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
// 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）

基本引用类型

1. 引用值（对象）是某个特定引用类型的实例
2. 引用类型：描述了自己的对象应有的属性和方法
3. 新对象通过new操作符后加一个构造函数（constructor）来创建
4. 构造函数就是用来创建新对象的函数
5. 函数也是一种引用类型

6. 所有的引用类型都是基于Object类型的

Date

1. 日期/时间组件方法，具体用例：

Date JavaScript 复制代码

```
// 创建一个日期对象
let now = new Date(); // Mon Jul 13 1987 20:29:48 GMT+0900 (中国夏令时间)

// Date.now()方法返回执行时日期和时间的毫秒数 可用于代码分析（获得时间差）
console.log(Date.now()); // 1657715845530

let t1 = now.getTime(); // 1657715388159

// 返回四位数年
let y1 = now.getFullYear(); // 2022

// 返回UTC日期的四位数年
let y2 = now.getUTCFullYear(); // 2022

// 设置日期的年
let y3 = now.setFullYear(1987); // 553174258602

// 设置日期的年后 now对应的年改变了
let y4 = now.getFullYear(); // 1987
```

2. 使用Date类型可以构建倒计时：

倒计时 JavaScript | 复制代码

```javascript
//定义倒数函数
function countDown(time) {
    var nowTime = +new Date(); //当前时间
    var inputTime = +new Date(time); //用户输入的时间
    var times = (inputTime - nowTime) / 1000; //由毫秒得到秒
    var d = parseInt(times / 60 / 60 / 24); //天
    d = d < 10 ? '0' + d : d;
    var h = parseInt(times / 60 / 60 % 24); //时
    h = h < 10 ? '0' + h : h;
    var m = parseInt(times / 60 % 60); //分
    m = m < 10 ? '0' + m : m;
    var s = parseInt(times % 60); //秒
    s = s < 10 ? '0' + s : s;
    return d + '天' + h + '时' + m + '分' + s + '秒'; //返回剩余时间
}
```

3. 其他的方法类似，具体参考MDN

RegExp

1. 通过RegExp支持正则表达式

创建正则表达式：

a. 方法1：let expression = /pattern/flags

b. 方法2：let pattern2 = new RegExp("pattern","flags")

c. 方法3：基于已有的正则表达式实例，并选择性地修改标记

创建正则表达式 JavaScript | 复制代码

```javascript
// 方法1
let pattern1 = /[bc]at/i;

// 方法2
let pattern2 = new RegExp("[bc]at","i");

// 方法3
const re1 = /cat/g;
const re2 = new RegExp(re1,"i");
```

正则表达式

标记（flags）

g	全局模式，表示查找字符串的全部内容，而不是匹配一个就结束
i	不区分大小写
m	多行模式，查到一行末会继续查找
y	粘附模式，只查找从lastIndex开始及之后的字符串（最后一个索引对应的值为首）
u	Unicode模式，启用Unicode匹配
s	dotAll模式，表示元字符，匹配任何字符串（包括\n和\r）

转义

1. 元字符：({ [\ ^ $ |)] } ? * + .
2. 要匹配上面这些字符本身，必须使用反斜杠（\）来转义

二次转义

1. 在RegExp构造正则表达式时，在需要一次转义的地方进行二次转义（元字符前面要加两条斜杠）

RegExp实例的方法

RegExp.exec(string)

1. 作用：配合捕获组使用（捕获组是指正则表达式中用小括号（）包起来的那一部分
2. 参数：一个，必选，是要应用模式的字符串
3. 返回值：包含匹配信息的数组，若没找到匹配项，则返回null
4. 返回值（数组）的属性：
    a. index：字符串中匹配模式的其实位置
    b. input：要查找的字符串
5. 注意：如果正则表达式没有设置全局标记，对同一个字符串调用多少次exec（），也只会返回第一个匹配信息；若设置了全局模式（g标记），则每次调用exec（）方法会返回一个匹配信息

RegExp.test(string)

1. 作用：测试模式是否匹配
2. 参数：一个，必选，用于测试的字符串
3. 返回值：若输入的文本域模式匹配，则返回true，否则返回false

原始值包装类型

1. 我们调用某个原始值的属性或者方法的时候，后台自动创建相应的原始包装类型对象，暴露属性和方法，用完之后，销毁该对象

Boolean

1. Boolean类型的对象，本身是一个对象，而不是布尔值，在a&&b这种情况下要注意

Number

Number.toFixed(参数)

1. 作用：返回包括小数点位数的数值字符串
2. 参数：一个参数，表示返回的数值字符串小数点的位数（范围：0–20）

Number.toExponential(参数)

1. 作用：返回以科学计数法表示的数值字符串
2. 参数：一个参数，表示结果中小数的位数

Number.toPrecision(参数)

1. 作用：根据情况返回最合理的结果，可能是固定长度，也可能是科学计数法
2. 参数：一个参数，表示结果中数字的总位数（不包括指数）（小数位范围：1–21）

Number.isInteger(参数)

1. 作用：用于辨别一个数值是否保存为整数
2. 参数：一个参数，被辨别的数值

Number.isSafeInteger(参数)

1. 作用：鉴别一个整数是否在安全范围内
2. 参数：一个参数，被鉴别的整数
3. 安全范围：Number.MIN_SAFE_INTEGER(−2^(53)+1)~~Number.MAX_SAFE_INTEGER(2^(53)−1)

String

String.length

1. 表示字符串中字符的数量

String.concat()

1. 作用：将一个或多个字符串拼接成一个新字符串
2. 参数：可以接收任一多个参数
3. 返回值：一个新字符串

String.slice(start，end)

