**JavaScript**

目次

内容

[１. JS入门篇 6](#_Toc169760351)

[１.１ 操作符 6](#_Toc169760352)

[１.１.１ 算术运算符 6](#_Toc169760353)

[１.１.２ 赋值运算符 6](#_Toc169760354)

[１.１.３ 比较运算符 7](#_Toc169760355)

[１.１.４ 逻辑运算符 7](#_Toc169760356)

[１.１.５ 类型运算符 7](#_Toc169760357)

[１.１.６ 一元运算符 7](#_Toc169760358)

[１.１.７ 关系运算符 7](#_Toc169760359)

[１.１.８ 位移运算符 8](#_Toc169760360)

[１.１.１ 其他 8](#_Toc169760361)

[１.２ 数据类型 8](#_Toc169760362)

[１.２.１ 原始值 9](#_Toc169760363)

[１.２.２ 引用值 9](#_Toc169760364)

[１.２.３ 判断数据类型 9](#_Toc169760365)

[１.２.４ 类型转换 9](#_Toc169760366)

[１.２.５ 显示转换 9](#_Toc169760367)

[１.２.６ 隐式转换 9](#_Toc169760368)

[１.２.７ 类型签名 10](#_Toc169760369)

[１.３ 变量和常量 10](#_Toc169760370)

[１.３.１ 变量声明 10](#_Toc169760371)

[１.３.２ 变量作用域 10](#_Toc169760372)

[１.４ 语句 10](#_Toc169760373)

[１.４.１ 控制流 10](#_Toc169760374)

[１.４.２ 迭代器 11](#_Toc169760375)

[do...while 11](#_Toc169760376)

[for 11](#_Toc169760377)

[for...in 11](#_Toc169760378)

[for...of 11](#_Toc169760379)

[For...await...of 11](#_Toc169760380)

[while 11](#_Toc169760381)

[forEach 11](#_Toc169760382)

[１.４.３ 跳转语句 11](#_Toc169760383)

[１.４.４ 其他语句 11](#_Toc169760384)

[１.４.５ 声明语句 11](#_Toc169760385)

[１.５ 选择器 12](#_Toc169760386)

[２. JS初级篇 13](#_Toc169760387)

[２.１ 表达式 13](#_Toc169760388)

[２.２ 事件 13](#_Toc169760389)

[２.２.１ 事件本质 13](#_Toc169760390)

[２.２.２ 事件定义 13](#_Toc169760391)

[２.２.３ 事件冒泡 13](#_Toc169760392)

[２.２.４ 事件绑定 13](#_Toc169760393)

[２.２.５ 事件捕获 13](#_Toc169760394)

[２.２.６ 事件命名空间 13](#_Toc169760395)

[２.２.７ 动态事件 14](#_Toc169760396)

[２.２.８ 静态事件 14](#_Toc169760397)

[２.２.９ 事件使用 14](#_Toc169760398)

[２.２.１０ 事件原理 14](#_Toc169760399)

[２.２.１１ 事件提示 14](#_Toc169760400)

[２.２.１２ 异步事件 14](#_Toc169760401)

[２.２.１３ 事件循环 14](#_Toc169760402)

[２.３ 类(Class) 14](#_Toc169760403)

[２.３.１ 类声明 14](#_Toc169760404)

[２.３.１ 类概念 14](#_Toc169760405)

[２.３.２ 类表达式 15](#_Toc169760406)

[２.３.３ 命名类 15](#_Toc169760407)

[２.３.４ 类的继承 15](#_Toc169760408)

[２.３.５ 静态属性和静态方法 15](#_Toc169760409)

[２.４ 对象(Obejct) 15](#_Toc169760410)

[２.４.１ 对象数据类型 15](#_Toc169760411)

[２.４.２ 对象字面量 15](#_Toc169760412)

[２.４.３ 数据属性 15](#_Toc169760413)

[２.４.４ 访问器属性 16](#_Toc169760414)

[２.４.５ New操作符 16](#_Toc169760415)

[２.５ 函数(Function) 16](#_Toc169760416)

[２.５.１ 函数声明 16](#_Toc169760417)

[２.５.１ 函数调用 17](#_Toc169760418)

[２.５.２ 函数赋值 17](#_Toc169760419)

[２.５.３ 函数提升 17](#_Toc169760420)

[２.５.４ 函数属性 17](#_Toc169760421)

[２.５.５ 自执行函数 17](#_Toc169760422)

[２.５.６ 构造函数 17](#_Toc169760423)

[２.５.７ 默认参数 17](#_Toc169760424)

[２.５.８ …剩余参数 17](#_Toc169760425)

[２.５.９ Super 17](#_Toc169760426)

[２.５.１ this 17](#_Toc169760427)

[２.５.２ 参数 18](#_Toc169760428)

[２.５.３ 函数签名 18](#_Toc169760429)

[２.５.４ Lambda表达式(箭头函数=>) 18](#_Toc169760430)

[２.６ 数组(Array) 18](#_Toc169760431)

[２.６.１ 数组定义 18](#_Toc169760432)

[２.７ 正则表达式(RegExp) 18](#_Toc169760433)

[２.７.１ RegExp构造函数 18](#_Toc169760434)

[２.７.２ 字符 19](#_Toc169760435)

[３. JS中级篇 20](#_Toc169760436)

[３.１ Ajax 20](#_Toc169760437)

[３.１.１ 同域 20](#_Toc169760438)

[３.１.２ 跨域 20](#_Toc169760439)

[３.２ 原型链 20](#_Toc169760440)

[３.３ 迭代器 20](#_Toc169760441)

[３.４ Promise 20](#_Toc169760442)

[３.５ Monad 20](#_Toc169760443)

[３.６ 函数链 20](#_Toc169760444)

[３.７ 组合 20](#_Toc169760445)

[３.８ 闭包 20](#_Toc169760446)

[３.９ 劫持 20](#_Toc169760447)

[４. JS高级篇 21](#_Toc169760448)

[４.１ 异步编程 21](#_Toc169760449)

[４.１.１ 同步回调 21](#_Toc169760450)

[４.１.２ 异步函数 21](#_Toc169760451)

[４.２ 函数式编程 21](#_Toc169760452)

[４.２.１ 函数式思想 21](#_Toc169760453)

[４.２.２ 函数式编程 21](#_Toc169760454)

[４.２.３ 柯里化 22](#_Toc169760455)

[４.２.４ 函数记忆 22](#_Toc169760456)

[４.３ 数据结构 22](#_Toc169760457)

[４.３.１ 解构赋值 22](#_Toc169760458)

[４.３.２ 栈 22](#_Toc169760459)

