JavaScript高级程序设计

什么是JavaScript

1. JavaScript是一门用来与网页交互的脚本语言，包含以下三个组成部分：
    a. ECMAScript：由ECMA-262定义并提供其核心功能
    b. DOM（文档对象模型）：提供与网页内容交互的方法的接口
    c. BOM（浏览器对象模型）：提供与浏览器交互的方法和接口

HTML中的JavaScript

<script>元素

属性

1. async：可选，表示应立即开始下载脚本，但不能阻止其他页面动作（异步执行）。只对外部脚本文件有效。
2. charset：可选，字符集。
3. crossorign：可选，配置相关请求的CORS（跨源资源共享）设置。默认不使用CORS
4. defer：可选，脚本立即下载，但推迟执行。仅对外部脚本文件有效。
5. integrity：可选，允许比对接收到的资源和指定的加密签名以验证子资源的完整性。
6. src：可选。
7. type：可选，代表代码块中脚本语言的内容类型（也称MIME类型）

执行顺序

1. 在不使用defer或者async的情况下，包含在<script>中的代码由上至下执行

注意点

1. <script>中的代码尽量不要出现</script>，浏览器会将其视为结束标志；如果一定要使用，使用转义字符，例：

```
1
```

2. 使用src属性的<script>标签元素不应该在<script></script>中再包含其他代码（也就是一个<script>标签，行内式和外部文件式只能选一个）

跨域

1. <script>元素可以包含来自外部域的JavaScript文件
2. 若src属性是一个指向不同于的url，则浏览器会向此指定路径发送一个GET请求，此初始请求不受浏览器同源策略的限制，但返回的JavaScript仍受限制
3. 好处：通过不同的域分发JavaScript（就是我们引入外部包的过程）
4. 可使用integrity属性进行防范

位置

1. 通常将所有的JavaScript引用放在<body>元素中的页面内容后面

动态加载脚本

```
1
```

<noscript>元素

1. <noscript>可以是一种出错提示手段
2. 在以下任一条件被满足时，<noscript>中的内容就会被渲染
    a. 浏览器不支持脚本
    b. 浏览器对脚本的支持被关闭

```
1
```

语言基础

语法

标识符

1. 标识符是变量、函数、属性或函数参数的名称
2. 标识符的组成如下：
    a. 第一个字符必须是一个字母、下划线、或美元符号
    b. 剩下的其他字符可以是字母、下划线、美元符号或数字
3. 推荐使用驼峰大小写形式
4. 关键字、保留字、true、false和null不能作为标识符

关键字

break	do	in	typeof
case	else	instanceof	var
catch	export	new	void
class	extends	return	while
const	finally	super	with
continue	for	switch	yield
default	if	throw	this
function	debugger	delete	import
try			

保留字

始终保留	严格模式下保留		模块代码中保留
enum	implements	package	await
	public	interface	
	protected	private	
	static	let	

变量

1. 可以保存任意类型的数据，每个变量都是一个保存任意值的占位符
2. 变量有三个：var、let和const
3. const优先，let次之，不使用var

var

1. 不初始化的情况下，变量会保存一个特殊值undefined
2. 使用var操作符定义的变量会成为包含它的函数的局部变量
3. 在函数内部定义变量时省略var，可以创建一个全局变量（严格模式下会报错，且不推荐这么做）
4. var声明提升：
    a. 使用var声明变量时，变量会发生变量提升
    b. 所谓提升，是把所有变量声明提升到函数作用域的顶部
5. 可以使用var声明同一个变量
6. 用var在全局作用域中声明的变量会成为window对象的属性
7. 具体用例：

```Git
// var的作用域

// var声明提升
```

let

1. let声明的范围是块作用域

2. let不允许在同一个块作用域中出现冗余声明
3. let声明的变量不会在作用域中提升
4. 用let在全局作用域中声明的变量不会成为window对象的属性
5. 具体用例：

```
// 混用var与let

// for循环中的let声明
```

const

1. const的行为与let基本相同
2. 用const声明变量的同时必须初始化变量，且该变量不允许进行修改
3. 如果const变量引用的是一个对象，修改该对象内部的属性方法是允许的
4. 如果想让整个对象（包括属性方法）不能修改，可以使用Object.freeze()
5. 具体用例：

```
// for-of 和 for-in

// Object.freeze() 再给属性赋值时不会报错 但会赋值失败
const o1 = {
  age: 13,
};

const o2 = Object.freeze({
  age: 14,
});

o1.age = 0;

o2.age = 0;

o2.name = `xiaoming`;

console.log(`${o1.age}  ${o2.age}  ${02.name}`); // 0  14  undefined
```

