JavaScript指导

简介

JavaScript简介

什么是JavaScript

JavaScript 最初的目的是为了**“让网页动来”**。 这种编程语言我们称之为脚本。它们可以写在 HTML 中，在页面加载的时候会自动执行。 脚本作为纯文本存在和执行。它们不需要特殊的准备或编译即可运行。 现在，JavaScript 不仅仅是在浏览器内执行，也可以在服务端执行。甚至在任意搭载了 JavaScript 引擎 的环境中都可以执行。

浏览器中的JavaScript能做什么

浏览器中的 JavaScript 可以完成所有和网页相关的操作、处理用户和 Web 服务器之间的交互。

- 在网页中插入新的 HTML，修改现有的网页内容和网页的样式。
- 响应用户的行为，响应鼠标的点击或移动、键盘的敲击。
- 向远程服务器发送请求，使用 AJAX 和 COMET 技术下载或上传文件。
- 获取或修改 cookie，向用访问者提出问题、发送消息。
- 记住客户端的数据（本地存储）。

浏览器中的JavaScript不能做什么

为了用户的（信息）安全，在浏览器中的 JavaScript 的能力是有限的。这样主要是为了阻止邪恶的网站获得或修改用户的私人数据。

- 网页中的 JavaScript 不能读、写、复制及执行用户磁盘上的文件或程序。它没有直接访问操作系统的功能。
- 不同的浏览器标签页之间基本彼此不相关。
    - 同源策略：为了解决不同标签页交互的问题，两个同源的网站必须都包含一些特殊的 JavaScript 代码，才能够实现数据交换。
- JavaScript 通过互联网可以很容易地和服务器（当前网页域名的服务器）通讯。但是从其他的服务器中获取数据的功能是受限的，需要服务器（在 HTTP 头中）添加某些参数。 *注意：* 浏览器环境外的 JavaScript 一般没有这些限制。例如服务端的 JavaScript 就没有这些限制。现代浏览器还允许通过 JavaScript 来安装浏览器插件或扩展，当然这也是在用户授权的前提下。

编辑器

编辑器主要分两种：IDE（集成开发环境）和轻量编辑器。

IDE

IDE（集成开发环境）是用于管理整个项目具有强大功能的编辑器。顾名思义，它不仅仅是一个编辑器，而且还是开发环境。

- WebStorm
- Visual Studio
- Netbeans

轻量编辑器

“轻量编辑器”没有 IDE 那么功能强大，但是他们一般很快、优雅而且简单，主要用于立即打开编辑一个文件。

- Visual Studio Code（跨平台、免费）。
- Atom（跨平台、免费）。
- Sublime Text（跨平台、共享软件）。
- Notepad++（Windows、免费）。
- Vim 和 Emacs 很棒，前提是如果你知道怎么用。

开发者控制台

但在浏览器中，默认情况下用户是看不到错误的。所以，如果脚本中有错误，我们看不到是什么错误，更不能够修复。为了发现错误并获得一些与脚本相关且有用的信息，浏览器内置了 “开发者工具”。

- Google Chrome
- Firefox
- Safari

JavaScript 基础知识

Hello, world!

script 标签

JavaScript 程序可以使用 `<script>` 标签插入到 HTML 的任何地方。

```
<!DOCTYPE HTML>
<html>
<body>
  <p>script 标签之前...</p>
*!*
  <script>
    alert( 'Hello, world!' );
  </script>
*/!*
```

```
  <p>...script 标签之后</p>
</body>
</html>
```

`<script>` 标签中包裹了 JavaScript 代码，当浏览器遇到 `<script>` 标签，代码会自动运行。

现代的标记

`<script>` 标签有一些现在很少用到的属性，但是我们可以在老代码中找到它们：

- type 属性: `<script type=...>`: 在老的 HTML4 标准中，`<script>` 标签有type属性。通常是 `type="text/javascript"` 。现在的 HTML 标准已经默认存在该 type 属性。该属性不是必须的。
- language 属性: `<script language=...>` : 这个属性是为了显示脚本使用的语言。就目前而言，这个属性没有任何意义，语言默认为 JavaScript。不再需要使用它了。

外部脚本

脚本文件可以通过 src 属性添加到 HTML 文件中。

```
<script src="/path/to/script.js"></script> # 绝对路径
<script src="script.js"></script> # 相对路径
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script> # 完整url地址
```

代码结构

语句

语句是执行操作的语法结构和命令。我们可以在代码中编写任意数量的语句。语句之间可以使用分号分割。

```
alert('Hello');
alert('World');
```

分号

多数情况下，当一个分行符（line break）存在时，分号可以省略。

```
alert('Hello')
alert('World')
```

此处，JavaScript 将分行符解释成“隐式”的分号。这也被称为自动分号插入。 **多数情况下，换行意味着一个分号。但是“多数情况”并不意味着“总是”！** 有很多换行并不是分号的例子，比如：

```
alert(3 +
1
```

```
+ 2); // 6
```

显而易见的是，如果一行以加号 "+" 结尾，那么这是一个“不完整的表达式”，不需要分号。**但存在 JavaScript 无法假设分号是否真正被需要的情况。**

```
alert("There will be an error")
[1, 2].forEach(alert)

# 如果我们运行代码，仅仅第一个 `alert` 显示了文本，接着我们收到了一个错误！但是，如果我们在第一个 `alert` 后加入一
```

