初中级前端 JavaScript 自测清单-2

前言

在《初中级前端 JavaScript 自测清单 - 1》部分中，和大家简单过了一遍 JavaScript 基础知识，没看过的朋友可以回顾一下😄

本部分内容，以 **JavaScript 对象**为主，大致包括以下内容：

一、对象

JavaScript 有八种数据额类型，有七种原始类型，它们值只包含一种类型（字符串，数字或其他），而对象是用来**保存键值对和更复杂实体**。 我们可以通过使用带有可选**属性列表**的花括号 `**{...}**` 来创建对象，一个属性就是一个键值对 `{"key" : "value"}` ，其中键（ `key` ）是一个字符串（或称属性名），值（ `value` ）可以是任何类型。

1. 创建对象

我们可以使用 2 种方式来创建一个新对象：

```
// 1. 通过“构造函数”创建
let user = new Object();

// 2. 通过“字面量”创建
let user = {};
```

2. 对象文本和属性

创建对象时，可以初始化对象的一些属性：

```
let user = {
    name : 'leo',
  age  : 18
}
```

然后可以对该对象进行属性对**增删改查**操作：

```
// 增加属性
user.addr = "China";
// user => {name: "leo", age: 18, addr: "China"}

// 删除属性
delete user.addr
// user => {name: "leo", age: 18}

// 修改属性
user.age  = 20;
// user => {name: "leo", age: 20}

// 查找属性
user.age;
// 20
```

3. 方括号的使用

当然对象的键（ `key` ）也可以是多词属性，但必须加引号，使用的时候，必须使用方括号（ `[]` ）读取：

```
let user = {
    name : 'leo',
  "my interest" : ["coding", "football", "cycling"]
}
user["my interest"]; // ["coding", "football", "cycling"]
delete user["my interest"];
```

我们也可以在方括号中使用变量，来获取属性值：

```
let key = "name";
let user = {
    name : "leo",
  age  : 18
}
// ok
user[key]; // "leo"
user[key] = "pingan";

// error
user.key; // undefined
```

4. 计算属性

创建对象时，可以在对象字面量中使用方括号，即 **计算属性**：

```
let key = "name";
let inputKey = prompt("请输入key", "age");
let user = {
    [key] : "leo",
  [inputKey] : 18
}
// 当用户在 prompt 上输入 "age" 时，user 变成下面样子：
// {name: "leo", age: 18}
```

当然，计算属性也可以是表达式：

```
let key = "name";
let user = {
    ["my_" + key] : "leo"
```

```
}
user["my_" + key]; // "leo"
```

5.属性名简写

实际开发中，可以将相同的属性名和属性值简写成更短的语法：

```
// 原本书写方式
let getUser = function(name, age){
  // ...
    return {
        name: name,
    age: age
    }
}

// 简写方式
let getUser = function(name, age){
  // ...
    return {
        name,
    age
    }
}
```

也可以混用：

```
// 原本书写方式
let getUser = function(name, age){
  // ...
    return {
        name: name,
    age: 18
    }
}

// 简写方式
let getUser = function(name, age){
  // ...
    return {
        name,
    age: 18
```

```
    }
}
```

6. 对象属性存在性检测

6.1 使用 in 关键字

该方法可以判断**对象的自有属性和继承来的属性**是否存在。

```
let user = {name: "leo"};
"name" in user;            //true，自有属性存在
"age"  in user;            //false
"toString" in user;     //true，是一个继承属性
```

6.2使用对象的 hasOwnProperty() 方法。

该方法只能判断**自有属性**是否存在，对于**继承属性**会返回 `false` 。

```
let user = {name: "leo"};
user.hasOwnProperty("name");       //true，自有属性中有 name
user.hasOwnProperty("age");        //false，自有属性中不存在 age
user.hasOwnProperty("toString");   //false，这是一个继承属性，但不是自有属性
```

6.3 用 undefined 判断

该方法可以判断对象的**自有属性和继承属性**。

```
let user = {name: "leo"};
user.name !== undefined;        // true
user.age  !== undefined;        // false
user.toString !== undefined     // true
```

该方法存在一个问题，如果属性的值就是 `undefined` 的话，该方法不能返回想要的结果：

```
let user = {name: undefined};
user.name !== undefined;        // false，属性存在，但值是undefined
```

```
user.age  !== undefined;        // false
user.toString !== undefined;    // true
```

6.4 在条件语句中直接判断

```
let user = {};
if(user.name) user.name = "pingan";
//如果 name 是 undefine, null, false, " ", 0 或 NaN,它将保持不变

user; // {}
```

7. 对象循环遍历

当我们需要遍历对象中每一个属性，可以使用 `for...in` 语句来实现

7.1 for...in 循环

`for...in` 语句以任意顺序遍历一个对象的除 `Symbol` 以外的可枚举属性。 **注意**： `for...in` 不应该应用在一个数组，其中索引顺序很重要。

```
let user = {
    name : "leo",
  age  : 18
}

for(let k in user){
    console.log(k, user[k]);
}
// name leo
// age 18
```

7.2 ES7 新增方法

ES7中新增加的 `Object.values()` 和 `Object.entries()` 与之前的 `Object.keys()` 类似，返回数组类型。

1. Object.keys()

返回一个数组，成员是参数对象自身的（不含继承的）所有**可遍历属性**的健名。

```
let user = { name: "leo", age: 18};
Object.keys(user); // ["name", "age"]
```

2. Object.values()

返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历属性的键值。

```
let user = { name: "leo", age: 18};
Object.values(user); // ["leo", 18]
```

如果参数不是对象，则返回空数组：

```
Object.values(10);   // []
Object.values(true); // []
```

3. Object.entries()

返回一个数组，成员是参数对象自身的（不含继承的）所有**可遍历属性**的键值对数组。

```
let user = { name: "leo", age: 18};
Object.entries(user);
// [["name","leo"],["age",18]]
```

手动实现 `Object.entries()` 方法：

```
// Generator函数实现：
function* entries(obj){
    for (let k of Object.keys(obj)){
        yield [k ,obj[k]];
    }
}
// 非Generator函数实现：
function entries (obj){
    let arr = [];
    for(let k of Object.keys(obj)){
        arr.push([k, obj[k]]);
    }
```

```
    return arr;
}
```

4. Object.getOwnPropertyNames(Obj)

该方法返回一个数组，它包含了对象 `Obj` 所有拥有的属性（**无论是否可枚举**）的名称。

```
let user = { name: "leo", age: 18};
Object.getOwnPropertyNames(user);
// ["name", "age"]
```

