀表示八进制字面量时抛出异常
       2. LHS引用时，若在全局作用域也找不到变量，会创建一个全局变量（非严格模式）/ 抛出异常（严格模式）
       3. 函数内部的this不允许指向window，会置为undefined。但定时器回调函数的this及在全局作用域下直接使用this不受影响
       4. 不允许使用with语句
       5. 对函数的限制：函数不能以eval或arguments作为函数名或形参名；两个命名参数不能拥有同一个名称
       6. eval内部创建的变量和函数无法被外部访问
       7. 只设置了获取函数/设置函数的访问器属性，尝试修改/读取属性会抛出异常

# 基本

* 1. 注释：单行注释：// XXXXX 多行注释：/\* XXXXXX \*/
  2. 输入输出语句：
     1. alert（msg）； 浏览器弹出警示框
     2. confirm（‘msg’） 弹出询问框，返回布尔
     3. console.log（msg）； 浏览器控制台打印输出信息
     4. console.dir（对象）； 打印对象，更好地查看属性和方法
     5. console.time（[str]）；console.timeEnd（[str]） 计算器，timeEnd（）方法会将两个方法间花费的时间打印在控制台，str需一致
     6. prompt（info）； 浏览器弹出输入框
  3. 标识符：变量、函数、属性、函数参数的名称。字母、数字、下划线、$组成；不能以数字开头或使用关键字。建议用小驼峰

# 数据类型

* 1. JS是弱类型语言，无需进行类型声明，可在任何阶段改变变量的数据类型
  2. 简单（原始）类型：String、Number、Boolean、null、undefined、Symbol、BigInt
  3. 复杂（引用）类型：Object（Object、Function、Array）
     1. 所有对象都是Object类型，Function是可执行的Object，Array是属性为数值下标，且内部数据有序的Object
  4. 数据、变量与内存：
     1. 数据为存储在内存中代表特定信息的东西，本质为0101，一切皆数据，内存中所有操作的目标为数据，可读可传递可运算
     2. 内存是可临时存储数据的空间，有栈空间（存储全局和局部变量）和堆空间（存储对象）；
     3. 变量是内存的标识，由变量名和变量值组成。变量名用于查找对应内存、变量值为指定内存中保存的数据。
  5. **Undefined**：代表未初始化的变量。该类型只有undefined一个值。声明而不初始化变量时，变量值为undefined
  6. **Null**：表示一个空对象指针，该类型只有null一个值。建议使用null对保存对象类型的变量进行初始化

Boolean

* + 1. 布尔型，值有true、false
    2. 转换为布尔型：Boolean（值），if等流程控制语句会自动执行此转换，转换规则：
       1. 转换为true：非空字符串、非零数值（包括无穷值）、任意对象（空对象、空数组）
       2. 转换为false：空字符串、0和NaN、null、undefined

## Number

* + 1. 数值型，包含整型和浮点型
    2. 字面量的进制表示（es6）：二进制：0b开头，八进制：0o开头【或0开头，不建议】，十六进制：0x开头
    3. 科学计数法表示：系数e幂数。如3.12e7 === 31200000。
       1. 内部存储时，默认按普通方式存储，仅当小数点后紧随至少6个零时，会自动转换为科学计数法存储
    4. Infinity和-Infinity：无限值，表示超过[Number.MIN\_VALUE, Number.MAX\_VALUE]的数
       1. isFinite（）判断数字是否在可表示范围内，返回布尔值
       2. 返回Infinity或-Infinity【详见操作符】：非0值除以0或-0
    5. NaN：非数字，表示本来要返回数值的操作失败了
       1. isNaN（） 判断变量是否为非数字，返回布尔值。会转换变量为数值后进行判断，若无法转换为数值，返回true。
          1. 若参数为对象，调用valueOf（）判断，若不能转换为数值，调用toString（）判断
       2. 返回NaN【详见操作符】：任何与NaN的运算操作
    6. 转换为数值型：
       1. 实现：
          1. Number（值） 或 +”123”【正运算符，返回值与Number一致】
          2. parseInt（string [, 2/8/16]）【指示string的基数】
          3. parseFloat（string [, 2/8/16]）
          4. 除+外算术运算符隐式转换（‘12’ - 0）（’123’ - ’120’）
       2. 区别：
          1. parseInt（）和parseFloat（）遇到不能解析的内容时会返回前面已经解析好的内容
          2. Number需要转换的内容不能转换为数字时返回NaN
       3. parseInt（）和parseFloat（）：
          1. 跳到第一个非空格字符开始解析，若第一位就无法解析则返回NaN
          2. 遇到数值型时，会转换为string后再解析，如果数值被自动转换为科学表示法表示，会出错
          3. 可以识别string的不同进制并转换为十进制整数/浮点数，如“0xA”
       4. Number（）转换规则：
          1. 数值直接返回；true和false分别返回1和0；null，返回0；undefined，返回NaN；
          2. 如果是字符串，遵循下列规则：

如果只包含数字（包括前面带正号或负号的情况），则将其转换为十进制数值

如果是有效的浮点格式，如“1.1”，则将其转换为对应的浮点数值

如果是有效的二/八/十六进制整数格式，例如"0xf"，则将其转换为相同大小的十进制整数值

如果字符串是空的，则将其转换为0

如果字符串中包含除上述格式之外的字符，则将其转换成NaN

* + - * 1. 如果是对象，调用valueOf()，按规则转换，若结果不是NaN，返回。否则调用toString()，再按照规则转换，返回结果。

## String

* + 1. 字符串型，16位Unicode字符序列。可用单引号、双引号、反引号标识。length属性获取字符个数，字符字面量算作一个字符。
    2. 字符串是不可变的，若要修改某字符串的值，会先销毁原字符串，再将包含新值的字符串保存到该变量
    3. 字符字面量：\n【换行】、\t【制表符】、\b【退格】、\r【回车】、\f【换页】、\\、\’、\”、\`、\xnn【十六进制编码nn表示的字符】、

\unnnn【十六进制编码nnnn表示的Unicode字符】

* + 1. 模板字符串（ES6）：用反引号` `标识的字符串，如`xxxxx$｛xxx｝xxxx`
       1. 模板字符串会保留字面量的换行符和空格，可以定义跨行字符串
       2. 字符串插值：通过$｛JavaScript表达式｝实现，表达式的值会被调用toString转换为字符串并插入指定位置
       3. 标签函数：通过 函数名`xxx$｛表达式1｝123$｛表达式2｝404` 的方式调用函数
          1. 函数接收到的参数依次为：由插值记号分隔的模板组成的数组、第一个表达式的值、第二个表达式的值。。。
       4. 原始字符串：原始的字符串字面量内容，而非转义后的内容
          1. String.raw`模板字符串` ：返回原始字符串
          2. 字符串数组的.raw属性：返回原始字符串数组
       5. 转换为字符串：
          1. 加号拼接字符串
          2. String（值）转换规则：

若值有toString（）方法，调用该方法并返回结果；若值为null，返回“null”；若值为undefined，返回“undefined”

* + - * 1. .toString（[基数r=10]） 除了null和undefined，所有值都有该方法

注意：

整数字面量不加括号直接调用toString，点会被识别为小数点，报错。解决方法：（123）.toString（）

正负数字面量需要加上括号后调用toString，否则会先执行toString，然后运算正负号，返回数值型

返回值：

布尔值或者 booleanObject 对象 返回"true" 或 "false"

对象Object 返回“[object Object]”

自定义函数 返回源代码，内置函数返回“function xxx() { [native code] }”

数组Array 返回值同arr.join（），即每个值之间由逗号拼接的字符串

时间Date 返回当前时区的时间的字符串表示。"Sun Jun 05 2016 10:04:53 GMT+0800 (中国标准时间)"

正则表达式RegExp 返回正则表达式字面量的字符串表示。'/ab/i'

## Symbol

* + 1. 符号，ES6新增的原始数据类型。符号实例唯一、不可变，不能与其他数据运算，用来创建唯一记号，用作非字符串形式的对象属性
    2. 创建：
       1. let s = Symbol（[‘描述字符串’]）； 创建符号实例，即使描述字符串相同，值也不同
       2. let s = Symbol.for（[‘描述字符串’]） 返回符号实例，以描述字符串为键，在全局符号注册表中创建并重用符号实例
          1. 若注册表中不存在指定键值对，创建符号实例作为值添加到注册表，返回符号实例；若已存在，返回相应的符号实例
          2. 即使描述字符串相等，注册表中定义的符号实例跟用Symbol（）定义的符号实例也不等同
    3. 使用符号作为属性：任何可以使用字符串或数值作为属性的地方，都可以使用符号。
       1. 使用：
          1. 方法一：let o =｛age：18，[符号实例]：22｝
          2. 方法二：o[符号实例]=22
          3. 方法三：Object.defineProperty（o，符号实例，｛value：22｝）【defineProperties同理】
       2. 注意：
          1. Object.getOwnPropertyNames（obj） 返回对象实例的常规属性数组
          2. Object.getOwnPropertySymbols（obj） 返回对象实例的符号属性数组
          3. Object.getOwnPropertyDescriptors（obj） 返回同时包含常规和符号属性描述符的对象
          4. Reflect.ownKeys（obj） 返回同时包含常规和符号属性描述符的数组
    4. 常用内置符号：用于暴露语言内部行为，这些内置符号以全局函数Symbol的字符串属性存在，值为符号
       1. Symbol.asyncIterator 用作对象的键，值为方法，该方法返回对象默认的AsyncIterator，由for-await-of使用。
       2. Symbol.iterator 对象，方法，该方法返回对象默认的迭代器。
       3. Symbol.hasInstance 对象，方法，该方法在对象被用于instanceof运算符时执行，方法的参数为instance
       4. Symbol.toPrimitive 对象，方法，当对象转换为对应的原始值时，会将该方法的返回值作为原始值
          1. 方法接收参数hint【表示要转换成的原始值的类型，取值有“number”、“string”或“default”】
       5. Symbol.toStringTag 对象，方法，Object.prototype.toString（）会将该方法的返回值作为标识【即 [object 标识]】
       6. Symbol.isConcatSpeadable 对象，布尔值，表示对象在被concat用作参数来合并时，选择是否展开
          1. 对于数组对象，默认展开【true或真值】；对于类数组对象，默认不展开【false或假值】
          2. 类数组对象若设置为展开，会根据其length属性【没有则不添加】，添加其键名为0、1、2...的值【无该键则值为undefined】
       7. Symbol.unscopables 对象, 配置对象, 将该对象的指定属性【配置对象中设置同名属性, 值为true】从with环境绑定中移除
       8. Symbol.match 正则对象，方法，String.prototype.match（）会调用该方法对正则表达式求值
          1. 方法接收参数target【调用match（）方法的字符串实例】，方法的返回值无限制
       9. Symbol.replace String.prototype.replace（）。接收参数target【同上】、replacement【替换字符串】，无限制
       10. Symbol.search String.prototype.search（）。接收参数target【同上】，无限制
       11. Symbol.split String.prototype.split（）。接收参数target【同上】，无限制
       12. Symbol.species 对象，方法，创建派生对象时，会调用该方法，将返回值作为其构造函数
           1. 用getter的方式定义为静态属性【static get [Symbol.species]｛｝】，方法返回值为构造函数名【默认相当于返回this】
           2. 会创建派生对象的方法：Array.prototype.map（）、.filter（）.concat（）
    5. 静态方法：
       1. .keyFor（symbol实例） 在全局符号注册表中查询symbol实例的字符串键，若查询不到，返回undefined

BigInt

* + 1. 大整数，ES11引入的第八种原始数据类型
    2. 可以用来表示任意大整数，与Number值相等时，==返回true，===返回false
    3. 创建：new BigInt（）【构造函数】 或 123456789n【整数字面量后面加n】
    4. 注意：可以使用 + - \* / \*\* % 以及除>>>外的位运算符，不能使用单目“+”运算符
  1. 隐式转换

# 操作符

* + 1. 一元操作符：
       1. 递增/递减操作符：++ -- 操作符可用作前缀/后缀，表示先递增减再返回右值/先返回临时变量【左值】再对原值递增减
          1. 运算符可用作任意值，规则如下：