[４.３.３ 队列 22](#_Toc169760460)

[４.３.４ 链表 22](#_Toc169760461)

[４.３.５ 集合 22](#_Toc169760462)

[４.３.６ 字典 22](#_Toc169760463)

[４.３.７ 散列表 22](#_Toc169760464)

[４.３.８ 树 22](#_Toc169760465)

[４.３.９ 图 22](#_Toc169760466)

[４.４ JSON 23](#_Toc169760467)

[Json.parse() 23](#_Toc169760468)

[Json.stringify() 23](#_Toc169760469)

[４.４.１ JSON值 23](#_Toc169760470)

[４.４.２ JSON数组 23](#_Toc169760471)

[４.５ 设计模式 23](#_Toc169760472)

[４.５.１ 工厂模式 23](#_Toc169760473)

[４.６ JS引擎和运行时 23](#_Toc169760474)

[４.６.１ JS引擎 23](#_Toc169760475)

[４.６.２ JS环境（运行时） 23](#_Toc169760476)

[５. JS技巧 24](#_Toc169760477)

[５.１ 控制台没有return的情况下输出undefined 24](#_Toc169760478)

[６. CSS入门篇 25](#_Toc169760479)

[６.１ CSS全称级联样式表有以下三种级联方式 25](#_Toc169760480)

[６.１.１ 外联式 25](#_Toc169760481)

[６.１.２ 内联式 25](#_Toc169760482)

[６.１.３ 嵌入式行内式（内联样式） 25](#_Toc169760483)

# JS入门篇

<https://zh.javascript.info/>

<https://vscode.dev/>

<https://www.runoob.com/manual/jquery/>

语雀：<https://www.yuque.com/yuejiangliu/dotnet/timothy-csharp-017>

插件：monokai pro；

|  |  |
| --- | --- |
| 模块 | 模块化：是从代码的角度分析的，把一些可复用的代码，抽离为单个模块，便于项目的维护和开发。可以调用组件来组成模块，多个模块可以组合成业务框架。  组件：组件是Web前端开发中的基本单元，可以是任何一个可独立使用和复用的元素，如按钮、输入框、导航栏等。组件具有封装性、复用性和扩展性，可以减少代码冗余，提高开发效率。  组件化思维：组件化思维是一种以组件为中心的开发模式，将页面看作是由多个可独立运行的组件构成的。开发人员通过定义和组合各种组件，可以快速构建出复杂的页面和应用。 |
| 组件 | 组件化：从UI界面的角度分析的，把一些可复用的UI元素，抽离为单个组件，便于项目的维护和开发。多个组件可以组合成组件库，方便调用和复用，组件间也可以嵌套，小组件组合成大组件。  将一个复杂的程序依据一定的规则(规范)封装成几个块(文件), 并进行组合在一起。  块的内部数据与实现是私有的, 只是向外部暴露一些接口(方法)与外部其它模块通信。 |
|  |  |

## 操作符

### 算术运算符

|  |  |
| --- | --- |
| + | 加法 |
| - | 减法 |
| \* | 乘法 |
| / | 除法 |
| % | 取模（余数） |
| ++ | 递加 |
| -- | 递减 |
| ()圆括号运算符/函数运算符/分组运算符 | 用于控制表达式中的运算优先级。如果表达式放在圆括号中，作用是求值；如果跟在函数后面，作用是调用函数；如果值在括号中表示返回这个值。 |

### 赋值运算符

|  |  |
| --- | --- |
| = | 赋值运算符 |
| += | 赋值求和 |
| -= | 赋值求差 |
| \*= | 赋值乘积 |
| /= | 赋值商 |
| %= | 赋值求余 |
| <<= | 左位移 |
| >>= | 右位移 |
| >>>= | 无符号右位移 |
| &= | 赋值与 |
| ^= | 赋值按位异或 |
| |= | 赋值或 |
| &&= | 逻辑和赋值运算符 |
| ||= | 逻辑或赋值运算符 |
| ?? | 逻辑空赋值运算符 |

### 比较运算符

|  |  |
| --- | --- |
| == | 等于 |
| === | 等值等型 |
| != | 不相等 |
| !== | 不等值或不等型 |
| > | 大于 |
| < | 小于 |
| >= | 大于或等于 |
| <= | 小于或等于 |
| ? | 三元运算符 |

### 逻辑运算符

|  |  |
| --- | --- |
| && | 逻辑与 |
| || | 逻辑或 |
| ! | 逻辑非 |

### 类型运算符

|  |  |
| --- | --- |
| typeof | 返回变量的类型。 |
| instanceof | 返回true，如果对象是对象类型的实例。 |
| void | 无论void后的表达式是什么，void操作符都会返回undefined |
| await |  |

### 一元运算符

一元运算符只有一个操作数

|  |  |
| --- | --- |
| delete | delete运算符用来删除对象的属性。 |
| void | void运算符表示表达式放弃返回值。 |
| typeof | typeof运算符用来判断给定对象的类型。 |
| + | 一元加运算符将操作转换为Number类型 |
| - | 一元减运算符将操作转换为Number类型并取反。 |
| ~ | 按位非运算符。 |
| ! | 逻辑非运算符 |

### 关系运算符

比较运算符比较二个操作数并返回基于比较结果的Boolean值。

|  |  |
| --- | --- |
|  |  |
| in | in操作符用于检查对象的属性是否存在，包括对象的原型链上的属性。 |
| instanceof | instanceof运算符判断一个对象是否是另一个对象的实例。 |

### 位移运算符

在二进制的基础上对数字进行移动操作

|  |  |
| --- | --- |
| << | 按位左移运算符 |
| >> | 按位右移运算符。 |
| >>> | 按位无符号右移运算符。 |
| & | 按位与 |

### 其他

|  |  |
| --- | --- |
| 空值合并运算符?? | a??b的结果是：如果a是已定义的，则结果为a，如果a不是已定义的，则结果为b。  通常??的使用场景是，为可能是未定义的变量提供一个默认值。 |
| 扩展运算符… | ...arr返回一个解压的数组。展开操作符够在函数的参数中或者数组，对象中使用。 |
| ()转型操作符/方法调用操作符 | 可以显示转换数据类型/调用方法；如果值在括号中表示返回这个值；  括号的二义性； |
| new | 主要是在内存当中创建类型的实例，并调用类的实例构造器，如果在类的左边有赋值符号（=）把内存地址交给负责访问的变量。还可以调用实例的初始化器，通过花括号可以为实例属性赋值。Form form =new Form(){};但是没有使用变量引用时，例如：new Form().showDialog();在执行后，垃圾回收器会自动回收，并且在后面的语句都无法在调用这个实例。  new 类名().对象成员，这种方式是在通过new关键字创建实例对象的同时就访问了对象的某个成员，并且在创建后只能访问其中某一个成员，而不能像对象引用那样可以访问多个对象成员。同时，由于没有对象引用的存在，在完成某一个对象成员的访问后，该对象就会变成垃圾对象。所以，在实际开发中，创建实例对象时多数会使用对象引用。 |

