

基础部分(1-11章)

第一章什么是JavaScript

1. JavaScript包含: ECMAScript(核心), DOM(文档对象模型),BOM(浏览器对象模型)
2. Web浏览器只是ECMAScript实现可能存在的一种宿主环境, 还可能其他的宿主环境例如Nodejs。其中ECMAScript版本可以根据年份来命名或者根据版本来命名。
3. DOM:文档对象模型:用于在HTML中使用XML拓展, DOM节点将页面抽象成一组分层节点, HTML或者XML页面的每个组成部分都是一种节点包含不同的数据。**(提供与网页内容交互的方法和接口)**
4. BOM:浏览器对象模型API, 用于支持访问和操作浏览器的窗口。**(提供与浏览器交互的方法和接口)**

第二章HTML的JavaScript

基本概念

1. JavaScript插入到Html的方法就是使用 `<script>` 俩引入
2. **再html中使用JS的方式,在head后面, 或者body里面最后面**
3. **使用了src属性的元素不应该在再这个标签里面进行写代码了, 如果写了的话就会只下载并执行脚本文件, 从而忽略行内标签**
4. 会按照script的顺序来解析他们, 前提是没有defer和async属性
5. 动态加载脚本: `createElement()`
6. : 早期浏览器不支持JavaScript或者关闭了对脚本的关闭, 那么显示出来,

面试可能会问的

script的defer和async(来防止解析JavaScript脚本而阻塞)

当浏览器加载 HTML 并遇到 `<script>...</script>` 标签时, 它无法继续构建 DOM。它必须立即执行脚本。外部脚本 `<script src="..."></script>` 也是如此: 浏览器必须等待脚本下载, 执行下载的脚本, 然后才能处理页面的其余部分。

区别:

1. async 执行与文档顺序无关, 先加载哪个就先执行哪个; defer会按照文档中的顺序执行。defer只对外部脚本有效
2. async 脚本加载完成后立即执行, 可以在DOM尚未完全下载完成就加载和执行; 而defer脚本需要等到文档所有元素解析完成之后才执行

 image-20240703161016251

行内脚本和外部脚本区别

行内脚本:直接把 js 代码写在 html 文件中

外部脚本:通过src引入,推荐使用:可维护性、缓存

放的位置

当把script元素放到了head标签里面，需要就是将这个脚本代码进行解析之后才能去解析HTML,渲染页面，对于需要很多js的页面就会堵塞页面使浏览器窗口出现空白。为了解决这个问题，就将js引用放在元素中的页面后面，这样就可处理JavaScript之前完全渲染页面，会感到加载页面更快了，空白页面时间就更加短了。

第三章语言基础

基本语法

1. 标识符:第一个字符必须是一个字母、下划线或者美元，其他的可以是字母、下划线或者美元或者数字
2. 严格模式
3. 关键字(do,continue,return,break,class,new)和保留字(enum,implements,package,let)

数据类型

1. 原始类型:Number,String,Boolean,Null,Undefined,Symbol。复杂数据类型:Object
2. undefined是定义了但是没有初始化
3. 浮点数使用的内存空间是存储整数值的2倍
4. 值的范围:最大值(Number.MAX_VALUE),最小值(Number.Min_Value),如果计算，如果某个计算得到的数值超过了JavaScript可以表示的范围，那么这个数值自动转化为特殊的Infinity，任何无法表示的负数以-Infinity表示，要确定一个值是不是有限大(isFinite())
5. NaN

```
0/0    //NaN
-0/+0  //NaN
NaN == NaN //false
isNaN()
```

isNaN()可以用于测试对象，首先会调用对象的valueOf()方法，然后在确定返回的值是否可以转化为数值，不能的话就调用toString()方法。

6. 数值转化:

Number转化

```
Number("001") //1
Number("0xf") //显示十进制
//如果是对象的话，调用valueOf()方法,并按照上述规则转化，如果为NaN，则调用toString()方法，按照转换字符串的规则转化
```

parseInt转化

```
parseInt()//第一个字符如果不是数值字符、加号或者减号，parseInt()立即返回NaN,空字符串也是转化为NaN
let num1 = parseInt("123blue") //123
let num2 = parseInt("") //NaN
let num3 = parseInt("0xA") //10
//可以有第二个参数，来解析为几进制
parseInt("10",2)
parseInt("10",8)
parseInt("10",10)
parseInt("10",16)
```

parseFloat() 直接解析为十进制

```
parseFloat('1234blue') //1234
parseFloat('0xA') //0
parseFloat('22.34.5')//22.34
parseFloat('0908.5') //908.5
parseFloat('3.125e7') //31250000
```

String:转化为字符串，toString()方法，其中null和undefined没有这个方法

其中这个可以传递参数(传递几进制)。String(null) 为"null",String(undefined)为"undefined"

原始字符串:String.raw``

7. Symbol: 符号实例是唯一、不可变的。确保对象属性使用唯一标识符，不会发生属性冲突危险。重用全局注册表: Symbol.for("foo"), Symbol.keyFor("foo") 查询全局注册表的key

操作符

位操作符

1. 非~: 取反
2. 与&:1 1 1,0 0 0,1 0 0,0 1 0
3. 或 |:1 1 1,1 0 1,0 1 1,0 0 0
4. 位异或 ^:1 1 0,0 0 0,1 0 1,0 1 1
5. 左移 <<:相当于×2
6. 有符号右移 >>:相当于/2
7. 无符号右移 >>>:负数的话右移会把它当成正数

布尔操作符

1. !!:把任意值转化为布尔，相当于Boolean
2. 逻辑与和逻辑或:&&和||

乘性操作符

1. 乘法:如果不能表示乘积, 则返回Infinity或者-Infinity

规则:

- 如果任意一方有NaN, 则返回NaN
- 如果Infinity * 0, 则返回NaN
- 如果Infinity 乘以非0的有限数值, 则根据第二个操作符的符号返回+Infinity或者-Infinity
- 如果时Infinity乘以Infinity, 则返回Infinity

2. 除法:

- 如果Infinity / Infinity, 则返回NaN
- 0 / 0, 则返回NaN
- 如果是非0的有限值除以0, 则根据第一个操作符的符号返回Infinity胡哦这-Infinity
- 如果时Infinity / 任何数值, 根据第二个操作数的符号返回Infinity胡哦这-Infinity

3. 取模操作符

- 如果被除数是无限值, 除数是有限值, 返回NaN
- 被除数是有限值, 除数是0, 返回NaN
- Infinity / Infinity, 返回NaN
- 除数是无限值, 被除数是有限值, 返回被除数
- 除数不是0, 被除数是0, 返回0

指数操作符

** 或者Math.pow

加性操作符

1. 加法

- Infinity + -Infinity返回NaN
- -0 + +0返回+0

2. 减法

- +0 - +0 ,+0
- +0 - -0,-0
- -0 - -0,+0
- 如果任一操作数是字符串、布尔值、null或者undefined, 首先在后台使用Number()将其转化为数值, 然后再根据前面的规则执行数学运算。如果转化结果是NaN, 则减法计算结果是NaN

关系运算符

1. 大写字符串编码 < 小写字符串编码
2. 比较NaN的时候, 无论是小于还是大于或者等于, 比较的结果都会返回false
3. ==中undefined和null不能转化为其他的类型再进行比较
4. null == undefiend,null != 0,undefined != 0

语句

函数

面试可能问到的

var、let和const区别

1. var有**变量提升**，let和const没有，var还可以**重复反复声明一个变量**
2. let的暂时性死局
3. 与var关键字不同，使用let在全局作用域中生命的变量不会成为window对象属性。
4. 条件声明:let作用域是块，所以不可能检查前面是否已经使用let声明过同名变量，**而且不能使用try/catch语句或者typeof操作符来实现，因为条件块中let声明的作用域仅仅限于该块**
5. 例子:for循环的let声明，以前使用的是**var会出现渗透到循环体外部，但是let不会仅限于for循环块内部**。还有一个问题：var的话退出循环，迭代变量保存的是导致循环退出的值，之后执行超时逻辑的时候还是同一个i。let的话每次迭代会生成一个新的变量。
6. const就相当于一个常量。const限制只适用于它指向的变量的引用，如果指向的是一个对象，那么修改这个对象内部的属性并不违背const限制。
7. 多使用const > let > var:const声明变量值是单一变量是不可修改的，JavaScript**运行时的编译器可以将其所有的实例都换成实际的值，而不会通过查询表进行变量查找**。谷歌V8引擎就执行了这种优化。

8. 常见js面试题：

当程序执行到包含 let 或 const 声明的代码块时，会创建一个称为**暂时性死区的区域**，该区域从声明开始直到块结束。在这个区域内，变量虽然已经被声明，但是在声明之前访问该变量会导致引擎抛出一个错误。

这种行为是为了解决 JavaScript 中变量声明提升带来的一些问题。通过暂时性死区，可以在变量被声明之前阻止对变量的访问，从而避免了在变量未初始化的情况下使用它，增强了代码的可靠性。

```
let i = 1
{
  //死区开始
  console.log(i) //死区里边拿不到外边的i，也拿不到本代码块内的i
  //死区结束
  let i = 2
  console.log(i) //直到这里才能正常使用 i
}
```

typeof和instanceof操作符

特殊的typeof null = 'object',函数的话也为object

typeof undefined = 'undefined'

`instanceof` 运算符用于检测构造函数的 `prototype` 属性是否出现在某个实例对象的原型链上

`Object.prototype.toString` 方法返回一个表示该对象的字符串

```
const toString = Object.prototype.toString;
toString.call(new Date()); // [object Date]
toString.call(new String()); // [object String]
```

手写instanceof

```
function myInstanceOf(left, right) {
  let proto = Object.getPrototypeOf(left); // 获取对象的原型
  let prototype = right.prototype; // 获取构造函数的 prototype 对象

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;

    proto = Object.getPrototypeOf(proto);
  }
}
```

Object.prototype.constructor()注意，此属性的值是对函数本身的引用，而不是一个包含函数名称的字符串。

```
const o = {}
o.constructor === Object // true

const o = new Object
o.constructor === Object // true

const a = []
a.constructor === Array // true

const a = new Array
a.constructor === Array // true

const n = new Number(3)
n.constructor === Number // true
```