字符串，若能转换为数值，设变量值为数值再操作；若不能转换为数值，设变量值为NaN

布尔值，若为false/true，转换为数值0/1再操作

对象，调用valueOf（）获取值，再应用上述规则，若得到NaN，再调用toString（）并再次应用上述规则

* + - 1. 一元加和减：+ - 用于操作数前，相当于正负号。若变量为非数值，先执行Number（）转换再操作
    1. 位操作符：
       1. 数值的底层存储：
          1. 所有数值都以IEEE754的64位格式存储，但位操作符先把值转换为32位整数【导致NaN和Infinity在位操作中被当成0处理】，执行完位操作后再转换为64位
          2. 有符号整数使用32位的前31位【从右往左数】表示整数值，第32位表示数值的符号（符号位，0为正，1为负）
          3. 正值以原码表示，负值以二补数【补码】表示。默认情况下，所有整数都表示为有符号数，对于无符号数，第32位表示数值
       2. 对非数值应用位操作时，会先用Number（）转换为数值，再操作
       3. 按位非： ~ 用于数值前，返回数值的一补数【所有位取反，相当于对数值取相反数并减1】
       4. 按位与/或/异或： & | ^ 需要左右两个操作数，对两个数的每一位进行逻辑与/或/异或操作，返回数值
       5. 左移： << 左操作数表示数值，右操作数表示移位位数。将数值的所有位向左移动，用0补充空位
       6. 有/无符号右移： >> >>> 将数值的 除符号位外所有位/所有位 向右移动指定位数，用0补充空位
    2. 布尔操作符：
       1. 逻辑非： ！ 用于操作数前，返回布尔值，为取反后的操作数，若操作数不为布尔类型，会调用Boolean（）执行转换
       2. 逻辑与： && 需要左右两个操作数，返回值如下：
          1. 对左操作数进行Boolean（）转换，若为true，返回右操作数；若为false，返回左操作数【转换前的值】
          2. 短路逻辑：若要返回左操作数，就不会对右操作数进行求值
       3. 逻辑或： || 需要左右两个操作数，对左进行转换，为true，返回左【转换前】；若为false，返回右。短路逻辑同上
    3. 乘性操作符：都为二元操作符，运算结果根据两个操作符符号而定【0的符号也有效】，对非数值的操作符，先进行Number（）转换
       1. \*【乘法操作符】 /【除法操作符】 %【取模操作符】
       2. 0除以0【包括正负0】，返回NaN；无限值乘以0，返回NaN；无限值乘以非零值，返回无限值；
       3. 无限值除以无限值，返回NaN；无限值除以有限值，返回无限值；非零有限值除以0，返回无限值；
       4. 无限值对所有数值取模，返回NaN；有限值对0取模，返回NaN；有限值对无限值取模，返回被除数；
    4. 指数操作符： \*\* 二元操作符，结果与Math.pow（）一样，非数值自动进行Number（）转换
    5. 加性操作符：二元操作符
       1. 加法操作符： +
          1. 不同符号无限值相加，返回NaN；-0加-0返回-0，其余0加0返回+0
          2. 返回数值：操作数没有字符串或对象，会将非数值操作数通过Number（）转换，返回相加后的值
          3. 返回字符串：若至少有一个操作数为字符串或对象，会将非字符串操作数通过String（）进行转换，并返回拼接后的字符串
       2. 减法操作符： - 非数值自动进行Number（）转换
          1. 同符号无限值相减，返回NaN；符号不同的0减0返回-0，相同的返回+0
    6. 关系操作符： < > <= >= 二元操作符，返回布尔值
       1. 对象类型的操作数，先调用valueOf（），若没有该方法，调用toString（）
       2. 若操作数都是字符串，逐个比较字符串中对应字符的编码；其余情况，将非数值通过Number（）转换后比较
       3. 若有操作数为NaN，返回false
    7. 相等操作符： == != === !== 二元操作符，返回布尔值，分别为等于、不等于、全等、不全等
       1. !=和==先转换类型再比较，===和!==需要在不转换类型的前提下相等/不相等时，返回true
          1. 转换规则：

对象 → 字符串 → 数值 ← 布尔

两个操作数类型不同时，按上述路线转换，直到类型相同

对象 → 字符串：调用valueOf（） 字符串/布尔 → 数值：通过Number（）

* + - * 1. 比较规则：

null与undefined相等，且比较前不能转换为其他值

若有一个操作数是NaN，则不相等【NaN不等于NaN】

若两个操作数都是对象，仅当指向同一个对象时相等

待补充

* + 1. 条件操作符： 条件表达式？表达式1：表达式2 当条件表达式为true时，返回表达式1；否则，返回表达式2
    2. 赋值操作符： = \*= /= %= += -= \*\*= <<= >>= >>>=
    3. 逗号操作符：
       1. 用于同时声明多个变量：let num1=1，num2=2，num3=3
       2. 用于辅助赋值：a=（5，2，0） 对它的每个操作数求值（从左到右），并返回最后一个操作数的值
    4. in操作符：
       1. 单独使用时：“属性名” in 对象 当可以通过对象访问到指定属性时，返回true
       2. 在for-in循环中使用：返回可以通过对象访问的非符号可枚举属性
    5. 优先级：
       1. 括号 >【成员访问、带参数的new、函数调用、可选链】> 无参数new > 后置递增减 > 一元操作符 > 指数操作符 > 乘性操作符 > 移位操作符 >【关系操作符、in、instanceof】> 相等操作符 > 按位与 > 按位异或 > 按位或 > 逻辑与 > 逻辑或 > 空值合并 > 条件操作符 > 赋值操作符 > yield > 逗号操作符

# 语句

* 1. if、while、for的条件表达式的值会自动通过Boolean（）转换
  2. if语句：当表达式的值为true时，执行语句1；否则执行语句2【可以是if语句】
     1. if（表达式）语句1或｛语句块1｝ [ else 语句2或｛语句块2｝ ]
  3. do-while语句：首先执行循环体内的代码，然后对条件表达式进行判断
     1. do 语句或｛循环体｝ while（表达式）；
  4. while语句：先检测条件表达式，再执行循环体内的代码
     1. while（表达式） 语句或｛循环体｝
  5. for语句：进入for首先执行初始化表达式，每次循环开始前检测条件表达式，再执行循环体，循环体结束后执行循环后表达式
     1. for（[初始化表达式]；[条件表达式]；[循环后表达式]） 语句或循环体
  6. for-in语句：枚举可以通过对象访问的非符号可枚举属性的属性名【字符串型】
     1. for（变量 in 表达式） 语句或循环体
  7. for-of语句：枚举可迭代对象的元素，按next（）方法产生值的顺序迭代元素
     1. for（变量 of 表达式） 语句或循环体
  8. for-await-of语句：
  9. 标签语句：给语句添加标签。使用方法： 标签：语句
  10. break和continue语句：两个语句都用于退出指定循环【无标签则为当前循环】。之后，break执行循环后的语句，continue回到循环顶部
      1. break [标签]； continue [标签]；
  11. with语句：将代码作用域设定为特定的对象，不推荐使用，严格模式下禁用
      1. with（表达式） 语句或循环体
  12. switch语句：当条件表达式**全等于**case后某表达式的值，执行该case下面的语句。若没有case满足，执行default后的语句。
      1. switch（条件表达式）｛case 表达式1：语句块1 break；case 表达式2：语句块2 break；。。。default：语句块｝

# 变量、作用域与内存

变量

* + 1. 变量声明：
       1. 声明语句：
          1. 声明变量： 声明关键字 变量名
          2. 声明变量并初始化： 声明关键字 变量名=表达式
          3. 同时声明多个变量： 声明关键字 变量名1，变量名2
          4. 同时声明多个变量并初始化： 声明关键字 变量名1=表达式1，变量名2=表达式2
       2. 变量命名不应使用window对象的属性名，如name，top，location，self等
       3. 声明关键字：有var、let和const。后两者为es6新增
          1. var：声明的变量被添加到最接近的函数/全局上下文，存在变量提升
          2. let：不能声明已被声明的变量，块级作用域，不存在变量提升。let在全局作用域中声明的变量不会成为window的属性。

for循环使用var声明迭代变量i，循环体内的异步函数的i保存结束循环后的值；使用let会为每个循环声明i，保存正确值

暂时性死区：在let声明之前的执行瞬间称为暂时性死区。此时调用后面才声明的变量会报错

* + - * 1. const：具有let的特点，外加：声明时必须同时初始化，常量的值不能修改
    1. 预解析【提升】：所有的var和function声明会被提升到当前函数/全局作用域的顶部
       1. var声明并初始化时，初始化代码会被拆分，只提升声明。
    2. 变量可包含两种类型的数据：
       1. 原始值：原始数据类型，存储于栈内存。保存原始值的变量是按值访问的【变量保存实际值】，原始值不能有属性。
       2. 引用值：引用数据类型，存储于堆内存。保存引用值的变量是按引用访问的【变量保存地址值】
    3. 类型判断：
       1. typeof 变量：返回数据类型的字符串表达，小写。
          1. 可能的返回值：“undefined”、“boolean”、“string”、“number”、“object”、“function”、“symbol”
          2. null和array返回object、NaN返回number、函数返回function、未声明变量返回undefined
       2. 实例 instanceof 构造函数名 返回布尔，判断实例的原型链中是否包含指定构造函数的原型
       3. Object.prototype.toString.call（变量） 返回“[object 大驼峰类型名]”，可识别标准类型和内置对象类型，不能识别自定义类型
    4. 左值与右值：
       1. 左值：可以出现在赋值语句左边或右边。特征为可以获取地址
       2. 右值：只能出现在赋值语句的右边，特征为不能获取地址，不能被赋值，临时变量及数字字面量一般为右值
    5. LHS和RHS查询：
       1. LHS（Left-hand Side）引用：通常指等号（赋值操作）的左边的变量，是赋值操作的目标
       2. RHS（Right-hand Side）引用：通常指等号（赋值操作）的右边的变量，仅查找并获取它的值
       3. 根据作用域链查找，若全局作用域中没有值，RHS抛出异常，LHS会自动创建一个全局变量（非严格模式）/ 抛出异常（严格模式）

作用域

* + 1. 执行上下文：简称上下文，上下文【指代整体环境】有时候也称作用域【关注标识符的可访问性】。上下文确定了，作用域也就确定了。
       1. 介绍：
          1. 任何变量都存在于某个上下文中
          2. 每一个上下文都有一个关联的变量对象，在这个上下文中定义的变量和函数都存在于该对象上。
          3. 通过var定义的全局变量和函数会成为window的属性和方法，let和const定义的则不会定义在全局上下文中。
          4. 上下文在其所有代码都执行完毕后会被销毁，全局上下文在应用程序退出前才会被销毁
       2. 全局上下文：是最外层的上下文，变量对象由运行环境确定，浏览器中的全局上下文为window对象。
          1. 在执行全局代码前，将window确定为全局执行上下文，对全局数据进行预处理：

var定义的全局变量进行声明（ES6前，if语句中的也为全局变量），其值为undefined，添加其为window的属性；function声明的全局函数进行函数赋值，添加为window的方法

* + - 1. 函数上下文：每个函数调用都有自己的上下文，进入函数时，上下文被压入上下文栈，函数执行完后弹出。
         1. 对局部数据进行预处理：依次对形参变量、arguments、var定义的局部变量、function声明的函数、this进行赋值，并将他们添加为执行上下文的属性或方法
      2. 块级作用域【es6】：由最近的一对包含花括号｛｝确定【if、while。。。】
      3. 上下文栈：window确定为全局执行上下文对象，将其压入栈中，函数执行上下文对象创建后，将其压入栈中，函数执行完毕后
      4. - 将栈顶对象出栈，所有代码执行完毕后，栈中只剩window
    1. 作用域链：
       1. 介绍：
          1. 上下文代码在执行时，会创建变量对象的作用域链。正在执行的上下文的变量对象始终位于作用域链的最前端。
          2. 若上下文为函数，则其活动对象用作变量对象。活动对象最初只有一个定义变量：arguments【全局上下文中没有这个变量】。
          3. 作用域链中的下一个变量对象来自包含上下文，以此类推至全局上下文。
          4. 全局上下文的变量对象始终为作用域链的最后一个变量对象。
       2. 代码执行时的标识符解析是通过沿作用域链逐级搜索标识符名称完成的。从作用域链最前端逐级往后搜索，直至找到标识符。
       3. 作用域增强：当代码执行到某些语句时，会在作用域链最前端临时添加一个上下文：
          1. try/catch语句的catch块【添加变量对象，包含要抛出的错误对象的声明】、with语句【添加指定对象作为变量对象】