注意： 不要使用 `(`，`[`，或者 ``` 等作为一行的开始。

注释

- 单行注释以两个正斜杠字符 // 开始。
- 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* "*/"。

use strict

新特性已添加到该语言，但是旧的功能也没有改变。这有利于不破坏现有的规范，但缺点是 JavaScript 创造者的任何错误和不完美的考虑也永远地停留在语言中。

直到 2009 年 ECMAScript 5 (ES5) 的出现。ES5 规范增加了新的语言特性并且修改了一些已经存在的特性。为了保证旧的功能能够使用，大部分的修改是默认不生效的。你需要一个特殊的指令 —— "use strict" 来明确地使用这些特性。

这个指令看上去是一个字符串 `"use strict"` 或者 `'use strict'`。当它处于脚本文件的顶部，则整个脚本文件都工作在“现代”的方式中。

变量

一个 `变量` 是数据的“命名存储”。我们可以使用变量来保存商品、访客和其他信息。

- let: 新时代的变量声明方式。Chrome（V8）中代码必须开启严格模式以使用 `let`。
- const: 类似于 `let`，但是变量的值无法被修改。
- var: 旧时代的变量声明方式。

变量命名

- 变量名称必须仅包含字母，数字，符号 `$` and `_`。
- 首字符必须非数字。

数据类型

JavaScript 中的变量可以保存任何数据。变量在前一刻可以是个字符串，然后又收到一个数值：

```
// 没有错误
let message = "hello";
message = 123456;
```

允许这种操作的编程语言称为“动态类型”（dynamically typed）的编程语言，意思是，拥有数据类型，但是变量并不限于数据类型中的任何一个。

- number 用于任何类型的数字：整数或者浮点数。
- 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。
- object 用于更复杂的数据结构。
- symbol 用于唯一的标识符。

typeof运算符

typeof 运算符返回参数的类型。当我们想要分别处理不同类型值的时候，或者简单地进行检验，就很有用。它支持两种语法形式：

- 作为运算符：typeof x。
- 函数形式：typeof(x)。

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof Math // "object"  # Math 是一个提供数学运算的内建对象
typeof null // "object" # 这是官方在 typeof 方面承认的错误，只是为了兼容性而保留
typeof alert // "function" # alert 在语言中是一个函数
```

类型转换

大多数情况下，运算符和函数会自动转换将值转换为正确的类型。称之为“类型转换”。 比如，`alert` 会自动将任何值转换为字符串。算术运算符会将值转换为数字。

ToString

当需要一个值的字符串形式，就会进行 string 类型转换。 比如，`alert(value)` 将 `value` 转换为 string 类型，然后显示这个值。 也可以显式地调用 `String(value)` 来达到这一目的：

```
let value = true;
alert(typeof value); // boolean
value = String(value); // 现在，值是一个字符串形式的 "true"
alert(typeof value); // string
```

ToNumber

在算术函数和表达式中，会自动进行 number 类型转换。

```
alert( "6" / "2" ); // 3, strings are converted to numbers

let str = "123";
alert(typeof str); // string

let num = Number(str); // 变成 number 类型 123
alert(typeof num); // number

let age = Number("an arbitrary string instead of a number");
alert(age); // NaN, conversion failed
```

number 类型转换规则：

值	结果
undefined	NaN
null	0
true 和 false	1 和 0
string	字符串开始和末尾的空白会被移除，剩下的如果是空字符串，结果为 0，否则 —— 从字符串中读出数字。错误返回 NaN。

ToBoolean

逻辑操作或显式调用 Boolean(value) 会触发 boolean 类型转换。 转换规则如下：

- 假值，比如 0、空的字符串、null、undefined 和 NaN 变成 false。
- 其他值变成 true。

运算符

- 运算符
- 一元运算符

- 二元运算符

字符串连接功能

通常，加号 + 用来求和。 但是如果加号 + 应用于字符串，它将合并（连接）各个字符串：

```
let s = "my" + "string";
alert(s); // mystring

alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"

alert(2 + 2 + '1' ); // "41" 而不是 "221"
```

数字转换功能

加号 `+` 有两种形式。一种是以上讨论的二元运算符，还有一种是一元运算符。

一元运算符加号，或者说，加号 `+` 应用于单个值，对数字没有作用。但是如果运算元是非数字，它会将其转化为数字。

```
// 对数字无效
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// 转化非数字
alert( +true ); // 1
alert( +"" );   // 0
```

运算符优先级

在 JavaScript 中有众多运算符。每个运算符都有对应的优先级数字。数字越大，越先执行。如果优先级相同，那么执行顺序由左至右。优先级表

赋值运算符

位运算符

位运算符把运算元当做 32 位比特序列，并在它们的二元表现形式上操作。

- 按位与 (&)
- 按位或 (|)
- 按位异或 (^)
- 按位非 (~)

- 左移 (<<)
- 右移 (>>)
- 无符号右移 (>>>)

值的比较

在 JavaScript 中，我们可以使用一些熟知的数学符号进行值的比较：

- 大于 / 小于：`a > b`，`a < b`。
- 大于等于 / 小于等于：`a >= b`，`a <= b`。
- 检测两个值的相等写为 `a == b`。
- 检测两个值的不等写为 `a != b`。

字符串间的比较

在比较字符串的大小时，会使用“字典”或“词典”顺序进行判定。换言之，字符串是按字符（母）逐个进行比较的。

不同类型间的比较

当不同类型的值进行比较时，它们会首先被转为数字（number）再判定大小。

