

JavaScript

半知半解

TG 著



目 录

前言

JavaScript简介

基本概念

 语法

 数据类型

 运算符

 表达式

 语句

 对象

 数组

 函数

引用类型 (对象)

 Object对象

 Array对象

 Date对象

 RegExp对象

基本包装类型 (Boolean、Number、String)

单体内置对象 (Global、Math)

 console对象

DOM

DOM-属性和CSS

BOM

Event 事件

正则表达式

JSON

AJAX

表单和富文本编辑器

 表单

 富文本编辑器

canvas

离线应用

客户端存储 (Cookie、Storage、IndexedDB)

HTML5 API

 Video/Audio

 Geolocation API

 requestAnimationFrame

 File API

 FullScreen API

 IndexedDB

- [检测设备方向](#)
- [Blob](#)
- [vibrate](#)
- [Luminosity API](#)
- [WebRTC](#)
- [Page Visibility API](#)
- [Performance API](#)
- [Web Speech](#)
- [Notification](#)
- [面向对象的程序设计](#)
 - [概述](#)
 - [this关键字](#)
 - [原型链](#)
 - [作用域](#)
- [常用API合集](#)
- [SVG](#)
- [错误处理机制](#)
- [JavaScript开发技巧合集](#)
- [编程风格](#)
- [垃圾回收机制](#)

前言

前言

之前在作者的[博客](#)上洋洋洒洒地将之前学习JavaScript的笔记整理了出来，一共17篇，感觉查找和翻阅还是不方便，所以产生了编辑成电子书的念头，一来方便作者个人查找，二来方便后续内容的补充，三来也方便喜欢JavaScript的伙伴们阅读。

由于作者不是大牛级别的，之前也没有编辑电子书的经验，水平有限，书中内容有限，也难免会出现一些错误或者不准确的地方，恳求读者批评指正。

如果你不喜欢作者编排的内容，你可以看下面的资料，作者认为是很不错的。

参考资料：

书籍：

- 《JavaScript权威指南（第6版）》
- 《JavaScript高级程序设计》
- 《你不知道的JavaScript（上、中卷）》
- 《Effective JavaScript：编写高质量JavaScript代码的68个有效方法》

还有阮一峰大神的两个神作：

[JavaScript标准参考教程](#)

[ECMAScript 6入门](#)

注：未经作者许可，请不要将此书的电子版上传到任何网站，谢谢各位支持。

JavaScript简介

JavaScript简介

JavaScript诞生于1995年，它的主要目的是处理以前由服务器端语言负责的一些输入验证操作。

完整的JavaScript实现由下列三个不同的部分组成：

- 核心 (ECMAScript)
- 文档对象模型 (DOM)
- 浏览器对象模型 (BOM)

一、JavaScript简介

1.1 ECMAScript

ECMAScript 是由ECMA-262定义的，它提供了核心语言功能。

Web浏览器只是ECMAScript实现可能的宿主环境之一。宿主环境不仅提供基本的ECMAScript实现，同时也会提供该语言的扩展，以便语言与环境之间对接交互。

ECMA-262规定这门语言的下列组成部分：语法、类型、语句、关键字、保留字、操作符、对象
ECMAScript就是对实现该标准规定的各个方面内容的语言的描述。

ECMAScript的不同版本又称为版次，以第x版表示。目前最新的是ECMAScript 6.0 (简称 : ES6)

1.2 文档对象模型 (DOM)

文档对象模型 (DOM , Document Object Model) 是针对XML但经过拓展用于HTML的应用程序接口 (API , Application Programming Interface) 。

DOM把整个页面映射为一个多层节点的结构 (结构树) 。HTML或XML页面中的每个组成部分都是某种类型的节点，这些节点又包含着不同类型的数据。

DOM提供访问和操作网页内容的方法和接口。

1.3 浏览器对象模型 (BOM)

浏览器对象模型 (BOM , Browser Object Model) 是指可以访问和操作浏览器窗口的应用程序接口 (API)

BOM提供与浏览器交互的方法和接口。

二、在HTML中使用JavaScript

2.1 <script> 元素

向HTML页面中插入JavaScript的主要方法，就是使用 <script> 元素。

<script> 中定义了下列6个属性：

- `async` : 可选，表示应该立即下载脚本，但不应妨碍页面中的其他操作。只对外部脚本文件有效
- `charset` : 可选，表示通过src属性指定的代码的字符集，比较少用。
- `defer` : 可选，表示脚本可以延迟到文档完全被解析和显示之后再执行。只对外部脚本文件有效。

- `language` : 已废弃
- `src` : 可选，表示包含要执行代码的外部文件
- `type` : 可选，表示编写代码使用的脚本语言的内容类型（也称为MIME类型）。在HTML5中，默认是`text/javascript`，所以不需要设置。

使用 `<script>` 元素嵌入JavaScript代码，有两种方式：

第一种：直接在页面中嵌入JavaScript代码，包裹在 `<script>` 元素之间：

```
<script>
    console.log('Hello World');
</script>
```

注意：在使用 `<script>` 嵌入JavaScript代码时，切记不要在代码中的任何地方出现 `</script>`。

执行下面的代码时，会产生一个错误：

```
<script>
function loadScript(){
alert('</script>');
}
</script>
```

JavaScript代码的执行顺序：只要不存在`defer`和`async`属性，JavaScript代码就会从上至下依次解析。

第二种：使用外链脚本形式，必须有`src`属性，而且指定一个外部JavaScript文件的链接。

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

注意：带有`src`属性的标签之间再包含额外的JavaScript代码，嵌入代码会被忽略。

只要不存在`defer`和`async`属性，浏览器都会按照 `<script>` 出现的先后顺序对它们依次进行解析。

一般将全部JavaScript引用放在元素中页面的内容后面。

2.2 延迟脚本

当给 `<script>` 元素添加了 `defer` 属性时，`src`指向的外部文件会立即下载，但包含的脚本会延迟到浏览器遇到 `</html>` 标签（整个页面解析完毕）后再执行（按添加顺序执行），会先于`DOMContentLoaded`事件执行。

```
<script defer="defer" src="example.js"></script>
<script async src="example2.js"></script>
```

会先执行`example.js`，然后执行`example2.js`

注意：`defer`只适合外部脚本文件。

2.3 异步脚本

`async` 与 `defer` 属性类似，都用于改变处理脚本的行为，适用于外部脚本文件，并告诉浏览器立即下载，但标记为 `async` 的脚本并不保证按照指定它们的先后顺序执行。

```
<script async src="example.js"></script>
<script async src="example2.js"></script>
```

两个执行顺序不定。

指定 `async` 属性的目的是不让页面等待两个脚本下载和执行，从而异步加载页面其他内容。

注意：异步脚本不要在加载期间修改DOM。

异步脚本一定会在页面的load事件前执行，但可能会在DOMContentLoaded事件触发之前或之后执行。

2.4 使用外部文件的好处

- 可维护性：将JavaScript文件都放在一个文件夹中，比每次都需要到不同的HTML页面修改JavaScript方便维护。
- 可缓存：浏览器会缓存所有外部JavaScript文件，所以当你有多个页面使用同一个JavaScript脚本时，这个脚本只需下载一次。

2.5 `<noscript>` 元素

当浏览器不支持JavaScript或被禁用时，显示里面的内容。

```
<noscript>
  本页面需要浏览器支持（启用）JavaScript
</noscript>
```

小结

- JavaScript由 `ECMAScript`、`DOM`、`BOM` 三部分组成；
- `ECMAScript`由ECMA-262定义，提供核心语言功能；
- 文档对象模型（`DOM`），提供访问和操作网页内容的方法和接口；
- 浏览器对象模型（`BOM`），提供与浏览器交互的方法和接口；
- 把JavaScript插入到HTML页面中要使用 `<script>` 元素，可以内嵌，也可以外链文件；
- 使用 `defer` 属性可以让脚本在文档完全呈现之后再执行，延迟脚本总是按照它们的顺序执行；
使用 `async` 属性表示当前脚本不必等待其他脚本，也不必阻塞文档呈现，不能保证按照它们在页面中出现的顺序执行。
- 使用 `<noscript>` 元素可以指定在不支持脚本的浏览器中显示的提示内容。

基本概念

基础概念

俗话说，基础才是革命的本钱，对于技术来说，基础更是重中之重，这一章节会细分为八小类：

- 语法
- 数据类型
- 运算符
- 表达式
- 语句
- 对象
- 数组
- 函数

语法

语法

2.1 语法

JavaScript程序是用Unicode字符集编写的。

2.1.1 区分大小写

JavaScript中的一切（变量、函数名和操作符）都区分大小写。比如变量名test和变量名Test代表的是两个不同的变量。

2.1.2 标识符

标识符是指变量、函数、属性的名字，或函数的参数。

标识符格式规则：

- 第一个字符必须是一个字母、下划线（_）或一个美元符号（\$）
- 其他字符可以是字母、下划线、美元符号或数字。

按照惯例，JavaScript标识符采用 驼峰大小写格式，也就是第一个字母小写，剩下的每个有意义的单词的首字母大写。

```
myName
```

注意：不能把关键字、保留字、true、false和null用作标识符。

2.1.3 注释

单行注释：

```
// 单行注释
```

多行注释（/**/）：

```
/*
 * 多行注释，这一行星号非必需
 */
```

注意：多行注释（/**/）不能嵌套。

2.1.4 严格模式

ECMAScript 5 引入了 严格模式（strict mode）的概念。严格模式是为JavaScript定义了一种不同的解析与执行模型。

在严格模式下，ECMAScript 3中的一些不确定的行为将得到处理，而且对某些不安全的操作也会抛出错误。

可以在整个脚本中启用严格模式，也可以在函数内的顶部启用：

```
"use strict";
function doSomething(){
"use strict";
// 函数体
}
```

如果在代码中使用"use strict"开启了严格模式，则下面的情况都会在脚本运行之前抛出SyntaxError异常：

- 八进制语法:var n = 023和var s = "\047"
- with语句
- 使用delete删除一个变量名(而不是属性名):delete myVariable
- 使用eval或arguments作为变量名或函数名
- 使用未来保留字(也许会在ECMAScript 6中使用):implements, interface, let, package, private, protected, public, static, 和yield作为变量名或函数名
- 在语句块中使用函数声明: if(a<b){ function f(){}}}

其他错误

- 对象字面量中使用两个相同的属性名:{a: 1, b: 3, a: 7}
- 函数形参中使用两个相同的参数名:function f(a, b, b){}

2.1.5 语句

ECMAScript中的语句以一个分号结尾；如果省略分号，则由解析器确定语句的结尾。

强烈建议在可用可不用分号的地方使用分号，因为加上分号，可以避免很多意想不到的错误，而且也可以放心的使用压缩工具来压缩JavaScript脚本。

(1) 可选的行尾分号

JavaScript使用分号(;)将语句分隔开。

注意：JavaScript并不是在所有换行处都填补分号，只有在缺少了分号就无法正确解析代码的时候，才会填补分号。也可以说，如果当前语句和随后的非空格字符不能当做一个整体来解析时，JavaScript就在当前语句行结束处填补分号。比如下面代码：

```
var a
a
=
3
console.log(a)
```

JavaScript将其解析为：

```
var a;
a=3;
console.log(a);
```

JavaScript给第一行换行处添加了分号，因为没有分号，JavaScript就无法解析代码var a a。第二个a可以单独当做一条语句“a;”，但JavaScript并没有给第二行结尾填补分号，因为它可以和第三行内容一起解析成“a=3”。上面的代码解析后是没有问题的，可是没有主动添加分号，有些时候会导致意想不到的情形。比如：

```
var y=x+f
(a+b).toString()
```

解析后：

```
var y=x+f(a+b).toString();
```

如果当前语句和下一行语句无法合并解析，JavaScript则在第一行后填补分号，这是通用规则，但有两个例外。第一个例外是在涉及return、break和continue语句的场景总，如果这三个关键字后紧跟换行，JavaScript则会在换行处填补分号。

例如：

```
return
true;
```

JavaScript会解析成：

```
return;
true;
```

第二个例外是在涉及“++”和“--”运算符的时候。

```
x
++
y
```

将会解析成“x; ++y”，而不是“x++; y”

适当的添加分号，可避免意想不到的错误。

2.2 关键字和保留字

ECMA-262描述了一组具有特定用途的关键字，这些关键字可用于控制语句的开始或结束，或者用于执行特定操作等，不能用作标识符。

```
break  do  instanceof  typeof
case  else  new  var
catch  finally  return  void
continue  for  switch  while
debugger  function  this  with
default  if  throw  delete
in  try
```

ECMA-262还描述了一组不能用作标识符的保留字：

```
abstract  enum  int  short
boolean  export  interface  static
byte  extends  long  super
char  final  native  synchronized
class  float  package  throws
const  goto  private  transient
debugger  implements  protected  volatile
double  import  public
```

如果使用关键字作标识符，会导致“Identifier Expected”错误。

有些时候，我们不得不用到保留字货关键字的，比如CSS样式中的float，这时就需要这样：

```
style.cssFloat
```

2.3 变量

ECMAScript的变量是松散类型的，所谓松散类型就是可以用来保存任何类型的数据。定义变量时要使用var操作符，后跟变量名。

```
var name;
```

注意：使用var操作符定义的变量将成为定义该变量的作用域中的局部变量，也就是说，如果在函数中使用var定义一个变量，那么这个变量在函数退出后就会被销毁。

```
function test(){
var name = 'tg'; //局部变量
}
test();
console.log(name); // 报错
```

如果省略了var操作符，就相当于定义了一个全局变量，在函数外部的任何地方都可以访问到。

```
function test(){
  name = 'tg'; //局部变量
}
test();
console.log(name); // "tg"
```

不推荐省略var操作符。

还有一种隐式全局变量：

```
function test() {
  var a = b = 0;
}
test();
console.log(b); // 0
console.log(a); // ReferenceError: a is not defined
```

在上面的代码中，由于从右至左的操作符优先级，所以表达式“`b=0`”是先执行的，而此时`b`未经过声明，所以它会成为全局变量。

注意：

- 使用var创建的全局变量不能删除。
- 不适应var创建的隐含全局变量可以使用delete删除（因为它并不是真正的变量，而是全局对象window的属性）。

```
function test() {
  var a = b = 0;
  delete a;
  delete b;
  console.log(a); // 0
  console.log(b); // ReferenceError: b is not defined
}
test();
```

2.3.1 变量提升

变量提升是指所有变量的声明语句，都会被提升到代码的头部。

在函数内也一样，函数中的所有变量声明会在函数执行时被“提升”至函数体顶端。

看个例子：

```
console.log(a); // undefined
var a = 1;
```

```
function test(){
    console.log(a); // undefined
    var a = 2;
}
test();
```

结果是不是有点出乎你的意料。

其实JavaScript的执行环境分为声明阶段和执行阶段，因此对于上面的代码，JavaScript会这样解释代码：

```
var a;
console.log(a); // undefined
a = 1;
function test(){
    var a;
    console.log(a); // undefined
    a = 2;
}
test();
```

2.3.2 复制变量值

如果从一个变量向另一个变量复制基本类型的值，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上。

```
var num1 = 5;
var num2 = num1;
num2 += 5;
console.log(num1); // 5
console.log(num2); // 10
```

从上面例子的结果，我们知道num1中的5和num2中的5是完全独立的，num2中的5只是num1中的5的一个副本。

当从一个变量向另一个变量复制引用类型的值时，同样会将存储在变量对象中的值复制一份放到为新变量分配的空间中。不同的是，这个值的副本实际上是一个指针，而这个指针指向存储在堆中的一个对象。复制操作结束后，两个变量实际上将引用同一个对象。因此，改变其中一个变量，就会影响到另一个变量。

```
var obj1 = new Object();
var obj2 = obj1;
obj2.name = 'tg';
console.log(obj1.name); // "tg"
```

在上面的例子中，我们将obj1赋给了obj2，两者就指向了同一个对象，然后给obj2添加了一个属性name，接着访问obj1中的name，发现值和obj2中的name是一样，因为这两个变量引用的都是同一个对象。

注意：对象和原始值（布尔值、数字、字符串、null和undefined）之间的主要区别在于比较方式。

原始值比较的是值，只要编码值相同，则认为相同：

```
var a = 1;
var b = 1;
console.log(a === b); // true
```

而对象比较的是引用（也可以说是引用地址）：

```
var obj1 = {};
var obj2 = {};
console.log(obj1 === obj2); // false
console.log(obj1 == obj2); // false

console.log(obj1 === obj1); // true
```

2.4 空格、换行符

JavaScript会忽略程序中标识符之间的空格。在大多数情况下忽略换行符。

我们可以使用空格和换行来提高代码的可读性。

除了可以识别普通的空格符（\u0020），JavaScript还可以识别如下表示空格的字符：

- 水平制表符（\u0009）
- 垂直制表符（\u000B）
- 换页符（\u000C）
- 不中断空表（\u00A0）
- 字节序标记（\uFEFF）

JavaScript会将如下字符识别为行结束符：

- 换行符（\u000A）
- 回车符（\u000D）
- 行分隔符（\u2028）
- 段分隔符（\u2029）

回车符加换行符在一起被解析为一个单行结束符。

2.5 Unicode转义序列

JavaScript定义了一种特殊序列，使用6个ASCII字符来代表任意16位Unicode内码。这些Unicode转义序列码均以\u为前缀，其后跟随4个十六进制（使用数字以及大写或小写的字母A~F a~f表示）。

数据类型

2.4 数据类型

ECMAScript中有5种简单数据类型（基本数据类型）：`Undefined`、`Null`、`Boolean`、`Number`

和`String`。还有1种复杂数据类型--`Object`，`Object`本质上是由一组无序的名值对组成的。

注：在ES6中新增了`Symbol`。

2.4.1 `typeof`操作符

`typeof` 操作符用来检测给定变量的数据类型，可能的返回值：

```
"undefined" 这个值未定义
"boolean" 这个值是布尔值
"string" 这个值是字符串
"number" 这个值是数值
"object" 这个值是对象或null
"function" 这个值是函数
```

例子：

```
var name = 'tg';
console.log(typeof name); // "string"
console.log(typeof (name)); // "string"
console.log(typeof {}); // "object"
```

`typeof` 是一个操作符，后面的圆括号可有可无。

2.4.2 `Undefined`类型

`Undefined` 类型只有一个值，即特殊的`undefined`。

如果在使用`var` 声明变量但未对其加以初始化时，这个变量的值就是`undefined`。

```
var name;
console.log(name == undefined); // true
```

对于未声明过的变量，只能执行一项操作，即使用`typeof` 操作符检测其数据类型（对未声明的变量调用`delete` 不会导致错误（在非严格模式下））

2.4.3 `Null`类型

`Null` 类型也是只有一个值的数据类型，这个特殊值就是`null`。`null` 值可以看作是一个空对象指针。

```
console.log(typeof null); // "object"
```

`undefined` 其实是派生自 `null` 值：

```
console.log(null == undefined); // true
```

注意：`null`和`undefined`没有属性，甚至连`toString()`这种标准方法都没有。

2.4.4 Boolean类型

`Boolean`类型有两个字面值：`true`和`false`，两个值是区分大小写的。

要将一个值转换为其对应的`Boolean`值，可有使用转型函数`Boolean()`：

```
var name = 'tg';
console.log(Boolean(name)); // true
```

可以对任何类型的值调用`Boolean`函数，而且总会返回一个`Boolean`值（`true`或`false`）

转换规则：

- 对于`true`或`false`，返回原值（`true`或`false`）
- 对于`String`类型的值，任何非空字符串返回`true`，空字符串（`" "`）返回`false`
- 对于`Number`类型的值，任何非零数字值（包括无穷大），返回`true`；`0`和`NaN`返回`false`
- 对于`Object`类型的值，任何对象返回`true`，`null`返回`false`
- 对于`Undefined`类型，`undefined`返回`false`（只有一个值）

2.4.5 Number类型

最基本的数值字面量格式是十进制整数。

```
var num = 15;
```

八进制（以8为基数）以`0`开头，后面是（`0~7`）

```
var num = 070; // 八进制的56
```

十六进制（以16为基数）以`0x`开头，后面是（`0~9及A~F`），字母`A~F`可以大小写。

```
var num = 0xA; //十六进制的10
```

注意：在进行算术计算时，所有以八进制和十六进制表示的数值都会被转换成是十进制。

1.浮点数值

`浮点数值` 指包含一个小数点，并且小数点后面必须至少有一个数字。

```
var floatNum = 1.1;
```

保存浮点数值需要的内存空间是保存整数值的两倍。

对于那些极大或极小的数值，我们可以使用科学计数法（e表示法）来表示浮点数值。

e表示法表示的数值等于e前面的数值乘以10的指数次幂：

```
var floatNum = 3.125e7; // 31250000  
var floatNum1 = 3e-7; // 0.0000003
```

默认情况下，ECMAScript会将那些小数点后面带有6个零以上的浮点数值转换为以e表示的数值。

```
var floatNum1 = 3e-7; // 0.0000003
```

浮点数的最高精度是17位小数。

```
0.1 + 0.2 = 0.3000000000000004; // 不是等于0.3
```

2. 数值范围

基于内存的限制，ECMAScript只能保存有限的数值。

ECMAScript能够表示的最小数值保存在 `Number.MIN_VALUE`（最小值）中，这个值是 $5e-324$ ；能够表示的最大数值保存在 `Number.MAX_VALUE`（最大值）中，这个值是 $1.7976931348623157e+308$ 。

如果某次计算的结果得到了一个超出JavaScript数值范围的值，那么这个数值就会被转换成特殊的 `Infinity` 值；如果这个值是负数，则会被转换为- `Infinity`（负无穷），如果这个数值是整数，则会转换成 `Infinity`（正无穷）。

注意：`Infinity`是不能参与计算的数值。

用 `isFinite()` 来判断这个值是否无穷，该函数接受一个参数。如果参数位于最小与最大数值之间，返回 `true`。

```
console.log(isFinite(1)); // true  
console.log(isFinite(Infinity)); // false
```

3. NaN

`NaN`（Not a Number）表示非数值，这个数值用于表示一个本来要返回数值的操作数未返回数值的情况。

注意点：

- 任何涉及`NaN`的操作都会返回`NaN`。
- `NaN`与任何值都不相等，包括`NaN`本身。

```
console.log(NaN == NaN); //false
```

我们可以用 `isNaN()` 函数来判断是否非数值，该函数接受一个参数，可以是任何类型。

```
console.log(isNaN(NaN)); // true
```

`isNaN()` 在接收到一个值（可以是任何类型）之后，会尝试将这个值转换为数值，某些不是数值的值会直接转换为数值，比如：字符串"10"或Boolean值。而任何不能被转换为数值的值都会导致这个函数返回true。

```
console.log(isNaN(NaN)); // true
console.log(isNaN(10)); // false
console.log(isNaN('blue')); // true
```

4. 数值转换

有3个函数可以把非数值转换为数值：`Number()`、`parseInt()` 和 `parseFloat()`

转型函数 `Number()` 可以用于任何数据类型，后面两个是专门用于把字符串转换为数值。

`Number()` 函数的转换规则：

- 如果是Boolean值，`true`和`false`将分别转换为1和0
- 如果是数字值，只是简单的传入和返回
- 如果是`null`值，返回0
- 如果是`undefined`，返回`Nan`
- 如果是字符串，遵循下列规则：
 - (
 - 如果是字符串中只包含数字（包括前面带正负号），则将其转换为十进制数值（前导的零会被忽略）
 - 如果字符串中包含有效的浮点格式，如`1.1`，则将其转换为对应的浮点数值
 - 如果字符串中包含有效的十六进制，如`0xf`，则将其转换为相同大小的十进制数值
 - 如果字符串是空的，返回0
 - 如果字符串中包含上述格式以外的字符，返回`Nan`
- 如果是对象，则调用对象的`valueOf()`方法，然后依照前面的规则转换返回的值。如果转换的结果是`Nan`，则调用对象的`toString()`方法，然后再次依照前面的规则转换返回字符串值

```
console.log(Number('tg')); // Nan
console.log(Number(' ')); // 0
console.log(Number('0011')); // 11
console.log(Number(true)); //1
```

一元加操作符的操作与Number函数规则相同。

parseInt()

`parseInt()` 会忽略字符串前面的空格，直到找到第一个非空格字符。如果第一个字符不是数字字符或负号，就会返回NaN。如果第一个字符是数字字符，就会继续解析，直到解析完所有后续字符或者遇到了一个非数字字符。

`parseInt()` 也能识别八进制（在ECMAScript 5中无法识别，将开头的0当作0）和十六进制，最后会转换成十进制。

```
console.log(parseInt('123tg')); // 123
console.log(parseInt(' ')); // NaN
console.log(parseInt('070')); // 70
console.log(parseInt('0xf')); // 15
console.log(parseInt(22.5)); // 22
```

我们还可以为 `parseInt()` 提供第二个参数，指定需要转换的进制。

```
console.log(parseInt('0xAF', 16)); // 175
console.log(parseInt('AF', 16)); // 175
console.log(parseInt('AF')); // NaN
console.log(parseInt('070', 8)); // 56
console.log(parseInt('70', 8)); // 56
```

如果提供了第二个参数，要转换八进制和十六进制时，可省略0和0x。

`parseFloat()` 和 `parseInt()` 类似，也是从第一个字符（位置0）开始解析每个字符，而且一直解析到字符串末尾，或者解析到遇到一个无效的浮点数字字符为止，换句话说，字符串中的第一个小数点是有效的，后面的小数点是无效的，它还会忽略前导的零，只会解析十进制值。

```
console.log(parseFloat('123tg')); // 123
console.log(parseFloat('22.12.4')); // 22.12
console.log(parseFloat('070')); // 70
console.log(parseFloat('0xf')); // 0
console.log(parseFloat(22.5)); // 22.5
```

2.4.6 String类型

String类型用于表示由零或多个16位Unicode字符组成的字符序列，即字符串。字符串可以由双引号或单引号表示。

```
var name = 'tg';
```

用双引号表示的字符串和用单引号表示的字符串完全相同，但要确保引号前后一致，也就是说以双引号开头的必须以双引号结尾，以单引号开头的必须以单引号结尾。

在ECMAScript 3中，字符串直接量必须写在一行中，而在ECMAScript 5中，字符串直接量可以拆分成数行，每行必须以反斜杠 (\) 结束。

```
// ES 5
'Hello \
world'
```

1.字符字面量

String数据类型包含了一些特殊的字符字面量，也叫转义序列。

\n	换号
\t	制表
\b	空格
\r	回车
\f	进纸
\\	斜杠
\'	单引号
\"	双引号
\xnn	以十六进制代码nn表示的一个字符（n为0~F），比如：\x41表示"A"
\unnnn	以十六进制代码nnnn表示的一个Unicode字符（n为0~F）。比如：\u03a3表示Σ