内存

* + 1. 垃圾回收：因浏览器而异，标记清除常用，引用计数不常用
       1. 标记清除：变量声明或进入上下文时，加上存在于上下文的标记；离开上下文时，加上离开上下文的标记。根据标记进行清理
       2. 引用计数：记录每个值被引用的次数，声明变量并赋引用值时，值的引用数为1，当同一个值又被赋给另一变量时引用数加1。
       3. 当引用该值的变量被其他值覆盖，引用数-1，当一个值的引用数为0时，回收其内存。（问题：循环引用）
    2. 内存管理：
       1. 内存生命周期：分配空间→存储数据→释放内存空间
       2. 内存分配：
          1. 基本数据类型：直接放入栈中，存放它的值；引用数据类型：值存入堆中，变量存入栈中，存放其值的地址
       3. 解除引用：当变量不再必要时，将其设置为null，从而释放其引用，减少不必要的内存占用。
       4. 内存释放：函数执行完毕会自动释放空间。局部变量自动释放，该变量所指对象在之后某时刻由垃圾回收器回收
       5. 内存溢出：程序运行所需内存超出了剩余的内存
       6. 内存泄露：占用的内存没有及时释放。常见泄露：意外全局变量、没及时清理的计时器或回调函数、闭包。

# 基本引用类型

* 1. 引用类型是把数据和功能组织到一起的结构，与类相似而非类。对象是特定引用类型的实例

## Date

* + 1. Date为构造函数：let now = new Date（[毫秒数]） 代表从1970年1月1日午夜之后的毫秒数【GMT】
       1. 不指定参数：保存当前系统时间 指定参数：数字型，保存指定时间
       2. 构造函数会根据传入的参数隐式调用parse（）和UTC（）【隐式调用UTC时，按本地时区返回毫秒数】
    2. 静态方法：
       1. .parse（‘日期字符串’） 解析字符串，返回对应的毫秒数【GMT】。解析失败返回NaN
          1. 支持的格式：“月/日/年”【“5/23/2019”】、“月名 日, 年”【“May 23, 2019”】、“周几 月名 日 年 时:分:秒 时区”【“Tue May 23 2019 00:00:00 GMT-0700”】、“YYYY-MM-DDTHH:mm:ss.sssZ”【“2019-05-23T00:00:00”】
       2. .UTC（年,零起点月[,日][,时][,分][,秒][,毫秒]） 返回对应毫秒数【GMT】，参数为数字型，日默认为1，其他默认为0
       3. .now（） 返回当前毫秒数【H5】
    3. 实例方法：
       1. .getFullYear（）【年】、.getMonth（）【零起点月】、.getDate（）【日】、.getDay（）【周几（0为周日，6为周六）】、.getHours（）【时（0~23）】、.getMinutes（）【分钟】、.getSeconds（）【秒】、.getMilliseconds（）【毫秒】
       2. 以上get类方法返回数值型。在“get”后添加“UTC”，可返回UTC日期的指定时间单位。改“get”为“set”，并传递一个数值参数，即为set类方法，用于改变实例的时间【没有setDay和setUTCDay】。set类也可加UTC，来按UTC更改实例时间。
       3. .getTimezoneOffset（） 返回以分钟计的UTC与本地时区的偏移量【本地快，偏移量为负】
       4. .toString（）/.toLocaleString（） 返回带时区的日期和时间/带AM或PM的本地日期和时间【因浏览器而异】
       5. .valueOf（） 返回日期的毫秒表示【数字型】
       6. .getTime（）/ .setTime（毫秒数） 返回值与valueOf（）相同 / 设置时间
    4. 返回时间戳：.valueOf() 或 .getTime() 或 date1 = +new Date（）； 或 Date.now() H5新增

## RegExp

* + 1. RegExp为构造函数：let reg = new RegExp（“模式字符串” [，“标记字符串”]） 或 let reg = /模式/标记【字面量形式】
       1. 通过构造函数模式创建实例，模式字符串中的转义字符在某些时候需要二次转义【\n要写为\\\\n】
       2. 构造函数也可传入（RegExp实例[，“标记字符串”]）
    2. 匹配模式：以下为匹配模式的标记字符，标记字符串由单个或多个标记字符组成。
       1. i【不区分大小写】、g【全局模式，不在匹配到第一个内容时结束】、m【多行模式，查找到一行文本末尾时会继续查找】
       2. u【开启Unicode匹配】、s【dotAll模式，元字符 . 匹配任何字符】
       3. y 粘附模式，将lastIndex开始及之后的字符串作为源字符串，且模式字符串的开头会被隐式加上^
    3. 静态属性：以下所有属性均为非标准属性，谨慎使用。仅exex和test方法有效。
       1. input / $\_【最后搜索的字符串】、lastMatch / $&【最后匹配的文本】、lastparen / $+【最后匹配的捕获组】
       2. leftContext / $`【input属性中出现在lastMatch前的文本】、rightContext【。。。后的文本】
       3. 以上的$为简写形式，尽量通过RegExp[“$+”]的形式访问
       4. $1~$9【分别包含第1-9个捕获组的匹配项】
    4. 实例属性：
       1. global【布尔值，表示是否设置了g标记】、ignoreCase【i】、unicode【u】、sticky【y】、multiline【m】、dotAll【s】
       2. source【字面量形式的模式字符串】、flags【字面量形式的标记字符串】
       3. lastIndex 整数，表示在源字符串中下次搜索的开始位置【仅test和exec方法有效】，不会因源字符串改变而改变
          1. 如果匹配模式不是g或y，该属性会被忽略；每次匹配成功后，该值被设置为紧随最近一次成功匹配的下一个位置的下标
          2. 匹配失败后，该值被设置为0
    5. 实例方法：
       1. exec（‘源字符串’） 若找到匹配项，返回包含第一个匹配信息的数组；没找到返回null
          1. 数组的第一个元素为匹配整个模式的字符串，其余元素为表达式中捕获组匹配的字符串
       2. test（‘源字符串’） 若字符串与模式匹配，返回true，否则返回false
       3. .toString（）/.toLocaleString（） 返回正则表达式字面量的字符串表示
       4. .valueOf（） 返回正则表达式本身【对象型】
    6. 相比Perl语言的欠缺：\A和\Z锚、联合及交叉类、原子组、x（忽略空格）匹配模式、条件式匹配、正则表达式注释

## 原始值包装类型

* + 1. 介绍：
       1. 使用某个原始值的方法或属性时，后台会自动创建一个相应原始包装类型的对象，从而暴露出操作原始值的各种方法。
       2. 以读模式访问原始值时【获取值或调用其方法】，后台会执行：创建一个对应原始包装类型的实例、调用实例方法、销毁实例
       3. 可以通过 new 操作符创建原始值包装类型的实例
       4. Object构造函数为工厂方法，new Object（原始值）会根据传入的值返回相应原始值包装类型的实例
    2. Boolean
       1. 构造函数：let booleanObject = new Boolean（值） 值会自动通过Boolean（）转换后存于实例中
       2. 实例方法：valueOf（）【返回原始值true/false】、toString（）【返回字符串“true”/“false”】
    3. Number
       1. 构造函数：let numberObject = new Number（值） 值会自动通过Number（）转换后存于实例中
       2. 静态属性：
          1. EPSILON【js表示的最小精度】、MAX\_SAFE\_INTEGER / MIN\_SAFE\_INTEGER【最大/最小安全整数】、MAX\_VALUE / MIN\_VALUE【最大/最小正数】、NEGATIVE\_INFINITY / POSITIVE\_INFINITY【负无穷大值/正无穷大值，溢出时返回该数】
       3. 静态方法：
          1. isFinite（数值）/ isNaN（数值） 不同于window对象上的同名方法，这两者不会进行类型转换
          2. isInteger（数值）/ isSafeInteger（数值） 判断是否为整数 / IEEE754数值格式范围内的整数【即[-2\*\*53+1, 2\*\*53-1]】
          3. parseInt（字符串）/ parseFloat（字符串） 同window对象上的同名方法
       4. 实例方法：
          1. valueOf（）【返回原始值】、toLocaleString（）/ toString（[基数]）【返回数值字符串】
          2. toFixed（小数位数） 返回数值字符串，保留指定位数小数【四舍五入】
          3. toExponential（小数位数） 返回以科学计数法表示的数值字符串，保留指定位数小数【四舍五入】
          4. toPrecision（数字总位数【不含指数】） 根据情况返回固定长度或科学计数法形式的数值字符串【四舍五入】

### String

* + - 1. 构造函数：let stringObject = new String（值） 值会自动通过String（）转换后存于实例中
      2. 实例属性：length【表示字符串中码元的数量】
      3. 静态方法：
         1. fromCharCode（多个数值） 每个数值对应一个码元，按顺序拼接为字符串返回
         2. fromCodePoint（多个数值） 每个数值对应一个码点，按顺序拼接为字符串返回
      4. 实例方法：
         1. valueOf（）、toLocaleString（）、toString（） 返回对象的原始字符串值
         2. charAt(索引号) / str[索引号] 返回指定索引位置的码元对应的字符
         3. charCodeAt（索引值） 返回指定索引位置的码元的值的十进制整数表示
         4. codePointAt（索引值） 返回指定索引位置的码元所在码点的值的十进制整数表示

若指定码元不是码点的开头，会返回错误的码点值；可以解构字符串再遍历

* + - * 1. normalize（“NFD”/“NFC”/“NFKD”/“NFKC”） 返回规范化后的字符串，可选四种规范化形式之一

有些Unicode字符具有多个编码，通过采用相同规范化形式将这些编码转换为一致的编码。

* + - * 1. concat（多个字符串） 返回拼接后的字符串【将参数按顺序追加在实例字符串后方】，等效于+
        2. substr(start, 长度) / slice(start, end) / substring（start, end） 返回从指定索引截取的子串

三个方法都可以省略第二个参数，若省略则取到末尾

substr把start负参数转换为字符串长度加start、把长度负参数转换为0；slice转换所有负参数为字符串长度加该值；substring转换所有负参数为0。所有负参数若转换后仍为负，则转换为0

slice第二个参数小于第一个时，返回“”；substring自动取较小的参数为start，较大的为end

* + - * 1. indexOf(‘子串’ [, start]) / .lastIndexOf（...） 返回从前往后/从后往前第一次匹配到子串的位置，若匹配失败，返回-1
        2. startsWith（‘子串’ [, start]）/ endsWith（‘子串’ [, end]）/ includes（‘子串’ [, start]）

返回布尔，表示字符串是否以子串开头/以子串结尾/包含子串。end相当于将字符串末尾定于此处

* + - * 1. trim（）/ trimStart（）/ trimEnd（） 返回删除前后所有/前面所有/后面所有空格符的字符串副本

trimLeft和trimRight方法分别为trimStart和trimEnd方法的别名

* + - * 1. repeat（整数） 将字符串复制指定次，返回拼接所有副本后的字符串
        2. padStart（长度[, ‘填充字符串’]）/ padEnd（...） 返回从前/后复制的指定长度子串。

长度不足时在相应一侧填充字符，默认填充字符串为空格

* + - * 1. toLowerCase（）/ toUpperCase（） 返回按照通用规则转换为小写/大写后的字符串
        2. toLocaleLowerCase（）/ toLocaleUpperCase（） 返回按照各语言规则转换为大写/小写后的字符串【推荐】
        3. match（字符串/正则表达式） 返回值同RegExp的实例方法exec（）
        4. search（字符串/正则表达式） 返回第一个匹配的索引，若没有搜索到返回-1

match（）、replace（）会自动将字符串参数转换为正则对象

* + - * 1. replace（字符串/正则表达式，‘替换字符串’/ fn（）） 返回修改过的字符串，用替换字符串/fn的返回值替换匹配到的子串。

替换字符串中可用特殊的字符序列表示指定值：$$【$】、$&【匹配整个模式的字符串】、$’【匹配的子串之前的字符串】$`【匹配的子串之后的字符串】、$n【匹配第n个捕获组的字符串，n为0-9】、$nn【匹配第nn个捕获组的字符串】

fn参数：与整个模式【一个匹配项】/捕获组【多个匹配项】匹配的字符串、匹配项在字符串中的起始位置、源字符串

* + - * 1. split（字符串/正则表达式，数组长度上限） 返回将字符串按分隔符分隔后的字符串数组
        2. localeCompare（字符串） 按字母表顺序逐位比较，排在后面的字母大【各语言有对应的字母表】