```
alert( '2' > 1 ); // true，字符串 '2' 会被转为数字 2
alert( '01' == 1 ); // true，字符串 '01' 会被转为数字 1

alert( true == 1 ); // true
alert( false == 0 ); // true

let a = 0;
alert( Boolean(a) ); // false
let b = "0";
alert( Boolean(b) ); // true
alert(a == b); // true!
```

严格相等

严格相等操作符 `===` 在进行比较时不会做任何的类型转换。换句话说，如果 `a` 和 `b` 属于不同的数据类型，那么 `a === b` 不会做任何的类型转换而立刻返回 `false`。

null 和 undefined

当使用严格相等 `===` 比较二者时： 它们是不相等的，因为它们属于不同的类型。

```
alert( null === undefined ); // false
```

当使用非严格相等 `==` 比较二者时： JavaScript 存在一个专属的规则，会判定它们互等。而它们就像“一对恋人”，仅仅等于（非严格相等下）对方而不等于其他任何的值。

```
alert( null == undefined ); // true
```

当使用数学式或其他比较方法 `< > <= >=` 时： `null/undefined` 的值会被转换为数字：`null` 转为 `0`，`undefined` 转为 `NaN`。

奇怪的结果

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) *!*true*/!*

alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

条件运算符

- if
- else
- else if
- ?
- switch

逻辑运算符

- ||
  - 从左到右依次计算操作数。
  - 将每一个操作数转化为布尔值。如果结果是 true，就停止计算，返回这个操作数的初始值。
  - 如果所有的操作数都被计算过（也就是，转换结果都是 false），返回最后一个操作数。
- &&
  - 从左到右依次计算操作数。
  - 将每一个操作数转化为布尔值。如果结果是 false，就停止计算，返回这个操作数的初始值。
  - 如果所有的操作数都被计算过（也就是，转换结果都是 true），返回最后一个操作数。
- !
  - 将操作数转化为布尔类型：`true/false`。
  - 返回相反的值。

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。这样的循环就像任何其他循环一样，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 continue 指令就会执行它。

break/continue 支持循环前的标签。标签是 break/continue 避免嵌套并转到外部循环的唯一方法。

函数

函数是程序的主要“构建模块”，它们允许不重复地多次调用代码。

函数声明

```
function showMessage() {
  alert( 'Hello everyone!' );
}
```

`function` 关键字首先出现，然后是**函数名**，然后是括号（在上述示例中为空）之间的**参数列表**，最后是花括号之间的代码（即“函数体”）。

局部变量

在函数中声明的变量只在该函数内部可见。

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // 局部变量
  alert( message );
}
showMessage(); // Hello, I'm JavaScript!
alert( message ); // <-- 错误！变量是函数的局部变量
```

外部变量

```
let userName = 'John';
function showMessage() {
  userName = "Bob"; // 改变外部变量
  let message = 'Hello, ' + userName;
  alert(message);
}
showMessage();
alert( userName ); // Bob, 值被函数修改
```

只有在没有本地变量的情况下才会使用外部变量。因此如果我们忘记了 `let`，可能会发生意外修改的情况。

如果在函数中声明了同名变量，那么它**遮蔽**外部变量。例如，在如下代码中，函数使用本地的 `userName`，外部部分被忽略：

```
let userName = 'John';
function showMessage() {
  let userName = "Bob"; // 声明一个局部变量
  let message = 'Hello, ' + userName; // Bob
  alert(message);
}
//  函数会创建并使用它自己的 userName
showMessage();
alert( userName ); // John，未更改，函数没有访问外部变量。
```

参数

我们可以使用参数（也称“函数参数”）来将任意数据传递给函数。 在如下示例中，函数有两个参数：`from` 和 `text`。

```
function showMessage(from, text) { // 参数：from、text
  alert(from + ': ' + text);
}
showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

当在行 `(*)` 和 `(**)` 中调用函数时，将给定的值复制到局部变量 from 和 text。然后函数使用它们。

我们有一个变量 from，将它传递给函数。请注意：函数会修改 from，但在外部看不到更改，因为函数修改的是变量的副本：

```
function showMessage(from, text) {
  from = '*' + from + '*'; // 让 "from" 看起来更优雅
  alert( from + ': ' + text );
}
let from = "Ann";
showMessage(from, "Hello"); // *Ann*: Hello
// "from" 值相同，函数修改了本地副本。
alert( from ); // Ann
```

默认值

如果未提供参数，则其值是 `undefined`。