这些字符字面量可以出现在字符串中的任意位置，会被当做一个字符来解析。

```
var name = 'tg\u03a3';
console.log(name); //tgΣ
console.log(name.length); // 3
```

任何字符串的长度都可以访问其length属性取得。

2.字符串的特点

ECMAScript中的字符串是不可变的。一旦创建，值就不会改变。

后台逻辑：要改变某个变量保存的字符串，首先要销毁原来的字符串，然后用另一个包含新值的字符串填充该变量。

3.转换为字符串

要将一个值转换为字符串有两种方式：

- 使用每个值（null和undefined除外）都有的 `toString()` 方法

```
var age = 1;
console.log(age.toString()); // "1"
```

我们还可以给 `toString()` 传入一个参数，输出数值的基数。

```
var num = 10;
console.log(num.toString()); // "10"
console.log(num.toString(2)); // 1010
console.log(num.toString(8)); // 12
```

- 可以使用转型函数`String()`，能将任何类型的值转换为字符串，转换规则：

如果值有`toString()`方法，则调用该方法并返回相应结果

如果值是`null`，则返回`null`

如果值是`undefined`，则返回`undefined`

```
console.log(String(10)); // "10"
console.log(String(true)); // "true"
console.log(String(null)); // "null"
console.log(String(undefined)); // "undefined"
```

要把某个值转为字符串，还可以使用加号操作符：

```
console.log(true + ''); // "true"
```

2.4.7 Object类型

ECMAScript中的对象其实就是一个组数据和功能的集合。

对象可通过执行`new`操作符后跟创建的对象类型的名称来创建。

```
var o = new Object();
```

如果不传参，可以省略后面的圆括号（不推荐）。

```
var o = new Object;
```

在ECMAScript中，`Object` 类型是所有对象的基础。

`Object` 的每个实例都具有下列属性和方法：

- `Constructor`：保存着用于创建当前对象的函数，比如上面的例子，构造函数就是`Object()`
- `hasOwnProperty(propertyName)`：用于检查给定的属性在当前对象实例中是否存在（而不是在实例的原型中），参数必须是字符串形式
- `isPrototypeOf(object)`：用于检查传入的对象是否是另一个对象的原型

- `propertyIsEnumerable(propertyName)` : 用于检查给定的属性是否能够使用for-in语句来枚举，参数必须是字符串形式
- `toLocaleString()` : 返回对象的字符串表示，该字符串与执行环境的地区对应
- `toString()` : 返回对象的字符串表
- `valueOf()` : 返回对象的字符串、数值或布尔值表示，通常和`toString()`返回的值相同

运算符

运算符

ECMA-262描述了一组用于操作数据值的运算符，包括算术运算符、位运算符、关系运算符和相等运算符。

1.1 一元运算符

只能操作一个值的运算符叫做一元运算符。

1.递增和递减

递增（`++`）和递减（`--`）都有两种使用方式：前置型和后置型。

前置型指的是运算符放在要操作的变量之前，而后置型则是运算符放在要操作的变量之后。

```
var age = 10;
var age2 = 10;
++age;
--age;
console.log(age); // 11
console.log(age2); //9
```

执行前置递增和递减操作时，变量的值都是在语句被求值以前改变的（也就是先自增或自减，再和其他值进行操作）：

```
var num = 2;
var age = ++num + 2;
console.log(age); // 5
```

在上面的例子中，在与2相加之前，`num`已经加1变成了2。

后置型递增和递减运算符不变（依然是`++`和`--`），不过除了要放到变量之后外，包含它们的语句是被求值之后才执行（先和其他值进行操作后，再自增或自减）：

```
var num = 2;
var age = (num++) + 2;
console.log(age); // 4
console.log(num); // 3
```

注意：由于JavaScript会自动进行分号补全，因此不能在后增量运算符和操作数之间插入换行符

```
num
++;
```

这会报错

这4个操作符对任何值都适用，不过，当应用于不同的值时，递减和递增操作符会遵循下列规则：

- 当操作数是一个包含有效数字字符的字符串，系统会将其转换为数字值，再执行递减或递增。
- 当操作数是一个不包含有效数字字符的字符串，系统将变量的值设置为NaN
- 当操作数是布尔值，会将其转为数值（true转为1，false转为0）再操作。
- 当操作数是浮点数值，直接执行递减或递增
- 当操作数是对象，先调用对象的valueOf()方法取得一个可供操作的值，然后再遵循上面的三条规则。如果结果是NaN，则在调用toString()方法后再遵循上面的规则转换。

```
var a = '2';
var b = 'a';
var c = false;
var d = 1.1;
var o = {
  valueOf: function() {
    return -1;
  }
};
a++; // 3
b++; // NaN
c--; // -1
d--; // 0.1000000000000009 (浮点数操作结果，类似0.1+0.2 != 0.3)
o--; -2
```

2.一元加和减运算符

一元加（+）和减（-）运算符和数学上的含义一样，表示负正。

不过，对非数值应用一元加运算符时，该运算符就像Number()转型函数一样对这个值执行转换。（转换规则参考数据类型那一节中的Number()的转换规则）

一元减运算符对数值是表示负数，而对于非数值，转换规则和一元加（也就是和Number()）一样，然后将转换后的数值转为负数。

1.2 位运算符

位运算符是按内存中表示数值的位来操作数值。

ECMAScript中的数值都以IEEE-75464位格式存储，但位运算符并不直接操作64位的值，而是先将64位的值转换为32位，然后执行操作，最后将结果转换回64位。

对于有符号的整数，32位（从右往左，最右边为第一位）中的前31位用于表示整数的值，第32位用于表示数值的符号：0表示正数，1表示负数。这个表示符号的位叫做符号位。符号位的值决定了其他位数值的格式。

正数以纯二进制格式存储，31位中的每一位（从右往左）都表示2的幂。第一位（叫做位0）表示 2^0 ，第二位表示 2^1 ，以此类推。没有用到的位以0填充，可以忽略。

18

二进制表示法 : 10010

$$2^4 \times 1 + 2^3 \times 0 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0 = 18$$

负数同样以二进制码存储，但使用的格式是二进制补码。

如何计算一个数值的二进制补码？需要3个步骤：

- 求这个数值绝对值的二进制码
- 求二进制反码，即0替换1，将1替换0
- 得到二进制反码加1

-18

```
//第一步，求18的二进制码
0000 0000 0000 0000 0000 0001 0010
// 求反码，后加1
1111 1111 1111 1111 1111 1111 1110 1101
1111 1111 1111 1111 1111 1111 1110 1110 // -18的二进制表示
```

1

如果对非数值应用位运算符，会先使用Number()函数将该值转换为一个数值（自动转换），再进行位运算符操作，最后得到一个数值

当你去求负数的二进制时，ECMAScript会尽力向我们隐藏所有这些信息。比如：

```
var num = -18;
console.log(num.toString(2)); // "-10010"
```

上面的代码对负数求二进制码，得到的结果只是这个负数绝对值的二进制码前面加上一个负号。

(1) 按位非 (NOT)

按位非运算符(~)进行 否运算，执行按位非的结果就是返回数值的反码。

```
~ 3 // -4
```

上面的表达式对3进行否运算，得到-4。

```
//3的二进制码
00000000000000000000000000000011

// 取反码
11111111111111111111111111111100
```

```
//补码转为十进制的规则，在下面可以看到  
11111111111111111111111111111111011 //减1  
  
00000000000000000000000000000000100 //再次取反得到~3的十进制，也就是4，加上负号，就是-4
```

负数二进制码（补码）转换为十进制的规则：需要将这个数减去1，再取一次反，然后加上负号，才能得到这个负数对应的10进制值。

看起来是不是眼花缭乱，其实我们可以不用管里面的转换逻辑，只需记得按位非（否运算）就是一个数与自身的取反值相加，等于-1。

```
~3 + 3 = -1 => -1 - 3 = -4 == ~3
```

2.按位与(AND)

使用按位与（&）运算符来进行 与运算，它有两个运算符数。

```
0 & 3 //0
```

0和1的二进制是00和11

```
00  
11  
00
```

运算规则是只有两个数值的对应位都是1时才返回1，其他情况返回0。

按位或 (OR)

使用按位或（|）运算符进行 或运算。

```
0 | 3 // 3
```

```
00  
11  
11
```

或运算的规则是只要有一个位是1就返回1，只有两个位都是0时才返回0

位运算只对整数有效，遇到小数时，会将小数部分舍去，只保留整数部分。所以，将一个小数与0进行或运算，等同于对该数去除小数部分，即取整数位。

4.按位异或 (XOR)

使用按位异或（^）来进行 异或运算。

```
0 ^ 3 // 3
```

11

异或运算的规则是两个数值的对应位上只有一个1时才返回1，其他情况返回0。

5.左移

左移运算符（<<），这个运算符会将所有位向左移动指定的位数，尾部补0。

```
2 << 5 // 64
2的二进制码
10
1000000
```

左移可看做是数值乘以2的指定次方

注意：左移不会影响操作数的符号位，也就是说，-2向左移动5位后，是-64

6.右移

(1) 有符号的右移

有符号的右移（>>）将数值向右移动，但保留符号位（即正负号标记，也就是第32位）

```
64 >> 5 // 2
1000000
10

-64 >> 5 // -2
```

有符号的右移可看做是数值除以2的指定次方

(2) 无符号的右移

无符号的右移（>>>），这个运算符会将数值的所有32位向右移动。对正数来说，无符号右移的结果与有符号右移相同：

```
64 >>> 5 // 2
```

但负数就不一样了，无符号右移是以0来填充空位的，所以无符号右移得到的结果都是正数。

```
-64 >>> 5 // 134217726
```

有一点需要特别注意，位运算符只对整数起作用，如果一个运算子不是整数，会自动转为整数后再执行。

1.3 布尔运算符

布尔运算符是用来比较两个值的关系的。

1. 逻辑非（or）

逻辑非 (!) 可以应用于JavaScript中的任何值，最终都会返回一个布尔值。

逻辑非运算符首先会将它的操作数转换为一个布尔值，然后再取反。

逻辑非规则：

- 如果操作数是一个对象、非空字符串、任意非0数值（包括Infinity），则返回false
- 如果操作数是空字符串、数值0、null、undefined、NaN，则返回true

如果同时使用两个 (!!)，就会像使用Boolean()转型函数一样的转换逻辑。

```
!false // true
!'tg' // false
!'' // true

!!0 //false
```

2.逻辑与

逻辑与 (&&) 有两个操作数，如果是布尔值，只有两个都是true时，才会返回true，否则返回false

```
var isTrue = true && false; //false
```

如果不是布尔值，它遵循下面的规则：

- 如果第一个操作数是对象，则返回第二个操作数
- 如果第二个操作数是对象，则只有在第一个操作数的求值为true时才会返回第二个操作数
- 如果有一个操作数是null，则返回null
- 如果有一个操作数是NaN，则返回NaN
- 如果有一个操作数是undefined，则返回undefined

逻辑与操作符也就是先将第一个操作数转换为Boolean类型判断是true或false，再根据结果决定是否执行第二个操作数

```
0 && 'tg' ; // 0
{} && 'tg'; // "tg"
```

逻辑与操作属于短路操作，也就是说如果第一个操作数能够决定结果（等于false时），就不会再对第二个操作数求值。

3.逻辑或

逻辑或 (||) 也有两个操作数。

如果两个操作数都是布尔值，则有一个为true时，就返回true，否则返回false。

如果有一个操作数不是布尔值，则遵循下列规则：

- 如果第一个操作数是对象，则返回第一个操作数
- 如果第一个操作数的求值结果是false，则返回第二个操作数
- 如果两个操作数都是对象，则返回第一个操作数
- 如果两个操作数都是null，则返回null
- 如果两个操作数都是NaN，则返回NaN
- 如果两个操作数都是undefined，则返回undefined

逻辑或也是短路运算符，也就是说，如果第一个操作数的求值结果为true，就不会对第二个操作数求值了。

```
true || 'tg'; // true
0 || 'tg'; // "tg"
```

一般情况下，我们可以使用逻辑或来避免变量赋null或undefined值：

```
function test(name){
  name = name || 'tg';
  console.log(name);
}
test(); // "tg"
test('tg2'); // "tg2"
```

上面的例子，表示当调用test()方法不传参时，name赋予一个默认值"tg"；如果带有参数，则使用参数值。

注意：逻辑与（&&）和逻辑或（||）返回的都是运算值。

1.4 乘性运算符

ECMAScript定义了3个乘性运算符：乘法、除法和求模。

当操作数是非数值时，会执行自动的类型转换。如果操作数不是数值，会先使用Number()转型函数将其转换为数值（后台自动），再进行运算。

1.乘法

乘法运算符（*），用于计算两个数值的乘积。

处理特殊值时，乘法运算符会遵循下列规则：

- 如果操作数都是数值，但乘积超过了ECMAScript数值范围，则返回Infinity或-Infinity
- 如果有一个操作数是NaN，结果是NaN
- 如果是Infinity乘以0，结果是NaN
- 如果是Infinity与非0数值相乘，结果是Infinity或-Infinity，取决于非0数值的符号
- 如果是Infinity与Infinity相乘，结果是Infinity
- 如果有一个操作数不是数值，则在后台调用Number()将其转换为数值，然后遵循上面的规则

```
console.log(1 * NaN); // NaN
console.log(Infinity * 2); // Infinity
console.log(Infinity * 0); // NaN
console.log(Infinity * Infinity); // Infinity
```

2.除法

除法运算符 (/) , 执行第二个操作数除第一个操作数计算。

处理特殊值 , 规则如下 :

- 如果操作数都是数值 , 但商超过了ECMAScript的表示范围 , 则返回Infinity或-Infinity
- 如果有一个操作数是NaN , 结果是NaN
- 如果是Infinity被Infinity除 , 结果是NaN
- 如果是零被零除 , 结果是NaN
- 如果是非零的有限数被零除 , 结果是Infinity或-Infinity , 取决于有符号的操作数
- 如果是Infinity被任何非零数值除 , 结果是Infinity或-Infinity
- 如果有一个操作数不是数值 , 则在后台调用Number()将其转换为数值 , 然后遵循上面的规则。

```
console.log(NaN / 1); // NaN
console.log(0 / 0); // NaN
console.log(1 / 0); // Infinity
console.log(2 / Infinity); // 0
console.log(Infinity / Infinity); // NaN
console.log(Infinity / 2); // Infinity
```

3.求模

求模 (余数) 运算符 (%)

处理特殊值 , 规则如下 :

- 如果被除数是无穷大值而除数是有限大的数值 , 结果是NaN
- 如果被除数是有限大的数值而除数是零 , 结果是NaN
- 如果是Infinity被Infinity除 , 结果是NaN
- 如果被除数是有限大的数值而除数是无穷大的数值 , 结果是被除数
- 如果被除数是零 , 结果是零
- 如果有一个操作数不是数值 , 则在后台调用Number()将其转换为数值 , 然后遵循上面的规则。

```
console.log(5 % 3); // 2
```

1.5加性运算符

加性运算符也会在后台转换不同的数据类型。

1. 加法

如果两个操作数都是数值，执行常规的加法计算，然后根据下面的规则返回结果：

- 如果有一个操作数是NaN，结果是NaN
- 如果Infinity加Infinity，结果是Infinity
- 如果是-Infinity加-Infinity，结果是-Infinity
- 如果是Infinity加-Infinity，结果是NaN
- 如果是+0加+0，结果是+0
- 如果是-0加-0，结果是-0
- 如果是+0加-0，结果是+0

如果有一个操作数是字符串，则遵循下列规则：

- 如果两个操作数都是字符串，则拼接两个字符串
- 如果只有一个操作数是字符串，则将另一操作数转换为字符串，然后拼接
- 如果有一个操作数是对象、数值或布尔值，则调用它们的toString()方法取得相应的字符串值，然后遵循上面的规则。对于undefined和null，则分别调用String()函数并取得字符串"undefined"和"null"。

```
5 + 5    //10
5 + '5'  // "55"
```

如果你要在有字符串操作数的加法中执行常规的加法操作，应该使用圆括号将要相加的数值括起来：

```
'tg' + 5 + 5 // "tg55"
'tg' + (5 + 5) // "tg10"
```

2. 减法

减法运算符 (-)

对于特殊值，减法操作会遵循下列规则：

- 如果有一个操作数是NaN，结果是NaN
- 如果Infinity减Infinity，结果是NaN
- 如果是-Infinity减-Infinity，结果是NaN
- 如果是Infinity减-Infinity，结果是Infinity
- 如果是-Infinity减Infinity，结果是-Infinity
- 如果是+0减+0，结果是+0
- 如果是-0加-0，结果是+0
- 如果是+0减-0，结果是-0

- 如果有一个操作数是字符串、布尔值、null或undefined，则先在后台调用Number()将其转换为数值，然后遵循上面的规则进行计算。
- 如果有一个操作数是对象，则调用对象的valueOf()方法以取得表示该对象的数值；如果该对象没有valueOf()方法，则调用其toString()方法将得到的字符串转换为数值，然后遵循上面的规则进行计算。

```
5 - true; // 4 (true转换成1)
5 - '2' // 3
5 - null; // 5 (null转换成0)
```

1.6 关系运算符

小于(<)、大于(>)、小于等于(<=)和大于等于(>=)四个关系运算符用来对两个值进行比较，最后返回一个布尔值。

当使用了非数值时，则会遵循下列规则：

- 如果两个操作数是字符串，则比较两个字符串对应的字符编码值
- 如果一个操作数是数值，则将另一个操作数转换为一个数值，然后进行比较
- 如果一个操作数是对象，则调用这个对象的valueOf()方法，用得到的结果按照前面的规则比较。如果对象没有valueOf()方法，则调用toString()方法，并用得到的结果按照上面的规则进行比较。
- 如果一个操作数是布尔值，则先将其转换为数值，然后进行比较。

注意：在使用关系运算符比较两个字符串时，比较的是两个字符串中对应位置的每个字符的字符编码值（从左往右），经过逐个比较后，直到有胜负（也就是不相等时），再返还布尔值。

```
'abc' > 'abd' //false
'A' < 'a' // true
'23' < '3' // true
```

在上面的例子中，第一行代码会先比较“a”和“a”，然后是“b”和“b”，最后是“c”和“d”，由于c的字符编码值是63，d的字符编码值是64，所以返还false。

由于小写字母的字符编码值总是大于大写字母的字符编码值，所以第二行代码返还true。

第三行代码也是字符串比较，所以比较的是字符编码值（2是50,3是51），所以返回true

注意：任何操作数与NaN比较，都会返回false。

1.7 相等运算符

相等运算符有两组：相等和不相等（先转换再比较）、全等和不全等（只比较不转换）

1.相等和不相等

相等(==)，不相等(!=)

这两个运算符都会先转换操作数（强制转换），然后比较。

对于不同的数据类型，转换规则如下：

- 如果有一个操作数是布尔值，则在比较前先将其转换为数值（`false`转为0，`true`转为1）
- 如果一个操作数是字符串，另一个操作数是数值，先将字符串转为数值
- 如果一个操作数是对象，另一个不是，先调用对象的`valueOf()`方法，用得到的基本类型值按照前面的规则进行比较
- `null`和`undefined`是相等的
要比较之前，不能将`null`和`undefined`转换成其他任何值
- 如果有一个操作数是`Nan`，则相等运算符返回`false`，不相等运算符返回`true`。（要记住`Nan`不等于本身原则）
- 如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等运算符返回`true`，否则返回`false`。

```
NaN == NaN // false
true == 1 // true
null == undefined // true
```

2.全等和不全等

除了在比较之前不转换操作数类型之外，全等和不全等与相等和不相等没什么区别。

```
'55' == 55 // true
'55' === 55 // false
```

1.8 条件运算符

条件运算符是ECMAScript中唯一的三元运算符。

```
variable = boolean_expression ? true_value : false_value;
```

如果`boolean_expression`返回`true`，则执行`true_value`，否则，执行`false_value`

```
var name = (1 > 2) ? 'tg' : 'tg2'; // "tg2"
```

上面的代码中`1`小于`2`，所以是`false`，则将"tg2"赋值给`name`。

1.9 赋值运算符

`(=)`是最简单的赋值操作，其作用是把右侧的值赋给左侧变量。

```
var name = 'tg';
```

复合赋值运算符

乘赋值 (<code>x *= y</code>)	等同于	<code>x = x * y</code>
除赋值 (<code>x /= y</code>)	等同于	<code>x = x / y</code>
模赋值 (<code>x %= y</code>)	等同于	<code>x = x % y</code>
加赋值 (<code>x += y</code>)	等同于	<code>x = x + y</code>
减赋值 (<code>x -= y</code>)	等同于	<code>x = x - y</code>
左移赋值 (<code>x <= y</code>)	等同于	<code>x = x << y</code>
有符号右移赋值 (<code>x >= y</code>)	等同于	<code>x = x >> y</code>
无符号右移赋值 (<code>x >>= y</code>)	等同于	<code>x = x >>> y</code>

例子：

```
var a = 1;
a += 1;
console.log(a); // 2
```

1.10 逗号运算符

使用逗号运算符可以在一条语句中执行多个操作：

```
var name = 'tg', age = 1;
```

逗号运算符多用于声明多个变量。

逗号运算符还可以用于赋值。在用于赋值时，逗号运算符总会返回表达式中的最后一项：

```
var num = (1, 5, 3); // num的值为3
```

3是表达式中的最后一项，因此num的值就是3.

1.11 in运算符

in运算符希望它的左操作数是一个字符串或可以转换成为字符串，希望它的右操作数是一个对象。如果右侧的对象拥有一个名为左操作数值的属性名，则返回true。

```
var o = {x:1};
"x" in o //true
```

1.12 instanceof运算符

`instanceof` 运算符希望左操作数是一个对象，右操作数标识对象的类。如果左侧的对象是右侧类的实例，则表达式返回true。

```
var a = new Array();
a instanceof Object; //true
```

注意：所有对象都是Object的实例

1.13 typeof运算符

`typeof` 是一元运算符，用来判断数据类型：

```
typeof 1 // "number"
```

1.14 delete运算符

`delete` 是一元运算符，它用来删除对象的属性或数组元素。

```
var o={x:1}
delete o.x;
"x" in o //false
```

1.15 void运算符

`void` 是一元运算符，它出现在操作数之前，操作数可以是任意类型。操作数会照常计算，但忽略计算结果并返回`undefined`。

```
void 0 //undefined
void(0) //undefined

var a = 1;
void (a=2);
a //2
```

这个运算符主要是用于书签工具（bookmarklet），以及用于在超级链接中插入代码，目的是返回`undefined`可以防止网页跳转。

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

2. 运算顺序

2.1 优先级

JavaScript各种运算符的优先级别（Operator Precedence）是不一样的。优先级高的运算符先执行，优先级低的运算符后执行。

2.2 圆括号

圆括号（()）可以用来提高运算的优先级，因为它的优先级是最高的，即圆括号中的表达式会第一个运算。

注意：因为圆括号不是运算符，所以不具有求值作用，只改变运算的优先级。

对于优先级别相同的运算符，大多数情况，计算顺序总是从左到右，这叫做运算符的“左结合”（left-to-right associativity），即从左边开始计算。

但是少数运算符的计算顺序是从右到左，即从右边开始计算，这叫做运算符的“右结合”（right-to-left associativity）。其中，最主要的是赋值运算符（=）和三元条件运算符（?:）。

表达式

表达式

表达式 (expression) 是JavaScript中的一个短语，JavaScript解释器会将其计算出一个结果。

将简单表达式组合成复杂表达式最常用的方法就是使用运算符 (operator)。运算符按照特定的运算规则对操作数进行运算，并计算出新值。

1、表达式

1.1 原始表达式

原始表达式是表达式的最小单位---它们不包含其他表达式。

JavaScript中的原始表达式包含常量、直接量、变量或关键字。

直接量是直接在程序中出现的常数值。

```
1.2
"hello"
//保留字
true
false
null
this
//变量
i
num
```

1.2 对象和数组的初始化表达式

对象和数组初始化表达式实际上是一个新创建的对象和数组。也可称为“对象直接量”和“数组直接量”。

```
var arr = []
var p = {};
```

注意：JavaScript对数组初始化表达式和对象初始化表达式求值的时候，数组初始化表达式和对象初始化表达式的元素表达式也都会各自计算一次。也就是说，元素表达式每次计算的值有可能是不同的。

1.3 函数定义表达式

函数定义表达式（函数直接量）定义一个JavaScript函数。表达式的值是这个新定义的函数。

```
var f = function(){};
```

1.4 属性访问表达式

属性访问表达式运算得到一个对象属性或一个数组元素的值。

```
var arr =[1];
var p = {x:1}
arr[0]
p.x
```

注意：在 “.” 和 “[” 之前的表达式总是会首先计算，如果计算结果是null或undefined，表达式会抛出一个类型错误异常，因为这两个值都不能包含任何属性。

1.5 调用表达式

JavaScript中的调用表达式是一种调用函数或方法的语法表示。

```
f()
a.sort()
```

1.6 对象创建表达式

对象创建表达式创建一个对象并调用一个函数（这个函数称做构造函数）初始化新对象的属性。

```
new Object()
```

JavaScript中的大多数运算符是一个二元运算符。不过，JavaScript支持一个三元运算符（ternary operator），条件判断运算符“?:”，它将三个表达式合并成一个表达式。

语句

语句

表达式在JavaScript中是短语，而语句（ statement ）就是JavaScript整句或命令。

JavaScript语句是以分号结束。

默认情况下，JavaScript解释器依照语句的编写顺序依次执行。

1. 表达式语句

(1) 复合语句

我们可以用花括号将多条语句括起来，这就是复合语句（语句块）。

```
{
  var a = 1;
  var b = 1;
}
```

(2) 空语句

分号前面可以没有任何内容，JavaScript引擎将其视为空语句。

```
;
```

当你使用空语句时，最好加上注释。

2. 声明语句

`var` 和 `function` 都是声明语句，它们声明或定义变量或函数。

`var` 语句用来声明一个或多个变量。

```
var a = 1;
var a = 1, b = 2;
```

注意：如果 `var` 语句中的变量没有指定初始化表达式，那么这个变量的初始值为`undefined`。

关键字 `function` 是用来定义函数的。

```
function a(){};
```

3. 条件语句

(1) if语句

条件语句是通过判断指定表达式的值来决定执行还是跳过某些语句。

```
if(expression) {
    statement;
}
```

其中的expression（条件）可以是任意表达式，而且对这个表达式求值的结果不一定是布尔值。ECMAScript会自动调用Boolean()转换函数将这个表达式的结果转换为一个布尔值。如果对expression求值为true时，则执行statement；如果为false，则跳过。

注意：JavaScript语法规定，if关键字和带圆括号的表达式之后必须跟随一条语句，但可以使用语句块将多条语句合并在一起。

if..else..

```
if(expression) {
    statement1;
} else {
    statement2;
}
```

在JavaScript中，if、else匹配规则是：else总是和就近的if语句匹配。

推荐使用代码块，即使只有一行代码。

(2) switch

```
switch(expression) {
    case "": statement ;break;
    case "": statement1;break;
    ...
    default: statements; break;
}
```

如果表达式等于这个值，则执行后面的语句；break关键字会导致代码执行流跳出switch语句。如果省略break，就会导致执行完当前case后，继续执行下一个case；当switch表达式与所有case表达式都不匹配时，则执行default。

当然，如果你有两个值是执行同一段代码的，可以这样：

```
switch(expression) {
    case "":
    case "":
        statement;break;
    ...
}
```

注意：由于对每个case的匹配操作实际是“==”全等运算符比较，而不是“==”相等运算符比较，因此，表

达式和case的匹配并不会做任何类型转换。比如字符串"10"不等于数值10.

