

【AAA货】前端Base64编码知识，一文打尽，探索起源，追求真相。

## 前言

本文收录在 [前端基础进阶](#) 专栏，欢迎关注和收藏，往期经典： \* [【干货】私藏的这些高级工具函数，你拥有几个?](#) 500+ star \* [三千文字，也没写好 Function.prototype.call](#) 200+ star \* [那些你熟悉而又陌生的函数](#) 200+ star \* [这16种原生函数和属性的区别，你真的知道吗？](#) 精心收集，高级前端必备知识，快快打包带走 100+ star

收藏不star就是要流氓！哈

## 大纲

方便移动端阅读：

- Base64在前端的应用
- Base64数据编码起源
- Base64编码64的含义
- Base64编码优缺点
- 一些计算机和前端基础知识
- ASCII码，Unicode，UTF-8
- Base64编码和解码
- 其他的成熟方案
- 写在最后

## Base64在前端的应用

Base64编码，你一定知道的，先来看看她在前端的一些常见应用：  
当然绝大部分场景都是基于[Data URLs](#)

### Canvas图片生成

canvas的 [toDataURL](#) 可以把canvas的画布内容转base64编码格式包含图片展示的 [data URI](#)。

```
const ctx = canvasEl.getContext("2d");
// ..... other code
const dataUrl = canvasEl.toDataURL();

// data:image/png;base64,iVBORw0KGgoAAAANSUxE.....
```

你画我猜，新用户加入，要获取当前的最新的绘画界面，也可以通过Base64格式的消息传递。

### 文件读取

FileReader的 [readAsDataURL](#) 可以把上传的文件转为base64格式的data URI，比较常见的场景是用户头像的剪裁和上传。

```
function readAsDataURL() {
  const fileEl = document.getElementById("inputFile");
  return new Promise((resolve, reject) => {
    const fd = new FileReader();
    fd.readAsDataURL(fileEl.files[0]);
    fd.onload = function () {
      resolve(fd.result);
      // data:image/png;base64,iVBORw0KGgoAAAA.....
    }
    fd.onerror = reject;
  });
}
```

jwt由header,payload,siganture三部分组成，前两个解码后，都是可以明文看见的。拿 [国服最强JWT生成Token做登录校验讲解，看完保证你学会！](#) 里面的token做测试。



## 网站图片和小图片

### 移动端网站图标优化

```
<link rel="icon" href="data:," />
<link rel="icon" href="data:;base64,=" />
```

至于怎么获得这个值data:,的:

```
<canvas height="0" width="0" id="canvas"></canvas>
<script>
  const canvasEl = document.getElementById("canvas");
  const ctx = canvasEl.getContext("2d");
  dataUrl = canvasEl.toDataURL();
  console.log(dataUrl); // data:,
</script>
```

### 小图片

这个就有很多场景了，比如img标签，背景图等

img标签:

```

```

css背景图:

```
.bg{
  background: url(data:image/png;base64,iVBORw0KGgoAAAA.....)
}
```

### 简单的数据加密

当然这不是好方法，但是至少让你不好解读。

```
const username = document.getElementById("username").vlaue;
const password = document.getElementById("password").vlaue;
const secureKey = "%$%$DS)_sdsdj_66";
const sPass = utf8_to_base64(password + secureKey);
```

```
doLogin({
  username,
  password: sPass
})
```

## SourceMap

借用阮大神的一段代码, 注意mappings字段, 这实际上就是base64编码格式的内容, 当然你直接去解, 是会失败的。

```
{
  version : 3,
  file: "out.js",
  sourceRoot : "",
  sources: ["foo.js", "bar.js"],
  names: ["src", "maps", "are", "fun"],
  mappings: "AAgBC, SAAQ, CAAEA"
}
```

具体的实现请看官方的[base64-vlq.js](#)文件。

## 混淆加密代码

著名的代码混淆库, [javascript-obfuscator](#), 其也是有应用base64几码的, 一起来看看选项:  
[webpack-obfuscator](#)也是基于其封装的。

```
--string-array-indexes-type '<list>' (comma separated) [hexadecimal-number,
--string-array-encoding '<list>' (comma separated) [none, base64, rc4]
--string-array-index-shift <boolean>
--string-array-wrappers-count <number>
--string-array-wrappers-chained-calls <boolean>
```

```
--source-map-mode <string> [inline, separate]
--source-map-sources-mode <string> [sources, sources-content]
--split-strings <boolean>
--split-strings-chunk-length <number>
--string-array <boolean>
--string-array-indexes-type '<list>' (comma separated) [hexadecimal-number, hexade
--string-array-encoding '<list>' (comma separated) [none, base64, rc4]
--string-array-index-shift <boolean>
--string-array-wrappers-count <number>
--string-array-wrappers-chained-calls <boolean>
--string-array-wrappers-parameters-max-count <number>
--string-array-wrappers-type <string> [variable, function]
```

## 其他

X.509公钥证书, github SSH key, mht文件, 邮件附件等等, 都有Base64的影子。

## Base64数据编码起源

早期邮件传输协议基于 ASCII 文本, 对于诸如图片、视频等二进制文件处理并不好。ASCII 主要用于显示现代英文, 到目前为止只定义了 128 个字符, 包含控制字符和可显示字符。为了解决上述问题, Base64 编码顺势而生。

Base64是编解码, 主要的作用不在于安全性, 而在于让内容能在各个网关间无错的传输, 这才是Base64编码的核心作用。

除了Base64数据编码, 其实还有Base32数据编码, Base16数据编码, 可以参见 [RFC 4648](#)。

## Base64编码64的含义

64就是64个字符的意思。

base64对照表， 借用 [Base64原理](#) 的一张图：

数值	字符	数值	字符	数值	字符	数值	字符
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

- 1. A-Z 26
- 2. a-z 26
- 3. 0-9 10
- 4. + / 2

$26 + 26 + 10 + 2 = 64$

当然还有一个字符=，这是填充字符，后面会提到，不属于64里面的范畴。

对照表的索引值，注意一下，后面的base64编码和解码会用到。

**Base64编码优缺点**

**优点**

- 1. 可以将二进制数据（比如图片）转化为可打印字符，方便传输数据
- 2. 对数据进行简单的加密，肉眼是安全的
- 3. 如果是在html或者css处理图片，可以减少http请求

**缺点**

- 1. 内容编码后体积变大， 至少1/3  
因为是三字节变成四个字节，当只有一个字节的时候，也至少会变成三个字节。
- 2. 编码和解码需要额外工作量

说完优缺点，回到正题：

我们今天的重点是 utf8编码转Base64编码：

**基本流程**

char => 码点 => utf-8编码 => base64编码

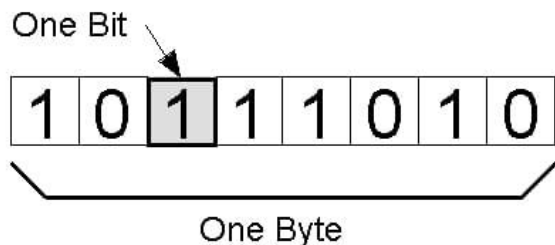
在之前要解一下编码的知识, 了解编码知识, 又要先了解一些计算机的基础知识。

## 一些计算机和前端基础知识

### 比特和字节

比特又叫位。在计算机的世界里, 信息的表示方式只有 0 和 1, 其可以表示两种状态。一位二进制可以表示两状态, N位可以表示 $2^N$ 种状态。

一个字节(Byte)有8位(Bit)



所以一个字节可以表示  $2^8 = 256$  种状态;

### 获得字符的 Unicode码点

[String.prototype.charCodeAt](#) 可以获取字符的码点, 获取范围为0 ~ 65535。这个地方注意一下, 关系到后面的utf-8字节数。

```
"a".charCodeAt(0) // 97
"中".charCodeAt(0) // 20013
```

### 进制表示

1. 0b开头, 可以表示二进制 注意0b10000000= 128 ,0b11000000=92, 之后会用到.

```
0b11111111 // 255
0b10000000 // 128 后面会用到
0b11000000 // 192 后面会用到
```

```
> console.log(0b11111111)
255
```

1. 0x开头, 可以表示16进制

```
0x11111111 // 286331153
```

```
> console.log(0x11111111)
286331153
```

0o开头可以表示8进制, 就不多说了, 本来不会涉及。

### 进制转换

#### 10进制转其他进制

[Number.prototype.toString\(radix\)](#)可以把十进制转为其他进制。

```
100..toString(2) // 1100100
100..toString(16) // 64, 也等于 0x64
```

## 其他进制转为10进制

[parseInt\(string, radix\)](#)可以把其他进制，转为10进制。

```
parseInt("10000000", 2) // 128
parseInt("10", 16) // 16
```

这里额外提一下一元操作符号+可以把字符串转为数字，后面也会用到,之前提到的0b,0o,0x这里都会生效。

```
+"1000" // 1000
+"0b10000000" // 128
+"0o10" // 8
+"0x10" // 16
```

## 位移操作

本文只涉及右移操作，就只讲右移，右移相当于除以2，如果是整数，简单说是去低位，移动几位去掉几位，其实和10进制除以10是一样的。

64 >> 2 = 16 我们一起看一下过程

```
0 1 0 0 0 0 0 0      64
-----
0 1 0 0 0 0 0 | 0 0  16
```

## 一元 & 操作和 一元 | 操作

### 一元&

当两者皆为1的时候，值为1。本文的作用可用来去高位，具体看代码。

3553 & 36 = 0b110111100001 & 0b111111 = 100001

因为高位缺失，不可能都为1，故均为0，而低位相当于复制一遍而已。

```
110111 100001
      111111
-----
000000 100001
```

### 一元|

当任意一个为1，就输出为1。本文用来填补0。比如，把3补成8位二进制

3 | 256 = 11 | 100000000 = 100000011

100000011.substring(1)是不是就等于8位二进制呢00000011

具备了这些基本知识，我们就开始先了解编码相关的知识。

## ASCII码，Unicode，UTF-8

### ASCII码

ASCII码第一位始终是0，那么实际可以表示的状态是  $2^7 = 128$  种状态。

ASCII 主要用于显示现代英文，到目前为止只定义了 128 个字符，包含控制字符和可显示字符。

- 0~31 之间的ASCII码常用于控制像打印机一样的外围设备
- 32~127 之间的ASCII码表示的符号，在我们的键盘上都可以被找到

完整的 ASCII码对应表，可以参见 [基本ASCII码和扩展ASCII码](#)

接下来是Unicode和UTF-8编码，请先记住这个重要的知识： - Unicode: 字符集 - UTF-8: 编码规则

Unicode

Unicode 为世界上所有字符都分配了一个唯一的编号(码点)，这个编号范围从 0x000000 到 0x10FFFF (十六进制)，有 100 多万，每个字符都有一个唯一的 Unicode 编号，这个编号一般写成 16 进制，在前面加上 U+。例如：掘的 Unicode 是U+6398。

- U+0000到U+FFFF 最前面的65536个字符位，它的码点范围是从0一直到 $2^{16}-1$ 。所有最常见的字符都放在这里。
- U+010000一直到U+10FFFF 剩下的字符都放着这里，码点范围从U+010000一直到U+10FFFF。

Unicode有平面的概念，这里就不拓展了。

Unicode只规定了每个字符的码点，到底用什么样的字节序表示这个码点，就涉及到编码方法。

UTF-8

UTF-8 是互联网使用最多的一种 Unicode 的实现方式。还有 UTF-16（字符用两个字节或四个字节表示）和 UTF-32（字符用四个字节表示）等实现方式。

UTF-8 是它是一种变长的编码方式, 使用的字节个数从 1 到 4 个不等，最新的应该不止4个， 这个1-4不等，是后面编码和解码的关键。

UTF-8的编码规则：

1. 对于只有一个字节的符号，字节的第一位设为0，后面 7 位为这个符号的 Unicode 码。此时，对于英语字母UTF-8 编码和 ASCII 码是相同的。
2. 对于 n 字节的符号 (n > 1)，第一个字节的前 n 位都设为 1，第 n + 1 位设为0，后面字节的前两位一律设为 10。剩下的没有提及的二进制位，全部为这个符号的 Unicode 码，如下表所示：

Unicode 码点范围（十六进制）	十进制范围	UTF-8 编码方式（二进制）	字节数
0000 0000 ~ 0000 007F	0 ~ 127	0xxxxxxx	1
0000 0080 ~ 0000 07FF	128 ~ 2047	110xxxxx 10xxxxxx	2
0000 0800 ~ 0000 FFFF	2048 ~ 65535	1110xxxx 10xxxxxx 10xxxxxx	3
0001 0000 ~ 0010 FFFF	65536 ~ 1114111	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4

我们可能没见过字节数为2或者为4的字符， 字节数为2的可以去[Unicode对应表](#)这里找，而等于4的可以去这看看[Unicode® 13.0 Versioned Charts Index](#)

下面这些码点都处于0000 0080 ~ 0000 07FF, utf-8编码需要2个字节

U+1F00 to U+2000							
	0	1	2	3	4	5	6
UNICODE: 1F00	ā	á	â	ã	ä	å	ă
UNICODE: 1F01	ē	ê	ë	ì	í	î	ï
UNICODE: 1F02	ñ	ò	ó	ô	õ	ö	÷
UNICODE: 1F03	ı	ġ	ĥ	ĵ	ĥ	ı	ı
UNICODE: 1F04	ó	ó	ô	ô	ö	ö	ı
UNICODE: 1F05	ı	ı	ı	ı	ı	ı	ı
UNICODE: 1F06	ı	ı	ı	ı	ı	ı	ı
UNICODE: 1F07	ı	ı	ı	ı	ı	ı	ı
UNICODE: 1F08	ı	ı	ı	ı	ı	ı	ı
UNICODE: 1F09	ı	ı	ı	ı	ı	ı	ı
UNICODE: 1FA	ı	ı	ı	ı	ı	ı	ı
UNICODE: 1FB	ı	ı	ı	ı	ı	ı	ı