二、对象拷贝

参考文章《搞不懂JS中赋值·浅拷贝·深拷贝的请看这里》

1. 赋值操作

首先回顾下基本数据类型和引用数据类型：

- 基本类型

概念：基本类型值在内存中占据固定大小，保存在 `栈内存` 中（不包含 `闭包` 中的变量）。 常见包括：undefined,null,Boolean,String,Number,Symbol

- 引用类型

概念：引用类型的值是对象，保存在 `堆内存` 中。而栈内存存储的是对象的变量标识符以及对象在堆内存中的存储地址(引用)，引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。 常见包括：Object,Array,Date,Function,RegExp等

1.1 基本数据类型赋值

在栈内存中的数据发生数据变化的时候，系统会自动为新的变量分配一个新的之值在栈内存中，两个变量相互独立，互不影响的。

```
let user  = "leo";
let user1 = user;
user1 = "pingan";
console.log(user);  // "leo"
console.log(user1); // "pingan"
```

1.2 引用数据类型赋值

在 JavaScript 中，变量不存储对象本身，而是存储其“内存中的地址”，换句话说就是存储对其的“引用”。 如下面 `leo` 变量只是保存对 `user` 对象对应引用：

```
let user = { name: "leo", age: 18};
let leo  = user;
```

其他变量也可以引用 `user` 对象：

```
let leo1 = user;
let leo2 = user;
```

但是由于变量保存的是引用，所以当我们修改变量 `leo` \ `leo1` \ `leo2` 这些值时，**也会改动到引用对象** `user` ，但当 `user` 修改，则其他引用该对象的变量，值都会发生变化：

```
leo.name = "pingan";
console.log(leo);   // {name: "pingan", age: 18}
console.log(leo1);  // {name: "pingan", age: 18}
console.log(leo2);  // {name: "pingan", age: 18}
console.log(user);  // {name: "pingan", age: 18}

user.name = "pingan8787";
console.log(leo);   // {name: "pingan8787", age: 18}
console.log(leo1);  // {name: "pingan8787", age: 18}
console.log(leo2);  // {name: "pingan8787", age: 18}
console.log(user);  // {name: "pingan8787", age: 18}
```

这个过程中涉及变量地址指针指向问题，这里暂时不展开讨论，有兴趣的朋友可以网上查阅相关资料。

2. 对象比较

当两个变量引用同一个对象时，它们无论是 `==` 还是 `===` 都会返回 `true` 。

```
let user = { name: "leo", age: 18};
let leo  = user;
let leo1 = user;
leo ==  leo1;   // true
leo === leo1;   // true
leo ==  user;   // true
leo === user;   // true
```

但如果两个变量是空对象 `{}` ，则不相等：

```
let leo1 = {};
let leo2 = {};
leo1 ==  leo2;  // false
leo1 === leo2;  // false
```

3. 浅拷贝

3.1 概念

概念：**新的对象复制已有对象中非对象属性的值和对象属性的引用**。也可以理解为：**一个新的对象直接拷贝已存在的对象的对象属性的引用**，即浅拷贝。

浅拷贝**只对第一层属性进行了拷贝**，当第一层的属性值是基本数据类型时，新的对象和原对象互不影响，但是如果第一层的属性值是复杂数据类型，那么新对象和原对象的属性值其指向的是同一块内存地址。

通过示例代码演示没有使用浅拷贝场景：

```
// 示例1 对象原始拷贝
let user = { name: "leo", skill: { JavaScript: 90, CSS: 80}};
let leo = user;
leo.name = "leo1";
leo.skill.CSS = 90;
console.log(leo.name);      // "leo1"
console.log(user.name);     // "leo1"
console.log(leo.skill.CSS); // 90
console.log(user.skill.CSS);// 90

// 示例2 数组原始拷贝
```

```
let user = ["leo", "pingan", {name: "pingan8787"}];
let leo  = user;
leo[0] = "pingan888";
leo[2]["name"] = "pingan999";
console.log(leo[0]);          // "pingan888"
console.log(user[0]);         // "pingan888"
console.log(leo[2]["name"]);  // "pingan999"
console.log(user[2]["name"]); // "pingan999"
```

从上面示例代码可以看出： 由于对象被直接拷贝，相当于拷贝 **引用数据类型** ，所以在新对象修改任何值时，都会改动到源数据。

接下来实现浅拷贝，对比以下。

3.2 实现浅拷贝

1. Object.assign()

语法： `Object.assign(target, ...sources)` ES6中拷贝对象的方法，接受的第一个参数是拷贝的目标target，剩下的参数是拷贝的源对象sources（可以是多个）。 详细介绍，可以阅读文档《MDN Object.assign》。

```
// 示例1 对象浅拷贝
let user = { name: "leo", skill: { JavaScript: 90, CSS: 80}};
let leo = Object.assign({}, user);
leo.name = "leo1";
leo.skill.CSS = 90;
console.log(leo.name);      // "leo1" ⚠ 差异!
console.log(user.name);     // "leo"  ⚠ 差异!
console.log(leo.skill.CSS); // 90
console.log(user.skill.CSS);// 90

// 示例2 数组浅拷贝
let user = ["leo", "pingan", {name: "pingan8787"}];
let leo  = user;
leo[0] = "pingan888";
leo[2]["name"] = "pingan999";
console.log(leo[0]);          // "pingan888"  ⚠ 差异!
console.log(user[0]);         // "leo"        ⚠ 差异!
console.log(leo[2]["name"]);  // "pingan999"
console.log(user[2]["name"]); // "pingan999"
```

从打印结果可以看出，浅拷贝只是在根属性(对象的第一层级)创建了一个新的对象，但是对于属性的值是对象的话只会拷贝一份相同的内存地址。

`Object.assign()` 使用注意：

- 只拷贝源对象的自身属性（不拷贝继承属性）；
- 不会拷贝对象不可枚举的属性；
- 属性名为 `Symbol` 值的属性，可以被Object.assign拷贝；
- `undefined` 和 `null` 无法转成对象，它们不能作为 `Object.assign` 参数，但是可以作为源对象。

```
Object.assign(undefined); // 报错
Object.assign(null);      // 报错

Object.assign({}, undefined); // {}
Object.assign({}, null);      // {}

let user = {name: "leo"};
Object.assign(user, undefined) === user; // true
Object.assign(user, null)      === user; // true
```

2. Array.prototype.slice()

语法： `arr.slice([begin[, end]])` `slice()` 方法返回一个新的数组对象，这一对象是一个由 `begin` 和 `end` 决定的原数组的浅拷贝（包括 `begin`，不包括 `end`）。原始数组不会被改变。 详细介绍，可以阅读文档《MDN Array slice》。

```
// 示例 数组深拷贝
let user = ["leo", "pingan", {name: "pingan8787"}];
let leo  = Array.prototype.slice.call(user);
leo[0] = "pingan888";
leo[2]["name"] = "pingan999";
console.log(leo[0]);          // "pingan888"  ⚠ 差异!
console.log(user[0]);         // "leo"        ⚠ 差异!
console.log(leo[2]["name"]);  // "pingan999"
console.log(user[2]["name"]); // "pingan999"
```