(3) 三元运算符 ?:

JavaScript还有一个三元运算符（即该运算符需要三个运算子）?:，也可以用于逻辑判断。

```
(contidion) ? expr1 : expr2
```

上面代码中，如果contidion为true，则返回expr1的值，否则返回expr2的值。

4、循环语句

循环语句就是程序路径的一个回路，可以让一部分代码重复执行。

(1) while语句

`while` 语句属于前测试循环语句，也就是说，在循环体内的代码被执行之前，就会对出口条件求值。因此，循环体内的代码有可能永远不会被执行。

语法：

```
while(expression) {
  statement
}
```

当expression计算为true时，则执行statement。

注意：使用 `while(true)` 则会创建一个死循环。

(2) do...while语句

`do...while`语句是一种后测试循环语句，即只有在循环体中的代码执行之后，才会测试出口条件。也可以说，在对条件表达式求值之前，循环体内的代码至少会被执行一次。

语法：

```
do {
  statement
} while(expression);
```

`do...while`循环和`while`循环非常相似，但是，`do...while`的循环体至少会执行一次。

(3) for语句

`for` 语句也是一种前测试循环语句，但它具有在执行循环之前初始化变量和定义循环后要执行的代码的能力

语法：

```
for(initialize ; test ; increment) {
  statement
}
```

initialize、test、increment三个表达式之间用分号隔开，分别负责初始化操作、循环条件判断和计数器变量的更新。

只有test返回true才会进入for循环，因此也有可能不会执行循环体内的代码。

initialize表达式只在循环开始之前执行一次。

注意：即使是循环内部定义的变量，在循环外部也可以访问到它。

for语句中的初始化表达式、控制表达式和循环后表达式都是可选的，如果都省略，就会创建一个无限循环：

```
for(;;){  
    //无限循环  
}
```

注意：由于JavaScript没有块级作用域，所以在for里面定义的变量都是全局变量。（ES6会有块级作用域）

(4) for...in语句

for...in语句是一种精准的迭代语句，可以用来枚举对象的属性。

语法：

```
for(property in object) {  
    statement  
}
```

例子：

```
var o = { name: 'tg', age: 18};  
for(var v in o) {  
    console.log(v + ': ' + o[v]);  
}
```

注意：ECMAScript对象的属性是没有顺序的，因此通过for...in循环输出的属性名的顺序是不可预测的。

5. 跳转语句

break语句是强制退出循环，然后执行循环后面的语句。

continue语句是终止本次循环的执行并开始下一次循环的执行。

JavaScript中的语句可以命名或带有标签（label），break和continue可以跳转到任意位置，也是在JavaScript中唯一可以使用标签语句的语句。

6. 标签语句

语句是可以添加标签的，标签是由语句前的标识符和冒号组成：

```
label : statement
```

label语句定义的标签一般由break或continue语句引用。加标签的语句一般要与for等循环语句配合使用。

```
var num = 0;
tip : for(var i = 0; i < 10; i++){
    num += i;
    console.log(i); // 轮流输出：0、1、2、3、4、5
    if(i == 5) {
        break tip;
    }
}
console.log(num); // 15
```

当执行到i=5时，会跳出循环，也就是tip对应的层，然后执行其下方的代码。

7. 其他语句

(1) return语句

`return` 语句只能在函数体内出现，否则报错。当执行到 `return` 语句时，函数终止执行，`return` 后面的代码永远不会被执行。

(2) throw语句

异常是指当发生了某种异常情况或错误时产生的一个信号。

```
throw expression
try..catch...finally语句
try{
}catch(e){
}finally{
}
```

(3) with语句

`with` 语句用于临时扩展作用域链，也就是将代码的作用域设置到一个特定的对象中。

```
with(object){
    statement
}
```

将object添加到作用域链的头部，然后执行statement，最后把作用域链恢复到原生状态。

```
var o = {
    name: 'tg',
    age: 24
};
with(o){
    console.log('name : ' + name); // name : tg
```

```
console.log('age : ' + age); // age : 24
```

```
}
```

with里面的name相当于o.name。

注意：在严格模式中是禁止使用with语句的。

(4) debugger语句

debugger语句用来产生一个断点（breakpoint），JavaScript代码的执行会停止在断点的位置。一般用来调试代码。

(5) "use strict"

使用"use strict"指令的目的是说明后续的代码将会解析成严格代码。

对象

对象

JavaScript是面向对象编程（Object Oriented Programming，OOP）语言。

面对对象编程的核心思想就是将真实世界中各种复杂的关系，抽象成一个个对象，然后由对象之间分工合作，完成对真实世界的模拟。

何为对象？

对象是单个实物的抽象。

一本书、一辆汽车、一个人都可以是“对象”，一个数据库、一张网页也可以是“对象”。世界上所有的对象都可以是“对象”。

对象是一个容器，封装了“属性”（property）和“方法”（method）。

属性，就是对象的状态，而方法，就是对象的行为。比如：我们可以把一辆汽车抽象成一个对象，它的属性就是它的颜色、重量等，而方法就是它可以启动、停止等。

1、对象

在Javascript中，对象是一个基本数据类型。

对象是一种复合值：它将很多值聚合在一起，可通过名字访问这些值。对象也可看做一种无序的数据集合，由若干个“键值对”（key-value）构成。

```
var o = {
  name: 'a'
}
```

上面代码中，大括号定义了一个对象，它被赋值给变量o。这个对象内部包含一个键值对（又称为“成员”），name是“键名”（成员的名称），字符串a是“键值”（成员的值）。

键名与键值之间用冒号分隔。如果对象内部包含多个键值对，每个键值对之间用逗号分隔。

键名：对象的所有键名都是字符串，所以加不加引号都可以。如果键名是数值，会被自动转为字符串。

对象的每一个“键名”又称为“属性”（property），它的“键值”可以是任何数据类型。如果一个属性的值为函数，通常把这个属性称为“方法”，它可以像函数那样调用。

```
var o = {
  go: function(x){
    return x+1;
  }
};
o.go(2) // 3
```

如果键名不符合标识名的条件（比如第一个字符为数字，或者含有空格或运算符），也不是数字，则必须加上引

号，否则会报错。

```
var o = {
  '1a' : 'a'
}
```

上面的代码中，如果键名'1a'不用引号引起来，就会报错。

注意：为了避免这种歧义，JavaScript规定，如果行首是大括号，一律解释为语句（即代码块）。如果要解释为表达式（即对象），必须在大括号前加上圆括号。

2、创建对象

在JavaScript中，有三种方法创建对象

- 对象直接量： var o={};
- 关键字new： var o=new Object();
- Object.create()函数： var o=Object.create(null)

2.1对象直接量

对象直接量是由若干名/值对组成的映射表。键名与键值之间用冒号分隔。如果对象内部包含多个键值对，每个键值对之间用逗号分隔。整个映射表用花括号括起来。

在ECMAScript 5中，保留字可以用做不带引号的属性名。

注意：对象直接量中的最后一个属性后的逗号可有可无，但是在ie中，如果多了一个逗号，会报错。 2.2通过new 创建对象

new运算符创建并初始化一个新对象。关键字new后跟随一个函数调用，这个函数称做构造函数
(constructor)。

例子：

```
var o1 = {};
var o2 = new Object();
var o3 = Object.create(null);
```

上面三行语句是等价的。

对象最常见的用法是创建 (create)、设置 (set)、查找 (query)、删除 (delete)、检测 (test) 和枚举 (enumerate) 它的属性。属性包括名字 (键名) 和值 (键值)。

属性名可以是包含空字符串在内的任意字符串，但对象中不能存在两个同名的属性。

提取方法

如果对对象中的方法进行提取，则会失去与对象的连接。

```
var obj = {
```

```

name: 'a',
get: function() {
  console.log(this.name);
}
};

console.log(obj.get()); // "a"

var func = obj.get;
console.log(func()); // undefined

```

在上面的例子中，object对象中有一个方法get()，用来获取obj对象中的name，而当get()方法赋值给一个变量func，再调用func()函数时，此时的this是指向window的，而非obj的。

注意：如果在严格模式下，this会是undefined。

3、属性特性

- 可写 (writable attribute)：可设置该属性的值。
- 可枚举 (enumerable attribute)：可通过for/in循环返回该属性。
- 可配置 (configurable attribute)：可删除或修改属性。

4、读取属性

读取对象的属性，有两种方法，一种是使用点运算符，还有一种是使用方括号运算符。

```

var o = {
  name : 'a'
}
o.name // "a"
o['name'] // "a"

```

注意：数值键名不能使用点运算符（因为会被当成小数点），只能使用方括号运算符。

JavaScript对象是动态的，可新增属性也可删除属性。但注意，我们是通过引用而非值来操作对象。

5、属性的查询和设置

在JavaScript中，我们可以通过点(.)或方括号([])运算符来获取属性的值。运算符左侧应当是一个表达式，它返回一个对象。

(1) for...in

for...in循环用来遍历一个对象的全部属性。

```

var o = {
  name : 'a',
  age : 12
}
for(var i in o){

```

```
console.log(o[i])
}
// "a"
// 12
```

(2) 查看所有属性

查看一个对象本身的所有属性，可以使用Object.keys方法，返回一个数组。

```
var o = {
  name : 'a',
  age : 12
}

Object.keys(o) //['name', 'age']
```

(3) 删除属性

delete运算符可以删除对象的属性。

```
var o = {
  name : 'a'
}
delete o.name //true
o.name //undefined
```

注意：delete运算符只能删除自有属性，不能删除继承属性。

删除一个不存在的属性，delete不报错，而且返回true。

只有一种情况，delete命令会返回false，那就是该属性存在，且不得删除。

(4) 检测属性

在JavaScript中，有多种方法检测某个属性是否存在于某个对象中。

用 “!=!” 来判断一个属性是否是undefined

(5) hasOwnProperty()方法

用于判断一个对象自身(不包括原型链)是否具有指定名称的属性。如果有，返回true，否则返回false。

(6) propertyIsEnumerable()方法

只有检测到是自有属性且这个属性的可枚举性为true时才返回true。

(7) in运算符

in运算符左侧是属性名（字符串），右侧是对象。如果对象的自有属性或继承属性中包含这个属性就返回true。

```
var o = {
  name : 'a'
}
'name' in o //true
```

6、对象的三个属性

每一个对象都有与之相关的原型 (prototype) 、类 (class) 和可扩展性 (extensible attribute)

将对象作为参数传入Object.getPrototypeOf()可以查询它的原型。

检测一个对象是否是另一个对象的原型，可以使用isPrototypeOf()方法。

7、序列化对象

对象序列化是指将对象的状态转换为字符串，也可将字符串还原为对象。

在JavaScript中，提供了内置函数**JSON.stringify()和JSON.parse()**用来序列化和还原JavaScript对象。

NaN、Infinity和-Infinity序列化的结果是null

```
var o = {
  name : 'a',
  age : 12,
  intro : [false, null, '']
}
s= JSON.stringify(o) // s {"name":"a", "age":12, "intro":[false, null, ""]}
p=JSON.parse(s) // p是o的深拷贝
```

注意：JSON.stringify()只能序列化对象可枚举的自有属性。对于一个不能序列化的属性来说，在序列化后的输出字符串中会将这个属性省略掉。

8、构造函数

构造函数，是用来生成“对象”的函数。一个构造函数可生成多个对象，这些对象都有相同的结构。

构造函数的特点：

- 函数体内使用了this关键字，代表了所要生成的对象实例
- 生成对象时，必需用new命令
- 构造函数名字的第一个字母通常大写。

例子：

```
function Car(){
  this.color = 'black';
}
var c = new Car();
```

上面的代码生成了Car的实例对象，保存在变量c中。

构造函数也可以传入参数：

```
function Car(color){
  this.color = color;
```

```
}
```

```
var c = new Car('red');
```

new命令本身就可以执行构造函数，所以后面的构造函数可以带括号，也可以不带括号。下面两行代码是等价的。

```
var c = new Car();
```

```
var c = new Car;
```

每一个构造函数都有一个prototype属性。

8.1 this关键字

this总是返回一个对象，简单说，就是返回属性或方法“当前”所在的对象。

this.property

上面的代码中，this就代表property属性当前所在的对象。

由于对象的属性可以赋给另一个对象，所以属性所在的当前对象是可变的，即this的指向是可变的。

```
var A = {
  name: '张三',
  describe: function(){
    return this.name;
  }
};
var B = {
  name: '李四'
};
B.describe = A.describe;
B.describe();
// "李四"
```

注意：如果一个函数在全局环境中运行，那么this就是指顶层对象（浏览器中为window对象）。

8.1.1 改变this指向

在JavaScript中，提供了call、apply、bind三种方法改变this的指向。

(1) funciton.prototype.call()

```
call(obj, arg1, arg2, ...)
```

第一个参数obj是this要指向的对象，也就是想指定的上下文；arg1,arg2...都是要传入的参数。

注意：如果参数为空、null和undefined，则默认传入全局对象。

(2) funciton.prototype.apply()

apply(obj, [arg1, arg2...])

apply()和call()方法原理类似，只不过，它第二个参数一个数组，里面的值就是要传入的参数。

(3) `function.prototype.bind()`

bind方法用于将函数体内的this绑定到某个对象，然后返回一个新函数。

bind(obj)

更多：[JS中的call、apply、bind方法](#)

9. 原型

9.1 原型

每一个JavaScript对象（null除外）都和另一个对象相关联，也可以说，继承另一个对象。另一个对象就是我们熟知的“原型”（prototype），每一个对象都从原型继承属性。只有null除外，它没有自己的原型对象。

所有通过对象直接量创建的对象都具有同一个原型对象，并可以通过JavaScript代码Object.prototype获得对原型对象的引用。

通过关键字new和构造函数调用创建的对象的原型就是构造函数的prototype属性的值。比如：通过new Object()创建的对象继承自Object.prototype；通过new Array()创建的对象的原型是Array.prototype。没有原型的对象为数不多，Object.prototype就是其中之一，它不继承任何属性。

所有的内置构造函数都具有一个继承自Object.prototype的原型。

9.2 原型链

对象的属性和方法，有可能是定义在自身，也有可能是定义在它的原型对象。由于原型本身也是对象，又有自己的原型，所以形成了一条原型链（prototype chain）。比如，a对象是b对象的原型，b对象是c对象的原型，以此类推。

如果一层层地上溯，所有对象的原型最终都可以上溯到Object.prototype，即Object构造函数的prototype属性指向的那个对象。那么，Object.prototype对象有没有它的原型呢？回答可以是有的，就是没有任何属性和方法的null对象，而null对象没有自己的原型。

“原型链”的作用：

当读取对象的某个属性时，JavaScript引擎先寻找对象本身的属性，如果找不到，就到它的原型去找，如果还是找不到，就到原型的原型去找。如果直到最顶层的Object.prototype还是找不到，则返回undefined。

继承

JavaScript对象具有“自有属性”，也有一些属性是从原型对象继承而来的。

当查询一个不存在的属性时，JavaScript不会报错，返回undefined。

如果对象自身和它的原型，都定义了一个同名属性，那么优先读取对象自身的属性，这叫做“覆盖”（overriding）。

9.2.1 constructor属性

prototype对象有一个constructor属性，默认指向prototype对象所在的构造函数。

9.3 操作符

(1) instanceof运算符

instanceof运算符返回一个布尔值，表示指定对象是否为某个构造函数的实例。

```
var c = new Car();
c instanceof Car //true
```

instanceof运算符的左边是实例对象，右边是构造函数。它的运算实质是检查右边构建函数的原型对象，是否在左边对象的原型链上。

(2) Object.getPrototypeOf()

Object.getPrototypeOf方法返回一个对象的原型。这是获取原型对象的标准方法

```
Object.getPrototypeOf(c) === Car.prototype //true
```

(3) Object.setPrototypeOf()

Object.setPrototypeOf方法可以为现有对象设置原型，返回一个新对象。Object.setPrototypeOf方法接受两个参数，第一个是现有对象，第二个是原型对象。

(4) Object.create()

Object.create方法用于从原型对象生成新的实例对象，可以替代new命令。

它接受一个对象作为参数，返回一个新对象，后者完全继承前者的属性，即原有对象成为新对象的原型。

(5) Object.prototype.isPrototypeOf()

对象实例的isPrototypeOf方法，用来判断一个对象是否是另一个对象的原型。

```
Object.prototype.isPrototypeOf({}) //true
```

(6) Object.prototype.proto

proto 属性（前后各两个下划线）可以改写某个对象的原型对象。

(7) Object.getOwnPropertyNames()

Object.getOwnPropertyNames方法返回一个数组，成员是对象本身的所有属性的键名，不包含继承的属性键名。

(8) Object.prototype.hasOwnProperty()

对象实例的hasOwnProperty方法返回一个布尔值，用于判断某个属性定义在对象自身，还是定义在原型链上。

数组

数组

1. 数组简介

数组是值的有序集合。每个值叫做一个元素，而每个元素在数组中有一个位置，以数字表示（从0开始），称为索引，整个数组用方括号表示。

```
var arr = [1, 2, 3];
```

除了在定义时赋值，数组也可以先定义后赋值。

```
var arr = [];
arr[0] = 1;
```

数组元素可以是任意类型。

```
var arr = [1, 'a', {name: 'a'}, function(){}];
```

上面数组arr的4个元素分别是数字，字符串，对象，函数。

数组属于一种特殊的对象。

```
typeof [1]
// "object"
```

JavaScript语言规定，对象的键名一律为字符串，所以，数组的键名其实也是字符串。之所以可以用数值读取，是因为非字符串的键名会被转为字符串。

2、 创建数组

2.1 创建数组

(1) 数组直接量

使用数组直接量是创建数组最简单的方法，在方括号中将数组元素用逗号隔开即可。

```
var arr = [1, 2, 3];
```

注：如果省略数组直接量的某个值，省略的元素将被赋予undefined值。

(2) new Array()

我们也可以调用构造函数Array()创建数组

```
var a=new Array();
var a2=new Array(10);
var a3=new Array(1,2,3,4);
```

上面的例子，a是一个空数组，a2是一个包含10个数组项但为undefined的数组，a3是包含4个数组项，并且分别有值。

2.2 数组长度

每个数组都有一个length属性，返回数组的元素数量

```
[1,2,3].length //3
```

JavaScript使用一个32位整数，保存数组的元素个数。这意味着，数组成员最多只有4294967295个($2^{32} - 1$)个，也就是说length属性的最大值就是4294967295。

只要是数组，就一定有length属性。该属性是一个动态的值，等于键名中的最大整数加上1.

length属性是可写的。如果人为设置一个小于当前成员个数的值，该数组的成员会自动减少到length设置的值。

```
var arr = [1,2,3]
arr.length //3

arr.length = 2;
arr // [1,2]
```

将数组清空的一个有效方法，就是将length属性设为0。

```
var arr = [1,2,3];
arr.length = 0;
arr // []
```

如果人为设置length大于当前元素个数，则数组的成员数量会增加到这个值，新增的位置都是空位。

```
var arr = [1];
arr.length=3;
arr[1] // undefined
```

在ECMAScript 5中，可以用Object.defineProperty()让数组的length属性变成只读。

2.3 空位

当数组的某个位置是空元素，即两个逗号之间没有任何值，我们称该数组存在空位(hole)。

```
var arr = [1,,2]
```

数组

```
arr.length //3
```

但是，如果最后一个元素后面有逗号，并不会产生空位。数组直接量的语法允许有可选的结尾的逗号，故[,,]只有两个元素而非三个。

```
var arr = [,,];
arr.length //2
```

2.4 数组元素的读和写

使用()操作符来访问数组中的一个元素。

```
var arr = [1,2]
arr[0] //1
arr[1]=3; // [1,3]
```

2.5 数组元素的添加和删除

可以使用push()方法在数组末尾添加一个或多个元素。

```
var arr = [1,2]
arr.push(3) // [1,2,3]
arr.push('a','b') // [1,2,3,'a','b']
// shift()是删除数组的一个元素。
arr.shift() // [2,3,'a','b']
```

2.6 稀疏数组

稀疏数组是指包含从0开始的不连续索引的数组。通常，数组的length属性值代表数组中元素的个数，但如果是稀疏数组，length属性值大于元素的个数。

2.7 多维数组

JavaScript不支持真正的多维数组，但可以用数组的数组来近似。也可以说，数组里放数组。

```
var arr = [[1],[2,3]];
arr[0][0] // 1
arr[1][1] //3
```

2.8 遍历数组

我们可以使用for循环、while循环、for..in或者forEach()方法来遍历数组

```
var a = [1, 2, 3];
// for循环
```

数组

```
for(var i = 0; i < a.length; i++) {
    console.log(a[i]);
}

//while
var i = 0;
while (i < a.length) {
    console.log(a[i]);
    i++;
}

//for..in
for (var i in a) {
    console.log(a[i]);
}

//forEach
a.forEach(function(v){
    console.log(v);
})
```

2.9 类数组对象

在JavaScript中，有些对象被称为“类数组对象”。意思是，它们看上去很像数组，可以使用length属性，但是它们并不是数组，无法使用一些数组的方法。

```
var o = {
    0: 'a',
    1: 'b',
    length:2
}
o[0] // "a"
o[1] // "b"
o.length // 2
o.push('d') // TypeError: o.push is not a function
```

上面代码中，变量o是一个对象，虽然使用的时候看上去跟数组很像，但是无法使用数组的方法。这就是类数组对象。

类数组对象有一个特征，就是具有length属性。换句话说，只要有length属性，就可以认为这个对象类似于数组。但是，对象的length属性不是动态值，不会随着成员的变化而变化。

由于类数组对象没有继承自Array.prototype，那就不能在它们上面直接调用数组方法。不过我们可以间接的使用Function.call方法调用。

```
var o = {
    0: 'a',
```

```
1: 'b',
length:2
};

Array.prototype.slice.call(o) // ["a", "b"]
```

典型的类似数组的对象是函数的arguments对象，以及大多数DOM元素集，还有字符串。

函数

函数

1. 函数简介

通过函数可以封装任意多条语句，而且可以在任何地方、任何时候调用。

ECMAScript中的函数使用 `function` 关键字来声明，后跟一组参数以及函数体，这些参数在函数体内像局部变量一样工作。

```
function functionName(arg0, arg1....argN) {
    statements
}
```

函数调用会为形参提供实参的值。函数使用它们实参的值来计算返回值，称为该函数调用表达式的值。

```
function test(name){
    return name;
}
test('tg');
```

在上面的例子中，`name`就是形参，调用时的'tg'就是实参。

除了实参之外，每次调用还会拥有另一个值---本次调用的上下文---这就是 `this` 关键字的值。

我们还可以通过在函数内添加 `return` 语句来实现返回值。

注意：遇到 `return` 语句时，会立即退出函数，也就是说，`return` 语句后面的语句不再执行。

```
function test(){
    return 1;
    alert(1); //永远不会被执行
}
```

一个函数中可以包含多个 `return` 语句，而且 `return` 语句可以不带有任何返回值，最终将返回 `undefined`。

```
function test(num){
    if(num > 2){
        return num;
    }else{
        return ;
    }
}
test(3); // 3
```

```
test(1); //undefined
```

如果函数挂载在一个对象上，将作为对象的一个属性，就称它为对象的方法。

```
var o = {
  test: function(){}
}
```

test()就是对象o的方法。

2、函数定义（声明）

JavaScript有三种方法，可以定义一个函数。

(1) function命令

```
function name() {}
```

name是函数名称标识符。函数名称是函数声明语句必需的部分。不过对于函数表达式来说，名称是可选的：如果存在，该名字只存在于函数体内，并指向该函数对象本身。

圆括号：圆括号内可放置0个或多个用逗号隔开的标识符组成的列表，这些标识符就是函数的参数名称。花括号：可包含0条或多条JavaScript语句。这些语句构成了函数体。一旦调用函数，就会执行这些语句。

(2) 函数表达式

```
var f = function(x){
  console.log(x);
}
```

采用函数表达式声明函数时，function命令后面不带有函数名。如果加上函数名，该函数名只在函数体内部有效，在函数体外部无效。

(3) Function()

Function()函数定义还可以通过Function()构造函数来定义

```
var f=new Function('x','y','return x+y');
```

等价于

```
var f=function(x,y){
  return x+y;
}
```

函数

除了最后一个参数是函数体外，前面的其他参数都是函数的形参。如果函数不包含任何参数，只须给构造函数简单的传入一个字符串---函数体---即可。不过，Function()构造函数在实际编程中很少会用到。

注意点：如果同一个函数被多次定义（声明），后面的定义（声明）就会覆盖前面的定义（声明）

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

function f(){
  console.log(2);
}
f() //2
```

函数可以调用自身，这就是递归（recursion）

```
function f(x){
  if(x>2){
    console.log(x);
    return f(x-1);
  }else{
    return 1;
  }
}
f(4);
// 4
//3
```

不能在条件语句中声明函数（在ES6中的块级作用域是允许声明函数的。）

3、函数命名

任何合法的JavaScript标识符都可以用做一个函数的名称。

函数名称通常是动词或以动词为前缀的词组。通常函数名的第一个字符为小写。当函数名包含多个单词时，可采取下划线法，比如：like_this()；也可以采取驼峰法，也就是除了第一个单词之外的单词首字母使用大写字母，比如：likeThis();

4、被提前

就像变量的“被提前”一样，函数声明语句也会“被提前”到外部脚本或外部函数作用域的顶部，所以以这种方式声明的函数，可以被在它定义之前出现的代码所调用。