```
function showMessage(from, text) { // 参数：from、text
  alert(from + ': ' + text);
}
showMessage('Ann'); // Ann: undefined

function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
```

```
}
showMessage("Ann"); // Ann: no text given
```

返回值

函数可以将一个值返回到调用代码中作为结果。

指令 `return` 可以在函数的任意位置。当执行到达时，函数停止，并将值返回给调用代码。

在一个函数中可能会出现很多次 `return`.

在没有值的情况下使用 `return` 可能会导致函数立即退出。

函数命名

- 名称应该清楚地描述函数的功能。当我们在代码中看到一个函数调用时，一个好的名称立即让我们了解它所做的和返回的事情。
- 函数是一种行为，所以函数名通常是动词。
- 有许多优秀的函数前缀，如 `create…`、`show…`、`get…`、`check…` 等等。使用它们来提示函数的作用。

函数表达式

下面的语法我们通常叫**函数声明**:

```
function sayHi() {
  alert( "Hello" );
}
```

下面是另一种创建函数的方法叫**函数表达式**:

```
let sayHi = function() {
  alert( "Hello" );
};
```

在这里，函数被创建并像其他赋值一样，明确的分配给了一个变量。不管函数如何定义，它只是一个存储在变量中的值 `sayHi`。

简单说就是 "创建一个函数并放进变量 `sayHi`"。

函数声明和表达式之间的关键区别

细微差别是在 JavaScript 引擎中在**什么时候**创建函数。

函数表达式在执行到达时创建并可用。

一旦执行到右侧分配 let sum = function…，就会创建并可以使用(复制，调用等)。

函数声明则不同。

函数声明可用于整个脚本/代码块。换句话说，当 JavaScript 准备运行脚本或代码块时，它首先在其中查找函数声明并创建函数。我们可以将其视为“初始化阶段”。

在处理完所有函数声明后，执行继续。

因此，声明为函数声明的函数可以比定义的更早调用。

```
sayHi("John"); // Hello, John
function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

```
sayHi("John"); // error!
let sayHi = function(name) {  // (*) no magic any more
  alert( `Hello, ${name}` );
};
```

回调函数

使用函数表达式传递函数。

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  alert( "You agreed." );
}
function showCancel() {
  alert( "You canceled the execution." );
}
// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

`ask` 参数调用回调函数或只是回调。

我们的想法是，我们传递一个函数，并希望稍后在必要时回调它。在我们的例子中，`showOk` 对应回答 `"yes"` 的回调，`showCancel` 对应回答“否”。

```
ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

这里函数直接写在 `ask(...)` 调用。他们没有名字，叫**匿名函数。**

对象: 基础知识

对象

对象用来存储键值对和更复杂的实体。对象可以通过中括号 `{…}` 和其中包含一些可选的**属性**来创建。属性是一个键值对，键是一个字符串（也叫做属性名），值可以是任何类型。

我们可以用下面两种语法的任一种来创建一个空的对象:

```
let user = new Object(); // “构造函数” 的语法
let user = {};  // “字面量” 的语法
```

文本和属性

我们可以在创建的时候立马给对象一些属性，在 `{...}` 里面放置一些键值对。

```
let user = {     // 一个对象
  name: "John",  // 键 "name"，值 "John"
  age: 30,        // 键 "age"，值 30
  "likes birds": true  // 两个单词被包在一起
};
```

任何时候我们都可以添加，删除，读取文件。

可以通过点语法来使用属性:

```
// 读取文件的属性:
alert( user.name ); // John
alert( user.age ); // 30

// 新增属性
user.isAdmin = true;
// 移除属性
delete user.age;
```

方括号

对于多字词语表示的属性，点操作就不能用啦。

```
let user = {};
// set
user["likes birds"] = true;
// 语法错误
```

```
user.likes birds = true
// get
alert(user["likes birds"]); // true
// delete
delete user["likes birds"];
```

存在值检查

对象的一个显著的特点就是可以访问任何属性，如果这个属性名没有值也不会有错误。访问一个不存在的属性会返回 `undefined`。它提供一种普遍的方法去检查属性是否存在 —— 获得值来与 undefined 比较：

```
let user = {};
alert( user.noSuchProperty === undefined ); // true 意思是没有这个属性
```

同样也有一个特别的操作符 `"in"` 来检查是否属性存在。

```
let user = { name: "John", age: 30 };
alert( "age" in user ); // true，user.age 存在
alert( "blabla" in user ); // false，user.blabla 不存在。
```

for...in 循环

为了使用对象所有的属性，就可以利用 `for..in` 循环。这跟 `for(;;)` 是完全不一样的东西。

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};
for(let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // 属性键的值
  alert( user[key] ); // John, 30, true
}
```

引用复制

对象和其他原始的类型相比有一个很重要的区别，对象都是按引用存储复制的。

原始类型是：字符串，数字，布尔类型 -- 是被整个赋值的。

```
let message = "Hello!";
let phrase = message;
```

结果是我们得到了不同的值，每个存的都是 `"Hello!"`。

对象跟这个不一样。

变量不存对象本身，只是对象的“内存地址”，是对象的引用。

当对象被复制的时候 -- 引用被复制了一份，对象并没有被复制。

```
let user = { name: "John" };
let admin = user; // 复制引用

admin.name = 'Pete'; //  改变 "admin" 的引用
alert(user.name); // 'Pete'
```