3. Array.prototype.concat()

语法： `var new_array = old_array.concat(value1[, value2[, ...[, valueN]]])` `concat()` 方法用于合并两个或多个数组。此方法不会更改现有数组，而是返回一个新数组。 详细介绍，可以阅读文档《MDN

Array concat》。

```
let user  = [{name: "leo"},   {age: 18}];
let user1 = [{age: 20},{addr: "fujian"}];
let user2 = user.concat(user1);
user1[0]["age"] = 25;
console.log(user);  // [{"name":"leo"},{"age":18}]
console.log(user1); // [{"age":25},{"addr":"fujian"}]
console.log(user2); // [{"name":"leo"},{"age":18},{"age":25},{"addr":"fujian"}]
```

`Array.prototype.concat` 也是一个浅拷贝，只是在根属性(对象的第一层级)创建了一个新的对象，但是对于属性的值是对象的话只会拷贝一份相同的内存地址。

4. 拓展运算符（...）

语法：`var cloneObj = { ...obj };` 扩展运算符也是浅拷贝，对于值是对象的属性无法完全拷贝成2个不同对象，但是如果属性都是基本类型的值的话，使用扩展运算符也是优势方便的地方。

```
let user = { name: "leo", skill: { JavaScript: 90, CSS: 80}};
let leo = {...user};
leo.name = "leo1";
leo.skill.CSS = 90;
console.log(leo.name);      // "leo1" ⚠ 差异!
console.log(user.name);     // "leo"  ⚠ 差异!
console.log(leo.skill.CSS); // 90
console.log(user.skill.CSS);// 90
```

3.3 手写浅拷贝

实现原理：新的对象复制已有对象中非对象属性的值和对象属性的**引用**,也就是说对象属性并不复制到内存。

```
function cloneShallow(source) {
    let target = {};
    for (let key in source) {
        if (Object.prototype.hasOwnProperty.call(source, key)) {
            target[key] = source[key];
        }
    }
    return target;
}
```

- **for in**

for...in语句以任意顺序遍历一个对象自有的、继承的、`可枚举的`、非Symbol的属性。对于每个不同的属性，语句都会被执行。

- **hasOwnProperty**

该函数返回值为布尔值，所有继承了 Object 的对象都会继承到 `hasOwnProperty` 方法，和 `in` 运算符不同，该函数会忽略掉那些从原型链上继承到的属性和自身属性。 语法：`obj.hasOwnProperty(prop)` `prop` 是要检测的属性**字符串名称**或者 `Symbol` 。

4. 深拷贝

4.1 概念

复制变量值，对于引用数据，则递归至基本类型后，再复制。深拷贝后的对象**与原来的对象完全隔离**，互不影响，对一个对象的修改并不会影响另一个对象。

4.2 实现深拷贝

1. JSON.parse(JSON.stringify())

其原理是把一个对象序列化成为一个JSON字符串，将对象的内容转换成字符串的形式再保存在磁盘上，再用 `JSON.parse()` 反序列化将JSON字符串变成一个新的对象。

```
let user = { name: "leo", skill: { JavaScript: 90, CSS: 80}};
let leo = JSON.parse(JSON.stringify(user));
leo.name = "leo1";
leo.skill.CSS = 90;
console.log(leo.name);      // "leo1" ⚠ 差异!
console.log(user.name);     // "leo"  ⚠ 差异!
console.log(leo.skill.CSS); // 90 ⚠ 差异!
console.log(user.skill.CSS);// 80 ⚠ 差异!
```

`JSON.stringify()` 使用注意：

- 拷贝的对象的值中如果有函数， `undefined` ， `symbol` 则经过 `JSON.stringify()` `序列化后的JSON字符串中这个键值对会消失；

- 无法拷贝不可枚举的属性，无法拷贝对象的原型链；
- 拷贝 `Date` 引用类型会变成字符串；
- 拷贝 `RegExp` 引用类型会变成空对象；
- 对象中含有 `NaN` 、 `Infinity` 和 `-Infinity` ，则序列化的结果会变成 `null` ；
- 无法拷贝对象的循环应用(即 `obj[key] = obj`)。

2. 第三方库

4.3 手写深拷贝

核心思想是**递归**，遍历对象、数组直到里边都是基本数据类型，然后再去复制，就是深度拷贝。 实现代码：

```
const isObject = obj => typeof obj === 'object' && obj != null;

function cloneDeep(source) {
    if (!isObject(source)) return source; // 非对象返回自身
    const target = Array.isArray(source) ? [] : {};
    for(var key in source) {
        if (Object.prototype.hasOwnProperty.call(source, key)) {
            if (isObject(source[key])) {
                target[key] = cloneDeep(source[key]); // 注意这里
            } else {
                target[key] = source[key];
            }
        }
    }
    return target;
}
```

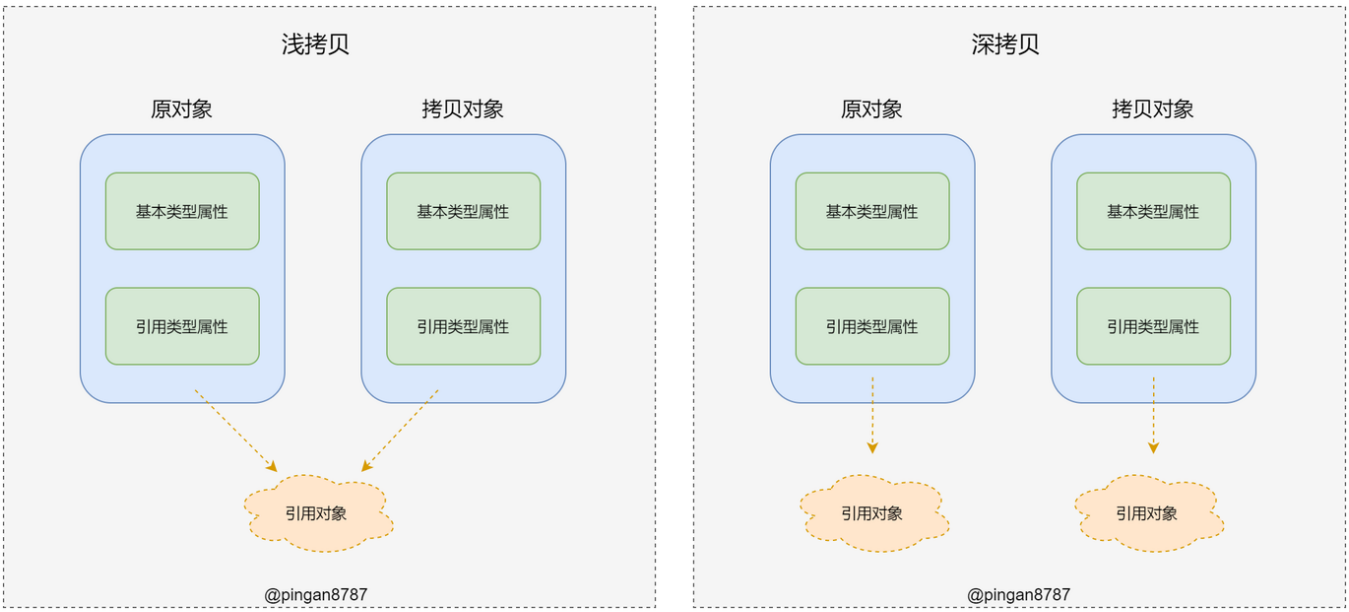
该方法缺陷： 遇到循环引用，会陷入一个循环的递归过程，从而导致爆栈。 其他写法，可以阅读《如何写出一个惊艳面试官的深拷贝?》 。