```
f()

function f(){}

```

其实JavaScript是这样解释的：

```
function f(){}
f()
```

注意：在函数提升中，函数体也会跟着提升（不像变量一样，只会提升变量声明），这也是我们可以引用后面声明的函数的原因。

此外，以表达式定义的函数并没有“被提前”，而是以变量的形式“被提前”。

```
f();
var f = function (){};
// TypeError: f is not a function
```

变量其实是分为声明，赋值两部分的，上面的代码等同于下面的形式

```
var f;
f();
f = function() {};
```

调用f的时候，f只是被声明了，还没有被赋值，等于undefined，所以会报错。

5、嵌套函数

JavaScript的函数可以嵌套在其他函数中定义，这样它们就可以访问它们被定义时所处的作用域中的任何变量，这就是JavaScript的闭包。

```
function test(){
  var name = 'tg';
  function test2(){
    var age = 10;
    console.log(name); // "tg"
  }
  console.log(age); // Uncaught ReferenceError: age is not defined
}
test();
```

从上面的例子可得，test2()可以访问name，但是如果在test()内，test2()外访问age，就会报错。

6、函数调用

构成函数主体的JavaScript代码在定义时并不会执行，只有调用该函数，它们才会执行。有4种方式调用JavaScript函数：

- 作为函数

- 作为方法
- 作为构造函数
- 通过它们的call()和apply()方法间接调用

6.1 函数调用

函数可以通过函数名来调用，后跟一对圆括号和参数（圆括号中的参数如果有多个，用逗号隔开）

```
function test(){}
test()
```

6.2 方法调用

```
var o = {
  f: function(){}
}
o.f();
```

6.3 构造函数调用

如果函数或者方法调用之前带有关键字 new，它就构成构造函数调用。凡是沒有形参的构造函数调用都可以省略圆括号（但不推荐）。

```
var o=new Object();
var o=new Object;
```

7、函数的实参和形参

7.1 可选形参

在ECMAScript中的函数在调用时，传递的参数可少于函数中的参数，没有传入参数的命名参数的值是undefined。

为了保持好的适应性，一般应当给参数赋予一个合理的默认值。

```
function go(x,y){
  x = x || 1;
  y = y || 2;
}
```

注意：当用这种可选实参来实现函数时，需要将可选实参放在实参列表的最后。那些调用你的函数的程序员是没法省略第一个参数并传入第二个实参的。

7.2 实参对象

当调用函数时，传入的实参数个数超过函数定义时的形参数个数时，是没有办法直接获得未命名值的引用。这时，标

识符 `arguments` 出现了，其指向实参对象的引用，实参对象是一个类数组对象，可以通过数字下标来访问传入函数的实参值，而不用非要通过名字来得到实参。

```
function go(x){
  console.log(arguments[0]);
  console.log(arguments[1]);
}
go(1, 2);
//1
//2
```

`arguments` 有一个 `length` 属性，用以标识其所包含元素的个数。

```
function f(x){
  console.log(arguments.length);
}
f(1, 2) // 2
```

注意：`arguments` 并不是真正的数组，它是一个类数组对象。每个实参对象都包含以数字为索引的一组元素以及 `length` 属性。

通过实参名字来修改实参值的话，通过 `arguments[]` 数组也可以获取到更改后的值。

在函数体内，我们可以通过 `arguments` 对象来访问这个参数类数组，我们可以使用方括号语法访问它的每一个参数（比如 `arguments[0]`），它还有一个 `length` 属性，表示传递进来的参数个数。

`arguments` 类数组中每一个元素的值会与对应的命名参数的值保持同步，这种影响是单向的，也可以这样说，如果是修改 `arguments` 中的值，会影响到命名参数的值，但是修改命名参数的值，并不会改变 `arguments` 中对应的值。

```
function f(x){
  console.log(x); // 1
  arguments[0]=null;
  console.log(x); // null
}
f(1);
```

在上面的例子中，`arguments[0]` 和 `x` 指代同一个值，修改其中一个的值会影响到另一个。注意：如果有同名的参数，则取最后出现的那个值。

```
function f(x, x){
  console.log(x);
}
```

```
f(1,2) // 2
```

callee 和 caller 属性

`arguments`对象带有一个 `callee` 属性，返回它所对应的原函数。

在一个函数调用另一个函数时，被调用函数会自动生成一个 `caller` 属性，指向调用它的函数对象。如果该函数当前未被调用，或并非被其他函数调用，则 `caller` 为`null`。

再次提醒，`arguments` 并不是真正的数组，它只是类数组对象（有`length`属性且可使用索引来访问子项）。

但我们可以借助`Array`类的原型对象的`slice`方法，将其转为真正的数组：

```
Array.prototype.slice.call(arguments, 0);
//更简洁的写法
[].slice.call(arguments, 0);
```

7.3 按值传参

ECMAScript中所有函数的参数都是 按值传递 的。也就是说，把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样。

在向参数传递基本类型的值时，被传递的值会被复制给一个局部变量（即命名参数，或者用ECMAScript的概念来说，就是`arguments`对象中的一个元素。）

例子：

```
var num = 1;
function test(count){
  count += 10;
  return count;
}
var result = test(num);
console.log(num); // 1
console.log(result); // 11
```

在上面的例子中，我们将`num`作为参数传给了`test()`函数，即`count`的值也是1，然后在函数内将`count`加10，但是由于传递的只是`num`的值的一个副本，并不会影响`num`，`count`和`num`是独立的，所以最后`num`的值依旧是1。在向参数传递引用类型的值时，会把这个值在内存中的地址复制给一个局部变量，若局部变量变化，则局部变量和复制给局部变量路径的全局变量也会发生改变。

```
function test(obj){
  obj.name = 'tg';
}
var person = new Object();
test(person);
console.log(person.name); // "tg"
```

但是，如果局部变量指向了一个新的堆内地址，再改变局部变量的属性时，不会影响全局变量。

看下面的例子：

```
function test(obj){
    obj.name = 'tg';
    obj = new Object();
    obj.name = 'tg2';
}

var person = new Object();
test(person);
console.log(person.name); // "tg"
```

在上面的例子中，全局的 `person` 和函数内局部的 `obj` 在初始传递时，两者指向的是内存中的同一个地址，但是，当在函数内创建了一个新的对象，并赋值给 `obj`（赋值的是新对象的地址）。这个时候，`obj` 指向的就不再是全局对象 `person`，而是指向了新对象的地址，所以给 `obj` 添加属性 `name` 时，全局对象 `person` 的属性不会被改变。

对于上面的例子中的 `obj`，也可以这样说，一旦 `obj` 的值发生了变化，那么它就不再指向 `person` 在内存中的地址了。

8、将对象属性用做实参

当一个函数包含超过三个形参时，要记住调用函数中实参的正确顺序是件让人头疼的事。不过，我们可以通过名/值对的形式传入参数，这样就不需要管参数的顺序了。

```
function f(params){
    console.log(params.name);
}
f({name: 'a'})
```

9、作为值的函数

在JavaScript中，我们可以将函数赋值给变量。

```
function f(){}
var a=f;
```

10、函数作用域

作用域（scope）指的是变量存在的范围。

Javascript只有两种作用域：一种是全局作用域，变量在整个程序中一直存在，所有地方都可以读取；另一种是函数作用域，变量只在函数内部存在。在函数外部声明的变量就是全局变量（global variable），它可以在函数内部读取。

```
var a=1;
function f(){
  console.log(a)
}
f() //1
```

上面的代码中，函数f内部可以读取全局变量a。

在函数内部定义的变量，外部无法读取，称为“局部变量”（local variable）。

```
function f(){
  var a=1;
}
v //ReferenceError: v is not defined
```

上面代码中，变量v在函数内部定义，所以是一个局部变量，函数之外就无法读取。

函数内部定义的变量，会在该作用域内覆盖同名全局变量。

```
var a=1;
function f(){
  var a=2;
  console.log(a);
}
f() //2
a //1
```

注意：对于var命令来说，局部变量只能在函数内部声明，在其他区块中声明，一律都是全局变量。

函数的执行依赖于变量作用域，这个作用域是在函数定义时决定的，而不是函数调用时决定的。

****11、函数内部的变量提升****

与全局作用域一样，函数作用域内部也会产生“变量提升”现象。

var 命令声明的变量，不管在什么位置，变量声明都会被提升到函数体的头部。

```
function f(x){
  if(x>10){
    var a = x -1;
  }
}
//等同于
function f(x){
  var a;
  if(x>10){
    a = x - 1;
  }
}
```

12、没有重载

ECMAScript函数没有重载的定义。

重载是指为一个函数编写两个定义，只要这两个定义的签名（接受的参数的类型和数量）不同即可。

对于ECMAScript函数，如果定义了两个同名的，后定义的函数会覆盖先定义的函数。

13、函数属性、方法和构造函数

13.1 函数的属性、方法

(1) name属性

name属性返回紧跟在function关键字之后的那个函数名。

```
function f(){}
f.name //f
```

(2) length属性

函数的length属性是只读属性，代表函数形参的数量，也就是在函数定义时给出的形参个数。

```
function f(x,y){}
f.length //2
```

(3) prototype属性

每一个函数都包含一个prototype属性，这个属性指向一个对象的引用，这个对象称做“原型对象”（prototype object）。

(4) call()

语法：

```
call([thisObj[,arg1[, arg2[, [, .argN]]]]])
```

定义：调用一个对象的一个方法，以另一个对象替换当前对象。

说明：call方法可以用来代替另一个对象调用一个方法。call方法可将一个函数的对象上下文从初始的上下文改变为由thisObj指定的新对象。

(5) apply()

语法：

```
apply([thisObj[,argArray]])
```

定义：应用某一对象的一个方法，用另一个对象替换当前对象。

说明：如果argArray不是一个有效的数组或者不是arguments对象，那么将导致一个TypeError。如果没有

提供 argArray 和 thisObj 任何一个参数，那么 Global 对象将被用作 thisObj，并且无法被传递任何参数。
bind()方法 bind()方法是在 ECMAScript 5 中新增的方法。toString()方法
函数的toString方法返回函数的源码。

```
function f(){
  return 1;
}
f.toString()
//function f(){
//  return 1;
//}
```

(6) bind()

bind()方法会创建一个新函数，称为绑定函数，当调用这个绑定函数时，绑定函数会以创建它时传入 bind()方法的第一个参数作为 this，传入 bind() 方法的第二个以及以后的参数加上绑定函数运行时本身的参数按照顺序作为原函数的参数来调用原函数。

```
var bar=function(){
  console.log(this.x);
}
var foo={
  x:3
}
bar();
bar.bind(foo)();
/*或*/
var func=bar.bind(foo);
func();
```

输出：
undefined
3

注意：bind()返回的是函数。

13.2 构造函数

构造函数和普通函数的定义并没有太大区别，不过我们使用 new 关键字来生成构造函数的实例对象。

```
function Test(){}
var t = new Test();
```

对于构造函数，一般首字母大写，便于和普通函数区别开来。

定义每个函数都会主动获取一个 prototype 属性，该属性拥有一个对象--该函数的原型，该原型有一个

`constructor` 属性，指向其当前所属的函数。

14、闭包

JavaScript的函数可以嵌套在其他函数中定义，这样它们就可以访问它们被定义时所处的作用域中的任何变量，这就是JavaScript的 闭包 。

闭包 会保存函数作用域中的状态，即使这个函数已经执行完毕。

闭包 的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中，即

闭包 可以使得它诞生环境一直存在。

闭包 的创建依赖于函数。

```
function f(a){
  return function(){
    return a++;
  };
}
var c=f(1);
console.log(c()); //1
console.log(c()); //2
console.log(c()); //3
```

闭包的另一个用处，是封装对象的私有属性和私有方法。

15、立即调用的函数表达式 (IIFE)

在Javascript中，一对圆括号()是一种运算符，跟在函数名之后，表示调用该函数。

```
(function(){
  statement
}())
```

上面的函数会立即调用。

注意：上面代码的圆括号的用法，`function`之前的左圆括号是必需的，因为如果不写这个左圆括号，JavaScript解释器会试图将关键字`function`解析为函数声明语句。而使用圆括号，JavaScript解释器才会正确地将其解析为函数定义表达式。

当然，下面的方法也会以表达式来处理函数定义的方法。

```
!function(){}();
~function(){}();
- function(){}();
+function(){}();
```

通常情况下，只对匿名函数使用这种“立即执行的函数表达式”。它的目的有两个：

- 一是不必为函数命名，避免了污染全局变量；
- 二是IIFE内部形成了一个单独的作用域，可以封装一些外部无法读取的私有变量。

16、eval命令

eval命令的作用是，将字符串当作语句执行。

```
eval('var a=1');
a //1
```

eval没有自己的作用域，都在当前作用域内执行

JavaScript规定，如果使用严格模式，eval内部声明的变量，不会影响到外部作用域。

```
(function(){
  'use strict';
  eval('var a=1');

  console.log(a); //ReferenceError: a is not defined
})();
```

17、严格模式下的函数

- 不能把函数命名为eval或arguments
- 不能把参数命名为eval或arguments
- 不能出现两个命名参数同名的情况

如果出现上面三种情况，都会导致语法错误，代码无法执行。

引用类型（对象）

引用类型

引用类型的值（对象）是引用类型的一个实例。在ECMAScript中，引用类型是一种数据结构，用于将数据和功能组织在一起。在其他语言中称为类，不过ECMAScript中没有类的说法（ES6中会有class类，这里暂不管）。

Object对象

Object类型

1. 实例化对象

所有其他对象都继承Object。

创建object实例的方式有两种：

- 第一个中是使用new操作符后跟Object构造函数

```
var person = new Object()
```

注意：O是大写

- 第二种是使用对象字面量：

```
var person = {
  name: 'tg'
};
```

访问对象属性使用的是点表示法，也可以用方括号表示法来访问。

```
var person = {
  name: 'tg'
};
console.log(person.name); // "tg"
console.log(person['name']); // "tg"
```

注意：如果对象属性名是不符合语法规的或属性名是关键字或保留字，只能使用方括号表示法。例如：

```
var person = {
  "1a" : 1
};
person["1a"]; // 1
person.1a // 会报错
```

2、给对象添加方法

我们可以像添加属性一样给对象添加方法：

```
var o = {};
o.test = function(){}
```

Object对象

```
console.log(1);
}

o.test(); //1
```

3、Object对象的静态方法

3.1 Object.keys()、Object.getOwnPropertyNames()

`Object.keys()` 方法和 `Object.getOwnPropertyNames()` 方法一般用来遍历对象的属性，它们的参数都是一个对象，返回一个数组，该数组的项都是对象自身的（不是继续原型的）所有属性名。两者的区别在于，`Object.keys()` 只返回可枚举的属性，`Object.getOwnPropertyNames()` 方法还返回不可枚举的属性名。

```
var arr = ['a', 'b'];
console.log(Object.keys(arr)); // ["0", "1"]
console.log(Object.getOwnPropertyNames(arr)); // ["0", "1", "length"]
```

数组的`length`属性是不可枚举的。

4、Object对象的实例方法

`valueOf()`：返回当前对象对应的值，默认情况下返回对象本身。
`toString()`：返回当前对象对应的字符串形式。
`toLocaleString()`：返回当前对象对应的本地字符串形式。
`hasOwnProperty()`：判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。如果是自身的属性，返回`true`，否则返回`false`。
`isPrototypeOf()`：判断当前对象是否为另一个对象的原型。如果是，返回`true`，否则返回`false`。
`propertyIsEnumerable()`：判断某个属性是否可枚举。

(1) valueOf()

`valueOf()` 方法返回当前对象对应的值，默认情况下返回对象本身。

```
var o = {
  name: 'tg',
  age: 24
};
console.log(o == o.valueOf()); // true
```

(2) toString()

`toString()` 返回当前对象对应的字符串形式。

```
var o = {
  name: 'tg',
```

```
age: 24
};

console.log(o.toString()); // "[object Object]"
```

(3) hasOwnProperty()

`hasOwnProperty()` 判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。如果是自身的属性，返回`true`，否则返回`false`。

```
var o = {
  name: 'tg',
  age: 24
};
console.log(o.hasOwnProperty('name')); // true
console.log(o.hasOwnProperty('toString')); // false
```

`toString()`方法是继承原型的，所以返回后`false`。

(4) isPrototypeOf()

`isPrototypeOf()` 方法判断当前对象是否为另一个对象的原型。如果是，返回`true`，否则返回`false`。

```
function Test(){}
var t = new Test();

console.log(Test.prototype.isPrototypeOf(t)); // true
```

Array对象

Array类型

ECMAScript数组的每一项可以保存任何类型的数据，也就是说，而且数组的大小是可以动态调整的，即可以随着数据的添加自动增长以容纳新增数据。

```
var colors = [1, 'tg', {}];
```

在上面的例子中，数组中保存了数值、字符串和对象。

1、初始化数组实例

创建数组的基本方式有多种：

(1) 使用Array构造函数

当传递一个参数时，如果传递的参数是数值，则会按照该数值创建包含给定项数的数组；如果是其他类型的参数，则会创建包含那个值的数组（只有一项）。

```
var colors1 = new Array();
var colors2 = new Array(20);
var colors3 = new Array('a', 1);
```

在上面的代码中，第一行代码创建了一个空数组；第二行代码创建了一个length为20的属性；第三行代码创建了一个包含两个数组项的数组。

可省略new，但不推荐

```
var colors = Array();
```

(2) 使用数组字面量

使用数组字面量表示法，由一对包含数组项的方括号表示，多个数组项之间用逗号隔开。

```
var colors = []; // 空数组
var colors2 = [1, 2]; // 包含两个数组项的数组
```

数组的项数保存在其 `length` 属性中，不小于0，可读写。可通过减小来移除数组项。所有数组项默认是 `undefined`（在不设置的情况下）。

2、检测数组 (`Array.isArray()`)

`Array.isArray()` 判断是否是数组，传入一个数组作为参数。

```
var colors = [];
console.log(Array.isArray(colors)); // true
console.log(Array.isArray(1)); // false
```

3、Array实例的方法

(1) valueof()

`var valueOf()` 方法返回数组本身

```
var arr = [1, 'test'];
console.log(arr.valueOf()); // [1, "test"]
```

(2) toStirng()

将数组中每个值的字符串拼接成一个字符串，中间以逗号隔开

```
var arr = [1, 'test'];
console.log(arr.toString()); // 1,test
```

(3) push()

用于在数组的末端添加一个或多个元素，并返回添加新元素后的数组长度，该方法会改变原数组。

```
var arr = [];
console.log(arr.push(3)); // 1
console.log(arr.push(true)); // 2
console.log(arr); // [3, true]
```

(4) pop()

用于删除数组的最后一个元素，并返回该元素，该方法会改变原数组。

```
var arr = [1, 'test'];
console.log(arr.pop()); // "test"
console.log(arr); // [1]
```

对空数组使用pop方法，不会报错，而是返回`undefined`。

(5) join()

以参数作为分隔符，将所有数组成员组成一个字符串返回。如果不提供参数，默认用逗号分隔。

```
var arr = [1, 'test'];
console.log(arr.join());
console.log(arr.join('||'));
```

注意：如果数组成员是undefined或null或空位，会被转成空字符串。

(6) concat()

用于多个数组的合并。它将新数组的成员，添加到原数组的尾部，然后返回一个新数组，原数组不变。

```
var arr = [1, 'test'];
console.log(arr.concat([2], ['a'])); // [1, "test", 2, "a"]
```

除了接受数组作为参数，concat也可以接受其他类型的值作为参数。它们会作为新的元素，添加数组尾部。

```
var arr = [1, 'test'];
console.log(arr.concat('b', [2])); // [1, "test", "b", 2]
```

(7) shift()

用于删除数组的第一个元素，并返回该元素，该方法会改变原数组。

```
var arr = [1, 'test'];
console.log(arr.shift()); // 1
console.log(arr); // ["test"]
```

(8) unshift()

用于在数组的第一个位置添加元素，并返回添加新元素后的数组长度，该方法会改变原数组。

```
var arr = [1, 'test'];
console.log(arr.unshift('a')); // 3
console.log(arr); // ["a", 1, "test"]
```

unshift() 可添加多个元素：

```
var arr = [1, 'test'];
console.log(arr.unshift('a', 'b')); // 4
```

(9) reverse()

reverse() 用于颠倒数组中元素的顺序，返回改变后的数组，该方法会改变原数组。

```
var arr = [1, 'test'];
console.log(arr.reverse()); // ["test", 1]
```

(10) slice()

slice() 用于提取原数组的一部分，返回一个新数组，原数组不变。

第一个参数为起始位置（从0开始），如参数是负数，则表示倒数计算的位置。

第二个参数为终止位置（但该位置的元素本身不包括在内），如参数是负数，则表示倒数计算的位置。

如果省略第二个参数，则一直返回到原数组的最后一个成员。负数表示倒数第几个。

如果参数值大于数组成员的个数，或者第二个参数小于第一个参数，则返回空数组。

```
var arr = ['a', 'b', 'c', 'd'];
console.log(arr.slice(1, 3)); // ["b", "c"]
console.log(arr.slice(2)); // ["c", "d"]
console.log(arr.slice(-2, -1)); // ["c"]
console.log(arr.slice(2, 1)); // []
```

(11) splice()

`splice()` 用于删除原数组的一部分成员，并可以在被删除的位置添加新的数组成员，返回值是被删除的元素，该方法会改变原数组。

第一个参数是删除的起始位置，如果是负数，就表示从倒数位置开始删除

第二个参数是被删除的元素个数。

如果后面还有更多的参数，则表示这些就是要被插入数组的新元素。

如只是单纯地插入元素，`splice`方法的第二个参数可以设为0。

如果只提供第一个参数，等同于将原数组在指定位置拆分成两个数组。

```
var arr = ['a', 'b', 'c', 'd'];
console.log(arr.splice(1, 1)); // ["b"]
console.log(arr); // ["a", "c", "d"]

var arr = ['a', 'b', 'c', 'd'];
console.log(arr.splice(1, 1, 'f')); // ["b"]
console.log(arr); // ["a", "f", "c", "d"]

var arr = ['a', 'b', 'c', 'd'];
console.log(arr.splice(1, 0, 'h')); // []
console.log(arr); // ["a", "h", "b", "c", "d"]
```

(12) sort()

`sort()` 对数组成员进行排序，默认是按照字典顺序排序。排序后，原数组将被改变。

如果想让`sort`方法按照自定义方式排序，可以传入一个函数作为参数，表示按照自定义方法进行排序。该函数本身又接受两个参数，表示进行比较的两个元素。如果返回值大于0，表示第一个元素排在第二个元素后面；其他情况下（小于或等于），都是第一个元素排在第二个元素前面。

```
var arr = ['5', '8', '9', '2'];
console.log(arr.sort()); // ["1", "2", "4", "5"]
```

Array对象

```
var newArr = arr.sort(function(a, b) {
    return b - a;
});
console.log(newArr); // ["5", "4", "2", "1"]
```

注意：sort方法不是按照大小排序，而是按照对应字符串的字典顺序排序。也就是说，数值会被先转成字符串，再按照字典顺序进行比较。

(13) map()

map() 对数组的所有成员依次调用一个函数，根据函数结果返回一个新数组。

接受一个函数作为参数。该函数调用时，map方法会将其传入三个参数，分别是当前成员、当前位置和数组本身

```
var arr = [1, 2, 3];

var newArr = arr.map(function(v){
    return v - 1;
});
console.log(newArr); // [0, 1, 2]
```

(14) forEach()

forEach() 遍历数组的所有成员，执行某种操作，参数是一个函数。它接受三个参数，分别是当前位置的值、当前位置的编号和整个数组。

```
var arr = ['a', 'b'];
arr.forEach(function(value, index){
    console.log(index + ':' + value);
});
// 
0 : "a"
1 : "b"
```

(15) filter()

filter() 参数是一个函数，所有数组成员依次执行该函数，返回结果为true的成员组成一个新数组返回。该方法不会改变原数组。

filter方法的参数函数可以接受三个参数，第一个参数是当前数组成员的值，这是必需的，后两个参数是可选的，分别是当前数组成员的位置和整个数组。

```
var arr = [1, 2, 3];

var newArr = arr.filter(function(v){
    return (v > 2);
});
console.log(newArr); // [3];
```

(16) some()、every()

`some()` 用来判断数组成员是否符合某种条件。接受一个函数作为参数，所有数组成员依次执行该函数，返回一个布尔值。

该函数接受三个参数，依次是当前位置的成员、当前位置的序号和整个数组。只要有一个数组成员的返回值是`true`，则整个`some`方法的返回值就是`true`，否则`false`。

```
var arr = [1, 2, 3];
var bool = arr.some(function(v){
    return (v == 3);
});
console.log(bool); // true
```

`every()` 用来判断数组成员是否符合某种条件。接受一个函数作为参数，所有数组成员依次执行该函数，返回一个布尔值。

该函数接受三个参数，依次是当前位置的成员、当前位置的序号和整个数组。所有数组成员的返回值都是`true`，才返回`true`，否则`false`。

```
var arr = [1, 2, 3];
var bool = arr.every(function(v){
    return (v == 3);
});
console.log(bool); // false

var bool2 = arr.every(function(v){
    return (v > 0);
});
console.log(bool2); // true
```

`some`和`every`方法还可以接受第二个参数，用来绑定函数中的`this`关键字。

(17) reduce()、reduceRight()

`reduce()` 依次处理数组的每个成员，最终累计为一个值。从左到右处理（从第一个成员到最后一个成员）

`reduceRight()` 依次处理数组的每个成员，最终累计为一个值。从右到左（从最后一个成员到第一个成员）

两个方法的第一个参数是一个函数，该函数可接收四个参数：

1. 累积变量，默认为数组的第一个成员
2. 当前变量，默认为数组的第二个成员
3. 当前位置（从0开始）
4. 原数组

对 `reduce()`：

```

var arr = [1, 2, 3, 4, 5];
var total = arr.reduce(function(x, y){
  console.log(x, y)
  return x + y;
});
console.log(total);

// 1 2
// 3 3
// 6 4
// 10 5
//最后结果：15

```

对 `reduceRight()` :

```

var arr = [1, 2, 3, 4, 5];
var total = arr.reduceRight(function(x, y){
  console.log(x, y)
  return x + y;
});
console.log(total);

// 5 4
// 9 3
// 12 2
// 14 1
// 15

```

两个方法还可传入第二个参数，表示累积变量的初始值：

(18) `indexOf()`、`lastIndexOf()`

`indexOf()` 返回给定元素在数组中第一次出现的位置，如果没有出现则返回-1。可以接受第二个参数，表示搜索的开始位置

`lastIndexOf()` 返回给定元素在数组中最后一次出现的位置，如果没有出现则返回-1。可以看作是从最后位置开始搜索。