下面这些码点都处于0001 0000 ~ 0010 FFFF, utf-8编码需要4个字节



2F801 丸 T6-2131 ≡ 4E38 丸 ~ 4E38 FE00	2F80E 免 T6-2352 → 5145 充 ≡ 2063A 充 ~ 2063A FE00	2F81A 冬 T4-210F ≡ 4ECC 冬 ~ 4ECC FE00
2F802 ㇏ T6-2121 ≡ 4E41 ㇏ ~ 4E41 FE00	2F80F 免 T3-2452 ≡ 514D 免 ~ 514D FE01	2F81B 况 T6-223C ≡ 51AC 况 ~ 51AC FE00
2F803 回 T6-2508 ≡ 20122 回 ~ 20122 FE00	2F810 𧯛 T3-2758 ≡ 5154 𧯛 ~ 5154 FE00	2F81C 𧯛 T3-2441 ≡ 51B5 𧯛 ~ 51B5 FE01
2F804 你 你 T6-2572 ≡ 4F60 你 ~ 4F60 FE00	2F811 具 T6-2754 ≡ 5177 具 ~ 5177 FE00	2F81D 𧯛 T7-307A ≡ 291DF 𧯛 ~ 291DF FE00
2F805 侮 侮 T4-2530 ≡ 4FAE 侮 ~ 4FAE FE01	2F812 𧯛 T6-303C ≡ 2051C 𧯛	2F81E 刃 T6-2108 ≡ 5203 刃 ~ 5203 FE00
2F806 倪 T6-2572 ≡ 4FAE 倪 ~ 4FAE FE01		

可能这里光说不好理解，我们分别以英文字符a和中文字符掘来讲解一下：

为了验证结果，可以去 [Convert UTF8 to Binary Bits - Online UTF8 Tools](#)

### 英文字符a

1. 先获得其码点，`"a".charCodeAt(0)` 等于 97
2. 对照表格，0~127, 需1个字节
3. `97..toString(2)` 得到编码 1100001
4. 根据格式0xxxxxxx进行填充，最终结果

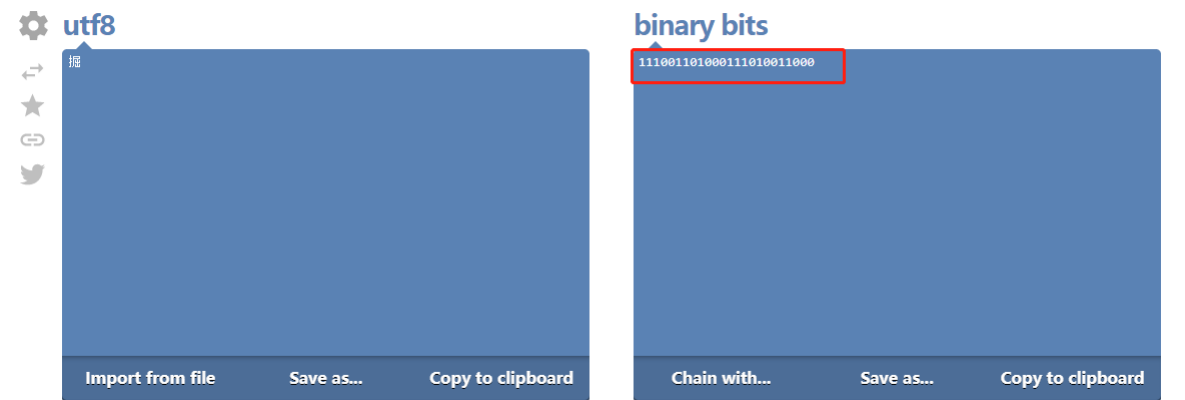
01100001

### 中文字符掘

1. 先获得其码点，`"掘".charCodeAt(0)` 等于 25496
2. 对照表格，2048~65535 需3个字节
3. `25496..toString(2)` 得到编码 110 001110 011000
4. 根据格式1110xxxx 10xxxxxx 10xxxxxx进行填充, 最终结果如下

11100110 10001110 10011000

[Convert UTF8 to Binary Bits - Online UTF8 Tools](#)执行结果：完全匹配



### 抽象把字符转为utf8格式二进制的方法

基于上面的表格和转换过程，我们抽象一个方法，这个方法在之后的Base64编码和解码至关重要：

先看看功能，覆盖utf8编码1-3字节范围

```
console.log(to_binary("A")) // 11100001
console.log(to_binary("1101100010110011 // ("س
```



```
console.log(to_binary("掘")) // 111001101000111010011000
```

方法如下

```
function to_binary(str) {
  const string = str.replace(/\r\n/g, "\n");
  let result = "";
  let code;
  for (var n = 0; n < string.length; n++) {
    //获取码点
    code = str.charCodeAt(n);
    if (code < 0x007F) { // 1个字节
      // 0000 0000 ~ 0000 007F 0 ~ 127 1个字节

      // (code | 0b100000000).toString(2).slice(1)
      result += (code).toString(2).padStart(8, '0');
    } else if ((code > 0x0080) && (code < 0x07FF)) {
      // 0000 0080 ~ 0000 07FF 128 ~ 2047 2个字节
      // 0x0080 的二进制为 10000000, 8位, 所以大于0x0080的, 至少有8位
      // 格式 110xxxxx 10xxxxxx

      // 高位 110xxxxx
      result += ((code >> 6) | 0b11000000).toString(2);
      // 低位 10xxxxxx
      result += ((code & 0b111111) | 0b10000000).toString(2);
    } else if (code > 0x0800 && code < 0xFFFF) {
      // 0000 0800 ~ 0000 FFFF 2048 ~ 65535 3个字节
      // 0x0800的二进制为 1000 00000000, 12位, 所以大于0x0800的, 至少有12位
      // 格式 1110xxxx 10xxxxxx 10xxxxxx

      // 最高位 1110xxxx
      result += ((code >> 12) | 0b11100000).toString(2);
      // 第二位 10xxxxxx
      result += (((code >> 6) & 0b111111) | 0b10000000).toString(2);
      // 第三位 10xxxxxx
      result += ((code & 0b111111) | 0b10000000).toString(2);
    } else {
      // 0001 0000 ~ 0010 FFFF 65536 ~ 1114111 4个字节
      // https://www.unicode.org/charts/PDF/Unicode-13.0/U130-2F800.pdf
      throw new TypeError("暂不支持码点大于65535的字符")
    }
  }
  return result;
}
```

方法中有三个地方稍微难理解一点, 我们一起来解读一下:

1. 二字节 (`code >> 6`) | `0b11000000` 其作用是生成高位二进制。

我们以实际的一个栗子来讲解, 以<sub>ω</sub>为例, 其码点为0x633,在0000 0080 ~ 0000 07FF之间, 占两个字节, 在其二进制编码为11 000110011, 其填充格式如下, 低位要用6位。

js 110xxxxx 10xxxxxx 为了方便观察, 我们把 11 000110011 重新调整一下 11000 110011。

(code >> 6) 等于 00110011 >> 6, 右移6位, 直接干掉低6位。为什么是6呢, 因为低位需要6位, 右移动6位后, 剩下的就是用于高位操作的位了。`js 11000000 11000 | 110011`

11011000  
```

## 1. 二字节 (code & 0b111111) | 0b10000000

作用，用于生成低位二进制。以ω为例，11000 110011, 填充格式

js 110xxxxx 10xxxxxx (code & 0b111111) 这步的操作是为了干掉6位以上的高位，仅仅保留低6位。一元&符号，两边都是1的时候才会是1，妙啊。

```
``js 11000 110011 111111
```

```
110011
```

`` 接着进行| 0b10000000，主要是按照格式10xxxxxx`进行位数填补，让其满8位。

```
11000 110011
      111111      (code & 0b111111)
-----
      110011
10 000000      (code & 0b111111) | 0b10000000
-----
10 110011
```

## Base64编码和解码

### utf-8转Base64编码规则

1. 获取每个字符的Unicode码，转为utf-8编码
2. 三个字节作为一组，一共是24个二进制位  
字节数不能被3整除，用0字节值在末尾补足
3. 按照6个比特位一组分组，前两位补0，凑齐8位
4. 计算每个分组的数值
5. 以第4步的值作为索引，去ASCII码表找对应的值
6. 替换第2步添加字节数个数的 =  
比如第2步添加了2个字节，后面是2个=

以大掘A为例，我们通过上面的utf8\_to\_binary方法得到utf8的编码

11100110 10001110 10011000 11000001, 其字节数不能被3整除，后面填补

```
11100110
10001110
10011000
01000001
-----
00000000
00000000
```

6位一组分为四组，高位补0, 用| 分割一下填补的。

```
00 | 111001 => 57 => 5
00 | 101000 => 40 => 0
00 | 111010 => 58 => 6
00 | 011000 => 24 => Y

00 | 110000 => 16 => Q
00 | 010000 => 16 => Q
00 | 000000 =>    => =
00 | 000000 =>    => =
```

结果是：506YQQ==，完美。

### utf-8转Base64编码规则代码实现

基于上面的to\_binary方法和base64的转换规则，就很简单啦：  
先看看执行效果，very good, 和 [base64.us](https://base64.us) 结果完全一致。

```
console.log(utf8_to_base64("a")); // YQ==

console.log(utf8_to_base64("Â")); // yII=

console.log(utf8_to_base64("中国人")); // 5Lit5Zu95Lq6

console.log(utf8_to_base64("Coding Writing 好文召集令 | 后端、大前端双赛道投稿，2万元奖金！"));
// Q29kaW5nIFdyaXRpbmcg5aW95paH5Y+s6ZuG5Luk772c5ZC056uv44CB5aSn5YmN56uv5Y+M6LWb6
```

完整代码如下：

```
const BASE64_CHARTS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=';

function utf8_to_base64(str: string) {
  let binaryStr = to_binary(str);
  const len = binaryStr.length;

  // 需要填补的=的数量
  let paddingCharLen = len % 24 !== 0 ? (24 - len % 24) / 8 : 0;

  // 6个一组
  const groups = [];
  for (let i = 0; i < binaryStr.length; i += 6) {
    let g = binaryStr.slice(i, i + 6);
    if (g.length < 6) {
      g = g.padEnd(6, "0");
    }
    groups.push(g);
  }

  // 求值
  let base64Str = groups.reduce((b64str, cur) => {
    b64str += BASE64_CHARTS[+`0b${cur}`];
    return b64str;
  }, "");

  // 填充=
  if (paddingCharLen > 0) {
    base64Str += paddingCharLen > 1 ? "==" : "=";
  }

  return base64Str;
}
```

至于解码，是其逆过程，留给大家去实现吧。

## 其他的成熟方案

1. 当然是基于已有的 btoa和atob, 但是 unescape是不被推荐使用的方法

```
function utf8_to_b64( str ) {
  return window.btoa(unescape(encodeURIComponent( str )));
}

function b64_to_utf8( str ) {
  return decodeURIComponent(escape(window.atob( str )));
}
```

```
return decodeURIComponent(escape(window.atob( str )));
}
```

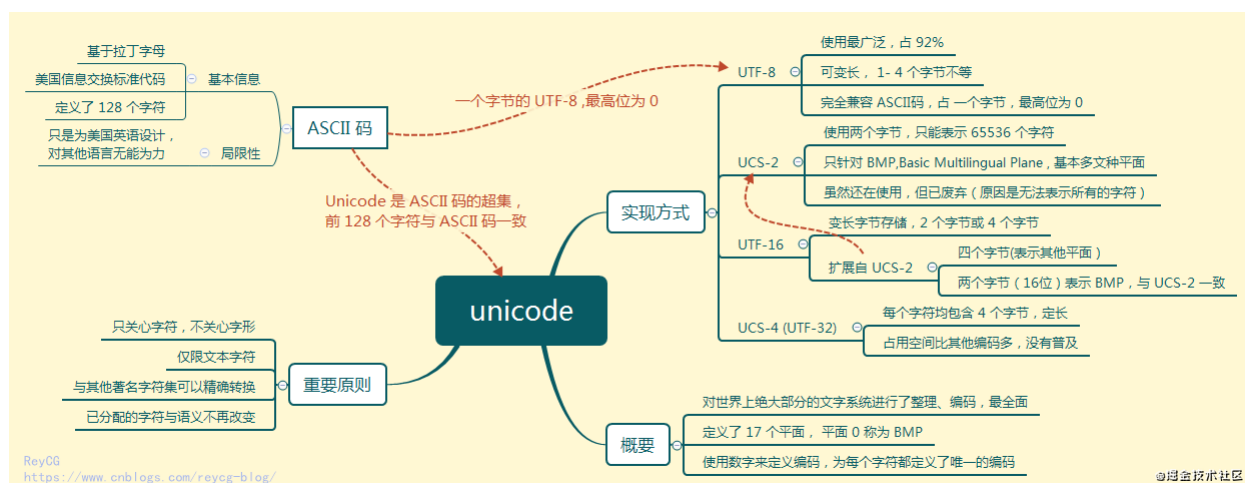
// Usage: utf8\_to\_b64('✓ à la mode'); // "4pyTIMOgIGxhIG1vZGU=" b64\_to\_utf8('4pyTIMOgIGxhIG1vZGU='); // "✓ à la mode"

1. MDN的 [rewriting atob\(\) and btoa\(\) using TypedArrays and UTF-8](#) 其支持到6字节，但是可读性并不好。
2. 第三方库 [base64-js](#) 与 [js-base64](#) 都是周下载量过百万的库。

虽然有那么多成熟的，但是我们理解和自己实现，才能更明白Base64的编码原理。

额外补充一点

1. 编码关系图 借用[[你真的了解 Unicode 和 UTF-8 吗?](#)]一张图：



1. `DOMString` 是 utf-16 编码

写在最后

写作不易，你的三连（一赞，一评，一收藏），就是我最大的动力。

点赞过百，再写篇浏览器，DOM, JS等关于编码的文章。

引用

[Version-Specific Charts](#)  
[Unicode13.0.0](#)  
[Unicode® 13.0 Versioned Charts Index](#)  
[RFC 4648 | The Base16, Base32, and Base64 Data Encodings](#)  
[Base64 encoding and decoding](#)  
[字符编码笔记：ASCII, Unicode 和 UTF-8](#)  
[Unicode与JavaScript详解](#)  
[Base64 编码入门教程 Base64原理](#)  
[详解base64原理](#)  
[一文读懂base64编码](#)  
[JS 中关于 base64 的一些事](#)  
[Base64 的原理、实现及应用](#)  
[图片与Base64换算关系](#)  
[\[你真的了解 Unicode 和 UTF-8 吗?\]](#)  
[Unicode中UTF-8与UTF-16编码详解](#)  
[Unicode对应表](#)  
[JavaScript Source Map 详解](#)

【干货】私藏的这些高级工具函数，你拥有几个？

本文已参与好文召集令活动，点击查看：[后端、大前端双赛道投稿，2万元奖池等你挑战！](#)

## 前言

很多功能，其实内置的Web API已支持，  
比如基于URLSearchParams或者URL的queryString获取和生成  
比如基于btoa,atob的base64的编码和解码  
比如基于sendBeacon的数据上报  
比如基于Array.from的序列生成 \* 比如基于canvas的视频截图 \* 比如基于URL的UUID生成

我们用精简的代码来实现相对复杂的功能，没有第三方库，你也能秀得飞起。

本文收录在 [基础进阶](#) 专栏，欢迎关注和收藏，往期经典： \* [三千文字，也没写好 Function.prototype.call](#) 200+ star \* [那些你熟悉而又陌生的函数](#) 200 star  
\* [这16种原生函数和属性的区别，你真的知道吗？](#) 精心收集，高级前端必备知识，快快打包带走 96 star

目录(方便移动端阅读) \* localStorage的已使用空间 \* 带图带事件的桌面通知 \* 原生30行代码实现视频截图 \* 基于URLSearchParams获取queryString的值 \* 基于atob和btoa的base64编码和解码 \* 非正则替换html代码encode和decode \* 相对地址转换为绝对地址 \* 基于URL或者Crypto.getRandomValues生成UUID \* 基于Array.from的序列生成器 \* 基于sendBeacon的安全的数据上报 \* 基于toLocaleString千分位 \* Promise顺序执行 \* 延时执行delay \* 进度值映射 \* 滑滚动页面到顶部 \* 禁止选择和复制 \* 禁止图片拖拽 \* 自增长ID

## localStorage的已使用空间

在较新的chrome上测试，localStorage的存储是按照字符个数来算的。包含键和值的。  
所以在测试代码中，你把a修改啊，不会影响存储的数量。但是键的长度，会影响存储的数量。

## 代码