Object.prototype.toString

```
const o = {}
o.constructor === Object // true

const o = new Object
o.constructor === Object // true

const a = []
a.constructor === Array // true

const a = new Array
a.constructor === Array // true
```

```
const n = new Number(3)
n.constructor === Number // true
```

以这种方式检查类是不可靠的，因为继承自 `Object` 的对象可能用它们自己的实现重写它。

```
const myDate = new Date();
Object.prototype.toString.call(myDate); // [object Date]

myDate[Symbol.toStringTag] = "myDate";
Object.prototype.toString.call(myDate); // [object myDate]
```

undefined和null相等以及他们的区别

undefined是由null值派生而来的，定义为表面上相等。

历史原因:原来，这与JavaScript的历史有关。1995年[JavaScript诞生](#)时，最初像Java一样，只设置了null作为表示"无"的值。

根据C语言的传统，null被设计成可以自动转为0。

但是，JavaScript的设计者Brendan Eich，觉得这样做还不够，有两个原因。

首先，null像在Java里一样，被当成一个对象。但是，JavaScript的数据类型分成原始类型（primitive）和合成类型（complex）两大类，Brendan Eich觉得表示"无"的值最好不是对象。

其次，JavaScript的最初版本没有包括错误处理机制，发生数据类型不匹配时，往往是自动转换类型或者默默地失败。Brendan Eich觉得，如果null自动转为0，很不容易发现错误。

因此，Brendan Eich又设计了一个undefined。

区别:null是一个表示"无"的对象，转为数值时为0；undefined是一个表示"无"的原始值，转为数值时为NaN。

目前用法:

undefined

- (1) 变量被声明了，但没有赋值时，就等于undefined。
- (2) 调用函数时，应该提供的参数没有提供，该参数等于undefined。
- (3) 对象没有赋值的属性，该属性的值为undefined。
- (4) 函数没有返回值时，默认返回undefined。

null

- (1) 作为函数的参数，表示该函数的参数不是对象。
- (2) 作为对象原型链的终点。

==和===

区别:==是判断值是否相等, ===是判断数据类型+值

==会进行强制数据类型转化, ===不会进行转化

== 将执行类型转换

=== 将进行相同的比较, 不做类型转换

Object.is 除下面三个值外, 表现与 === 相同

-0 和 +0 不相等

Object.is(NaN, NaN) 为 true

||=、&&= 和 ??= 是什么?

逻辑或赋值运算符 `||=` 的含义是: 如果 x 为假, 将 y 赋值给 x, 即:

```
if (!x) {  
  x = y  
}
```

逻辑与赋值运算符 `&&=` 的含义是: 如果 x 为真, 将 y 赋值给 x, 即:

```
if (x) {  
  x = y  
}
```

逻辑空赋值运算符 `x ??= y` 的含义是: 如果 x 为空值 (`null` 或 `undefined`), 将 y 赋值给 x, 即

可选链

`?.` 运算符的功能类似于 `.` 链式运算符, 不同之处在于, 在引用为空 (`null` 或者 `undefined`) 的情况下不会引起错误, 返回 `undefined`。

forEach和for...in和for...of

js中的forEach、for-in-、for-of的区别是: forEach一般用于对数组的遍历, 不用于对对象的遍历, 此方法会对数组中的每一个值进行遍历, 直至全部遍历完成。故在其中的return, break会失效。for in一般用于遍历对象, 循环遍历对象的key, 不推荐遍历数组。for of一般用于遍历对象, 循环遍历对象的value, 与forEach不同的是, 它可以正常的响应break, return, continue。

for..in:会遍历原型链上的东西。通过 `Array.prototype` 添加了 `ufo` 属性, 由于继承和原型链, 所以 `arr` 也继承了 `ufo` 属性, 属于可枚举属性, 所以 `for...in` 会遍历出来, 而 `for...of` 则不会。

推荐:for..in遍历对象的键(会遍历原型上的属性和数组上的可枚举的属性), 使用for...of遍历数组的值。

with语句

应用场景:针对一个对象反复操作

影响性能并且难于调试其中的代码


```
//不用with
let qs = location.search.substring(1)
let hostname = location.hostname
let url = location.href
//不用with
with(location){
    let qs = search.substring
    let hostname = hostname
    let url = href
}
```

?? 和 || 和 ?? =

?? 当前面的为null或undefined的时候就选择后面的值

?? =当前面是null或undefined才进行赋值

||是null、undefined、"、NaN、0都会执行后面的

第四章变量、作用域与内存

原始值与引用值

1. 原始值是值的引用，引用值是保存在内存中的对象。原始值是存放在栈中的，引用值是放在堆中的。
2. 动态属性:对于引用值而言，可以随时添加、修改和删除其属性和方法。原始值不行。
3. 复制值:原始值赋值的到新变量的位置，但是引用值是复制的指针，执行存储在堆内存中的对象，当一个对象发生改变的时候会在另一个对象上面反映出来。
4. 传递参数(函数):参数的传递是值的引用。**特殊:即使对象是按值传进函数的，obj也会通过引用访问对象，当函数内部给obj设置了name属性，函数外部的对象也会反应这个变化。证明一下这个参数是按值传递的。**

```
function setName(obj) {
    obj.name = "Nicholas";
    obj=newObject();
    obj.name="Greg";
}
let person = new Object();
setName(person);
console.log(person.name);    // "Nicholas"
```

例子中函数参数传入person，在这里传入的是person保存的堆内存中对象的引用，而不是person本身，所以当在函数内部参数被重新赋予新的对象，不会导致person保存的引用改变

执行上下文和作用域

1. 作用域链:指程序源代码中定义变量的那个区域,当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做作用域链。
2. 局部作用域定义的变量可用于在局部上下文中替换全局变量。

3. 内部上下文可以通过作用域链访问外部上下文的一切内容，但是外部上下文无法访问内部上下文的东西。
4. 作用域链的加强:某些语句会导致在作用域链前端临时添加一个上下文，这个上下文在代码执行后被删除。（try/catch with）
5. 变量声明:var换成let、const

垃圾回收

1. JavaScript使用的是垃圾回收的语言，就是说**执行的环境 负责在代码执行时管理内存**。
2. 主要使用的标记策略有:**标记清理和引用计数**。
3. 最常用的是标记清理法:首先给所有的变量添加标记，然后他会将所有在上下文的变量，已经被上下文中的变量引用的变量的标记去掉。在此之后再被标记的变量就是准备回收的变量。
4. 引用计数:对每一个值都记录他被引用的次数，初始化为1，当同一个值又被赋给另一个变量，引用值加一，当该值的引用变量被其他值的引用变量所覆盖了，那么引用数-1。当为0的时候就会被回收。但是出现循环引用就会出现这个问题。当对象A有一个指针指向对象B，对象B也引用了对象A，在内存中就不会回收。

```
function problem() {  
    let objectA = new Object();  
    let objectB = new Object();  
    objectA.someOtherObject = objectB;  
    objectB.anotherObject = objectA;  
}  
//解决办法给他们手动的设置为null
```

5. 以前的IE8及以前的版本，并非所有的对象都是JavaScript对象，**BOM和DOM中的对象都是C或者C++实现(COM),COM使用的是引用计数实现垃圾回收**。为了改变这个情况，IE9之后都改为了JavaScript对象，避免了存在两套垃圾回收算法导致问题。
6. 性能方面:垃圾回收调度很重要，因为会因为这个内存有限的条件下，垃圾会后机制有可能会明显拖慢渲染速度和帧率。现代垃圾回收程序会基于对JavaScript运行时环境的探测来决定何时运行。探测机制因引擎而异，但基本上都是根据已分配对象的大小和数量来判断的。
7. 现在通过变量的多少来动态的改变阈值，变量越多，相应的阈值就设置高一些。
8. 内存管理:let const 多使用，可以更早的让垃圾回收机制介入
9. 隐藏类和删除操作: 是的，V8 引擎在将解释后的 JavaScript 代码编译为实际的机器码时，会使用“隐藏类”（hidden classes）的技术。隐藏类是一种在 JavaScript 中实现多态性的方式，它允许在运行时动态地创建和修改 JavaScript 对象的类型。
在 V8 引擎中，每个 JavaScript 对象都有一个隐藏类，用于描述该对象的类型和属性。当对象的类型发生变化时，V8 引擎会自动更新其隐藏类。这种技术可以使得 JavaScript 代码更加灵活和高效，因为它允许在运行时动态地创建和修改对象类型，从而避免了不必要的类型转换和内存分配。
对于注重性能的 JavaScript 开发者来说，了解隐藏类的技术非常重要。通过使用隐藏类，可以避免不必要的类型转换和内存分配，从而提高 JavaScript 代码的执行效率。此外，隐藏类还可以用于实现 JavaScript 中的继承和多态性等特性，进一步提高代码的灵活性和可扩展性。
10. 优化的方法:1.一次添加对象的所有属性，避免先创建对象再补充属性 2.不用的属性将其值置为null

```
function Article(opt_author) {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = opt_author;
}
let a1 = new Article();
let a2 = new Article('Jake');
```

//没优化前

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = 'Jake';
}
let a1 = new Article();
let a2 = new Article();
delete a1.author;
```

//优化后

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = 'Jake';
}
let a1 = new Article();
let a2 = new Article();
a1.author = null;
```

面试会问到的

判断类型

typeof和instanceof

typeof是可以判断变量是否为原始类型，对于引用类型的话用处不大，需要使用instanceof

Object.freeze()

冻结对象，不能再给这个对象来进行添加和删除属性了，相当于只读了。

注意对象一旦被冻结之后该对象的原型也不可以被修改

```
function Test() {
  this.a = 1;
  this.b = 2;
}
Test.prototype.c=3
let obj = new Test();
Object.freeze(obj);
obj.__proto__={aaa:111}
console.log(obj) // Error: #<Test> is not extensible 是不可扩
```

但是原型中的属性却可以被修改，这是因为Object.freeze方法是浅冻结，也就是说只会冻结一层：

```
function Test() {
```

```

    this.a = 1;
    this.b = 2;
  }
  Test.prototype.c=3
  let obj = new Test();
  Object.freeze(obj);
  obj.__proto__.c=4
  console.log(obj.c) // 4

