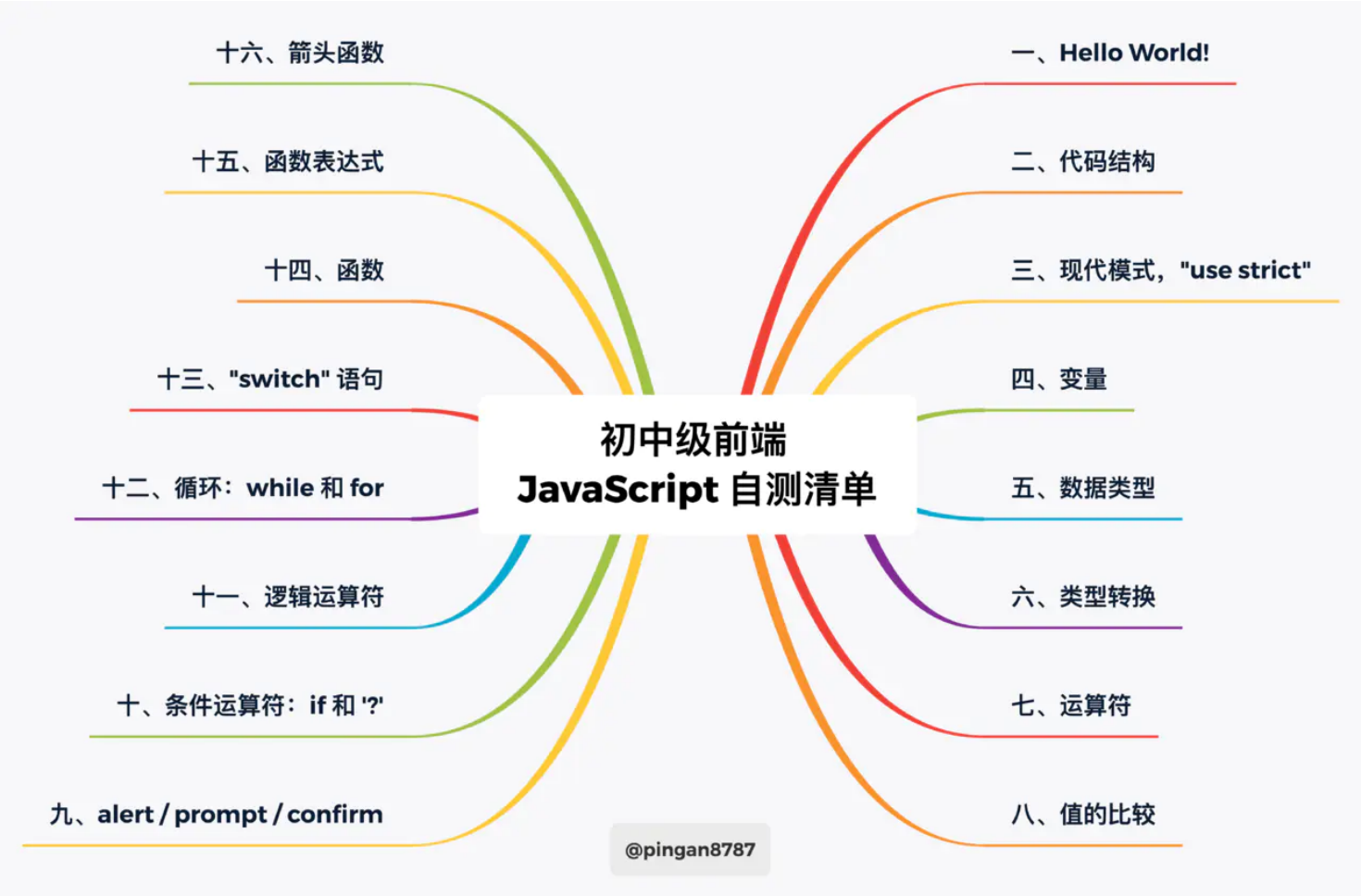
1.2w字 | 初中级前端 JavaScript 自测清单 - 1

前言

接下来开始分享自测清单的内容。



一、Hello World!

1. 脚本引入方式

JavaScript 脚本引入方式有两种：

- `<script>` 标签插入脚本；
- `<script>` 标签 `src` 设置脚本地址。

2. script 标签属性

`<script>` 标签有以下常用属性：

2.1 src

`src` ： 指定外部脚本的URI， 如果设置了 `src` 特性，script 标签内容将会被忽略；

```
<script src="example-url.js"></script>
```

2.2 type

`type` ： 指定引用脚本的语言，属性值为 MIME 类型，包括 `text/javascript` , `text/ecmascript` , `application/javascript` , 和 `application/ecmascript` 。如果没有定义这个属性，脚本会被视作 JavaScript。

ES6 新增了属性值 `module` ，代码会被当做 JavaScript 模块。

```
<script type="text/javascript"></script>
```

2.3 async

`async` 规定一旦脚本可用，则会异步执行。 注意：async 属性仅适用于外部脚本（**只有在使用 src 属性时**）。有多种执行外部脚本的方法： 如果 `async="async"` ： 脚本相对于页面的其余部分异步地执行（当页面继续进行解析时，脚本将被执行）； 如果不使用 `async` 且 `defer="defer"` ： 脚本将在页面完成解析时执行； 如果既不使用 `async` 也不使用 `defer` ： 在浏览器继续解析页面之前，立即读取并执行脚本；

```
<script async="async"></script>
```