比较引用

等号 `==` 和严格等 `===` 对于对象来说没差别。**当两个引用指向同一个对象的时候他们相等。**

常量对象

一个被 `const` 修饰的对象可以修改。

```
const user = {
  name: "John"
};
user.age = 25; //
alert(user.age); // 25
```

这是因为 `const` 仅仅修饰 `user`。在这里 `user` 存的是一个对象的引用。引用的地址没有变，只是引用的对象被修改了。

复制和合并

那么我们该怎么复制一个对象呢？创建一份独立的拷贝，一份复制？

```
let user = {
  name: "John",
  age: 30
};
let clone = {}; // 新的空对象
// 复制所有的属性值
for (let key in user) {
  clone[key] = user[key];
}
// 现在复制是独立的复制了
clone.name = "Pete"; // 改变它的值
alert( user.name ); // 原对象属性值不变
```

我们也可以用 `Object.assign` 来实现。

深拷贝?

对象方法与 this

对象方法

```
let user = {
  name: "John",
  age: 30
};
user.sayHi = function() {
  alert("Hello!");
};
user.sayHi(); // Hello!
```

```
// 这些对象作用一样
let user = {
  sayHi: function() {
    alert("Hello");
  }
};
// 方法简写看起来更好，对吧？
let user = {
  sayHi() { // 与 "sayHi: function()" 一样
    alert("Hello");
  }
};
```

this

对象方法需要访问对象中的存储的信息完成其工作是很常见的。

为了访问该对象，方法中可以使用 `this` 关键字。

`this` 的值就是在点之前的这个对象，即调用该方法的对象。

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    alert(this.name);
  }
};
user.sayHi(); // John
```

在 JavaScript 中，`"this"` 关键字与大多数其他编程语言中的不同。首先，它可以用于任何函数。

`this` 是在运行时求值的。它可以是任何值。

```
function sayHi() {
  alert(this);
}
sayHi(); // undefined
```

在这种情况下，严格模式下的 `this` 值为 `undefined`。如果我们尝试访问 `this.name`，将会出现错误。

在非严格模式（没有使用 `use strict`）的情况下，`this` 将会是全局对象（浏览器中的 window，我们稍后会进行讨论）。`"use strict"` 可以修复这个历史行为。

对象原始值转换

当对象相加 `obj1 + obj2`，相减 `obj1 - obj2`，或者使用 `alert(obj)` 打印时会发生什么？

对于对象，不存在 `to-boolean` 转换，因为所有对象在布尔上下文中都是 `true`。所以只有字符串和数值转换。

ToPrimitive

当一个对象被用在需要原始值的上下文中时，例如，在 `alert` 或数学运算中，它会使用 `ToPrimitive` 算法转换为原始值。

该算法允许我们使用特殊的对象方法自定义转换。

取决于上下文，转换具有所谓的“暗示”：

- 调用 `obj[Symbol.toPrimitive](hint)` 如果这个方法存在的话，
- 否则如果暗示是 `"string"`
  - 尝试 `obj.toString()` 和 `obj.valueOf()`，无论哪个存在。
- 否则，如果暗示 `"number"` 或者 `"default"`
  - 尝试 `obj.valueOf()` 和 `obj.toString()`，无论哪个存在。

构造函数和操作符 new

构造函数

构造函数在技术上是常规函数。不过有两个约定：

1. 他们首先用大写字母命名。
2. 它们只能用 "new" 操作符来执行。

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Jack");
```

```
alert(user.name); // Jack
alert(user.isAdmin); // false
```

当一个函数作为 `new User(...)` 执行时，它执行以下步骤：

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。

数据类型

- number
- string
- array
- iterable
- map
- set

函数

Rest 参数和 Spread 操作符

Rest 参数

在 JavaScript 中，无论函数定义了多少个形参，你都可以传入任意个实参进行调用。

```
function sum(a, b) {
  return a + b;
}
alert( sum(1, 2, 3, 4, 5) );
```

我们可以在定义函数时使用 `Rest` 参数，`Rest` 参数的操作符表示为3个点 `...`。直白地讲，它的意思就是“把剩余的参数都放到一个数组中”。

```
function sumAll(...args) { // 数组变量名为 args
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}
alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

我们也可以显式地定义和取用前面部分的参数，而把后面部分的参数收集起来。

下面的例子即把前两个参数定义为变量，同时把剩余的参数收集到 `titles` 数组中：

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar
  // titles 数组中包含了剩余的参数
  // 也就是有 titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}
showName("Julius", "Caesar", "Consul", "Imperator");
```

"arguments" 变量

函数的上下文会提供一个非常特殊的类数组对象 `arguments`，所有的参数被按序放置。

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );
  // 它是可遍历的
  // for(let arg of arguments) alert(arg);
}
// 依次弹出提示：2，Julius，Caesar
showName("Julius", "Caesar");
// 依次弹出提示：1，Ilya，undefined（不存在第二个参数）
showName("Ilya");
```

即使 `arguments` 是一个类数组且可遍历的变量，但它终究不是数组。它没有数组原型链上的函数，我们没法直接调用诸如 `arguments.map(...)` 等这样的函数。

Spread 操作符

它看起来和 `Rest` 参数操作符很像，都表示为 `...`，但是二者完全做了相反的事。

当在函数调用时使用 `...arr`，它会把可迭代的对象 `arr` “展开”为参数列表。

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];
alert( Math.max(...arr1, ...arr2) ); // 8
```