## 数据类型

|  |  |  |
| --- | --- | --- |
| 类别 |  | 说明 |
| 原始值 | 基本类型 | 字符串(String) |
| 数字(Number)NaN |
| 布尔(Boolean) |
| 未定义(Undefined) |
| 符号Symbol |
| Bigint(可以表示任意大的整数) |
| 空指针对象(Null) |
| 引用值 | 对象类型 | 所有其他类型的最终基类：object |
| 正则类型 | RegExp(…)形式的用户定义的类型 |
| 数组类型 | 一维和多维数组，Array |
| 函数类型 | function(...)形式的用户定义的类型 |
| 日期类型 | Date |

### 原始值

原始值是没有实例的。原始值只需要一段单独的内存，用于存储实际的数据。原始值就是最简单的数据。

原始值的变量直接包含它们的数据，对于原始值，每个变量都有它们自己的数据副本，因此对一个变量的操作不可能影响另一个变量。原始值是按值比较的，即在比较两个字符串时，当且仅当这两个字符串长度相当并且每个索引的字符也相同时js才认为相等。

### 引用值

引用值则是由多个值构成的对象；引用值变量里存储的数据是对象的内存地址。引用值所有需要给它分配堆上的内存。如果没有进行初始化，就是一个空引用。引用值需要两段内存，第一段存储实际的数据，它总是位于堆中。第二段是一个引用，指向数据在堆中的存放位置。而引用值的变量存储对它们的数据的引用，后者称为对象。对于引用值，两个变量可能引用同一个对象(指针)，因此对一个变量的操作可能影响另一个变量所引用的对象。

### 判断数据类型

|  |  |
| --- | --- |
| Typeof | 返回的数据类型是字符串，可以区别:数值,字符串,布尔值,undefined,function;不能区别:null与对象,一般对象与数组 |
| Instanceof | 判断对象类型;专门用来判断对象数据的类型:Object,Array与Function |
| ===(权等) | 可以判断:undefined和null。 |

Undefined和null的区别：undefined代表定义未赋值，null代表定义并赋值,只是值为null。使用typeof判断null返回的是object类型。初始赋值为null，表明将要赋值为对象。最后在把变量赋值为null，表示释放内存。让变量指向的对象被垃圾回收器回收。

### 类型转换

|  |  |
| --- | --- |
| parseInt() | 把值转换成整数 |
| parseFloat() | 把整数转换成浮点数 |
| Number() | 任何数据类型进行转换 |

### 显示转换

显式类型转换，又叫强制类型转换。与隐式转换正好相反，显式转换需要用户明确地指定转换的类型。显式转换包括所有的隐式转换，也就是说把任何编译器允许的隐式转换写成显式转换的形式都是允许的。用()实现显示转换，这表示，把转换的目标类型名放在要转换的值之前的圆括号中。提醒：

（1)、显式转换可能会导致错误。进行这种转换时编译器将对转换进行溢出检测。如果有溢出说明转换失败，就表明源类型不是一个合法的目标类型。无法进行类型转换。

(2)、强制类型转换会造成数据精度丢失。

### 隐式转换

隐式转换就是编译器默认的、不需要加以声明就可以进行的转换。隐式转换不要求在源代码中使用任何特殊语法，编译器自动执行隐式强制转换。在隐式转换过程中，编译器无需对转换进行详细检查就能够安全地执行转换。隐式强制转换也称为“扩展转换“，因为要将窄数据类型转换为宽数据类型，且还将确保不会在转换过程中丢失数据。在进行隐式转换使用赋值符号（n=i）;（注意：（1）转换前后的类型必须相兼容 (例如：int和double （2)、隐式数值转换实际上就是从低精度的数值类型到高精度的数值类型的转换，即小的类型转大的类型）；子类向父类转换，不丢失精度的转换。

### 类型签名

## 变量和常量

### 变量声明

变量定义没有进行初始化，默认值为undefined。在js中，作用域涉及到变量的作用范围。在函数外定义的变量具有全局作用域。 这意味着，具有全局作用域的变量可以在代码的任何地方被调用;

|  |  |
| --- | --- |
| var | 声明一个变量，可同时将其初始化为一个值。变量没有使用var声明时，变量默认为全局变量。  在函数中不使用var声明的变量都是全局变量，使用var的变量是局部变量。var在运行之前类型就确定了。Var是函数作用域，块级作用域是函数作用域的子集。 |
| let | 声明一个块级本地变量，可同时将其初始化为一个值。Let变量不可以重复声明，不重在变量提升。块级作用域【{}】不影响作用域链。Let不存在变量提升，因此在声明前调用被称为【暂时性死区】，在此阶段调用任何后面才什声明的变量都会抛出ReferenceError。Let在全局作用域中声明不会成为windows对象的属性，不过let声明仍然是在全局作用域中发生的，相应变量会在页面的我生命周期内存续(作用域没有发生改变，生命周期提升为全局，不允许在声明同名变量)。 |
| const | 声明一个只读的命名常量，声明必须赋初始值。变量不会被声明提前。不允许重复声明，初始化后不允许赋值。对象属性修改和数组元素变化不会触发const错误。 |
| 模版字符串 | `${}` |
| funtion | 函数声明会创建一个函数对象 |
| Class | Class声明会创建一个新类并为其赋予一个名字 |

### 变量作用域

#### 变量生命周期

变量的生存期是指变量在计算机存在的有效时间，全局变量在主程序中定义，其有效范围从其定义开始，一直到本程序结束为止。局部变量在程序的函数中定义，其有效范围只在改函数之中。函数结束后，局部变量的生存期也就结束了。每调用一次就会创建一个新的函数作用域，它们是互相独立的。

#### 变量查找

Var关键字定义的变量是全局变量，调用变量时遵循就近原则，例如调用name的变量时在当前代码行找不到时，就会往上查找，直到找到为止。(变量作用域，调用方式，js编译方式，变量声明)，变量声明没有使用var操作符默认为全局变量。Var调用时会自动提升到当前域的最顶端，可以理解为在调用之前就进行声明并且赋值(默认值)。