5. 小结

浅拷贝：将对象的每个属性进行依次复制，但是当对象的属性值是引用类型时，实质复制的是其引用，当引用指向的值改变时也会跟着变化。

深拷贝：复制变量值，对于引用数据，则递归至基本类型后，再复制。深拷贝后的对象**与原来的对象完全隔离**，互不影响，对一个对象的修改并不会影响另一个对象。

深拷贝和浅拷贝是针对复杂数据类型来说的，浅拷贝只拷贝一层，而深拷贝是层层拷贝。



三、垃圾回收机制（GC）

垃圾回收（Garbage Collection，缩写为GC）是一种自动的存储器管理机制。当某个程序占用的一部分内存空间不再被这个程序访问时，这个程序会借助垃圾回收算法向操作系统归还这部分内存空间。垃圾回收器可以减轻程序员的负担，也减少程序中的错误。垃圾回收最早起源于LISP语言。 目前许多语言如Smalltalk、Java、C#和D语言都支持垃圾回收器，我们熟知的 JavaScript 具有自动垃圾回收机制。

在 JavaScript 中，原始类型的数据被分配到栈空间中，引用类型的数据会被分配到堆空间中。

1. 栈空间中的垃圾回收

当函数 `showName` 调用完成后，通过下移 ESP（Extended Stack Pointer）指针，来销毁 `showName` 函数，之后调用其他函数时，将覆盖掉旧内存，存放另一个函数的执行上下文，实现垃圾回收。

图片来自《浏览器工作原理与实践》

2. 堆空间中的垃圾回收

堆中数据垃圾回收策略的基础是：**代际假说（The Generational Hypothesis）**。即：

1. 大部分对象在内存中存在时间极短，很多对象很快就不可访问。
2. 不死的对象将活得更久。

这两个特点不仅仅适用于 JavaScript，同样适用于大多数的动态语言，如 Java、Python 等。 V8 引擎将堆空间分为**新生代**（存放生存**时间短**的对象）和**老生代**（存放生存**时间长**的对象）两个区域，并使用不同的垃圾回收器。

- 副垃圾回收器，主要负责新生代的垃圾回收。
- 主垃圾回收器，主要负责老生代的垃圾回收。

不管是哪种垃圾回收器，都使用相同垃圾回收流程：**标记活动对象和非活动对象，回收非活动对象的内存，最后内存整理。** **

1.1 副垃圾回收器

使用 Scavenge 算法处理，将新生代空间对半分为两个区域，一个对象区域，一个空闲区域。

图片来自《浏览器工作原理与实践》

执行流程：

- 新对象存在在**对象区域**，当对象区域将要写满时，执行一次垃圾回收；
- 垃圾回收过程中，首先对对象区域中的垃圾做标记，然后副垃圾回收器将存活的对象复制并有序排列到空闲区域，相当于完成内存整理。
- 复制完成后，将对象区域和空闲区域翻转，完成垃圾回收操作，这也让新生代中两块区域无限重复使用。

当然，这也存在一些问题：若复制操作的数据较大则影响清理效率。 JavaScript 引擎的解决方式是：将新生代区域设置得比较小，并采用对象晋升策略（经过两次回收仍存活的对象，会被移动到老生区），避免因为新生代区域较小引起存活对象装满整个区域的问题。

1.2 主垃圾回收器

分为：**标记 - 清除（Mark-Sweep）算法**，和**标记 - 整理（Mark-Compact）算法**。

a)标记 - 清除（Mark-Sweep）算法 过程：

- 标记过程：从一组根元素开始遍历整个元素，能到达的元素为活动对象，反之为垃圾数据；
- 清除过程：清理被标记的数据，并产生大量碎片内存。（缺点：导致大对象无法分配到足够的连续内存）

图片来自《浏览器工作原理与实践》

b)标记 - 整理（Mark-Compact）算法 过程：

- 标记过程：从一组根元素开始遍历整个元素，能到达的元素为活动对象，反之为垃圾数据；
- 整理过程：将所有存活的对象，向一段移动，然后清除端边界以外的内容。

图片来自《浏览器工作原理与实践》

3. 拓展阅读

1.《图解Java 垃圾回收机制》 2.《MDN 内存管理》

四、对象方法和 this

1. 对象方法

具体介绍可阅读 《MDN 方法的定义》 。 将作为对象属性的方法称为“对象方法”，如下面 `user` 对象的 `say` 方法：

```
let user = {};
let say = function(){console.log("hello!")};

user.say = say;  // 赋值到对象上
user.say(); // "hello!"
```

也可以使用更加简洁的方法：

```
let user = {
    say: function(){}
```

```
  // 简写为
    say (){console.log("hello!")}

    // ES8 async 方法
    async say (){/.../}
}
user.say();
```

当然对象方法的名称，还支持计算的属性名称作为方法名：

```
const hello = "Hello";
let user = {
    ['say' + hello](){console.log("hello!")}
}
user['say' + hello](); // "hello!"
```

另外需要注意的是：所有方法定义不是构造函数，如果您尝试实例化它们，将抛出 `TypeError` 。

```
let user = {
    say(){};
}
new user.say; // TypeError: user.say is not a constructor
```

2. this

2.1 this 简介

当对象方法需要使用对象中的属性，可以使用 `this` 关键字：

```
let user = {
    name : 'leo',
  say(){ console.log(`hello ${this.name}`)}
}

user.say(); // "hello leo"
```