```

// 原型也是一个对象，根据以下结构可以看出，原型里又是一个对象，所以原型对象里的属性是冻结不了的

```

// obj:{
//   a:1,
//   b:2,
//   原型:{
//     c:3
//   }
// }

```

不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。

```

let obj={
  a:1,
  b:2
}
Object.defineProperty(obj,"a",{
  value:1,
  //enumerable:false
  //configurable:false,
  //writable:false
})
console.log(obj) // Cannot redefine property: a at Function.defineProperty

```

我们可以用 `Object.isFrozen` 方法来检测一个对象是否是被冻结过的对象，

参数可以传入原始值，对象，数组

闭包带来的内存泄漏

手写Object.freeze()

第五章基本引用类型

Date

1. `Date.parse()`、`Date.UTC`、`Date.now()`：返回表示方法执行时日期和时间的毫秒数
2. `toLocaleString`、`toString()`、`valueOf`：该方法被重写后返回的不是字符串，返回的是日期的毫秒数
3. 一些方法:`getFullYear()`、`getMonth()`、`getDate()`、`getDay()`：星期、`getHours()`、`getMinutes()`、`getSeconds()`

RegExp

1. g、i、m:多行模式、y: 战俘模式, 表示只查找lastIndex开始及以后的字符串
2. /为转义符号
3. 实例属性:global、ignoreCase、unicode、sticky、lastIndex、multiline、dotAll、flags
4. 实例方法:exec(): 返回来的有index:字符串匹配模式的起始位置和input属性:返回来的匹配的字符串, 如果设置了全局标识, 每次调用她都会在字符串中向前搜索下一个匹配项

```
let text = "cat, bat, sat, fat";
let pattern = /.at/g;
let matches = pattern.exec(text);
console.log(matches.index);           // 0
console.log(matches[0]);              // cat
console.log(pattern.lastIndex);       // 3
matches = pattern.exec(text);
console.log(matches.index);           // 5
console.log(matches[0]);              // bat
console.log(pattern.lastIndex);       // 8
matches = pattern.exec(text);
console.log(matches.index);           // 10
console.log(matches[0]);              // sat
console.log(pattern.lastIndex);       // 13
```

5. test()来进行判断文本与模式是否匹配
6. 构造函数属性:RegExp.input,leftContext,rightContext,lastMatch,lastParen

原始值包装类型

1. String、Boolean、Number
2. 引用类型和原始值包装类型主要区别在于的生命周期。通过new实例化引用类型后, 得到的实例会离开作用域被销毁, 自动创建的包装对象只存在于访问它的那行代码执行期间, 也就是不能在运行时给原始值添加属性和方法。
3. Object构造含糊作为一个工厂函数, 根据传入值的类型返回相应的原始值包装类型的实例。
4. Boolean:不要使用Boolean包装对象
5. Number:toFixed()、处理浮点计算不一定得到精确的结果。toExponential,toPrecision(),isInteger() 是否为整数, Number.isSafeInteger()
6. String:charAt(),charCodeAt()可以查看指定码元的字符编码, concat, slice(),substr(),substring()
7. indexOf、lastIndexOf、startsWith、endsWith、includes、trim、trimLeft、trimRight、repeat、padStart、padEnd、split、toLowerCase、toUpperCase、match、search、replace、localeCompare

单例内置对象

Global

1. `encodeURIComponent`和`encodeURIComponent`用于编码统一资源标识符。区别:`encodeURIComponent`不会编码属于URL组件的特殊字符, 比如冒号、斜杠、问号和井号, `encodeURIComponent`会编码所有非标准字符。`decodeURIComponent`和`decodeURIComponent`
2. `eval`: 是一个es的解释器, 严格模式中内部创建的变量很熟无法被外部访问, 也不能给`eval`赋值

```
eval("console.log(msg)")  
=>  
console.log(msg)
```

3. `window`对象:为Global对象的代理
4. `Math.PI`,`max()`,`min()`,`ceil()`,`floor()`,`round()`,`random()`,`abs()`,
5. `pow()`

Math

第六章集合引用类型

Array

1. 方法:`from`,`of`,`keys()`,`values()`,`entries()`,`copyWithin()`,`fill()`,`join()`,增删改,`sort()`,`reverse()`,`concat()`,`slice()`,`splice()`:删除、插入、替换的功能,`indexOf`,`lastIndexOf`,`includes()`,`find()`,`findIndex()`,`every()`,`filter()`,`forEach()`,`map()`,`some()`,`reduce()`,`reduceRight()`
2. `forEach`与`map`的最大区别就是`forEach`不能返回数组, `map`能返回数组对原来的数组不会发生改变, `forEach`会发生改变

Map

1. 根据键值来存储Object方便高效地完成, 使用对象属性作为键, 属性作为引用值。
2. 基本方法:`set()`,`get()`,`has()`,`delete()`,`clear()`, `keys()`,`values()`
3. 可以使用任何JavaScript数据类型作为键
4. 使用`map`还是`object`:
 - (1) 给固定大小的内存, `Map`(使用的是hash表和链表)大小可以比`Object`(使用的是数组)多存储50%的键/值对
 - (2) `Map`比较快一些, 尤其设计大量的插入操作
 - (3) 查询:`Object`比较快一些
 - (4) 删除:`Map`的较快

Set

1. 方法: add, get, has, clear, delete, values, keys, enteries
2. 去重

面试

slice, substr, substring区别

slice对于负值参数都当成字符串长度加上负参数值。

substr将第一个负参数值当成字符串长度加上该值，第二个负参数转化为0

substring：是负参数都转化为0

Map和weakMap区别

1. Map对键的引用是强引用，weakMap是弱引用
2. Map的key可以是任何值，但是weakMap必须是object或者继承了object的类
3. WeakMap不会妨碍垃圾回收机制，当没有别的地方引用了该键，就会被回收。但是Map的话只有明确的解除了关系才能被回收
4. weakMap是不可迭代的，因为不知道这个什么时候被回收，所以这个迭代性就没有必要，而且也没有size方法，clear

weakMap使用场景

`weakMap` 对象的一个用例是存储一个对象的私有数据或隐藏实施细节。

```
const privates = new weakMap();

function Public() {
  const me = {
    // 私有数据
  };
  privates.set(this, me);
}

Public.prototype.method = function () {
  const me = privates.get(this);
  // ...
};

module.exports = Public;
```

Map和Object区别

1. Object是有原型的，原型链上的键名有可能和你自己在对象上的设置的键名产生冲突。
2. Map键可以是**任意值**，包括函数、对象或任意基本类型。`Object` 的键必须是一个 `String` 或是 `Symbol`。
3. Map有序。`Object` 的键目前是有序的，顺序是复杂的。因此，最好不要依赖属性的顺序
4. Map可通过 `size` 属性获取。Object键值对个数只能手动计算。
5. Map在频繁增删键值对的场景下表现更好
6. Map没有元素的序列化和解析的支持。可以自己实现这个需求。Object可以使用 `JSON.stringify()`、`JSON.parse()` 互相转换。

set和map的解构,set有forEach嘛

Array的静态方法和实例方法

静态方法:`Array.isArray()`,`Array.from()`将类数组对象或可迭代对象转换为数组, `Array.of()`创建一个新的数组实例

实例方法:增删改查, 查询,`concat`,`slice`,`split`,`forEach`,`map`,`filter`,`reduce`,`join`,`sort`,`reverse`,`flat`

改变原数组的:增删改查, `sort`, `reverse`, `copyWithin`, `fill`

实现浅拷贝的:`slice`,`concat`,`Array.from`

`slice`第二个参数为-1:

如果 `slice()` 的第二个参数是负数, 它将被视为 `length + end`, 其中 `length` 是数组的长度。因此, `slice(1, -1)` 表示从第二个元素开始 (索引为 1), 到倒数第二个元素结束 (索引为 `length - 1`) 。

`splice`:

```
splice(start, deleteCount, item1, item2, ...):
```

第七章 迭代器与生成器

- 可迭代协议 (iterable protocol)
- 迭代器协议 (iterator protocol)

可迭代协议

可迭代协议**允许 JavaScript 对象定义或定制它们的迭代行为**, 例如, 在一个 `for..of` 结构中, 哪些值可以被遍历到。

要成为可迭代对象, 该对象必须实现 `@@iterator` 方法, 可通过常量 `[Symbol.iterator]` 访问该属性, 它的返回值为一个符合迭代器协议的对象。

迭代器协议

[Symbol.iterator] 返回的对象就是一个迭代器对象。它提供 next() 方法，返回一个具有以下属性的对象：

value: 迭代器返回的任何 JavaScript 值。done 为 true 时可省略。

done: 迭代器能否返回下一个值，不能则为 false，否则为 true。

内置可迭代对象

- String、Array、TypedArray、Map、Set 都是内置的可迭代对象，因为它们的每个 prototype 对象都实现了 @@iterator 方法。
- arguments 对象和一些 DOM 集合类型，如 NodeList 也是可迭代的。目前，没有内置的异步可迭代对象。
- 生成器函数（generator）返回生成器对象，它们是可迭代的迭代器。

实现一个迭代器

看手写js

生成器(generator)

常规函数只会返回一个单一值（或者不返回值），而 ES6 的 generator 可以按需一个接一个地返回（yield）多个值。

```
function* foo() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
let f = foo();  
  
console.log(f.next()); // { value: 1, done: false }
```

yield* 表达式迭代操作数，并产生它返回的每个值。我们可以使用 yield* 返回数组的每个元素：

```
function* yieldArrayElements() {  
  yield 1;  
  yield* [ 20, 30, 40 ];  
}  
  
let a = yieldArrayElements();  
  
console.log(a.next()); // { value: 1, done: false }  
console.log(a.next()); // { value: 20, done: false }  
console.log(a.next()); // { value: 30, done: false }  
console.log(a.next()); // { value: 40, done: false }
```

generator是可迭代的

```
function* generate() {
  console.log("invoked 1st time"); // "invoked 1st time"
  yield 1;
  console.log("invoked 2nd time"); // "invoked 2nd time"
  yield 2;
}
let gen = generate();
console.log(gen); // Object [Generator] {}

// for...of
for (const g of gen) {
  console.log("for...of g", g);
}
```

