**JavaScript**

目次

内容

[１. 入门篇 5](#_Toc109974979)

[１.１ 操作符 5](#_Toc109974980)

[１.１.１ 算术运算符 5](#_Toc109974981)

[１.１.２ 赋值运算符 5](#_Toc109974982)

[１.１.３ 比较运算符 5](#_Toc109974983)

[１.１.４ 逻辑运算符 6](#_Toc109974984)

[１.１.５ 类型运算符 6](#_Toc109974985)

[１.１.６ 扩展运算符… 6](#_Toc109974986)

[１.１.７ 空值合并运算符 6](#_Toc109974987)

[１.１.８ 一元运算符 6](#_Toc109974988)

[１.１.９ 关系运算符 6](#_Toc109974989)

[１.１.１０ 位移运算符 7](#_Toc109974990)

[１.２ 数据类型 7](#_Toc109974991)

[１.２.１ 原始值 7](#_Toc109974992)

[１.２.２ 对象引用值 7](#_Toc109974993)

[１.２.３ 判断数据类型 8](#_Toc109974994)

[１.２.４ 类型转换 8](#_Toc109974995)

[１.３ 变量和常量 8](#_Toc109974996)

[１.３.１ 变量声明 8](#_Toc109974997)

[１.３.２ var 8](#_Toc109974998)

[１.３.３ let 8](#_Toc109974999)

[１.３.４ const 8](#_Toc109975000)

[１.３.５ 模版字符串 8](#_Toc109975001)

[１.３.６ 变量作用域 8](#_Toc109975002)

[１.４ 控制流 9](#_Toc109975003)

[１.５ 迭代器 9](#_Toc109975004)

[do...while 9](#_Toc109975005)

[for 9](#_Toc109975006)

[for...in 9](#_Toc109975007)

[for...of 9](#_Toc109975008)

[forawait...of 10](#_Toc109975009)

[while 10](#_Toc109975010)

[forEcth 10](#_Toc109975011)

[$.each(object,[callback]) 10](#_Toc109975012)

[１.６ 选择器 10](#_Toc109975013)

[１.６.１ 基本选择器 10](#_Toc109975014)

[１.６.２ 层级选择器 10](#_Toc109975015)

[１.６.３ 属性过滤选择器([]) 10](#_Toc109975016)

[２. 初级篇 11](#_Toc109975017)

[２.１ 事件 11](#_Toc109975018)

[２.１.１ 事件冒泡 11](#_Toc109975019)

[２.１.２ 事件绑定 11](#_Toc109975020)

[２.１.３ 事件捕获 11](#_Toc109975021)

[２.１.４ 事件命名空间 11](#_Toc109975022)

[２.１.５ 动态事件 11](#_Toc109975023)

[２.１.６ 静态事件 11](#_Toc109975024)

[２.１.７ 定义 11](#_Toc109975025)

[２.１.８ 对象 11](#_Toc109975026)

[２.１.９ 使用 12](#_Toc109975027)

[２.１.１０ 原理 12](#_Toc109975028)

[２.１.１１ 提示 12](#_Toc109975029)

[２.２ 类(Class) 12](#_Toc109975030)

[２.２.１ 类声明 12](#_Toc109975031)

[２.２.２ 类表达式 12](#_Toc109975032)

[２.２.３ 命名类 12](#_Toc109975033)

[２.２.４ 类的继承 12](#_Toc109975034)

[２.２.５ 静态属性和静态方法 13](#_Toc109975035)

[２.３ 对象(Obejct) 13](#_Toc109975036)

[２.３.１ 对象数据类型 13](#_Toc109975037)

[２.４ 函数(Function) 13](#_Toc109975038)

[２.４.１ 函数声明 14](#_Toc109975039)

[２.４.２ 函数赋值 14](#_Toc109975040)

[２.４.３ 函数提升 14](#_Toc109975041)

[２.４.４ 自执行函数 14](#_Toc109975042)

[２.４.５ 回调函数 15](#_Toc109975043)

[２.４.６ 嵌套函数 15](#_Toc109975044)

[２.４.７ 递归函数 15](#_Toc109975045)

[２.４.８ 箭头函数 15](#_Toc109975046)

[２.４.９ 默认参数 15](#_Toc109975047)

[２.４.１０ 剩余参数 15](#_Toc109975048)

[２.５ 数组(Array) 15](#_Toc109975049)

[２.５.１ 数组定义 15](#_Toc109975050)

[２.５.２ 解构赋值 15](#_Toc109975051)

[２.６ 正则表达式(RegExp) 16](#_Toc109975052)

[２.６.１ RegExp构造函数 16](#_Toc109975053)

[２.７ JSON 16](#_Toc109975054)

[Json.parse() 16](#_Toc109975055)