## 语句

### 控制流

|  |  |
| --- | --- |
| return | 退出函数并返回函数的值 |
| continue | 终止执行当前或标签循环的语句，直接执行下一个迭代循环。 |
| break | 退出最内部循环、switch或有名字的闭合语句 |
| Block | 一个块语句可以用来管理零个或多个语句。该区块是由一对大括号分隔。 |
| Empty | 空语句用来表示没有语句的情况，尽管JavaScript语法期望有语句提供。 |
| if...else | 如果指定的条件是true，则执行相匹配的一个语句，若为false，则执行另一个语句。 |
| switch:case:brack;default:brack; | 计算表达式，将子句于表达式的值做匹配，执行与该值相关联的语句。 |

### 迭代器

|  |  |
| --- | --- |
|  |  |
| do...while | 创建一个循环来执行语句，直到该语句条件表达式的值为false。先执行语句，再执行条件表达式，该语句至少会执行一次。 |
| for | 创建一个由3个的表达式组成的循环，该循环用括号包裹，分号分割，并在循环体中执行语句。 |
| for...in | 无序遍历对象的可枚举属性。语句针对每个唯一的属性 |
| for...of | 遍历可迭代的对象（包括数组、类数组对象、迭代器和生成器)，对每个不同属性的属性，调用一个自定义的有执行语句的迭代钩子。 |
| For...await...of | 在异步可迭代对象、类数组对象、迭代器和生成器上迭代，调用自定义迭代钩子，其中包含要为每个不同属性的值执行的语句。 |
| while | 创建一个循环语句，循环会一直持续到该语句条件表达式的值为false。先执行条件表达式，然后执行语句。 |
| forEach | 在js中用于调用数组的每个元素，并将元素传递给回调函数。 |

### 跳转语句

|  |  |
| --- | --- |
| break | 终止并跳出循环体。 |
| continue | 终止当前循环，重新开始一个新的循环。 |
| return | 跳出循环及其包含的函数。Return可以退出一个返回值 |
| Yield | 提供一个被迭代的值；只用在生成器函数中 |
| throw new | 抛出一个用户定义的异常。 |
| try{} catch(){} **finally{}** | 处理异常和代码清理 |

### 其他语句

|  |  |
| --- | --- |
| with | 扩展作用域链。(在严格模式下被废弃并禁止) |
| debugger | 当 debugger 被调用时，执行暂停在 debugger 语句的位置。 |
| “use strict”指令 | 严格模式是一种不同于js解析和执行模型。在这种模式下不规范的写法会被处理，不安全的活动会被抛出错误，在整个脚本启用严格模式，在脚本开头加上“use strict”，也可以指定单独函数为严格模式，在函数体内的开头即可。 |

### 声明语句

|  |  |
| --- | --- |
| funtion | 函数声明会创建一个函数对象 |
| Class | Class声明会创建一个新类并为其赋予一个名字 |
| Import | 为其他模块中定义的值声明名字 |
| export | 声明可以被导入其他模块的值 |
| import echarts from 'echarts'；  import 'iview/dist/styles/iview.css';  <style>  @import './test.css';  </style> | 引入第三方插件；  导入 css文件；  如果是在.vue文件中那么在外面套个style；  import引入文件路径  import引入一个依赖包，不需要相对路径。如：import app from ‘app’;  import 引入一个自己写的js文件，是需要相对路径的。如：import app from ‘./app.js’; |
| export default | 1.export与export default均可用于导出常量、函数、文件、模块等  2.在一个文件或模块中，export、import可以有多个，export default仅有一个  3.通过export方式导出，在导入时要加{ }，export default则不需要{ } |

## 选择器

|  |  |
| --- | --- |
| 基本选择器 | |
| ID选择器（#） | 在ID名前面加上#-$(“#ID”) |
| 类选择器（.） | 英文小圆点开头.-$(“.Class”) |
| 元素/标签选择器 | $(“a”) |
| 通配选择器（\*） | 匹配HTML中所有标签元素 |
| 层级选择器 | |
| 后代选择器（空格） | 子选择器（>） |
| 相邻选择器（+） | 兄弟选择器（~） |

# JS初级篇

## 表达式

一个表达式由操作数与操作符组成，操作数可以由变量、常量或者另一个表达式表示。

## 事件

事件的功能=通知+可选的事件参数（即详细信息）

事件模型的五个组成部分=事件的拥有者（event source，对象）→事件成员（event，成员）→事件响应者（event subscriber，对象）→事件处理器（event handler，成员）=本质上是一个回调方法→事件订阅=把事件处理器与事件关联在一起，本质上是一种以委托类型为基础的“约定”。

注意=事件的处理器是方法的成员；挂接事件处理器的时候，可以使用委托实例，也可以直接使用方法，这是“语法糖“；事件处理器对事件的订阅不是随意的，匹配与否由声明事件时所用的委托类型来检测；事件可以同步调用也可以异步调用。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事件的订阅者 | 事件消息的接收者 | 事件的响应者 | 事件的处理者 | 被事件所通知的对象 |
| 事件参数 | 事件信息 | 事件消息 | 事件数据 |  |

事件流指事件发生顺序。早绑定指的是编译器在编译时就知道这个成员是属于哪个类或者对象，晚编译指的是程序调用到成员时才知道它属于哪个类或者对象,这种语言叫做动态语言(Javascript)。事件的功能=通知+可选的事件参数（即详细信息）

### 事件本质

事件的本质是一个蒙版。

蒙板 Mask： 事件这个包装器对委托字段的访问起限制作用，只让你访问 +=、-= ，让你只能给事件添加或移除事件处理器。让程序更加安全更好维护。

封装（encapsulation）的重要功能就是隐藏，事件对外界隐藏了委托实例的大部分功能，仅暴露添加，移除事件处理器的功能。

### 事件定义

单词Event，译为“事件“；

《《牛津词典》》中的解释是“a thing that happens，especially something important“；

通顺的解释就是“能够发生的什么事情“；

### 事件冒泡

事件的冒泡是指，父子元素同时监听同一个事件。当触发子元素的事件的时候，同一个事件也被传递到了父元素的事件里去响应。因此在事件冒泡的概念下在p元素上发生click事件的顺序应该是p->div->body->html->document。

在冒泡中，内部元素的事件会先被触发，然后再触发外部元素，即：<p>元素的点击事件先触发，然后会触发<div>元素的点击事件。（由内而外）

### 事件绑定

使用对象.事件=函数的形式绑定响应函数，它只能同时为一个元素的一个事件响应绑定函数。

不能同时绑定多个，如果同时绑定多个事件，后边会覆盖掉前面。

### 事件捕获

与事件冒泡相反，事件会从最外层开始发生，直到最具体的元素。

事件捕获的概念下在p元素上发生click事件的顺序应该是document->html->body->div->p。

在捕获中，外部元素的事件会先被触发，然后才会触发内部元素的事件，即：<div>元素的点击事件先触发，然后再触发<p>元素的点击事件。（由外而内）

### 事件命名空间

事件命名空间就是在事件类型后面以点语法附加一个别名，以便引用事件。如click.a，其中a就是click事件的别名，即事件命名空间。这样删除事件时，可以直接指定命名空间即可。