若字符串小于参数字符串，返回负值【通常为-1】；等于返回0；大于返回正值【通常为1】

* + - * 1. HTML方法：辅助生成HTML标签，基本无人使用。如link（url），返回<a href=”url”>string</a>

## 单例内置对象

* + 1. 单例内置对象是任何由ECMAScript实现提供、与宿主环境无关，并在ECMAScript程序开始执行时就存在的对象
    2. Global
       1. 代码不会显式地访问Global，Global对象是一种兜底对象，定义于全局作用域的变量和函数都会成为其属性。如isNaN（）等
       2. 属性：undefined、NaN和Infinity等特殊值；所有原生引用类型构造函数【如Object、Function等】都是Global对象的属性
       3. 方法：
          1. encodeURI（uri）/ decodeURI（...） 返回字符串，编码/解码不属于URL组件【#/?：】的特殊字符
          2. encodeURIComponent（uri）/decodeURIComponent（...） 返回字符串，编码/解码所有非标准字符

一般用encodeURI（）编码整个URI，使用带Component的方法编码将要追加到已有URI后面的字符串

* + - * 1. eval（‘JS代码字符串’） 执行代码，解释器会直接在原地插入代码【无变量提升】

字符串内若有{},它会将{}当成是代码块，若不希望将其当成代码块解析，需要在参数前后加（）

严格模式下，无法在外部访问内部创建的变量和函数；执行性能较差，且具有安全隐患

* + - 1. 访问Global对象：浏览器中window对象为Global对象的代理；或 在任何上下文创建立即调用函数，函数内直接返回this
    1. Math：
       1. 属性：
          1. E、LN10【】、LN2【】、LOG2E【】、LOG10E【】、PI、SQRT1\_2【根号0.5】、SQRT2【根号2】
       2. 方法：
          1. max（多个数值）/ min（...）【最大/最小值】、random（）【返回[0, 1)的随机小数】、abs（数值）【绝对值】
          2. sign（数值）【根据数值符号返回1、0、-0、-1】、hypot（多个数值）【所有数平方和的平方根】
          3. floor（数值）【向下取整】、ceil（...）【向上取整】、round（...）【四舍五入，负数.5往正方向取】
          4. fround（...）【返回数值最接近的单精度（32位）浮点值表示】、trunc（..）【抹去小数部分】
          5. exp（...）【】、log（...）【】、pow（数值x，数值n）【】、sqrt（...）【平方根】、cbrt（...）【立方根】
          6. cos（...）、acos（...）【arccos】、acosh（...）【反双曲余弦】

# 集合引用类型

* 1. Object：详见[对象](#_对象)

## Array

* + 1. 创建：
       1. 数组字面量：const arr = [“xx”, 18, true]【不会调用Array（）构造函数】
       2. 通过构造函数：const arr = [new] Array（数值【生成指定长度的空数组】或多个值【生成包含这些值的数组】）
       3. 通过from方法与of方法：见静态方法
    2. 数组空位【避免使用】：字面量的逗号之间留空表示对应索引处为空位。ES6新增方法将空位当做undefined，ES6之前的方法会忽略空位
    3. arr.length = 5 调整数组长度，若5大于原长度，扩大数组到5，新增元素为undefined。若5小于原长，删去多余元素。
    4. 数组索引：arr[ 8 ] = 99 若数组长度小于9，数组长度会自动扩展到9。否则修改索引号为8的数据；
       1. 若对超过数组长度的索引进行读取，返回undefined
    5. 静态方法：
       1. from（任何可迭代结构或有length属性和可索引元素的结构[, 映射函数][, this]） 将类数组结构转换为数组实例，可用于浅复制
          1. 映射函数：接收元素为参数，对数组内所有元素执行函数，将返回值作为数组实例对应位置的值
          2. this：该参数将作为映射函数中this的值，若映射函数为箭头函数则不适用
       2. of（多个参数） 将一组参数转换为数组实例
       3. isArray（参数） 若参数为数组，返回true，否则false
    6. 实例方法：
       1. valueOf（） 返回数组本身
       2. toString（）/ toLocaleString（） 返回字符串，对数组内所有元素调用toString/toLocaleString后用逗号拼接
       3. join（[‘分隔符’=‘,’]） 返回字符串，对所有元素调用toString后用分隔符拼接【null和undefined转换为“”】
       4. keys（）/ values（）/ entries（） 返回数组索引 / 数组元素 / 索引值对的迭代器，可用from（）转换为数组实例
       5. fill（填充值[，start][，end]） 返回填充后的数组
       6. copyWithin（目标索引值[，start][，end]） 无返回，从数组[start，end)区间的元素复制到目标索引处
          1. fill和copyWithin都支持负索引，忽略反向索引。start和end不写默认开头/结尾。越界部分忽略
       7. push（多个参数） / unshift（...） 在数组末尾 / 头部添加任意个元素，返回添加后的数组长度
       8. pop（） / shift（） 删除数组最后 / 最前一个元素，返回该元素
       9. reverse（） 无返回，翻转数组
       10. sort( [ (a, b) => a - b 【或b - a】] ) 无返回，排序数组
           1. 若不写参数，根据每一项调用String（）的结果字符串来决定顺序，升序排列
           2. 若接收比较函数，如果a应该排在b之前，需要返回负值。
       11. concat（多个参数） 添加参数【自动对数组参数进行解构】到数组副本末尾，返回数组副本
       12. slice（start [, end]） 返回包含原数组指定区间内元素的数组副本
       13. splice（start，删除个数[，多个元素]） 删除数组的元素，然后在删除处添加元素，返回被删除的元素组成的数组
       14. indexOf（元素）/ lastIndexOf（元素） 返回数组元素第一次 / 最后一次出现时的索引号，不存在时返回-1
       15. includes（元素） 返回布尔，检测元素是否存在于数组
       16. find（fn（element，index，array）｛｝[，this]）/ findIndex（...）
           1. 遍历元素执行函数，若函数返回true，结束遍历并返回该元素/该元素的索引，找不到返回undefined/-1
       17. every（...）/ some（...） 遍历元素执行函数，若每一项/有一项函数都返回true，则该方法返回true
       18. forEach（...） 无返回，遍历元素执行函数，不改变原数组
       19. map（...）/ filter（...） 遍历元素执行函数，返回由每次函数返回结果/返回true的项组成的数组
       20. reduce（fn（pre，current，index，array）｛｝[，初始值]） / reduceRight（...）
           1. 从左向右/从右向左遍历元素执行函数，最后一次的返回值作为返回值。【若不传初始值，从第二项开始迭代，第一项作为pre】
           2. pre在第一次调用时为初始值，之后为前一次调用的返回值；current为当前项；index为当前项的索引
       21. flat（[整数n=1]） 将多维数组转换为低n维的数组，返回新数组
       22. flatMap（fn） 相当于arr.map（fn）.flat（）

## 定型数组

* + 1. ElementType：ES6支持以下类型：Int8【8位整数】、Uint8【8位无符号整数】、Int16、Uint16、Int32、Uint32、Float32、Float64
       1. Uint8ClampedArray：尽量别用，除上述8种类型外的“夹板”数组类型。超出255的值舍入为255、小于0的值舍入为0
    2. 字节序：
       1. 字节序由JS运行时所在系统决定，大端/小端字节序表示高位字节处于低位字节的前侧/后侧
       2. 所有DataView的API可接受一个可选的布尔值，若为true，启用小端字节序。默认为false
    3. ArrayBuffer
       1. 缓冲，是预分配内存，是所有定型数组及视图引用的基本单位，可被垃圾回收机制回收。
       2. 在写入或读取超过缓冲范围时，抛出异常；在写入缓冲时会尽最大努力把值转换为恰当的类型，若无法转换，抛出异常
       3. 创建：new ArrayBuffer（字节数） 在内存中分配特定数量的字节空间，创建后不能调整大小
       4. 实例属性：byteLength【字节数】
       5. 实例方法：slice（...）【同普通数组】
    4. DataView
       1. 一种视图，专为文件和网络I/O设计，支持对缓冲数据的高度控制，相比其他视图性能略差，对缓冲内容无预设，无法迭代
       2. 创建：new DataView（ArrayBuffer实例 [，start][，字节数]） 默认使用全部ArrayBuffer，可设置start和长度
       3. 实例属性：buffer【创建时传入的实例】
       4. 实例方法：
          1. getInt8（索引）/ setInt8（start，值） 从指定索引的字节处开始，获取/设置一个8位整数。支持所有ElementType
    5. 定型数组
       1. 一种视图，特定于一种ElementType并且遵循系统原生的字节序
       2. 创建：
          1. 构造函数：new Int8Array（ArrayBuffer实例 / 长度 / 普通数组 / 定型数组） 没用值进行初始化，各元素为0

传入其他类型定型数组，生成定型数组长度不变，各元素自动转换格式，缓冲字节数会自动调整

* + - * 1. from/of方法：Int8Array.from（普通数组）/ of（参数序列）
      1. 实例属性：length【长度=缓冲字节数/单个元素所占字节数】、BYTES\_PER\_ELEMENT【每个元素的大小】
      2. 实例方法：
         1. 具有数组的大部分实例方法，除了可能修改数组大小的方法【concat、pop、push、shift、splice、unshift】
         2. set（定型数组/普通数组[，start]） 复制传入数组的值到指定索引处
         3. subarray（[start][，end]） 返回子数组
      3. 上溢和下溢：上溢和下溢表示超过类型的最大/最小值。上溢和下溢不影响其他索引。上溢时舍去溢出部分

## Map

* + 1. 映射，用于存储**有序（按插入顺序）键值对**，是可迭代对象，可以将任何JS数据类型作为键，使用SameValueZero进行相等比较[详细](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Equality_comparisons_and_sameness)
    2. 创建：new Map（[包含键值对的可迭代对象]） 如：[[“key”,”val”]]，自定义迭代器等
    3. 实例属性：size【键值对的个数】
    4. 实例方法：
       1. set（键名，键值）【添加，返回映射实例】、get（键名）【获取，若不存在返回undefined】、has（键名）【查询是否存在，返回布尔】
       2. delete（键名）【删除】、clear（）【清空】
       3. keys（）/ values（）/ entries（）【默认迭代器，按插入顺序生成[key, value]形式的数组，也可以通过Symbol.iterator属性获取】
       4. forEach（fn（val，key）｛｝[，this]）
    5. 相比Object：节约内存、插入性能更好、查找速度稍慢、删除性能更好
  1. WeakMap
     1. 弱映射，不可迭代，不能使用非对象作为键，weak指键不属于正式的引用，不会阻止垃圾回收，键被回收后值也会被回收
     2. 创建：new WeakMap（...）
     3. 静态属性：length【永远为0】
     4. 实例方法：set、get、has、delete
     5. 应用：
        1. 实现私有变量：闭包是为了防止外部访问到弱映射，实现真正私有
           1. const ***User*** = (()=>{  
               const wm = new ***WeakMap***()  
               class User{  
               setData(data){  
               const container = wm.get(this) || []  
               container.push(data)  
               wm.set(this, container)  
               }  
               }  
               return User  
              })()
        2. 保存DOM节点元数据：节点被删除后，自动从弱映射中删除