第八章对象、类与面向对象编程

理解对象

1. 属性分为:数据属性和访问器属性
2. 数据属性:[[Configurable]]、[[Enumerable]]、[[Writable]]、[[Value]],可以通过Object.defineProperty()来定义这些
3. 访问器属性:[[Configurable]]、[[Enumerable]]、[[Get]]、[[Set]]

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },
  edition: {
    value: 1
  },
  year: {
    get() {
      return this.year_;
    },
    set(newValue) {
      if (newValue > 2017) {
        this.year_ = newValue;
        this.edition += newValue - 2017;
      }
    }
  }
});
```

4. 读取属性特性:Object.getOwnPropertyDescriptor()可以获得指定属性的属性描述符。

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
```

```

    },
    edition: {
      value: 1
    },
    year: {
      get: function() {
        return this.year_;
      },
      set: function(newValue){
        if (newValue > 2017) {
          this.year_ = newValue;
          this.edition += newValue - 2017;
        }
      }
    }
  });
let descriptor = Object.getOwnPropertyDescriptor(book, "year_");
console.log(descriptor.value);           // 2017
console.log(descriptor.configurable);    // false
console.log(typeof descriptor.get);      // "undefined"
let descriptor = Object.getOwnPropertyDescriptor(book, "year");
console.log(descriptor.value);           // undefined
console.log(descriptor.enumerable);      // false
console.log(typeof descriptor.get);      // "function"

```

Object.definePropertyDescriptors()传入对象，可以获得其中属性的访问器属性之类的。

5. Object.assign():浅复制。其中get和set无法进行赋值

```

let dest = {}, src = {a:{}}
Object.assign(dest, src)
console.log(dest.a === src.a) //true

```

复制期间错误，则操作会终止，同时抛出错误，Object.assign()没有回滚之前赋值的概念。

6. 对象标识及其相等判定：Object.is(),因为有的啥时候===也无能为力了

```

console.log(Object.is(true, 1)); // false
console.log(Object.is({}, {})); // false
console.log(Object.is("2", 2)); // false
// 正确的0、-0、+0 相等/不等判定
console.log(Object.is(+0, -0)); // false
console.log(Object.is(+0, 0)); // true
console.log(Object.is(-0, 0)); // false
// 正确的NaN相等判定
console.log(Object.is(NaN, NaN)); // true

```

其余的是

```
// 这些是===符合预期的情况
console.log(true === 1);    // false
console.log({} === {});    // false
console.log("2" === 2);    // false
// 这些情况在不同JavaScript引擎中表现不同，但仍被认为相等
console.log(+0 === -0);    // true
console.log(+0 === 0);     // true
console.log(-0 === 0);     // true
// 要确定NaN的相等性，必须使用极为讨厌的isNaN()
console.log(NaN === NaN);  // false
console.log(isNaN(NaN));   // true
```

7. 属性值简写和可计算属性，简写方法名

8. 对象解构:可以起别名、给默认值，解构在内部使用函数ToObject()把源数据结构转化为对象。null和undefined不能被解构，否则会报错。部分解构:当其中一个解构出现错误，就会从这里终止，后面都变为undefined

创建对象

构造函数创建、字面量创建、Object.create() 方法创建一个新的对象，使用现有的对象作为新创建对象的原型、类、工厂模式、原型模式、组合模式

```
function createPerson(name) {
  let o = new Object();
  o.name = name;
  o.sayName = function() {
    console.log(this.name);
  };
  return o;
}
```

```
let person1 = createPerson("Lucy");
let person2 = createPerson("Joe");
```

优点：可以解决创建多个类似对象的问题

缺点：没有解决对象标识问题（即新创建的对象是什么类型，类型如 Array）

自定义构造函数模式的优缺点如下：

优点：可以确保实例被标识为特定类型，相比于工厂函数，这是一个很大的好处。

缺点：定义的方法在每个实例上都创建一遍。

使用原型对象的好处是，在它上面定义属性和方法可以被对象实例共享

```
function Person() {}

// 将属性和方法添加到 prototype 属性上
Person.prototype.name = "Lucy";
Person.prototype.sayName = function() {
  console.log(this.name)
}

let person1 = new Person();
person1.sayName(); // "Lucy"
```

```
let person2 = new Person();
person2.sayName(); // "Lucy"
。
```

```
function Person(name) {
  this.name = name;
}

Person.prototype = {
  constructor: Person,
  sayName: function() {
    console.log(this.name)
  }
};

let person1 = new Person("Lucy");
let person2 = new Person("Joe");

person1.sayName(); // "Lucy"
person1.constructor === Person; // true

person2.sayName(); // "Joe"
person2.constructor === Person; // true
```

ES6的类旨在完全涵盖之前规范设计的**基于原型的继承模式**，构造函数 + 原型继承

1. Object方法:Object.keys()、Object.values()、Object.entries()
2. 可以这样来赋值prototype

```
function Person() {
}

Person.prototype = {
  constructor: Person,
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};

//或者使用definePrototype
Object.defineProperty(Person.prototype, "constructor", {
  enumerable: false,
  value: Person
});
```

 image-20240710132524965

重写原型之后创建一个实例的时候指向了一个新的原型，而之前创建的实例仍然会引用最初的原型。

继承**

1. js继承本质上是原型链的继承

```
function SuperType() {  
    this.property = true;  
}  
  
SuperType.prototype.getSuperValue = function() {  
    return this.property;  
};  
  
function SubType() {  
    this.subproperty = false;  
}  
  
// 继承SuperType  
SubType.prototype = new SuperType();  
SubType.prototype.getSubValue = function () {  
    return this.subproperty;  
};  
  
let instance = new SubType();  
console.log(instance.getSuperValue()); // true
```

2. 以对象字面量方式创建原型方法会破坏之前的原型链，相当于重写了原型链。

3. 原型链的问题:

```
function SuperType() {  
    this.colors = ["red", "blue", "green"];  
}  
  
function SubType() {}  
  
// 继承SuperType  
SubType.prototype = new SuperType();  
let instance1 = new SubType();  
instance1.colors.push("black");  
console.log(instance1.colors); // "red, blue, green, black"  
let instance2 = new SubType();  
console.log(instance2.colors); // "red, blue, green, black"
```

4. 为了解决这个原型链的问题：盗用构造函数

```
function SuperType() {  
    this.colors = ["red", "blue", "green"];  
}  
  
function SubType() {  
    //继承SuperType  
    SuperType.call(this);  
}  
  
let instance1 = new SubType();  
instance1.colors.push("black");  
console.log(instance1.colors); // "red, blue, green, black"  
let instance2 = new SubType();  
console.log(instance2.colors); // "red, blue, green"
```

就是在这个SubType对象上运行了SuperType的初始化代码

好处:可以给父级传递参数。可以用于初始化代码

缺点: 方法只能再构造函数中创建, 因此不能重用, 子类还不能访问父类原型上定义的方法。

5. 组合继承(常用的继承)

原型链和盗用构造函数来进行集合起来。思路:使用原型链继承原型上的属性和方法, 盗用构造函数来继承实例属性。

```
function Parent (name) {
  this.name = name;
  this.colors = ["red", "blue", "green"];
}

Parent.prototype.getName = function () {
  console.log(this.name)
}

function Child (name, age) {
  Parent.call(this, name);
  this.age = age;
}

Child.prototype = new Parent();
Child.prototype.constructor = Child;

let child1 = new Child("kevin", "18");

child1.colors.push("black");

console.log(child1.name); // kevin
console.log(child1.age); // 18
console.log(child1.colors); // ["red", "blue", "green", "black"]

let child2 = new Child("daisy", "20");

console.log(child2.name); // daisy
console.log(child2.age); // 20
console.log(child2.colors); // ["red", "blue", "green"]
```

6. 原型式继承:

创建一个对象, 让这个对象原型是另一个对象, 继承了另一个对象的属性和方法

有点:再不预先定义构造函数情况下, 实现对父类继承。

缺点:共性属性, 对引用类型的属性, 一个实例影响, 会影响到所有实例。

就相当于Object.create(),第二个参数和Object.defineProperties()相同

```
function create(o){
  const F = new Function()
  F.prototype = o
  return new F()
}
```

7. 寄生继承

接受一个参数，是一个新对象的基准对象，这个对象original会被传递给object函数，付给clone，给clone添加方法，返回。

创建一个仅用于封装继承过程的函数，该函数在内部以某种形式来做增强对象，最后返回对象。

```
function createObj (o) {  
  let clone = Object.create(o);  
  clone.sayName = function () {  
    console.log("hi");  
  }  
  return clone;  
}
```

缺点：跟借用构造函数模式一样，每次创建对象都会创建一遍方法。

8. 寄生组合继承（最佳模式）

因为组合式继承调用了两次父类构造函数，一次是call的时候，一次是原型的时候

寄生组合继承解决这个问题

```
function inheritPrototype(subType, superType) {  
  let prototype = Object.create(superType.prototype); // 创建对象  
  prototype.constructor = subType; // 增强对象  
  subType.prototype = prototype; // 赋值对象  
}
```

这种方式的高效率体现它只调用了一次 Parent 构造函数，并且因此避免了在 Parent.prototype 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变；因此，还能够正常使用 instanceof 和 isPrototypeOf。

```
function Parent (name) {  
  this.name = name;  
  this.colors = ["red", "blue", "green"];  
}  
  
Parent.prototype.getName = function () {  
  console.log(this.name)  
}  
  
function Child (name, age) {  
  Parent.call(this, name);  
  this.age = age;  
}  
  
// 封装创建新的对象的方法，以传入的原型作为新对象的原型  
function object(o) {  
  function F() {}  
  F.prototype = o;  
  return new F();  
}  
  
//就是封装了一层继承  
function prototype(child, parent) {
```



```

    var prototype = object(parent.prototype);
    prototype.constructor = child;
    child.prototype = prototype;
}

prototype(Child, Parent);

let child1 = new Child("kevin", "18");

child1.colors.push("black");

console.log(child1.name); // kevin
console.log(child1.age); // 18
console.log(child1.colors); // ["red", "blue", "green", "black"]

let child2 = new Child("daisy", "20");

console.log(child2.name); // daisy
console.log(child2.age); // 20
console.log(child2.colors); // ["red", "blue", "green"]