### 动态事件

动态注册，需要我们先获取到标签对象。然后通过对标签对象的的属性进行赋值一个function函数的形式。

### 静态事件

就是在标签上使用事件属性赋值的形式给这个标签的事件响应添加javaScript代码的方式，我们称之为静态注册事件。

### 事件使用

用于对象或者类间的动作协调与信息传递（消息推送）

### 事件原理

事件模型（Event model）中的两个“5“

“发生→响应中的5个部分“＝闹钟响了你起床，孩子饿了你做饭．．．这里隐含着订阅关系；

“发生→响应中的5个动作“＝（１）我有一个事件→（２）一个人或者一群人关心者我的这个事件→（３）我的这个事件发生了→（４）关心这个事件的人会被依次通知到→（５）被通知到人根据拿到的事件信息（又称＂事件数据＂，＂事件参数＂，＂通知＂）对事件进行响应（又称＂处理事件＂）

### 事件提示

事件多用于桌面，手机等开发的客户端编程，因为这些程序经常是用户通过事件来＂驱动＂的；

各种编程语言对这个机制的实现方法不尽相同；

Java语言里没有事件这种成员，也没有委托这种数据类型。Java的“事件”是使用接口来实现的。

MVC，MVP，MVVM等模式，是事件模式更高级，更有效的“玩法”；

日常开发的时候，使用已有事件的机会比较多，自己声明事件的机会比较少，所以先学使用。

内容决定形式，功能决定方法。

### 异步事件

### 事件循环

## 类(Class)

类是一种数据结构，它可以包含数据成员（常量和字段）、函数成员（方法、属性、事件、索引器、运算符、实例构造函数、静态构造函数和析构函数）以及嵌套类型。类类型支持继承，继承是一种机制，它使派生类可以对基类进行扩展和专用化。 —— 《C# 语言规范》

注：这是在描述类是什么，讲的是类的外延而不是类的内涵。

计算机领域的类有下面三个方面

是一种数据结构（data structure）；一种数据类型；代表现实世界中的“种类”

类是最基础的C#类型。类是一个数据结构，将状态（字段）和操作（方法和其他函数成员）组合在一个单元中。类为动态创建的类实例（instance）提供了定义，实例也称为对象（object）。类支持继承（inheritance）和多态（polymorphism），这是一个派生类（derived class）可用来扩展和专用化基类（base class）的机制。js类中必须创建constructor() {}函数。

### 类声明

|  |  |
| --- | --- |
| Class Example{}; | 全局类 |

### 类概念

类是一种数据结构（data structure），是一种抽象数据结构。类本身是一种抽象的结果（概念），比如现实世界当中有学生，从中抽取一些数据（ID，name），把它们封装在一起，形成学生类，而且学生类里面有学生的成员。

类本身还是抽象结果的载体。类本身还是抽象数据和抽象行为的载体。抽象行为：考试，行为等，考试，学习是学生的行为，抽象出考试和学习的方法后，封装到类里面成为类的方法成员。

类是面向对象核心的概念，也承载了抽象编程的方法，比如说多态和继承。所以类在面向对象编程中是起到枢纽和支点作用的数据结构；

类是一种引用类型，具体到每一个类上，每一个类都是一个自定义类型。（每一个类都是自定义的引用类型），可以拿类去声明变量，拿这个类去创建实例，当拿这个类去创建实例的时候，这个类又体现出另一种功能，类可以被视为实例的模版，拿这个类像模版一样创建一个个实例；

类代表现实世界中的＂种类＂（概念），程序当中的这个类，在一定程度上代表现实当中的某些种类。（这个时候程序当中类的概念与数学和哲学类的概念最接近）

### 类表达式

|  |  |
| --- | --- |
| Let Example = class{} |  |

### 命名类

|  |  |
| --- | --- |
| let Example=class Example{} |  |

### 类的继承

Extends，class

#### 原型链继承

原型式继承可以无须明确定义构造函数而实现继承，本质上是对给定对象执行浅复制。这种操 作的结果之后还可以再进一步增强。

#### 借用构造函数继承

#### 原型链+借用构造函数的组合继承

#### 组合继承优化1

#### 组合继承优化2

#### ES6中class的继承

### 静态属性和静态方法

通常，静态方法用于实现属于该类但不属于该类任何特定对象的函数。

## 对象(Obejct)

### 对象数据类型

对象数据类型比原生数据类型强大了不少，原生数据类型，比如数值型、浮点型、布尔型等都只能存放一些直接量，也就是说单一的数据。而对象数据类型却是可以存放一大堆数据的集合，是以键值对的方式集合。

键名可允许不加，或加单引号，或加双引号都可以；值没有限制；最后一个属性后面允许有逗号；

|  |  |
| --- | --- |
| Var obj={}；var Person={name:"Jason"}  等于Var obj=new Object(); | 空对象，字面量对象 |
| Var obj=new Object(); | 空构造对象 |
| Var fon=new Function(); | 空构造函数 |
| Function name(){} | 声明式全局函数 |
| Var name = function(){} | 表达式函数(匿名函数) |

### 对象字面量

对象字面量是对象定义的简写形式，目的是为了简化包含大量属性的对象的创建。

### 数据属性

in 操作符用于检查对象的属性是否存在，包括对象的原型链上的属性。要判断属性是否是对象自身的而非原型链上的，可以结合 hasOwnProperty 使用。Object.keys 方法：Object.keys 方法返回一个数组，包含对象自身可枚举属性的名称。在js对象中数据属性是在花括号中{}，格式为属性：属性值，由冒号来分隔键/值;

### 访问器属性

访问器属性（accessor property）。它们本质上是用于获取和设置值的函数，但从外部代码来看就像常规属性。

|  |  |
| --- | --- |
| getter | 将对象属性绑定到查询该属性时将被调用的函数。 |
| setter | 当尝试设置属性时，set 语法将对象属性绑定到要调用的函数。 |