闭包

几个问题

1. 函数 sayHi 用到 name 这个外部变量 。当函数执行时，它会使用哪一个值呢?

var

变量声明的三种方式：`let`，`const`，`var`

`let` 和 `const` 在词法环境中的行为完全一样。

```
function sayHi() {
  var phrase = "Hello"; // 局部变量，使用 "var"，而不是 "let"
  alert(phrase); // Hello
}
sayHi();
alert(phrase); // 报错：phrase is not defined
```

var 没有块级作用域

用 `var` 声明的变量，不是函数范围就是全局的，它们在块内是可见的。

```
if (true) {
  var test = true; // 用 "var" 而不是 "let"
}
alert(test); // true，变量在 if 结束后仍存在

for (var i = 0; i < 10; i++) {
  // ...
}
alert(i); // 10, "i" 在循环结束后仍然可见，它会成为一个全局变量
```

```
  if (true) {
    var phrase = "Hello";
  }
  alert(phrase); // works
}
sayHi();
alert(phrase); // 报错：phrase is not defined
```

可以看到，`var` 穿透了 `if`、`for` 或其它块级代码。

var 在函数开头被处理

`var` 声明在函数开始时处理（或者全局声明之于脚本开始）。

换言之，`var` 变量会在函数开头被定义，与它在代码里定义的位置无关（这里不考虑定义在嵌套函数里的场景）。

```
function sayHi() {
  phrase = "Hello";
  alert(phrase);
  var phrase;
}
```

```
function sayHi() {
  var phrase;
  phrase = "Hello";
  alert(phrase);
}
```

声明会被提升，但是赋值不会。

```
function sayHi() {
  alert(phrase);
  var phrase = "Hello";
}
sayHi();
```

全局对象

JavaScript 初创时，存在一种借助「全局对象」提供全局变量和函数的思想。它的初衷是多个浏览器中的代码可以通过一个全局对象共享变量。

从那之后，JavaScript 有了极大的发展，那种通过全局变量来连通代码的想法不再受欢迎。在现代 JavaScript 中，模块化概念成为主流。

但是全局对象仍然在规范中保留着。

函数对象

在 JavaScript 里，函数是对象。

- name
- length
- 自定义属性

命名函数表达式

命名函数表达式（NFE，Named Function Expression），指代有名字的函数表达式的术语。

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

然后给它加一个名字：

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
sayHi("John"); // Hello, John
```

关于名字 func，有两个特殊的地方：

- 它允许函数在内部引用自己。
- 它在函数外是不可见的。

```
let sayHi = function *!*func*/!*(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // 使用 func 再次调用自己
    // sayHi("Guest");
  }
};
sayHi(); // Hello, Guest
// 但这个无法生效
func(); // Error, func is not defined（在函数外不可见）
```

我们为什么使用 `func` 呢？为什么不直接在嵌套调用里使用 `sayHi`？

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Error: sayHi is not a function
  }
};

let welcome = sayHi;
sayHi = null;
welcome(); // Error，嵌套调用 sayHi 不再有效！
```

那是因为函数从它的外部词法环境获取 `sayHi`。没有局部的 `sayHi`，所以外部变量被使用。而当调用时，外部的 `sayHi` 是 `null`。

我们给函数表达式增加的可选的名字正是用来解决这个问题的。

new Function

语法

```
let func = new Function ([arg1[, arg2[, ...argN]],] functionBody)
```

即在创建函数时，先传入函数所需的参数（准确地说是形参名），最后传入函数的函数体。传入的所有参数均为字符串。

```
let sum = new Function('a', 'b', 'return a + b');
alert( sum(1, 2) ); // 3
```

与已知方法相比这种方法最大的不同是，它是在运行时使用描述函数的字符串来创建函数的。

使用 `new Function` 创建函数的应用场景非常特殊，比如需要从服务器获取代码或者动态地按模板编译函数时才会使用，在一般的程序开发中很少使用。

闭包

通常，函数会使用一个特殊的属性 `[[Environment]]` 来记录函数创建时的环境，它具体指向了函数创建时的词法环境。

但是如果我们使用 `new Function` 创建函数，函数的 `[[Environment]]` 并不指向当前的词法环境，而是指向全局环境。

```
function getFunc() {
  let value = "test";
  // let func = function() { alert(value); };
  let func = new Function('alert(value)');
  return func;
}
getFunc()(); // error: value 未定义
```

对象、类和继承

异常处理

try..catch

```
try {
  // 代码...
} catch (err) {
  // 异常处理
}
```

1. 首先，执行 try {...} 里面的代码。
2. 如果执行过程中没有异常，那么忽略 catch(err) 里面的代码，try 里面的代码执行完之后跳出该代码块。
3. 如果执行过程中发生异常，控制流就到了 catch(err) 的开头。变量 err（可以取其他任何的名称）是一个包含了异常信息的对象。