数据类型

1. 6种简单数据类型：Undefined、Null、Boolean、Number、String、Symbol
2. 1种复杂数据类型：Obiect

typeof操作符

1. 使用typeof返回的字符串及其意义

字符串	意义
"undefined"	值未定义
"boolean"	值为布尔值
"string"	值为字符串
"number"	值为数值
"object"	值为对象或null
"function"	值为函数
"symbol"	值为符号

2. 具体用例：

```Git
// typeof 操作符
```

undefined类型

1. undefine类型只有一个值，就是特殊值undefined
2. 变量声明了但是没有赋值是，变量的值为undefined（明确是空对象指针null和为初始变量的区别）
3. 对未声明的变量，只能执行一个有用的操作，就是对它调用typeof，返回值为undefined
4. 具体事例：

```
// 包含undefined值的变量与未定义变量的区别

// 明确检测undefined这个字面值
```

Null类型

1. Null类型只有一个值，即特殊值null
2. 在定义将来要保存对象值的变量时，建议用null初始化
3. 具体事例：

```
// null与undefined表面相等

// 明确检测null这个字面值
```

Boolean类型

1. Boolean类型有两个字面值，true和false
2. 所有其他ECMAScript类型的值都有相应的布尔值的等价形式
3. 可使用Boolean（[任意类型的数据]）转型函数将其他值转换为布尔值
4. 不同类型与布尔值的转换规则

数据类型	转换为true的值	转换为false的值
Boolean	true	false
String	非空字符串	“”（空字符串）
Number	非零数值（包括无穷值）	0、NaN
Object	任意对象	null
Undefined	N/A（不存在）	undefined

Number类型

进制

1. 八进制需要前缀0（零），但如果字面量中的数字超出了应有的范围，会将整个数字视为十进制数（如079）
2. 十六进制需要前缀0x
3. 使用八进制和十六进制的格式创建的数值在所有数学操作中均视为十进制数值

浮点值

1. 小数点前可以没有整数
2. 若小数点后面没有数字，数值自动转化为整数
3. 若数值本身就是整数。只是小数点后面跟着零，自动转化为整数
4. 永远不要测试某个特定的浮点值，理由如下：

值的范围

1. Number.MIN_VALUE 最小数值
2. Number.MAX_VALUE 最大数值
3. 超过了最大值，会被转换为+Infinity
4. 超过了最小值，会被转化为-Infinity
5. isInfinity()：确定一个值是不是有限大

NaN（Not a Number）

1. 用来表示本来要返回数值的操作失败了（而不是抛出错误）
2. 0、-0、+0相除会返回NaN
3. 若分子是非0值，分母是有符号0或者无符号0，则会返回Infinity或-Infinity
4. isNaN（）函数：
    a. 参数：接受一个任意数据类型的参数
    b. 功能：判断参数是否“不是数值”
    c. 返回值：布尔值
    d. 原理：任何不能转换为数值的值都会导致这个函数返回true

5. 具体用例：

```
// 0、+0、-0 相除

// infinity情况

// isNaN() 函数
```

数值转换

Number（）函数

1. 具体用例：

```
// 布尔值

// 数值

// null

// undefined

// 字符串

// 对象
```

parseInt（）函数

1. 需要得到整数时优先使用parseInt()函数
2. 参数：
    a. 第一个参数：需要被转换的数据
    b. 第二个参数：可选，指定进制数，默认为10
3. 具体用例：

```
// 空字符串

// 数字+其他字符  组成的字符串

// 其他进制数
```

parseFloat（）函数

1. 工作方式与parseInt()类似
2. 具体用例：

```
// 第一次出现的小数点是有效的，第二次出现的小数点是无效的

// 只解析十进制数，不能指定底数

// 始终忽略字符串开头的0

// 若字符串表示整数（没有小数点或者小数点后面只有0，则返回整数）
```

string类型

1. 可以使用单引号('')、双引号("")、反引号(``)表示

字符字面量

\n	换行
\t	制表
\b	退格
\r	回车
\f	换页
\\	反斜杠
\'	单引号
\"	双引号
\`	反引号
\xnn	以十六进制编码nn表示的字符（n是十六进制数字0~F）
\unnnn	以十六进制编码nnnn表示的Unicode字符