```

var arr = ['a', 'b', 'a', 'c'];
console.log(arr.indexOf('a')); // 0
console.log(arr.indexOf('a', 1)); // 2

console.log(arr.lastIndexOf('a')); // 2

```

Date对象

Date类型

在JavaScript中，Date类型是用来保存日期的，它能精确到1970年1月1日之前或之后的285616年。

1、创建日期对象

要创建一个日期对象，使用new操作符和Date构造函数即可：

```
var now = new Date();
```

在调用Date构造函数而不传递参数时，新创建的对象自动获得当前日期和时间。

如果要根据特定的日期和时间创建日期对象，必须传入表示该日期的毫秒数（即从UTC时间1970年1月1日起至该日期止经过的毫秒数）

注意：如果给Date构造函数传入的是其他格式，也会在后台调用Date.parse()，将其转换成毫秒数。

1.1 Date.parse()

Date.parse()接收一个表示日期的字符串参数，然后尝试根据这个字符串返回相应日期的毫秒数。

将地区设置为美国的浏览器，通常可以接受下列日期格式：

- “月/日/年”，如11/11/2016
- “英文月名 日,年”，如 January 12,2016
- “英文星期几 英文月名 日 年 时:分:秒 时区”，如Tue May 25 2016 00:00:00 GMT-0700
- ISO 8601扩展格式 YYYY-MM-DDTHH:mm:ss:sssZ，如2016-11-11T00:00:00。兼容ECMAScript 5的浏览器支持这种格式。

```
var someDate = new Date(Date.parse('2016-11-11'));
//Fri Nov 11 2016 08:00:00 GMT+0800 (中国标准时间)
```

如果传入Date.parse()方法的字符串不能表示日期，则会返回NaN。

前面说过：如果给Date构造函数传入的是其他格式，也会在后台调用Date.parse()，将其转换成毫秒数，所以我们也同样可以这样设置：

```
var someDate = new Date('2016-11-11');
//Fri Nov 11 2016 08:00:00 GMT+0800 (中国标准时间)
```

1.2 Date.UTC()

Date.UTC()同样能返回表示日期的毫秒数。不过，它传入的参数有所不同，分别是年份、基于0的月份（一月是0，二月是1，以此类推）、月中的哪一天（1到31）、小时数（0到23）、分钟、秒数以及毫秒数，只有前两个

Date对象

参数是必须的。如果没有提供月中的天数，则假设天数为1。如省略其他参数，默认为0。

```
var y2k = new Date(Date.UTC(2016, 0));
//Fri Jan 01 2016 08:00:00 GMT+0800 (中国标准时间)

var someDate = new Date(Date.UTC(2016, 0, 1, 0, 0, 0));
//Fri Jan 01 2016 08:00:00 GMT+0800 (中国标准时间)
```

上面的y2k和someDate是等价的。

如同模仿Date.parse()一样，Date构造函数也会模仿Date.UTC()，它会基于本地时区来创建。

```
var y2k = new Date(2016, 0)

var someDate = new Date(2016, 0, 1, 0, 0, 0);
```

上面两个是等价的。

1.3 Date.now()

Date.now()返回表示调用这个方法时的日期和时间的毫秒数。

```
var times = Date.now();
//1479438986198
```

我们也可以使用+操作符实现Date.now()的功能：

```
var times2 = +new Date();
//1479438986198
```

2、Date实例对象的方法

2.1 继承的方法

(1) toLocaleString()

Date类型的toLocaleString()方法会按照与浏览器设置的时区相适应的格式返回日期和时间，包含AM或PM，但不包含时区。

```
var dateString = new Date();
console.log(dateString.toLocaleString());
// 2016/11/18 下午12:19:47
```

(2) toString()

Date类型的toString()返回带有时区信息的日期和时间，其中时间一般以军用时间（即小时的范围是0到23）

```
var dateString = new Date();
console.log(dateString.toString());
// Fri Nov 18 2016 12:19:47 GMT+0800 (中国标准时间)
```

(3) valueOf()

Date类型的valueOf()返回日期的毫秒数，也就是返回Date实例本身。

```
var dateString = new Date();
console.log(dateString.valueOf());
// 1479442787239
```

2.2 日期格式化方法

Date类型提供了一些将日期格式化为字符串的方法，如下：

(1) toDateString()

toDateString()方法返回格式为星期几、月、日和年的字符串。

```
var dateString = new Date();
console.log(dateString.toDateString());
// Fri Nov 18 2016
```

(2) toTimeString()

toTimeString()返回格式为时、分、秒和时区的字符串

```
var dateString = new Date();
console.log(dateString.toTimeString());
// 12:30:35 GMT+0800 (中国标准时间)
```

(3) toLocaleDateString()

toLocaleDateString()返回格式为星期几、月、日和年的字符串（依照时区的不同，显示的格式也会所有不同）

```
var dateString = new Date();
console.log(dateString.toLocaleDateString());
// 2016/11/18
```

(4) toLocaleTimeString()

toLocaleTimeString() 返回格式为时、分、秒的字符串

```
var dateString = new Date();
console.log(dateString.toLocaleTimeString());
// 下午12:30:35
```

(5) toUTCString()

toUTCString()返回完整的UTC日期

```
var dateString = new Date();
console.log(dateString.toUTCString());
// Fri, 18 Nov 2016 04:30:35 GMT
```

2.2 日期/时间组件方法

日期/时间组件方法可分为两类：

- get类方法：获取Date对象的日期和时间
- set类方法：设置Date对象的日期和时间

2.2.1 get类方法

Date对象提供了一些获取日期和时间的方法：

`getTime()`：返回距离**1970年1月1日00:00:00**的毫秒数，等同于`valueOf`方法。
`getDate()`：返回实例对象对应每个月的几号（从**1**开始）。
`getDay()`：返回星期几，星期日为**0**，星期一为**1**，以此类推。
`getYear()`：返回距离**1900**的年数。
`getFullYear()`：返回四位的年份。
`getMonth()`：返回月份（**0**表示**1月**，**11**表示**12月**）。
`getHours()`：返回小时（**0-23**）。
`getMilliseconds()`：返回毫秒（**0-999**）。
`getMinutes()`：返回分钟（**0-59**）。
`getSeconds()`：返回秒（**0-59**）。
`getTimezoneOffset()`：返回当前时间与UTC的时区差异，以分钟表示，返回结果考虑到了夏令时因素。

上面这些方法返回的都是正数，不同值返回的范围不一样：

分钟和秒：**0** 到 **59**
 小时：**0** 到 **23**
 星期：**0**（星期天）到 **6**（星期六）
 日期：**1** 到 **31**
 月份：**0**（一月）到 **11**（十二月）
 年份：距离**1900**年的年数

上面这些get类方法返回的都是当前时区的时间，Date对象还提供了这些方法对应的UTC版本，用来返回UTC时间。

`getUTCDate()`
`getUTCFullYear()`

```
getUTCMonth()
getUTCDay()
getUTCHours()
getUTCMilliseconds()
getUTCSeconds()
getUTCMilliseconds()
```

2.2.2 set类方法

Date对象提供了一些设置日期和时间的方法：

```
 setDate(date) : 设置实例对象对应的每个月的几号（1-31），返回改变后毫秒时间戳。
setYear(year) : 设置距离1900年的年数。
setFullYear(year [, month, date]) : 设置四位年份。
setHours(hour [, min, sec, ms]) : 设置小时（0-23）。
setMilliseconds() : 设置毫秒（0-999）。
setMinutes(min [, sec, ms]) : 设置分钟（0-59）。
setMonth(month [, date]) : 设置月份（0-11）。
setSeconds(sec [, ms]) : 设置秒（0-59）。
setTime(milliseconds) : 设置毫秒时间戳。
```

注意：月份都是从0开始的。

set类方法的参数都会自动折算。下面以setDate为例，如果参数超过当月的最大天数，则向下一个月份顺延，如果参数是负数，表示从上个月的最后一天开始减去的天数。

(1) setDate()

这里重点讲一些setDate()方法，平常用得比较多。

setDate()传入特殊值时：

- 当传入0时，表示为上一个月的最后一天；
- 当传入负数时，表示上一个月最后一天之前的第几天（比如-1表示上一个月最后一天之前的一天）
- 当传入32时，如果当月有31天，32表示下个月的第一天；如果当月有30天，32表示为下个月的第二天

传入32获取当月天数

```
var now = new Date();
console.log(now);
now.setDate(32);
console.log(now);
var aMonthDay = 32 - now.getDate();
console.log(aMonthDay);

// Fri Nov 18 2016 13:13:03 GMT+0800 (中国标准时间)
// Fri Dec 02 2016 13:13:03 GMT+0800 (中国标准时间)
// 30
```

set类方法也有对应的设置UTC时间的方法：

```
setUTCDate()
setUTCFullYear()
setUTCHours()
setUTCMilliseconds()
setUTCMinutes()
setUTCMonth()
setUTCSeconds()
```

RegExp对象

RegExp类型

正则表达式 (regular expression) 是一个描述字符模式的对象。JavaScript的RegExp类表示正则表达式，String和RegExp都定义了方法。

1.1 正则表达式的定义

正则表达式有两种方法定义：

- 一种是使用正则表达式直接量，将其包含在一对斜杠 (/) 之间的字符。

```
var pattern = /s$/;
```

- 另一种是使用RegExp()构造函数。

```
var pattern = new RegExp('s');
```

上面两种方法是等价的，用来匹配所有以字母 “s” 结尾的字符串。

正则表达式的模式规则是由一个字符序列组成的，所有字母和数字都是按照字面含义进行匹配的。

1.1.1 直接量字符

JavaScript正则表达式语法也支持非字母的字符匹配，这些字符需要通过反斜杠 (\) 作为前缀进行转义。

有如下直接量字符：

\0 匹配null字符 (\u0000)。

[\b] 匹配退格键(\u0008)，不要与\b混淆。

\t 匹配制表符tab (\u0009)。

\n 匹配换行键(\u000A)。

\v 匹配垂直制表符 (\u000B)。

\f 匹配换页符 (\u000C)。

\r 匹配回车键(\u000D)。

\xnn 匹配一个以两位十六进制数 (\x00-\xFF) 表示的字符。

\uxxx 匹配一个以四位十六进制数 (\u0000-\uFFFF) 表示的unicode字符。

\cX 表示Ctrl-[X]，其中的X是A-Z之中任一个英文字母，用来匹配控制字符。

1.1.2 字符类

将直接量字符单独放进方括号就组成了字符类 (character class)。一个字符类可以匹配它所包含的任意字符。

比如：/[abc]/就和字母 “a” 、 “b” 、 “c” 中的任意一个都匹配。

我们还可以通过 “^” 符号来定义否定字符类，它匹配所有不包含括号内的字符。定义否定字符类时，将一个 “^” 符号作为左方括号内的第一个字符。比如：/[^abc]/ 匹配的是 “a” 、 “b” 、 “c” 之外的所有字

符。'

字符类还可以使用连字符来表示字符范围。比如要匹配拉丁字母表中的小写字母，可以使用 `/[a-z]/`，要匹配拉丁字母表中的任何字母和数字，则使用 `/[a-zA-Z0-9]/`。

注意：字符类的连字符必须在头尾两个字符中间，才有特殊含义，否则就是字面含义。比如，`[-9]`就表示匹配连字符和9，而不是匹配0到9。

正则表达式的字符类：

<code>[. . .]</code>	方括号内的任意字符
<code>[^ . . .]</code>	不在方括号内的任意字符
<code>.</code>	除换行符和其他Unicode行终止符之外的任意字符
<code>\w</code>	任何ASCII字符组成的单词，等价于 <code>[a-zA-Z0-9]</code>
<code>\W</code>	任何不适ASCII字符组成的单词，等价于 <code>[^a-zA-Z0-9]</code>
<code>\s</code>	任何Unicode空白符
<code>\S</code>	任何非Unicode空白符的字符
<code>\d</code>	任何非ASCII数字，等价于 <code>[0-9]</code>
<code>\D</code>	除了ASCII数字之外的任何字符，等价于 <code>[^0-9]</code>
<code>[\b]</code>	退格直接量

注意：在方括号之内也可以写上面这些特殊转义字符。但有一个特例：当转义符 `\b` 用在字符类中时，它表示的是退格符，所以要在正则表达式中按照直接量表示一个退格符时，只需使用 `[\b]` 即可。

1.1.3 重复

在正则模式之后跟随用以指定字符重复的标记。

正则表达式的重复字符语法：

<code>{n, m}</code>	匹配前一项至少n次，至多m次
<code>{n, }</code>	匹配前一项n次或者更多次，也可以说至少n次
<code>{n}</code>	匹配前一项n次
<code>?</code>	匹配前一项0次或者1次，等价于 <code>{0, 1}</code>
<code>+</code>	匹配前一项1次或多次，等价于 <code>{1, }</code>
<code>*</code>	匹配前一项0次或多次，等价于 <code>{0, }</code>

注意：在使用 `" "` 和 `"?"` 时，由于这些字符可能匹配0个字符，因此它们允许什么都不匹配。比如：正则表达式/a/实际上与字符串“bbb”匹配，因为这个字符串含有0个a。

1.1.4 非贪婪的重复

默认情况下，匹配重复字符是尽可能多的匹配，而且允许后续的正则表达式继续匹配，即匹配直到下一个字符不满足匹配规则为止，这称为“贪婪的”匹配。

当然，我们也可以使用正则表达式进行非贪婪匹配，一旦条件满足，就不再往下匹配。只须在待匹配的字符后跟随一个问号即可：`"??"`、`"+??"`、`"*??"` 或 `"{1,5}?"`。

```
/a+/ .exec('aaa') //["aaa"]
```

```
/a+?/.exec('aaa') //["a"]
```

1.1.5 选择、分组和引用

正则表达式的语法还包括指定选择项、子表达式分组和引用前一子表达式的特殊字符。

字符“|”用于分隔供选择的字符，比如：/ab|cd|ef/可以匹配字符串“ab”，也可以匹配字符串“cd”，还可以匹配字符串“ef”。

注意：选择项的尝试匹配次序是从左到右，直到发现了匹配项。如果左边的选择项匹配，就会忽略右边的匹配项，即使它产生更好的匹配。比如：当正在表达式/a|ab/匹配字符串“ab”时，它只能匹配第一个字符。

圆括号“()”可以把单独的项组合成子表达式。

带圆括号的表达式的另一个用途是允许在同一正则表达式的后部引用前面的子表达式，这是通过在字符“\”后加一位或多位数字来实现的，这个数字指定了带圆括号的子表达式在正则表达式中的位置。比如：\1引用的是第一个带圆括号的子表达式，\3引用的是第三个带圆括号的子表达式。

```
/( . )b( . )\1b\2/.test('abcabc') //true
```

上面的代码中，\1表示第一个括号匹配的内容，即第一个(.)，匹配的是“a”；\2表示第二个括号(.)，匹配的是“b”。

注意：因为子表达式可以嵌套另一个子表达式，所以引用的位置是参与计数的左括号的位置。比如：(s(ss))，\2表示的是(ss)。

对正则表达式中前一个子表达式的引用，并不是指对子表达式模式的引用，而指的是那个模式相匹配的文本的引用。

```
/(a|b)c\1/.test('aca') //true
/(a|b)c\1/.test('acb') //false
```

上面的代码中，由于(a|b)匹配的是a，所以\1所对应的也应该是a。

正则表达式的选择、分组和引用字符

| 选择，匹配的是该符号左边的子表达式或右边的子表达式

(...) 组合，将几个项组合为一个单位，这个单位可通过“*”、“+”、“?”和“|”等符号加以修饰，而且可以记住和这个组合相匹配的字符串以供此后的引用使用。

(?:...) 只组合，把项组合到一个单位，但不记忆与该组相匹配的字符

\n 和第n个分组第一次匹配的字符相匹配，组是圆括号中的子表达式（也有可能是嵌套的），组索引是从左到右的左括号数，“(?:”形式的分组不编码

1.1.6 指定匹配位置

除了匹配字符串中的字符外，有些正则表达式的元素匹配的是字符之间的位置，亦可称为正则表达式的锚。比如：\b匹配一个单词的边界，即位于\w（ASCII单词）字符和\W（非ASCII单词）之间的边界，或位于一个ASCII

单词与字符串的开始或结尾之间的边界。

正则表达式中的锚字符：

- ^ 匹配字符串的开头，在多行检索中，匹配一行的开头
- \$ 匹配字符串的结尾，在多行检索中，匹配一行的结尾
- \b 匹配一个单词的边界
- \B 匹配非单词边界的位置
- (?=p) 零宽正向先行断言，要求接下来的字符都与p匹配，但不能包括匹配p的那些字符
- (?!p) 零宽负向先行断言，要求接下来的字符不与p匹配

1.1.7 修饰符

正则表达式的修饰符，用以说明高级匹配模式的规则。修饰符是放在“/”符号之外的，也就是说，它们不是出现在两条斜杠之间的，而是在第二条斜杠之后。

在JavaScript中，支持三个修饰符：

- i 执行不区分大小写的匹配
- g 执行一个全局匹配，即找到所有的匹配，而不是在找到第一个之后就停止
- m 多行模式匹配，在这种模式下，如果待检索的字符串包含多行，那么^和\$锚字符除了匹配整个字符串的开始和结尾之外，还能匹配每行的开始和结尾

这些修饰符可以任意组合。比如：

```
/test/ig
```

1.2 用于模式匹配的String方法

String支持4种使用正则表达式的方法。

- search()：按照给定的正则表达式进行搜索，返回一个整数，表示第一个与之匹配的字符串的起始位置，如果找不到匹配的子串，将返回-1。
- match()：返回一个数组，成员是所有匹配的子字符串。
- replace()：按照给定的正则表达式进行替换，返回替换后的字符串。
- split()：按照给定规则进行字符串分割，返回一个数组，包含分割后的各个成员。

1.2.1 search()

按照给定的正则表达式进行搜索，返回一个整数，表示第一个与之匹配的字符串的起始位置，如果找不到匹配的子串，将返回-1。

```
"javascript".search(/script/i);
```

上面的代码的返回值为4

如果search()的参数不是正则表达式，则首先会通过RegExp构造函数将它转换成正则表达式，search()方法不支持全局检索，因为它忽略正则表达式参数中的修饰符g。

1.2.2 match()

match()方法的唯一参数是一个正则表达式，返回的是一个由匹配结果组成的数组。如果该正则表达式设置了修饰符g，则返回的数组包含字符串中的所有匹配结果。

```
'1 plus 2 equals 3'.match(/\d+/g) //返回["1", "2", "3"]
```

返回来的数组还带有另外两个属性：index和input，分别表示包含发生匹配的字符位置和引用的正在检索的字符串。

1.2.3 replace()

replace()方法用以执行检索与替换操作。其中第一个参数是一个正则表达式，第二个参数是要进行替换的字符串。

如果replace()的第一个参数是字符串而不是正则表达式，则replace()将直接搜索这个字符串，而不会像search()一样首先通过RegExp()将它转换为正则表达式。

replace方法的第二个参数可以使用美元符号\$，用来指代所替换的内容。

\$& 指代匹配的子字符串。

- \$` 指代匹配结果前面的文本。
- \$' 指代匹配结果后面的文本。
- \$n 指代匹配成功的第n组内容，n是从1开始的自然数。
- \$\$ 指代美元符号\$。

比如：

```
'hello world'.replace(/(\w+)\s(\w+)/, '$2 $1')
// "world hello"
```

replace方法的第二个参数还可以是一个函数，将每一个匹配内容替换为函数返回值。

```
'abca'.replace(/a/g, function(match){
  return match.toUpperCase();
});
// "AbcA"
```

replace()方法的第二个参数可以接受多个参数。第一个参数是捕捉到的内容，第二个参数是捕捉到的组匹配（有多少个组匹配，就有多少个对应的参数）。

1.2.4 split()

split()方法用以将调用它的字符串拆分为一个子串组成的数组。

```
'123,456,789'.split(',') //返回["123","456","789"]
```

split()方法的参数也可以是一个正则表达式。

1.3 RegExp对象

RegExp()构造函数带有两个字符串参数，第二个参数是可选的，它指定正则表达式的修饰符（可传入修饰符g、i、m或者它们的组合）；第一个参数包含正则表达式的主体部分，也就是正则表达式直接量中两条斜线之间的文本。

```
var regexp = new RegExp('\\d{5}', 'g')
```

上面的代码表示会全局的查找5个数字。

1.3.1 RegExp对象的属性

每个RegExp对象都包含5个属性：

- source 只读字符串，包含正则表达式的文本
- global 只读布尔值，用以说明这个正则表达式是否带有修饰符g
- ignoreCase 只读布尔值，用以说明正则表达式是否带有修饰符i
- multiline 只读布尔值，用以说明正则表达式是否带有修饰符m
- lastIndex 可读写的整数，如果匹配模式带有g修饰符，这个属性存储在整个字符串中下一次检索的开始位置。

1.3.2 RegExp的方法

RegExp对象定义了两个用于执行模式匹配操作的方法。

(1) exec()

正则对象的exec方法，可以返回匹配结果。如果发现匹配，就返回一个数组，成员是每一个匹配成功的子字符串，否则返回null。

```
/a|b|c/.exec('abc') // ["a"]
```

```
/a|b|c/.exec('qwe') // null
```

(2) test()

正则对象的test方法返回一个布尔值，表示当前模式是否能匹配参数字符串。

```
var s = /a/g;
var a = 'baba';
```

```
s.lastIndex //0  
s.test(a); //true  
  
s.lastIndex; //2  
s.test(a); //true
```

注意：如果正则表达式带有g修饰符，则每一次test方法都从上一次结束的位置开始向后匹配，也可以通过正则对象的lastIndex属性指定开始搜索的位置。

基本包装类型 (Boolean、Number、String)

基本包装类型 (Boolean、Number、String)

1、基本包装类型简介

ECMAScript提供了三个基本包装类型： Boolean 、 Number 、 String 。

实际上，每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象，从而让我们能过调用一些方法来操作这些数据。执行步骤如下：

- 创建S提让那个类型的一个实例
- 在实例上调用指定的方法
- 销毁这个实例

```
var s1 = new String('test');
var s2 = s1.substring(2);
s1 = null;
```

上面三个步骤也分别适用于Boolean和Number类型对应的布尔值和数字值。

引用类型与基本包装类型的区别在于对象的生存期：使用new操作符创建的引用类型的实例，在执行流离开当前作用域之前都一直保存在内存中，而自动创建的基本包装类型的对象，则只存在于一行代码的执行瞬间，然后立即销毁，这也是我们不能再运行时为基本类型值添加属性和方法的原因。

```
var s = 'tg';
s.age =10;
console.log(s.age); // undefined
```

上面代码执行输出的是undefined，这是由于第二行创建的String对象在执行第三行代码时已经被销毁了，第三行又创建自己的String对象，而该对象没有age属性。

当然，我们也可以将Boolean、Number和String类型当做工具方法，将任何类型转为布尔值、数值和字符串。

2、Boolean类型

Boolean类型是与布尔值对应的引用类型。常用于生成布尔值的包装对象的实例。

```
var bool = new Boolean(true);
```

Boolean类型的实例重写了valueOf()方法，返回的基本类型值true或false，重写了toString()方法，返回字符串“true”和“false”。

要注意一点的是，即使你使用false创建一个Boolean实例对象，当进行逻辑运算时，它会被转为true，因为它是

一个对象，而所有对象在逻辑运算中都会返回true。

```
var bool = new Boolean(false);
if(bool){
    console.log(true);
}

// true
```

对于Boolean类型，我们几乎不用用它来创建实例对象。

3、Number类型

Number是与数字值对应的引用类型。

创建Number对象：

```
var num = new Number();
var num2 = new Number(10);
```

Number类型的toString()方法返回数值的字符串，可传递一个表示基数的参数，默认是10进制。

```
var num = 10;
console.log(num.toString()); // "10"
console.log(num.toString(2)); // "1010"
```

3.1 属性

Number类型有以下属性：

Number.POSITIVE_INFINITY：正的无限，指向Infinity。
Number.NEGATIVE_INFINITY：负的无限，指向-Infinity。
Number.NaN：表示非数值，指向NaN。
Number.MAX_VALUE：表示最大的正数，相应的，最小的负数为-Number.MAX_VALUE。
Number.MIN_VALUE：表示最小的正数（即最接近0的正数，在64位浮点数体系中为5e-324），相应的，最接近0的负数为-Number.MIN_VALUE。
Number.MAX_SAFE_INTEGER：表示能够精确表示的最大整数，即9007199254740991。
Number.MIN_SAFE_INTEGER：表示能够精确表示的最小整数，即-9007199254740991。

3.2 格式化方法

(1) Number.prototype.toFixed()

toFixed()方法会按照指定的小数位来返回数值的字符串表示（四舍五入）。

```
var num = 10.005;
console.log(num.toFixed(2)); // "10.01"
```

注意：如果没有小数位，则以0填补。

```
var num = 10;
console.log(num.toFixed(2)); // "10.00"
```

注意：toFixed()参数的有效范围为0~20个小数位的数值。

(2) Number.prototype.toExponential()

toExponential()方法用于将一个数转为科学计数法形式。

```
var num = 10;
console.log(num.toExponential()); // 1e+1
```

toExponential()方法的页可以接受一个参数，参数表示小数点后有效数字的位数，范围为0到20，超出这个范围

```
var num = 1234;
console.log(num.toExponential(2)); // 1.23e+3
console.log(num.toExponential(1)); // 1.2e+3
```

(3) Number.prototype.toPrecision()

toPrecision()方法可能返回固定大小格式，也可能返回指数格式，具体规则是看哪种格式最合适的。它也接收一个参数，表示数值的所有数字的位数（不包含指数部分）

```
var num = 99;
console.log(num.toPrecision(1)); // 1e+2
console.log(num.toPrecision(3)); // 99.0
```

4、String类型

String类型是字符串的对象包装类型。

创建String对象

```
var text = new String('tg');
```

String构造函数有一个静态方法：

(1) String.fromCharCode()

fromCharCode()方法的参数是一系列Unicode码点，返回对应的字符串。

```
var str = String.fromCharCode(104, 101, 108, 108, 111);
console.log(str); // "hello"
```

4.1 String实例对象的属性和方法

4.1.1 属性

length属性

String类型的每个实例都有一个length属性，表示字符串中包含多少个字符（从0开始）。

4.1.2 方法

String类型提供了很多方法：

(1) 字符方法

charAt() 接收一个基于0的字符位置的参数，返回指定位置的字符

```
var str = 'hello world';
console.log(str.charAt(1)); // e
```

charCodeAt() 接收一个基于0的字符位置的参数，返回给定位置字符的Unicode码点（十进制表示）