## Set

* + 1. 集合，用于存储任意类型的唯一值，**有序（按插入顺序）**，使用SameValueZero进行比较
    2. 创建：new Set（[可迭代对象]） 如：[...]
    3. 实例属性：size
    4. 实例方法：
       1. add（值）【返回集合实例，添加已有值时直接返回】、has（值）、delete（值）【返回布尔，表示是否有该值】、clear（）
       2. keys（）/ values（）【keys为values的别名，默认迭代器，按插入顺序生成集合内容】
       3. entries（）【返回迭代器，按插入顺序生成[元素n，元素n]】、forEach（fn（val，key）｛｝[，this]）【val与key完全相等】
    5. 自定义API：应创建一个Set的子类，在子类上实现静态方法，并在子类实例方法中调用这些静态方法。
       1. 需要考虑：方法要能处理多个集合实例；返回的集合保证有序；避免使用扩展运算符；不修改已有集合实例
  1. WeakSet
     1. 弱集合，不能使用非对象作为值。不可迭代。weak指值不属于正式的引用，不会阻止垃圾回收
     2. 创建：new WeekSet（...）
     3. 实例方法：add、has、delete
     4. 应用：保存DOM节点元数据：节点删除后，自动从弱集合中删除
  2. 迭代：Array、定型数组、Map、Set都定义了默认迭代器。都支持顺序迭代和扩展操作符

# 迭代器与生成器

* 1. 迭代即按照顺序反复执行一段程序，通常有终止条件。通过循环迭代不理想，使用迭代器可以实现通用迭代
  2. 迭代器
     1. **可迭代协议（Iterable接口）**：实现该协议需要：支持迭代的自我识别、能创建迭代器。【在ES中，即具有默认迭代器属性】
        1. **默认迭代器属性**：一种属性，以Symbol.iterator为键，值为一个无参数函数，其返回值为一个迭代器
        2. 实现了可迭代协议的内置类型：String、Array、定型数组、Set、Map、arguments对象、NodeList等DOM集合类型
     2. **可迭代对象**：实现了可迭代协议的对象
        1. 接收可迭代对象的原生语言特性：会在后台调用默认迭代器属性创建迭代器来获取值，会自动忽略done为true时的value值
           1. for-of、数组解构、扩展操作符、Array.from（）、创建集合和映射、Promise.all（）/ race（）、yield\*操作符
     3. **迭代器协议（Iterator接口）**：实现了next（）方法，每次调用返回一个具有value和done属性的对象
        1. value属性：下一个将要返回的值，默认值为undefined
        2. done属性：布尔值，默认值为false。迭代器已将序列迭代完毕时，done为true；否则为false
     4. **迭代器**（Iterator）：实现了迭代器协议的，按需创建的一次性对象
        1. 迭代器维护着对可迭代对象的引用，会阻止垃圾回收程序回收可迭代对象
        2. 原生迭代器都实现了可迭代协议，调用其默认迭代器属性会返回迭代器本身
        3. 提前终止迭代器：如果迭代器没有被关闭，则可以从上次离开的地方继续迭代【数组的迭代器不可被关闭】
           1. return方法：迭代器可实现一个会被自动调用的return（）方法，调用时关闭迭代器【存疑，自定义对象无法关闭】

调用return的场景：for-of循环通过break、continue、return、throw提前退出；解构赋值未用到所有值时

return方法需要返回一个具有done属性，值为true的对象

* 1. 生成器
     1. **生成器**：生成器是一个特殊函数，可以在函数块中暂停和恢复代码执行
        1. 创建：在任何可定义函数【箭头函数除外】的地方均可创建。函数名前加 \* 表示它是一个生成器，匿名函数加在function关键字后
        2. yield关键字：yield [值] 只能在生成器中使用，用于退出（暂停）生成器，自身值默认为undefined
        3. yield\*关键字：yield\* 可迭代对象 依次迭代该对象并暂停生成器。将可迭代对象返回done：true时的value值作为自身值
           1. 应用：可迭代对象可以是生成器本身，以此实现递归。
        4. 生成器很适合作为默认迭代器使用
     2. **生成器对象**：调用生成器返回一个处于暂停状态（suspended）的生成器对象。生成器对象实现了迭代器协议
        1. 同一个生成器创建的多个生成器对象不相同，具有不同的作用域
        2. next（[值]）方法：
           1. 调用生成器对象的next（）方法，可以使暂停状态的生成器恢复执行，返回一个对象：

若执行中通过yield关键字退出，返回对象的value属性值为yield的值，done为false

若执行中通过yield\*关键字退出，返回对象的value属性值为该次迭代返回的value值，done为false

若执行中通过return关键字退出，返回对象的value属性值为return的值，done为true

* + - * 1. next（）方法可传入一个参数，作为上一次使生成器暂停的yield关键字的值（第一次调用next方法传入的值会被忽略）
      1. 提前关闭生成器：生成器进入关闭状态后，后续的next（）调用都返回｛value：undefined，done：true｝
      2. return（[值]）方法：强制生成器进入关闭状态。返回一个对象：｛value：传入的参数【默认undefined】，done：true｝
      3. throw（[值]）方法：在生成器暂停时使用，将提供的错误注入到生成器中（由上一次使生成器暂停的yield关键字抛出）
         1. 若错误未被处理，关闭生成器；若错误被处理，则会跳过该次的yield
         2. 若生成器还没有开始执行，调用throw视为从throw（）方法处抛出错误

# 对象、类与面向对象编程

## 对象

* + 1. 是一组属性的无序集合。
    2. 创建：
       1. ①字面量创建：const obj =｛属性：值，属性：值，方法名：function（）｛｝。。。｝； 【不会实际调用Object（）函数，无法识别类】
       2. ②空对象创建：const obj = new Object（）； obj. 属性名 = 值；
       3. ③工厂函数：定义一个工厂函数，调用时返回对象【无法识别类】
       4. ④构造函数：
          1. 声明构造函数：function 构造函数名（）｛｝ 或 let 构造函数名 = function（）｛｝
          2. 创建对象：new 构造函数名（） 返回实例对象，若不传参数可以省略小括号
          3. 注意：

按照惯例，构造函数名使用大驼峰；

构造函数也是函数，不通过new调用的函数为普通函数，通过new调用的函数为构造函数

* + - * 1. 不足：对于同名方法，如果在构造函数内声明方法，创建对象时，会给每个实例对象创建唯一函数实例，浪费空间。

解决方式：在构造函数外部定义函数，在构造函数内创建变量引用；或使用原型模式（推荐）

* + - * 1. new操作符的执行过程：

在内存中创建空的对象；该对象的[[Prototype]]特性被赋值为构造函数的prototype属性；构造函数内的this指向新对象；执行构造函数的代码赋予对象属性和方法；若构造函数返回非空对象，则返回该对象，否则返回新对象

* + 1. 内部特性：内部特性用来描述属性或实例的特征。内部特性不能在JS中直接访问，规范中会用两个中括号将其名称括起来
       1. [[Prototype]] 指向该实例对象的构造函数的原型对象
       2. [[Configurable]] 描述属性，值为布尔值，默认为true
          1. 表示属性是否可以通过delete删除并重新定义，其特性是否可以被修改，是否可以被改为访问器/数据属性
       3. [[Enumerable]] 属性，布尔值，默认为true，表示属性是否可枚举，即是否可以通过for-in循环返回【符号属性除外】
       4. [[Writable]] 属性，布尔值，默认为true，表示属性的值是否可被修改
       5. [[Value]] 描述属性，包含属性实际的值，默认为undefined
       6. [[Get]] 获取函数，在读取属性时调用，默认为undefined
       7. [[Set]] 设置函数，在写入属性时调用，默认为undefined
    2. 属性： 对象的属性名是字符串【可以包含非字母数字字符，中括号语法访问】、数值【自动转换为字符串】或符号
       1. 属性操作：
          1. 访问属性：对象名. 属性名 或 对象名[ ‘属性名’ / 返回属性名字符串的表达式 ]
          2. 删除属性：delete 对象名. 属性名
       2. 属性类型：
          1. 数据属性：包含一个保存数据值的位置，值会被读取或写入到该位置。由4个特性描述：

[[Configurable]]、[[Enumerable]]、[[Writable]]、[[Value]]

* + - * 1. 访问器属性：包含一个获取（getter）函数和一个设置（setter）函数。获取函数和设置函数不是必须的。由4个特性描述：

[[Configurable]]、[[Enumerable]]、[[Get]]、[[Set]]

只设置了获取函数，尝试修改属性会被忽略（严格模式报错）。只设置了设置函数，尝试读取会返回undefined（严格报错）

* + 1. 对象的增强语法：同样适用于类
       1. 属性名的简写：属性名与变量名一样时，可以只写变量名【name：name => name】
       2. 可计算属性：可以在对象字面量中，通过 [表达式] 的方式，将表达式的值作为属性名【若求值过程抛出异常，保留已完成的修改】
       3. 方法名的简写：方法名：function（）｛｝ => 方法名（）｛｝ 方法名也可使用可计算属性
    2. 对象解构
       1. 解构赋值：
          1. 声明并赋值：let ｛[属性名：]变量名[=默认值]。。。｝= 源数据结构

在内部会使用函数ToObject（）【无法在运行环境中直接访问】将源数据结构转换为对象

待补充ToObject

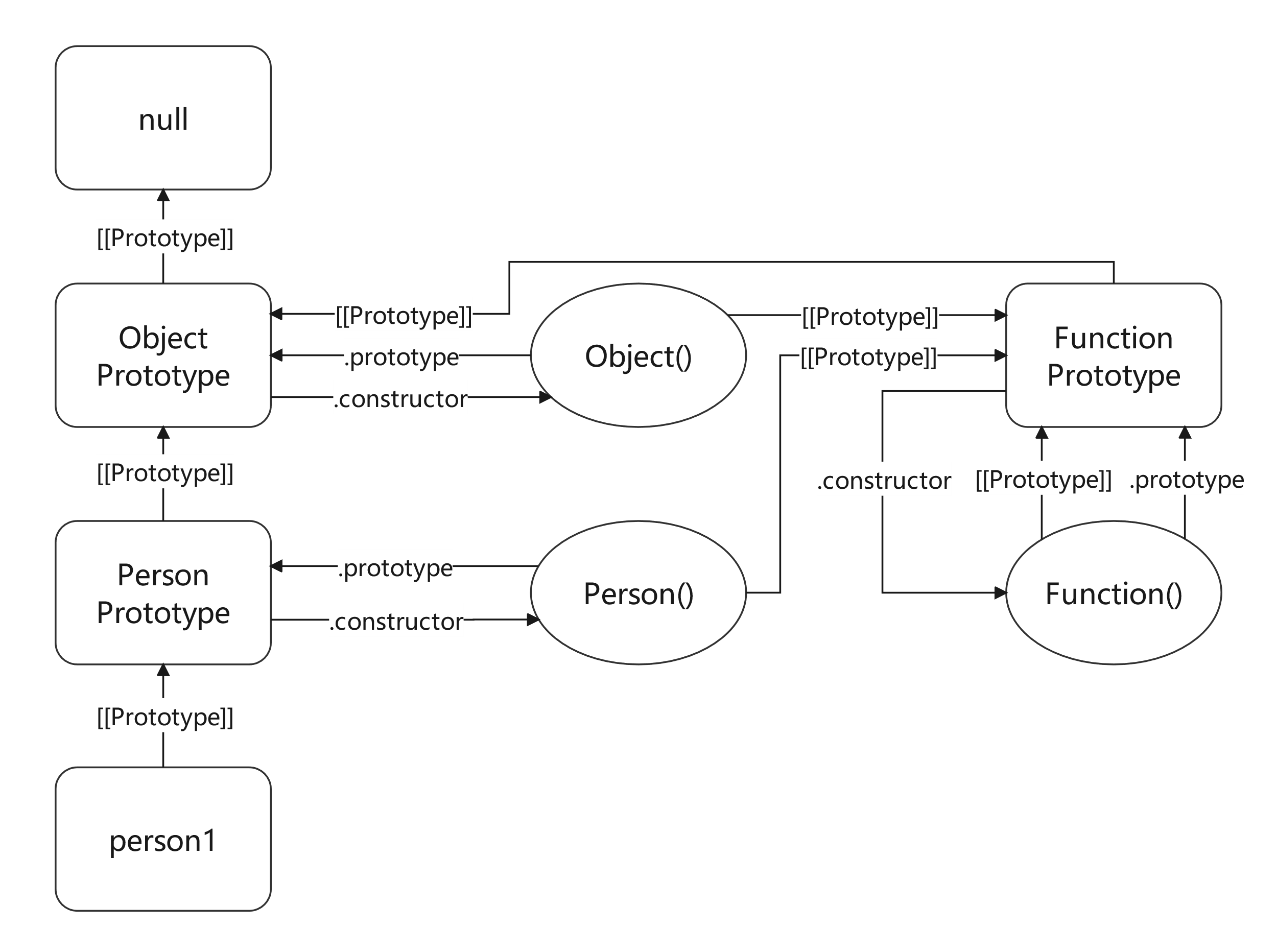
声明变量，从对象读取指定属性并赋值给变量，若对象没有该属性则赋默认值，若没有指定默认值则赋undefined