2.4 defer

`defer` 属性规定是否对脚本执行进行延迟，直到页面加载为止。

如果您的脚本不会改变文档的内容，可将 defer 属性加入到 `<script>` 标签中，以便加快处理文档的速度。因为浏览器知道它将能够安全地读取文档的剩余部分而不用执行脚本，它将推迟对脚本的解释，直到文档已经显

示给用户为止。

```
<script defer="defer"></script>
```

详细介绍可以阅读《MDN `<script>` 章节》。

二、代码结构

1. 语句

语句是执行行为（action）的语法结构和命令。如：`alert('Hello, world!')` 这样可以用来显示消息的语句。

2. 分号

存在分行符时，多数情况下可以省略分号。但不全是，比如：

```
alert(3 +
1
+ 2);
```

建议新人最好不要省略分号。

3. 注释

单行注释以两个正斜杠字符 `//` 开始。

```
// 注释文本
console.log("leo");
```

多行注释以一个正斜杠和星号开始 `“/*”` 并以一个星号和正斜杆结束 `“*/”`。

```
/*
这是多行注释。
```

```
第二行注释。
*/
console.log("leo");
```

三、现代模式，"use strict"

1. 作用

JavaScript 的严格模式是使用受限制的 JavaScript 的一种方式，从而隐式地退出“草率模式”。

`"use strict"` 指令将浏览器引擎转换为“现代”模式，改变一些内建特性的行为。

2. 使用

通过在脚本文件/函数开头添加 `"use strict";` 声明，即可启用严格模式。 全局开启严格模式：

```
// index.js
"use strict";
const v = "Hi!  I'm a strict mode script!";
```

函数内开启严格模式：

```
// index.js
function strict() {
  'use strict';
  function nested() {
    return "And so am I!";
  }
  return "Hi!  I'm a strict mode function!  " + nested();
}
```

3. 注意点

1. `"use strict"` 需要定义在脚本最顶部（函数内除外），否则严格模式可能无法启用。
2. 一旦进入了严格模式，就无法关闭严格模式。

4. 体验

启用 `"use strict"` 后，为未定义元素赋值将抛出异常：

```
"use strict";
leo = 17; // Uncaught ReferenceError: leo is not defined
```

启用 `"use strict"` 后，试图删除不可删除的属性时会抛出异常：

```
"use strict";
delete Object.prototype; // Uncaught TypeError: Cannot delete property 'prototype' of function Objec
```

详细介绍可以阅读《MDN 严格模式章节》。

四、变量

1. 介绍

变量是数据的“命名存储”。

2. 使用

目前定义变量可以使用三种关键字：var / let / const。三者区别可以阅读《let 和 const 命令》。

```
let name = "leo";
let name = "leo", age, addr;
let name = "leo", age = 27, addr = "fujian";
```

3. 命名建议

变量命名有 2 个限制：

1. 变量名称必须仅包含**字母，数字，符号** `$` 和 `_`。
2. 首字符必须**非数字**。 变量命名还有一些建议：

- 常量一般用全大写，如 `const PI = 3.141592` ；
- 使用易读的命名，比如 `userName` 或者 `shoppingCart` 。

4. 注意点

- JavaScript 变量名称区分大小写，如变量 `leo` 与 `Leo` 是不同的；
- JavaScript 变量名称允许非英文字母，但不推荐，如 `let 平安 = "leo"` ；
- 避免使用 `a` 、 `b` 、 `c` 这种缩写。

五、数据类型

JavaScript 是一种**弱类型**或者说**动态语言**。这意味着你不用提前声明变量的类型，在程序运行过程中，类型会被自动确定。这也意味着你可以使用同一个变量保存不同类型的数据：

```
var foo = 42;    // foo is a Number now
foo = "bar"; // foo is a String now
foo = true;  // foo is a Boolean now
```

详细介绍可以阅读《MDN JavaScript 数据类型和数据结构 》。

1. 八大数据类型

前七种为基本数据类型，也称为原始类型（值本身无法被改变），而 `object` 为复杂数据类型。 八大数据类型分别是：

- `number` 用于任何类型的数字：整数或浮点数，在 ±2 范围内的整数。
- `bigint` 用于任意长度的整数。
- `string` 用于字符串：一个字符串可以包含一个或多个字符，所以没有单独的单字符类型。
- `boolean` 用于 `true` 和 `false` 。
- `null` 用于未知的值 —— 只有一个 `null` 值的独立类型。
- `undefined` 用于未定义的值 —— 只有一个 `undefined` 值的独立类型。
- `symbol` 用于唯一的标识符。
- `object` 用于更复杂的数据结构。 **每个类型后面会详细介绍。**

2. 检测数据类型

通过 `typeof` 运算符检查：

- 两种形式：`typeof x` 或者 `typeof(x)`。
- 以字符串的形式返回类型名称，例如 `"string"`。
- `typeof null` 会返回 `"object"` —— 这是 JavaScript 编程语言的一个错误，实际上它并不是一个 `object`。

```
typeof "leo" // "string"
typeof undefined    // "undefined"
typeof 0     // "number"
typeof NaN   // "number"
typeof 10n   // "bigint"
typeof true  // "boolean"
typeof Symbol("id") // "symbol"
typeof [1,2,3,4]    // "object"
typeof Math  // "object"  (1) Math 是一个提供数学运算的内建 object。
typeof null  // "object"  (2) JavaScript 语言的一个错误，null 不是一个 object。null 有自己的类型，它是一个特
typeof alert // "function"  (3) alert 在 JavaScript 语言中是一个函数。
```