```

类

1. 类就是一个语法糖，表面上是实现了面向对象编程，但实际上还是原型和构造函数概念。

面试

浅拷贝和深拷贝

浅拷贝

1. 赋值语句
2. 拓展运算符
3. 数组可以使用内置的方法 `.slice()`，它的作用和扩展运算符一样，都可以实现浅拷贝
4. `assign`
5. `Array.from`

```

const array = [1, 2, 3];

const copyWithSlice = array.slice();
console.log(copyWithSlice === array); // false

// 改变原数组 array
array[0] = 4;

console.log(array); // [4, 2, 3] 原数组改变了
console.log(copyWithSlice); // [1, 2, 3] 拷贝的数组没有受到影响

//assign

```

```

const array = [1, 2, 3];

const copyWithEquals = array; // 没有拷贝数组
const copyWithAssign = [];
Object.assign(copyWithAssign, array);

array[0] = 4;
console.log(array); // [4, 2, 3] 原数组改变了
console.log(copyWithAssign); // [1, 2, 3] 拷贝的数组没有受到影响

//Array.from
const array = [1, 2, 3];

const copyWithEquals = array; // 没有拷贝数组
const copyWithArrayFrom = Array.from(array);

array[0] = 4;
console.log(array); // [4, 2, 3] 原数组改变了
console.log(copyWithArrayFrom); // [1, 2, 3] 拷贝的数组没有受到影响

```

深拷贝

1. JSON.stringify/parse
2. lodash
3. 自己实现

```

import _ from "lodash"

const nestedArray = [[1], [2], [3]];
const shallowCopyWithLodashClone = _.clone(nestedArray);
const deepCopyWithLodashClone = _.cloneDeep(nestedArray);

```

```

//自己实现的
const deepCopy = (obj, cache = new WeakMap()) => {
  // 如果不是对象或者为 null, 直接返回
  if (typeof obj !== "object" || obj === null) {
    return obj;
  }

  // 如果已经复制过该对象, 则直接返回
  if (cache.has(obj)) {
    return cache.get(obj);
  }

  // 创建一个新对象或数组
  const newObj = Array.isArray(obj) ? [] : {};

  // 将新对象放入 cache 中
  cache.set(obj, newObj);

  // 处理特殊对象的情况

```

```

    if (obj instanceof Date) {
        return new Date(obj.getTime());
    }

    if (obj instanceof RegExp) {
        return new RegExp(obj);
    }

    // 处理循环引用的情况
    Object.keys(obj).forEach((key) => {
        newObj[key] = deepCopy(obj[key], cache);
    });

    return newObj;
};

// 测试 Date 和 regexp 的深拷贝:
const date = new Date();
const regexp = /test/g;

const cloneDate = deepCopy(date); // 深拷贝
const cloneRegExp = deepCopy(regexp); // 深拷贝

console.log(cloneDate === date); // false
console.log(cloneRegExp === regexp); // false

```

设计模式

工厂模式:抽象创建特定的对象的过程

```

function createPerson(name, age, job) {
    let o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function() {
        console.log(this.name);
    };
    return o;
}

let person1 = createPerson("Nicholas", 29, "Software Engineer");
let person2 = createPerson("Greg", 27, "Doctor");

```

缺点是:虽然可以解决创建多个类似的对象的问题,但没有解决创建这个对象是什么类型。可以使用构造函数来解决

构造函数模式:其中构造函数首字母大写,工厂模式的话首字母不是大写。

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function() {
        console.log(this.name);
    };
}
let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");
person1.sayName(); // Nicholas
person2.sayName(); // Greg
```

构造函数的实例对象里面都有一个constructor属性指向Person，这个作为标识对象类型。

缺点:其定义的方法会在每个实例上面创建一遍，就是每一个实例他们都有各自的Function实例。

解决办法:

1. 可以放到外面，但是打乱了全局作用域
2. 使用原型

原型模式:每个函数都有一个prototype属性，这个属性就是一个对象，包含由特定引用类型的实例共享的属性和方法。

缺点:当放入引用类型的时候，一个实例上修改了这个，再另一个实例上也会进行修改。

new操作符内部原理

1. 创建一个新的对象
2. 其中新对象的[[Prototype]]属性被赋值为构造函数的prototype:实例化的对象的proto = 构造函数的prototype
3. 构造函数内部的this被赋值为这个新的对象。
4. 给新对象添加属性
5. 如果构造函数时非空对象，则返回该对象；否则，返回刚创建的对象。

```
function newOperator(Constructor, ...args) {
    let thisValue = Object.create(Constructor.prototype); // 对应上文操作步骤: 1、2
    let result = Constructor.apply(thisValue, args); // 对应上文操作步骤: 3、4
    return typeof result === 'object' && result !== null ? result : thisValue; //
    对应上文操作步骤: 5
}
```

```
function Factory(...args){
    const obj = new Object()
    let constructor = args[0]
    //将构造函数上面的共享的属性方法赋值给新创建的对象
    obj.__proto__ = constructor.prototype
    //将构造函数的实例上的方法和属性赋值给对象
    const ret = constructor.apply(obj, ...args)
    return typeof ret === 'object'?ret:obj
}
```

原型

记住:当创建一个函数的时候,就会有 `prototype` 属性 (指向原型对象)。然后这个原型对象有一个 `constructor` 属性,指回与之关联的构造函数。

```
function Person() {}
console.log(Person.prototype)
// {
//   constructor: f Person(),
//   __proto__: Object
// }
console.log(Person.prototype.constructor === Person) // true

/* *
 * 正常的原型链都会终止于Object的原型对象
 * Object原型的原型是null
 */
console.log(Person.prototype.__proto__ === Object.prototype); // true
console.log(Person.prototype.__proto__.constructor === Object); // true
console.log(Person.prototype.__proto__.__proto__ === null); // true
console.log(Person.prototype.__proto__);
// {
//   constructor: f Object(),
//   toString: ...
//   hasOwnProperty: ...
//   isPrototypeOf: ...
//   ...
// }
let person1 = new Person(),
    person2 = new Person();
/* *
 * 构造函数、原型对象和实例
 * 是3 个完全不同的对象:
 */
console.log(person1 !== Person); // true
console.log(person1 !== Person.prototype); // true
console.log(Person.prototype !== Person); // true
/* *
 * 实例通过__proto__链接到原型对象,
 * 它实际上指向隐藏特性[[Prototype]]
 *
 * 构造函数通过prototype属性链接到原型对象
 *
 * 实例与构造函数没有直接联系,与原型对象有直接联系
 */
console.log(person1.__proto__ === Person.prototype); // true
console.log(person1.__proto__.constructor === Person); // true
/* *
 * 同一个构造函数创建的两个实例
 * 共享同一个原型对象:
 */
console.log(person1.__proto__ === person2.__proto__); // true
/* *
 * instanceof检查实例的原型链中
 * 是否包含指定构造函数的原型:
```

```

*/
console.log(person1 instanceof Person);           // true
console.log(person1 instanceof Object);           // true
console.log(Person.prototype instanceof Object);  // true

```

判断是否指向内部的[[Prototype]]:isPrototypeOf,Object.getPrototypeOf

```

console.log(Person.prototype.isPrototypeOf(person1)); // true
console.log(Person.prototype.isPrototypeOf(person2)); // true

```

```

console.log(Object.getPrototypeOf(person1) == Person.prototype); // true
console.log(Object.getPrototypeOf(person1).name);                  //
"Nicholas"

```

设置值，就可以重写一个对象的原型继承关系：Object.setPrototypeOf(目标，新加的)

原型链:就是在实例身上寻找相关属性或者方法，但是实例本身并没有这些，就会向它的原型上去寻找。

只要对象实例上添加一个属性，这个属性就会遮蔽对象上的同名属性。

使用delete操作符可以完全删除实例上的属性。

hasOwnProperty()方法用于确定某个属性是实例上的还是原型对象的。

判断是否在原型上

```

function hasPrototypeProperty(object,name){
    return !object.hasOwnProperty(name)&&name in object
}

```

性能

在原型链上查找属性比较耗时，对性能有副作用，这在性能要求苛刻的情况下很重要。另外，试图访问不存在的属性时会遍历整个原型链。

遍历对象的属性时，原型链上的每个可枚举属性都会被枚举出来。要检查对象是否具有自己定义的属性，而不是其原型链上的某个属性，则必须使用所有对象从 Object.prototype 继承的 hasOwnProperty 方法。

总结

Object() 函数有一个 prototype 属性，指向 Object.prototype 对象。Object.prototype 对象有对象所有的属性和方法，比如 toString()、valueOf()。

Object.prototype 对象有 constructor 属性，指向 Object 函数。

每个函数都有一个 prototype 对象，这个原型对象的 **proto** 指向 Object.prototype，通过 [[prototype]] 或者 **proto** 属性查找属性和方法。

Object.getPrototypeOf() 方法返回给定对象的原型对象，建议使用 Object.getPrototypeOf() 替代 **proto**。通过 hasOwnProperty 检查对象是否具有自己定义的属性。

类

1. 类就是原型和构造函数的语法糖
2. 类的继承就是原型的语法糖

第九章代理和反射

代理基础

1. 代理和反射为开发者提供了一个拦截并向基本操作嵌入额外行为的能力。
2. 严格相等可以区分代理和目标对象。===, Proxy.prototype 为undefined。所以不能用instanceof
3. 定义捕获器, 代理可以在这些操作传播到目标对象之前, 先调用捕获器函数, 拦截并修改相应行为
4. 捕获器参数: 目标对象, 捕获的属性, 代理对象, 这样就可以重建捕获方法的原始行为, 也可以使用Reflect来反射。

```
const target = {
  foo: 'bar'
};
const handler = {
  get() {
    return Reflect.get(...arguments);
  }
};
//或者
const handler = {
  get: Reflect.get
};
//或者
const proxy = new Proxy(target, Reflect);

const proxy = new Proxy(target, handler);
console.log(proxy.foo);    // bar
console.log(target.foo);  // bar
```

5. 撤销代理, 中断目标元素和代理对象之间的联系