当代码 `user.say()` 执行过程中， `this` 指的是 `user` 对象。当然也可以直接使用变量名 `user` 来引用 `say()` 方法：

```
let user = {
    name : 'leo',
  say(){ console.log(`hello ${user.name}`)}
}

user.say(); // "hello leo"
```

但是这样并不安全，因为 `user` 对象可能赋值给另外一个变量，并且将其他值赋值给 `user` 对象，就可能导致报错：

```
let user = {
    name : 'leo',
  say(){ console.log(`hello ${user.name}`)}
}

let leo = user;
user = null;

leo.say(); // Uncaught TypeError: Cannot read property 'name' of null
```

但将 `user.name` 改成 `this.name` 代码便正常运行。

2.2 this 取值

`this` 的值是在 **代码运行时计算出来** 的，它的值取决于代码上下文：

```
let user = { name: "leo"};
let admin = {name: "pingan"};
let say = function (){
    console.log(`hello ${this.name}`)
};

user.fun = say;
admin.fun = say;

// 函数内部 this 是指“点符号前面”的对象
user.fun();     // "hello leo"
admin.fun();    // "hello pingan"
admin['fun'](); // "hello pingan"
```

规则：如果 `obj.fun()` 被调用，则 `this` 在 `fun` 函数调用期间是 `obj` ，所以上面的 `this` 先是 `user` ，然后是 `admin` 。

但是在全局环境中，无论是否开启严格模式， `this` 都指向全局对象

```
console.log(this == window); // true

let a = 10;
this.b = 10;
a === this.b; // true
```

2.3 箭头函数没有自己的 this

箭头函数比较特别，没有自己的 `this` ，如果有引用 `this` 的话，则指向外部正常函数，下面例子中， `this` 指向 `user.say()` 方法：

```
let user = {
    name : 'leo',
  say : () => {
    console.log(`hello ${this.name}`);
  },
  hello(){
        let fun = () => console.log(`hello ${this.name}`);
    fun();
    }
}

user.say();   // hello      => say() 外部函数是 window
user.hello(); // hello leo  => fun() 外部函数是 hello
```

2.4 call / apply / bind

详细可以阅读《js基础-关于call,apply,bind的一切》 。 当我们想把 `this` 值绑定到另一个环境中，就可以使用 `call` / `apply` / `bind` 方法实现：

```
var user = { name: 'leo' };
var name = 'pingan';
function fun(){
    return console.log(this.name); // this 的值取决于函数调用方式
}
```

```
fun();           // "pingan"
fun.call(user);  // "leo"
fun.apply(user); // "leo"
```

注意：这里的 `var name = 'pingan';` 需要使用 `var` 来声明，使用 `let` 的话， `window` 上将没有 `name` 变量。

三者语法如下：

```
fun.call(thisArg, param1, param2, ...)
fun.apply(thisArg, [param1,param2,...])
fun.bind(thisArg, param1, param2, ...)
```

五、构造函数和 new 运算符

1. 构造函数

构造函数的作用在于 **实现可重用的对象创建代码** 。 通常，对于构造函数有两个约定：

- 命名时首字母大写；
- 只能使用 `new` 运算符执行。

`new` **运算符**创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例。 语法如下：

```
new constructor[([arguments])]
```

参数如下：

- `constructor` 一个指定对象实例的类型的类或函数。
- `arguments` 一个用于被 `constructor` 调用的参数列表。

2. 简单示例

举个简单示例：

```
function User (name){
    this.name = name;
  this.isAdmin = false;
}
const leo = new User('leo');
console.log(leo.name, leo.isAdmin); // "leo" false
```

3. new 运算符操作过程

当一个函数被使用 `new` 运算符执行时，它按照以下步骤：

1. 一个新的空对象被创建并分配给 `this`。
2. 函数体执行。通常它会修改 `this`，为其添加新的属性。
3. 返回 `this` 的值。

以前面 `User` 方法为例：

```
function User(name) {
  // this = {};（隐式创建）

  // 添加属性到 this
  this.name = name;
  this.isAdmin = false;

  // return this;（隐式返回）
}
const leo = new User('leo');
console.log(leo.name, leo.isAdmin); // "leo" false
```

当我们执行 `new User('leo')` 时，发生以下事情：

1. 一个继承自 `User.prototype` 的新对象被创建；
2. 使用指定参数调用构造函数 `User`，并将 `this` 绑定到新创建的对象；
3. 由构造函数返回的对象就是 `new` 表达式的结果。如果构造函数没有显式返回一个对象，则使用步骤1创建的对象。

需要注意：

1. 一般情况下，构造函数不返回值，但是开发者可以选择主动返回对象，来覆盖正常的对象创建步骤；
2. `new User` 等同于 `new User()`，只是没有指定参数列表，即 `User` 不带参数的情况；

```
let user = new User; // <-- 没有参数
// 等同于
let user = new User();
```

3. 任何函数都可以作为构造器，即都可以使用 `new` 运算符运行。

4. 构造函数中的方法

在构造函数中，也可以将方法绑定到 `this` 上：

```
function User (name){
    this.name = name;
  this.isAdmin = false;
    this.sayHello = function(){
        console.log("hello " + this.name);
    }
}
const leo = new User('leo');
console.log(leo.name, leo.isAdmin); // "leo" false
leo.sayHello(); // "hello leo"
```

六、可选链 "?."

详细介绍可以查看 《MDN 可选链操作符》 。

1. 背景介绍

在实际开发中，常常出现下面几种报错情况：

```
// 1. 对象中不存在指定属性
const leo = {};
console.log(leo.name.toString());
// Uncaught TypeError: Cannot read property 'toString' of undefined

// 2. 使用不存在的 DOM 节点属性
const dom = document.getElementById("dom").innerHTML;
// Uncaught TypeError: Cannot read property 'innerHTML' of null
```

在可选链 `?.` 出现之前，我们会使用短路操作 `&&` 运算符来解决该问题：

```
const leo = {};
console.log(leo && leo.name && leo.name.toString()); // undefined
```

这种写法的缺点就是 **太麻烦了** 。

2. 可选链介绍

可选链 `?.` 是一种 **访问嵌套对象属性的防错误方法** 。即使中间的属性不存在，也不会出现错误。 如果可选链 `?.` 前面部分是 `undefined` 或者 `null`，它会停止运算并返回 `undefined`。

语法：

```
obj?.prop
obj?.[expr]
arr?.[index]
func?.(args)
```

** **我们改造前面示例代码：**

```
// 1. 对象中不存在指定属性
const leo = {};
console.log(leo?.name?.toString());
// undefined

// 2. 使用不存在的 DOM 节点属性
const dom = document?.getElementById("dom")?.innerHTML;
// undefined
```