* + - * 1. 给已声明变量赋值：（｛[属性名：]变量名[=默认值]。。。｝= 源数据结构）
        2. 给函数的参数赋值：fn xx（｛。。。｝）｛｝ xx（源数据结构）；
      1. 注意：
         1. 解构赋值可以使用嵌套结构：如，将person.job.title赋值给title【let｛job：｛title｝｝=person】
         2. 若解构过程中抛出异常，保留已完成的修改；给函数的参数解构赋值不会影响arguments对象
    1. 静态方法：Object.xx
       1. is（值1，值2） 判断两个值是否为同一个值
       2. defineProperty（对象，属性名，｛配置项｝） 在对象上定义新属性，或修改现有属性，返回此对象
          1. 可选的配置项：配置该属性的特性

value：值【默认undefined】，enumerable：布尔【默认false】，writable...，configurable...、get（）｛｝、set（newValue）｛｝

* + - 1. defineProperties（对象，｛属性名：｛配置项｝。。。｝） 同时定义或修改多个属性，返回此对象
      2. getOwnPropertyNames（对象） 返回包含该对象所有实例属性（非符号）属性名的数组
      3. getOwnPropertySymbols（对象） 返回包含该对象所有实例属性（符号）属性名的数组
         1. 以上两个方法及assign（）枚举顺序确定，先升序枚举数值键，再按插入顺序枚举字符串和符号键
      4. getOwnPropertyDescriptor（对象，属性名） 返回某实例【不沿原型链查找】指定属性的属性描述对象
         1. 属性描述对象：含有value、enumerable等属性，其值为被描述属性的同名特性的值
      5. getOwnPropertyDescriptors（对象） 描述该实例所有属性，返回｛属性名：属性描述对象。。。｝
      6. assign（目标对象，多个源对象） 将每个源对象中可枚举和自有属性复制到目标对象，返回目标对象
         1. 执行的是浅复制；若复制期间出错，会中止操作并抛出异常，保留已完成的修改
         2. 若源对象设置了[[Get]]，会通过getter获取属性的值；若目标对象设置了[[Set]]，会通过setter设置属性的值
      7. setPrototypeOf（实例，原型对象）/ getPrototypeOf（实例） 设置/获取实例的隐式原型属性的值
         1. 使用setPrototypeOf会严重影响性能，推荐使用create（）创建新对象并通过assign（）复制属性
      8. create（原型对象[, ｛属性名：｛配置项｝。。。｝]） 创建新对象，指定隐式原型并返回，第二个参数同defineProperties
      9. keys（对象） 返回包含该对象所有可枚举属性名的字符串数组
         1. keys（）与for-in循环的枚举顺序不确定，取决于JS引擎
    1. 静态属性：直接通过 类名.属性名=xx 的方式添加，无法从实例上访问
       1. Object.freeze（对象1） 返回对象1，该对象的属性被修改时会静默失败
    2. es6扩展静态方法：
       1. Object.
    3. es8扩展静态方法：
       1. Object.entries（对象） 返回一个二维数组， 每个元素都是数组，小数组的第一个元素是键名，第二个元素是值
       2. Object.values（对象） 返回由对象值组成的数组
    4. es10扩展静态方法：
       1. Object.fromEntries（参数） 创建对象，返回一个对象
          1. 参数：二维数组：[ [键，值]，[键，值] 。。。]
    5. 实例方法：
       1. constructor（...） 构造函数
       2. .toString（） 返回对象的字符串表示
       3. .valueOf（） 返回对象对应的字符串、数值或布尔值表示
       4. hasOwnProperty（’属性名’） 返回布尔，检测实例上（不沿原型链查找）是否有指定属性
          1. 与in操作符结合，可以判断属性是否只存在于原型链上
       5. isPrototypeOf（obj） 返回布尔，判断参数的隐式原型属性是否指向该实例
       6. propertyIsEnumerable（’属性名’） 返回布尔，判断给定的属性是否可枚举
       7. .toLocaleString（） 返回对象的字符串表示，反映对象所在的本地化执行环境

## 原型

* + 1. **原型对象**： 创建函数时，自动生成一个对应的原型对象。自定义构造函数的原型对象默认只有constructor属性，指回自定义构造函数。
       1. Object（）函数的原型对象的原型对象是null
    2. **显式原型属性**：即每个函数都有的prototype属性，在函数创建时自动添加，指向原型对象
    3. **隐式原型属性**：即实例的[[Prototype]]特性，实例创建时该特性自动赋值为其构造函数的原型对象，可用实例对象的\_\_proto\_\_属性访问
       1. 该属性为非标准属性，建议改用ES6方法Object.getPrototypeOf（）和Object.setPrototypeOf（）操作[[Prototype]]特性
    4. **原型链**：任何原型对象也有其原型对象。从实例开始沿着**隐式原型属性**层层向上，直到null为止的链条，称为原型链
       1. 读取对象的属性时，会沿着原型链层层查找，若找不到，则返回undefined；设置对象的属性则直接设置在该对象上
    5. 
    6. Function xx（）声明函数时，相当于创建了一个函数对象，函数对象同时拥有\_\_proto\_\_和prototype属性
    7. 所有函数的\_\_proto\_\_属性都为Function.prototype。Function的构造函数为自己，所以两个属性指向同一个原型对象
    8. instanceof二元运算符：实例对象A instanceof 构造函数B 若B的显式原型对象在A的原型链上，返回true，否则返回false
    9. 利用原型链继承：子类的原型设置为父类的一个实例对象。并将该实例对象的constructor属性指向子类
    10. 借用构造函数继承（假继承）：在子类构造函数内通过call调用父类的构造函数。如：父类.call(this, 属性。。。)，无法获取原型链上的数据
    11. 组合继承：利用原型链实现对父类型对象的方法继承，利用父类构造函数获取相同属性
    12. 直接继承：将子类的prototype指向父类的prototype，优点效率高，缺点：对子类原型的修改会体现在父类上

# 函数

* + 1. 函数是实现特定功能的n条语句的封装体
    2. 声明函数：
       1. function 函数名（形参）｛｝
       2. var fun = function（形参）｛｝ fun为变量名而非函数名，内部存的是值，存在变量提升问题
    3. 调用函数：函数名（参数序列）
    4. return语句：用于函数体内，将表达式的值作为函数的返回值，然后退出函数。若没有表达式或函数没有return语句，返回undefined
       1. return [表达式]；
    5. 命名参数有多个时，以第一个为准
    6. 实参的个数多余形参的个数时，多余的实参会被舍弃；实参的个数小于形参的个数时，未接受值的形参值为undefined
    7. 所有函数的参数都是按值传递的。参数的值会被赋值给一个局部变量【形参】。传递引用值时，传递的是地址值，本质是值传递。
    8. 函数没有return时，返回undefined
    9. 函数内置arguments对象：存储了所有的实参，以伪数组的形式传递
       1. arguments.callee 保存当前执行的函数对象
    10. 函数方法：
        1. 函数名.call（this对象[，参数1，参数2。。。]） 将指定对象作为函数的this，指定参数交给函数调用
        2. 函数名.bind（this对象[，参数1，参数2。。。]）（） 返回一个函数，函数体内的this会被绑定xx。也可以把参数写在调用的（）中
        3. 函数名.apply（this对象[，arguments对象或数组]） 同call，区别在于参数以数组形式传递
    11. 回调函数：就是一个被作为参数传递的函数，会在特定的条件发生时被调用。
        1. dom事件回调函数、定时器回调函数、ajax请求回调函数、生命周期回调函数
    12. 立即执行函数表达式： （function（）｛｝）（） 或 （function（）｛｝（）） 独立创建了一个作用域，隐藏实现
    13. this：
        1. 函数中，this指向调用函数的执行上下文对象，在全局上下文中调用this指向window，new关键字会改变构造函数中this指向实例
        2. 在调用函数而没有明确设置this值的情况下（没有作为对象的方法调用，或通过call apply调用），this指向Global对象
        3. 函数（）与 （函数）（）是相等的，this的指向不会发生改变。只要函数赋值给其他变量，this就会改变
        4. 严格模式下：函数内部的this不允许指向window，会置为undefined；但定时器的this及在全局作用域下直接使用this不受影响
    14. 闭包（closure）：
        1. 闭包的生命周期：在定义一个函数，且该函数引用了另一个函数作用域中的变量时（通常在嵌套函数中），会产生闭包。包含闭包的函- 数对象成为垃圾对象时，会释放闭包
        2. 闭包是包含被引用变量（函数）的对象，该对象在内部函数里面。当不再有变量指向闭包函数时，闭包会被释放
        3. 每一次引用外部变量都会产生闭包，例：引用的变量为i，i变化了x次，会产生x个闭包，记录各个i
        4. 作用：延长局部变量的生命周期；让函数外部可以操作函数内部的数据
        5. 常见的闭包：将函数作为另一个函数的返回值；将函数作为实参传递给另一个函数调用
        6. 闭包中的this：若一个函数返回一个闭包函数，在闭包函数中的this指向闭包函数的调用者，而非外部函数，可以在外部函数中用变- 量that保存this，然后在闭包函数内调用that实现闭包
        7. 应用：自定义JS模块：
           1. 方法一：模块中只有一个大的函数，内部装有数据和方法，大函数最后返回一个对象，对象中装有数据和方法
           2. 方法二：大函数改为匿名函数，接收window参数，不返回，最后加window.xxx = 装有数据的对象，再在大函数外部加括号（外- 部括号传递window，防止代码压缩出错）立即执行
        8. 注意事项：
           1. 内存泄漏：IE9之前，将HTML元素保存在闭包作用域中，就相当于宣布该元素不能被销毁。

解决：获取HTML元素，将指定属性保存为变量，闭包中引用该变量，之后将保存HTML元素的变量置为null

* + - * 1. 缺点：函数执行完后，函数局部变量没有释放，占用内存时间变长；容易造成内存泄漏

解决：谨慎使用闭包；及时释放

* 1. 多线程（Web Workers）：HTML5提供的js多线程解决方案，子线程完全受主线程控制，且不得操作DOM
     1. 使用：
        1. 创建在分线程执行的js文件：
           1. var onmessage = function（event）｛

var number = event.data； 通过event.data获取发送来的数据

postMessage（number + 1）； 将获取到的数据发送回主线程

* + - * 1. ｝
      1. 在主线程中的js中发送消息：
         1. var worker = new Worker（“xx.js”）； 创建Worker对象并向它传递在新线程中执行的脚本的URL
         2. worker.onmessage = function（event）｛

console.log（event.data）；

* + - * 1. ｝ 当接收到worker的数据时触发函数
        2. worker.postMessage（“hello world”）； 向worker发送数据
    1. 不足：速度慢；不能跨域加载js；worker内代码不能访问DOM；兼容性问题
  1. ES6-11：
     1. ES6：
        1. let、const声明关键字，详见“变量”
        2. let [a, b] = [100, 200] 数组的解构赋值。右边可换成值为数组的变量。相当于声明了ab两个变量并按顺序赋值
        3. let {name, age} = {name:xx, age:fn（）{} } 对象的解构赋值
           1. 右边同样可以换成值为对象的变量。左边接收变量必须与右边对应属性名同名，左边可以只写需要的项，也没有顺序要求
           2. 可以通过｛name=“aa”，age｝的方式指定默认值，也可在函数的形参中使用解构赋值
        4. const a = {name, act（）｛xx｝} 对象声明的简化格式，相当于name：name，act：function（）｛xx｝
        5. obj[表达式]=xxx 对象的属性名可以用表达式
        6. let fn = （参数）=> ｛函数体｝ 箭头函数声明，相当于let fn = function（参数）｛函数体｝
           1. 特点：