Error对象

4. `name`：异常名称，对于一个未定义的变量，名称是 "ReferenceError"
5. `message`：异常详情的文字描述。
6. `stack`：当前的调用栈：用于调试的,一个包含引发异常的嵌套调用序列的字符串。

抛出自定义的异常

"Throw" 运算符

```
let json = '{ "age": 30 }'; // 不完整的数据

try {
  let user = JSON.parse(json); // <-- 没有异常
  if (!user.name) {
    抛出 new SyntaxError("Incomplete data: no name"); // (*)
  }
  alert( user.name );
} catch(e) {
  alert( "JSON Error: " + e.message ); // JSON Error: Incomplete data: no name
}
```

try..catch..finally

```
try {
  alert( 'try' );
  if (confirm('Make an error?'))            BAD_CODE();
} catch (e) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

这段代码有两种执行方式：

1. 如果对于 "Make an error?" 你的回答是 "Yes"，那么执行 try -> catch -> finally。
2. 如果你的回答是 "No"，那么执行 try -> finally。 finally 的语法通常用在：我们在 try..catch 之前开始一个操作，不管在该代码块中执行的结果怎样，我们都想结束的时候执行某个操作。

全局 catch

即使我们没有使用 `try..catch`，绝大多数执行环境允许我们设置全局的异常处理机制来捕获出现的异常。浏览器中，就是 `window.onerror`。

如何编写不良代码?

> 学而不思则罔，思而不学则殆。

过去的程序员忍者使用这些技巧来使代码维护者的头脑更加敏锐。

代码审查大师在测试任务中寻找它们。

一些新入门的开发者有时候甚至比忍者程序员更好的使用它们。

仔细阅读它们，找出你是谁 —— 一个忍者、一个新手、或者一个代码审查者？

> 许多人试图追随忍者的脚步。只有极少数成功了。

简洁是智慧的灵魂

让代码尽可能地短一点。展示出你是多么的聪明啊。

让一些巧妙的语言特性来指导你。

例如，看一下这个三元运算符 `'?'`：

```
// 从一个著名的 javascript 库中取到的代码
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

很酷，对吗？如果你这样写了，那些看到这一行代码并尝试去理解 `i` 的值是什么的开发者们就会有一个“快活的”的时光了。然后会来找你寻求答案。

告诉他短一点总是更好的。引导他进入忍者之路。

一个字母的变量

> 道隐无名。夫唯道善贷且成。

编码更快（也更糟糕）的另一种方式是到处使用单字母的变量名。像是 `a`、`b` 或 `c`。

短变量会像森林中真正的忍者一样在代码中消失不见。没有人能够通过编辑器的“搜索”找到它。即使有人做到了，他也不能“破译”出变量名 `a` 或 `b` 是什么意思。

...但是有一个例外情况。一个真正的忍者绝不会在 `"for"` 循环中使用 `i` 作为计数器。在任何地方都可以，但是这里不会用。看一下四周吧，还有很多不常用的字母呢。例如 `x` 或 `y`。

如果循环体能够达到 1-2 页（如果可以的话可以让它更长）那么长的话，使用一个不常用的变量作为循环的计数器就更酷了。如果某人看到循环内部的深处后，他就不能很快地找出变量 `x` 是循环计数器啦。

使用缩写

如果团队规则中禁止使用一个字母和模糊的名字，那就缩短他们，使用缩写吧。

像这样：

- `list` -> `lst`
- `userAgent` -> `ua`
- `browser` -> `brsr`
- ...等等

只有具有真正良好直觉的人才能够理解所有的这些名字。尽量缩短一切。只有一个有价值的人才能够维护这种代码的开发。

Soar high，抽象化。

```
大方无隅<br>
大器晚成<br>
大音希声<br>
大象无形
```

当选择一个名字时尽可能尝试使用最抽象的词语。例如 `obj`、`data`、`value`、`item`、`elem` 等等。

- **一个变量的理想名称是 `data`。** 在任何能用的地方都使用它。的确，每个变量都持有 **data**，对吧？

  ...但是 `data` 已经用过了怎么办？可以尝试一下 `value`，它也很普遍呢。一个变量总会有一个 **value**，对吧？

- **根据变量的类型命名：`str`、`num`...**

  尝试一下吧。新手可能会诧异 —— 这些名字对于忍者来说真的有用吗？事实上，是的！

  一方面，变量名仍然有着一些含义。它说明了变量内是什么：一个字符串、一个数字或是其他的东西。但是当一个局外人试图理解代码时，他会惊讶地发现实际上没有任何有效信息！最终无法改变你精心思考过的代码。

  事实上，值的类型很容易就能通过调试看出来。但是变量名的含义呢？它存了哪一个字符串/数字？

  如果不深思是没有办法找出来的！

- **...但是如果找不到更多这样的名字呢？** 可以加一个数字：`item1`、`item2`、`elem5`、`data1`...