六、类型转换

JavaScript 变量可以转换为新变量或其他数据类型：

- 通过使用 JavaScript 函数
- 通过 JavaScript 自身自动转换

1.字符串转换

通过全局方法 `String()` 将**其他类型数据（任何类型的数字，字母，布尔值，对象）**转换为 String 类型：

```
String(123);   // "123"
// Number方法toString()/toExponential()/toFixed()/toPrecision() 也有同样效果。
String(false); // "false"
// Boolean方法 toString() 也有同样效果。
String(new Date()); // "Sun Jun 07 2020 21:44:20 GMT+0800 (中国标准时间)"
// Date方法 toString() 也有同样效果。
String(leo);
```

2. 数值转换

通过以下几种方式能将其他类型数据转换为 Number 类型：

- 一元运算符 `+`

```
const age = +"22"; // 22
```

- `Number` 方法

```
const age = Number("22"); // 22
Number.parseFloat("22");  // 22
Number.parseInt("22");  // 22
```

- 其他方式转 Number 类型

```
// 布尔值
Number(false)     // 返回 0
Number(true)      // 返回 1
// 日期
const date = new Date();
Number(date);     // 返回 1591537858154
date.getTime();   // 返回 1591537858154，效果一致。
// 自动转换
5 + null    // 返回 5          null 转换为 0
"5" + null  // 返回"5null"   null 转换为 "null"
"5" + 1     // 返回 "51"       1 转换为 "1"
"5" - 1     // 返回 4          "5" 转换为 5
```

3. 布尔值转换

转换规则如下：

- 直观上为“空”的值（如 `0`、空字符串、`null`、`undefined` 和 `NaN`）将变为 `false`。
- 其他值变成 `true`。

```
Boolean(1); // true
Boolean(0); // false
Boolean("hello"); // true
Boolean(""); // false
```

```
Boolean("0"); // true
Boolean(" "); // 空白, 也是 true (任何非空字符串是 true)
```

4.小结

原始值	转换为数字	转换为字符串	转换为布尔值	实例
false	0	"false"	false	尝试一下 »
true	1	"true"	true	尝试一下 »
0	0	"0"	false	尝试一下 »
1	1	"1"	true	尝试一下 »
"0"	0	"0"	true	尝试一下 »
"000"	0	"000"	true	尝试一下 »
"1"	1	"1"	true	尝试一下 »
NaN	NaN	"NaN"	false	尝试一下 »
Infinity	Infinity	"Infinity"	true	尝试一下 »
-Infinity	-Infinity	"-Infinity"	true	尝试一下 »
""	0	""	false	尝试一下 »
"20"	20	"20"	true	尝试一下 »
"Runoob"	NaN	"Runoob"	true	尝试一下 »
[ ]	0	""	true	尝试一下 »
[20]	20	"20"	true	尝试一下 »
[10,20]	NaN	"10,20"	true	尝试一下 »
["Runoob"]	NaN	"Runoob"	true	尝试一下 »
["Runoob","Google"]	NaN	"Runoob,Google"	true	尝试一下 »

function(){}	NaN	"function(){}"	true	尝试一下 »
{ }	NaN	"[object Object]"	true	尝试一下 »
null	0	"null"	false	尝试一下 »
undefined	NaN	"undefined"	false	尝试一下 »

七、运算符

1、运算符概念

常见运算符如加法 `+` 、减法 `-` 、乘法 `*` 和除法 `/` ，举一个例子，来介绍一些概念：

```
let sum = 1 + 2;
let age = +18;
```

其中：

- 加法运算 `1 + 2` 中， `1` 和 `2` 为 2 个运算元，左运算元 `1` 和右运算元 `2` ，即**运算元就是运算符作用的对象。**
- `1 + 2` 运算式中包含 2 个运算元，因此也称该运算式中的加号 `+` 为 **二元运算符。**
- 在 `+18` 中的加号 `+` 对应只有一个运算元，则它是 **一元运算符** 。

2、+ 号运算符

```
let msg = "hello " + "leo"; // "hello leo"
let total = 10 + 20;  // 30
let text1 = "1" + "2"; // "12"
let text2 = "1" + 2;   // "12"
let text3 = 1 + "2";   // "12"
let text4 = 1 + 2 + "3";  // "33"
let num = +text1; //  12 转换为 Number 类型
```

3、运算符优先级

运算符的优先级决定了表达式中运算执行的先后顺序，优先级高的运算符最先被执行。 下面的表将所有运算符按照优先级的不同从高（20）到低（1）排列。

优先级	运算类型	关联性	运算符
20	圆括号	n/a（不相关）	`( … )`
19	成员访问	从左到右	`… . …`
	需计算的成员访问	从左到右	`… [ … ]`
	`new` (带参数列表)	n/a	`new … ( … )`
	函数调用	从左到右	`… ( … )`
	可选链（Optional chaining）	从左到右	`?.`
18	new (无参数列表)	从右到左	`new …`
17	后置递增(运算符在后)	n/a	
`… ++`			
	后置递减(运算符在后)		`… --`
16	逻辑非	从右到左	`! …`
	按位非		`~ …`
	一元加法		`+ …`
	一元减法		`- …`
	前置递增		`++ …`
	前置递减		`-- …`
优先级	运算类型	关联性	运算符
	typeof		typeof

	typeof		typeof …
	void		void …
	delete		delete …
	await		await …
15	幂	从右到左	… ** …
14	乘法	从左到右	
… * …			
	除法		… / …
	取模		… % …
13	加法	从左到右	
… + …			
	减法		… - …
12	按位左移	从左到右	… << …
	按位右移		… >> …
	无符号右移		… >>> …
11	小于	从左到右	… < …
	小于等于		… <= …
	大于		… > …
	大于等于		… >= …
	in		… in …
优先级	**运算类型** instanceof	**关联性**	**运算符** … instanceof …