没有自己的this，它的this捕获为函数声明时所处的执行上下文，无法被call等函数改变；

不能作为构造函数实例化对象；不能使用arguments变量（通过结构赋值获取可变参数）

* + - * 1. 箭头函数的简写：

当形参只有一个时，可以省略小括号；

当函数体只有一条语句，且是return语句时，可以省略花括号和return，语句的执行结果就是返回值

* + - 1. function xxx（a，b，c=10）｛｝ 给参数设置初始值
      2. function xxx（a，b，...args）｛args｝ rest参数，用于获取函数的实参，使用时用args，是一个数组。声明时必须放在最后一位
      3. xxx（...数组等） 扩展运算符，将数组转换为用逗号分隔的参数序列。用于数组合并、浅拷贝、伪数组转数组
      4. Promise：ES6引入的异步编程的新解决方案，解决了回调地狱。语法上Promise是一个构造函数
         1. Promise有三种状态：pending（等待）、fulfilled（成功）、rejected（失败）
         2. const p = new Promise（function（resolve，reject）｛异步操作放在这｝）

在函数体内调用resolve（data）方法，会将p对象的状态更改为成功；调用reject（err），更改为失败

* + - * 1. Promise对象的方法：设对象为p

p.then（function（value）｛｝[，function（reason）｛｝]） 若状态为成功，执行第一个方法，为失败执行第二个方法

then方法返回一个Promise对象，对象状态由回调函数的执行结果决定，若回调函数返回的结果是非promise类型的属性，或没有返回，则对象状态为成功，Promise对象的值属性为回调函数的返回值；若回调函数返回的结果为promise对象，则外对象状态为内对象状态，外对象的值属性为内对象的值属性；then方法可以链式调用

p.catch（function（）｛｝） 当对象状态为失败时执行

then与catch的注意点：p.then（）.catch（）

若不报错，只有在对象状态为失败，且不存在then的第二个方法时，才会进入catch

若then执行时报错，可以被catch捕获报错信息；then的第二个方法无法捕获第一个方法的报错

若Promise对象执行时报错，Promise对象状态变更为失败，可被then的第二个方法或catch捕获错误

若catch执行时报错，可被链式调用的下一个catch捕获

* + - * 1. 应用：

封装读取文件：需求：读取多个文件并拼接

const fs = required（‘fs’）

// fs.readFile（‘路径’，（err，data）=>｛if（err）throw err；console.log（data.toString（））｝） 用fs读取的用法

const p = new Promise（function（resolve，reject）｛ 用Promise封装

fs.readFile（‘路径1’，（err，data）=>｛if（err）reject（err）；resolve（data）｝） 读取第一个文件

｝）

p.then（value =>｛return new Promise（（resolve，reject）=>｛读取代码同上｝）｝）.then。。 读取第二三。。个文件

封装AJAX请求：参考上文，把语句放入Promise函数，通过resolve和reject返回xhr的response和status属性

* + - 1. Class：
         1. 创建：

class 类名｛

constructor（参数）｛该方法为构造方法，在构造时自动执行｝ this指向实例对象，构造函数可以不要

方法名（）｛方法体｝ ｝

* + - * 1. 使用：let xx = new 类名（） 由class创建的类本质是一个特殊的构造函数，不通过new调用时会报错，不能调用call
        2. 注意：

在class中声明的方法都保存在原型对象中

* + - * 1. 静态成员：在类声明体中，成员声明前添加static关键字，可以指定成员为类的成员而非实例的成员
        2. 继承：class 类名 extends 父类名 ｛constructor（参数）｛super（父类构造函数的参数）｝｝

super关键字：

作为函数调用，代表父类的构造函数，只能在子类的构造函数内使用，子类若有构造函数，必须执行一次super（）

此时super中的this指向子类的实例

作为对象使用，指向父类的原型对象；在静态方法中使用，super指向父类

子类方法中通过super调用父类方法时，方法内部的this指向当前子类实例

super单独使用时会报错

* + - * 1. 重写：直接在子类中声明同名方法
        2. getter与setter：get 属性名（）｛｝ / set 属性名（newVal）｛｝ 当属性被读取或被赋值时，调用该函数
      1. 模块化：
      2. ES6规范：
         1. 暴露：

分别暴露：在每个要暴露的数据前写 export 数据

统一暴露：整个文件单独写export｛属性1，属性2｝

默认暴露：export default｛属性1：值，属性2：值｝

使用：xx.default.属性1

* + - * 1. 引入：在html文件文件中不指定script标签的src，指定type属性值为module

通用导入：在标签中写上import \* as xx from “js文件url”；

解构赋值：import｛变量1，变量2｝from “url”； 或 import｛default as 变量｝ from “url”

变量重名时可用 变量1 as 变量3 代替 变量1

**分别暴露和统一暴露在引入时必须通过解构赋值的方法引入**

简便形式：针对默认暴露：import 变量 from “url” 此时通过 变量.属性1 访问属性

专用引入app.js文件：在一个js文件中引入所有模块，html中指定：<script src=“app.js” type=“module”>

* + - * 1. babel转换：转换代码为es5

安装：npm i babel-cli babel-preset-env browserify -D

转换：babel 文件或文件夹路径 -d 导出文件夹路径 --presets=babel-preset-env

打包：browserify 转换后的app.js文件路径 -o 输出文件bundle.js路径

引入：<script src=“bundle.js”>

* + - * 1. 特点：

1、ES6模块中的值属于【动态只读引用】。

2、只读，即不允许修改引入变量的值，import的变量是只读的，但可修改复杂引用类型的属性。**当模块遇到import命令时，就会生成一个只读引用。等到使用时，再根据这个只读引用，到被加载的那个模块里面去取值**

3、动态，即原始值发生变化，import加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。

* + - 1. commonjs：
         1. 特点：
         2. 1、对于基本数据类型，属于赋值。引入模块修改变量值不会对原模块变量产生影响。可通过导出getter使两个变量同步
         3. 2、对于复杂数据类型，属于引用赋值。两个模块引用的对象指向同一个内存空间，对该模块的值做修改时会影响另一个模块。
         4. 3、当使用require命令加载某个模块时，就会运行整个模块的代码。
         5. 4、当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。
         6. 5、循环加载时，属于加载时执行。即脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"（c加载a，a加载b，b加载a），就只输出已被执行部分的缓存中的值，还未执行的部分不会输出，不会执行。
      2. Proxy：[window.]Proxy
         1. const p = Proxy（源对象，｛配置对象｝） 对p的属性的增删改查操作会映射到对象上，并触发getter、setter（）等

get（target【源对象】，propName【属性名字符串】）｛**return Reflect.get（target，propName）**｝ 读取属性时调用

set（target，propName，value）｛**Reflect.set（target，propName，value）**｝ 修改或追加属性时调用

deleteProperty（target，propName）｛**return Reflect.deleteProperty（target，propName）**｝ 删除属性时调用

* + - 1. Reflect：[window.]Reflect
         1. Reflect.get（对象，‘属性’） 获取对应属性的值 Reflect.set（对象，‘属性’，值） 修改或追加属性的值
         2. Reflect.deleteProperty（对象，‘属性’） 删除指定属性
         3. Object.defineProperty（..）重复添加已有属性时，会抛出异常；Reflect.defineProperty（..）不抛异常，返回布尔值判断是否成功
      2. 省略语法：
         1. 如果定义的属性名称与属性值变量同名，可以省略属性名称以及冒号
         2. [表达式] 将表达式执行到最后的值作为该处的值。例如{ [ ‘red’.toUpperCase（） ]：200 }
         3. 在对象中定义方法可以省略冒号以及function
  1. ES7-11：
     1. ES7：
     2. ES8：
        1. async函数：async function 函数名（）｛｝
           1. async声明的函数表明内部含有异步操作，调用该函数时，会立即返回一个Promise对象（pending状态）
           2. async内return的返回值将作为then方法回调函数的参数：

若return的不是一个Promise对象或是一个成功的promise对象，则类型为成功，值为ruturn值。

若return一个失败的promise或抛出错误throw new Error（‘xx’），则类型为失败。

* + - * 1. 对于 async函数（）.then（），会在调用时返回pending状态的Promise对象，然后在async函数return语句执行完毕后执行then
      1. await表达式：await promise对象
         1. await表达式必须在async函数中，await返回promise对象成功的值。若promise对象状态为失败，就会抛出异常，值为失败值
         2. 遇到await语句时，async函数会在执行完await语句后再继续向下执行
      2. 用于异步编程，使用：封装函数，返回Promise对象。声明async函数，使用变量接收await的返回值，await表达式调用函数
    1. ES9：
       1. 对象的Rest参数：function 函数名（｛参数1，参数2，...参数3｝）｛｝ 调用时多余的参数都会存到参数3（是对象）中
       2. 扩展运算符：...对象 => 属性1：值1，属性2：值2
    2. ES10：
       1. Symbol.prototype.description 获取创建Symbol时传入的字符串
    3. ES11：
       1. 私有属性：在class内声明变量时，在变量名前加#。无法在类的外部调用该变量，类中访问时需用#变量名。
       2. Promise.allSettled（[Promise对象1，Promise对象2。。。]）
          1. 返回一个成功状态的Promise对象，其值为一个数组，其元素为对象，有status、value/reason属性，代表参数的状态和值
       3. Promise.all（同上） 返回Promise，若参数内Promise对象有失败，则返回对象为失败状态，值为失败对象的值
       4. str.matchAll（reg） 返回一个可迭代对象，可用let...of获取结果，也可使用扩展运算符
       5. ?. 可选链操作符，读取位于连接对象链深处的属性的值，而不必明确验证链中的每个引用是否有效
          1. 通过undefined.xx调用时，可能会报错，而用undefined ?. xx ，不会报错，执行值为undefined
       6. ?? 空值合并运算符，当左侧的操作数为null 或 undefined 时，返回其右侧操作数，否则返回左侧操作数
       7. import（‘url’） 动态导入，返回一个Promise对象，对象的值是导入模块暴露的变量
       8. globalThis 指向全局对象

# **DOM**

* + 1. Document Object Model文档对象模型。将文档看作一个模型，顶级对象为document
    2. 获取元素：
       1. DOM树：文档（页面）、元素（标签）、节点（所有内容）。以上内容都看作对象
       2. 父元素.getElementById（’id’） 获取父元素中有指定ID的元素，返回一个元素对象，若没有返回null
       3. 父元素.getElementsByTagName（’id’） 获取父元素中指定标签名的元素，返回元素对象集合（伪数组），若没有则返回空的伪数组
       4. document.body / .documentElement 获取body / html元素
       5. HTML5新增：
          1. 父元素.getElementsByClassName（‘类名’） 返回指定类名的元素集合，若无返回空伪数组
          2. 父元素.querySelector（‘选择器’） 返回指定选择器的第一个对象，若无返回null
          3. 父元素.querySelectorAll（‘选择器’） 返回指定选择器的元素集合，若无返回空伪数组
    3. 事件：由事件源、事件类型、事件处理程序组成（事件三要素）
       1. DOM事件流：捕获阶段、当前目标阶段、冒泡阶段
          1. js代码只能执行捕获或冒泡中的一个阶段、onclick和attachEvent只能得到冒泡阶段、addEventListener可以选择冒泡或捕获
       2. 注册事件：
          1. 传统方式：html标签内onclick=”js代码” 或 js内btn.on事件名 = function(){} 只能设置一个处理函数
          2. 方法监听注册方式：推荐。可以绑定多个方法

eventTarget.addEventListener(type，listener[，useCapture]) IE9+

type：事件类型字符串

listener：处理函数，事件发生时调用。若为已有函数，不需要加小括号

useCapture：若为true，表示在捕获阶段调用。若为false或省略，表示在冒泡阶段调用。

eventTarget.attachEvent（eventNameWithOn，callback）非标准，仅适配ie9以下，尽量别用

eventNameWithOn：带on的事件类型字符串

* + - 1. 删除事件：
         1. 传统方式：on事件 = null；
         2. 方法监听方式：

eventTarget.removeEventListener(type, listener[, useCapture])

eventTarget.detachEvent(eventNameWithOn, callback)