```
var str = 'hello world';
console.log(str.charCodeAt(1)); // 101
```

String对象实例是一个类数组对象，所以我们可以使用方括号表示法访问特定字符。

```
var str = 'hello world';
console.log(str[1]); // e
```

(2) 字符串操作方法

concat() 用于连接两个字符串，参数可以是一个或多个字符串，返回拼接得到的新字符串

```
var str1 = 'hello';
var str2 = str1.concat(' world');
console.log(str2); // "hello world"
console.log(str1); // "hello"
```

slice(start,end) 用于从原字符串取出子字符串并返回子字符串，不改变原字符串。

第一个参数：表示子字符串的开始位置，如果参数是负值，表示从结尾开始倒数计算的位置，即该负值加上字符串长度

第二个参数：可选，表示子字符串的结束位置（不含该位置）；如果参数是负值，表示从结尾开始倒数计算的位置，即该负值加上字符串长度；如果不设置，表示将字符串的长度作为结束位置

```
var str = 'hello world';
console.log(str.slice(1,3)); // "el"
console.log(str.slice(1)); // "ello world"
```

基本包装类型 (Boolean、Number、String)

```
console.log(str.slice(-1)); // "d"
console.log(str); // "hello world"
```

注意：如果第一个参数大于第二个参数，则返回空字符串

`substring()` 用于从原字符串取出子字符串并返回，不改变原字符串，也接收一个或两个参数。

第一个参数表示子字符串的开始位置；

第二个位置表示结束位置，如果不设置，表示将字符串的长度作为结束位置

注意：`substring()`方法有些不一样，当传入负数时，它会自动将负数转为0；当第一个参数大于第二个参数时，它会将两个参数的位置对换。

```
var str = 'hello world';
console.log(str.substring(-1, 2)); // "he"
console.log(str.substring(3, 1)); // "el"
```

从上面运行结果我们可以看到，`substring()`自动将-1转为0；将第二行的3和1交换了。

`substr()` 用于从原字符串取出子字符串并返回，不改变原字符串。

第一个参数：表示子字符串的开始位置；

第二个参数：表示子字符串的长度。如果第一个参数是负数，表示倒数计算的字符位置。如果第二个参数是负数，将被自动转为0，因此会返回空字符串；如果不设置，表示将字符串的长度作为结束位置

```
var str = 'hello world';
console.log(str.substr(1, 2)); // "el"
console.log(str.substr(-1, 2)); // "d"
console.log(str.substr(1, -1)); // ""
```

(3) 字符串位置方法

`indexOf()` 返回给定元素在字符串中第一次出现的位置，如果没有出现则返回-1。第一个参数为要查找的子字符串，可以接受第二个参数，表示搜索的开始位置

```
var str = 'hello world';
console.log(str.indexOf('o')); // 4
console.log(str.indexOf('h', 2)); // -1
```

`lastIndexOf()` 返回给定元素在字符串中最后一次出现的位置，如果没有出现则返回-1。第一个参数为要查找的子字符串，可以接受第二个参数，表示搜索的开始位置

```
var str = 'hello world';
console.log(str.lastIndexOf('o')); // 4
console.log(str.lastIndexOf('h', 2)); // 0
```

(4) trim()

trim() 是ECMAScript 5新增的，用于去除字符串两端的空格，返回一个新字符串，不改变原字符串。

```
var str = '    hello';
console.log(str.trim()); // "hello"
```

(5) 字符串大小写转换

toLowerCase() 用于将一个字符串全部转为小写，返回一个新字符串，不改变原字符串。

toUpperCase() 将字符串的字符全部转为大写

```
var str = 'hello';
var upper = str.toUpperCase();
var lower = upper.toLowerCase();
console.log(upper); // "HELLO"
console.log(lower); // "hello"
```

(6) 字符串的模式匹配方法

match() 用于确定原字符串是否匹配某个子字符串，返回一个数组，成员为匹配的第一个字符串。如果没有找到匹配，则返回null。

search() 返回值为匹配的第一个位置。如果没有找到匹配，则返回-1。

replace() 用于替换匹配的子字符串，一般情况下只替换第一个匹配（除非使用带有g修饰符的正则表达式）。

split() 按照给定规则分割字符串，返回一个由分割出来的子字符串组成的数组。还可传入第二个参数，决定了返回数组的成员数。

(7) localeCompare()方法

localeCompare(s2) 用于比较两个字符串。它返回一个整数：

如果小于0，表示第一个字符串小于第二个字符串；

如果等于0，表示两者相等；

如果大于0，表示第一个字符串大于第二个字符串。

```
console.log('a'.localeCompare('a')); // 0
console.log('a'.localeCompare('b')); // -1
```

注意：比较的是字符串在字母表中的顺序。

单体内置对象 (Global、Math)

单体内置对象 (Global、Math)

内置对象也就是我们不必显式地实例化，直接就可以使用了，因为它们已经实例化了。

1、Global 对象

Global对象是一个全局对象。所有在全局作用域中定义的属性和函数，都是Global对象的属性，比如： isNaN()、isFinite()、parseInt()以及parseFloat()，实际上都是Global对象的方法，它还包括了其他一些方法：

1.1 Global的方法

(1) URI编码方法

Global对象的 `encodeURI()` 和 `encodeURIComponent()` 方法可以对URI (Uniform Resource Identifiers，通用资源标识符) 进行编码，以便发送给服务器 (在GET 请求中很重要的方法)。

有效的URI中不能包含某些字符，比如空格。

`encodeURI()` 主要用于整个URI；`encodeURIComponent()` 主要用于对URI中的某一段 (比如 : illegal value.html这一段)

`encodeURI()` 和 `encodeURIComponent()` 的区别：

- `encodeURI()` 不会对本身属于URI的特殊字符进行编码 (比如冒号、正斜杠、问号和井字号) ；但 `encodeURIComponent()` 则会对它发现的任何非标准字符进行编码。

```
var uri = 'http://www.example.com/illegal value.html#start';
console.log(encodeURI(uri)); // http://www.example.com/illegal%20value.html#start
console.log(encodeURIComponent(uri)); // http%3A%2F%2Fwww.example.com%2Fillegal%20value.html%23start
```

从上面的例子的结果，我们也可以看出两者的区别，`encodeURI()`只对空格进行了编码，而 `encodeURIComponent()`使用对应的编码替换了所有非标准的字符。

与 `encodeURI()` 和 `encodeURIComponent()` 方法对应的是 `decodeURI()` 和 `decodeURIComponent()`。

`decodeURI()`只能对使用`encodeURI()`替换的字符进行解码；`decodeURIComponent()`能够解码使用 `encodeURIComponent()`编码的所有字符。

```
console.log(decodeURI('http://www.example.com/illegal%20value.html#start'));
// http://www.example.com/illegal value.html#start
console.log(decodeURIComponent('http%3A%2F%2Fwww.example.com%2Fillegal%20value.
```

```
html%23start'));  
// http://www.example.com/illegal value.html#start
```

(2) eval() 方法

eval()方法类似一个完整的ECMAScript解析器，它接受一个参数，既要执行的JavaScript字符串。

```
eval("alert('hi'));  
//等价于  
alert('hi');
```

当解析器发现代码中调用eval()方法时，它会将传入的参数当作实际的ECMAScript语句来解析，然后把执行结果插入到原位置。

通过eval()执行的代码被认为是包含该次调用的执行环境的一部分，因此被执行的代码具有与该执行环境相同的作用域链。

```
var name = 'tg';  
eval("console.log(name)"); // "tg"  
  
eval("function test() { console.log(1); }");  
test(); // 1
```

注意：在eval()创建的任何变量或函数都不会被提升。而且在严格模式下，在外部是访问不到eval()中创建的任何变量或函数。

1.2 Global对象的属性

undefined、NaN、Infinity、Object、Array、Function、Boolean、String、Number、Date、RegExp、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError

1.3 window对象

ECMAScript并没有指出如何直接访问Global对象，但Web浏览器都是讲这个全局对象作为window对象的一部分加以实现的。因此，在全局作用域总声明的所有变量和函数，都会成为window对象的属性。

2、Math 对象

Math对象中保存了数学公式和信息。

2.1 Math对象的属性

Math.E	自然对数的底数，即常量e的值
Math.LN10	10 的自然对数
Math.LN2	2 的自然对数
Math.LOG2E	以 2 为底e的对数
Math.LOG10E	以 10 为底e的对数
Math.PI	π 的值
Math.SQRT1_2	$1/2$ 的平方根（即 2 的平方根的倒数）

Math.SQRT2 2的平方根

2.2 Math对象的方法

(1) Math.min()和Math.max()

Math.min() 和 Math.max() 分别用于确定一组数值中的最小值和最大值，这两个方法都可以接收任意个数值参数。

```
console.log(Math.max(3, 10, 2, 100)); // 100
```

```
console.log(Math.min(3, 10, 2, 100)); // 2
```

如果要找到数组中的最大或最小值，可以这样：

```
var arr = [3, 10, 2, 100];
var max = Math.max.apply(Math, arr);

console.log(max); // 100
```

apply()方法是用来改变一个函数内的this指向，第一个参数就是要this指向的对象，第二个参数是一个数组。

(2) 舍入方法

Math.ceil() 执行向上舍入，即它总是取最接近数值且大于数值的整数

```
console.log(Math.ceil(1.4)); // 2
console.log(Math.ceil(1.5)); // 2
```

Math.floor() 执行向下舍入，即它总是取最近数值且小于数值的整数。

```
console.log(Math.floor(1.4)); // 1
console.log(Math.floor(1.5)); // 1
```

Math.round() 执行标准舍入，即它总是将数值四舍五入为最接近的整数。

```
console.log(Math.round(1.4)); // 1
console.log(Math.round(1.5)); // 2
```

(3) random() 方法

Math.random() 方法返回介于0和1之间的一个随机数（不包括0和1）。

```
console.log(Math.random()); // 介于0~1之间的值
```

如果你要取其他的随机数，可以这样：

```
var rand = Math.random() * 10; // 介于0~10之间的值  
var iRand = Math.floor(rand); // 介于0~10之间的整数
```

总结了一个方法：

```
function getRandom(max, min) {  
    min = arguments[1] || 0;  
    return Math.floor(Math.random() * (max - min + 1) + min);  
};
```

getRandom()方法接受两个参数：应该返回的最小值和最大值。

(4) 其他方法

Math.abs(num) 返回num的绝对值

Math.exp(num) 返回Math.E的num次幂

Math.log(num) 返回num的自然对数

Math.pow(num, power) 返回num的power次幂

Math.sqrt(num) 返回num的平方根

Math.acos(x) 返回x的反余弦值

Math.asin(x) 返回x的反正弦值

Math.atan(x) 返回x的反正切值

Math.atan2(y, x) 返回y/x的反正切值

Math.cos(x) 返回x的余弦值

Math.sin(x) 返回x的正弦值

Math.tan(x) 返回x的正切值

console对象

console对象

console 对象是JavaScript的原生对象，可以将信息输出在控制台。

1、打开控制台

不同系统平台打开浏览器的控制台可能有些不一样，下面以chrome浏览器为例，可以使用下列方式打开：

- 按F12
- 鼠标右键里选择“检查”
- 在右上角的菜单中点击“更多工具/开发者工具”

打开控制台以后，你可以看到下列不同的面板：

Elements：查看网页的HTML源码和CSS代码。

Resources：查看网页加载的各种资源文件（比如代码文件、字体文件、css文件等），以及在硬盘上创建的各种内容（比如本地缓存、Cookie、Local Storage等）。

Network：查看网页的HTTP通信情况。

Sources：查看网页加载的所有源码。

Timeline：查看各种网页行为随时间变化的情况。

Profiles：查看网页的性能情况，比如CPU和内存消耗。

Console：用来运行JavaScript命令。

2、console对象的方法

2.1 log()、warn()、info()、error()、debug()

console.log()，在控制台中打印信息，它会自动在每次输出的结尾，添加换行符，它可以接受任何字符串、数字和JavaScript对象，也可以接受换行符n以及制表符t。

```
console.log(1); // 1
console.log('hello'); // "hello"
```

它可以接受多个参数，将它们的结果连接起来输出。

```
console.log('a', 'b', 1); // a b 1
```

当然 console 可不止这一个方法，它还有：

- `console.info` 用于输出提示性信息，会带上一个蓝色图标
- `console.error` 用于输出错误信息，会在最前面带上红色的叉，表示出错
- `console.warn` 用于输出警示信息，会在最前面带上黄色三角

- console.debug用于输出调试信息

```
console.info("提醒");
console.error("报错了");
console.warn("警告");
console.debug("调试信息");
```

console对象的上面5种方法，如果第一个参数是格式字符串（使用了格式占位符），将依次用后面的参数替换占位符，然后再进行输出。

不过，占位符的种类比较少，只支持下列五种：

- 字符 (%s)
- 整数 (%d或%i)
- 浮点数 (%f)
- 对象 (%o)
- CSS格式字符串 (%c)

例子：

```
console.log('%s%s', 1, 2); // 12

console.log("%c自定义样式", "font-size:20px;color:green");
console.log("%c我是%c自定义样式", "font-size:20px;color:green", "font-weight:bold;color:red");
```

2.2 trace()、clear()

console.trace() 方法显示当前执行的代码在堆栈中的调用路径。

用 console.clear() 清空控制台。

2.3 dir()、dirxml()

console.dir 直接将该DOM结点以DOM树的结构进行输出，可以详细查对象的方法发展等等

```
console.log({name: 'tg', age: 12});
// Object {name: "tg", age: 12}

console.dir({name: 'tg', age: 12});
// Object
  name: "tg",
  age: 12,
  __proto__: Object
```

console.dirxml 用来显示网页的某个节点（node）所包含的html/xml代码

```

<table>
  <thead>
    <tr>
      <th></th>
      <th></th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>
</table>
<script>
  var table=document.querySelector("table");
  console.dirxml(table);
</script>

//输出：
<table>
  <thead>
    <tr>
      <th></th>
      <th></th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>
</table>

```

2.4 group()、groupEnd()、groupCollapsed()

`console.group` 和 `console.groupend` 这两个方法用于将显示的信息分组。它只在输出大量信息时有用，分在一组的信息，可以用鼠标折叠/展开。

`console.group` 输出一组信息的开头

`console.groupEnd` 结束一组输出信息

`console.groupCollapsed` 方法与 `console.group` 方法很类似，唯一的区别是该组的内容，在第一次显示时是收起的（collapsed），而不是展开的。

2.5 assert()

`console.assert` 对输入的表达式进行断言，接受两个参数，第一个参数是表达式，第二个参数是字符串。只有表达式为`false`时，才输出相应的信息到控制台，否则不输出。

```

var.isTrue=true;
console.assert(isTrue, "我是错误");
isTrue=false;
console.assert(isTrue, "我是错误2"); // "我是错误2"

```

2.6 count()

`console.count` 当你想统计代码被执行的次数，这个方法很有用

```
function play(){
    console.count("执行次数：");
}
play(); // 执行次数：1
play(); // 执行次数：2
play(); // 执行次数：3
```

2.7 time()、timeEnd()

这两个方法用于计时，可以算出一个操作所花费的准确时间。

`console.time` 计时开始

`console.timeEnd` 计时结束

```
console.time("array");
var a=0;
for(var i=0;i<100000;i++){
    a += i;
}
console.timeEnd("array"); // array : 6.063ms
```

2.8 profile()、profileEnd()

`console.profile` 和 `console.profileEnd` 配合一起使用来查看CPU使用相关信息

打开浏览器的开发者工具，在profile面板中，可以看到这个性能调试器的运行结果。

2.9 timeLine()、timeLineEnd()

`console.timeLine` 和 `console.timeLineEnd` 配合一起记录一段时间轴

3、自定义console对象的方法

console对象的所有方法都可以被覆盖，也就是说，我们可以自定义方法。

```
console['log'] = console['log'].bind(console, '温馨提示');

console.log('自定义方法'); // 温馨提示 自定义方法
```

DOM

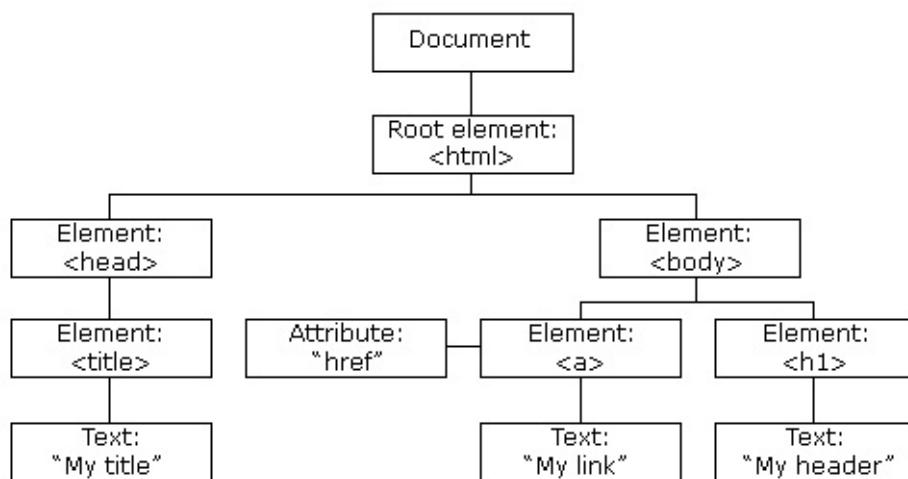
DOM

1、DOM

文档对象模型 (Document Object Model , DOM) 是表示和操作HTML和XML文档内容的基础API。

当网页被加载时，浏览器会根据DOM模型，将结构化文档（比如HTML和XML）解析成一系列的节点，再由这些节点组成一个树状结构 (DOM Tree) 。

如下图：



上图中的每一个方框是文档的一个节点，它表示一个Node对象，而所有节点组成了节点树 (DOM树) 。

节点有7种类型：

Document：整个文档树的顶层节点
DocumentType：doctype标签（比如`<!DOCTYPE html>`）
Element：网页的各种HTML标签（比如`<body>`、`<a>`等）
Attribute：网页元素的属性（比如`class="right"`）
Text：标签之间或标签包含的文本
Comment：HTML或XML的注释
DocumentFragment：文档的片段
Document和**Element**是两个重要的DOM类。

1.1 节点之间的关系

在一个节点之上的直接节点是其 父节点，在其下一层的直接节点是其 子节点。在同一层上具有相同父节点的节点是 兄弟节点。在一个节点之下所有层级的一组节点是其 后代节点。一个节点的任何父节点、祖父节点和其上层的所有节点是 祖先节点。

通用的 `Document` 和 `Element` 类型与 `HTMLDocument` 和 `HTMLElement` 类型之间是有严格区别的。

`Document` 类型代表一个HTML或XML文档， `Element` 类型代表该文档中的一个元素。

`HTMLDocument` 和 `HTMLElement` 子类只是针对于HTML文档和元素。

`CharacterData` 通常是 `Text` 和 `Comment` 的祖先，它定义两种节点所共享的方法。

`Attr` 节点类型代表XML或HTML属性。

`Element` 类型定义了将属性当做“名/值”对使用的方法。

`DocumentFragment` 类型在实际文档中并不存在的一种节点，它代表一系列没有常规父节点的节点。

1.2 选取文档元素

在JavaScript中，有多种方法选取元素。

- 用指定的id属性
- 用指定的name属性
- 用指定的标签名字
- 用指定的CSS类
- 匹配指定的CSS选择器

1.2.1 用指定ID选取元素

任何HTML元素都可以有一个id元素，但在文档中该值必须唯一，即同一个文档中的元素不能出现有相同的ID。

可以用 `Document` 对象的 `getElementById()` 方法选取特定ID的元素。

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

document.getElementById('div');
```

1.2.2 用指定名字选取元素

一些HTML元素拥有 `name` 属性（比如 `<form>`、`<radio>`、``、`<frame>`、`<embed>` 和 `<object>` 等），非唯一，所以多个元素可能有相同的名字。

基于`name`属性的值选取HTML元素，可以使用 `Document` 对象的 `getElementsByName()` 方法，返回一个`NodeList`对象（类数组对象）。

```
<input name="input"/>

var inputs = document.getElementsByName('input');
inputs[0].tagName //input
```

注意：`getElementsByName()` 定义在`HTMLDocument`类中，而在`Document`类中，所以它只针对HTML文档可用，在XML中不可用。

1.2.3 用指定标签名选取元素

`Document` 对象的 `getElementsByTagName()` 方法可用来选取指定类型（标签名）的所有HTML或XML元素，也是返回一个`NodeList`对象。

```
document.getElementsByTagName('span') // 选取所有span元素
```

给 `getElementsByName()` 传递通配符参数 “*” , 将获得一个代表文档中所有元素的 `NodeList` 对象。在 `Element` 类中也同样定义了 `getElementsByName()` 方法 , 其原理和 `Document` 版本是一样的 , 不过它只选取调用该方法的元素的后代元素。

下面的代码就是查找文档中第一个元素里面的所有元素。

```
var p = document.getElementsByTagName('p')[0];
var span = p.getElementsByTagName('span');
```

1.2.4 用指定CSS类选取元素

HTML元素的 `class` 属性值是一个以空格隔开的列表 , 可以为空或包含多个标识符。

在HTML文档和HTML元素上 , 我们可以调用 `getElementsByClassName()` 来选择指定CSS类的元素 , 它返回一个实时的 `NodeList` 对象 , 包含文档或元素所有匹配的后代节点。

`getElementsByClassName()` 只需要一个字符串参数 , 但是该字符串可以由多个空格隔开的标识符组成 , 只有当元素的 `class` 属性值包含所有指定的标识符时才匹配。

在 `Element` 类中也同样定义了 `getElementsByClassName()` 方法 , 其原理和 `Document` 版本是一样的 , 不过它只选取调用该方法的元素的后代元素。

1.2.5 通过CSS选择器选取元素

`Document` 对象的方法 `querySelectorAll()` , 它接受一个CSS选择器的字符串参数 , 返回一个代表文档中匹配选择器的所有元素的 `NodeList` 对象 , 并不是实时的。如果没有匹配的元素 , 则返回一个空的 `NodeList` 对象。

```
document.querySelectorAll('.div') // 匹配所有class名为div的元素
```

还有一个 `querySelector()` 方法 , 其原理和 `querySelectorAll()` 是一样的 , 不过它返回第一个匹配的元素 (以文档顺序) , 如果没有匹配的元素就返回 `null` 。

它们支持复杂的CSS选择器。

```
// 选中data-tip属性等于title的元素
document.querySelectorAll('[data-tip="title"]');

// 选中div元素, 那些class含ignore的除外
document.querySelectorAll('div:not(.ignore)');
```

但是 , 它们不支持CSS伪元素的选择器 (比如 :first-line 和 :first-letter) 和伪类的选择器 (比如 :link 和 :visited) ,

即无法选中伪元素和伪类。

这两个方法在 `Element` 节点上也有定义。

1.2.6 `document.all[]`

`document.all[]`也可用来选择元素，不过已经废弃了。

```
document.all[0] //文档中第一个元素
document.all['navbar'] // id或name为“navbar”的元素
```

1.3 文档结构和遍历

1.3.3 作为节点树的文档

`Document` 对象、它的 `Element` 对象 和文档中表示文本的 `Text` 对象都是 `Node` 对象。

`Node` 属性：

(1) `parentNode`

该节点的父节点，或者针对类似 `Document` 对象应该是 `null`，因为它没有父节点。

(2) `childNodes`

返回只读的类数组对象（`NodeList` 对象），它是该节点的子节点的实时表。

注意：该属性还包括文本节点和评论节点。

(3) `firstChild`、`lastChild`

该节点的子节点中的第一个和最后一个，如果该节点没有子节点则为 `null`

注意：这两个属性返回的除了 HTML 元素子节点，还可能是文本节点或评论节点。

(4) `nextSibling`、`previousSibling`

该节点的兄弟节点中的前一个和下一个。具有相同父节点的两个节点称为兄弟节点。节点的顺序反映了它们在文档中出现的顺序。这两个属性将节点之间以双向链表形式连接起来。

注意：这两个属性返回的除了 HTML 元素子节点，还可能是文本节点或评论节点。

(5) `textContent`

返回该节点和它的所有后代节点的文本内容。

```
<div id="div">我是<span>textContent</span></div>

document.getElementById('div').textContent // 我是textContent
```

(6) `nodeType`

该节点的类型。

9 : `Document` 节点

1 : `Element` 节点

3 : `Text` 节点

8 : Comment节点

11 : DocumentFragment节点

(7) nodeValue

Text节点或者Comment节点的文本内容。只有Text节点和Comment节点的nodeValue可以返回结果，其他类型的节点一律返回null。

(8) nodeName

元素的标签名，以大写形式表示。

nodeType和nodeName

类型	nodeName	nodeType
ELEMENT_NODE	大写的HTML元素名	1
ATTRIBUTE_NODE	等同于Attr.name	2
TEXT_NODE	#text	3
COMMENT_NODE	#comment	8
DOCUMENT_NODE	#document	9
DOCUMENT_FRAGMENT_NODE	#document-fragment	11
DOCUMENT_TYPE_NODE	等同于DocumentType.name	10

使用这些node属性，我们可以便捷的得到各个节点的引用

```
document.childNodes[0].childNodes[1]
//等价于
document.firstChild.firstChild.nextSibling
```

1.3.4 作为元素树的文档

当我们的关注点在文档的元素上而非它们之间的文本上时，JavaScript提供了另外一个API，它将文档看做是E乐门头对象树，忽略部分文档：Text和Comment节点。

属性

(1) children

类似childNodes，返回一个NodeList对象，但children列表只包含Element对象。

注意：Text和Comment节点没有children属性，意味着node.parentNode属性不可能返回Text或Comment节点。任何Element的parentNode总是另一个Element，或者，追溯到树根的Document或DocumentFragment节点。

(2) firstElementChild、lastElementChild

类似firstChild和lastChild，但只代表子Element。

(3) nextElementSibling、previousElementSibling

类似nextSibling和previousSibling，但只代表兄弟Element。

(4) childElementCount

子元素的数量。返回的值和children.length值相等。

(5) offsetParent

offsetParent属性返回当前HTML元素的最靠近的、并且CSS的position属性不等于static的父元素。如果某个元素的所有上层节点都将position属性设为static，则Element.offsetParent属性指向<body>元素。

1.4 NodeList对象和HTMLCollection对象

1.4.1 NodeList对象

NodeList实例对象是一个类数组对象，它的成员是节点对象。比如node.childNodes、document.querySelectorAll()返回的都是NodeList实例对象。

```
document.childNodes instanceof NodeList //true
```