3. 使用注意

可选链虽然好用，但需要注意以下几点：

1. **不能过度使用可选链**；

我们应该只将 `?.` 使用在一些属性或方法可以不存在的地方，以上面示例代码为例：

```
const leo = {};
console.log(leo.name?.toString());
```

这样写会更好，因为 `leo` 对象是必须存在，而 `name` 属性则可能不存在。

2. **可选链 `?.` 之前的变量必须已声明**；

在可选链 `?.` 之前的变量必须使用 `let/const/var` 声明，否则会报错：

```
leo?.name;
// Uncaught ReferenceError: leo is not defined
```

3. **可选链不能用于赋值**；

```
let object = {};
object?.property = 1;
// Uncaught SyntaxError: Invalid left-hand side in assignment
```

4. **可选链访问数组元素的方法**；

```
let arrayItem = arr?.[42];
```

4.其他情况：?.() 和 ?.[]

需要说明的是 `?.` 是一个特殊的语法结构，而不是一个运算符，它还可以与其 `()` 和 `[]` 一起使用：

4.1 可选链与函数调用 ?.()

`?.()` 用于调用一个可能不存在的函数，比如：

```
let user1 = {
  admin() {
    alert("I am admin");
  }
}

let user2 = {};
```

```
user1.admin?.(); // I am admin
user2.admin?.();
```

`?.()` 会检查它左边的部分：如果 admin 函数存在，那么就调用运行它（对于 `user1`）。否则（对于 `user2`）运算停止，没有错误。

4.2 可选链和表达式 ?.[]

`?.[]` 允许从一个可能不存在的对象上安全地读取属性。

```
let user1 = {
  firstName: "John"
};

let user2 = null; // 假设，我们不能授权此用户

let key = "firstName";

alert( user1?.[key] ); // John
alert( user2?.[key] ); // undefined

alert( user1?.[key]?.something?.not?.existing); // undefined
```

5. 可选链 `?.` 语法总结

可选链 `?.` 语法有三种形式：

1. `obj?.prop` —— 如果 `obj` 存在则返回 `obj.prop`，否则返回 `undefined`。
2. `obj?.[prop]` —— 如果 `obj` 存在则返回 `obj[prop]`，否则返回 `undefined`。
3. `obj?.method()` —— 如果 `obj` 存在则调用 `obj.method()`，否则返回 `undefined`。

正如我们所看到的，这些语法形式用起来都很简单直接。`?.` 检查左边部分是否为 `null/undefined`，如果不是则继续运算。`?.` 链使我们能够安全地访问嵌套属性。

七、Symbol

规范规定，JavaScript 中对象的属性只能为 **字符串类型** 或者 **Symbol类型**，毕竟我们也只见过这两种类型。

1.概念介绍

ES6引入`Symbol`作为一种新的**原始数据类型**，表示**独一无二**的值，主要是为了**防止属性名冲突**。 ES6之后，JavaScript一共有其中数据类型：`Symbol`、`undefined`、`null`、`Boolean`、`String`、`Number`、`Object`。 **简单使用**:

```
let leo = Symbol();
typeof leo; // "symbol"
```

Symbol 支持传入参数作为 Symbol 名，方便代码调试： **

```
let leo = Symbol("leo");
```

2.注意事项**

- `Symbol`函数不能用`new`，会报错。

由于`Symbol`是一个原始类型，不是对象，所以不能添加属性，它是类似于字符串的数据类型。

```
let leo = new Symbol()
// Uncaught TypeError: Symbol is not leo constructor
```

- `Symbol`都是不相等的，**即使参数相同**。

```
// 没有参数
let leo1 = Symbol();
let leo2 = Symbol();
leo1 === leo2; // false

// 有参数
let leo1 = Symbol('leo');
let leo2 = Symbol('leo');
leo1 === leo2; // false
```

- `Symbol`不能与其他类型的值计算，会报错。

```
let leo = Symbol('hello');
leo + " world!";  // 报错
```

```
`${leo} world!`;  // 报错
```

- `Symbol` 不能自动转换为字符串，只能显式转换。

```
let leo = Symbol('hello');
alert(leo);
// Uncaught TypeError: Cannot convert a Symbol value to a string

String(leo);    // "Symbol(hello)"
leo.toString(); // "Symbol(hello)"
```

- `Symbol` 可以转换为布尔值，但不能转为数值：

```
let a1 = Symbol();
Boolean(a1);
!a1;        // false
Number(a1); // TypeError
a1 + 1 ;    // TypeError
```

- `Symbol` 属性不参与 `for...in/of` 循环。

```
let id = Symbol("id");
let user = {
  name: "Leo",
  age: 30,
  [id]: 123
};

for (let key in user) console.log(key); // name, age (no symbols)

// 使用 Symbol 任务直接访问
console.log( "Direct: " + user[id] );
```

3. 字面量中使用 Symbol 作为属性名

在对象字面量中使用 `Symbol` 作为属性名时，需要使用 **方括号** （ `[]` ），如 `[leo]: "leo"` 。 好处：防止同名属性，还有防止键被改写或覆盖。

```
let leo = Symbol();
// 写法1
let user = {};
```

```
user[leo] = 'leo';

// 写法2
let user = {
    [leo] : 'leo'
}

// 写法3
let user = {};
Object.defineProperty(user, leo, {value : 'leo' });

// 3种写法 结果相同
user[leo]; // 'leo'
```

需要注意： Symbol作为对象属性名时，不能用点运算符，并且必须放在方括号内。

```
let leo = Symbol();
let user = {};
// 不能用点运算
user.leo = 'leo';
user[leo] ; // undefined
user['leo'] ; // 'leo'

// 必须放在方括号内
let user = {
    [leo] : function (text){
        console.log(text);
    }
}
user[leo]('leo'); // 'leo'

// 上面等价于 更简洁
let user = {
    [leo](text){
        console.log(text);
    }
}
```

常常还用于创建一组常量，保证所有值不相等：

```
let user = {};
user.list = {
    AAA: Symbol('Leo'),
    BBB: Symbol('Robin'),
```

```
    CCC: Symbol('Pingan')
}
```

4.应用：消除魔术字符串

魔术字符串：指代码中多次出现，强耦合的字符串或数值，应该避免，而使用含义清晰的变量代替。

```
function fun(name){
    if(name == 'leo') {
        console.log('hello');
    }
}
fun('leo');   // 'hello' 为魔术字符串
```