```
const target = {
  foo: 'bar'
};
const handler = {
  get() {
    return 'intercepted';
  }
};
const { proxy, revoke } = Proxy.revocable(target, handler);
console.log(proxy.foo);    // intercepted
console.log(target.foo);  // bar
revoke();
console.log(proxy.foo);    // TypeError
```

6. 反射api, 大多数反射api在Object上都有方法。反射方法返回称作"状态标记的"布尔值, 例如Object.defineProperty和Reflect.defineProperty, 前者出现错误会报错, 后者会返回false。

7. 使用一等函数代替操作符

```
Reflect.get()->.  
.set()->=  
.has()->in  
.deleteProperty->delete  
.construct->new  
//第二个参数是数组的形式，相当于传递给构造函数的参数：  
function func1(a, b, c) {  
  this.sum = a + b + c;  
}  
const args = [1, 2, 3];  
// 下面两者相等  
const object1 = new func1(...args);  
const object2 = Reflect.construct(func1, args);
```

8. 代理另一个代理，一样的操作。

9. Proxy的不足:就是this问题，正常情况下就是调用方法那个对象作为this值，但是遇到了目标对象依赖对象标识，就会出现错误。

```
const wm = new WeakMap();  
class User {  
  constructor(userId) {  
    wm.set(this, userId);  
  }  
  set id(userId) {  
    wm.set(this, userId);  
  }  
  get id() {  
    return wm.get(this);  
  }  
}  
//将对象作为WeakMap键  
const user = new User(123);  
console.log(user.id); // 123  
const userInstanceProxy = new Proxy(user, {});  
console.log(userInstanceProxy.id); // undefined  
//因为一开始使用的是User类作为WeakMap键，代理就会尝试从自身找这个实例，解决该问题将代理user类实例变为代理User类本身  
const UserClassProxy = new Proxy(User, {});  
const proxyUser = new UserClassProxy(456);  
console.log(proxyUser.id);
```

10. 有一些ES内置类型可能会依赖代理不可控制的机制，如果代理拦截后转发给目标对象会报出错误
TypeError

代理

1. ownKeys()会在Objcet.keys()及其类似方法中调用

代理模式

常见的使用场景

1. 跟踪属性访问，可以操作get,set,has等操作，观察属性什么时候被访问，设置，查询
2. 隐藏属性

```
const hiddenProperties = ['foo', 'bar'];
const targetObject = {
  foo: 1,
  bar: 2,
  baz: 3
};
const proxy = new Proxy(targetObject, {
  get(target, property) {
    if (hiddenProperties.includes(property)) {
      return undefined;
    } else {
      return Reflect.get(...arguments);
    }
  },
  has(target, property) {
    if (hiddenProperties.includes(property)) {
      return false;
    } else {
      return Reflect.has(...arguments);
    }
  }
});
// get()
console.log(proxy.foo);    // undefined
console.log(proxy.bar);    // undefined
console.log(proxy.baz);    // 3
// has()
console.log('foo' in proxy); // false
console.log('bar' in proxy); // false
console.log('baz' in proxy); // true
```

3. 属性验证

```
const target = {
  onlyNumbersGoHere: 0
};
const proxy = new Proxy(target, {
  set(target, property, value) {
    if (typeof value !== 'number') {
      return false;
    } else {
      return Reflect.set(...arguments);
    }
  }
});
proxy.onlyNumbersGoHere = 1;
console.log(proxy.onlyNumbersGoHere); // 1
```

```
proxy.onlyNumbersGoHere = '2';  
console.log(proxy.onlyNumbersGoHere); // 1
```

4. 函数和构造函数参数验证

```
function median(...nums) {  
    return nums.sort()[Math.floor(nums.length / 2)];  
}  
const proxy = new Proxy(median, {  
    apply(target, thisArg, argumentsList) {  
        for (const arg of argumentsList) {  
            if (typeof arg !== 'number') {  
                throw 'Non-number argument provided';  
            }  
        }  
        return Reflect.apply(...arguments);  
    }  
});  
console.log(proxy(4, 7, 1)); // 4  
console.log(proxy(4, '7', 1));  
// Error: Non-number argument provided
```

必须初始化

```
class User {  
    constructor(id) {  
        this.id_ = id;  
    }  
}  
const proxy = new Proxy(User, {  
    construct(target, argumentsList, newTarget) {  
        if (argumentsList[0] === undefined) {  
            throw 'User cannot be instantiated without id';  
        } else {  
            return Reflect.construct(...arguments);  
        }  
    }  
});  
new proxy(1);  
new proxy();  
// Error: User cannot be instantiated without id
```

5. 数据绑定与可观察对象

可以将被代理的类绑定到一个全局实例集合，让所有创建的实例都被添加到这个集合中：

```
const userList = [];  
class User {  
    constructor(name) {  
        this.name_ = name;  
    }  
}  
const proxy = new Proxy(User, {
```

```

    construct() {
        const newUser = Reflect.construct(...arguments);
        userList.push(newUser);
        return newUser;
    }
});
new proxy('John');
new proxy('Jacob');
new proxy('Jingleheimerschmidt');
console.log(userList); // [User {}, User {}, User{}]

```

总结:

代理可以定义包含**捕获器**的处理程序对象，而这些捕获器可以拦截绝大部分JavaScript的基本操作和方法。在这个捕获器处理程序中，可以修改任何基本操作的行为，当然前提是遵从捕获器不变式。

反射API，则封装了一整套与捕获器拦截的操作相对应的方法。可以把反射API看作一套基本操作，这些操作是绝大部分JavaScript对象API的基础。

面试

Object.defineProperty和Proxy区别

`Object.defineProperty()` 可以定义一个对象的属性或修改对象上已存在的属性，并返回这个对象。语法如下：

```

let o = {};
let value = "value";

Object.defineProperty(o, "b", {
  get() {
    // 获取 o.b 的值
    console.log(`get ${value}`);
    return value;
  },
  set(newValue) {
    // 设置 o.b 的值
    console.log(`set ${newValue}`);
    value = newValue;
  },
  enumerable: true,
  configurable: true
});

// 获取对象 o 的属性 b 的值
console.log(o.b);
// 依次输出：
// "get value"
// "value";

// 设置对象 o 的属性 b 的值为 value1
o.b = "value1"; // "set value1"

// 重新获取对象 o 的属性 b 的值
console.log(o.b);

```

```
// 依次输出：
// "get value1"
// "value1"

delete o.b; // 删除属性，未触发 get、set 操作

console.log(o.b); // undefined
```

由上可知，Object.defineProperty 在劫持对象和数组时的缺陷：

- 无法检测到对象属性的添加或删除
- 监听对象的多个属性，需要遍历该对象
- 无法检测数组元素的变化，需要进行数组方法的重写
- 无法检测数组的长度的修改

Proxy 直接代理了 target 整个对象，并且返回了一个新的对象；能监听到属性的增加、删除操作

Proxy 中虽然完成了对目标对象的代理，但是即使 handler 为空对象，它代理的对象中的 this 指向的是代理对象，而不是目标对象。

```
let target = {
  m() {
    // 检查 this 的指向是不是 proxyObj
    console.log(this === proxyObj)
  }
}
let handler = {}
let proxyObj = new Proxy(target, handler)

proxyObj.m() // 输出: true
target.m() // 输出: false
```

如果想要获取目标对象的 this，可以使用 Reflect

Reflect作用

"Reflect是一个内置的JavaScript对象，它提供了一组用于访问和操作对象的方法。

就是挂在Reflect上的方法都可以在原始方法中找到

最大的区别

1. 返回值，Reflect的一些方法会有着返回值，能让你知道这个操作有没有成功
2. Reflect方法还有个好处，不会因为报错而中断正常的代码逻辑执行。
3. 还有一个receiver参数可以指定this值。通常情况下，receiver参数是无需使用的，但是如果发生了继承，为了明确调用主体，receiver参数就需要出马了。

```
//例如
let cat = {
  _name: '中华田园猫',
  get name () {
    return this._name
  }
}
```

```

    }
}

let baiMao = new Proxy(cat, {
  get (target, prop) {
    return target[prop];
  }
})

let xiaoBai = {
  __proto__: baiMao,
  _name: "小白"
}

console.log(xiaoBai.name); // "中华田园猫"

let cat = {
  _name: '中华田园猫',
  get name () {
    return this._name
  }
}

let baiMao = new Proxy(cat, {
  get (target, prop, receiver) {
    return Reflect.get(target, prop, receiver);
  }
})

let xiaoBai = {
  __proto__: baiMao,
  _name: "小白"
}

console.log(xiaoBai.name); // "小白"

```

实现一个简单的观察者模式

看手写js

第十章函数

理解参数

1. 箭头函数没有arguments，普通函数可以通过arguments来访问参数，这是一个类数组，在非严格模式下当修改arguments的值，会反映到命名参数。

没有重载

1. 因为没有方法签名，js的函数参数是根据数组类型显示的。
2. 可以通过检查参数的类型和数量来模拟实现函数重载。

默认参数

1. 使用默认参数，arguments对象的值不反映参数的默认值，只反映传给函数的参数。
2. 作用域和暂时性死局

参数初始化是按照顺序初始化的，所以定义默认值的参数不可以引用没有先定义的参数。

作用域，不能引用函数里面的参数

拓展运算符

函数声明和函数表达式

1. 函数声明提升,会将函数提升到最前面,但是函数表达式没有

面试

箭头函数

```
var name = "window";

var person1 = {
  name: "person1",
  foo1: function () {
    console.log(this.name);
  },
  foo2: () => console.log(this.name),
  foo3: function () {
    return function () {
      console.log(this.name);
    };
  },
  foo4: function () {
    return () => {
      console.log(this.name);
    };
  }
};

var person2 = { name: "person2" };

person1.foo1(); // person1
person1.foo1.call(person2); // person2

person1.foo2(); // window

// foo2 返回一个箭头函数，箭头函数使用 call 绑定失效，第一个参数被忽略
person1.foo2.call(person2); // window

person1.foo3()(); // window
person1.foo3.call(person2)(); // window
person1.foo3().call(person2); // person2

person1.foo4()(); // person1
person1.foo4.call(person2)(); // person2
```

```
// 箭头函数的 this 在定义的时候就被指定了，之后改变不了
person1.foo4().call(person2); // person1
```

call、apply、bind书写

看手写js篇

this的指向

函数的调用方式决定了 `this` 的值（运行时绑定）。`this` 不能在执行期间被赋值，并且在每次函数被调用时 `this` 的值也可能会不同。

1. 全局上下文，`this`指向window

2. 函数上下文

- 作为对象的方法调用，`this` 指向该对象
 - 作为普通函数调用：严格模式下，指向全局对象，浏览器中就是 window；非严格模式下，为 undefined
 - `call`、`apply`、`bind` 调用，`this` 指向绑定的对象
 - 作为构造函数调用，如使用 `new`，`this` 指向新的对象
3. 类：`this` 指向类。在类的构造函数中，`this` 是一个常规对象。类中所有非静态的方法都会被添加到 `this` 的原型中
4. 箭头函数：`this` 是在定义时确定的，而不是调用时。它继承自外部（定义时）上下文中的 `this`，并且不会被 `call`、`apply` 或 `bind` 改变。箭头函数在定义时从它的词法作用域中继承 `this`。