- 注意：转义序列表示一个字符（在计算的时候算一个）
- string.length ： length属性用于获取字符串的长度

字符串的特点

1. 这里的字符串是不可变的，要修改必须某个变量的字符串值必须先销毁原字符串，再保存新变量

转换为字符串：

1. toString()方法：
    a. 可用于数值、布尔值、对象和字符串
    b. 不可用于null和undefined
    c. 参数：仅在对数值进行转换时接受参数，参数为转换的进制数
2. String（）转型函数，规则如下：
    a. 若值有toString()方法，则调用此方法并返回结果
    b. 若值为null，返回“null”
    c. 若值为undefined，返回“undefined”
3. 具体用例：

```
// toString()

// String()
```

模板字面量（反引号）

1. 模板字面量在定义模板时特别有用（？？？）
2. 模板字面量会保持反引号内部的空格
3. 具体用例：

```
// 定义模板 HTML模板 可以安全地插入到HTML中

// 保持内部空格
```

字符串插值

1. 通过`${[JavaScript表达式]}`来实现
2. 插入值会使用toString()强制转换为字符串
3. 插值表达式中可以调用函数和方法
4. 插值表达式中可以插入自己之前的值
5. 具体用例：

```
// toString () 转换

// 表达式中调用函数和方法

// 表达式中插入自己之前的值
```

模板字面量标签函数

1. 模板字面量支持定义标签函数（？？？）
2. 通过标签函数可以自定义插值行为

3. 标签函数会接收被插值记号分隔后的模板和对每个表达式求值的结果
4. 具体用例：

```Git
// 标签函数
```

原始字符串

1. 使用 String.Raw`字符串内容` 可以直接获取原始的模板字面量，而不是被转移后的字符表示

Symbol类型（符号）

1. 符号是原始值，且符号实例是唯一、不可变的。
2. 用途：确保对象属性使用唯一标识符，不会发生属性冲突的危险
3. 具体用例：

```Git
// typeof操作符

// 创建Symbol（`字符串参数`）实例并将其用作对象的新属性 即使参数相同，Symbol也是不同的

// 全局符号注册表 Symbol.for(`字符串参数`)方法
```

4. 常用内置符号：
    a. 这些内置符号最重要的用途之一是重新定义它们，从而改变原生结构的行为
    b. 所有内置符号属性都是不可写、不可枚举、不可配置的

Git | 复制代码

```
// Symbol.asyncIterator
// 一个方法，该方法返回对象默认的AsyncIterator。由for-await-of使用。实现了异步迭代器API的函数

// Symbol.hasInstance
// 一个方法，该方法决定一个构造器对象是否认可一个对象是它的实例。由操作符instanceof操作符使用。
// instanceof操作符可以用来确定一个对象实例的原型链上是否有原型

// Symbol.isConcatSpreadable
// 一个布尔值，如果是true或真值，类数组对象会被打平到数组实例，用Array.prototype.concat()打平
// 如果是false或假值，整个对象被追加到数组末尾

// Symbol.iterator
// 一个方法，该方法返回对象默认的迭代器。由for-of语句使用。这个符号实现了迭代器API的函数

// Symbolmatch
// 一个正则表达式方法，该方法用正则表达式去匹配字符串。由String.prototype.match()方法使用

// Symbol.replace
// 一个正则表达式方法，该方法替换一个字符串中的匹配字符串。由String.prototype.replace()方法使用

// Symbol.search
// 一个正则表达式方法，该方法返回字符串中匹配正则表达式的索引。由String.prototype.search()使用

// Symbol.species
// 一个函数值，该函数作为创建派生对象的构造函数

// Symbol.split
// 一个正则表达式方法，该方法在匹配正则表达式的索引位置拆分字符串。由String.prototype.split()使用

// Symbol.toPrimitive
// 一个方法，该方法将对象转换为相应的原始值。由ToPrimitive抽象对象使用

// Symbol.toStringTag
// 一个字符串，该字符串用于创建对象默认字符串描述。由内置方法Object.prototype.toString()使用
```

Object类型

1. 对象是一组数据和功能的集合体
2. 对象通过new操作符后跟对象类型的名称来创建
3. Object是派生其他对象的基类，Object类型的所有属性和方法在派生的对象上同样存在
4. Object实例的属性和方法：

constructor	用于创建当前对象的函数
hasOwnProperty(PropertyName)	用于判断当前对象实例上是否存在给定的属性。PropertyName是字符串
isPrototypeOf(object)	用于判断当前对象是否为另一个对象的原型
propertyIsEnumerable(PropertyName)	用于判断给定的属性是否可以使用for–in语句枚举。PropertyName是字符串
toLocaleString()	返回对象的字符串表示，该字符串反映对象所在的本地化执行环境
toString()	返回对象的字符串表示
valueOf()	返回对象对应的字符串、数值或布尔值表示