1. 作用：提取字符串，范围[start,end)
2. 参数为负数的情况：范围以字符串长度加上负值参数

String.substr(start，number)

1. 作用：提取字符串，范围为从start开始，数量为number
2. 参数为负数的情况：第一个负值参数变成字符串长度加负值参数，第二个参数变0

String.substring(start，end)

1. 作用：提取字符串，范围[start,end)
2. 参数为负数的情况：负值参数全部变0

String.indexOf(string，start) 与 String.lastIndexOf(string，start)

1. 作用：从字符串中搜索传入的字符串，前者从前往后，后者从后往前
2. 参数：string为传入的字符串，start为搜索的开始位置
3. 返回值：目标字符串所在位置的索引，若没找到，返回-1
4. 具体用例：

搜索所有目标字符串 JavaScript 复制代码

```javascript
let stringValue = `today is a sunny day,it's good to have a travel`;

// 存放目标字符串的位置
let position = new Array();

let pos = stringValue.indexOf(`a`);

// 找不到就停止
while (pos > -1) {
    console.log(pos);
    position.push(pos);
    pos = stringValue.indexOf(`a`, pos + 1);// 往后一位
}

console.log(position); // [ 3, 9, 18, 35, 39, 43 ]
```

字符串迭代

1. 具体用例：

字符串迭代 JavaScript | 复制代码

```javascript
for (const c of `abcd`){
  console.log(c);
}
// a
// b
// c
// d
```

大小写转换

1. String.toLowerCase()
2. String.toLocaleLowerCase()
3. String.toUpperCase()
4. String.toLocaleCase()

字符串模式匹配

1. String.match（）：根据参数匹配字符串并返回数组
2. String.replace（）：参数1表示匹配模式，参数2表示替换字符串
3. String.split（）：参数1是分隔符，参数2是数组大小，按照参数1将字符串分隔并存放到数组中

单例内置对象

Global

1. 在全局作用域中定义的变量和函数都会变成Global对象的属性

Math

数学特殊值

Math.E	自然对数的基数e的值
Math.LN10	10为底的自然对数
Math.LN2	2为底的自然对数
Math.LOG2E	以2为底e的对数
Math.LOG10E	以10为底e的对数
Math.PI	pi的值
Math.SQRT1_2	1/2的平方根
Math.SQRT2	2的平方根

min()和max()方法

1. 均可以接受任意多的参数，确定这组数的最小值或者最大值

舍入方法

1. Math.ceil()：向上舍入
2. Math.floor()：向下舍入
3. Math.round()：四舍五入
4. Math.fround()：返回数值最接近的单精度（32位）浮点值表示

random（）方法

1. random（）方法返回一个0~1范围内的随机数[0,1}
2. 随机数生成：

```javascript
// 函数 功能：生成指定范围的随机数
function selectFrom(lowerValue, upperValue) {
    let choices = upperValue - lowerValue + 1;
    return Math.floor(Math.random() * choices + lowerValue);
}
```

3. 若为了加密需要而生成随机数，使用window.crypto.getGandomValues()

其他方法

Math.abs(x)	绝对值
Math.exp(x)	e的x次幂
Math.expm1(x)	e的x次幂-1
Math.log(x)	x的自然对数
Math.log1p(x)	x的自然对数+1
Math.pow(x,power)	x的power次幂
Math.hypot(...nums)	每个数平方和的平方根
Math.clz32(x)	32位整数x的前置0的数量
Math.sign(x)	x的符号
Math.trunc(x)	x的整数部分
Math.sqrt(x)	平方根
Math.cbrt(x)	立方根
Math.acos(x)	反余弦
Math.acosh(x)	反双曲余弦
Math.asin(x)	反正弦
Math.asinh(x)	反双曲正弦
Math.atan(x)	反正切
Math.atanh(x)	反双曲正切
Math.atan2(x)	y/x的反正切
Math.cos(x)	余弦
Math.sin(x)	正弦
Math.tan(x)	正切

集合引用类型

Object

1. 在对象字面量中，属性名可以是字符串或者数值（数值会自动转换为字符串）
2. 对象字面量很适合用于给函数传递大量参数，可选参数用对象来封装，必选参数使用命名参数
3. 一般通过点语法使用对象的参数和方法
4. 使用中括号访问属性的场景：
    a. 想要通过变量访问属性的时候
    b. 属性命中包含非字母数字字符的时候

Array

1. 数组中每个槽位都可以存储任意类型的数据
2. 数组是动态大小的，会随着数据添加而自动增长
3. 在实践中要避免使用数组空位。如果确实需要空位，可以显式地使用undefined替代
4. 使用Array.length属性可以很方便地向数组末尾添加元素
5. Array.isArray()用于确定一个值是否为数组

创建数组

1. 使用new
2. 使用数组字面量
3. Array.from（）：
    a. 将类数组结构转换为数组实例
    b. 第一个参数是一个类数组对象，即任何可迭代结构（字符串，集合，映射，数组，arguments，某些自定义对象）
    c. 第二个参数可选，为映射函数参数，用于直接增强新数组的值（不必再创建中间数组，新数组元素与可迭代结构相应元素一一对应）
    d. 第三个参数可选：用于指定映射函数中this的值，但这个重写的this在箭头函数中不适用
4. Array.of（）：将一组参数转换为数组实例
5. 具体用例：

Array.from()

JavaScript | 复制代码

```javascript
const a1 = [1, 2, 3, 4];

// 使用映射函数参数，直接增强新数组的值
const a2 = Array.from(a1, x => x - Math.pow(x, 2));

// 指定this的值
const a3 = Array.from(a1, function(x) {
    return x ** this.exponent;
}, { exponent: 2 });

console.log(a2); // [ 0, -2, -6, -12 ]

console.log(a3); // [ 1, 4, 9, 16 ]
```