NodeList实例对象可能是动态集合，也可能是静态集合。所谓动态集合就是一个活的集合，DOM树删除或新增一个相关节点，都会立刻反映在NodeList接口之中。Node.childNodes返回的，就是一个动态集合。

NodeList接口实例对象提供length属性和数字索引，因此可以像数组那样，使用数字索引取出每个节点，但是它本身并不是数组，不能使用pop或push之类数组特有的方法。

1.4.2 HTMLCollection对象

HTMLCollection实例对象与NodeList实例对象类似，也是节点的集合，返回类数组对象。

在HTMLDocument类中，有一些快捷属性来访问各种各样的节点。比如：images、forms和links等属性指向类数组的、<form>和<a>（只包含那些有href属性的<a>标签）元素集合。这些属性都是返回HTMLCollection实例对象。

HTMLDocument也定义了embeds和plugins属性，它们是同义词，都是HTMLCollection类型的<embed>元素的集合。anchors是非标准属性，它指代有一个name属性的<a>元素。

在HTML5中，加入了scripts，它是HTMLCollection类型的

DOM-属性和CSS

DOM--属性和CSS

一、属性

HTML元素由一个标签和一组称为属性 (attribute) 的名/值对组成。

1、HTML属性作为Element的属性

表示HTML文档元素的HTMLElement对象定义了读写属性，它们映射了元素的HTML属性。

HTMLElement定义了通用的HTTP属性（如id、标题lang和dir）的属性，以及事件处理程序属性（如onclick）。

特定的Element子类型为其元素定义了特定的属性。例如：

```
var image = document.getElementById("myimage")
var imgurl = image.src //src属性是图片的URL
image.id === "myimage" // 判定要查找图片的id
```

同样的，我们也可以为一个

BOM

BOM

Window对象是客户端JavaScript程序的全局对象。

11.1 计时器

`setTimeout()` 和 `setInterval()` 可以用来注册在指定的时间之后单次或重复调用的函数。两者都是客户端JavaScript中的全局函数，也就是Window对象的方法。

(1) setTimeout()

`setTimeout()` 方法用来实现一个函数在指定的毫秒数之后运行，返回一个值，这个值可以传递给 `clearTimeout()` 用于取消这个函数的执行。

```
var times = setTimeout(function(){}, 1000); //1000毫秒后执行
clearTimeout(times); //取消执行
```

(2) setInterval()

`setInterval()`方法和`setTimeout()`一样，只不过这个函数会在指定毫秒数的间隔里重复调用，也返回一个值，这个值传递给`clearInterval()`，用于取消后续函数的调用。

```
var times = setInterval(function(){}, 1000); //每隔1000毫秒调用一次function
clearInterval(times); //取消后续函数执行
```

注意：如果以0毫秒的超时时间来调用`setTimeout()`，那么指定的函数不会立刻执行。相反，会把它放到队列中，等到前面处于等待状态的事件处理程序全部执行完成后，再“立即”调用它。

11.2 浏览器定位和导航

Window对象的 `location` 属性引用的是 `Location` 对象，它表示该窗口中当前显示的文档的URL，并定义了方法来使窗口载入新的文档。

Document对象的`location`属性也引用到`Location`对象：

```
window.location === document.location //总是返回true
```

Document对象也有一个URL属性，是文档首次载入后保存该文档的URL的静态字符串。如果定位到文档中的片断标识符（如`#top`），`Location`对象会做相应的更新，而`document.URL`属性却不会改变。

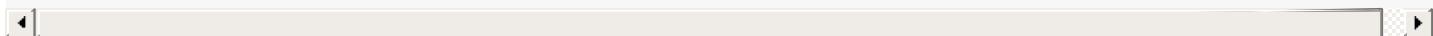
11.2.1 解析URL

`Location`对象`href`属性是一个字符串，`href`属性包含URL的完整文本。`Location`对象的`toString()`方法返回`href`属

性的值，因此在会隐式调用toString()的情况下，可以用location代替location.href。

```
// 当前网址为 http://user:passwd@www.example.com:4097/path/a.html?x=111#part1
location.href // "http://user:passwd@www.example.com:4097/path/a.html?x=111#part1"

location.protocol // "http:"
location.host // "www.example.com:4097"
location.hostname // "www.example.com"
location.port // "4097"
location.pathname // "/path/a.html"
location.search // "?x=111"
location.hash // "#part1"
location.user // "user"
location.password // "passed"
```



protocol、host、hostname、port、pathname和search属性是“URL分解”属性，是可写的，对它们重新赋值的话，会改变URL的位置，并且浏览器会载入一个新的文档。

`Location` 对象的方法：

- `location.assign()`：使窗口载入并显示指定的URL中的文档。
- `location.replace()`：和assign()方法类似，但它在载入新文档之前会从浏览历史中把当前文档删除。这样“后退”按钮就不会把浏览器带回到原始文档。
- `location.reload()`：重新载入当前文档，可传入一个布尔值为参数，默认false。如果为true，则优先从服务器重新加载；否则优先从本地缓存中重新加载。

当然，我们还有更直接跳转到新页面的方法：

```
location = 'http://baidu.com';
//或者
location.href = 'http://baidu.com';
```

纯粹的片断标识符是相对URL的一种类型，它不会让浏览器载入新文档，而是使浏览器滚动到文档的某个位置。

注意：`#top`标识符是个特殊值：如果文档中没有元素的ID是“`top`”，它会让浏览器滚动到文档开始处。

```
location = '#top'; //跳转到文档的顶部
```

11.3 浏览历史

Window对象的history属性引用的是该窗口的History对象。History对象是用来把窗口浏览历史用文档和文档状态列表的形式表示。

History对象的length属性表示浏览历史表中的元素数量。比如你在当前窗口访问了三个不同的网址，那么`history.length`就等于3。

History对象还提供了一系列的方法，让我们在历史记录中自由前进和后退。

- `back()`：移动到上一个访问页面，等同于浏览器的后退键。
- `forward()`：移动到下一个访问页面，等同于浏览器的前进键。
- `go()`：接受一个整数作为参数，移动到该整数指定的页面，比如`go(1)`相当于`forward()`，`go(-1)`相当于`back()`。

如果移动的位置超出了访问历史的边界，以上三个方法并不报错，而是默默的失败。

`history.go(0)`相当于刷新当前页面。

```
history.go(0)
```

注意：如果窗口包含多个子窗口（比如

Event 事件

Event 事件

客户端JavaScript程序采用了异步事件驱动编程模型。

一、相关术语

事件流 描述的是从页面中接收事件的顺序。

事件 就是Web浏览器通知应用程序发生了什么事情。

事件类型 (event type) 是一个用来说明发生什么类型事件的字符串。例如， “mousemove” 表示用户移动鼠标， “keydown” 表示键盘上某个键被按下等等。

事件目标 (event target) 是发生的事件与之相关的对象。当讲事件时，我必须同时指明类型和目标。比如：window上的load事件或 <button> 元素的click事件。

在客户端JavaScript应用程序中，Window、Document和Element对象是最常见的事件目标。

事件处理程序 (event handler) 或 事件监听程序 (event listener) 是处理或响应事件的函数。

事件对象 (event object) 是与特定事件相关且包含有关该事件详细信息的对象。事件对象作为参数传递给事件处理程序函数 (不包括IE8及之前版本，在这些浏览器中有时仅能通过全局变量event才能得到)。所有的事件对象都用来指定事件类型的type属性和指定事件目标的target属性。(在IE8及之前的版本中用srcElement而非target)

事件传播 (event propagation) 是浏览器决定哪个对象触发其事件处理程序的过程。当文档元素上发生某个类型的事件时，它们会在文档树上向上传播或“冒泡”(bubble)。

事件传播的另一种方式：事件捕获 (event capturing)：在容器元素上注册的特定处理程序有机会在事件传播到真实目标之前捕获它。

二、Event事件

10.1注册事件处理程序

注册事件处理程序有两种基本方式：

(1) 一种是给事件目标对象或文档元素设置属性。

按照约定，事件处理程序属性的名字由“on”后面跟着事件名组成：onclick、 onchange等。这些属性名是区分大小写的，所有都是 小写，即使是事件类型是由多个词组成的(比如“readystatechange”)。

```
<div onclick="alert(1);"></div>
```

注：尽量少用内联事件

还可以这样：

```
var div = document.querySelector('div');
div.onclick=function(){
```

```
    alert(1);
}
```

如果要删除"on"类型的事件，只需将其设为null：

```
div.onclick = null;
```

再点击就不会有任何反应。

事件处理程序属性的缺点是其设计都是围绕着假设每个事件目标对于每种事件类型将最多只有一个处理程序。

(2) 另一种是通过addEventListener()

window对象、Document对象和所有的文档元素 (Element) 都定义了一个名为 `addEventListener()` 方法，使用这个方法可以为事件目标注册事件处理程序。

```
target.addEventListener(type, listener[, useCapture]);
```

`addEventListener`方法接受三个参数。

- type：事件名称（事件类型），字符串，大小写不敏感。
- listener：监听函数。事件发生时，会调用该监听函数。
- useCapture：布尔值，表示监听函数是否在捕获阶段（capture）触发，默认为false（监听函数只在冒泡阶段被触发）。老式浏览器规定该参数必写，较新版本的浏览器允许该参数可选。为了保持兼容，建议总是写上该参数。

使用`addEventListener()`方法时，事件类型不应包括前缀“on”，比如：“`onclick`”改成“`click`”等。

```
addEventListener('click', listener, false);
```

注意：调用`addEventListener()`并不会影响`onclick`属性的值。

```
<button id="mybutton">点击</button>

var v = document.getElementById('mybutton');
v.onclick = function() {alert('1')};
v.addEventListener('click', function(){alert('2')}, false);
```

上面的代码中，单击按钮会产生两个`alert()`对话框。

能通过多次调用`addEventListener()`方法为同一个对象注册同一事件类型的多个处理程序函数。

所有该事件类型的注册处理程序都会按照注册的顺序调用。使用相同的参数在同一个对象上多次调用

`addEventListener()`是没用的，处理程序仍然只注册一次，同时重复调用也不会改变调用处理程序的顺序（也就

是说，如果为同一个事件多次添加同一个监听函数，函数只会执行一次，多余的添加将自动删除）。

相对 `addEventListener()` 的是 `removeEventListener()` 方法。

`removeEventListener`方法的参数，与`addEventListener`方法完全一致。它的第一个参数“事件类型”，也是大小写不敏感。

注意：`removeEventListener()`方法的事件处理程序函数必须是函数名。

`dispatchEvent()`

`dispatchEvent`方法在当前节点上触发指定事件，从而触发监听函数的执行。该方法返回一个布尔值，只要有一个监听函数调用了`Event.preventDefault()`，则返回值为`false`，否则为`true`。

```
target.dispatchEvent(event)
```

`dispatchEvent`方法的参数是一个`Event`对象的实例。

在IE上

IE支持事件冒泡流，不支持事件捕获流。

IE9以前的IE不支持 `addEventListener()` 和 `removeEventListener()`。不过我们可以使用类似的方法 `attachEvent()` 和 `detachEvent()`；

`attachEvent()` 和 `detachEvent()` 方法的工作原理与`addEventListener()`和`removeEventListener()`类似，但有所区别：

- 因为IE事件模型不支持事件捕获，所以`attachEvent()`和`detachEvent()`只有两个参数：事件类型和处理程序函数
- IE方法的第一个参数使用了带“on”前缀的事件处理程序属性名。例如：当给`addEventListener()`传递“click”时，要给`attachEvent()`传递“onclick”
- `attachEvent()`允许相同的事件处理程序函数注册多次。当特定的事件类型发生时，注册函数的调用次数和注册次数一样。

下面的代码中，创建一个`EventUtil`工具类，可以兼容ie浏览器：

```
var EventUtil = {
    addHandler: function(element, type, handler, useCapture) {
        if(element.addEventListener) {
            element.addEventListener(type, handler, useCapture ? true : false);
        } else if(element.attachEvent) {
            element.attachEvent('on' + type, handler);
        } else if(element != window) {
            element['on' + type] = handler;
        }
    },
    removeHandler: function(element, type, handler, useCapture) {
```

```

        if(element.removeEventListener) {
            element.removeEventListener(type, handler, useCapture ?
true : false);
        } else if(element.detachEvent) {
            element.detachEvent('on' + type, handler);
        } else if(element != window) {
            element['on' + type] = null;
        }
    }
};


```

在上面的EventUtil工具类中，我们创建addHandler()用来绑定事件，removeHandler用来删除事件。

10.2 事件处理程序的调用

一旦注册了事件处理程序，浏览器就会在指定对象上发生指定类型事件时自动调用它。

10.2.1 事件处理程序的参数

通常调用事件处理程序时把 事件对象 (event) 作为它们的一个参数。事件对象的属性提供了有关事件的详细信息。例如，type属性指定了发生的事件类型。

在IE8及以前的版本中，通过设置属性注册事件处理程序，当调用它们时并未传递事件对象。取而代之，需要通过全局对象window.event来获得事件对象。

下面的代码就是考虑了兼容性：

```

function handler(event){
    event = event || window.event;
}

```

10.2.2 event事件对象

(1) DOM中的事件对象

event对象包含与创建它的特定事件有关的属性和方法，触发的事件类型不一样，可用的属性和方法也不一样。

不过，所有事件都有下列属性和方法：

- bubbles 布尔值，只读，表示事件是否冒泡
- cancelable 布尔值，只读，表示是否可以取消事件的默认行为
- currentTarget Element类型，只读，其事件处理程序当前正在处理事件的那个元素
- defaultPrevented 布尔值，只读，为true时表示已经调用了preventDefault()
- detail Integer类型，只读，与事件相关的细节信息
- eventPhase Integer类型，只读，调用事件处理程序的阶段：1表示捕获阶段，2表示目标阶段，3表示冒泡阶段
- preventDefault() Function类型，只读，取消事件的默认行为
- stopImmediatePropagation() Function类型，只读，取消事件的进一步捕获或冒泡，同时组织任何事件处

理程序被调用

- stopPropagation() Function类型，只读，取消事件的进一步捕获或冒泡
- target Element，只读，事件的目标
- trusted 布尔值，只读，为true时表示事件是浏览器生成的，为false时表示事件是开发人员创建的
- type String类型，只读，事件类型
- view AbstractView，只读，与事件关联的抽象视图，等同于发生事件的window对象

在事件处理程序内部，对象this始终等于currentTarget的值，而target则表示实际的目标。

(2) IE中的事件对象

对于IE，event对象是绑定在window对象中的：

```
window.event
```

IE的event对象也同样包含与创建它的事件相关的属性和方法，也具有一些共同属性和方法：

- cancelBubble 布尔值，可读写，默认值为false，将其设置为true时，作用和DOM中的stopPropagation()方法一样
- returnValue 布尔值，可读写，默认为true，将其设为false时，作用和DOM中的preventDefault()方法的作用一样。
- srcElement Element，只读，事件的目标（与DOM中的target属性相同）
- type String，只读，事件类型

基于IE和DOM事件对象不一样，我们可以工具类：EventUtil里添加方法：

```
var EventUtil = {
    addHandler: function(element, type, handler, useCapture) {
        // 省略代码
    },
    removeHandler: function(element, type, handler, useCapture) {
        // 省略代码
    },
    getEvent: function(event){
        return event || window.event;
    },
    getTarget: function(event){
        return event.target || event.srcElement;
    },
    preventDefault: function(event){
        if(event.preventDefault){
            event.preventDefault();
        }else{
            event.returnValue = false;
        }
    }
}
```

```

        }
    },
    stopPropagation: function(event){
        if(event.stopPropagation){
            event.stopPropagation();
        }else{
            event.cancelBubble = true;
        }
    }
);

```

我给工具类EventUtil添加了4个新方法，第一个是getEvent()，返回对event对象的引用；第二个是getTarget()，返回事件的目标；第三个是preventDefault()，用于取消事件的默认行为；第四个是stopPropagation()，用来取消事件冒泡。

调用方式也很简单：

```

div.onclick = function(event){
    event = EventUtil.getEvent(event);
    target = event.getTarget();
    EventUtil.preventDefault();
    EventUtil.stopPropagation();
}

```

10.2.2 事件处理程序的运行环境

当通过设置属性注册事件处理程序时，看起来就好像是在文档元素上定义了新方法：

```
e.onclick=function(){}

```

事件处理程序在事件目标上定义，所以它们作为这个对象的方法来调用。也就是说，在事件处理程序内，this关键字指向事件目标。

10.2.3 事件处理程序的作用域

事件处理程序在其定义的作用域而非调用时的作用域中执行，并且它们能存取那个作用域中的任何一个本地变量。

10.2.4 事件处理程序的返回值

通过设置对象属性或HTML属性注册事件处理程序的返回值有时是非常有意义的。通常情况下，返回值false就是告诉浏览器不要执行这个事件相关的默认操作。比如，表单提交按钮的onclick事件处理程序能返回false阻止浏览器提交表单。

```

v.onclick = function() {
    return false;
}

```

理解事件处理程序的返回值只对通过属性注册的处理程序才有意义。

10.2.5 调用顺序

文档元素或其他对象可以指定事件类型注册多个事件处理程序。当适当的事件发生时，浏览器必须按照下面的规则调用所有的事件处理程序：

- 通过设置对象属性或HTML属性注册的处理程序一直优先调用。
- 使用addEventListener()注册的处理程序按照它们的注册顺序调用。
- 使用attachEvent()注册的处理程序可能按照任何顺序调用，所以代码不应该依赖于调用顺序。

10.2.6 事件传播

当事件目标是Window对象或其他一些单独对象（比如XMLHttpRequest）时，浏览器会简单的通过调用对象上适当的处理程序响应事件。

在调用在目标元素上注册的事件处理函数后，大部分事件会“冒泡”到DOM树根。

发生在文档元素上的大部分事件都会冒泡，但有些例外，比如focus、blur和scroll事件。文档元素上的load事件会冒泡，但它会在Document对象上停止冒泡而不会传播到Window对象。只有当整个文档都加载完毕时才会触发window对象的load事件。

当事件目标是文档或文档元素时，它会在不同的DOM节点之间传播（propagation）。

分为三个阶段：

- 捕获阶段（capture phase）：从window对象传导到目标对象。（window--document--……--目标对象）
- 目标阶段（target phase）：目标对象本身的事件处理程序调用。
- 冒泡阶段（bubbling phase）：从目标对象传导回window对象。（目标对象--父元素--……--document--window）

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="myDiv">点击</div>
</body>
</html>
```

```
//事件捕获阶段，click事件的传播顺序
window
document
<html>
<body>
<div>
```

```
// 目标阶段
div

// 事件冒泡阶段, click事件的传播顺序
<div>
<body>
<html>
document
window
```

事件代理（事件委托）

基于事件会在冒泡阶段向上传播到父节点，我们可以将子节点的监听事件定义在父节点上，由父节点的监听函数统一处理多个子元素的事件。这种方法叫做事件的代理（delegation）。

```
<div id="div">
  <div id="item">123</div>
</div>

document.getElementById('div').addEventListener('click', function(e) {
  var target = e.target;
  if(target.getAttribute('id').toLowerCase() == 'item') {
    alert(1);
  }
});
```

如果使用事件代理，以后插入的新节点仍然可以监听的到。

如果使用JQuery，我们要为新增节点添加事件，除了在新增事件后添加事件外，还可以用下面的代码：

```
$(document).on('click', 'div', function(){})
```

这种方式其实就是使用了事件代理。

10.2.7 事件取消

用属性注册的世界处理程序的返回值能用于取消事件的浏览器默认操作。在支持addEventListener()的浏览器中，也能通过调用事件对象的preventDefault()方法取消事件的默认操作。

在IE9之前的IE中，可以通过设置事件对象的returnValue属性为false来达到同样的效果。

```
function cancelHandler(event){
  var event = event || window.event;
  if(event.preventDefault) {event.preventDefault();} // 标准
  if(event.returnValue) { event.returnValue = false;} // IE
  return false; // 用于处理使用对象属性注册的处理程序
}
```

Event对象提供了一个属性defaultPrevented，返回一个布尔值，默认false，表示该事件是否调用过preventDefault方法。

取消事件传播

在支持addEventListener()的浏览器中，可以调用事件对象的一个stopPropagation()方法以阻止事件的继续传播。

```
e.stopPropagation()
//IE
e.cancelBubble = true;
```

在Event对象上还有一个方法 stopImmediatePropagation()，阻止同一个事件的其他监听函数被调用。也就是说，如果同一个节点对于同一个事件指定了多个监听函数，这些函数会根据添加的顺序依次调用。只要其中有一个监听函数调用了stopImmediatePropagation方法，其他的监听函数就不会再执行了。

```
e.addEventListener('click', function(event){
    event.stopImmediatePropagation();
});
e.addEventListener('click', function(event){
    //不会触发
});
```

10.3 文档事件

(1) beforeunload事件、unload事件、load事件、error事件、pageshow事件、pagehide事件

beforeunload

当浏览器将要跳转到新页面时触发这个事件。如果事件处理程序返回一个字符串，那么它将出现在询问用户是否想离开当前页面的标准对话框中。

```
window.addEventListener('beforeunload', function(e){
    var message = '你确认要离开吗！';
    e.returnValue = message;
    return message
});
```

unload

unload事件在窗口关闭或者document对象将要卸载时触发，发生在window、body、frameset等对象上面。

它的触发顺序排在beforeunload、pagehide事件后面。unload事件只在页面没有被浏览器缓存时才会触发，换言之，如果通过按下“前进/后退”导致页面卸载，并不会触发unload事件。

load、error

load 事件直到文档加载完毕时（包括所有图像、JavaScript文件、CSS文件等外部资源）才会触发。

error 事件在页面加载失败时触发。注意，页面从浏览器缓存加载，并不会触发load事件。
这两个事件实际上属于进度事件，不仅发生在document对象，还发生在各种外部资源上面。浏览网页就是一个加载各种资源的过程，图像（image）、样式表（style sheet）、脚本（script）、视频（video）、音频（audio）、Ajax请求（XMLHttpRequest）等等。这些资源和document对象、window对象、XMLHttpRequestUpload对象，都会触发load事件和error事件。

pageshow、pagehide

默认情况下，浏览器会在当前会话（session）缓存页面，当用户点击“前进/后退”按钮时，浏览器就会从缓存中加载页面。

pageshow事件在页面加载时触发，包括第一次加载和从缓存加载两种情况。如果要指定页面每次加载（不管是从浏览器缓存）时都运行的代码，可以放在这个事件的监听函数。

pageshow事件有一个persisted属性，返回一个布尔值。页面第一次加载时，这个属性是false；当页面从缓存加载时，这个属性是true。

pagehide事件与pageshow事件类似，当用户通过“前进/后退”按钮，离开当前页面时触发。

pagehide事件的event对象有一个persisted属性，将这个属性设为true，就表示页面要保存在缓存中；设为false，表示网页不保存在缓存中，这时如果设置了unload事件的监听函数，该函数将在pagehide事件后立即运行。

(2) DOMContentLoaded事件、readystatechange事件

DOMContentLoaded 事件：当文档加载解析完毕且所有延迟（deferred）脚本（图片未加载完毕）都执行完毕时会触发，此时图片和异步（async）脚本可能依旧在加载，但是文档已经为操作准备就绪了。也就是说，这个事件，发生在load事件之前。

```
document.addEventListener('DOMContentLoaded', handler, false);
```

readystatechange 事件：document.readyState属性会随着文档加载过程而变，而每次状态改变，Document对象上的readystatechange事件都会触发。

```
document.onreadystatechange = function() {
  if(document.readyState == 'complete'){
    }
}
```

(3) scroll事件、resize事件

scroll 事件在文档（window）或文档元素滚动时触发，主要出现在用户拖动滚动条。

resize 事件在改变浏览器窗口大小时触发，发生在window、body、frameset对象上面。

(4) hashchange事件、popstate事件

hashchange事件在URL的hash部分（即#号后面的部分，包括#号）发生变化时触发。如果老式浏览器不支持该

属性，可以通过定期检查location.hash属性，模拟该事件。

popstate事件在浏览器的history对象的当前记录发生显式切换时触发。注意，调用history.pushState()或history.replaceState()，并不会触发popstate事件。该事件只在用户在history记录之间显式切换时触发，比如鼠标点击“后退/前进”按钮，或者在脚本中调用history.back()、history.forward()、history.go()时触发。

(5) cut事件、copy事件、paste事件

这三个事件属于文本操作触发的事件。

- cut事件：在将选中的内容从文档中移除，加入剪贴板后触发。
- copy事件：在选中的内容加入剪贴板后触发。
- paste事件：在剪贴板内容被粘贴到文档后触发。

这三个事件都有一个clipboardData只读属性。该属性存放剪贴的数据，是一个DataTransfer对象。

(6) 焦点事件

焦点事件发生在Element节点和document对象上。

- focus事件：Element节点获得焦点后触发，该事件不会冒泡。
- blur事件：Element节点失去焦点后触发，该事件不会冒泡。
- focusin事件：Element节点将要获得焦点时触发，发生在focus事件之前。该事件会冒泡。Firefox不支持该事件。
- focusout事件：Element节点将要失去焦点时触发，发生在blur事件之前。该事件会冒泡。Firefox不支持该事件。

这四个事件的事件对象，带有target属性（返回事件的目标节点）和relatedTarget属性（返回一个Element节点）。对于focusin事件，relatedTarget属性表示失去焦点的节点；对于focusout事件，表示将要接受焦点的节点；对于focus和blur事件，该属性返回null。

由于focus和blur事件不会冒泡，只能在捕获阶段触发，所以addEventListener方法的第三个参数需要设为true。

10.4 鼠标事件

(1) click

click事件当用户在Element节点、document节点、window对象上，单击鼠标（或者按下回车键）时触发。

“鼠标单击”定义为，用户在同一个位置完成一次mousedown动作和mouseup动作。它们的触发顺序是：mousedown首先触发，mouseup接着触发，click最后触发。

(2) contextmenu

contextmenu事件在一个节点上点击鼠标右键时触发，或者按下“上下文菜单”键时触发。

可以通过下面的方式阻止“上下文菜单”的出现：