```
function getLSUsedSpace() {
  return Object.keys(localStorage).reduce((total, curKey) => {
    if (!localStorage.hasOwnProperty(curKey)) {
      return total;
    }
    total += localStorage[curKey].length + curKey.length;
    return total;
  }, 0)
}
```

## 示例

```
localStorage.clear();
localStorage.a = "啊";
console.log(getLSUsedSpace()); // 2
```

## 溢出测试：

key的值为长度为10的 kkkkkkkkkk:  
输出结果：Max: 5242880 value Length: 5242870  
当你把key修改长度为1的k:  
输出结果：Max: 5242880 value Length: 5242879

```
localStorage.clear();
let valLength = 0
try {
  let str = Array.from({ length: 5242800 }, () => "啊").join("");
  valLength = str.length;
  for (let i = 0; i < 100000000000000; i++) {
    str += "a"
  }
}
```

```

    valLength += 1;
    localStorage.setItem(`kkkkkkkkkk`, str);
}

} catch (err) {
    console.error("存储失败", err);
    console.log("Max:", getLSUsedSpace(), " value Length:", valLength)
}

```

✖ Expression  
not available

✖ ▶ 存储失败 DOMException: Failed to execute 'setItem' on 'Storage': Setting the value of 'kkkkkkkkkk' exceeded the quota. [ls.html:29](#)  
 at <http://127.0.0.1:5500/%E9%AB%98%E6%95%88%E7%A7%81%E6%88%BF%E5%87%BD%E6%95%B0/ls.html:24:22>  
 Max: 5242880 value Length: 5242870 [ls.html:30](#)

掘金技术社区

## 注意

### 1. 超过存储上线是会报错的:

✖ Uncaught DOMException: Failed to execute 'setItem' on 'Storage': Setting the value of '764855' exceeded the quota.  
 at <http://127.0.0.1:5500/%E9%AB%98%E6%95%88%E7%A7%81%E6%88%BF%E5%87%BD%E6%95%B0/index.html:58:26>

掘金技术社区

### 1. 如何捕获错误,可以参考 MDN [testing\\_for\\_availability](#)

大致是对Error的错误码和name进行判断

```

e instanceof DOMException && (
    // everything except Firefox
    e.code === 22 ||
    // Firefox
    e.code === 1014 ||
    // test name field too, because code might not be present
    // everything except Firefox
    e.name === 'QuotaExceededError' ||
    // Firefox
    e.name === 'NS_ERROR_DOM_QUOTA_REACHED') &&
    // acknowledge QuotaExceededError only if there's something already
    (storage && storage.length !== 0);

```

## 参考引用

[calculating-usage-of-localstorage-space](#)

[what-is-the-max-size-of-localstorage-values](#)

[Test of localStorage limits/quota](#)

## 带图带事件的桌面通知

网页也可以以桌面弹框的形式进行通知,先看个效果图:

有头像,有标题,有文本,点击消息通知还能让窗体聚焦,真帅。



## 代码

```
function doNotify(title, options = {}, events = {}) {
  const notification = new Notification(title, options);
  for (let event in events) {
    notification[event] = events[event];
  }
}

function notify(title, options = {}, events = {}) {
  if (!("Notification" in window)) {
    return console.error("This browser does not support desktop notification");
  }
  else if (Notification.permission === "granted") {
    doNotify(title, options, events);
  } else if (Notification.permission !== "denied") {
    Notification.requestPermission().then(function (permission) {
      if (permission === "granted") {
        doNotify(title, options, events);
      }
    });
  }
}
```

## 示例

tag还可以用去重消息。

```
notify("中奖提示", {
  icon: "https://sf1-ttcdn-tos.pstatp.com/img/user-avatar/fla9f122e92!",
  body: "恭喜你，掘金签到一等奖",
  tag: "prize"
}, {
  onclick(ev) {
    console.log(ev);
    ev.target.close();
    window.focus();
  }
})
```

## 参考引用

[notification](#)

[使用 Web Notifications](#)

原生30行代码实现视频截图



基本原理就是把视频画到Canvas里面，然后调用toDataURL或者toBlob，再利用a标签模拟点击，download属性指定名字。

看一下效果：



代码

```
function captureVideo(videoEl) {
  let canvasEl;
  let dataUrl;
  try {
    const cps = window.getComputedStyle(videoEl);
    const width = +cps.getPropertyValue("width").replace("px", "");
    const height = +cps.getPropertyValue("height").replace("px", "");

    canvasEl = document.createElement("canvas");
    canvasEl.style.cssText = `position:fixed;left:-9999px`;
    canvasEl.height = height;
    canvasEl.width = width;

    document.body.appendChild(canvasEl);

    const ctx = canvasEl.getContext("2d");
    ctx.drawImage(videoEl, 0, 0, width, height);
    // const image = canvas.toDataURL("image/png");
    dataUrl = canvasEl.toDataURL();

    document.body.removeChild(canvasEl);
    canvasEl = null;
    return dataUrl;
  } finally {
    if (canvasEl) {
      document.body.removeChild(canvasEl);
    }
    if (dataUrl) {
      return dataUrl;
    }
  }
}
```

示例

注意添加crossorigin="anonymous"，不然转为图片会失败。

```
<video id="videoEL" controls autoplay crossorigin="anonymous"
  src="https://api.dogecloud.com/player/get.mp4?vcode=5ac682e6f8231991&us
```

```
function download(url) {
    const aEl = document.createElement("a");
    aEl.href = url;
    aEl.download = "视频.png";
    aEl.click();
}

function doCaptureVideo() {
    const url = captureVideo(videoEL);
    download(url);
}

doCaptureVideo()
```

## 基于URLSearchParams或URL获取queryString的值

常用的方式是使用正则或者split方法，其实不然，URLSearchParams和URL都能很好的实现功能。

### 代码

```
const urlSP = new URLSearchParams(location.search);
function getQueryString(key) {
    return urlSP.get(key)
}

const urlObj = new URL(location.href);
function getQueryString(key) {
    return urlObj.searchParams.get(key)
}
```

### 示例

测试地址:

```
const log = console.log;
getQueryString

log("pid", getQueryString("pid")); // pid 10
log("cid", getQueryString("cid")); // cid null
```

### 参考引用

MDN文献: [URLSearchParams-MDN](#)

CanIUse兼容性: [URLSearchParams: 95.63%](#)

Polyfill: [url-search-params-polyfill](#)

## 基于atob和btoa的base64编码和解码

浏览器内置了base64编码和解码的能力，第三方库，不需要的。

### 代码

```
function utf8_to_b64( str ) {
    return window.btoa(unescape(encodeURIComponent( str )));
}

function b64_to_utf8( str ) {
    return decodeURIComponent(escape(window.atob( str )));
}
```

```
}
```

## 示例

```
utf8_to_b64('✓ à la mode'); // "4pyTIMOgIGxhIG1vZGU="
b64_to_utf8('4pyTIMOgIGxhIG1vZGU='); // "✓ à la mode"
```

## 参考引用

MDN文献: [atob](#), [btoa](#)

CanIUse兼容性: [btoa](#) 99.68%

Polyfill: [MDN Polyfill](#)

[Base64](#)

## 非正则替换的html代码encode和decode

常规的方式是使用正则替换，这里是另外一种思路。

## 代码

```
function htmlencode(s) {
    var div = document.createElement('div');
    div.appendChild(document.createTextNode(s));
    var result = div.innerHTML;
    div = null;
    return result;
}
function htmldecode(s) {
    var div = document.createElement('div');
    div.innerHTML = s;
    var result = div.innerText || div.textContent;
    div = null;
    return result;
}
```

## 示例

```
htmlencode("<div>3>5 & 666</div>"); // &lt;div&gt;3&gt;5 &amp; 666&lt;/div&gt;
htmldecode("&lt;div&gt;3&gt;5 &amp; 666&lt;/div&gt;") // <div>3>5 & 666</div>
```

## 相对地址转换为绝对地址

基于当前页面的相对地址转换为绝对地址。

## 代码

```
function relativeToAbs(href) {
    let aEl = document.createElement("a");
    aEl.href = href;

    const result = aEl.href;
    aEl = null;
    return result;
}
```

## 示例

```
console.log("relativeToAbs", relativeToAbs("../a/b/b/index.html"));
// relativeToAbs http://127.0.0.1:5500/a/b/b/index.html
```

## 基于URL或者Crypto.getRandomValues生成UUID

基于[URL.createObjectURL](#)或者[Crypto.getRandomValues](#)

URL.createObjectURL 产生的地址为 blob:https://developer.mozilla.org/cb48b940-c625-400a-a393-176c3635020b, 其后部分就是一个UUID

### 代码

方式一:

```
function genUUID() {
  const url = URL.createObjectURL(new Blob([]));
  // const uuid = url.split("/").pop();
  const uuid = url.substring(url.lastIndexOf('/') + 1);
  URL.revokeObjectURL(url);
  return uuid;
}
```

```
genUUID() // cd205467-0120-47b0-9444-894736d873c7
```

方式二:

```
function uuidv4() {
  return ([1e7]+-1e3+-4e3+-8e3+-1e11).replace(/[018]/g, c =>
    (c ^ crypto.getRandomValues(new Uint8Array(1))[0] & 15 >> c / 4).toString(16)
  );
}
```

```
uuidv4() // 38aa1602-ba78-4368-9235-d8703cdb6037
```

### 参考引用

[generating-uuids-at-scale-on-the-web-2877f529d2a2](#)

[collisions-when-generating-uuids-in-javascript](#)

## 基于Array.from的序列生成器

造有序数据，无序数据，等等。

### 代码

```
const range = (start, stop, step) => Array.from(
  { length: (stop - start) / step + 1 },
  (_, i) => start + (i * step)
);
```

### 示例

```
range(0, 4, 1); // [0, 1, 2, 3, 4]
range(0, 9, 3); // [0, 3, 6, 9]
range(0, 8, 2.5) // [0, 2.5, 5, 7.5]
```

## 基于sendBeacon的安全的数据上报

[sendBeacon](#) 异步地向服务器发送数据，同时不会延迟页面的卸载或影响下一导航的载入性能。

```
function report(url, data) {
  if (typeof navigator.sendBeacon !== "function") {
    return console.error("sendBeacon不被支持");
  }
  navigator.sendBeacon(url, data);
}
```

}

## 示例

```
window.addEventListener('unload', logData, false);
function logData() {
    report("/log", "被卸载了");
}
```

## 基于toLocaleString千分位

正则？遍历？不需要的。内置函数就解决。当然，如果是超大的数，可能是会有问题的。

## 代码

```
function formatMoney(num) {
    return (+num).toLocaleString("en-US");
}
```

## 示例

```
console.log(formatMoney(123456789)); // 123,456,789
console.log(formatMoney(6781)) // 6,781
console.log(formatMoney(5)) // 5
```

## 超大的数

```
formatMoney(199999999333333333333333) // 19,999,999,933,333,333,000,000
```

## Promise顺序执行

让Promise顺序的执行，并支持初始化参数和结果作为参数传递。

## 代码

```
function runPromises(promiseCreators, initData) {  
  return promiseCreators  
    .reduce((promise, next) => promise  
      .then((data) => next(data))  
    , Promise.resolve(initData));  
}
```

## 示例