10	等号	从左到右	
… == …			
	非等号		… != …
	全等号		… === …
	非全等号		… !== …
9	按位与	从左到右	… & …
8	按位异或	从左到右	… ^ …
7	按位或	从左到右	… | …
6	逻辑与	从左到右	… && …
5	逻辑或	从左到右	… || …
4	条件运算符	从右到左	… ? … : …
3	赋值	从右到左	… = …
			… += …
			… -= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
优先级	**运算类型**	**关联性**	**运算符**

			`… &= …`
			`… ^= …`
			`… |= …`
2	yield	从右到左	`yield …`
	yield*		`yield* …`
1	展开运算符	n/a	`...` …
0	逗号	从左到右	`… , …`

```
3 > 2 && 2 > 1
// return true
3 > 2 > 1
// 返回 false，因为 3 > 2 是 true，并且 true > 1 is false
// 加括号可以更清楚：(3 > 2) > 1
```

八、值的比较

1. 常见比较

在 JS 中的值的比较与数学很类型：

- 大于/小于/大于等于/小于等于： `a>b` / `a<b` / `a>=b` / `a<=b` ；
- 判断相等：

```
// 使用 ==，非严格等于，不关心值类型
// == 运算符会对比较的操作数做隐式类型转换，再比较
'1' == 1; // true
// 使用 ===，严格相等，关心值类型
// 将数字值 -0 和 +0 视为相等，并认为 Number.NaN 不等于 NaN。
'1' === 1; // false
```

x	y	==	===	Object.is
undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
"foo"	"foo"	true	true	true
0	0	true	true	true
+0	-0	true	true	false
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
"0"	0	true	false	false
"17"	17	true	false	false
[1,2]	"1,2"	true	false	false
new String("foo")	"foo"	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: "bar" }	{ foo: "bar" }	false	false	false
new String("foo")	new String("foo")	false	false	false
0	null	false	false	false
0	NaN	false	false	false
"foo"	NaN	false	false	false
NaN	NaN	false	false	true

（图片来自：《MDN JavaScript 中的相等性判断》）

- 判断不相等： 和判断相等一样，也有两种： `!=` / `!==` 。

2. 相等性判断（Object.is()）

另外 ES6 新增 Object.is 方法判断两个值是否相同，语法如下：

```
Object.is(value1, value2);
```

以下任意项成立则两个值相同：

- 两个值都是 `undefined`
- 两个值都是 `null`
- 两个值都是 `true` 或者都是 `false`
- 两个值是由相同个数的字符按照相同的顺序组成的字符串
- 两个值指向同一个对象
- 两个值都是数字并且
  - 都是正零 `+0`
  - 都是负零 `-0`
  - 都是 `NaN`
  - 都是除零和 `NaN` 外的其它同一个数字 使用示例：

```
Object.is('foo', 'foo');     // true
Object.is(window, window);   // true
Object.is('foo', 'bar');     // false
Object.is([], []);           // false
var foo = { a: 1 };
var bar = { a: 1 };
Object.is(foo, foo);         // true
Object.is(foo, bar);         // false
Object.is(null, null);       // true
// 特例
Object.is(0, -0);            // false
Object.is(0, +0);            // true
Object.is(-0, -0);           // true
Object.is(NaN, 0/0);         // true
```

兼容性 Polyfill 处理：

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
```

```
    }
  };
}
```

3. null 与 undefined 比较

对于相等性判断比较简单：

```
null == undefined;  // true
null === undefined; // false
```

对于其他比较，它们会先转换位数字： `null` 转换为 `0` ， `undefied` 转换为 `NaN` 。

```
null > 0;  // false 1
null >= 0; // true  2
null == 0; // false 3
null < 1;  // true  4
```

需要注意： `null == 0; // false` 这里是因为： `undefined` 和 `null` 在相等性检查 `==` 中**不会进行任何的类型转换**，它们有自己独立的比较规则，所以除了它们之间互等外，不会等于任何其他的值。

```
undefined > 0;  // false  1
undefined > 1;  // false  2
undefined == 0; // false  3
```

第 1、2 行都返回 `false` 是因为 `undefined` 在比较中被转换为了 `NaN` ，而 `NaN` 是一个特殊的数值型值，它与任何值进行比较都会返回 `false` 。 第 3 行返回 `false` 是因为这是一个相等性检查，而 `undefined` 只与 `null` 相等，不会与其他值相等。

九、alert / prompt / confirm

1. alert

显示一个警告对话框，上面显示有指定的文本内容以及一个“确定”按钮。 **注意：弹出模态框，并暂停脚本，直到用户点击“确定”按钮。**