### New操作符

主要是在内存当中创建类型的实例，并调用类的实例构造器，如果在类的左边有赋值符号（=）把内存地址交给负责访问的变量。还可以调用实例的初始化器，通过花括号可以为实例属性赋值。Form form =new Form(){};但是没有使用变量引用时，例如：new Form().showDialog();在执行后，垃圾回收器会自动回收，并且在后面的语句都无法在调用这个实例。

new 类名().对象成员，这种方式是在通过new关键字创建实例对象的同时就访问了对象的某个成员，并且在创建后只能访问其中某一个成员，而不能像对象引用那样可以访问多个对象成员。同时，由于没有对象引用的存在，在完成某一个对象成员的访问后，该对象就会变成垃圾对象。所以，在实际开发中，创建实例对象时多数会使用对象引用。

## 函数(Function)

函数声明：可以在函数声明之前去调用该函数。它会在所有代码执行之前就创建好了。

函数表达式：不允许在函数声明之前去调用。变量会声明提前，但是没有赋值，把函数赋值给变量。

函数是一种叫做function引用值的实例，因此函数是一个对象。对象是保存在内存中的，函数名则是指向这个对象的指针。如果函数名后面加上圆括号（带括号）就表示立即调用（执行）这个函数里面的代码（花括号部分的代码）。

不加括号的，都是把函数名称作为函数的指针，一个函数的名称就是这个函数的指针，此时不是得到函数的结果，因为不会运行函数体代码。它只是传递了函数体所在的地址位置，在需要的时候好找到函数体去执行。函数是任何可调用且可通过()操作求值的表达式。函数会向调用者返回一个经过计算的值或是 undefined（无值函数）。

在把函数定义为对象属性时，该函数为方法。

### 函数声明

#### 声明式函数(具名函数)

|  |  |
| --- | --- |
| function name(){} | 全局函数 |

代码执行前被加载到作用域中，函数声明会给函数一个指定的名字。

#### (字面量⇒匿名函数)

|  |  |
| --- | --- |
| Var name = function(){} | 调用只能写在函数创建后面 |

代码执行到那一行的时候才会有定义。函数表达式则是：创建一个匿名函数，然后将这个匿名函数赋给一个变量。

#### New Function(构造函数)

|  |  |
| --- | --- |
| Var fy=new Function(); | 构造函数 |

构造函数中可以添加一些成员，可以在构造函数本身上添加，也可以在构造函数内部的this上添加。通过这两种方式添加的成员，就分别称为静态成员和实例成员。

#### 静态成员

静态成员：在构造函数本上添加的成员称为静态成员，只能由构造函数本身来访问。

静态成员在构造函数本身上添加的成员（函数名.静态成员）

#### 实例成员

实例成员：在构造函数内部创建的对象成员称为实例成员，只能由实例化的对象来访问。

实例成员就是构造函数内部通过this添加的成员。

### 函数调用

在 js中，调用函数的过程中;加括号：表示想要得到函数的执行结果，此时在调用的时候函数体已经执行，返回函数值。此时表示重新开了一个线程，实现多线程的运行。函数体；不加括号：表示得到函数的指针，转到函数定义的位置去执行这个函数，仍然是单线程。函数名；加括号：如果函数返回对象，那么基于这个函数调用的返回值还可以继续调用其他方法。这样就会得到表现为一个表达式的一系列方法调用（或函数调用链）。在js中方法调用是对象点函数名()PS：把函数赋值给对象的属性叫方法；函数调用是函数签名();

### 函数赋值

函数赋值就是把函数赋值给变量。一种是函数名()赋值，在没有return返回值的情况下，显示undefind，在有return的情况下显示的是返回值。另一种是函数名赋值，直接把整个函数赋值给变量，可以理解为变量就是函数本身的对象。

### 函数提升

js中创建函数有两种方式：函数声明式(具名函数)和函数字面量式(匿名函数)。只有函数声明才存在函数提升。

无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最前端，这个过程被称为变量（函数）提升。

### 函数属性

### 自执行函数

自执行函数在调用上与普通函数一样，可以匿名，可以传参。只不过是在声明的时候自调用了一次。在表达式后加上括号会立即执行。

|  |  |
| --- | --- |
| (function(x,y){returnx+y;})(3,4); | (function(){})(); |
| (function(){}()); | (functionfoo(){})(); |
| (functionfoo(){}()); |  |

### 构造函数

自定义构造函数，以函数的形式为自己的对象类型定义属性和方法。

构造函数和工厂模式区别：没有显式地创建对象。属性和方法直接赋值给了 this。没有 return。

要创建 构造函数的实例，应使用 new 操作符。以这种方式调用构造函数会执行如下操作: 在内存中创建一个新对象。这个新对象内部的[[Prototype]]特性被赋值为构造函数的 prototype 属性。

构造函数内部的 this 被赋值为这个新对象（即 this 指向新对象）。

执行构造函数内部的代码（给新对象添加属性）。如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

### 默认参数

函数默认参数允许在没有值或undefined被传入时使用默认形参;

### …剩余参数

剩余参数语法允许我们将一个不定数量的参数表示为一个数组。

### Super

用于访问对象字面量或类的原型（[[Prototype]]）上的属性，或调用父类的构造函数。

### this

#### 副作用

使用this的缺点是它给予了超出方法作用域的实例层级的数据访问能力，从而可能导致副作用。只要和外部环境发生一定交互的都是副作用。更改文件系统，往数据库插入记录，发送一个 http 请求，可变数据，打印/log，获取用户输入，DOM 查询，访问系统状态;改变一个全局的变量、属性或数据结构。改变一个函数参数的原始值。处理用户输入。抛出一个异常，除非它又被当前函数捕获了。屏幕打印或记录日志。查询 HTML 文档、浏览器的 cookie或访问数据库。在js中嵌套函数被当作方法来调用，那么它的this值就是调用它的对象。如果嵌套函数(不是箭头函数)被当作函数来调用，则它的this值要么是全局对象(非严格模式)，要么是undefined(严格模式)。

### 参数

js中所有函数的参数都是按值传递的，函数参数是局部变量。这意味着函数外的值会被复制到函数内部的参数中，就像从一个变量复制到另一个变量一样。如果是原始值，那么就跟原始值变量的复制一样，如果是

引用值，那么就跟引用值变量的复制一样。

在按值传递参数时，值会被复制到一个局部变量。

### 函数签名

重载方法之间的方法名是相同的，那么我们势必要从构成方法的其他几个要素中找到另一个要素与方法名组成能够唯一标示方法的签名，方法体当然不予考虑。那么就是形参列表和返回值了，但是由于对于调用方法的人来说，方法的形参数据类型列表的重要程度要远远高于返回值，所以方法签名就由方法名+形参列表构成，也就是说，方法名和形参数据类型列表可以唯一的确定一个方法，与方法的返回值一点关系都没有，这是判断重载重要依据。