```
var promise1 = function (data = 0) {
  return new Promise(resolve => {
    resolve(data + 1000);
  });
}

var promise2 = function (data) {
  return new Promise(resolve => {
    resolve(data - 500);
  });
}

runPromises([promise1, promise2], 1).then(res=>console.log(res));
```

## 延时执行delay

延时执行某函数，且只会执行一次。

## 代码

```
function delay(fn = () => { }, delay = 5000, context = null) {
  let ticket = null;
  let runned = false;
  return {
    run(...args) {
      return new Promise((resolve, reject) => {
        if (runned === true) {
          return;
        }
        runned = true;
        ticket = setTimeout(async () => {
          try {
            const res = await fn.apply(context, args);
            resolve(res);
          } catch (err) {
            reject(err)
          }
        }, delay)
      })
    },
    cancel: () => {
      clearTimeout(ticket);
    }
  }
}
```

## 示例

```
delay(function () {
  console.log("你们好");
}).run();

const { run, cancel } = delay(function (name) {
  console.log("你好: ", name);
});

run("吉他");
run("吉他");

// 你们好
// 你好:  吉他
```

## 进度值映射

进度映射，比较只有 10%的进度，确要显示50%的进度的场景。

## 代码

```
function adjustProgress(progress: number, mapping: { real: number; target: number }) {
  if (progress < 0) {
    return 0;
  }
  if (!mapping || mapping.length <= 0) {
```

```

    return progress;
}
// 第一个
const f = mapping[0];
if (progress <= f.real) {
    return progress * (f.target / f.real);
}
// 最后一个
const l = mapping[mapping.length - 1];
if (progress >= l.target) {
    return l.target;
}
const curIndex = mapping.findIndex(m => m.real >= progress);
if (!curIndex) {
    return progress;
}
const cur = mapping[curIndex];
const pre = mapping[curIndex - 1];
// 原基数 + 实际进度/最大实际进度 * 期望间距
return pre.target + (progress - pre.real) / (cur.real - pre.real) * (cur.ta:
}

```

## 示例

```

const mapping = [{
  real: 0,
  target: 0,
}, {
  real: 30,
  target: 50
}, {
  real: 60,
  target: 80
}, {
  real: 100,
  target: 100
}];

console.log("15", adjustProgress(15, mapping)); // 15 25
console.log("25", adjustProgress(25, mapping)); // 25 41.666666666666667
console.log("50", adjustProgress(50, mapping)); // 50 70
console.log("60", adjustProgress(60, mapping)); // 60 80
console.log("100", adjustProgress(100, mapping)); // 100 100

```

## 滑滚动页面到顶部

### 代码

PC端滚动的根元素是document.documentElement,  
 移动端滚动的的根元素是document.body,  
 有一个更好的属性document.scrollingElement能自己识别文档的滚动元素, 其在PC端等于document.documentElement, 其在移动端等于document.body

```

// smooth 选项在Safari上支持不好
function scrollToTop() {
  window.scrollTo({

```



```

        left: 0,
        top: 0,
        behavior: 'smooth'
    })
}

function scrollToTop() {
    let scrollTop = document.documentElement.scrollTop || document.body.scrollTop
    if (scrollTop > 0) {
        window.requestAnimationFrame(scrollToTop);
        window.scrollTo(0, scrollTop - scrollTop / 8);
    }
};

```

## 禁止选择和复制

### 代码

```

['contextmenu', 'selectstart', 'copy'].forEach(function(ev) {
    document.addEventListener(ev, function(ev) {
        ev.preventDefault();
        ev.returnValue = false;
    })
});

```

当然也有CSS方案

```

body {
    -moz-user-select: none;
    -webkit-user-select: none;
    -ms-user-select: none;
    -khtml-user-select: none;
    user-select: none;
}

```

## 禁止图片拖拽

### 代码

```

['dragstart'].forEach(function(ev) {
    document.addEventListener(ev, function(ev) {
        ev.preventDefault();
        ev.returnValue = false;
    })
});

```

## 自增长ID

自己生产自增长的ID值，当然可以更复杂一些。

### 代码

```

let id = 0;
function getId() {
    return id++;
}

```

### 示例

```
console.log(getId()); // 1
console.log(getId()); // 2
```

## 写在后面

写作不易，你的一赞一评，就是我前行的最大动力。

## 三千文字，也没写好 `Function.prototype.call`

### 前言

`Function.prototype.call`，手写系列，万文面试系列，必会系列必包含的内容，足见其在前端的分量。

本文基于MDN 和 ECMA 标准，和大家一起从新认识`call`。

涉及知识点：1. [undefined](#) 2. [void 一元运算符](#)

3. [严格模式](#)和非严格模式 4. 浏览器和nodejs环境识别 5. 函数副作用（纯函数） 6. [eval](#) 7. [Content-Security-Policy](#) 8. [delete](#) 9. [new Function](#)

10. [Object.freeze](#) 11. 对象属性检查 12. 面试现场 13. ECMA规范和浏览器厂商之间的爱恨情仇

### 掘金流行的版本

面试官的问题：

麻烦你手写一下`Function.prototype.call`

### 基于ES6的拓展运算符版本

```
Function.prototype.call = function(context) {
  context = context || window;
  context["fn"] = this;
  let arg = [...arguments].slice(1);
  context["fn"](...arg);
  delete context["fn"];
}
```

这个版本，应该不是面试官想要的真正答案。不做太多解析。

### 基于eval的版本

```
Function.prototype.call = function (context) {
  context = (context == null || context == undefined) ? window : new Object(context);
  context.fn = this;
  var arr = [];
  for (var i = 1; i < arguments.length; i++) {
    arr.push('arguments[' + i + ']');
  }
  var r = eval('context.fn(' + arr + ')');
  delete context.fn;
  return r;
}
```

这个版本值得完善的地方 1. `this` 是不是函数没有进行判断 2. 使用`undefined`进行判断，安全不安全 `undefined` 可能被改写，（高版本浏览器已做限制）。 3. 直接使用`window`作为默认上下文，过于武断。

脚本运行环境，浏览器？ nodejs？

函数运行模式，严格模式，非严格模式？ 4. `eval` 一定会被允许执行吗 5. `delete context.fn` 有没有产生副作用 `context`上要是原来有`fn`属性呢

在我们真正开始写`Function.prototype.call`之前，还是先来看看MDN和 ECMA是怎么定义她的。

### MDN `call` 的说明

## 语法

```
function.call(thisArg, arg1, arg2, ...)
```

## 参数

thisArg

可选的。在 function 函数运行时使用的 this 值。请注意，this可能不是该方法看到的实际值：如果这个函数处于非严格模式下，则指定为 null 或 undefined 时会自动替换为指向全局对象，原始值会被包装。

arg1, arg2, ...  
指定的参数列表。

## 透露的信息

这里透露了几个信息，我已经加粗标注：1. 非严格模式，对应的有严格模式 2. 这里说的是指向 全局对象，没有说是window。当然MDN这里说是window也没太大问题。我想补充的是 nodejs 也实现了 ES标准。所以我们实现的时候，是不是要考虑到 nodejs环境呢。3. 原始值会被包装。怎么个包装呢，Object(val)，即完成了对原始值val的包装。

## ES标准

在 [Function.prototype.call\(\) - JavaScript | MDN](#)的底部罗列了ES规范版本，每个版本都有call实现的说明。

我们实现的，是要基于ES的某个版本来实现的。

因为ES的版本不同，实现的细节可能不一样，实现的环境也不一样。

| 规范版本                                                    | 状态              | 说明                         |
|---------------------------------------------------------|-----------------|----------------------------|
| <a href="#">ECMAScript 1st Edition (ECMA-262)</a>       | Standard        | 初始定义。在 JavaScript 1.3 中实现。 |
| <a href="#">ECMAScript 5.1 (ECMA-262)</a>               | Standard        |                            |
| <a href="#">Function.prototype.call</a>                 |                 |                            |
| <a href="#">ECMAScript 2015 (6th Edition, ECMA-262)</a> | Standard        |                            |
| <a href="#">Function.prototype.call</a>                 |                 |                            |
| <a href="#">ECMAScript (ECMA-262)</a>                   | Living Standard |                            |
| <a href="#">Function.prototype.call</a>                 |                 |                            |

在ES3标准中关于call的规范说明在11.2.3 Function Calls, 直接搜索就能查到。

我们今天主要是基于2009年ES5标准下来实现Function.prototype.call，有人可能会说，你这，为嘛不在 ES3标准下实现，因为ES5下能涉及更多的知识点。

## 不可靠的undefined

```
(context == null || context == undefined) ? window : new Object(context)
```

上面代码的 undefined 不一定是可靠的。

引用一段MDN的话：

在现代浏览器（JavaScript 1.8.5/Firefox 4+），自ECMAScript5标准以来undefined是一个不能被配置（non-configurable），不能被重写（non-writable）的属性。即便事实并非如此，也要避免去重写它。

在没有交代上下文的情况使用 void 0 比直接使用 undefined 更为安全。

有些同学可能没见过undefined被改写的情况，没事，来一张图：

```

undefined = 10;
console.log("undefined:", undefined);
console.log("undefined == 10:", undefined == 10);
console.log("void 0:", void 0);

```

|                       |
|-----------------------|
| undefined             |
| undefined: 10         |
| undefined == 10: true |
| void 0: undefined     |

void 这个一元运算符除了这个 准备返回 undefined外， 还有另外两件常见的用途： 1. a标签的href， 就是什么都不做

```
<a href="javascript:void(0);">
```

## 1. IIFE立即执行

```

;void function(msg) {
    console.log(msg)
} ("你好啊");

```

当然更直接的方式是:

```

;(function(msg) {
    console.log(msg)
}) ("你好啊");

```

## 浏览器和nodejs环境识别

浏览器环境:

```
typeof self == 'object' && self.self === self
```

nodejs环境:

```
typeof global == 'object' && global.global === global
```

现在已经有 [globalThis](#), 在高版本浏览器和nodejs里面都支持。

显然，在我们的这个场景下，还不能用，但是其思想可以借鉴：

```

var getGlobal = function () {
    if (typeof self !== 'undefined') { return self; }
    if (typeof window !== 'undefined') { return window; }
    if (typeof global !== 'undefined') { return global; }
    throw new Error('unable to locate global object');
};

```

## 严格模式

### 是否支持严格模式

[Strict mode](#) 严格模式，是ES5引入的特性。那我们怎么验证你的环境是不是支持严格模式呢？

```

var hasStrictMode = (function() {
    "use strict";
    return this == undefined;
})();

```

正常情况都会返回true,放到IE8里面执行：

```
var hasStrictMode = (function () {
    "use strict";
    return this == undefined;
})();
console.log("hasStrictMode:", hasStrictMode);
undefined
hasStrictMode: false
```

在非严格模式下，函数的调用上下文（this的值）是全局对象。在严格模式下，调用上下文是undefined。

## 是否处于严格模式下

知道是不是支持严格模式，还不够，我们还要知道我们是不是处于严格模式下。

如下的代码可以检测，是不是处于严格模式：

```
var isStrict = (function(){
    return this === undefined;
})();
```

这段代码在支持严格模式的浏览器下和nodejs环境下都是工作的。

## 函数副作用

```
var r = eval('context.fn(' + arr + ')');
delete context.fn;
```

如上的代码直接删除了context上的fn属性，如果原来的context上有fn属性，那会不会丢失呢？

我们采用eval版本的call, 执行下面的代码

```
var context = {
    fn: "i am fn",
    msg: "i am msg"
}

log.call(context); // i am msg

console.log("msg:", context.msg); // i am msg
console.log("fn:", context.fn); // fn: undedined
```

可以看到context的fn属性已经被干掉了，是破坏了入参，产生了不该产生的副作用。  
与副作用对应的是函数式编程中的 纯函数。

对应的我们要采取行动，基本两种思路： 1. 造一个不会重名的属性 2. 保留现场然后还原现场

都可以，不过觉得 方案2更简单和容易实现：  
基本代码如下：

```
var ctx = new Object(context);

var propertyName = "__fn__";
var originVal;
var hasOriginVal = ctx.hasOwnProperty(propertyName)
if(hasOriginVal){
    originVal = ctx[propertyName]
}

..... // 其他代码
```

```

if(hasOriginVal){
    ctx[propertyVal] = originVal;
}

```

基于eval的实现，基本如下

基于标准[ECMAScript 5.1 \(ECMA-262\) Function.prototype.call](#)

When the call method is called on an object func **with** argument thisArg and optio

1. If IsCallable(func) is **false**, then **throw** a TypeError exception.
  2. Let argList be an empty List.
  3. If **this** method was called **with** more than one argument then **in** left to right order starting **with** arg1 append each argument **as** the last element **of** argList
  4. Return the result **of** calling the **[[Call]]** internal method **of** func, providing thisArg **as** the **this** value and argList **as** the list **of** arguments.
- The length property **of** the call method is 1.

NOTE The thisArg value is passed without modification **as** the **this** value. This is change **from** Edition 3, where a **undefined** or **null** thisArg is replaced **with** the global object and ToObject is applied to all other values and that result is passed **as** the **this** value.

对我们比较重要的是 1和 Note:

看看我们的基础实现

```

var hasStrictMode = (function () {
    "use strict";
    return this == undefined;
})();

var isStrictMode = function () {
    return this === undefined;
};

var getGlobal = function () {
    if (typeof self !== 'undefined') { return self; }
    if (typeof window !== 'undefined') { return window; }
    if (typeof global !== 'undefined') { return global; }
    throw new Error('unable to locate global object');
};

function isFunction(fn){
    return typeof fn === "function";
}

function getContext(context) {

    var isStrict = isStrictMode();

    if (!hasStrictMode || (hasStrictMode && !isStrict)) {
        return (context === null || context === void 0) ? getGlobal() : Object(c

```