```
// 语法
window.alert(message);
alert(message);
// 示例
alert('hello leo!');
```

`message`是要显示在对话框中的文本字符串，如果传入其他类型的值,会转换成字符串。

2. prompt

显示一个对话框，对话框中包含一条文字信息，用来提示用户输入文字。 **注意：弹出模态框，并暂停脚本，直到用户点击“确定”按钮。** 当点击确定返回文本，点击取消或按下 Esc 键返回 `null`。 语法如下：

```
let result = window.prompt(text, value);
```

- `result` 用来存储用户输入文字的字符串，或者是 null。
- `text` 用来提示用户输入文字的字符串，如果没有任何提示内容，该参数可以省略不写。
- `value` 文本输入框中的默认值，该参数也可以省略不写。不过在 Internet Explorer 7 和 8 中，省略该参数会导致输入框中显示默认值"undefined"。

3. confirm

`Window.confirm()` 方法显示一个具有一个可选消息和两个按钮(确定和取消)的模态对话框。 **注意：弹出模态框，并暂停脚本，直到用户点击“确定”按钮。** 语法如下：

```
let result = window.confirm(message);
```

- message 是要在对话框中显示的可选字符串。
- result 是一个布尔值，表示是选择确定还是取消 (true表示OK)。

十、条件运算符：if 和 '?'

1. if 语句

当 if 语句当条件表达式，会将表达式转换为布尔值，当为 truthy 时执行里面代码。 转换规则如：

- 数字 `0`、空字符串 `""`、`null`、`undefined` 和 `NaN` 都会被转换成 `false`。因为他们被称为 “falsy” 值。
- 其他值被转换为 `true`，所以它们被称为 “truthy”。

2. 三元运算符

条件（三元）运算符是 JavaScript 仅有的使用三个操作数的运算符。一个条件后面会跟一个问号（?），如果条件为 truthy，则问号后面的表达式A将会执行；表达式A后面跟着一个冒号（:），如果条件为 falsy，则冒号后面的表达式B将会执行。本运算符经常作为 `[if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else)` 语句的简捷形式来使用。 语法：

```
condition ? exprIfTrue : exprIfFalse
```

- condition 计算结果用作条件的表达式。
- exprIfTrue 如果表达式 condition 的计算结果是 truthy（它和 true 相等或者可以转换成 true ），那么表达式 exprIfTrue 将会被求值。
- exprIfFalse 如果表达式 condition 的计算结果是 falsy（它可以转换成 false ），那么表达式 exprIfFalse 将会被执行。 示例：

```
let getUser = function(name){
        return name === 'leo' ? 'hello leo!' : 'unknow user';
}
// 可以简写如下：
let getUser = name => name === 'leo' ? 'hello leo!' : 'unknow user';
getUser('leo'); // "hello leo!"
getUser('pingan'); // "unknow user"
```

十一、逻辑运算符

详细可以阅读《MDN 逻辑运算符》。

1. 运算符介绍

逻辑运算符如下表所示 (其中 `_expr_` 可能是任何一种类型, 不一定是布尔值):

运算符	语法	说明
逻辑与，AND（`&&`）	`_expr1_ && _expr2_`	若 `expr**1**` 可转换为 `true`，则返回 `expr**2**`；否则，返回 `expr**1**`。
逻辑或，OR（`||`）	`_expr1_ || _expr2_`	若 `expr**1**` 可转换为 `true`，则返回 `expr**1**`；否则，返回 `expr**2**`。
逻辑非，NOT（`!`）	`!_expr_`	若 `expr` 可转换为 `true`，则返回 `false`；否则，返回 `true`。

如果一个值可以被转换为 `true`，那么这个值就是所谓的 truthy，如果可以被转换为 `false`，那么这个值就是所谓的 falsy。 会被转换为 `false` 的表达式有：

- `null`；
- `NaN`；
- `0`；
- 空字符串（`""` or `''` or ```` ``` ````）；
- `undefined`。 尽管 `&&` 和 `||` 运算符能够使用非布尔值的操作数, 但它们依然可以被看作是布尔操作符，因为它们的返回值总是能够被转换为布尔值。如果要显式地将它们的返回值（或者表达式）转换为布尔值，请使用双重非运算符（即 `!!`）或者Boolean构造函数。 JavaScript 里有三个逻辑运算符：`||`（或），`&&`（与），`!`（非）。

2. 运算符示例

- 逻辑与（&&） 所有条件都为 true 才返回 true，否则为 false。

```
a1 = true  && true      // t && t 返回 true
a2 = true  && false     // t && f 返回 false
a3 = false && true      // f && t 返回 false
a4 = false && (3 == 4)  // f && f 返回 false
a5 = "Cat" && "Dog"     // t && t 返回 "Dog"
a6 = false && "Cat"     // f && t 返回 false
a7 = "Cat" && false     // t && f 返回 false
a8 = ''    && false     // f && f 返回 ""
a9 = false && ''        // f && f 返回 false
```

- 逻辑或（ || ） 所有条件有一个为 true 则返回 true，否则为 false。

```
o1 = true  || true      // t || t 返回 true
o2 = false || true      // f || t 返回 true
o3 = true  || false     // t || f 返回 true
o4 = false || (3 == 4)  // f || f 返回 false
o5 = "Cat" || "Dog"     // t || t 返回 "Cat"
o6 = false || "Cat"     // f || t 返回 "Cat"
o7 = "Cat" || false     // t || f 返回 "Cat"
o8 = ''    || false     // f || f 返回 false
o9 = false || ''        // f || f 返回 ""
```

- 逻辑非（!）