迭代器方法

1. Array.keys（）：返回数组索引的迭代器
2. Array.values（）：返回数组元素的迭代器
3. Array.entries（）：返回索引/值对的迭代器

迭代器方法 JavaScript | 复制代码

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

let aEntries = Array.from(a.entries());

let aKeys = Array.from(a.keys());

let aValues = Array.from(a.values());

console.log(aEntries); // [ [ 0, 1 ], [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 4, 5 ] ]

console.log(aKeys); // [ 0, 1, 2, 3, 4 ]

console.log(aValues); // [ 1, 2, 3, 4, 5 ]

// 利用解构拆分键值对
for (const [idx, element] of a.entries()) {
    console.log(`${idx}:${element}`);
}
// 0:1
// 1:2
// 2:3
// 3:4
// 4:5
```

复制和填充

Array.fill（参数1，[start]，[end]）

1. 参数1是填充物
2. 填充范围为[start，end）（注意start和end都可以是负数，若是负值，则用数组长度加上负值）
3. 静默忽略超出数组边界、零长度及方向相反的索引范围

Array.copyWithin(insert,start,end)

1. 从一个数组中[start,end)的范围进行复制，插入到另一个数组的insert位置
2. 静默忽略超出数组边界、零长度及方向相反的索引范围

转换方法

Array.toString() Array.toLocaleString()

1. 对每个字符调用Array.toString()或者Array.toLocaleString()方法，拼接而成新字符串

Array.join(参数)

1. 接受一个参数，参数表示字符串分隔符；若不传入参数或者传入undefined，默认逗号进行分隔
2. 返回包含所有项的字符串

栈方法与队列方法

1. 栈：先进后出（LIFO,Last-In-First-Out），最先添加的项先被删除
2. 队列：先进先出（FIFO,First-In-First-Out），最后添加的先被删除
3. Array.push()：接受任意数量的参数，添加到数组末尾，返回数组的最新长度
4. Array.pop()：删除数组的最后一项，返回被删除的项
5. Array.shift()：删除数组的第一项并返回它
6. Array.unshift()：在数组开头增添任意多元素，返回数组的新长度
7. 以上四个方法均改变了原数组

排序方法

Array.reverse()

1. 将数组反向排列

Array.sort()

1. sort（）方法对数组中元素调用String()方法转型后再进行比较（比较个位数可以，但是多位数不行）
2. sort（）方法可以接受一个比较参数，用于判断哪个值应该排在前面（如果第一个参数应该排在第二个参数前面，就返回负值）

数组排序 JavaScript 复制代码

```javascript
let values = [0, 4, 33, -9, -9, 0];

// 反向排序 使用箭头函数+条件操作符简化代码
values.sort((a, b) => a < b ? 1 : a > b ? -1 : 0);

console.log(values); // [ 33, 4, 0, 0, -9, -9 ]

// 正向排序
values.sort((a, b) => a > b ? 1 : a < b ? -1 : 0);

console.log(values); // [ -9, -9, 0, 0, 4, 33 ]
```

操作方法

Array.concat()

1. 不影响原数组
2. 在现有数组全部元素的基础上创建一个副本（新数组），再把参数接到副本末尾
3. 如果参数是数组，默认会将数组打平再接到副本后面，此行为由Symbol.isConcatSpreadable控制，此值默认为true，将其设置为false就可以阻止concat（）打平参数数组

Symbol.isConcatSpreadable JavaScript 复制代码

```javascript
const a1 = [1, 2, 3, 4];

const a2 = [5, 6];

a2[Symbol.isConcatSpreadable] = false;

const a3 = {
    [Symbol.isConcatSpreadable]: true,
    length: 2,
    0: 5,
    1: 6
};

console.log(a1.concat(a2));

console.log(a1.concat(a3));
```

Array.slice(start，[end])

1. 创建一个原数组[start,end)的元素组成的新数组

Array.splice(start,number,[insert])

1. start:要删除的第一个元素的位置
2. number：要删除元素的数量
3. insert：要插入的元素，元素插入到删除第一个删除元素的位置，插入元素的数量不限
4. 返回值：返回被删除的元素组成的数组，若没有则返回空数组
5. 通过灵活使用三个参数，可以实现删除，插入和替换

搜索和位置方法

严格相等

1. 以下三个方法在比较的时候会使用全等（===）比较

Array.indexOf(target，start)

1. 参数：要查找的元素以及可选的起始搜索位置
2. 返回值：目标元素位置，没找到则返回-1
3. 方向：从前往后

Array.lastIndexOf(target，start)

1. 参数：要查找的元素以及可选的起始搜索位置
2. 返回值：目标元素位置，没找到则返回-1
3. 方向：从后往前

Array.includes(target，start)

1. 参数：要查找的元素以及可选的起始搜索位置
2. 返回值：找到返回true，找不到返回false
3. 方向：从前往后

断言函数

1. 断言函数接受三个参数：元素、索引和数组本身

2. Array.find()返回第一个匹配的元素　Array.findIndex()返回第一个匹配元素的索引
3. 这两个断言函数都可以接收第二个可选参数，用于指定断言函数内部this的值

断言函数　JavaScript | 复制代码