```

}

// 严格模式下，妥协方案
return Object(context);
}

Function.prototype.call = function (context) {

    // 不可以被调用
    if (typeof this !== 'function') {
        throw new TypeError(this + ' is not a function');
    }

    // 获取上下文
    var ctx = getContext(context);

    // 更为稳妥的是创建唯一ID，以及检查是否有重名
    var propertyName = "__fn__" + Math.random() + "_" + new Date().getTime();
    var originVal;
    var hasOriginVal = isFunction(ctx.hasOwnProperty) ? ctx.hasOwnProperty(propertyName) : false;
    if (hasOriginVal) {
        originVal = ctx[propertyName]
    }

    ctx[propertyName] = this;

    // 采用string拼接
    var argStr = '';
    var len = arguments.length;
    for (var i = 1; i < len; i++) {
        argStr += (i === len - 1) ? 'arguments[' + i + ']' : 'arguments[' + i + ', ';
    }
    var r = eval('ctx["' + propertyName + '"](' + argStr + ')');

    // 还原现场
    if (hasOriginVal) {
        ctx[propertyName] = originVal;
    } else {
        delete ctx[propertyName]
    }

    return r;
}

```

当前版依旧存在问题，

1. 严格模式下，我们用依然用Obeject进行了封装。会导致严格模式下传递非对象的时候，**this**的指向是不准的，不得以的妥协。哪位同学有更好的方案，敬请指导。

1. 虽说我们把临时的属性名变得难以重名，但是如果重名，而函数调用中真调用了此方法，可能会导致异常行为。

所以完美的解决方法，就是产生一个UID.

1. eval的执行，可能会被 [Content-Security-Policy](#) 阻止

大致的提示信息如下：



[Report Only] Refused to evaluate a string as JavaScript because 'unsafe-eval' is not an allowed source of script in the following Content Security Policy directive: "script-src 'self' \*.speedcurve.com 'sha256-g7c3JdQdN02e1L5U1tvJ1LhvnYN7yJXgG07b6h9xkL1o=' 'www.google-analytics.com/analytics.js' 'sha256-3fT9Nmc38P88wxu7Zm9aKnu87vEgGmKwIzzy/vb1KPs=' polyfill.io/v3/polyfill.min.js assets.codepen.io production-assets.codepen.io".

```
Uncaught (in promise) EvalError: Refused to evaluate a string as JavaScript because 'unsafe-eval' is not an allowed source of script in the following Content Security Policy directive: "script-src 'self' *.speedcurve.com 'sha256-g7c3JdQdN02e1L5U1tvJ1LhvnYN7yJXgG07b6h9xkL1o=' 'www.google-analytics.com/analytics.js' 'sha256-3fT9Nmc38P88wxu7Zm9aKnu87vEgGmKwIzzy/vb1KPs=' polyfill.io/v3/polyfill.min.js assets.codepen.io production-assets.codepen.io".
    at Function.call (VM212:1:50)
    at use-swr.js:132
    at Object.next (use-swr.js:113)
    at use-swr.js:17
    at new Promise (<anonymous>)
    at b (use-swr.js:13)
    at use-swr.js:233
    at r (use-swr.js:521)
    at Array.<anonymous> (use-swr.js:539)
    at L (use-swr.js:62)
```

前面两条都应该还能接受，至于第三条，我们不能妥协。

这就得请出下一位嘉宾，new Function。

## new Function

new Function ([arg1[, arg2[, ...argN]],] functionBody)

其基本格式如上，最后一个为函数体。

举个简单的例子：

```
const sum = new Function('a', 'b', 'return a + b');

console.log(sum(2, 6));
// expected output: 8
```

我们call的参数个数是不固定，思路就是从arguments动态获取。

这里我们的实现借用面试官问：能否模拟实现JS的call和apply方法 实现方法：

```
function generateFunctionCode(argsArrayLength) {
  var code = 'return arguments[0][arguments[1]](';
  for(var i = 0; i < argsArrayLength; i++){
    if(i > 0){
      code += ',';
    }
    code += 'arguments[2][' + i + ']';
  }
  code += ')';
  // return arguments[0][arguments[1]](arg1, arg2, arg3...)
  return code;
}
```

## 基于 new Function的实现

```
var hasStrictMode = (function () {
  "use strict";
  return this === undefined;
})();

var isStrictMode = function () {
  return this === undefined;
};

var getGlobal = function () {
  if (typeof self !== 'undefined') { return self; }
```

```

    if (typeof window !== 'undefined') { return window; }
    if (typeof global !== 'undefined') { return global; }
    throw new Error('unable to locate global object');
};

function isFunction(fn){
    return typeof fn === "function";
}

function getContext(context) {
    var isStrict = isStrictMode();

    if (!hasStrictMode || (hasStrictMode && !isStrict)) {
        return (context === null || context === void 0) ? getGlobal() : Object(c
    }
    // 严格模式下, 妥协方案
    return Object(context);
}

function generateFunctionCode(argsLength) {
    var code = 'return arguments[0][arguments[1]](';
    for(var i = 0; i < argsLength; i++){
        if(i > 0){
            code += ',';
        }
        code += 'arguments[2][' + i + ']';
    }
    code += ')';
    // return arguments[0][arguments[1]](arg1, arg2, arg3...)
    return code;
}

Function.prototype.call = function (context) {

    // 不可以被调用
    if (typeof this !== 'function') {
        throw new TypeError(this + ' is not a function');
    }

    // 获取上下文
    var ctx = getContext(context);

    // 更为稳妥的是创建唯一ID, 以及检查是否有重名
    var propertyName = "__fn__" + Math.random() + "_" + new Date().getTime();
    var originVal;
    var hasOriginVal = isFunction(ctx.hasOwnProperty) ? ctx.hasOwnProperty(prop
    if (hasOriginVal) {
        originVal = ctx[propertyName]
    }

    ctx[propertyName] = this;

    var argArr = [];
    var len = arguments.length;

```

```

    for (var i = 1; i < len; i++) {
        argArr[i - 1] = arguments[i];
    }

    var r = new Function(generateFunctionCode(argArr.length))(ctx, propertyName,

    // 还原现场
    if (hasOriginVal) {
        ctx[propertyName] = originVal;
    } else {
        delete ctx[propertyName]
    }

    return r;
}

```

## 评论区问题收集

评论区最精彩：

1. 为什么不用 Symbol 因为是基于ES5的标准来写，如果使用Symbol，那拓展运算符也可以使用。考察的知识面自然少很多。

1. 支付宝小程序eval、new Function都是不给用的 这样的话，可能真的无能为力了。

2. Object.freeze后的对象是不可以添加属性的 感谢[虚鲲菜菜子](#)的指正，其文章[手写 call 与 原生 Function.prototype.call 的区别](#) 推荐大家细读。

如下的代码，严格模式下会报错，非严格模式复制不成功：

```

"use strict";
var context = {
    a: 1,
    log(msg) {
        console.log("msg:", msg)
    }
};

Object.freeze(context);
context.fn = function() {

};

console.log(context.fn);

```

```

VM111 call:12 Uncaught TypeError: Cannot add property fn, object is not extensibl
    at VM49 call:12

```

这种情况怎么办呢，我能想到的是两种方式： 1. 复制对象 2. Object.create 这也算是一种妥协方法，毕竟链路还是变长了。

```

"use strict";
var context = {
    a: 1,
    log(msg) {
        console.log("msg:", msg)
    }
};

```

```
Object.freeze(context);

var ctx = Object.create(context);

ctx.fn = function() {

}

console.log("fn:", typeof ctx.fn); // fn: function
console.log("ctx.a", ctx.a); // ctx.a 1
console.log("ctx.fn", ctx.fn); // ctx.fn f () {}
```

## 小结

回顾一下依旧存在的问题 1. 严格模式下，我们用依然需要用Object进行了封装基础数据类型 会导致严格模式下传递非对象的时候，this的指向是不准的，不得以的妥协。 哪位同学有更好的方案，敬请指导。

1. 虽说我们把临时的属性名变得难以重名，但是如果重名，而函数调用中真调用了此方法，可能会导致异常行为
2. 小程序等环境可能禁止使用eval和new Function
3. 对象被冻结，call执行函数中的this不是真正传入的上下文对象。

所以，我还是修改标题为三千文字，也没写好 **Function.prototype.call**

## 面试现场

一个手写call涉及到不少的知识点，本人水平有限，如有遗漏，敬请谅解和补充。

当面试官问题的时候，你要清楚自己面试的岗位，是P6,P7还是P8。  
是高级开发还是前端组长，抑或是前端负责人。  
岗位不一样，面试官当然期望的答案也不一样。

## 写在最后

写作不易，您的支持就是我前行的最大动力。

## 参考和引用

[Function.prototype.call\(\) - JavaScript | MDN](#)  
[Strict mode - JavaScript | MDN](#)  
[ECMAScript 5 Strict Mode](#)  
[ES合集](#)  
[手写call、apply、bind实现及详解](#)  
[call、apply、bind实现原理](#)  
[面试官问：能否模拟实现JS的call和apply方法](#)

## 那些你熟悉而又陌生的函数

这是我参与更文挑战的第10天，活动详情查看：[更文挑战](#)

## 前言

### 五层境界

1. 我不知道我不知道
2. 我知道我不知道
3. 我知道我知道
4. 我不知道我知道

第三层境界以上的兄台，可以直接离开了。

至于第五层境界，无需知道还是不知道。

废话不多说，进入正题，一起探索那些你熟悉和又陌生的函数。

## 目录

因为手机端目录不显示，单独写一份：[\\* setTimeout && setInterval](#) \* [JSON.parse](#) \* [JSON.stringify](#) \* [addEventListener](#) \* [Array.from](#) \* [String.replace](#) \* [window.getComputedStyle](#) \* [localStorage.setItem](#) \* [控制台特有的](#)

## [setTimeout && setInterval](#)

### 语法

```
var timeoutID = scope.setTimeout(function[, delay, arg1, arg2, ...]);
var timeoutID = scope.setTimeout(function[, delay]);
var timeoutID = scope.setTimeout(code[, delay]);
```

我们最常用的是语法中的第二种

```
var timeoutID = scope.setTimeout(function[, delay]);
```

举个例子

```
setTimeout(()=>{
  console.log(`当前时间:${Date.now()}`);
}, 1000)
```

接着我们再看第三种 `setTimeout(code[, delay])`, `code` 就是代码字符串，其底层实现就是调用 `eval`，你可能问题怎么知道的，因为MDN对这个 `code` 参数有解释。

#### code

An alternative syntax that allows you to include a string instead of a function, which is compiled and executed when the timer expires. This syntax is not recommended for the same reasons that make using `eval()` a security risk.

刚才的代码就等于下面的代码

```
setTimeout(`console.log("当前时间:${Date.now()}")`, 1000)
```

这里有些掘友可能就直接贴到浏览器的控制台去测试，我得提醒你，很可能你不能正确执行，而是收到类似下面的提示

```
[Report Only] Refused to evaluate a string as JavaScript because 'unsafe-eval':
allowed source of script in the following Content Security Policy directive: "s
.....
```

这是因为CSP拦截了，更多关于CSP的内容可以查看[Content-Security-Polic](#) 或者 阮大神的 [Content Security Policy 入门教程](#)

我们最后看第一种 `setTimeout(function[, delay, arg1, arg2, ...])`, 从第三个参数开始，都会传给回调函数，我们看一段代码

```
setTimeout((startTime, tip)=>{
  const endTime = Date.now();
  console.log("时间已经过去了", endTime - startTime, "ms", tip); // 时间已经过去
}, 500, Date.now(), "该休息了");
```

其实 `clearTimeout`, `clearInterval` 也有点小意思，`clearTimeout` 能不能清除 `setInterval`? 那反过来呢?

```
var ticket = setTimeout(()=>{
  console.log("100ms后执行");
```

```

},100);

var ticket2 = setTimeout(()=>{
    console.log("200ms后执行");
},200);
clearInterval(ticket);

// 200ms后执行

```

有人会说，这这这.....，年轻人，不要较真。

题外话：

有传说 `setTimeout` 4ms 的最小间隔。

这是有前提的，如果四次以上的嵌套调用 `setTimeout`，第五次起，才会有这个4ms的约束。

更多可以参见 HTML standard [timers-and-user-prompts](#)。

你也可以去 [MDN setTimeout](#) 亲手测测。

这里补充一句，标准归标准，实现归实现，浏览器厂商那么多，哈。

## JSON.parse

语法

```

JSON.parse(text)
JSON.parse(text, reviver)

```

我们平时 99.99% 的时候，都在使用第一种。

`reviver` 参数用 TypeScript 表示，差不多是这样子的 `(key:string, value: any)=> any`。

如果复用程序只转换某些值而不转换其他值，那么一定要按原样返回所有未转换的值，否则，将从结果对象中删除这些值。

看个例子，转换成对象后，IDCard 属性就没了。

```

JSON.parse(JSON.stringify({
    name: "tom",
    age: 18,
    phone: "15888787788",
    IDCard: "xxxx"
}), (key, value)=>{

    if(key === "IDCard"){
        return undefined
    }
    return value;
}); // {name: "tom", age: 18, phone: "15888787788"}

```

这里有两点补充：1. `JSON.parse` 转换的时候，是深度优先遍历 2. 最后一条 key 的值，是空值

看代码：

```

JSON.parse('{"1": 1, "2": 2, "3": {"4": 4, "5": {"6": 6}}}', (key, value) => {
    console.log(key); // log the current property name, the last is "".
    return value;     // return the unchanged property value.
});