```
n1 = !true              // !t 返回 false
n2 = !false             // !f 返回 true
n3 = !''                // !f 返回 true
n4 = !'Cat'             // !t 返回 false
```

- 双重非运（!!）

```
n1 = !!true                  // !!truthy 返回 true
n2 = !!{}                    // !!truthy 返回 true: 任何 对象都是 truthy 的…
n3 = !!(new Boolean(false))  // …甚至 .valueOf() 返回 false 的布尔值对象也是!
n4 = !!false                 // !!falsy 返回 false
n5 = !!""                    // !!falsy 返回 false
n6 = !!Boolean(false)        // !!falsy 返回 false
```

3. 布尔值转换规则

- 将 && 转换为 ||

```
condi1 && confi2
// 转换为
!(!condi1 || !condi2)
```

- 将 || 转换为 &&

```
condi1 || condi2
// 转换为
!(!condi1 && !condi2)
```

4. 短路取值

由于逻辑表达式的运算顺序是从左到右，也可以用以下规则进行"短路"计算：

- `(some falsy expression) && (_expr)_` 短路计算的结果为假。
- `(some truthy expression) || _(expr)_` 短路计算的结果为真。 短路意味着上述表达式中的expr部分**不会被执行**，因此expr的任何副作用都不会生效（举个例子，如果expr是一次函数调用，这次调用就不会发生）。造成这种现象的原因是，整个表达式的值在第一个操作数被计算后已经确定了。看一个例子：

```
function A(){ console.log('called A'); return false; }
function B(){ console.log('called B'); return true; }
console.log( A() && B() );
// logs "called A" due to the function call,
// then logs false (which is the resulting value of the operator)
console.log( B() || A() );
// logs "called B" due to the function call,
// then logs true (which is the resulting value of the operator)
```

5. 注意

与运算 `&&` 的优先级比或运算 `||` 要高。 所以代码 `a && b || c && d` 完全跟 `&&` 表达式加了括号一样：`(a && b) || (c && d)` 。

十二、循环：while 和 for

1. while 循环

详细可以阅读《MDN while》 。 **while 语句**可以在某个条件表达式为真的前提下，循环执行指定的一段代码，直到那个表达式不为真时结束循环。 如：

```
var n = 0;
var x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

当循环体为单行时，可以不写大括号：

```
let i = 3;
while(i) console.log(i --);
```

2. do...while 循环

详细可以阅读《MDN do...while》。 `do...while` 语句创建一个执行指定语句的循环，直到 `condition` 值为 false。在执行 `statement` 后检测 `condition`，所以指定的 `statement` 至少执行一次。 如：

```
var result = '';
var i = 0;
do {
   i += 1;
   result += i + ' ';
} while (i < 5);
```

3. for 循环

详细可以阅读《MDN for》。 `for` 语句用于创建一个循环，它包含了三个可选的表达式，这三个表达式被包围在圆括号之中，使用分号分隔，后跟一个用于在循环中执行的语句（通常是一个块语句）。 语法如：

```
for (begin; condition; step) {
  // ……循环体……
}
```

示例：

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}
```

描述：

begin	**`i = 0`**	**进入循环时执行一次。**
condition	`i < 3`	在每次循环迭代之前检查，如果为 false，停止循环。
body（循环体）	`alert(i)`	条件为真时，重复运行。
step	`i++`	在每次循环体迭代后执行。

4. 可选的 for 表达式

`for` 语句头部圆括号中的所有三个表达式都是可选的。

- 不指定表达式中初始化块

```
var i = 0;
for (; i < 3; i++) {
    console.log(i);
}
```

- 不指定表达式中条件块，这就必须要求在循环体中结束循环，否则会出现死循环

```
for (var i = 0;; i++) {
   console.log(i);
   if (i > 3) break;
}
```

- 不指定所有表达式，也需要在循环体中指定结束循环的条件

```
var i = 0;
for (;;) {
  if (i > 3) break;
  console.log(i);
  i++;
}
```

5. break 语句

详细可以阅读《MDN break》。break 语句中止当前循环，`switch` 语句或 `label` 语句，并把程序控制流转到紧接着被中止语句后面的语句。在 while 语句中：

```
function testBreak(x) {
  var i = 0;
  while (i < 6) {
    if (i == 3) {
      break;
    }
    i += 1;
  }
  return i * x;
}
```

另外，也可以为代码块做标记，并在 break 中指定要跳过的代码块语句的 label：

```
outer_block:{
  inner_block:{
    console.log ('1');
    break outer_block;      // breaks out of both inner_block and outer_block
    console.log (':-(');    // skipped
  }
  console.log ('2');        // skipped
}
```

需要注意的是：break 语句需要内嵌在它所应用的标签或代码块中，否则报错：

```
block_1:{
  console.log ('1');
  break block_2;            // SyntaxError: label not found
}
block_2:{
  console.log ('2');
}
```

6. continue 语句

continue 声明终止当前循环或标记循环的当前迭代中的语句执行，并在下一次迭代时继续执行循环。与 `break` 语句的区别在于，continue 并不会终止循环的迭代，而是：

- 在 `while` 循环中，控制流跳转回条件判断；
- 在 `for` 循环中，控制流跳转到更新语句。 注意：continue 也必须在对应循环内部，否则报错。