```
const people = [{
        name: `xiaoming`,
        age: 11,
    },
    {
        name: `xiaowang`,
        age: 27,
    },
];

console.log(people.findIndex((element, index, array) => element.age < 17));
// 0

console.log(people.find((element, index, array) => element.age < 17));
// { name: 'xiaoming', age: 11 }
```

迭代方法

1. 每个方法接收两个参数，以每一项为参数运行的函数，以及可选的作为函数运行上下文的作用域对象（影响函数中this的值）。
2. 传给每个方法的函数接收3个参数：数组元素，元素索引，数组本身
3. 这些方法都不改变原数组
4. Array.every（）：对数组的每一项都运行传入的函数，如果对每一项函数都返回true，则这个方法返回true（适合用于判断数组是否符合某一个条件）
5. Array.filter（）：对数组的每一项都运行传入的参数，函数返回true的项会组成数组之后返回（适合用于从数组中筛选满足给定条件的元素）
6. Array.forEach（）：对数组的每一项都运行传入的函数，没有返回值（相当于使用for循环遍历数组）
7. Array.map（）：对数组的每一项都运行传入的函数，返回由每次函数调用的结果构成的数组（适用于获得一个与原始数组元素一一对应的新数组）
8. Array.some（）：对数组的每一项都运行传入的参数，如果有一项返回true，则这个方法返回true（适合用于判断数组是否符合某一个条件）

```JavaScript
// every () 方法
const a = [1, 2, 3, 4];

console.log(a.every((item, index, array) => item < 5));// true
```

归并方法

1. Array.reduce() Array.reduceRight()
2. 这两个方法都会迭代数组的所有项，并在此基础上构建一个返回值
3. 两个方法的区别点在于reduce（）从前往后，reduceRight（）从后往前
4. 接收两个参数，第一个参数是对每一项都运行的归并函数，第二个参数是可选的以之为归并起点的初始值

```JavaScript
const a1 = [1, 2, 3, 4];

// 归并函数接收4个参数
// prev：上一个归并值   cur:当前项   index当前项的索引   array：数组本身
console.log(a1.reduce((prev, cur, index, array) => prev + cur));
// 10
```

定型数组

ArrayBuffer()

1. ArrayBuffer是一个构造函数，可用于在内存中分配特定数量的内存空间，ArrayBuffer已经创建就不能调整大小
2. 参数，一个参数，为创建的arrayBuffer的大小
3. ArrayBuffer会将所有的二进制位初始化为0
4. 要读取或者写入ArrayBuffer，必须通过视图

定型数组

1. 定型数组是一种ArrayBuffer视图，但并不是唯一一种
2. 定型数组的类型以<ElenentType>Array 来进行划分（如Int8Array是一种类型）

ElementType

ElementType	字节	说明	范围
Int8	1	8位有符号整数	–128–127
Uint8	1	8位无符号整数	0–255
Int16	2	16位有符号整数	–32768–32767
Uint16	2	16位无符号整数	0–65535
Int32	4	32位有符号整数	–2147483648–2147483647
Uint32	4	32位无符号整数	0–4294967295
Float32	4	32位IEEE–754浮点数	–3.4e+38~+3.4e+38
Float64	8	64位IEEE–754浮点数	–1.7e+308~+1.7e+308

定型数组行为

1. 定型数组大部分的方法都可以照搬普通数组，但要记住，直接修改原数组的方法不可以用到定型数组上面

新增方法

1. set（）
    a. 从提供的数组或定型数组中把值复制到当前定型数组中指定的索引位置
    b. 参数：第一个参数必选，为提供的数组或者定型数组；第二个参数可选，偏移量，默认为0
2. subarray（）：
    a. 基于从原始定型数组中复制的值返回一个新的定型数组
    b. 复制时开始和结束的索引都是可选的

下溢和上溢

1. 定型数组的下溢和上溢不会影响到其他索引
2. 可以理解为每一个位置都是固定的，溢出的部分按照二进制固定位切除

Map

1. 一种新的集合类型

创建，增添，查询，删除

1. Map.set()：增添键值对，参数为一个键值对，返回新的集合
2. Map.get()：通过键名进行查询，返回对应的值（严格对象相等）
3. Map.has()：通过键名进行查询，返回布尔值（严格对象相等）
4. Map.delete：通过键名进行对应键值对的删除（严格对象相等）
5. Map.clear()：清除这个映射中的所有键值对
6. Map.size：获取该映射中键值对的数量

顺序与迭代

1. entries（）方法，keys（）方法，values（）方法：分别返回以插入顺序生成的键值对，键，值的迭代器
2. 通过for–of进行迭代就好了

优点

1. Map的内存占用更小
2. Map插入速度较Object快一点
3. Map的删除操作较快

WeakMap

1. 弱映射
2. 行为模式方法都与Map差不多
3. WeakMap的键只能是Object类型或者是继承自Object的类型，值的类型没有限制
4. 若没有指向键这个对象的引用，这个对象键会被自动回收
5. WeakMap的键值对不可迭代
6. 应用场景：保存关联元数据
    a. 保存DOM节点元数据，当对应的DOM 节点被删除后，若没有其他的关联，该键就会被销毁

Set

1. Set在大部分方面与Map类似（就把它当做键值都是同一个东西的Map好了）
2. 与Map不同的是，使用add（）方法增添元素
3. 定义一个实用的Set函数库（函数定义书上有，但是看不懂）

Set函数库 JavaScript | 复制代码