// 1

```

```
// 2
// 4
// 6
// 5
// 3
// ""
```

## [JSON.stringify](#)

### 语法

```
JSON.stringify(value)
JSON.stringify(value, replacer)
JSON.stringify(value, replacer, space)
```

第一种就不多说了。

后两个参数的说明如下

replacer 可选

- 如果该参数是一个函数，则在序列化过程中，被序列化的值的每个属性都会经过该函数的转换和处理；
- 如果该参数是一个数组，则只有包含在这个数组中的属性名才会被序列化到最终的 JSON 字符串中；
- 如果该参数为 null 或者未提供，则对象所有的属性都会被序列化。

space 可选

指定缩进用的空白字符串，用于美化输出（pretty-print）； \* 如果参数是个数字，它代表有多少的空格；上限为10。该值若小于1，则意味着没有空格； \* 如果该参数为字符串（当字符串长度超过10个字母，取其前10个字母），该字符串将被作为空格； \* 如果该参数没有提供（或者为 null），将没有空格。

到这里，有些同学吃惊一把。其实掘金有很多关于这篇文章 \* 爆款 [你不知道的 JSON.stringify\(\) 的威力](#) \* 有意思的[JSON.parse、JSON.stringify](#) \* 深拷贝系列 [———— 自己实现一个JSON.stringify和JSON.parse](#)

当然我说的两点： \* toJSON \* space参数

**toJSON:** 当一个需要被转换的对象定义了toJSON方法， 会直接返回toJSON的值。 举个栗子：

```
var obj = {
  foo: 'foo',
  toJSON: function () {
    return 'bar';
  }
};
JSON.stringify(obj); // '"bar"'
```

**space:** 参数是用于美化转化后格式的，举个栗子: 保存JSON文件的时候，尤其有用。当然，下面的代码你放到浏览器，有些浏览器可能看不出什么效果，在nodejs执行，然后保存文件就看到效果了。

```
JSON.stringify({ uno: 1, dos : 2 }, null, '\t')
// '{
//   "uno": 1, \
//   "dos": 2  \
// }'
```

### addEventListener

就这，就这，还怀疑我不会用。我就真觉得，看到这里的有50%以上，就真不全会。等看完，如果是真不全会的，麻烦评论区评论 一起围观作者。；



```
target.addEventListener(type, listener, options);
target.addEventListener(type, listener, useCapture)
```

四个9的可靠性，99.9999%的通知，都完全知道第三个参数是布尔值的情况。我也就不说了。

## options 可选

可用的选项：\* capture: Boolean，表示 listener 会在该类型的事件捕获阶段传播到该 EventTarget 时触发。\* once: Boolean，表示 listener 在添加之后最多只调用一次。如果是 true，listener 会在其被调用之后自动移除。\* passive: Boolean，设置为true时，表示 listener 永远不会调用 preventDefault()。如果 listener 仍然调用了这个函数，客户端将会忽略它并抛出一个控制台警告。查看 [使用 passive 改善的滚屏性能](#) 了解更多。\* signal: AbortSignal，该 AbortSignal 的 abort() 方法被调用时，监听器会被移除。

### once

看到once是不是很熟悉，EventEmitter, socket.io等等是不是都有，这就是我们熟悉的那个啥模式。

其实 window本身一定程度上就是一个订阅发布中心：

```
function _dispatch(type, data) {
  const ev = new CustomEvent(type, { detail: data });
  window.dispatchEvent(ev);
}

window._on = window.addEventListener
window._emit = _dispatch;
window._once = (type, callback) => {
  window.addEventListener(type, callback, {
    once: true,
    capture: true
  })
}

window._on("event-x", (ev) => {
  console.log("event-x 收到数据:", ev.detail);
});

window._once("event-once", (ev) => {
  console.log("event-once 收到数据:", ev.detail);
}, {
  once: true,
});

window._emit("event-x", {
  uid: 100,
  message: "i love you"
})

window._emit("event-once", {
  uid: -100,
  message: "you love me"
});

window._emit("event-once", {
  uid: -100,
  message: "you love me"
});
```

```
// event-x 收到数据: {uid: 100, message: "i love you"}
// event-once 收到数据: {uid: -100, message: "you love me"}
```

实际上，不仅仅是Window，任何继承了 [EventTarget](#) 的对象，都具备这样的能力。

## passive

查看 [使用 passive 改善的滚屏性能](#) 了解更多

## signal

这个signal是[AbortController](#)的一部分，其主要作用是用来终止请求。

而在此处的作用，是用来移除监听器。

如下的代码，如果注释controller.abort();，就会收到两条消息。

```
var controller = new AbortController();
var signal = controller.signal;

function _dispatch(type, data) {
  const ev = new CustomEvent(type, { detail: data });
  window.dispatchEvent(ev);
}

window.addEventListener("event-x", ev => {
  console.log("event-x:收到数据: ", ev.detail);
}, {
  signal
});

_dispatch("event-x", "i love you");
// 终止
controller.abort();
_dispatch("event-x", "i love you");

// event-x:收到数据: i love you
```

到此如果你真的全会，麻烦评论区评论 一起围观作者。

## Array.from

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

重点是这个mapFn:

apFn 可选 如果指定了该参数，新数组中的每个元素会执行该回调函数。

先看妙用，造一个1-100的数组，哦，so easy!

```
Array.from({length:100}, (val, index)=> index + 1 );
```

字符串

```
Array.from('foo');
// [ "f", "o", "o" ]
```

Set

```
const set = new Set(['foo', 'bar', 'baz', 'foo']);
```

```
Array.from(set);
// [ "foo", "bar", "baz" ]
```

Map 这个地方大家看着可能有点迷，更多详情[Map 与数组的关系](#)

```
const map = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(map); // 等同于 [...map.entries()]
// [[1, 2], [2, 4], [4, 8]]

const mapper = new Map([['1', 'a'], ['2', 'b']]);
Array.from(mapper.values());
// ['a', 'b'];

Array.from(mapper.keys());
// ['1', '2'];
```

## 序列生成器

```
const range = (start, stop, step) => Array.from(
  { length: (stop - start) / step + 1 },
  (_, i) => start + (i * step)
);

// Generate numbers range 0..4
range(0, 4, 1);
// [0, 1, 2, 3, 4]
```

range在Python里面是自带的，你看我们JS的实现也很简单。

## String.replace

```
str.replace(regex|substr, newSubStr|function)
```

## 特殊变量名

特殊变量名 让这个replace属于化神后期，下面的\$2，\$1就是字符串参数。

```
var re = /(\w+)\s(\w+)/;
var str = "John Smith";
var newstr = str.replace(re, "$2, $1");
// Smith, John
console.log(newstr)
```

所有的字符串参数如下：|变量名|代表的值|---|---|\$\$ 插入一个 "\$" ||\$&插入匹配的子串。||\$`插入当前匹配的子串左边的内容。||\$'插入当前匹配的子串右边的内容。||\$n|假如第一个参数是 RegExp对象，并且 n 是个小于100的非负整数，那么插入第 n 个括号匹配的字符串。提示：索引是从1开始。如果不存在第 n 个分组，那么将会把匹配到内容替换为字面量。比如不存在第3个分组，就会用“\$3”替换匹配到的内容。|\$| 这里Name 是一个分组名称。如果在正则表达式中并不存在分组（或者没有匹配），这个变量将被处理为空字符串。只有在支持命名分组捕获的浏览器中才能使用|

## 第二个参数为函数

为了方便理解，先看：

```
function replacer(match, p1, p2, p3, offset, string) {
  // p1 is nondigits, p2 digits, and p3 non-alphanumerics
  return [p1, p2, p3].join(' - ');
}

var newString = 'abc12345#$*%'.replace(/([^\d]*) (\d*) ([^\w]*)/, replacer);
```

```
console.log(newString); // abc - 12345 - #${*%}
```

p1, p2, p3对应着 特殊变量名 \$1, \$2, \$3,

更多的细节移步MDN [函数参数](#)

## window.getComputedStyle

大家都知道，CSS有权重的说法，有style的，有css里面的各种，有浏览器的默认样式，等等。鬼知道，最后生效的是哪些样式。鬼就等于这个getComputedStyle。

先看个例子：

```
<style>
#elem-container{
  position: absolute;
  left:      100px;
  top:       200px;
  height:    100px;
  z-index: 1;
}
</style>

<div id="elem-container">dummy</div>

<script>
  let elem = document.getElementById("elem-container");
  const ps = window.getComputedStyle(elem, null)
  console.log(ps.getPropertyValue("height")); // 100px
</script>
```

## 读值方式

1. ps.getPropertyValue("z-index");
2. ps["z-index"];
3. ps.zIndex （同style的读取）

上面的三种方式都是可行的。

## 返回值为解析值

getComputedStyle的返回值是 [解析值](#)，常跟CSS2.1中的[计算值](#)是相同的值。但对于一些旧的属性，比如width, height, padding 它们的值又为 [应用值](#)。

**计算值：**例如，如一个元素的属性值为 font-size:16px 和 padding-top:2em, 则 padding-top 的计算值为 32px (字体大小的2倍)。

**应用值：**比如：span 指定 position: absolute 后display 变为 block。

其次值和设置的值不一定是同样的格式，

比如你设置的 color的值是 red, 返回的值是 rgb(255, 0, 0);

比如你说的 transform transform: translate(10px, 10px),返回的值是 matrix(1, 0, 0, 1, 10, 10)。

## 第二个参数 pseudoElt

pseudoElt 可选

指定一个要匹配的伪元素的字符串。必须对普通元素省略（或null）

就是用来获取[伪元素](#)的最终样式，常见的有 ::after, ::before, ::marker, ::line-marker。

```

<style>
  h3::after {
    content: "rocks!";
  }
</style>

<h3>generated content</h3>

<script>
  let h3 = document.querySelector('h3'),
  result = getComputedStyle(h3, '::after').content;
  alert(`the generated content is: ${result}`);
  console.log(`the generated content is: ${result}`);
  // the generated content is: "rocks!"
</script>

```

写在最后，getComputedStyle会触发重排，当然很多获取宽高的都会触发重排。心里有这个概念就行。

## localStorage && sessionStorage

### storage事件

这里要说的是localStorage.setItem，其本身没有什么好说的。与其对应的有一个[storage事件](#)，其可以监听storage的值的變化。

#### A.html

```

localStorage.setItem("data", JSON.stringify({
  a: 1,
  b: 3
}))

```

#### B.html

```

window.addEventListener("storage", (ev) => {
  console.log("key:", ev.key); // key: data
  console.log("oldValue:", ev.oldValue); // oldValue: null
  console.log("newValue:", ev.newValue); // newValue: {"a":1,"b":3}
  console.log("storageArea:", ev.storageArea);
  // storageArea: Storage {ev: {"a":1,"b":3}, length: 1}
  console.log("url:", ev.url); // url: http://127.0.0.1:8080/A.html
});

```

从上面隐藏两个细节 1. A.html调用setItem, B.html监听事件 因为A.html自身是监听不到这个事件的

1. storage事件对象上有oldValue喝newValue属性 可以得出，设置同样的值，不会触发事件。

### 让setItem的页面监听storage事件

原理，重写setItem事件

```

var originalSetItem = sessionStorage.setItem;
localStorage.setItem = function (key, newValue) {
  var itemEvent = new CustomEvent("c-storage", {
    detail: {
      oldValue: localStorage.getItem(key),
      newValue,
      key,
      url: location.href
    }
  });
  originalSetItem.call(localStorage, key, newValue);
  itemEvent.dispatchEvent(window);
}

```

```

    }
    });
    window.dispatchEvent(itemEvent);
    originalSetItem.apply(this, arguments);
}

window.addEventListener("c-storage", function (ev) {
    console.log("ev", ev.detail);
})
// key: "data"
// newValue: 1624506604683
// oldValue: "1624506595323"
// url: "http://127.0.0.1:8080/LA.html"

localStorage.setItem("data", Date.now());

```

IE貌似有bug, [同页面 Storage 变化监听](#)有提到

控制台特有的

**console系列**

那些 console.table, console.time, console.assert, console.count, console.group 我就不说了。

偏偏说一下 console.log

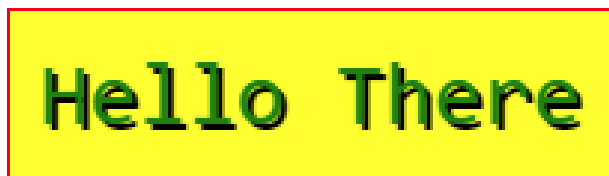
console.log 定义样式: 来源 [多彩的 console.log](#)

```

// 1. 将css样式内容放入数组
const styles = [
    'color: green',
    'background: yellow',
    'font-size: 30px',
    'border: 1px solid red',
    'text-shadow: 2px 2px black',
    'padding: 10px',
].join(';');
// 2. 利用join方法讲各项以分号连接成一串字符串
// 3. 传入styles变量
console.log('%cHello There', styles);

```

输出:



**符号函数系列**

方法	说明
\$_	最后一个评估值
\$(selector)	document.querySelector
?(selector)	document.querySelectorAll
\$x(path)	XPathEvaluator

```
$x("/html/body")
```

```
> $x("/html/body")  
< [body]
```

## 其他

getEventListeners:

你有没有碰到过，想查询某个元素的所有监听函数的需求，在页面里面是没有这种功能。但是控制台是有的  
看图