```
i = 0;
n = 0;
while (i < 5) {
   i++;
   if (i === 3) {
      continue;
   }
   n += i;
}
```

带 label：

```
var i = 0,
    j = 8;
checkiandj: while (i < 4) {
   console.log("i: " + i);
   i += 1;
   checkj: while (j > 4) {
      console.log("j: "+ j);
      j -= 1;
      if ((j % 2) == 0)
         continue checkj;
      console.log(j + " is odd.");
   }
   console.log("i = " + i);
   console.log("j = " + j);
}
```

7. 注意

禁止 `break/continue` 在 '?' 的右边：

```
(i > 5) ? console.log(i) : continue; // continue 不允许在这个位置
```

这样会提示语法错误。 请注意非表达式的语法结构不能与三元运算符 `?` 一起使用。特别是 `break/continue` 这样的指令是不允许这样使用的。

8. 总结

三种循环：

- `while` —— 每次迭代之前都要检查条件。
- `do..while` —— 每次迭代后都要检查条件。
- `for (;;)` —— 每次迭代之前都要检查条件，可以使用其他设置。 通常使用 `while(true)` 来构造“无限”循环。这样的循环和其他循环一样，都可以通过 `break` 指令来终止。 如果我们不想在当前迭代中做任何事，并且想要转移至下一次迭代，那么可以使用 `continue` 指令。 `break/continue` 支持循环前的标签。标签是 `break/continue` 跳出嵌套循环以转到外部的唯一方法。

十三、"switch" 语句

`switch` 语句用来将表达式的值与 case 语句匹配，并执行与情况对应的语句。 `switch` 语句可以替代多个 if 判断，为多个分支选择的情况提供一个更具描述性的方式。

1. 语法

`switch` 语句至少包含一个 `case` 代码块和一个可选的 `default` 代码块：

```
switch(expression) {
  case 'value1':
    // do something ...
    [break]

  default:
    // ...
    [break]
}
```

当 `expression` 表达式的值与 `value1` 匹配时，则执行其中代码块。 如果没有 `case` 子句匹配，则会选择 `default` 子句执行，若连 `default` 子句都没有，则直接执行到 `switch` 结束。

2. 使用 case 分组

所谓 case 分组，就是与多个 case 分支共享同一段代码，如下面例子中 `case 1` 和 `case 2`：

```
let a = 2;
switch (a) {
  case 1: // (*) 下面这两个 case 被分在一组
  case 2:
    console.log('case is 1 or 2!');
    break;
  case 3:
    console.log('case is 3!');
    break;
  default:
    console.log('The result is default.');
}
// 'case is 1 or 2!'
```

3.注意点

1. `expression` 表达式的值与 `case` 值的比较是严格相等：

```
function f(n){
    let a ;
    switch(n){
        case 1:
            a = 'number';
            break;
        case '1':
            a = 'string';
            break;
        default:
            a = 'default';
            break;
    }
    console.log(a)
}
f(1);   // number
f('1'); // string
```

2. **如果没有 `break`，程序将不经过任何检查就会继续执行下一个 ** `**case**` ：

```
let a = 2 + 2;
switch (a) {
        case 3:
    console.log( 'Too small' );
  case 4:
```

```
    console.log( 'Exactly!' );
  case 5:
    console.log( 'Too big' );
  default:
    console.log( "I don't know such values" );
}
// Exactly!
// Too big
// I don't know such values
```

3. **`**default**`** **放在 ****`**case**`** **之上不影响匹配：**

```
function f(n){
  switch (n) {
    case 2:
      console.log(2);
      break;
    default:
      console.log('default')
      break;
    case 1:
      console.log('1');
      break;
  }
}
f(1); // 1
f(2); // 2
f(3); // default
```

- **`switch`** **语句中存在** **`let`** **/** **`const`** **重复声明问题：**

```
// 以下定义会报错
function f(n){
    switch(n){
        case 1:
            let msg = 'hello';
            console.log(1);
            break;
        case 2:
            let msg = 'leo';
            break;
        default:
            console.log('default');
            break;
    }
```

```
}
// Error: Uncaught SyntaxError: Identifier 'msg' has already been declared
```

这是由于两个 `let` 处于同一个块级作用域，所以它们被认为是同一变量名的重复声明。 解决方式，只需要将 `case` 语句包装在括号内即可解决：

```
function f(n){
    switch(n){
        case 1:{ // added brackets
            let msg = 'hello';
            console.log(msg);
            break;
        }
        case 2: {
            let msg = 'leo';
            console.log(msg);
            break;
        }
        default:
            console.log('default');
            break;
    }
}
```

十四、函数

函数可以让一段代码被多次调用，避免重复代码。 如之前学习到的一些内置函数： `alert(msg)` / `prompt(msg, default)` / `confirm(quesyion)` 等。

1. 函数定义

定义函数有两种方式：**函数声明**和**函数表达式**。

1.1 函数声明

如定义一个简单 `getUser` 函数：

```
function getUser(name){
        return 'hello ' + name;
```

```
}
getUser('leo"); // 函数调用
```

通过函数声明来定义函数时，需要由以下几部分组成：

- 函数名称 - `getUser` ；
- 函数参数列表 - `name` ；
- 函数的 JS 执行语句 - `return 'hello ' + name` 。

1.2 函数表达式

类似声明变量，还是以 `getUser` 为例：

```
let getUser = function(name){
        return 'hello ' + name;
}
```

另外，函数表达式也可以提供函数名，并用于函数内部指代函数本身：