```
1
```

WeakSet

1. WeakSet与Set的关系跟WeakMap与Map的关系是一样的

迭代与扩展操作

1. 扩展操作符（...),扩展操作符对可迭代对象实行的是浅复制（意味着只会复制对象的引用）

迭代器与生成器

迭代

1. 迭代器是一个可以由任意对象实现的接口，支持连续获取对象产出的每一个值
2. 任何实现Iterable接口的对象都有一个Symbol.iterator属性，这个属性引用默认迭代器
3. 默认迭代器是一个工厂函数，调用之后会返回一个实现Iterator接口的对象
4. 迭代器必须通过连续调用next（）方法才可以连续获取值，这个方法返回一个IteratorObject。
    a. 这个对象包含一个done属性和一个value属性。
    b. 前者是一个布尔值，表示是否还有更多的值可以访问
    c. 后者包含迭代器返回的当前值
    d. 这个方法可以通过手动调用next（）方法进行消费，也可以通过原生消费，比如for-of循环来自动消费
5. 提前终止迭代器：
    a. 可选的return（）方法可以提前终止迭代器

b. for-of循环通过break，continue，return，或throw提前退出迭代

c. 解构操作并未消费所有值的时候提前退出迭代

d. 后面两个是表层的实现，a是迭代器内部的方法，是实现b，c的前提

迭代器 JavaScript 复制代码

```
const a = [1, 2, 3, "4", { b: 5, c: 6 }];

const a1 = [1];

// 调用数组的[Symbol.iterator]属性这个函数 返回一个迭代器
let iter = a[Symbol.iterator]();

let iter2 = iter[Symbol.iterator]();

// 迭代器与迭代器的迭代器全等
console.log(iter2 === iter); // true

// 通过break停止了迭代 但迭代器本身是没有停止的
for (const i of iter) {
    console.log(i);
    if (i > 2) {
        console.log(`stop or not?`);
        break;
    }
}
// 1
// 2
// 3
// stop or not?

for (const i of iter) {
    console.log(i);
}
// 4
// { b: 5, c: 6 }
```

生成器

1. 生成器是一种特殊的函数，调用之后返回一个生成器对象
2. 生成器对象实现了Iterator接口，可以用在任何消费可迭代对象的地方
3. 生成器的独特之处在于支持yield关键字，这个关键字能够暂停生成器函数

4. 使用yield关键字之后还可以通过next（）方法接收输入和产生输出（next是生成器函数继续运转）

生成器 JavaScript 复制代码

```
// 创建一个生成器
function* generatorFn(initial) {
  console.log(initial);
  console.log(yield);
              }

// 实例化一个生成器对象
const generatorObject = generatorFn('foo');

// 第一个next（）方法启动生成器对象，此传入值无效
generatorObject.next(`zro`);// foo
generatorObject.next(`abb`);// abb
// 因为最后没有代码了，再调用next（）也没有用了
generatorObject.next(`cdd`);
```

5. yield*可以将跟在它后面的可迭代对象序列化为一连串值

yield*实现迭代 JavaScript 复制代码

```
function* generatorFn() {
  // 使用yield可以很方便地进行迭代
    yield*[1, 2, 3];
}

const g = generatorFn();

for (const k of g) {
    console.log(k);
}
// 1
// 2
// 3
```

6. 使用yield*实现递归算法

yield*实现递归 JavaScript 复制代码

```javascript
function* generatorFn(n) {
    if (n > 0) {
        yield* generatorFn(n - 1);
        yield n - 1;
    }
}

const g = generatorFn(3);

for (const k of g) {
    console.log(k);
}
// 0
// 1
// 2
```

7. value属性是生成器的返回值，默认为undefined，可以通过生成器的返回指定

JavaScript 复制代码

```javascript
function* generatorFn() {
  // 注意这个是生成器的返回 后面还有一个生成器的方法return（） 是用来终止生成器的
    return `foo`;
}

const g = generatorFn();

console.log(g.next()); // { value: 'foo', done: true }
```

8. 提前终止生成器
    a. return（）方法用于提前终止生成器，进入了关闭状态之后就无法恢复了
    b. throw（）方法在暂停的时候会将错误注入到生成器对象中。
        i. 如果错误未被处理，生成器就会被关闭。
        ii. 如果错误被处理了，生成器会跳过对应的yield，但可以恢复执行

```javascript
function* generatorFn() {
    for (const x of[1, 2, 3]) {
        try {
            yield x;
        } catch (e) {}
    }
}

const g = generatorFn();

console.log(g.next());
// 这个结果与我想象中不太一样 调用throw（）方法后2还是被返回了 我不理解
// 并且如果调用throw（）方法 生成器内部不处理的话 编辑器是会报错的
// 如果再第一次调用next（）方法之前就使用throw（）方法的话 也会报错 因为此时生成器还没有启动
console.log(g.throw());
console.log(g.next());
// { value: 1, done: false }
// { value: 2, done: false }
// { value: 3, done: false }
```

对象、类与面向对象编程

对象

属性的类型

数据属性

1. 数据属性包含一个保存数据值的位置
2. 数据属性有四个特性：
    a. [[Configurable]]：属性是否可以通过delete删除并重新定义，是否可以修改它的特性，以及是否可以把它修改为访问器属性。默认为true
    b. [[Enumerable]]：表示属性是否可枚举（是否可以通过for-in循环返回）。默认为true
    c. [[Writable]]：表示属性值（[[Value]]）是否可修改。默认为true
    d. [[Value]]：属性值。默认为undefined

数据属性 JavaScript | 复制代码

```javascript
let person = {};

Object.defineProperty(person, `age`, {
    configurable: false,
    enumerable: true,
    writable: true,
    value: 3,
});

console.log(person.age);