常使用变量，消除魔术字符串：

```
let obj = {
    name: 'leo'
};
function fun(name){
    if(name == obj.name){
        console.log('hello');
    }
}
fun(obj.name); // 'hello'
```

使用`Symbol`消除强耦合，使得不需关系具体的值:

```
let obj = {
    name: Symbol()
};
function fun (name){
    if(name == obj.name){
        console.log('hello');
    }
}
fun(obj.name); // 'hello'
```

5.属性名遍历

Symbol作为属性名遍历，不出现在 `for...in` 、 `for...of` 循环，也不被 `Object.keys()` 、 `Object.getOwnPropertyNames()` 、 `JSON.stringify()` 返回。

```
let leo = Symbol('leo'), robin = Symbol('robin');
let user = {
    [leo]:'18', [robin]:'28'
}
for(let k of Object.values(user)){console.log(k)}
// 无输出

let user = {};
let leo = Symbol('leo');
Object.defineProperty(user, leo, {value: 'hi'});
for(let k in user){
    console.log(k); // 无输出
}
Object.getOwnPropertyNames(user);   // []
Object.getOwnPropertySymbols(user); // [Symbol(leo)]
```

`Object.getOwnPropertySymbols` 方法返回一个数组，包含当前对象所有用做属性名的Symbol值。

```
let user = {};
let leo = Symbol('leo');
let pingan = Symbol('pingan');
user[leo] = 'hi leo';
user[pingan] = 'hi pingan';
let obj = Object.getOwnPropertySymbols(user);
obj; //  [Symbol(leo), Symbol(pingan)]
```

另外可以使用 `Reflect.ownKeys` 方法可以返回所有类型的键名，包括常规键名和 Symbol 键名。

```
let user = {
    [Symbol('leo')]: 1,
    age : 2,
    address : 3,
}
Reflect.ownKeys(user); // ['age', 'address',Symbol('leo')]
```

由于Symbol值作为名称的属性不被常规方法遍历获取，因此常用于定义对象的一些非私有，且内部使用的方法。

6. Symbol.for()、Symbol.keyFor()

6.1 Symbol.for()

用于重复使用一个Symbol值，接收一个**字符串**作为参数，若存在用此参数作为名称的Symbol值，返回这个Symbol，否则新建并返回以这个参数为名称的Symbol值。

```
let leo = Symbol.for('leo');
let pingan = Symbol.for('pingan');
leo === pingan;  // true
```

`Symbol()` 和 `Symbol.for()` 区别：

```
Symbol.for('leo') === Symbol.for('leo'); // true
Symbol('leo') === Symbol('leo');         // false
```

6.2 Symbol.keyFor()

用于返回一个已使用的Symbol类型的key:

```
let leo = Symbol.for('leo');
Symbol.keyFor(leo);   //  'leo'

let leo = Symbol('leo');
Symbol.keyFor(leo);   //  undefined
```

7. 内置的Symbol值

ES6提供11个内置的Symbol值，指向语言内部使用的方法：

7.1 Symbol.hasInstance

当其他对象使用 `instanceof` 运算符，判断是否为该对象的实例时，会调用这个方法。比如， `foo instanceof Foo` 在语言内部，实际调用的是 `Foo[Symbol.hasInstance](foo)` 。

```
class P {
    [Symbol.hasInstance](a){
        return a instanceof Array;
    }
}
[1, 2, 3] instanceof new P(); // true
```

P是一个类，new P()会返回一个实例，该实例的`Symbol.hasInstance`方法，会在进行`instanceof`运算时自动调用，判断左侧的运算子是否为`Array`的实例。

7.2 Symbol.isConcatSpreadable

值为布尔值，表示该对象用于`Array.prototype.concat()`时，是否可以展开。

```
let a = ['aa','bb'];
['cc','dd'].concat(a, 'ee');
// ['cc', 'dd', 'aa', 'bb', 'ee']
a[Symbol.isConcatSpreadable]; // undefined
let b = ['aa','bb'];
b[Symbol.isConcatSpreadable] = false;
['cc','dd'].concat(b, 'ee');
// ['cc', 'dd',[ 'aa', 'bb'], 'ee']
```

7.3 Symbol.species

指向一个构造函数，在创建衍生对象时会使用，使用时需要用`get`取值器。

```
class P extends Array {
    static get [Symbol.species](){
        return this;
    }
}
```

解决下面问题：

```
// 问题：  b应该是 Array 的实例，实际上是 P 的实例
class P extends Array{}
let a = new P(1,2,3);
let b = a.map(x => x);
```

```
b instanceof Array; // true
b instanceof P; // true
// 解决：  通过使用 Symbol.species
class P extends Array {
  static get [Symbol.species]() { return Array; }
}
let a = new P();
let b = a.map(x => x);
b instanceof P;     // false
b instanceof Array; // true
```

7.4 Symbol.match

当执行 `str.match(myObject)`，传入的属性存在时会调用，并返回该方法的返回值。

```
class P {
    [Symbol.match](string){
        return 'hello world'.indexOf(string);
    }
}
'h'.match(new P());   // 0
```

7.5 Symbol.replace

当该对象被 `String.prototype.replace` 方法调用时，会返回该方法的返回值。

```
let a = {};
a[Symbol.replace] = (...s) => console.log(s);
'Hello'.replace(a , 'World') // ["Hello", "World"]
```

7.6 Symbol.hasInstance

当该对象被 `String.prototype.search` 方法调用时，会返回该方法的返回值。

```
class P {
    constructor(val) {
        this.val = val;
    }
    [Symbol.search](s){
        return s.indexOf(this.val);
```

```
    }
}
'hileo'.search(new P('leo')); // 2
```

7.7 Symbol.split

当该对象被 `String.prototype.split` 方法调用时，会返回该方法的返回值。

```
// 重新定义了字符串对象的split方法的行为
class P {
    constructor(val) {
        this.val = val;
    }
    [Symbol.split](s) {
        let i = s.indexOf(this.val);
        if(i == -1) return s;
        return [
            s.substr(0, i),
            s.substr(i + this.val.length)
        ]
    }
}
'helloworld'.split(new P('hello')); // ["hello", ""]
'helloworld'.split(new P('world')); // ["", "world"]
'helloworld'.split(new P('leo'));   // "helloworld"
```

7.8 Symbol.iterator

对象进行 `for...of` 循环时，会调用 `Symbol.iterator` 方法，返回该对象的默认遍历器。

```
class P {
    *[Symbol.interator]() {
        let i = 0;
        while(this[i] !== undefined ) {
            yield this[i];
            ++i;
        }
    }
}
let a = new P();
a[0] = 1;
a[1] = 2;
```

```
for (let k of a){
    console.log(k);
}
```