操作符

1. 在应用给对象时，操作符通常会调用valueOf（）和\或toString（）方法来取的可以计算的值

一元操作符

1. 递增/递减操作符（++/--）：类比于c语言的递增递减操作符
2. 一元加和减：若应用于非数值，则相当于执行了Number()转型函数
3. 具体用例：

JavaScript | 复制代码

```
let s = 1.1;

console.log(--s); // 0.10000000000000009
```

位操作符

1. 位操作作用于32位的整数，但ECMAScript中数值以IEEE 754 64位格式存储
2. 前32位表示整数值，第32位表示符号（从左到右，为从后到前）
3. 正值以真正的二进制格式存储，负值以补码的形式存储
4. NaN和Infinity在位操作中会被当成0处理
5. 位操作符应用于非数值，自动使用Number()函数转换该值

按位非（~）

1. 一个操作数
2. 作用：返回数值的一补数（二进制数直接取反），在十进制中相当于取反并加1

按位与（&）

1. 两个操作数
2. 作用：将两个数的二进制表示对齐，执行”与“操作进行合并

按位或（|）

1. 两个操作数
2. 作用：将两个数的二进制表示对齐，执行“或”操作进行合并

按位异或（^）

1. 两个操作数
2. 作用：将两个数的二进制表示对齐，执行“异或”操作进行合并

左移（<<）

1. a<<b：a是被左移的数，b是左移的位数
2. 作用：按照指定的位数将数值的所有位向左移动
3. 因左移而在右边空出来的位置用0填充

有符号右移（>>）

1. a>>b：a是被右移的数，b是右移的位数
2. 作用：按照指定的位数将数值的所有位向右移动，同时保留符号位

3. 因右移而在左边空出来的位置用0填充

无符号右移（>>>）

1. a>>>b：a是被右移的数，b是右移的位数
2. 作用：按照指定的位数将数值的所有位向右移动，不管是不是符号位（因此负数左移后结果很大）
3. 因右移而在左边空出来的位置用0填充

布尔操作符

逻辑非（！）

1. 返回值：布尔值

逻辑与（&&）

1. 可应用于任何类型的操作数
2. &&是一个短路操作符:a&&b，若a对应的布尔值为true，则a&&b的结果是b；若a对应的布尔值为false，则a&&b的结果是false，第二个值就不管了
3. 如果有一个操作数是null，返回null
4. 如果有一个操作数是NaN，返回NaN
5. 如果有一个操作数是undefined，返回undefined

逻辑或（||）

1. 可应用于任何类型的操作数
2. ||是一个短路操作符:a||b，若a对应的布尔值为false，则a||b的结果是b；若a对应的布尔值为true，则a||b的结果是true，第二个值就不管了
3. 如果有一个操作数是null，返回null
4. 如果有一个操作数是NaN，返回NaN
5. 如果有一个操作数是undefined，返回undefined
6. 典型应用：

```JavaScript
// 第一个不是null或者undefined 第二个值就不管了
let myObject = preferredObject || backupObject
```

乘性操作符

乘法操作符（*）

1. 可应用于任何类型的操作数
2. 任一操作符是NaN，返回NaN
3. Infinity*0=NaN
4. Infinity*Infinity=Infinity

除法操作符（/）

1. 可应用于任何类型的操作数
2. 任一操作符是NaN，返回NaN
3. Infinity/Infinity=NaN
4. 0/0=NaN

取模操作符（%）

1. 可应用于任何类型的操作数
2. 任一操作符是NaN，返回NaN
3. Infinity%c=NaN
4. c%0=NaN
5. Infinity%Infinity=NaN

指数操作符（**）

加性操作符

加法操作符（+）

1. 用于数值求和：
    a. 任一操作符是NaN，返回NaN
    b. Infinity+(−Infinity)=NaN
    c. +0+(+0)=+0
    d. −0+(+0)=+0
    e. −0+(−0)=−0
2. 字符串拼接
    a. 只要有一个操作数是字符串，就会将另一个操作数转换为字符串，并将两者拼接
    b. 若任一操作数是对象、布尔值或数值，则调用toString()转换
    c. 对于undefined和null，调用String()转换为“undefined”和“null”