Object.defineProperty(person, `name`, {
    value: 2,
});

console.log(person.age);
```

访问器属性

1. 访问器属性不包含数据值
2. 访问器属性有四个特性：
    a. [[Configurable]]：属性是否可以通过delete删除并重新定义，是否可以修改它的特性，以及是否可以把它修改为访问器属性。默认为true
    b. [[Enumerable]]：表示属性是否可枚举（是否可以通过for–in循环返回）。默认为true
    c. [[Get]]：获取函数，在读取访问器属性时，会调用获取函数，并返回一个有效值。默认为undefined
    d. [[Set]]：设置函数，在写入访问器属性时，会调用设置函数并传入新值，这个函数必须决定对数据做出什么修改。默认为undefined
3. 获取函数和设置函数不一定都要定义。只定义获取函数意味着属性是只读的，尝试修改属性会被忽略

对属性的操作

定义和修改属性

1. Object.defineProperty()方法用于一次设置一个属性
2. Object.defineProperties()方法可用于一次设置多个属性

3. 三个参数：要给其添加属性的对象、属性的名称、一个描述符对象（描述符对象上的属性可以包括四个特性，根据有没有数据位置来分辨数据属性和访问器属性）

读取属性特性

1. Object.getOwnPropertyDescriptor()方法
    a. 获取指定属性的属性描述符
    b. 两个参数：属性所在的对象、属性名
    c. 返回值：一个对象
2. Object.getOwnPropertyDescriptor()方法
    a. 获取某个对象全部属性的描述符
    b. 一个参数：某个对象
    c. 返回值：一个对象

合并对象

1. Object.assign()方法
2. 参数：一个目标对象、一个或多个源对象
3. 原理：将源对象中可枚举属性浅复制到目标对象（从源对象的[[Get]]取得属性的值，使用目标函数对象上的[[Set]]设置属性的值。

对象标识及相等判定

1. Object.is()方法
2. 参数：两个用于比较的对象
3. 若想检查超过两个值，函数如下：

对象标识及相等判定 JavaScript | 复制代码

```
// 函数 用于比较一个或多个值是否相等
function recursivelyCheckEqual(x, ...rest) {
    return Object.is(x, rest[0]) && (rest.length < 2 || recursivelyCheckEqual(...rest));
}

console.log(recursivelyCheckEqual(1, ...[1, 1, 1])); // true
console.log(recursivelyCheckEqual(...[1, 1, 1, 1])); // true
console.log(recursivelyCheckEqual([1, 1, 1, 1])); // false
console.log(recursivelyCheckEqual(1)); // false rest[0]是undefined，即没有第二个参数的情况下，默认为undefined
console.log(recursivelyCheckEqual(1, 1, 1, 1, 1, 1)); // true
```

增强对象的语法

1. 属性值简写：简写属性名只要使用变量名（不用冒号）就会被自动解释为同名的属性键（这是有同名变量的情况下）
2. 可计算属性：可以在对象字面量中完成动态属性赋值。中括号包围的对象属性键本身可以是复杂的表达式。
3. 简写方法名

增强对象的语法 JavaScript | 复制代码

```javascript
const methodKey = `sayName`;
let person = {
    name_: ``,
    // 简写了方法名
    get name() {
        return this.name_;
    },
    // 由name属性的get函数进行简单的获取  没有这个get函数并不影响设置this.name_的值
    // 再由name属性的set函数设置this.name_
    // 这样调用sayName () 的时候this.name_就是`Matt`了
    set name(name) {
        this.name_ = name;
    },
    // 这里使用可计算属性
    [methodKey]() {
        console.log(`My name is ${this.name_}`);
    }
};

person.name = `Matt`; // get和set都是访问器属性name的函数
person.sayName(); // My name is Matt
```

对象解构

1. 对象解构是使用与对象匹配的结构来实现对象属性赋值
2. 使用解构，可以在一个类似对象字面量的结构中，声明多个变量，同时执行多个赋值操作，如果想让变量直接使用属性的名称，可以使用简写语法（类似于属性值简写，调换过来了）
3. 如果引用的属性不存在，则该变量的值是undefined
4. 可以在解构赋值的同时定义默认值
5. 解构会把源数据结构转换为对象（导致原始值会被当成对象，也就是原始包装类型的属性和方法是可以使用的）

嵌套解构

1. 解构对于嵌套的属性或赋值目标没有限制

嵌套解构 JavaScript | 复制代码

```javascript
let person = {
    name: `Matt`,
    age: 16,
    job: {
        title: `software engineer`,
    }
};

let personCopy = {};

// 注意：这里是把一个对象的引用直接赋值给personCopy了 也就是只传了地址 一改全改
// 个人认为这里本质上是用一个对象作为过渡
// 但是我不明白为什么personCopy.name可以使用 不是没有定义吗
// 还是说 这一整块放进去括号里面 就默认声明了？
({
    name: personCopy.name,
    age: personCopy.age,
    job: personCopy.job
} = person);
console.log(personCopy);
```

创建对象

构造函数模式

1. 使用new操作符调用构造函数创建对象实例
    a. 在内存中创建一个新对象
    b. 在这个对象内部的[[Prototype]]特性被赋值为构造函数的prototype属性
    c. 构造函数内部的this指向新对象
    d. 执行构造函数内部代码（给新对象加属性和方法）
    e. 若构造函数返回非空对象O，则返回非空对象O；否则，返回刚创建的新对象
2. 构造函数的问题：
    a. 若方法函数在构造函数内部声明，则不同实例上的函数虽然同名但不相等
    b. 若方法函数统一放在外面声明，则自定义类型引用的代码不能很好地聚在一起

构造函数模式 JavaScript | 复制代码

```javascript
// 构造器函数
function Person(name, age, job) {
    console.log(this);
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayNmae = function() {
        console.log(this.name);
    }
}

// 实例化对象 类型为Person
let person1 = new Person(`kitty`, `22`, `worker`); // Person {}