[Json.stringify() 16](#_Toc109975056)

[２.７.１ JSON值 16](#_Toc109975057)

[３. 柯里化 17](#_Toc109975058)

[４. 中级篇 18](#_Toc109975059)

[４.１ Ajax 18](#_Toc109975060)

[４.１.１ 同域 18](#_Toc109975061)

[４.１.２ 跨域 18](#_Toc109975062)

[４.２ 原型链 18](#_Toc109975063)

[５. 高级篇 19](#_Toc109975064)

[５.１ 函数式编程 19](#_Toc109975065)

[５.２ 数据结构 19](#_Toc109975066)

[５.２.１ 栈 19](#_Toc109975067)

[５.２.２ 队列 19](#_Toc109975068)

[５.２.３ 链表 19](#_Toc109975069)

[５.２.４ 集合 19](#_Toc109975070)

[５.２.５ 字典 19](#_Toc109975071)

[５.２.６ 散列表 19](#_Toc109975072)

[５.２.７ 树 19](#_Toc109975073)

[５.２.８ 图 19](#_Toc109975074)

# 入门篇

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

<https://vscode.dev/>

## 操作符

### 算术运算符

|  |  |
| --- | --- |
| + | 加法 |
| - | 减法 |
| \* | 乘法 |
| / | 除法 |
| % | 取模（余数） |
| ++ | 递加 |
| -- | 递减 |

### 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 |

### 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。

### 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。

### 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运算符用来判断对象是否拥有给定属性。 |
| 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。 |

## 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。

### 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

### 隐式转换

### 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的变量是全局变量。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

`${}`

### 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。

#### 变量提升

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 | break语句中止当前循环，switch语句或label 语句，并把程序控制流转到紧接着被中止语句后面的语句。 |
| 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:default: | 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。 |
| Throw new | 抛出一个用户定义的异常。 |
| Try{} catch(){} **finally{}** | 标记一个语句块，并指定一个应该抛出异常的反馈。 |
| with | with 语句 扩展一个语句的作用域链。 |

### 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。 |
| forawait...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。 |
| forEcth |  |

## 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（~） |

# 初级篇

## 严格模式use strict

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。

## 事件

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。

### 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“；

### 对象

使对象或类基本通知能力的成员；

（中译）事件（Event）是一种使对象或类能够提供通知的成员；

（原文）An event is a member that enables an object or class to provide notifications；

对象它拥有一个事件E“想表达的思想是=当事件E发生的时候，它有能力通知别的对象“

### 使用

用于对象或者类间的动作协调与信息传递（消息推送）

### 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＂）

### 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。

## 类(Class)

类中必须创建constructor() {}函数。

### 类声明

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

### 类表达式

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

### 命名类

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

### 类的继承

extends

#### 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。

|  |  |
| --- | --- |
| Var obj={}; | 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。

## 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。

#### 表达式(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(); | 空构造函数 |

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。

### 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(){}()); |  |

### 回调函数

#### 同步回调

#### 异步回调

### 嵌套函数

即在函数的内部再定义一个函数。优点在于可以使内部函数轻松获得外部函数的参数以及函数的全局变量等。

### 递归函数

递归函数就是函数在自身的函数体内调用自身。在定义递归函数时需要两个必要条件：包括一个结束递归的条件。包括一个递归调用语句，用于实现调用递归函数。

### 纯函数

纯函数不改变原始的输入值。避免无用的副作用。

### 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。

### 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。

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。

### 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

### 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。

### 上下文

#### 有全局上下文

#### 函数上下文

### 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

## 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]();

### 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 arr=new Array; | 空数组 |
| 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); |  |

## JSON

Json有两种情况，由对象初始化器写的对象，对象是名称-值对的无序集合；由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；花括号保存对象，方括号保存数组

JSON名称/值对，JSON数据的书写格式是：名称/值对

名称/值对组合中的名称卸载前面(在双引号中)，值对写在后面（同样在双引号中）,中间用冒号隔开：”name”:”JSON”

|  |  |
| --- | --- |
| 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

|  |  |
| --- | --- |
| varstr='{"name":"lin","age":"18"}'; | varara='["name":"lin","age":"18"]'; |

# 中级篇

## Ajax

Ajax

### 同域

### 跨域

## 原型链

## Lambda表达式

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。

## 迭代器

## Promise

## Monad

## 函数链

## 组合

# 高级篇

## 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。

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

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 |

### 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。

## 数据结构

### 解构赋值

将数组中的值或对象的属性取出，赋值给其他变量。可以在一条语句中使用嵌套数据实现一个或多个赋值操作。

#### 解构对象

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。

|  |  |
| --- | --- |
| 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 |

#### 解构数组

#### 多重解构

#### 解构参数

### 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

## 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。

# 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 。