* + - 1. 事件对象：div.onclick = function(e){} e为事件对象，当做形参，系统自动创建不需要传递参数。是事件相关数据的集合
         1. ie678需要通过window.event获取事件对象。兼容性写法：e = e || window. event
         2. e .target 触发事件的对象。例：ul绑定事件，li触发事件。ie678中用e.srcElement（非标准）
         3. e .currentTarget 绑定事件的对象，ie678不可用，相当于this，一般用this
         4. e .type 事件类型
      2. 阻止默认事件行为：e .preventDefault()（ie9+）、e .returnValue=false（ie678）、return false；（传统注册方式，后续代码不执行）
      3. 阻止冒泡：e .stopPropagation()（ie9+）、e .cancelBubble = true（ie678）
      4. 事件委托：不在每个子节点单独设置事件监听器，而是监听器设置在父节点上，然后利用冒泡原理影响每个子节点
      5. 常用鼠标事件： mousemove鼠标移动、mouseup、mousedown、mouseenter、mouseover、mouseout、mouseleave
         1. 禁止鼠标右键菜单：contextmenu。阻止默认行为即可 禁止鼠标选中：selectstart，阻止默认行为
         2. e .clientX / .clientY 返回鼠标相对于浏览器窗口可视区的X / Y坐标，不带单位
         3. e .pageX / .pageY 返回鼠标相对于文档页面的X / Y坐标，不带单位，IE9+
         4. e .screenX / .screenY 返回鼠标相对于电脑屏幕的X / Y坐标，不带单位
         5. e .offsetX / .offsetY 返回鼠标相对于事件元素的X / Y坐标，不带单位
         6. mouseenter和mouseover的区别：mouseover在鼠标经过自身和子元素时都会触发；mouseenter只在经过自身触发（无冒泡）

mouseleave和mousedown同理，mouseleave不会冒泡

* + - 1. 常用键盘事件：keyup按键弹起时、keydown按键按下时（长按连续触发）、keypress（长按连续触发，不识别功能键）
         1. 执行顺序为：keydown – keypress – keyup。在文本框中：keydown – 文字落入 – keyup
         2. 事件对象为：keyboardEvent。

e.keyCode 返回该键的ASCII码值（keyup和keydown不区分字母大小写，默认大写；keypress区分大小写）

* + - 1. 事件大全：
         1. 鼠标：

click、mousemove、mouseup、mousedown、mouseenter、mouseover、mouseout、mouseleave、wheel

* + - * 1. 键盘：keydown、keyup、keypress
        2. 表单：submit（表单提交）、change（表单元素的value发生改变）
        3. 窗体：

scroll（滚动条滚动）、contextmenu（开启右键菜单）、selectstart（拖动左键选择）、resize（窗体大小发生改变）、load（页面完全加载完毕）、DOMContentLoaded（DOM加载完毕并执行完js代码）

* + - * 1. 注：若要监视DOM元素的大小变化，可使用ResizeObserver API
      1. load事件和pageshow事件：
         1. load事件在以下情况触发：a标签的超链接、F5或者刷新按钮（强制刷新）、前进后退按钮、打开页面
         2. 火狐中存储往返缓存，前进后退按钮不能刷新页面，需要用pageshow事件触发。
         3. pageshow事件：添加给window对象，页面显示时触发，load事件后触发，事件对象.persisted判断是否为缓存中页面触发事件
    1. 操作元素：元素属性的值为字符串形式
       1. 元素 .innerText / .innerHTML 为属性，元素的内容（双标签），前者不识别html标签并去除空格和换行，后者识别（推荐）
       2. 元素 .src / .href / .id / .alt / .title / .type / .value / .checked（布尔型）/ .selected / .disabled 无法获取自定义属性
       3. 元素 .getArrtibute（‘属性’） 获取元素属性。相较上一方法，可以获得元素的自定义属性。
       4. getComputedStyle(obj)[‘属性arrt’] 获取元素CSS属性，ie9+
       5. 元素 .currentStyle[‘属性arrt’] 获取元素CSS属性（ie）
       6. 自定义属性（H5）：在元素标签内直接填写。如：<div id=“demo” data-index=“1”></div>，命名规范：data-。。。-。。。
          1. 获取自定义属性：兼容性：.getAttribute（‘属性’）。

ie11+：.dataset.属性 或 .dataset[‘属性’]。此处的属性为去掉data-，再将剩余部分改为小驼峰的属性名

* + - 1. 元素 .setAttribute（‘属性’，‘值’）； 设置属性的值。
      2. 元素 .removeAttribute（‘属性’）； 移除属性
      3. 元素 .style .样式名 修改元素样式，样式名采用小驼峰，该操作产生行内样式
      4. 元素 .className 修改元素类名，会覆盖原来的类名。需要修改多个属性时用这个比改style性能好
    1. 节点操作：
       1. 节点类型 .nodeType：元素节点为1，属性节点为2，文本节点为3
       2. 节点层级：找不到返回null
          1. 父节点：.parentNode，返回节点
          2. 子节点：

.childNodes 返回所有子节点集合，包含元素节点，文本节点

.children 非标准但可使用，返回所有子元素节点集合

.firstChild / .lastChild 返回第一个/最后一个子节点，不管是文本节点还是元素节点

.firstElementChild / .lastElementChild 返回第一个/最后一个子元素节点（IE9以上，一般用children[0]）

* + - * 1. 兄弟节点：

.nextSibling / .previousSibling 获得下一个/上一个兄弟节点，包括元素节点、文本节点

.nextElementSibling / .previousElementSibling 获得下一个/上一个兄弟元素节点（IE9以上，一般自建函数使用）

* + - 1. 创建节点：
         1. document.createElement(‘标签名’) 返回节点
         2. document.write(‘HTML代码’) 直接将内容写入，如果文档流执行完毕，会导致回流
         3. 父元素.innerHTML = ‘HTML代码’ 直接将内容写入。创建多个元素效率更高（数组拼接）
      2. 添加节点：
         1. 节点1 .appendChild(节点2) 将节点2添加为节点1的最后一个子节点
         2. 节点1 .insertBefore(节点2，节点3) 将节点2添加到节点1的子元素节点3之前
         3. 节点1 .insertAdjacentHTML（position，text） 将text所代表的HTML元素添加到节点1的指定位置

position：

'beforebegin': 节点1之前 'afterend':节点1之后

'afterbegin':节点1内部的开头 'beforeend': 节点1内部的末尾

* + - * 1. 节点1 .insertAdjacentHTML（position，节点2） 将节点2添加到节点1的指定位置
      1. 删除节点：
         1. 节点1 .removeChild（子节点） 删除节点1的指定子节点，返回删除的节点
         2. ~~节点1 .remove（） 删除节点1~~
      2. 复制节点：
         1. 节点 .cloneNode（） 返回节点的副本，括号参数为空或false则只复制标签不复制内容。若为true，则完全复制

# **BOM**

* + 1. Browser Object Model：浏览器对象模型。缺乏标准
    2. 将浏览器视为一个对象，顶级对象为window
    3. window对象：是一个全局对象。所有全局变量都会变为window的一个属性。调用时可省略window.。一个特殊属性：window.name
       1. 属性：window .innerWidth
       2. 方法：window .scroll（x，y） 滚动窗口至文档指定位置
       3. 常见事件：
          1. document.addEventListener(‘DOMContentLoaded’, function(){}) 当DOM加载完成后执行，ie9+。在图片多的时候可用此事件
       4. 定时器：
          1. [window.]setTimeout（回调函数（）[，延迟的毫秒数]） 返回定时器id，设定完成后到指定时间调用函数。省略默认为0
          2. [window.]clearTimeout（timeoutID） 清除指定定时器
          3. [window.]setInterval（回调函数[，间隔的毫秒数]） 返回循环定时器id，间隔指定时间执行一次，间隔毫秒数默认为0
          4. [window.]clearInterval（IntervalID） 清除指定循环定时器
       5. location对象：
          1. 属性：

.href 整个URL .protocol 协议（最后带“:”） .host 域名（可能带“:端口号”）

.hostname 域名 .port 端口号 .pathname 路径（开头“/”）

.search 参数 .hash #及后面的URL片段标识符 .origin URL的协议、主机名与端口号

* + - * 1. 方法：

.assign() 跳转页面（重定向页面） .replace() 替换当前网页，不记录历史所以不能后退页面

.reload() 重新加载页面，若参数为true，则强制刷新

* + - 1. navigator对象：
         1. 属性：

.userAgent 浏览器用于 HTTP 请求的用户代理头的值的字符串

* + - 1. history对象：
         1. 方法：

history.back() 后退 .forward() 前进 go(参数) 前进后退，参数为1前进1个页面，为-1后退1个页面

1. 布局信息：
   1. 元素偏移量offset：获取位置，数值不带单位
      1. element.offsetParent 该元素带定位的父元素，若父级无定位则返回body
      2. .offsetTop / .offsetLeft 元素相对带定位父元素上方 / 左边框的偏移
      3. .offsetWidth / .offsetHeight 自身包括padding、边框、内容区的宽度 / 高度
      4. offset系列和style系列的区别：
         1. style相当于操作行内样式表，无法获得内嵌样式表的值；offset的值为无单位数值，style.width为带单位字符串
         2. offsetWidth为只读属性，style.width是可读写属性；因此需要获取元素大小位置用offset，需要更改元素值，用style
   2. 元素可视区client系列：获取大小，数值不带单位
      1. element.clientTop / .clientLeft 元素上/左边框的大小
      2. element.clientWidth / .clientHeight 自身包含padding、内容区的宽度/高度，不含边框
   3. 元素滚动scroll系列：数值不带单位
      1. element.scrollTop / .scrollLeft 被卷去上侧/左侧距离（元素溢出上界/左界的高度）
      2. window.pageYOffset / .pageXOffset 只读，页面被卷去上部/左侧的距离，ie9+
         1. 声明了DTD，使用document.documentElement.scrollTop 可读写
         2. 未声明DTD，使用document.body.scrollTop 可读写
      3. .scrollWidth / .scrollHeight 自身实际宽度/高度，不含边框（内容不溢出时等于盒子大小，溢出时为元素大小）
   4. 动画函数：原理：通过定时器setInterval不断改变元素位置
      1. 定时器可以直接作为调用对象的属性以优化性能、函数开始需要清除该元素原先的定时器
      2. 缓动效果：步长=（目标值-现在的位置）/10)，执行间隔15。步长大于零向上取整，步长小于零向下取整
      3. 可将函数传入动画函数，在最后执行，实现回调函数的效果

# **JSON**

* 1. JavaScript Object Notation（js对象表示法），是一个特殊格式的字符串，用来转换为任意语言的对象
  2. JSON表示的对象：var obj = ‘{“name”：”孙悟空”，”age”：”18”，”gender”：”男”}’； JSON中属性名必须加双引号
  3. JSON表示的数组：var arr = ‘[1, 2, 3, “hello”, true]’；
  4. JSON中允许的值：字符串、数值、布尔值、null、对象、数组
  5. 对象与JSON字符串转换（内置JSON对象）：
     1. JSON.parse(‘JSON字符串’) 返回转换出来的对象
     2. JSON.stringify(对象) 返回转换出来的JSON字符串
  6. 兼容ie8+，若要在ie7及以下浏览器使用，可从外部引入一个js文件，里面写上JSON的代码

# **常用插件**

* 1. laydate：日期控件插件
     1. 将need文件夹、skins文件夹、laydate.js导入项目。引用时只需引入laydate.js
     2. 皮肤切换：laydate.skin（‘xxx’）
     3. 日期控件创建：
        1. <input placeholder=“默认文字” class=“laydate-icon” onclick=“laydate（）”> 或
        2. <input class=“laydate-icon” id=“demo”>，在js代码：laydate（｛elem：‘#demo’｝）
        3. laydate函数的参数：｛
           1. elem：‘选择器’，min：laydate.now（-1），max：laydate.now（+2）， 允许的选择范围
           2. format：‘YYYY/MM’，festival：true，choose：function（datas）｛alert（‘得到’+datas）｝｝