```
const obj = {
  name: 'Alice',
  regularFunction: function() {
    console.log(this.name);
  },
  arrowFunction: () => {
    console.log(this.name);
  }
};
```

`obj.regularFunction();` // 输出: Alice

`obj.arrowFunction();` // 输出: undefined

在这个例子中，箭头函数定义在全局作用域中（因为它是直接作为对象字面量的一部分定义的，而不是在一个方法中定义的）。

因此，箭头函数的 `this` 指向全局对象。在全局对象中没有 `name` 属性，所以输出 `undefined`。

//在方法中定义箭头函数

```
const obj = {
  name: 'Alice',
  regularFunction: function() {
    const arrowFunction = () => {
      console.log(this.name);
    };
    arrowFunction();
  }
}
```

```
};
```

```
obj.regularFunction(); // 输出: Alice
```

```
obj.regularFunction():
```

`regularFunction` 是一个普通函数。

当 `obj.regularFunction()` 被调用时, `this` 指向 `obj` 对象。

在 `regularFunction` 中定义的 `arrowFunction` 是一个箭头函数, 它继承了 `regularFunction` 的 `this`。

因此, 当 `arrowFunction` 被调用时, `this` 仍然指向 `obj`, 输出 `Alice`。

词法作用域 (Lexical Scope), 也称为静态作用域 (Static Scope), 是在代码编写时确定的作用域。它描述了变量和函数在不同作用域 (例如, 全局作用域、函数作用域、块作用域) 之间的可见性。

5. 原型链上的this:如果该方法存在于一个对象的原型链上, 那么 `this` 指向的是调用这个方法的对象。

```
var o = {  
  f: function() {  
    return this.a + this.b;  
  }  
};  
var p = Object.create(o);  
p.a = 1;  
p.b = 4;  
  
console.log(p.f()); // 5  
。
```

6. 作为一个DOM事件处理函数

当函数被用作事件处理函数时, 它的 `this` 指向触发事件的元素。

```
function bluify(e) {  
  console.log(this === e.currentTarget); // true  
  this.style.backgroundColor = 'blue'  
}  
  
// 获取 id 为 test 的 button  
let testBtn = document.getElementById('test');  
  
// 将 bluify 作为元素的点击监听函数, 当元素被点击时, 就会变成蓝色  
testBtn.addEventListener('click', bluify, false);
```

闭包

内部的函数和周边环境进行绑定, 就是内部定义一个函数可以让外部的作用域来访问内部的变量。

闭包让你可以在一个内层函数中访问到其外层函数的作用域

闭包的应用场景

创建私有变量

```
let Counter = (function() {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
})();

console.log(Counter.value()); // 0
Counter.increment();
Counter.increment();
console.log(Counter.value()); // 2
Counter.decrement();
console.log(Counter.value()); // 1
```

缓存变量值

```
function createInc(startValue) {
  let index = -1;
  return (step) => {
    startValue += step;
    index++;
    return [index, startValue];
  };
}

const inc = createInc(5);
console.log(inc(2)); // [0, 7]
console.log(inc(2)); // [1, 9]
console.log(inc(2)); // [2, 11]
```

注意:如果不是某些特定任务需要使用闭包,在其它函数中创建函数是不明智的,因为闭包在处理速度和内存消耗方面对脚本性能具有负面影响

类数组转化为数组

1. Array.from()
2. slice(): `Array.prototype.slice.call(arguments)`, `[].slice.call(arguments)`;
3. 拓展运算符

第十一章期约与异步函数

异步编程

1. 为了优化因计算量大而时间长的操作。
2. 为了后续代码使用x, 异步执行函数更新x后通知其他代码。
3. 变化:回调函数(通过参数来获取回调后的值, 如果异步的值有依赖另一个异步的返回值, 那么会变得更加复杂了, 回调地狱)

期约(Promise)

1. 三个状态:pending,resolve,rejected,一旦状态改变了之后, 就不能再进行变化了。
2. Promise.resolve(),Promise.rejected()传入参数之后会返回一个期约。
3. 不能通过try/catch (捕获的是同步代码) 来捕获, 只能通过拒绝处理程序捕获。

Promise是同步初始化对象, 但是异步来进行执行的, 拒绝Promise的错误没有抛到执行同步代码的线程中, 而是通过浏览器异步消息队列处理的。

4. Promise.prototype.then,Promise.prototype.catch,之后都返回一个新的期约
5. then会把onResolved处理程序推到消息队列中, 这个处理程序不会在同步任务执行前执行。
6. 当期约进行状态落定的时候, 与该状态相关的处理程序会进行排期, 不是立即执行的, 添加在这个处理程序之后的同步代码一定会在处理程序之前执行。

先解决期约或者后解决期约都是这种效果, 这个体现在onResolved, onRejected, catch, finally上面。

7. 在状态改变之前还是可以使用try/catch在执行函数中捕捉错误
8. Promise.all:全部解决之后才结束, 当有一个失败, 那么就返回那个失败的, 都成功, 返回最后的值通过数组形式按照顺序
9. Promise.race: 最先解决状态的值
10. 串行期约:

```
function addTwo(x) {return x + 2; }
function addThree(x) {return x + 3; }
function addFive(x) {return x + 5; }
function addTen(x) {
  return Promise.resolve(x)
    .then(addTwo)
    .then(addThree)
    .then(addFive);
}
addTen(8).then(console.log); // 18
```

或者

```
function addTwo(x) {return x + 2; }
function addThree(x) {return x + 3; }
function addFive(x) {return x + 5; }
function compose(...fns){
  return(x)=>fns.reduce((promise, fn)=>promise.then(fn), Promise.resolve(x))
}
let addTen = compose(addTwo, addThree, addFive);
addTen(8).then(console.log); // 18
function addTwo(x) {return x + 2; }
function addThree(x) {return x + 3; }
function addFive(x) {return x + 5; }
function compose(...fns){
  return(x)=>fns.reduce((promise, fn)=>promise.then(fn), Promise.resolve(x))
}
let addTen = compose(addTwo, addThree, addFive);
addTen(8).then(console.log); // 18
```

面试

Promise缺点

无法取消：一旦创建，无法中途取消 Promise，需要额外的工作来实现取消逻辑。

只能处理单一值：每个 Promise 实例只能处理一次结果，无法像 RxJS 那样处理多个值的流。

嵌套问题：虽然 Promise 可以通过链式调用解决回调地狱问题，但过深的链式调用仍然可能难以管理和调试。

async和await本质

async是声明一个异步函数的，返回的值包裹了Promise对象，通过return来确定返回的值

await是来暂停执行这个操作的，会等待Promise的状态

async

返回 Promise 对象：

- `async` 函数总是返回一个 Promise 对象。如果在函数中显式地返回一个值，这个值会被自动包装成一个已解决的 Promise。

自动转换：

- 在调用 `async` 函数时，函数体内部的代码会立即执行，但 `async` 函数本身会立即返回一个 Promise 对象，并在内部代码执行完毕后，根据 `return` 或 `throw` 的结果来决定 Promise 的最终状态。

await

暂停执行：

- 当 `await` 关键字用于等待一个 Promise 对象时，它会暂停当前 `async` 函数的执行，直到 Promise 对象的状态变为 `fulfilled` 或 `rejected`。

非阻塞：

- 使用 `await` 不会阻塞其他 JavaScript 代码的执行，因为 `async` 函数本身也是异步的，它会在 Promise 进入队列时暂停执行，允许其他代码继续执行。

ES6+有哪些

1. let, const
2. Promise
3. 箭头函数
4. 默认参数
5. async、await
6. 模块化
7. 拓展运算符
8. Map、Set
9. 类和继承
10. 模板字符串

性能优化

CSS:

1. 使用精灵图:

设置背景图像为精灵图。

使用 `background-position` 属性来显示特定的小图标。

设置元素的宽高为小图标的宽高。

2. 减少层次嵌套
3. css最好放在头部

JS:

1. 最好放在底部或者使用async和defer
2. 减少DOM操作，将多次DOM操作合并为一次，减少重绘和重排。

打包:

1. 代码分割
2. 压缩
3. 使用Webpack等工具按需加载代码，减少初始包大小。

CDN:

1. 将静态资源放到CDN上

浏览器和DOM相关API（12-22章）

第十二章BOM


window

1. 窗口关系:window.top,window.parent,window.self
2. 窗口大小:innerWidth,innerHeight,outerWidth,outerHeight返回浏览器窗口自身的大小。
document.documentElement.clientWidth和document.documentElement.clientHeight返回页面视口的宽度和高度。
3. 视口缩放:window.resizeTo(),window.resizeBy()
4. 视口位置:window.pageXOffset,window.scrollX,window.pageYOffset,window.scrollY。滚动页面:scroll,scrollTo,scrollBy还可以设置行为

```
window.scrollTo({
  left:100,
  top:100,
  behavior:'auto'
})
还有smooth
```

5. 打开导航到指定URL，也可以打开新浏览器窗口。四个参数:url、目标窗口、特性字符串和新窗口在浏览器历史中是否替代当前加载页面的布尔值。第二个属性:self,blank,top,parent
6. 定时器:setInterval在向任务队列中添加任务的时候，会检查任务队列中是否有未完成的setInterval任务队列，有就不会添加，会导致任务到了下一次添加任务的时候，上一个任务还未执行完成，会导致任务丢失。setTimeout会在超时后将任务添加到任务队列中，不会判断是否还有未完成的。

location对象

1. 888adfdf01c2b521ead72b75efa2448
2. 查询字符串可以使用URLSearchParams提供的一些API，有has(),get(),set(),delete()方法

3.