7.9.Symbol.toPrimitive

该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。调用时，需要接收一个字符串参数，表示当前运算模式，运算模式有：

- Number : 此时需要转换成数值
- String : 此时需要转换成字符串
- Default : 此时可以转换成数值或字符串

```
let obj = {
  [Symbol.toPrimitive](hint) {
    switch (hint) {
      case 'number':
        return 123;
      case 'string':
        return 'str';
      case 'default':
        return 'default';
      default:
        throw new Error();
     }
   }
};
2 * obj // 246
3 + obj // '3default'
obj == 'default' // true
String(obj) // 'str'
```

7.10 Symbol.toStringTag

在该对象上面调用 `Object.prototype.toString` 方法时，如果这个属性存在，它的返回值会出现在 `toString` 方法返回的字符串之中，表示对象的类型。也就是说，这个属性可以用来定制 `[object Object]` 或 `[object Array]` 中 `object` 后面的那个字符串。

```
// 例一
({[Symbol.toStringTag]: 'Foo'}.toString())
// "[object Foo]"
```

```
// 例二
class Collection {
  get [Symbol.toStringTag]() {
    return 'xxx';
  }
}
let x = new Collection();
Object.prototype.toString.call(x) // "[object xxx]"
```

7.11 Symbol.unscopables

该对象指定了使用with关键字时，哪些属性会被with环境排除。

```
// 没有 unscopables 时
class MyClass {
  foo() { return 1; }
}
var foo = function () { return 2; };
with (MyClass.prototype) {
  foo(); // 1
}
// 有 unscopables 时
class MyClass {
  foo() { return 1; }
  get [Symbol.unscopables]() {
    return { foo: true };
  }
}
var foo = function () { return 2; };
with (MyClass.prototype) {
  foo(); // 2
}
```

上面代码通过指定 `Symbol.unscopables` 属性，使得 `with` 语法块不会在当前作用域寻找 `foo` 属性，即 `foo` 将指向外层作用域的变量。

八、原始值转换

前面复习到字符串、数值、布尔值等的转换，但是没有讲到对象的转换规则，这部分就一起看看： 。 需要记住几个规则：

1. 所有对象在布尔上下文中都为 `true` ，并且不存在转换为布尔值的操作，只有字符串和数值转换有。

2. 数值转换发生在对象相减或应用数学函数时。如 `Date` 对象可以相减，如 `date1 - date2` 结果为两个时间的差值。
3. 在字符串转换，通常出现在如 `alert(obj)` 这种形式。

当然我们可以使用特殊的对象方法，对字符串和数值转换进行微调。下面介绍三个类型（hint）转换情况：

1. object to string

对象到字符串的转换，当我们对期望一个字符串的对象执行操作时，如 “alert”：

```
// 输出
alert(obj);
// 将对象作为属性键
anotherObj[obj] = 123;
```

2. object to number

对象到数字的转换，例如当我们进行数学运算时：

```
// 显式转换
let num = Number(obj);
// 数学运算（除了二进制加法）
let n = +obj; // 一元加法
let delta = date1 - date2;
// 小于/大于的比较
let greater = user1 > user2;
```

3. object to default

少数情况下，**当运算符“不确定”期望值类型时**。 例如，二进制加法 `+` 可用于字符串（连接），也可以用于数字（相加），所以字符串和数字这两种类型都可以。因此，当二元加法得到对象类型的参数时，它将依据 `"default"` 来对其进行转换。 此外，如果对象被用于与字符串、数字或 symbol 进行 `==` 比较，这时到底应该进行哪种转换也不是很明确，因此使用 `"default"` 。

```
// 二元加法使用默认 hint
let total = obj1 + obj2;
```

```
// obj == number 使用默认 hint
if (user == 1) { ... };
```

4. 类型转换算法

为了进行转换，JavaScript 尝试查找并调用三个对象方法：

1. 调用 `obj[Symbol.toPrimitive](hint)` —— 带有 symbol 键 `Symbol.toPrimitive`（系统 symbol）的方法，如果这个方法存在的话，
2. 否则，如果 hint 是 `"string"` —— 尝试 `obj.toString()` 和 `obj.valueOf()`，无论哪个存在。
3. 否则，如果 hint 是 `"number"` 或 `"default"` —— 尝试 `obj.valueOf()` 和 `obj.toString()`，无论哪个存在。

5. Symbol.toPrimitive

详细介绍可阅读《MDN | Symbol.toPrimitive》。`Symbol.toPrimitive` 是一个内置的 Symbol 值，它是作为对象的函数值属性存在的，当一个对象转换为对应的原始值时，会调用此函数。简单示例介绍：

```
let user = {
  name: "Leo",
  money: 9999,

  [Symbol.toPrimitive](hint) {
    console.log(`hint: ${hint}`);
    return hint == "string" ? `{name: "${this.name}"}` : this.money;
  }
};

alert(user);     // 控制台: hint: string 弹框: {name: "John"}
alert(+user);    // 控制台: hint: number 弹框: 9999
alert(user + 1); // 控制台: hint: default 弹框: 10000
```

6. toString/valueOf

`toString` / `valueOf` 是两个比较早期的实现转换的方法。当没有 `Symbol.toPrimitive`，那么 JavaScript 将尝试找到它们，并且按照下面的顺序进行尝试：

- 对于 “string” hint，`toString -> valueOf`。

- 其他情况，`valueOf -> toString`。

这两个方法必须返回一个原始值。如果 `toString` 或 `valueOf` 返回了一个对象，那么返回值会被忽略。默认情况下，普通对象具有 `toString` 和 `valueOf` 方法：

- `toString` 方法返回一个字符串 `"[object Object]"`。
- `valueOf` 方法返回对象自身。

简单示例介绍：

```
const user = {name: "Leo"};

alert(user); // [object Object]
alert(user.valueOf() === user); // true
```

我们也可以结合 `toString` / `valueOf` 实现前面第 5 点介绍的 `user` 对象：

```
let user = {
  name: "Leo",
  money: 9999,

  // 对于 hint="string"
  toString() {
    return `{name: "${this.name}"}`;
  },

  // 对于 hint="number" 或 "default"
  valueOf() {
    return this.money;
  }

};

alert(user);     // 控制台: hint: string 弹框: {name: "John"}
alert(+user);    // 控制台: hint: number 弹框: 9999
alert(user + 1); // 控制台: hint: default 弹框: 10000
```

总结

本文作为《初中级前端 JavaScript 自测清单》第二部分，介绍的内容以 JavaScript 对象为主，其中有让我眼前一亮的知识点，如 `Symbol.toPrimitive` 方法。我也希望这个清单能帮助大家自测自己的 JavaScript 水平并

查缺补漏，温故知新。