// 若直接调用函数 内部this指向为window
Person(`hello`, `3`, `play game`); // 指向window
```

原型模式

1. 每个函数都会创建一个prototype属性，这个属性是一个对象，包含由特定引用类型的实例共享的属性和方法（这个就是原型）
2. 默认情况下，所有原型对象自动获得一个名为constructor的属性，指向与之关联的构造函数
3. 每次调用构造函数创建一个新实例，这个实例内部的[[Prototype]]指针就会被赋值为构造函数的原型对象
4. 实例与构造函数原型之间有直接联系（prototype指向原型），原型与构造器之间有直接联系（constructor指向构造函数，prototype指向原型）
5. 方法：
    a. Function.prototype.isPrototypeOf(Object)
        i. 确定两个对象时间是否是对应的原型与实例
        ii. Function.prototype是原型
        iii. Object是对象实例
    b. Object.getPrototypeOf(Object)
        i. 获取对象实例的原型
        ii. 这是一个Object类型的方法
        iii. 参数Object是对象实例

c. Object.create(Function.prototype)
    i. 创建一个新对象，同时为其指定原型
    ii. 参数为指定的原型，是一个对象
    iii. 返回值：对象

属性查找机制

1. 在通过对象访问属性的时候，会按照属性名进行查找
2. 搜索开始于对象本身，找到返回，找不到下一步
3. 搜索进入原型（进入prototype属性）进行查找，找到返回

遮蔽效果

1. 给对象实例添加一个属性，这个属性会遮蔽原型上的同名属性（即你引用这个属性名，得到的是你自己给对象实例设置的那个属性数据）
2. 使用delete操作符可以完全删除实例上的属性，取消遮蔽效果

in操作符

1. 单独使用：
    a. in操作符会在可以通过对象访问指定属性时返回true
    b. hasOwnPrototype()方法会在属性存在于调用它的对象实例上时返回true

确定属性是否在原型上 JavaScript | 复制代码

```javascript
// 函数 作用 确定某个属性是否存在于实例对象原型上
// Object是实例对象 name是属性名 为字符串
function hasPrototypeProperty(Object, name) {
    return !Object.hasOwnProperty(name) && (name in Object);
}

function Person() {
    Person.prototype.name = 'Mit';
    Person.prototype.age = 13;
}

const person = new Person;

console.log(hasPrototypeProperty(person, `name`)); // true

person.name = `Code`;

console.log(hasPrototypeProperty(person, `name`)); // false

for (k in person) {
    console.log(k);
}
```

2. for–in
    a. 可以通过对象访问且可以被枚举的属性会全部返回，遮蔽原型中不可枚举属性的实例属性也会被返回

属性枚举

1. for–in
2. Object.keys()方法：接受一个对象作为参数，返回包含该对象所有可枚举属性名称的字符串数组
3. Object.getOwnPrototypeNames()：接受一个对象作为参数，列出所有实例属性，无论是否可枚举
4. Object.getOwnPrototypeStmbol()：针对符号，与上面类似
5. 2、3、4的枚举顺序是确定的，先以升序枚举数值键，再插入顺序枚举字符串和符号建

对象迭代

1. Object.values()：接收一个对象，返回对象值的数组
2. Object.entries()：接收一个对象，返回键值对的数组

3. 注意：
    a. 非字符串属性会被转换为字符串输出
    b. 对执行对象进行的是浅复制
    c. 符号属性会被忽略

其他原型语法

1. 直接通过一个包含所有属性和方法的对象字面量来重写原型

原型的动态性

1. 任何时候对原型所做的修改也会在实例上面反映出来
2. 前提是你这个实例确确实实是指向这个原型（就是说你不是先创建实例再重写原型）

原型的问题

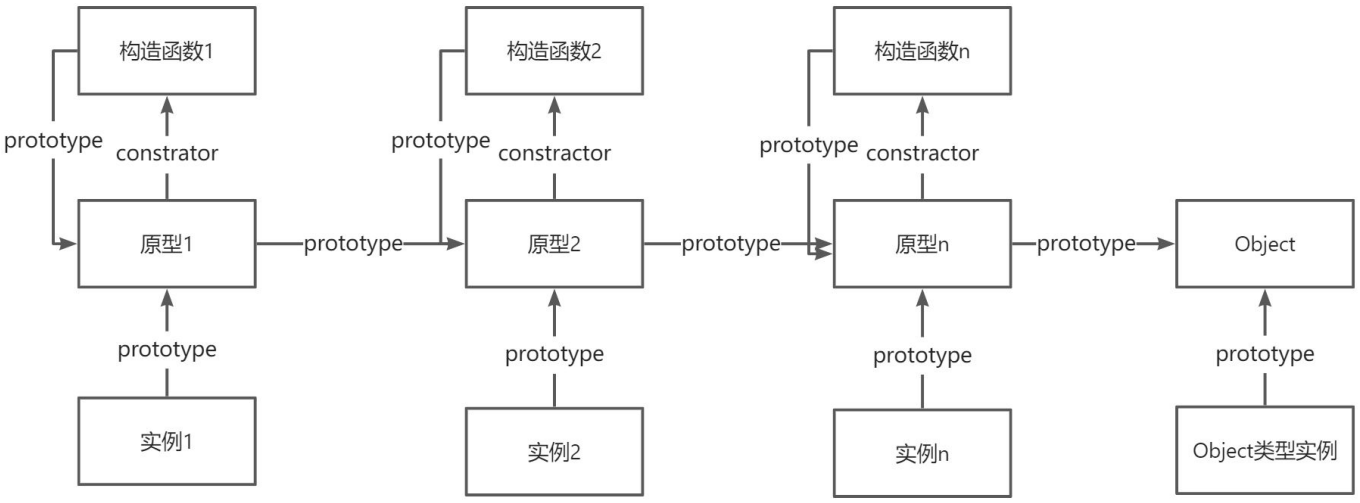
1. 原型的问题主要是共享性
2. 大家都是可以通过属性名访问原型的，数据一个改了就相当于全部改了

继承

1. 原型链是主要继承方式

原型链

1. 任何函数的默认原型都是一个Object的实例
2. 属性可以沿着原型链一直向上查找直到Object
3. 原型与继承关系：
    a. instanceof操作符：如果一个实例的原型链中出现过相应的构造函数，则返回true
    b. isPrototypeOf()方法，只要原型链中包含这个原型，就会返回true
4. 以对象字面量创建原型的方法会破坏之前的原型链，因为这相当于重写了原型链
5. 原型链的问题：原型链中包含引用值的时候，数据就共享了，大家都可以改
6. 原型链本链：



6. 具体实例：

原型链 JavaScript 复制代码