```
let getQueryStringArgs = function() {
  // 取得没有开头问号的查询字符串
  let qs = (location.search.length > 0 ? location.search.substring(1) :
    ""),
    // 保存数据的对象
    args = {};
  // 把每个参数添加到args对象
  for (let item of qs.split("&").map(kv => kv.split("="))) {
    let name = decodeURIComponent(item[0]),
        value = decodeURIComponent(item[1]);
    if (name.length) {
      args[name] = value;
    }
  }
  return args;
}
```

4. 修改浏览器地址:location.href,只要修改了location的一个属性就会导致页面重新修改。
location.replace(),location.reload()

navigator对象

1. 检查插件

```
let hasPlugin = (name)=>{
  name = name.toLowerCase()
  for(let plugin of window.navigator.plugins){
    if(plugin.name.toLowerCase().indexOf(name) > -1){
      return true
    }
  }
  return false
}
```

2. 注册处理程序registerProtocolHandler(),可以把一个网站注册为处理某种特定类型信息应用程序。

Navigator 的方法 registerProtocolHandler() 让 web 站点为自身注册用于打开或处理特定 URL 方案（又名协议）的能力。

举个例子，此 API 允许 web 邮件站点打开 mailto: URL，或让 VoIP 站点打开 tel: URL。

语法

```
navigator.registerProtocolHandler(scheme, url, title);
```

screen

history对象

1. history.go(),history.back(),history.forward(),history.length
2. 历史状态管理。

十三章 客户端检验(使用库来进行检查即可)

十四章 DOM

为了操作HTML，浏览器实现了DOM这个接口

具体的一些方法可以看我之前写的js笔记

MoutationObserver

1. DOM修改后进行异步回调
- 2.

十五章 DOM拓展

看js笔记这部分

Selectors API

querySelector,querySelectorAll,matches

元素遍历

getElementClassName,

classList有add、contains、remove、toggle

十六章DOM2

十七章 事件

其中的一些api可以看我刚学js的笔记

内存与性能

事件委托:本质上使用了事件冒泡，只使用一个事件处理程序管理一种类型的事件

- 减少了整个页面所使用的内存，提高了整体性能
- 节约了花在设置页面事件处理程序上的事件。

最好限制一个页面中事件处理程序的数量。

十八章 动画和Canvas图形

使用requestAnimationFrame

canvas

这个可以查看以前的笔记，ppt

十九章表单脚本

二十章 JavaScript API

Encoding API

File API 与Blob API

媒体元素

1. audio和video
2. 有一些属性
3. 事件
4. 检测编解码器

```
if(audio.canPlayType("audio/mpeg")){  
  
}  
//会返回"probably"、"maybe"或者""  
//也可以检测视频格式
```

原生拖放

1. 拖放可以跨窗格、跨浏览器有时候甚至可以跨应用程序拖放元素
2. 事件顺序:dragstart、drag、dragend
3. dragenter、dragover、dragleave/drop
4. 将元素转化为目标放置的元素

```
let droptarget = document.getElementById("droptarget");  
droptarget.addEventListener("dragover", (event) => {  
    event.preventDefault();  
});  
droptarget.addEventListener("dragenter", (event) => {  
    event.preventDefault();  
});
```

5. dataTransfer对象

可以像防止目标传递数据: getData()、setData()

6. dataTransfer对象可以用于实现简单的数据传输, 还可以用于对被拖动的元素和放置的目标执行什么操作。dropEffect和effectAllowed
7. dropEffect: 可以告诉浏览器允许哪种放置行为:none、copy:被拖放的元素应该复制到目标元素、move: 被拖放的元素应该移动到放置目标、link: 放置目标会导航到被拖放元素。这个需要在ondropenter事件处理程序设置
8. effectAllowed需要设置, 这样dropEffect才能使用生效。可以在ondropstart事件处理程序中设置属性。effectAllowed表示对拖放元素是否允许dropEffect
9. 设置可拖放能力:在标签里面设置draggable属性

Notifications API

1. 通知权限:Notification有一个requestPermission方法, 会返回一个期约, dranted表示成功, 失败表示denied
2. 显示和隐藏:

```
new Notification("Title text!")
```

close()方法取消

3. 方法:onshow、onclick、onclose、onerror

Page Visibility API

1. 开发中提供页面对用户是否可见的信息
2. document.visibilityState值表示hidden、visible、prerender
3. visibilitychange事件

Stream API

1. 可读流和可写流
2. 轮换流

计时API

1. High Resolution Time API: 时间精度API, 解决Date.now()的精度问题。

window.performance.now()代替Date.now()。performance.now()最常见的用例是监控一段代码的执行时间。现实生活中的用例可以包括视频、音频、游戏和其他媒体的基准测试和监控性能。

Performance.now()和Date.now()之间最大的区别是, Date.now()返回一个与Unix时间有关的时间戳(从00:00:00 UTC, 1970年1月1日开始的时间)。这就是我们可能面临的一个问题。JavaScript如何知道从这个日期开始已经过了多少时间? 它从系统时钟中获取。但是, 由于系统时钟存在于我们的机器中, 它可以手动或通过程序进行调整, 因此, 时间精度无法保证。

在这种情况下, performance.now()就比较可靠。它既不依赖于历史上的某个特定时间点, 也不依赖于我们的系统, 因为它返回的是自文档生命期开始以来所经过的毫秒数量。

2. performance timeline api: Performance API 提供了重要的内置指标, 并能够将你自己的测量结果添加到浏览器的性能时间线 (performance timeline) 中。

3.

```
performance.mark("squirrel");
//自定义时间
// 创建一些标记。
performance.mark("squirrel");
performance.mark("squirrel");
performance.mark("monkey");
performance.mark("monkey");
performance.mark("dog");
performance.mark("dog");

// 获取所有的 PerformanceMark 条目。
const allEntries = performance.getEntriesByType("mark");
console.log(allEntries.length);
// 6

// 获取所有的 "monkey" PerformanceMark 条目。
const monkeyEntries = performance.getEntriesByName("monkey");
console.log(monkeyEntries.length);
// 2

// 删除所有标记。
performance.clearMarks();
```

4.

```
performance.measure(name, startMark, endMark);
```

```

// 以一个标志开始。
performance.mark("mySetTimeout-start");

// 等待一些时间。
setTimeout(function () {
    // 标志时间的结束。
    performance.mark("mySetTimeout-end");

    // 测量两个不同的标志。
    performance.measure("mySetTimeout", "mySetTimeout-start", "mySetTimeout-end");

    // 获取所有的测量输出。
    // 在这个例子中只有一个。
    var measures = performance.getEntriesByName("mySetTimeout");
    var measure = measures[0];
    console.log("SetTimeout milliseconds:", measure.duration);

    // 清除存储的标志位
    performance.clearMarks();
    performance.clearMeasures();
}, 1000);

```

5. `PerformanceNavigationTiming` 提供了用于存储和检索有关浏览器文档事件的指标的属性和方法。例如，此接口可用于**确定加载或卸载文档需要多少时间**。

```

const [performanceNavigationTimingEntry] =
performance.getEntriesByType('navigation');
console.log(performanceNavigationTimingEntry);
// PerformanceNavigationTiming {
//   connectEnd: 2.259999979287386
//   connectStart: 2.259999979287386
//   decodedBodySize: 122314
//   domComplete: 631.9899999652989
//   domContentLoadedEventEnd: 300.92499998863786
//   domContentLoadedEventStart: 298.8950000144541
//   domInteractive: 298.88499999651685
//   domainLookupEnd: 2.259999979287386
//   domainLookupStart: 2.259999979287386
//   duration: 632.819999998901
//   encodedBodySize: 21107
//   entryType: "navigation"
//   fetchStart: 2.259999979287386
//   initiatorType: "navigation"
//   loadEventEnd: 632.819999998901
//   loadEventStart: 632.0149999810383
//   name: " https://foo.com "
//   nextHopProtocol: "h2"
//   redirectCount: 0
//   redirectEnd: 0
//   redirectStart: 0
//   requestStart: 7.7099999762140214
//   responseEnd: 130.50999998813495
//   responseStart: 127.16999999247491
//   secureConnectionStart: 0

```

```
//   serverTiming: []
//   startTime: 0
//   transferSize: 21806
//   type: "navigate"
//   unloadEventEnd: 132.73999997181818
//   unloadEventStart: 132.41999997990206
//   workerStart: 0
// }
console.log(performanceNavigationTimingEntry.loadEventEnd -
            performanceNavigationTimingEntry.loadEventStart);
// 0.805000017862767
```

6. PerformanceResourceTiming 接口可以检索和分析有关加载应用程序资源的详细网络计时数据。应用程序可以使用 timing 指标来确定获取特定资源所需的时间长度，例如 [XMLHttpRequest](#)，[](#)，[<image>](#) 或 [script](#)。

```
const performanceResourceTimingEntry = performance.getEntriesByType('resource')
[0];
console.log(performanceResourceTimingEntry);
// PerformanceResourceTiming {
//   connectEnd: 138.11499997973442
//   connectStart: 138.11499997973442
//   decodedBodySize: 33808
//   domainLookupEnd: 138.11499997973442
//   domainLookupStart: 138.11499997973442
//   duration: 0
//   encodedBodySize: 33808
//   entryType: "resource"
//   fetchStart: 138.11499997973442
//   initiatorType: "link"
//   name: "https://static.foo.com/bar.png",
//   nextHopProtocol: "h2"
//   redirectEnd: 0
//   redirectStart: 0
//   requestStart: 138.11499997973442
//   responseEnd: 138.11499997973442
//   responseStart: 138.11499997973442
//   secureConnectionStart: 0
//   serverTiming: []
//   startTime: 138.11499997973442
//   transferSize: 0
//   workerStart: 0
// }
console.log(performanceResourceTimingEntry.responseEnd -
            performanceResourceTimingEntry.requestStart);
// 493.9600000507198
```

web组件

1. 提前做好标记，将标记解析为DOM，跳过渲染，这是HTML模板的核心，