```
> getEventListeners(document)  
▼ {scroll: Array(2), visibilitychange: Array(9), click: Array(3), keyup: Array(1), keypress: Array(2)}  
  ► click: (3) [(), {}, {}]  
    ► keypress: (2) [(), {}]  
    ► keyup: [{}]  
  ► scroll: (2) [(), {}]  
  ► visibilitychange: (9) [(), {}, {}, {}, {}, {}, {}, {}, {}]  
  ► __proto__: Object
```

clear: 清空控制台

## 写在最后

写作不易，一赞一评，就是我最大的动力。

[同页面 Storage 变化监听](#)

[setTimeout](#)

[setInterval](#)

[Map 与数组的关系](#)

[timers-and-user-prompts](#)

[多彩的console.log](#)

[推荐几个不错的console调试技巧](#)

这16种原生函数和属性的区别，你真的知道吗？精心收集，高级前端必备知识，快快打包带走

本文已参与好文召集令活动，点击查看：[后端、大前端双赛道投稿，2万元奖池等你挑战！](#)

## 前言

原生内置了很多API, 作用类似，却也有差千差万别，了解其区别，掌握前端基础，是修炼上层，成为前端高级工程师的必备知识，让我们一起来分类归纳，一起成长吧。

上一篇前端基础好文：[那些你熟悉而又陌生的函数](#)

属性获取 `keys`, `getOwnPropertyNames`, `getOwnPropertySymbols`

### [Object.keys](#)

返回一个由一个给定对象的自身可枚举属性组成的数组，数组中属性名的排列顺序和正常循环遍历该对象时返回的顺序一致。

### [Object.getOwnPropertyNames](#)

返回一个由指定对象的所有自身属性的属性名（包括不可枚举属性但不包括Symbol值作为名称的属性）组成的数组。

### [Object.getOwnPropertySymbols](#)

一个给定对象自身的所有 Symbol 属性的数组。

### [Reflect.ownKeys](#)

返回一个由目标对象自身的属性键组成的数组。

等同于 `Object.getOwnPropertyNames(target).concat(Object.getOwnPropertySymbols(target))`

### 例子 ```js const symbolSalary = Symbol.for("salary"); const symbolIsAnimal = Symbol.for("isAnimal"); const symbolSay = Symbol.for("say");

function Person(age, name){ this.age = age; this.name = name;

```
this.walk = function () {  
  console.log("person:walk");  
}
```

}

// 原型方法 Person.prototype.say = function(words){ console.log("say:", words); } Person.prototype[symbolSay] = function(words){ console.log("symbolSay", words); }

// 原型属性 Person.prototype[symbolIsAnimal] = true; Person.prototype.isAnimal = true;

const person = new Person(100, "程序员");

person[symbolSalary] = 6000; person["sex"] = "男";

// sex 不可枚举 Object.defineProperty(person, "sex", { enumerable: false });

Object.defineProperty(person, symbolSalary, { enumerable: false, // 无效的设置 value: 999 });

const keys = Object.keys(person); const names = Object.getOwnPropertyNames(person); const symbols = Object.getOwnPropertySymbols(person); const ownKeys = Reflect.ownKeys(person);

console.log("keys", keys); // [ 'age', 'name', 'walk' ] console.log("getOwnPropertyNames", names); // [ 'age', 'name', 'walk', 'sex' ] console.log("getOwnPropertySymbols", symbolSalary); // [ Symbol(salary) ] console.log("ownKeys", ownKeys); // [ 'age', 'name', 'walk', 'sex', Symbol(salary) ]

console.log("-----") console.log(person.isAnimal); // true console.log(person[symbolIsAnimal]); // true console.log(person[symbolSalary]); // 999 person[symbolSay]; // symbolSay hello world person.say("hello world"); // say: hello world person.walk(); // person:walk ```

## 总结

1. Object.keys: 则返回的是所有可枚举属性键，也就是属性下的enumerable: true。但不包括Symbol值作为名称的属性键。
2. Object.getOwnPropertyNames: 返回的是对象所有自己的属性键，包括不可枚举属性但不包括Symbol值作为名称的属性键。
3. Object.getOwnPropertySymbols: 方法返回一个给定对象自身的所有 Symbol 属性键的数组。
4. Reflect.ownKeys: 返回一个由目标对象自身的属性键组成的数组。等同于 `Object.getOwnPropertyNames(target).concat(Object.getOwnPropertySymbols(target))`。

## 节点位置关系 Node.contains, Node.compareDocumentPosition

### Node.compareDocumentPosition

比较当前节点与任意文档中的另一个节点的位置关系

语法 `compareMask = node.compareDocumentPosition( otherNode )`

返回值是一个具有以下值的位掩码: | 常量名 | 十进制值 | 含义 | |---|---|---|

DOCUMENT\_POSITION\_DISCONNECTED | 1 | 不在同一文档中 | | DOCUMENT\_POSITION\_PRECEDING | 2 | otherNode在node之前 | | DOCUMENT\_POSITION\_FOLLOWING | 4 | otherNode在node之后 |



DOCUMENT\_POSITION\_CONTAINS | 8 | otherNode 包含 node | | DOCUMENT\_POSITION\_CONTAINED\_BY | 16 | otherNode 被 node 包含 | | DOCUMENT\_POSITION\_IMPLEMENTATION\_SPECIFIC | 32 | 待定 |

在一些场景下，可能设置了不少一位比特值。比如 otherNode 在文档中是靠前的且包含了 Node，那么 DOCUMENT\_POSITION\_CONTAINS 和 DOCUMENT\_POSITION\_PRECEDING 位都会设置，所以结果会是 0x0A 即十进制下的 10。

看代码：

结果是：20

1. child 在 parent 之后，赋值得 4 2. child 被 parent 包含，赋值的 16

4 + 16 = 20

```
<div id="parent">
  <div id="child"></div>
</div>

<script>

  const pEl = document.getElementById("parent");
  const cEl = document.getElementById("child");
  console.log(pEl.compareDocumentPosition(cEl)); // 20

</script>
```

## Node.contains

返回的是一个布尔值，来表示传入的节点是否为该节点的后代节点

基本等于 compareDocumentPosition 的 | DOCUMENT\_POSITION\_CONTAINED\_BY | 16 | otherNode 被 node 包含 |

## 总结

1. compareDocumentPosition 返回的是数字，带组合意义的信息，不仅仅可以返回包含，还可以返回在之前之后等信息
2. contains 返回的是布尔值，仅仅告诉你是否有包含关系

取文本 **innerText**, **textContent**

## HTMLElement.innerText

解析过程：1. 对 HTML 标签进行解析；2. 对 CSS 样式进行带限制的解析和渲染；3. 将 ASCII 实体转换为对应的字符；4. 剔除格式信息（如 \t、\r、\n 等），将多个连续的空格合并为一个

## Node.textContent

解析过程：1. 对 HTML 标签进行剔除；2. 将 ASCII 实体转换为相应的字符。  
需要注意的是：

1. 对 HTML 标签是剔除不是解析，也不会出现 CSS 解析和渲染的处理，因此 <br/> 等元素是不生效的。
2. 不会剔除格式信息和合并连续的空格，因此 \t、\r、\n 和连续的空格将生效

例子

```
<p id="source">
  <style>
    #source {
      color: red;
    }
  </style>
  source
</p>
```

```

</style>
Take a look at<br>how this text<br>is interpreted
below.
<span style="display:none">HIDDEN TEXT</span>
</p>

<h3>Result of textContent:</h3>
<textarea id="textContentOutput" rows="12" cols="50" readonly>...</textarea>
<h3>Result of innerText:</h3>
<textarea id="innerTextOutput" rows="12" cols="50" readonly>...</textarea>

<script>
  const source = document.getElementById('source');
  const textContentOutput = document.getElementById('textContentOutput');
  const innerTextOutput = document.getElementById('innerTextOutput');

  textContentOutput.innerHTML = source.textContent;
  innerTextOutput.innerHTML = source.innerText;
</script>

```

看看结果:

Take a look at  
how this text  
is interpreted below.

**Result of textContent:**

```

#source {
  color: red;
}

Take a look at how this text is interpreted
below.
HIDDEN TEXT

```

**Result of innerText:**

Take a look at  
how this text  
is interpreted below.

## 总结

1. innerText是会解析css的, <br/>有效, 剔除格式信息(如\t、\r、\n等), 将多个连续的空格合并为一个。
2. textContent是剔除html标签, <br/>无效, \t、\r、\n和连续的空格将生效。

节点取值 **value**, **nodeValue**

### Node.nodeValue

- 对于text, comment, 和 CDATA 节点来说, nodeValue返回该节点的文本内容。
- 对于 attribute 节点来说, 返回该属性的属性值。

对应着下面表格的 nodeType的值 text 3,4,8

常量	nodeType 值	描述
----	---------------	----

常量	nodeType 值	描述
Node.ELEMENT_NODE	1	一个 元素 节点, 例如 和
Node.TEXT_NODE	3	Element 或者 Attr 中实际的 文字
Node.CDATA_SECTION_NODE	4	一个 CDATASection, 例如 \<!CDATA[[ ... ]]>。
Node.PROCESSING_INSTRUCTION_NODE	7	一个用于XML文档的 ProcessingInstruction (en-US) , 例如 <?xml-stylesheet ... ?> 声明。
Node.COMMENT_NODE	8	一个 Comment 节点。
Node.DOCUMENT_NODE	9	一个 Document 节点。
Node.DOCUMENT_TYPE_NODE	10	描述文档类型的 DocumentType 节点。例如 <!DOCTYPE html> 就是用于 HTML5 的。
Node.DOCUMENT_FRAGMENT_NODE	11	一个 DocumentFragment 节点

## value

特定的一些HTMLElement元素, 用value属性获取其值。常见的有value属性的元素如下: \*

[HTMLInputElement](#) <input value="1" /> \* [HTMLTextAreaElement](#) <textarea value= "你哈" /> \*  
[HTMLButtonElement](#) <button value= "提交" /> \* [HTMLDataElement](#) <data value="21053">圣女  
 果</data> \* [HTMLSelectElement](#) <select><option value ="volvo">Volvo</option> \*  
[HTMLOptionElement](#) <select><option value ="volvo">Volvo</option>  
 \* [HTMLParamElement](#)

```
``js
```

```
...
```

- [HTMLProgressElement](#) <progress value="22" max="100"></progress>

## ### 总结

1. nodeValue 是文本节点, 属性节点, 注释节点等类型的节点用来取值的方法
2. vlaue是特定的元素节点用来取值的方法

## 节点复制 adoptNode, importNode, cloneNode

### [Document.adoptNode](#)

将外部文档的一个节点拷贝一份,然后可以把这个拷贝的节点插入到当前文档中。

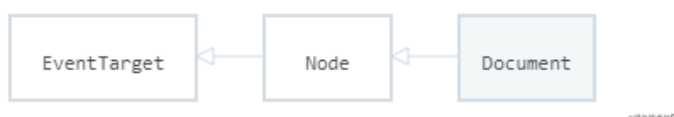
### [Document.importNode](#)

从其他的document文档中获取一个节点。 该节点以及它的子树上的所有节点都会从原文档删除, 并且它的 ownerDocument 属性会变成当前的document文档。 之后你可以把这个节点插入到当前文档中。

### [Node.cloneNode](#)

生成节点的一个副本。

实际上 Document 是继承自 Node, 也具备cloneNode方法。



这里提一个问题:

Document.adoptNode与Document.importNode是操作外部文档的, 那么操作所在的文档会有什么效果

呢？

`Node.cloneNode` 有一个 `boolean` 类型的可选参数 `deep`：

\* `true`: 则该节点的所有后代节点也都会被克隆 \* `false`: 则只克隆该节点本身.

### 注意

1. `cloneNode` `deep` 参数在不同版本的浏览器实现中，默认值可能不一样，所以强烈建议写上值。2. `cloneNode` 会克隆一个元素节点会拷贝它所有的属性以及属性值,当然也就包括了属性上绑定的事件(比如 `onclick="alert(1)"`),但不会拷贝那些使用 `addEventListener()` 方法或者 `node.onclick = fn` 这种用 JavaScript 动态绑定的事件

### 总结

1. `adoptNode` 从外部文档进行拷贝
2. `importNode` 从外部文档进行拷贝，并从外部文档删除
3. `cloneNode` 从本文档进行复制，有浅复制和深复制

父节点 `childNodes`, `children`

### [Node.childNodes](#)

节点的子节点集合，包括元素节点、文本节点还有属性节点

### [ParentNode.children](#)

返回的只是节点的元素节点集合, 即 `nodeType` 为 1 的节点。

### 例子

来实际看一段代码：

```
<div id="root">
  1
  <span>2</span>
  3
  <!-- <div></div> -->
  <![CDATA[[ 4 ]]>
</div>

<script>
  const rootEl = document.getElementById("root");
  console.log(rootEl.children);
  console.log(rootEl.childNodes);
</script>
```

返回结果截图：

```

▼ HTMLCollection [span] 1
  ▶ 0: span
    length: 1
  __proto__: HTMLCollection
▼ NodeList(7) [text, span, text, comment, text, comment, text] 7
  ▶ 0: text
  ▶ 1: span
  ▶ 2: text
  ▶ 3: comment
  ▶ 4: text
  ▶ 5: comment
    baseURI: "http://127.0.0.1:5500/%E4%B0%A0%E7%9C%9F%E7%9A%84%E7%9F%A5%E9%81%93/children.html"
    childNodes: NodeList(1)
    data: "CDATA[[ 4 ]]"
    firstChild: null
    isConnected: true
    lastChild: null
    length: 12
    nextElementSibling: null
    nextSibling: text
    nodeName: "#comment"
    nodeType: 8
    nodeValue: "CDATA[[ 4 ]]"
    ownerDocument: document
    parentElement: div#root
    parentNode: div#root
    previousElementSibling: span
    previousSibling: text
    textContent: "CDATA[[ 4 ]]"
    __proto__: Comment
  ▶ 6: text
    length: 7

```

Node.parentNode与Node.parentElement也是同样的道理。

## 总结

1. children只返回元素节点，也就是 nodeType为1的节点
2. childNodes 返回所有类型的节点

## 添加节点 append, appendChild

### node.appendChild

将一个节点附加到指定父节点的子节点列表的末尾处

### ParentNode.append

方法在 ParentNode的最后一个子节点之后插入一组 Node 对象或 DOMString 对象。被插入的 DOMString 对象等价于 Text 节点。

## 例子

我们一次append三个节点，其中两个文本节点，一个div节点。