```
// 4. 在SuperType的方法上面寻找 找不到
function SuperType() {
    this.property = true;
}

// 5. 在SuperType的prototype的方法上面寻找 找到了 返回值是this.prototype 此值为true
SuperType.prototype.getSuperValue = function() {
    return this.property;
};

// 1. 在SubType的方法上面寻找 找不到
function SubType() {
    this.subProperty = false;
}

// 3. SubType的prototype定义在SuperType上面
SubType.prototype = new SuperType();

// 2. 在SubType的prototype的方法上面寻找 找不到
SubType.prototype.getSubValue = function() {
    return this.subProperty;
};

let instance = new SubType();

// 目的：寻找getSuperValue方法
console.log(instance.getSuperValue()); // true
```

盗用构造函数

1. 在子类构造函数中调用父类构造函数
2. 优点：可以在子类构造函数中向父类构造函数传参
3. 缺陷：必须在构造函数中定义方法，因此函数不能重用

组合继承

1. 综合了原型链和盗用构造函数，通过原型链实现方法，且通过盗用构造函数让每实例都有自己的属性
2. 具体用例：

组合继承 JavaScript | 复制代码

```javascript
// 组合继承实现了数据的私有以及方法的共享
function SuperType(name) {
    this.name = name;
    this.color = [`black`, `blue`, `white`];
}

// 这是原型链 定义方法
SuperType.prototype.sayName = function() {
    console.log(this.name);
};

// 这是盗用构造函数 定义属性
function SubType(name, age) {
    SuperType.call(this, name);
  // 增添新数据
    this.age = age;
}

SubType.prototype = new SuperType();

// 向自己的原型链增添新方法
SubType.prototype.sayAge = function() {
    console.log(this.age);
}

// 创建SubType的实例1
const instance1 = new SubType(`Marry`, 19);
instance1.color.shift();

console.log(instance1.color); // [ 'blue', 'white' ]
instance1.sayAge(); // 19
instance1.sayName(); // Marry

// 创建SubType的实例2
const instance2 = new SubType(`Tracy`, 23);
instance2.color.push(`gray`);

// 实例1与实例2的数据并不会互相影响
console.log(instance2.color); // [ 'black', 'blue', 'white', 'gray' ]
// 但实例1、实例2和原型链上的方法是共享的
instance2.sayAge(); // 23
instance2.sayName(); // Tracy
```

原型式继承

1. 原型式继承适用于不需要单独创建构造函数，但仍需要在对象间共享信息的场合
2. 原理：通过原型链来共享信息
3. Object.create()方法：
    a. 此方法将原型式继承概念化
    b. 两个参数：作为新对象原型的对象、给新对象定义额外属性的对象（可选）
    c. 以同一原型创建的对象实例间可以信息共享
4. 具体用例

原型式继承核心函数 JavaScript | 复制代码

```javascript
// 这个object（）函数会创建一个临时的构造函数，将传入的对象赋值给这个构造函数的原型，
// 然后返回这个临时类型的一个实例
// 本质上，object（）是对传入的对象进行了一次浅复制
function object(o) {
  function F() {}
  F.prototype = o;
  return new F;
}
```

寄生式继承

1. 创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象

寄生式组合继承

1. 组合继承存在效率问题：父类构造函数始终会被调用两次
2. 寄生式组合继承可以算是目前引用类型继承的最佳模式
3. 具体用例：

寄生式组合继承 JavaScript 复制代码

```javascript
// 寄生式的核心函数
function inheritPrototype(SubType, SuperType) {
    const prototype = new Object(SuperType.prototype); // 创建对象
    prototype.constructor = SubType; // 增强对象
    SubType.prototype = prototype; // 赋值对象
}

function SuperType(name) {
    this.name = name;
    this.color = [`black`, `blue`, `white`];
}

SuperType.prototype.sayName = function() {
    console.log(this.name);
}

function SubType(name, age) {
    SuperType.call(this, name);
    this.age = age;
}

// SubType.prototype = new SuperType(); 这一行被取代了
// 原本是直接创建一个SuperType实例作为SubType.prototype
inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function() {
    console.log(this.age);
}

const instance1 = new SubType(`Marry`, 19);
instance1.color.shift();

console.log(instance1.color); // [ 'blue', 'white' ]
instance1.sayAge(); // 19
instance1.sayName(); // Marry

const instance2 = new SubType(`Tracy`, 23);
instance2.color.push(`gray`);

console.log(instance2.color); // [ 'black', 'blue', 'white', 'gray' ]
instance2.sayAge(); // 23
instance2.sayName(); // Tracy
```