### Lambda表达式(箭头函数=>)

Lambda表达式(箭头函数)用于表示一个函数，所以它和函数一样，也拥有参数、返回值、函数体，但它没有函数名，所以Lambda表达式相当于一个匿名函数。

1)如果形参只有一个，则小括号可以省略，2)函数体如果只有一条语句，则花括号可以省略，函数的返回值为该条语句的执行结果，3)箭头函数this指向声明时所在作用域下this的值

4)箭头函数不能作为构造函数实例化，5)不能使用arguments

箭头函数适合纯函数的操作，比如map、filter、reduce等操作

箭头函数内部，一般不要使用this，因为this是和外部作用域绑定的，容易产生问题。

如果函数体不是只一行，应该用花括号，并显式地返回（如果需要返回值）

如果函数体只一行，可以忽略花括号，可以隐式地返回。

## 数组(Array)

数组时内置对象，数组可以保存复数的值。调用时通过索引调用。数组可以添加任意值。使用逗号分隔。

有中括号的是索引器运算符【[]】。数组初始化/字面量语法。[]元素访问操作符。["Symbol(Symbol.iterator)"]

Array[Symbol.iterator](); 逗号分隔是水平数组，分号分隔是垂直数组。

在js中数组有关联数组和索引数组。索引数组取值通过索引array.[0]，关联数组(对象)取值通过属性名array[“性别”]。

### 数组定义

创建数组时，并不对其进行初始化，length为1。

|  |  |
| --- | --- |
| var array=new Array(); | 空构造数组 |
| var arr=[]; | 空数组，字面量数组 |
| var name =[[]] | 多维数组 |

## 正则表达式(RegExp)

### RegExp构造函数

|  |  |
| --- | --- |
| var name=new RegExp(字符串); | Var name=/at/i; |
| var name=new RegExp(正则表达式); |  |

### 字符

|  |  |
| --- | --- |
| \转义 | ^ |
| \*通配符0个或多个 | ? 非贪婪匹配如果紧跟在任何量词 \*、 +、? 或 {} 的后面，将会使量词变为非贪婪（匹配尽量少的字符）  匹配前面一个表达式 0 次或者 1 次。等价于 {0,1}。 |
| $匹配输入的结束。如果多行标志被设置为 true，那么也匹配换行符前的位置。 | . （小数点）默认匹配除换行符之外的任何单个字符。 |
| (x)捕获组和反向引用 | +匹配前面一个表达式 1 次或者多次。等价于 {1,} |

# JS中级篇

## Ajax

Ajax

### 同域

### 跨域

## 原型链

## 迭代器

## Promise

## Monad

## 函数链

## 组合

## 闭包

闭包是一个环境和词法的组合，闭包具体是指函数作用域，这个环境包含了这个闭包创建时所能访问的所有局部变量。在js中父对象的所有变量，对子对象都是可见的，反之则不成立。因此外部函数（域）访问内部函数（域）需要特殊的环境也就是闭包（跨域）。

## 劫持

函数劫持，顾名思义，即在一个函数运行之前把它劫持下来，添加我们想要的功能。当这个函数实际运行的时候，它已经不是原本的函数了，而是带上了被我们添加上去的功能，这也是我们常见的 钩子函数 的原理之一。

一般的劫持原理都是一个思路：

1.使用新的变量保存即将被劫持的函数。

2.改写被劫持函数的功能。

3.在被劫持函数的末尾段（或者其他适当部位）重新调用重写之前的函数。

# JS高级篇

## 异步编程

异步函数通常可以分为两大类：I/O 函数和计时函数

### 同步回调

### 异步函数

调用一个函数时，程序只在该函数返回之后才能继续。JavaScript 写手如果称一个函数为“异步的”，

其意思是这个函数会导致将来再运行另一个函数，后者取自于事件队列（若后面这个函数是作为参数传递给前者的，则称其为回调函数，简称为回调）。

## 函数式编程

函数式编程是指为创建不可变的程序，通过消除外部可见的副作用，来对纯函数的声明式的求值过程。

函数式编程（FP）通过最小化变化使得代码更易理解。面向对象编程（OO）通过封装变化使得代码更易理解。

可扩展性——我是否需要不断地重构代码来支持额外的功能？易模块化——如果我更改了一个文件，另一个文件会不会受到影响？

可重用性——是否有很多重复的代码？可测性——给这些函数添加单元测试是否让我纠结？易推理性——我写的代码是否非结构化严重并难以推理？

### 函数式思想

函数式编程基于一个前提，即使用纯函数构建具有不变性的程序。纯函数具有以下性质。

仅取决于提供的输入，而不依赖于任何在函数求值期间或调用间隔时可能变化的隐藏状态和外部状态。不会造成超出其作用域的变化，例如修改全局对象或引用传递的参数。

|  |  |  |
| --- | --- | --- |
|  | 函数式 | 面向对象 |
| 组合单元 | 函数 | 对象（类） |
| 编程风格 | 声明式 | 命令式 |
| 数据和行为 | 独立且松耦合的纯函数 | 与方法紧耦合的类 |
| 状态管理 | 将对象视为不可变的值 | 主张通过实例方法改变对象 |
| 程序流控制 | 函数与递归 | 循环与条件 |
| 线程安全 | 可并发编程 | 难以实现 |
| 封装性 | 因为一切都是不可变的，所以没有必要 | 需要保护数据的完整性 |

### 函数式编程

命令式计算数组的平方；函数式计算数组的平方；lambda表达式计算数组的平方；

### 柯里化

使用柯里化，可以允许部分地传递函数参数，以便将函数的参数减少为一个。

### 函数记忆

## 数据结构

### 解构赋值

解构赋值语法是一种js表达式。可以将数组中的值或对象的属性取出，赋值给其他变量。

使用不带声明的对象文字解构赋值时，在赋值语句外面加上( )是必需的语法。这是因为左侧的{}被认为是一个块，而不是对象文字。

#### 解构对象和数组

对象解构就是使用与对象匹配的结构来实现对象属性赋值。数组解构为方括号[]。

|  |  |
| --- | --- |
| Const{johnDoe:{age,email}}= user; | 解构对象的属性值赋值给具有相同名字的变量 |
| const{johnDoe:{age: userAge, email: userEmail}} = user; | 对象的属性值赋值给具有不同名字的变量 |

#### 绑定模式

在绑定模式中，模式以声明关键字（var、let 或 const）开始。然后，每个单独的属性必须绑定到一个变量或进一步解构。

所有变量共享相同的声明，因此，如果你希望某些变量可重新分配，而其他变量是只读的，则可能需要解构两次,一次使用 let，一次使用 const。

#### 赋值模式

在赋值模式中，模式不以关键字开头。每个解构属性都被赋值给一个赋值目标——这个赋值目标可以事先用 var 或 let 声明，也可以是另一个对象的属性——一般来说，可以是任何可以出现在赋值表达式左侧的东西。