```
let fun = function f(n){
    return n < 3 ? 1 : n * f(n - 1);
}
fun(3);  // 3
fun(5);  // 60
```

2. 函数调用

当定义一个函数后，它并不会自动执行，而是需要使用函数名称进行调用，如上面例子：

```
fun(3);  // 3
```

只要注意： 使用 **函数表达式** 定义函数时，调用函数的方法必须写在定义之后，否则报错：

```
console.log(fun());  // Uncaught ReferenceError: fun is not defined
let fun = function(){return 1};
```

而使用 **函数声明** 则不会出现该问题：

```
console.log(fun());  // 1
function fun(){return 1};
```

原因就是：函数提升仅适用于函数声明，而不适用于函数表达式。

3. 函数中的变量

在函数中，可以使用局部变量和外部变量。

3.1 局部变量

函数中声明的变量只能在该函数内可见。

```
let fun = function(){
        let name = 'leo';
}
fun();
console.log(name); // Uncaught ReferenceError: name is not defined
```

3.2 全局变量

函数内可以使用外部变量，并且可以修改外部变量的值。

```
let name = 'leo';
let fun = function(){
        let text = 'Hello, ' + name;
  console.log(text);
}
fun(); // Hello, leo
```

当函数内也有与外部变量名称相同的变量，会忽略外部变量：

```
let name = 'leo';
let fun = function(){
  let name = 'pingan8787';
```

```
        let text = 'Hello, ' + name;
  console.log(text);
}
fun(); // Hello, pingan8787
```

4. 函数参数

从ECMAScript 6开始，有两个新的类型的参数：默认参数，剩余参数。

4.1 默认参数

若函数没有传入参数，则参数默认值为`undefined`，通常设置参数默认值是这样做的：

```
// ES6 之前，没有设置默认值
function f(a, b){
    b = b ? b : 1;
    return a * b;
}
f(2,3);  // 6
f(2);    // 2
// ES6，设置默认值
function f(a, b = 1){
    return a * b;
}
f(2,3);  // 6
f(2);    // 2
```

4.2 剩余参数

可以将参数中不确定数量的参数表示成数组，如下：

```
function f (a, ...b){
    console.log(a, b);
}
f(1,2,3,4); // a => 1 b => [2, 3, 4]
```

既然讲到参数，那就不能少了 arguments 对象。

4.3 arguments 对象

函数的实际参数会被保存在一个**类似数组的arguments对象**中。在函数内，我们可以使用 arguments 对象获取函数的所有参数：

```
let fun = function(){
    console.log(arguments);
    console.log(arguments.length);
}
fun('leo');
// Arguments ["leo", callee: ƒ, Symbol(Symbol.iterator): ƒ]
// 1
```

以一个实际示例介绍，实现将任意数量参数连接成一个字符串，并输出的函数：

```
let argumentConcat = function(separator){
        let result = '', i;
  for(i = 1; i < arguments.length; i ++){
        result += arguments[i] + separator;
  }
  return result;
}
argumentConcat(',', 'leo', 'pingan'); //"leo,pingan,"
```

5. 函数返回值

在函数任意位置，指定 `return` 指令来停止函数的执行，并返回函数指定的返回值。

```
let sum = function(a, b){
        return a + b;
};
let res = sum(1, 2);
console.log(res); // 3
```

默认空值的 `return` 或没有 `return` 的函数返回值为 `undefined` 。

十五、函数表达式

函数表达式是一种函数定义方式，在前面章节中已经介绍到了，这个部分将着重介绍 **函数表达式** 和 **函数声明** 的区别：

1.语法差异

```
// 函数表达式
let fun = function(){};
// 函数声明
function fun(){}
```

2.创建时机差异

函数表达式会在代码执行到达时被创建，并且仅从那一刻可用。 而函数声明被定义之前，它就可以被调用。

```
// 函数表达式
fun();  // Uncaught ReferenceError: fun is not defined
let fun = function(){console.log('leo')};
// 函数声明
fun();  // "leo"
function fun(){console.log('leo')};
```

3.使用建议

建议优先考虑函数声明语法，它能够为组织代码提供更多灵活性，因为我们可以在声明函数前调用该函数。

十六、箭头函数

本章节简单介绍箭头函数基础知识，后面章节会完整介绍。 函数箭头表达式是ES6新增的函数表达式的语法，也叫**胖箭头函数**，变化：更简洁的函数和 `this` 。

1.代码更简洁

```
// 有1个参数
let f = v => v;
// 等同于
let f = function (v){return v};
// 有多个参数
let f = (v, i) => {return v + i};
// 等同于
```

```
let f = function (v, i){return v + i};
// 没参数
let f = () => 1;
// 等同于
let f = function (){return 1};
let arr = [1,2,3,4];
arr.map(ele => ele + 1);  // [2, 3, 4, 5]
```

2.注意点

1. 箭头函数不存在`this`；
2. 箭头函数不能当做**构造函数**，即不能用`new`实例化；
3. 箭头函数不存在`arguments`对象，即不能使用，可以使用`rest`参数代替；
4. 箭头函数不能使用`yield`命令，即不能用作Generator函数。 一个简单的例子：

```
function Person(){
  this.age = 0;
  setInterval(() => {
    this.age++;
  }, 1000);
}
var p = new Person(); // 定时器一直在执行 p的值一直变化
```