减法操作符（−）

1. 任一操作符是NaN，返回NaN
2. Infinity−Infinity=NaN
3. +0−(+0)=+0
4. −0−(+0)=−0
5. −0−(−0)=+0（无论是加还是减，都只有−0−0=−0）

关系操作符（< 、>、<=、>=）

1. 任一操作符是NaN，返回false
2. 结果为布尔值
3. 若有任一操作数是字符串、对象或者是布尔值，最终都是数值的比较
4. 若两个对象都是字符串，会逐个比较它们的字符编码

相等操作符

等于和不等于（==、！=）

1. 任一操作符是NaN，相等操作返回false，不相等操作返回true（NaN==NaN 是false）
2. 结果为布尔值
3. 任一操作数是数值、布尔值，或者一个操作数是对象，另一个不是，都会转换为数值进行比较
4. 两个操作数都是对象，比较它俩是否都指向同一个对象

5. null==undefined 是true
6. null和undefined不能转换成其他类型再比较

全等和不全等（===、！==）

1. 任一操作符是NaN，相等操作返回false，不相等操作返回true（NaN===NaN 是false）
2. 结果为布尔值
3. 在比较是不转换操作数
4. null===undefined是false

条件操作符

1. 具体用例：

```JavaScript
let max = (num > num2) ? num1 : num2;
// 若(num > num2)为true   (num > num2) ? num1 : num2===num1
// 若(num > num2)为false   (num > num2) ? num1 : num2===num2
```

赋值操作符

逗号操作符

1. 可以用于在一条语句中执行多个操作
2. 在赋值时用逗号操作符分隔值，最终会返回表达式的最后一个值

语句

1. if语句、do-while语句、while语句、for语句、break语句、continue语句与c语言几乎一样

for-in语句

1. 一种严格的迭代语句，用于枚举对象中的非符号键属性
2. 因为对象的属性是无序的，所以for-in语句不能保证返回对象属性的顺序
3. 具体用例：

```javascript
// 语法：for(property in expression) statement

let o = {
    name: `xiaoming`,
    age: 11,
    height: 180,
    weight: 150,
};

for (const propName in o) {
    console.log(`${propName}`);
}
// name
// age
// height
// weight
```

for–of语句

1. 一种严格的迭代语句，用于遍历可迭代对象的元素
2. for–of循环会按照可迭代对象的next（）方法产生值的顺序迭代元素
3. 具体用例：

```javascript
// 语法：for(property of expression) statement

for (const el of[1, 4, 3, 2]) {
    console.log(el);
}
// 1
// 4
// 3
// 2
```

标签语句

1. break语句和continue语句都可以与标签语句一起使用，返回代码中的特定位置（可以很方便地退出

多层循环）

2. 具体用例：

break与标签语句 JavaScript 复制代码

```javascript
// break
let num = 0;

for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
        if (i == 5 && j == 5) {
            break;
        }
        num++;
    }
}

console.log(num);// 95

// break+标签 如果不用标签 会退出一层循环 这个标签在循环最外层 因此退出到了最外层循环
num = 0;

outermost:
    for (let i = 0; i < 10; i++) {
        for (let j = 0; j < 10; j++) {
            if (i == 5 && j == 5) {
                break outermost;
            }
            num++;
        }
    }

console.log(num);// 55
```

continue+标签语句 JavaScript 复制代码

```javascript
// continue
num = 0;

for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
        if (i == 5 && j == 5) {
            continue;
        }
        num++;
    }
}

console.log(num);

// continue+标签 如果不用标签，会结束本轮j循环进入j+1循环
// 用了标签，结束本轮i循环，进入i+1循环
num = 0;

outermost2:
    for (let i = 0; i < 10; i++) {
        for (let j = 0; j < 10; j++) {
            if (i == 5 && j == 5) {
                continue outermost2;
            }
            num++;
        }
    }

console.log(num);
```

switch语句

1. switch语句与c语言的类似
2. 判断条件：switch的参数与条件相等的情况下进入该语句
3. 但是，switch语句可以用于所有的变量类型（字符串、变量都是可以的）
4. 条件的值可以是常量，变量或者是表达式
5. switch语句在比较条件的值时会使用全等操作符
6. 具体用例：

switch JavaScript 复制代码

```
// 案例中switch语句是布尔值 条件是表达式 若条件为true 与switch参数相等 就进入语句
let num = 25;

switch (true) {
    case num < 0:
        console.log(`num<0`);
        breakl;
    case num >= 0 && num < 10:
        console.log(`0<=num<10`);
        break;
    case num >= 10 && num < 20:
        console.log(`10<=num<20`);
        break;
    default:
        console.log(`num>=20`);
}
```