```

<div id="root"></div>
<script>
  function createEl(type, innerHTML){
    const el = document.createElement(type);
    el.innerHTML = innerHTML;
    return el;
  }
  const rootEl = document.getElementById("root");

  rootEl.append("我们", createEl("div", "都是"), "好孩子");
</script>

```

我们  
都是  
好孩子

```

<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div id="root">
      "我们"
      <div>都是</div>
      ...
      "好孩子" == $0
    </div>
    <script>...</script>
    <!-- Code injected by live-server -->
    <script type="text/javascript">...</script>
  </body>
</html>

```

## 总结

- `ParentNode.append()` 允许追加 `DOMString` 对象, 而 `Node.appendChild()` 只接受 `Node` 对象。
- `ParentNode.append()` 没有返回值, 而 `Node.appendChild()` 返回追加的 `Node` 对象。
- `ParentNode.append()` 可以追加多个节点和字符串, 而 `Node.appendChild()` 只能追加一个节点。

简直说, `append` 强大太多了。

文档可见状态 `Document.hidden`, `Document.visibilityState`

### [document.hidden](#)

返回布尔值, 表示页面是 (true) 否 (false) 隐藏。

### [Document.visibilityState](#)

返回 `document` 的可见性, 由此可以知道当前文档(即为页面)是在背后, 或是不可见的隐藏的标签页, 或者(正在)预渲染. 可用的值如下: \* `'visible'`: 此时页面内容至少是部分可见. 即此页面在前景标签页中, 并且窗口没有最小化. \* `'hidden'`: 此时页面对用户不可见. 即文档处于背景标签页或者窗口处于最小化状态, 或者操作系统正处于 '锁屏状态'. \* `'prerender'`: 页面此时正在渲染中, 因此是不可见的. 文档只能从此状态开始, 永远不能从其他值变为此状态. 注意: 浏览器支持是可选的.

当此属性的值改变时, 会递交 `visibilitychange` 事件给 `Document`.

## 例子

我们先输出当前状态, 然后点击别的tab, 等下再点击回来。

```
console.log(
  "visibilityState:",
  document.visibilityState,
  " hidden:",
  document.hidden,
);
console.log("");
document.addEventListener("visibilitychange", function () {
  console.log(
    "visibilityState:",
    document.visibilityState,
    " hidden:",
    document.hidden
  );
});
```

not available
visibilityState: visible hidden: false
visibilityState: hidden hidden: true
visibilityState: visible hidden: false
>

我们日常可以用 `visibilitychange` 来监听当前页面处于隐藏时, 去清除定时器或页面中的动画, 停止音乐视频等的播放。

我还想到一个有意思的 1. 广告倒计时 你离开后, 不算倒计时, 会不会被骂死

1. 阅读某些协议 你离开后, 停止倒计时

## 总结

1. hidden 与 visibilityState 返回值不同，一个是布尔值，一个是字符串
2. visibilityState 的状态多一种 prerender, 其对应的hidden的值是true
3. visibilityState 有相关的事件

函数调用 **call, apply, bind**

### [Function.prototype.call](#)

使用一个指定的 this 值和单独给出的一个或多个参数来调用一个函数

### [Function.prototype.apply](#)

调用一个具有给定this值的函数，以及以一个数组（或类数组对象）的形式提供的参数

### [Function.prototype.bind](#)

方法创建一个新的函数，在 bind() 被调用时，这个新函数的 this 被指定为 bind() 的第一个参数，而其余参数将作为新函数的参数，供调用时使用

例子

```
function sum(...args) {
  const total = args.reduce((s, cur) => {
    return s + cur;
  }, 0);

  return (this.base || 0) + total;
}

const context = {
  base: 1000
};

const bindFun = sum.bind(context, 1, 2);

const callResult = sum.call(context, 1, 2, 3, 4);
const applyResult = sum.apply(context, [1, 2, 3, 4]);
const bindResult = bindFun(3, 4);

console.log("call:", callResult); // 1010
console.log("apply:", applyResult); // 1010
console.log("bind:", bindResult); // 1010
```

总结

相同点，都能改变被调用函数的this指向。

1. call: 第二个参数开始，可以接收任意个参数
2. apply: 第二个参数，必须是数组或者类数组
3. bind: 第二个参数开始，可以接收任意个参数，返回的是一个新的函数

注意点： 1. bind调用多次，this指向第一次第一个参数

log 调用了两次bind, 第一次bind{ val: 1 }, 第二次bind{ val: 2 }, 输出的this是一次bind的上下文

```
function log() {
  console.log("this", this);
}

console.log(log.bind({ val: 1 }).bind({ val: 2 })()) // { val: 1 }
```

再看一段类似的代码：

虽然this的指向不会再变改变，但是参数还是继续接受，arguments 长度为2，第一次bind的1，第二次bind的2，都照单全收。

```
function log() {
  console.log("this", this);    // { val: 1 }
  console.log("arguments", arguments); // { '0': 1, '1': 2 }
}

console.log(log.bind({ val: 1 }, 1).bind({ val: 2 }, 2)()) // 1
```

## 字符串截取 substr, substring

### [String.prototype.substr](#)

返回一个字符串中从指定位置开始到指定字符数的字符

语法： 第二参数，是需要截取的长度

```
str.substr(start[, length])
```

### [String.prototype.substring](#)

返回一个字符串在开始索引到结束索引之间的一个子集, 或从开始索引直到字符串的末尾的一个子集。

语法： 第二参数，结束索引

```
str.substring(indexStart[, indexEnd])
```

## 例子

提示： 1. 两个参数都没设置的时候，效果相同 2. 第一个参数是大于等于0的整数，没设置第二参数的时候，效果相同

```
const str = "我们都是好孩子";

console.log(str.substr())    // 我们都是好孩子
console.log(str.substring()) // 我们都是好孩子

console.log(str.substr(1))   // 们都是好孩子
console.log(str.substring(1)) // 们都是好孩子

console.log(str.substr(-1))  // 子
console.log(str.substring(-1)) // 我们都是好孩子

console.log(str.substr(1, 2)) // 们都
console.log(str.substring(1, 2)) // 们
```

## 总结

1. substr 第二个参数是需要截取的长度
2. substring 第二个参数是结束索引值的
3. 没指定参数或者第一个参数是大于等于0的整数时，效果相同
4. 第一个参数是负数或者第二个参数是负数，处理规则不通 具体参见 [substr](#)和 [substring](#)

## 遍历 for of, for in

### for in



获取`enumerable:true`的属性键

## for of

遍历属性值。不受到`enumerable`限制。

## 例子

1. 在数组原型上增加了方法`gogo`, `for in`结果中出现了, 而 `for of`结果中未出现。
2. 定义了 属性2不能被遍历, `for in`结果中未出现, 而 `for of`结果中出现了。

```
// 原型上增加方法
Array.prototype.gogo = function() {
    console.log("gogo");
}

var a = [1,2,3];

// key值2不可以枚举
Object.defineProperty(a, 2, {
    enumerable: false
});
Object.defineProperty(a, "2", {
    enumerable: false
});

for(let p in a){
    // 索引被遍历出来是字符串类型
    console.log(p, typeof p); // 0 string; 1 string; gogo string
}

console.log("---")

for(let v of a){
    console.log(v); // 1 2 3
}
```

## 总结

`for in` 1. 获取`enumerable:true`的属性键。 2. 可以遍历对象。 3. 可以获取原型上的属性键。 4. 数字属性键被遍历出来是字符串。 比如索引值 `for of`:

1. 遍历属性值。不受到`enumerable`限制。
2. 可遍历数组。 一般不可以遍历对象, 如果实现了`Symbol.iterator`, 可以遍历。 如 `Array`, `Map`, `Set`, `String`, `TypedArray`, `arguments` 对象等等
3. 不能获取原型上的值

当前时间 `Date.now()`, `Performance.now()`

### [Date.now\(\)](#)

方法返回自 1970 年 1 月 1 日 00:00:00 (UTC) 到当前时间的毫秒数。

### [Performance.now](#)

获取当前的时间戳的值（自创建上下文以来经过的时间），其值是一个精确到毫秒的 [DOMHighResTimeStamp](#)。

<!DOCTYPE html>

```

<html lang="en">

<head>
  <script>
    console.log("p1", performance.now())
  </script>
</head>

<body>

  <script>
    console.log("p2", performance.now());

    setTimeout(() => {
      console.log("p3", performance.now());
    }, 1000)
  </script>
</body>

</html>

```

not available

p1 71.7999999821186

p2 72.09999999403954

p3 1073.7999999821186

>

## 总结

1. Date.now()的基准是 1970 年 1 月 1 日 00:00:00 (UTC), 而Performance.now是上下文创建。
2. Date.now() 返回的是整数, Performance.now返回的是double类型
3. 理论上Performance.now精度更高

## 域名信息 host, hostname

### location.host

其包含: 主机名, 如果 URL 的端口号是非空的, 还会跟上一个 ':', 最后是 URL 的端口号

### location.hostname

返回域名

## 例子

https与http的默认端口号, 是不会被 host包含的, 看下面的代码

https://developer.mozilla.org:443 的host是 developer.mozilla.org, 因为443是https的默认端口。

```

var anchor = document.createElement("a");

anchor.href = "https://developer.mozilla.org:443/en-US/Location.host";
console.log(anchor.host == "developer.mozilla.org:443") // false
console.log(anchor.host == "developer.mozilla.org") // true

```

```

console.log(anchor.hostname == "developer.mozilla.org:443"); // false
console.log(anchor.hostname == "developer.mozilla.org"); // true

anchor.href = "https://developer.mozilla.org:4097/en-US/Location.host";
console.log(anchor.host == "developer.mozilla.org:4097") // true
console.log(anchor.hostname == "developer.mozilla.org") // true

```

## 总结

1. 默认端口下， host等于hostname
2. host额外包含端口号

## 事件注册 on, addEventListener

### 内联事件

注册事件。

### [EventTarget.addEventListener](#)

方法将指定的监听器注册到 EventTarget 上，当该对象触发指定的事件时，指定的回调函数就会被执行。

### 例子

分别注册两次onclick和click事件， onclick只输出一次， click输出两次。

```

<button id="btn" >点我</button>

<script>

    const btnEl = document.getElementById("btn");

    btnEl.onclick = () => console.log("onclick", 1);
    btnEl.onclick = () => console.log("onclick", 1);

    btnEl.addEventListener("click", ()=> console.log("click", 1));
    btnEl.addEventListener("click", ()=> console.log("click", 2));

</script>

```

```
onclick 1
```

```
click 1
```

```
click 2
```

## 总结

1. 内联事件是覆盖型，只能使用事件冒泡， addEventListener支持多个事件处理程序，并支持事件捕获。
2. 内联事件特定情况下可以被Node.cloneNode复制， addEventListener的不行  
更多细节参见 [Node.cloneNode](#)
3. addEventListener为DOM2级事件绑定， onclick为DOM0级事件绑定

## 按键时间 keypress, keydown

### [keypress](#)

当按下产生字符值的键时触发按键事件。产生字符值的键的示例有字母键、数字键和标点键。不产生字符值的键的例子是修改键，如 Alt、Shift、Ctrl 或 Meta。

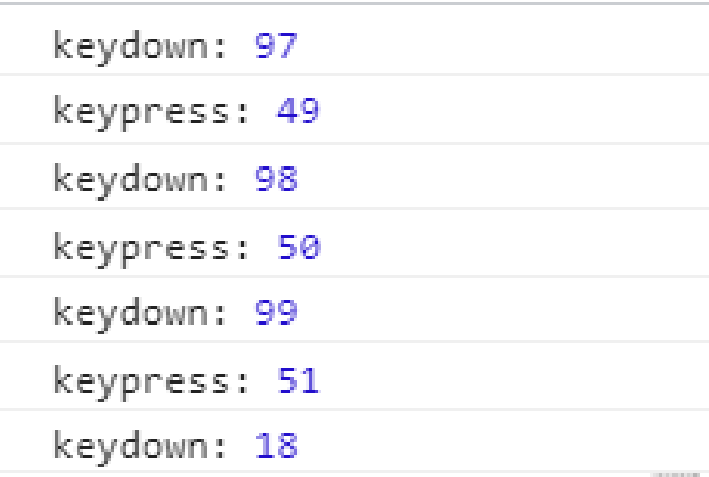
不再推荐使用此功能。尽管一些浏览器可能仍然支持它，但它可能已经从相关的 web 标准中删除

keydown

与keypress事件不同，无论是否生成字符值，所有键都会触发 keydown 事件。

例子

输入123，keydown和keypress的值keyCode一样



总结

- 1. 触发顺序keydown -> keypress
- 2. keydown: 当用户按下键盘上的任意键时触发；
- 3. keypress: 当用户按下键盘上的字符键时触发； 对中文输入法支持不好，无法响应中文输入
- 4. keypress的keyCode与keydown不是很一致；

异步加载脚本 defer,async

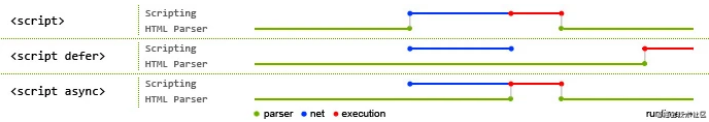
defer

异步加载，按照加载顺序执行脚本的

async

异步加载，乱序执行脚本。

这个一图胜千文



例子

四个script标签，两个async，两个defer。

代码内容如下： \* async1: console.log("async1"); \* async2: console.log("async2"); \* defer1: console.log("defer1"); \* defer2: console.log("defer2");

```
<script src="./async1.js" async ></script>
<div>
  sdfsd fsdfsdfsdfsdfd
</div>
```

```
<script src="./async2.js" async ></script>

<script src="./defer1.js" defer ></script>
<script src="./defer2.js" defer ></script>
```

× Expression
not available
async2
defer1
async1
defer2

not available
defer1
async1
async2
defer2

从上面可以看出，有时候 async2会比async1输出早，defer的输出也可能比async的输出早。但是defer的输出一定 defer1然后defer2

## 总结

1. 都是异步加载，defer会按照加载顺序执行，async乱序执行

[JS魔法堂：被玩坏的innerHTML、innerText、textContent和value属性](#)  
[keydown,keypress,keyup三者之间的区别](#)