注意测试

只有一个真正细心的程序员才能理解代码。但是怎么检验呢？

方式之一 —— 使用相似的变量名，像 `date` 和 `data`。

尽你所能地将它们混合在一起。

快速阅读这些代码是不可能的。并且如果有一个错别字时... (⊙v⊙)嗯...我们卡在这里很久没有喝茶了。

智能同义词

> 最难的事情是在黑暗的房间里找到一只黑猫，特别是如果没有猫。

对于**同样的**事情使用**相同**的名字，可以使生活更有趣，并向公众展示出你的创意。

例如，函数前缀。如果一个函数是在屏幕上展示一个消息 —— 可以以 `display…` 开始，例如 `displayMessage`。如果另一个函数展示别的东西，比如一个用户名，可以以 `show…` 开始（例如 `showName`）。

暗示这些函数之间有微妙的差异，实际上并没有。

与团队中的其他忍者们达成一个协议：如果 John 在他的代码中以 `display...` 来开始一个"显示"函数，那么 Peter 可以用 `render..`，Ann 可以使用 `paint...`。你可以发现代码变得多么的有趣多样呀。

现在是帽子戏法！

对于有非常重要的差异的两个函数使用相同的前缀。

例如，`printPage(page)` 函数会使用一个打印机。`printText(text)` 函数会将文字显示到屏幕上。让一个陌生的读者来思考一下：“`printMessage(message)` 会将消息放到哪里呢？打印机还是屏幕上？”，为了使它真正耀眼，`printMessage(message)` 应该将消息输出在新窗口中！

重用名字

> 始制有名，
> 名亦既有，
> 夫亦将知止，
> 知止可以不殆。

仅在绝对必要时才添加新变量。

否则，重用已经存在的名字。只需要将新值写进变量即可。

在一个函数中，尝试仅使用作为参数传递的变量。

这样就无法确定这个变量现在是什么了。也不知道它是从哪里来的。一个弱直觉的人必须逐行分析代码，并通过每个代码分支跟踪变化。

这个方法的一个进阶方案是在一个循环或函数中偷偷地替换掉它的值。

例如：

```
function ninjaFunction(elem) {
  // 基于变量 elem 进行工作的 20 行

  elem = clone(elem);

  // 又 20 行，现在是使用 clone 后的 elem 变量。
}
```

想要在第二部分中使用 `elem` 的程序员会非常的诧异滴...只有在调试期间，在检查代码之后，他会发现他正在使用克隆过的变量！

经常看到这样的代码，即使对经验丰富的忍者来说也是致命的。

下划线的乐趣

在变量名前使用 `_` 和 `__`。例如 `_name` 和 `__value`。如果只有你知道他们的含义的话将会非常棒。或者，更棒的是，其实没有意义。

你一枪杀死了两只兔子。首先，代码变得更长降低了可读性；第二，你的开发者小伙伴可能会花费很长时间来弄清楚下划线是什么意思。

一个聪明的忍者会在代码的一个地方使用下划线然后在其他地方刻意避免使用它们。这会使得代码变得更加脆弱，并增加未来出现错误的可能性。

展示你的爱

让大家看看你的实体是多么壮观！像 `superElement`、`megaFrame` 和 `niceItem` 这样的名字一定会启发读者。

事实上，从一方面来说，看似写了一些东西：`super..`、`mega..`、`nice..`，但是从另一方面来说 —— 并没有提供任何细节。读者可能要寻找一个隐藏的含义或深思一两个小时。

重叠外部变量

```
处明者不见暗中一物<br>，
处暗者能见明中区事。
```

对函数内外的变量使用相同的名称。很简单，一点也不费劲。

```
let user = authenticateUser();

function render() {
  let user = anotherValue();
  ...
  ...many lines...
  ...
  ... // <-- 某个程序员想要在这里使用 user 变量...
  ...
}
```

跳过 `render` 的程序员可能不会注意到有一个本地 `user` 遮挡外部的 `user` 了。

然后他会假设 `user` 仍然是外部的变量然后使用它， `authenticateUser()` 的结果... 陷阱出来啦！你好呀，调试器...

无处不在的副作用！

有些函数看起来它们不会改变任何东西。例如 `isReady()`、 `checkPermission()`、 `findTags()` ... 它们被假定为会执行计算，查找和返回数据，而不需要更改任何外部的数据。这被称为“无副作用”。

一个非常好的技巧 - 除了主要任务之外，还要向它们添加一个“有用的”动作

当你的同事看到被命名为 `is..`、 `check..` 或 `find...` 的函数改变了某些东西的时候，他的脸上肯定是一脸懵逼的状态。

另一种惊喜的方式是返回非标准的结果。

展示你原来的想法！让调用 `checkPermission` 时返回的不是 `true/false`，而是一个包含检查结果的复杂对象。

那些尝试写 `if (checkPermission(..))` 的开发者会怀疑为什么它不能工作。告诉他们：“去读文档吧”。然后给出这篇文章。

强大的函数！

```
大道泛兮，<br>
其左可右。
```

不要让函数受限于名字中写的那样。变得更宽泛一点吧。

例如，函数 `validateEmail(email)` 可以（除了检查邮件的正确性之外）显示一个错误消息并要求重新输入邮件。

额外的动作在函数名称中不应该很明显。一个真正的忍者会使它们在代码中也不明显。

将多个动作加入到一起可以保护您的代码避免重用。

想象一下，另一个开发者只想检查邮箱而不想输出任何信息。你的函数 `validateEmail(email)` 对他而言就不合适啦。所以他不会找你问一些关于这些函数的事情从而打断你的思考。