函数

1. 不需要指定是否返回值
2. 碰到return语句，函数会立即停止执行并退出
3. 推荐函数要么返回值，要么不返回值
4. 不指定返回值的函数实际上会返回特殊值undefined

变量、作用域与内存

原始值与引用值

1. 原始值：
    a. Undefined,Null,Boolean,Number,String,Symbol
    b. 保存原始值的变量是按值访问的，我们操作的就是存储在变量中的实际值
2. 引用值：
    a. 保存在内存中的对象
    b. 保存引用值的变量是按引用访问的，实际操作的是对该对象的引用

动态属性

1. 原始值：
    a. 原始值不能有属性（给其添属性不会报错，显示为undefined）
    b. 原始类型的初始化：
        i. 只使用原始字面量形式
        ii. 使用new关键字，会创建一个Object类型的实例但其行为类似原始值
2. 引用值：可以随时增添、修改、删除其属性和方法
3. 具体用例：

JavaScript | 复制代码

```
// 原始类型使用new的初始化
let name = new String(`xiaoming`);

console.log(typeof name);//Object
```

复制值

1. 原始值：通过变量把原始值赋值给另一个变量时，原始值复制了一份，放到了新变量的位置（两者完全独立）
2. 引用值：复制的是对象的地址，两者实际上指向同一个对象

传递参数

1. 无论是原始值还是引用值，都是按值传递的，只是引用值的值是一个地址，因此指向同一个地址

确定类型（instanceof）

1. typeof适用于判断一个变量是否是原始类型，是什么原始类型
2. instanceof适用于确定一个对象是什么类型的对象
3. 语法：
    a. result = variable instanceof constructor
    b. result：布尔值
    c. variable：实例对象

    d. constructor：某个构造函数（Object/Array/RegExp等）
4. 用instanceof检测原始值会返回false

执行上下文和作用域

执行上下文

1. 执行上下文是一个名词概念，表示一个变量或者函数的关联区域
2. 每个上下文都有一个关联的变量对象，而这个上下文中定义的所有变量和函数都存在于这个对象上面
3. 上下文在其所有代码执行完毕后会被销毁，包括定义在其上面的变量和函数
4. 全局上下文是最外层的上下文 （window对象）
    a. var声明的全局变量会成为window的属性和方法
5. 每个函数都有自己的上下文
    a. 通过上下文栈来控制程序的执行流
    b. 代码执行流进入函数->函数上下文入栈->函数执行完毕->弹出函数上下文->控制权还给之前的执行上下文

作用域链

1. 上下文中的代码在执行的时候，会创建变量对象的一个作用域链。
2. 代码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
3. 如果上下文是函数，则其活动对象用作变量对象
4. 作用域链的下一个变量对象来自包含上下文（指包含自己这个上下文的上下文），再下一个对象来自再下一个包含上下文，依次类推至全局上下文。（实际上就是由大到小一条链，越小的越往链的前端跑）
5. 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
6. 内部上下文可以沿作用域链访问外部上下文的一切
7. 外部上下文无法访问内部上下文的一切
8. 函数参数被认为是当前上下文中的变量

作用域链增强

1. try/catch语句，with语句会导致作用域链增强
2. 指在作用域前端临时添加一个上下文，这个上下文在代码执行后会被删除

变量声明

1. var声明的变量会被自动添加到最接近的上下文
2. let和const的作用域是块级的
3. 标识符查找
    a. 搜索开始于作用域链前端，以给定的名称搜索对应的的标识符
    b. 作用域中的对象也有一个原型链，因此搜索可能会涉及每个对象的原型链
    c. 搜索到了就不会再搜索下去了

垃圾回收

1. 执行环境负责在代码执行期间管理内存
2. 最常用的策略是标记清理
3. 引用计数也是一种回收策略，但是在循环引用等有很大的问题
    a. 为避免循环引用，应在确保不使用的情况下切断原生JavaScript对象和DOM元素之间的联系

内存管理

1. 如果数据不必要，那就把它设置为null，从而释放其引用（解除引用）
2. 使用const和let而不是var有助于提升性能
3. 尽量避免动态属性赋值或者动态添加属性，并在构造函数中一次性声明所有属性（这样多个对象共同使用一个隐藏类）
4. 意外声明全局变量会造成内存泄漏
5. 定时器也会导致内存泄露
6. 使用闭包会造成内存泄露（闭包是指在函数里面声明函数，闭包也有许多优点）