### 栈

后进先出的有序集合

### 队列

先进先出的有序集合(区分堆栈); 在计算机科学中队列（queue）是一个抽象的数据结构（Data Structure），队列中的条目都是有秩序的。 新的条目会被加到队列的末尾，旧的条目会从队列的头部被移出。

### 链表

有序元素组合，但不是连续存放，每节点包含自身和下一个节点的指针

### 集合

无序且唯一的项组成的（数学概念的有限集合不重复）值-值存储，区分对象

### 字典

不重复的存储，键-值保存

### 散列表

通过函数计算出键的位置，也就是说保存的键值不是我们提供的原始键

### 树

非顺寻数据结构

### 图

G=(V,E)由边和顶点组成

## JSON

Json有两种数据结构：json对象初始化器，对象是名称-值对的无序集合，花括号保存对象，结构是：键:值，数据用冒号区分，对象的属性用.（点）叠加，；json数组是属性过滤选择器的数组，值的有序集合；数据在键值对中，数据由逗号分隔，数组用 [下标] 来访问。{} 和 [] 一起使用，可以组成一个对象数组。

|  |  |
| --- | --- |
| Json.parse() | 解析JSON字符串 |
| Json.stringify() | 将一个JavaScript对象或值转换为JSON字符串 |

### JSON值

对象，数组，数值，null，布尔，字符串

|  |  |
| --- | --- |
| Var str='{"name":"lin","age":"18"}'; | Var ara='["name":"lin","age":"18"]'; |

### JSON数组

## 设计模式

### 工厂模式

工厂模式是一种众所周知的设计模式，广泛应用于软件工程领域，用于抽象创建特定对象的过程。

## JS引擎和运行时

### JS引擎

JS的引擎总是包含一个调用栈和一个堆。调用栈是代码实际执行的地方；堆是一个非结构化的内存池，存储了应用程序需要的所有对象；编译：所有源代码立即被转换成机器码，然后编写此机器代码成可在任何计算机上执行的可移植文件。解释：有一个贯穿源代码的解释器，并逐行执行；代码是同时读取和执行的，代码还是需要转换成机器码，但它实在执行之前发生，而不是提前。现在JS引擎混合使用编译和解释，这称为即时编译；这种方法基本编译了整个代码，立即转换成机器码，然后立即执行。所以js混合引擎的步骤：定期提前编译，但是没有可移植的文件执行。当一段js代码进入引擎时，第一步解析代码（阅读代码），在解析过程中，代码被解析成一个数据结构，称为抽象语法树或AST。这通过首先拆分每一行代码来工作，成对语言有意义的片段。像const和function关键字，然后保存所有这些碎片以结构化的方式进入树中。此步骤还检查是否有任何语法错误，并且生成的树稍后将被使用生成机器码。

### JS环境（运行时）

作用域链无关调用堆栈中执行上下文顺序。

# JS技巧

## 控制台没有return的情况下输出undefined

**js中一个顶层函数（全局函数）eval（），或可以理解为js自带的系统函数，eval（string）**

**其作用是将 接收的 string 字符串作为参数，对其进行JavaScript 表达式或语句 计算，返回得到的值；**

**如果是没有返回值的表达式或语句，则会返回 undefined ；如果没有合法的表达式和语句，则会抛出 SyntaxError 异常 。**

于是我们可以猜测Console控制台的实质 就是 调用了eval（）函数，

验证：console.log(eval("console.log(1);"));

原因：因为Console控制台的实质，即eval（）函数，所以当输入的表达式或语句没有返回值时，会返回 undefined 。

# CSS入门篇

## CSS全称级联样式表有以下三种级联方式

### 外联式

CSS代码作为文件单独存放，如以style.css文件包含所有样式在HTML中的外部级联采用<link>标记或者 @import 语句来引入，外联优点：结构化样式分离,好维护，样式重用多个页面使用。缺点：HTTP请求多，浏览器要加载完CSS才能渲染页面，因此影响页面的性能，需要引入范围：控制整个站点

### 内联式

门户网站的 CSS 代码通常采用嵌入式，即通常所说的内联方式 (Inline Style)，其使用 <style> 标记将样式定义为内部块对象，内联有效减少HTTP请求，提升页面性能，缓解服务器压力。浏览器加载完CSS才能渲染页面，能防止CSS文件无法读取而造成页面裸奔的现象，缺点：每次修改需要修改多个页面，控制一个页面（中）

### 嵌入式行内式（内联样式）

最初级的CSS写法即把代码直接添加于所修饰的标记元素写在标签里。<div style=""></div>

嵌入式优点：书写方便，权重高，缺点：没有实现样式和结构相分离，范围：控制一个标签

选择器的优先级别；

|  |  |
| --- | --- |
| 内联样式 ， 优先级 1000 | id选择器， 优先级 100 |
| 类和伪类， 优先级 10 | 元素选择器，优先级 1 |
| 通配\* ， 优先级 0 | 继承的样式，没有优先级 |
| 通配选择器 作用：他可以用来选中页面中的所有的元素语法：\*{ } | id选择器-优先级90 通过元素的id属性值选中唯一的一个元素语法：#id属性值 { } |
| 类选择器-优先级80 通过元素的class属性值选中一组元素语法：.class属性值{ }匹配所有class属性中包含class属性值的元素 | 并集选择器 通过选择器分组同时选中多个选择器对应的元素语法：选择器1,选择器2,选择器N{ }选择器之间利用,可以是标签名称,id、class名称 |
| 交集选择器 作用：可以选中同时满足多个选择器的元素语法：选择器1选择器2选择器N{ } | 后代元素选择器-优先级100 作用：选中指定元素的指定后代元素语法：祖先元素 后代元素{} |
| 子元素选择器 作用：选中指定父元素的指定子元素语法：父元素 > 子元素 | 伪类选择器 伪类专门用来表示元素的一种的特殊的状态。'：元素名，当需要为处在这些特殊状态的元素设置样式时，就可以使用伪类 |
| 元素选择器标签选择器(类型选择器)-优先级70 p{font-size:15px;color:#00796b;｝ |  |

元素之间的关系 父元素：直接包含子元素的元素

元素之间的关系 子元素：直接被父元素包含的元素

元素之间的关系 祖先元素：直接或间接包含后代元素的元素，父元素也是祖先元素

元素之间的关系 后代元素：直接或间接被祖先元素包含的元素，子元素也是后代元素

元素之间的关系 兄弟元素：拥有相同父元素的元素叫做兄弟元素