```
document.oncontextmenu=function(){
    return false;
```

}

(3) dblclick

dblclick事件当用户在element、document、window对象上，双击鼠标时触发。该事件会在mousedown、mouseup、click之后触发。

(4) mousedown、mouseup

mouseup事件在释放按下的鼠标键时触发。

mousedown事件在按下鼠标键时触发。

(5) mousemove

mousemove事件当鼠标在一个节点内部移动时触发。当鼠标持续移动时，该事件会连续触发。为了避免性能问题，建议对该事件的监听函数做一些限定，比如限定一段时间内只能运行一次代码。

(6) mouseover、mouseenter

mouseover事件和mouseenter事件，都是鼠标进入一个节点时触发。

两者的区别是，mouseenter事件只触发一次，而只要鼠标在节点内部移动，mouseover事件会在子节点上触发多次。

(7) mouseout、mouseleave

mouseout事件和mouseleave事件，都是鼠标离开一个节点时触发。

除了“mouseenter”和“mouseleave”外的所有鼠标事件都能冒泡。链接和提交按钮上的click事件都有默认操作且能够阻止。可以取消上下文菜单事件来阻止显示上下文菜单。

传递给鼠标事件处理程序的事件对象有clientX和clientY属性，它们指定了鼠标指针相对于包含窗口的坐标。加入窗口的滚动偏移量可以把鼠标位置转换成文档坐标。

MouseEvent对象的属性

(1) button、buttons

button 属性指定当事件发生时哪个鼠标按键按下。

- -1：没有按下键。
- 0：按下主键（通常是左键）。
- 1：按下辅助键（通常是中键或者滚轮键）。
- 2：按下次键（通常是右键）。

buttons属性返回一个3个比特位的值，表示同时按下了哪些键。它用来处理同时按下多个鼠标键的情况。

- 1：二进制为001，表示按下左键。
- 2：二进制为010，表示按下右键。
- 4：二进制为100，表示按下中键或滚轮键。

同时按下多个键的时候，每个按下的键对应的比特位都会有值。比如，同时按下左键和右键，会返回3（二进制为011）。

注意：IE中的button属性拥有不同的参数：

- 1：鼠标左键
- 4：鼠标中键
- 2：鼠标右键

(2) clientX , clientY

`clientX` 属性返回鼠标位置相对于浏览器窗口左上角的水平坐标，单位为像素，与页面是否横向滚动无关。

`clientY` 属性返回鼠标位置相对于浏览器窗口左上角的垂直坐标，单位为像素，与页面是否纵向滚动无关。

(3) movementX , movementY

- `movementX` 属性返回一个水平位移，单位为像素，表示当前位置与上一个`mousemove`事件之间的水平距离。在数值上，等于`currentEvent.movementX = currentEvent.screenX - previousEvent.screenX`。
- `movementY` 属性返回一个垂直位移，单位为像素，表示当前位置与上一个`mousemove`事件之间的垂直距离。在数值上，等于`currentEvent.movementY = currentEvent.screenY - previousEvent.screenY`。

(4) screenX , screenY

`screenX` 属性返回鼠标位置相对于屏幕左上角的水平坐标，单位为像素。

`screenY` 属性返回鼠标位置相对于屏幕左上角的垂直坐标，单位为像素。

(7) pageX、pageY

`pageX` 和 `pageY` 分别是触点相对HTML文档左边沿的X坐标和触点相对HTML文档上边沿的Y坐标。只读属性。

当存在滚动的偏移时，`pageX`包含了水平滚动的偏移，`pageY`包含了垂直滚动的偏移。

(6) relatedTarget

`relatedTarget` 属性返回事件的次要相关节点。对于那些没有次要相关节点的事件，该属性返回`null`。

10.5 鼠标滚轮事件

所有的现代浏览器都支持鼠标滚轮，并在用户滚动滚轮时触发事件。浏览器通常使用鼠标滚轮滚动或缩放文档，但可以通过取消`mousewheel`事件来阻止这些默认操作。

所有浏览器都支持“`mousewheel`”事件，但Firefox使用“`DOMMouseScroll`”事件。

传递给“`mousewheel`”处理程序的事件对象有`wheelDelta`属性，其指定用户滚动滚轮有多远（根据这个判断滚动方向）。

远离用户方向的一次鼠标滚轮“单击”的`wheelDelta`值通常是120，而接近用户方向的一次“单击”的值是-120。返回的总是120的倍数（120表明mouse向上滚动，-120表明鼠标向下滚动）

在Safari和Chrome中，为了支持使用二维轨迹球而非一维滚轮的Apple鼠标，除了`wheelDelta`属性外，事件对象还有`wheelDeltaX`和`wheelDeltaY`，而`wheelDelta`和`wheelDeltaY`的值一直相同。

而在Firefox中，传递给“`DOMMouseScroll`”的属性是`detail`。不过，`detail`属性值的缩放比率和正负符号不同`wheelDelta`，`detail`值乘以-40和`wheelDelta`值相等。记录其滚动距离的是“`detail`”属性，它返回的是3的倍数

(3表明mouse向下滚动 , - 3表明mouse向上滚动)。

```
window.onmousewheel = document.onmousewheel = scrollwheel;

function scrollWheel(e){
    e = e || window.event;
    if(e.wheelDelta) { //判断浏览器IE, 谷歌滑轮事件
        if(e.wheelDelta > 0) {
            //当滑轮向上滚动时
        } else if(e.wheelDelta < 0) {
            //当滑轮向下滚动时
        };
    } else if(e.detail) { //Firefox滑轮事件
        if(e.detail < 0) {
            //当滑轮向上滚动时
        } else if(e.detail > 0) {
            //当滑轮向下滚动时
        };
    };
}
```

10.6 键盘事件

键盘事件用来描述键盘行为，主要有keydown、keypress、keyup三个事件。

keydown：按下键盘时触发该事件。

keypress：只要按下的键并非Ctrl、Alt、Shift和Meta，就接着触发keypress事件。

keyup：松开键盘时触发该事件。

textinput

任何时候，只要用户输入文本都会触发。在Webkit浏览器中支持“textInput”事件。

事件对象属性data（保存输入文本），inputMethod属性（用于指定输入源）

注意：keypress和textinput事件是在新输入的文本真正插入到聚焦的文档元素前触发的。

如果用户一直按键不放开，就会重复触发keydown、keypress，直到用户松开才会触发keyup。

属性

keycode

指定了输入字符的编码。在Firefox中使用的是charCode属性。

altKey, ctrlKey, metaKey, shiftKey

altKey、ctrlKey、metaKey和shiftKey属性指定了当事件发生时是否有各种键盘辅助键按下。

altKey属性：alt键

ctrlKey属性：key键

metaKey属性：Meta键（Mac键盘是一个四瓣的小花，Windows键盘是Windows键）

shiftKey属性：Shift键

key , charCode key属性返回一个字符串，表示按下的键名。如果同时按下一个控制键和一个符号键，则返回符号键的键名。比如，按下Ctrl+a，则返回a。如果无法识别键名，则返回字符串Unidentified。

主要功能键的键名（不同的浏览器可能有差异）：Backspace , Tab , Enter , Shift , Control , Alt , CapsLock , CapsLock , Esc , Spacebar , PageUp , PageDown , End , Home , Left , Right , Up , Down , PrintScreen , Insert , Del , Win , F1 ~ F12 , NumLock , Scroll等。

charCode属性返回一个数值，表示keypress事件按键的Unicode值，keydown和keyup事件不提供这个属性。注意，该属性已经从标准移除，虽然浏览器还支持，但应该尽量不使用。

String.fromCharCode()

一个keypress事件表示输入的单个字符。事件对象以数字Unicode编码的形式指定字符，所以必须用String.fromCharCode()把它转换成字符串。

10.7 表单事件

(1) input、propertychange

检测文本输入元素的value属性改变，这两个事件是在新输入的文本真正插入到聚焦的文档元素前触发的。

一般用 `<input>` 和 `<textarea>` 里，不过，当将contenteditable属性设置为true时，只要值变化，也会触发这两个事件。

(2) change

当 `<input>` 、`<select>` 和 `<textarea>` 的值发生变化时都会触发change事件。只有全部修改完成时它才会触发，这也是它和input事件的区别。

具体分下面几种情况：

- 激活单选框（radio）或复选框（checkbox）时触发。
- 用户提交时触发。比如，从下列列表（select）完成选择，在日期或文件输入框完成选择。
- 当文本框或textarea元素的值发生改变，并且丧失焦点时触发。

(3) select

当 `<input>` 和 `<textarea>` 中选中文本时触发select事件。

(4) reset、submit

这两个事件是发生在表单对象上，而不是发生在表单的成员上。

reset事件：当表单重置（所有表单成员的值变回默认值）时触发。

submit事件：当表单数据向服务器提交时触发。

注意：submit事件的发生对象是form元素，而不是button元素（即使它的类型是submit），因为提交的是表单，而不是按钮。

10.8 触控事件

触控事件提供了响应用户对触摸屏或触摸板上操作的能力。

触控API提供了下面三个接口

- TouchEvent：代表当触摸行为在平面上发生变化时发生的事件
- Touch：代表用户与触摸屏幕间的一个接触点
- TouchList：代表一系列的Touch；一般在用户多个手指同时解除屏幕时使用

10.8.1 TouchEvent

TouchEvent是一类描述手指在触摸平面的状态变化的事件。

(1) 触摸事件的类型

- touchstart：用户接触触摸屏时触发，它的target属性返回发生触摸的Element节点。
- touchend：用户不再接触触摸屏时（或者移出屏幕边缘时）触发，它的target属性与touchstart事件的target属性是一致的，它的changedTouches属性返回一个TouchList对象，包含所有不再触摸的触摸点（Touch对象）。
- touchmove：用户移动触摸点时触发，它的target属性与touchstart事件的target属性一致。如果触摸的半径、角度、力度发生变化，也会触发该事件。
- touchcancel：触摸点取消时触发，比如在触摸区域跳出一个弹出框窗口（modal window）、触摸点离开了文档区域（进入浏览器菜单栏区域）、用户放置更多的触摸点（自动取消早先的触摸点）。
- touchenter：当触点进入某个element时触发。没有冒泡过程。
- touchleave：当触点离开某个element时触发。没有冒泡过程。

(2) TouchEvent的属性

键盘属性

以下属性都为只读属性，返回一个布尔值，表示触摸的同时，是否按下某个键。

altKey 是否按下alt键

ctrlKey 是否按下ctrl键

metaKey 是否按下meta键

shiftKey 是否按下shift键

changedTouches

返回一个TouchList对象，包含了代表所有从上一次触摸事件到此次事件过程中，状态发生了改变的触点的Touch对象。只读属性。

targetTouches

返回一个TouchList对象，包含了所有当前接触触摸平面的触点的Touch对象（也可以说是处于活动状态的触点的Touch对象）。只读属性。

touches

返回一个TouchList对象，包含了所有当前接触触摸平面的触点的Touch对象。只读属性。

type

此次触摸事件的类型。

target

此次触摸事件的目标元素 (element)。这个目标元素对应 TouchEvent.changedTouches 中的触点的起始元素。

10.8.2 Touch

Touch对象表示在触控设备上的触控点。通常是指手指或触控笔在触屏设备上的操作。

每个Touch对象代表一个触点，每个触点由其位置、大小、形状、压力大小和目标element描述。

(1) Touch属性

以下属性描述了用户的触摸行为

identifier

此Touch对象的唯一标识符。一次触摸动作在平面上移动的整个过程中，该标识符不变，可以根据它来判断跟踪是否在同一次触摸过程。只读属性。

screenX、screenY

screenX和screenY分别是触点相对屏幕左边沿的X坐标和触点相对屏幕上边沿的Y坐标。只读属性。

clientX、clientY

clientX和clientY分别是触点相对于可视区左边沿的X坐标和触点相对可视区上边沿的Y坐标。两个属性都不包括任何滚动偏移。只读属性。

pageX、pageY

pageX和pageY分别是触点相对HTML文档左边沿的X坐标和触点相对HTML文档上边沿的Y坐标。只读属性。

当存在滚动的偏移时，pageX包含了水平滚动的偏移，pageY包含了垂直滚动的偏移。

radiusX、radiusY、rotationAngle

radiusX：能够包围用户和触摸平面的接触面的最小椭圆的水平轴(X轴)半径。这个值的单位和 screenX 相同。只读属性。

radiusY：能够包围用户和触摸平面的接触面的最小椭圆的垂直轴(Y轴)半径。这个值的单位和 screenY 相同。只读属性。

rotationAngle：由radiusX 和 radiusY 描述的正方向的椭圆，需要通过顺时针旋转这个角度值，才能最精确地覆盖住用户和触摸平面的接触面，单位为度数，在0到90度之间。只读属性。

force

手指挤压触摸平面的压力大小，从0.0(没有压力)到1.0(最大压力)的浮点数. 只读属性.

target

当这个触点最开始被跟踪时(在 touchstart 事件中)，触点位于的HTML元素。也就是触摸发生时的那个节点。

10.8.3 TouchList

一个TouchList代表一个触摸平面上所有触点的列表。比如一个用户用三根手指接触平面，与之相关的TouchList对于每根手指都会生成一个Touch对象，共计三个。

(1) TouchList的属性

length

返回TouchList中Touch对象的数量，只读属性。

(2) 方法

`identifiedTouch()`

返回列表中标识符与指定值匹配的第一个Touch对象。

`item()`

返回列表中以指定索引值的Touch对象。也可以使用数组的语法：`touchlist[index]`

10.8.4 其他触控事件

`gesturestart`、`gestureend`

`scale`、`rotation`

10.9 进度事件

进度事件用来描述一个事件进展的过程。比如XMLHttpRequest对象发出的HTTP请求的过程。

``、`<audio>`、`<video>`、`<style>`、`<link>` 加载外部资源的过程。下载和上传都会发生进度事件。

进度事件有以下几种：

- `abort`事件：当进度事件被中止时触发。如果发生错误，导致进程中止，不会触发该事件。
- `error`事件：由于错误导致资源无法加载时触发。
- `load`事件：进度成功结束时触发。
- `loadstart`事件：进度开始时触发。
- `loadend`事件：进度停止时触发，发生顺序排在`error`事件\`abort`事件\`load`事件后面。
- `progress`事件：当操作处于进度之中，由传输的数据块不断触发。
- `timeout`事件：进度超过限时触发。

10.11 拖放事件

拖放 (Drag-and-Drop , DnD) 是在 “拖放源 (drag source) ” 和 “拖放目标 (drop target) ” 之间传输数据的用户界面。

拖拉的对象有好几种，包括Element节点、图片、链接、选中的文字等等。在HTML网页中，除了Element节点默认不可以拖拉，其他（图片、链接、选中的文字）都是可以直接拖拉的。为了让Element节点可拖拉，可以将该节点的`draggable`属性设为`true`。

```
<div draggable="true"> 此区域可拖拉 </div>
```

`draggable`属性可用于任何Element节点，但是图片（`img`元素）和链接（`a`元素）不加这个属性，就可以拖拉。对于它们，用到这个属性的时候，往往是将其设为`false`，防止拖拉。

注意：一旦某个Element节点的`draggable`属性设为`true`，就无法再用鼠标选中该节点内部的文字或子节点了。

10.11.1 拖放事件

- `dragstart`：当一个元素开始被拖拽的时候触发。用户拖拽的元素需要附加`dragstart`事件。在这个事件中，

监听器将设置与这次拖拽相关的信息，例如拖动的数据和图像。

- dragenter：当拖拽中的鼠标第一次进入一个元素的时候触发。这个事件的监听器需要指明是否允许在这个区域释放鼠标。如果没有设置监听器，或者监听器没有进行操作，则默认不允许释放。当你想要通过类似高亮或插入标记等方式来告知用户此处可以释放，你将需要监听这个事件。
- dragover：当拖拽中的鼠标移动经过一个元素的时候触发。大多数时候，监听过程发生的操作与dragenter事件是一样的。
- dragleave：当拖拽中的鼠标离开元素时触发。监听器需要将作为可释放反馈的高亮或插入标记去除。
- drag：这个事件在拖拽源触发。即在拖拽操作中触发dragstart事件的元素。
- drop：这个事件在拖拽操作结束释放时于释放元素上触发。一个监听器用来响应接收被拖拽的数据并插入到释放之地。这个事件只有在需要时才触发。当用户取消了拖拽操作时将不触发，例如按下了Escape (ESC) 按键，或鼠标在非可释放目标上释放了按键。
- dragend：拖拽源在拖拽操作结束将得到dragend事件对象，不管操作成功与否。

注意点：

拖拉过程只触发以上这些拖拉事件，尽管鼠标在移动，但是鼠标事件不会触发。

将文件从操作系统拖拉进浏览器，不会触发dragStart和dragend事件。

dragenter和dragover事件的监听函数，用来指定可以放下（drop）拖拉的数据。由于网页的大部分区域不适合作为drop的目标节点，所以这两个事件的默认设置为当前节点不允许drop。如果想要在目标节点上drop拖拉的数据，首先必须阻止这两个事件的默认行为，或者取消这两个事件。

```

<div ondragover="return false">
<div ondragover="event.preventDefault()">
下面是一个例子，将图片拖放到另一个div中：
<div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>


function allowDrop(ev)
{
  ev.preventDefault();
}
function drag(ev)
{
  ev.dataTransfer.setData("Text",ev.target.id);
}
function drop(ev)
{
  ev.preventDefault();
  var data=ev.dataTransfer.getData("Text");
  ev.target.appendChild(document.getElementById(data));
}

```

10.11.2 DataTransfer对象

所有的拖拉事件都有一个dataTransfer属性，用来保存需要传递的数据。返回一个DataTransfer对象。

拖拉的数据保存两方面的数据：数据的种类（又称格式）和数据的值。数据的种类是一个MIME字符串，比如text/plain或者image/jpeg，数据的值是一个字符串。一般来说，如果拖拉一段文本，则数据默认就是那段文本；如果拖拉一个链接，则数据默认就是链接的URL。

当拖拉事件开始的时候，可以提供数据类型和数据值；在拖拉过程中，通过dragenter和dragover事件的监听函数，检查数据类型，以确定是否允许放下（drop）被拖拉的对象。比如，在只允许放下链接的区域，检查拖拉的数据类型是否为text/uri-list。

发生drop事件时，监听函数取出拖拉的数据，对其进行处理。

(1) DataTransfer对象的属性

dropEffect

dropEffect属性设置放下（drop）被拖拉节点时的效果，可能的值包括copy（复制被拖拉的节点）、move（移动被拖拉的节点）、link（创建指向被拖拉的节点的链接）、none（无法放下被拖拉的节点）。设置除此以外的值，都是无效的。

```
e.dataTransfer.dropEffect = 'copy';
```

dropEffect属性一般在dragenter和dragover事件的监听函数中设置，对于dragstart、drag、dragleave这三个事件，该属性不起作用。

effectAllowed

effectAllowed属性设置本次拖拉中允许的效果，可能的值包括copy（复制被拖拉的节点）、move（移动被拖拉的节点）、link（创建指向被拖拉节点的链接）、copyLink（允许copy或link）、copyMove（允许copy或move）、linkMove（允许link或move）、all（允许所有效果）、none（无法放下被拖拉的节点）、uninitialized（默认值，等同于all）。如果某种效果是不允许的，用户就无法在目标节点中达成这种效果。

dragstart事件的监听函数，可以设置被拖拉节点允许的效果；dragenter和dragover事件的监听函数，可以设置目标节点允许的效果。

```
e.dataTransfer.effectAllowed = 'copy';
```

**files **

files属性是一个FileList对象，包含一组本地文件，可以用来在拖拉操作中传送。如果本次拖拉不涉及文件，则属性为空的FileList对象。

types

types属性是一个数组，保存每一次拖拉的数据格式，比如拖拉文件，则格式信息就为File。

(2) DataTransfer对象的方法

setData()

setData方法用来设置事件所带有的指定类型的数据。它接受两个参数，第一个是数据类型，第二个是具体数据。如果指定的类型在现有数据中不存在，则该类型将写入types属性；如果已经存在，在该类型的现有数据将被替换。

```
e.dataTransfer.setData('text/plain','bb');
```

getData()

getData方法接受一个字符串（表示数据类型）作为参数，返回事件所带的指定类型的数据（通常是用setData方法添加的数据）。如果指定类型的数据不存在，则返回空字符串。通常只有drop事件触发后，才能取出数据。如果取出另一个域名存放的数据，将会报错。

clearData()

clearData方法接受一个字符串（表示数据类型）作为参数，删除事件所带的指定类型的数据。如果没有指定类型，则删除所有数据。如果指定类型不存在，则原数据不受影响。

```
e.dataTransfer.clearData('text/plain');
```

setDragImage()

拖动过程中（dragstart事件触发后），浏览器会显示一张图片跟随鼠标一起移动，表示被拖动的节点。这张图片是自动创造的，通常显示为被拖动节点的外观，不需要自己动手设置。setDragImage方法可以用来自定义这张图片，它接受三个参数，第一个是img图片元素或者canvas元素，如果省略或为null则使用被拖动的节点的外观，第二个和第三个参数为鼠标相对于该图片左上角的横坐标和右坐标。

10.12 模拟事件

模拟事件要经过三步：

- 创建event对象
- 初始化
- 使用dispatchEvent()方法触发事件

(1) document.createEvent()

document.createEvent方法用来新建指定类型的事件。它所生成的Event实例，可以传入dispatchEvent方法。createEvent方法接受一个字符串作为参数，表示要创建的事件类型的字符串，可能值是：

- UIEvents：一般化的UI事件（文档事件）。鼠标事件和键盘事件都继承自UI事件。DOM3级中是UIEvent
- MouseEvents：一般化的鼠标事件。DOM3中是MouseEvent
- MutationEvents：一般化的DOM变动事件。DOM3中是MutationEvent
- HTMLEvents：一般化的HTML事件。

(2) event.initEvent()

事件对象的initEvent方法，用来初始化事件对象，还能向事件对象添加属性。该方法的参数必须是一个使用Document.createEvent()生成的Event实例，而且必须在dispatchEvent方法之前调用。initEvent方法可以接受四个参数。

- type：事件名称，格式为字符串。

- bubbles : 事件是否应该冒泡，格式为布尔值。可以使用event.bubbles属性读取它的值。
- cancelable : 事件是否能被取消，格式为布尔值。可以使用event.cancelable属性读取它的值。
- option : 为事件对象指定额外的属性。

```
var event = document.createEvent('MouseEvent');
event.initEvent('click', true, false);
div.dispatchEvent(event);
```

也可以使用相应的构造函数来创建event

```
var event = new MouseEvent('click', {
  'bubbles': true,
  'cancelable': true
});
div.dispatchEvent(event);
```

10.13 自定义事件

我们可以使用自定义事件

```
//新建事件实例
var event = new Event('play');
//添加监听函数
element.addEventListener('play', funciton(e){}, false);
//触发事件
element.dispatchEvent(event);
```

CustomEvent()

Event构造函数只能指定事件名，不能在事件上绑定数据。如果需要在触发事件的同时，传入指定的数据，需要使用CustomEvent构造函数生成自定义的事件对象。

```
var event = new CustomEvent('play', {detail: 'play'});
//添加监听函数
element.addEventListener('play', handler, false);
//触发事件
element.dispatchEvent(event);
```

CustomEvent构造函数的第一个参数是事件名称，第二个参数是一个对象。在上面的代码中，该对象的detail属性会绑定在事件对象之上。

```
funciton handler(e){
  var data = e.detail;
```

}

在IE上，并不支持上面的自定义事件写法，不过，我们可以采用老式写法：

```
// 新建Event实例
var event = document.createEvent('Event');
// 事件的初始化
event.initEvent('play', true, true);
// 加上监听函数
document.addEventListener('play', handler, false);
// 触发事件
document.dispatchEvent(event);
```

正则表达式

正则表达式

正则表达式 (regular expression) 是一个描述字符模式的对象。JavaScript的RegExp类表示正则表达式，String和RegExp都定义了方法。

1.1 正则表达式的定义

正则表达式有两种方法定义：

- 一种是使用正则表达式直接量，将其包含在一对斜杠 (/) 之间的字符。

```
var pattern = /s$/;
```

- 另一种是使用RegExp()构造函数。

```
var pattern = new RegExp('s');
```

上面两种方法是等价的，用来匹配所有以字母 “s” 结尾的字符串。

正则表达式的模式规则是由一个字符序列组成的，所有字母和数字都是按照字面含义进行匹配的。

1.1.1 直接量字符

JavaScript正则表达式语法也支持非字母的字符匹配，这些字符需要通过反斜杠 (\) 作为前缀进行转义。

有如下直接量字符：

```
\0 匹配null字符 (\u0000)。
[\b] 匹配退格键(\u0008)，不要与\b混淆。
\t 匹配制表符tab (\u0009)。
\n 匹配换行键(\u000A)。
\v 匹配垂直制表符 (\u000B)。
\f 匹配换页符 (\u000C)。
\r 匹配回车键(\u000D)。
\xnn 匹配一个以两位十六进制数 (\x00-\xFF) 表示的字符。
\uxxx 匹配一个以四位十六进制数 (\u0000-\uFFFF) 表示的unicode字符。
\cX 表示Ctrl-[X]，其中的X是A-Z之中任一个英文字母，用来匹配控制字符。
```

1.1.2 字符类

将直接量字符单独放进方括号就组成了字符类 (character class)。一个字符类可以匹配它所包含的任意字符。

比如：/[abc]/就和字母 “a” 、 “b” 、 “c” 中的任意一个都匹配。

我们还可以通过 “^” 符号来定义否定字符类，它匹配所有不包含括号内的字符。定义否定字符类时，将一个 “^” 符号作为左方括号内的第一个字符。比如：/[^abc]/ 匹配的是 “a” 、 “b” 、 “c” 之外的所有字

符。‘

字符类还可以使用连字符来表示字符范围。比如要匹配拉丁字母表中的小写字母，可以使用 `/[a-z]/`，要匹配拉丁字母表中的任何字母和数字，则使用 `/[a-zA-Z0-9]/`。

注意：字符类的连字符必须在头尾两个字符中间，才有特殊含义，否则就是字面含义。比如，`[-9]`就表示匹配连字符和9，而不是匹配0到9。

正则表达式的字符类：

<code>[. . .]</code>	方括号内的任意字符
<code>[^ . . .]</code>	不在方括号内的任意字符
<code>.</code>	除换行符和其他Unicode行终止符之外的任意字符
<code>\w</code>	任何ASCII字符组成的单词，等价于 <code>[a-zA-Z0-9]</code>
<code>\W</code>	任何不适ASCII字符组成的单词，等价于 <code>[^a-zA-Z0-9]</code>
<code>\s</code>	任何Unicode空白符
<code>\S</code>	任何非Unicode空白符的字符
<code>\d</code>	任何非ASCII数字，等价于 <code>[0-9]</code>
<code>\D</code>	除了ASCII数字之外的任何字符，等价于 <code>[^0-9]</code>
<code>[\b]</code>	退格直接量

注意：在方括号之内也可以写上面这些特殊转义字符。但有一个特例：当转义符 `\b` 用在字符类中时，它表示的是退格符，所以要在正则表达式中按照直接量表示一个退格符时，只需使用 `/[\b]/` 即可。

1.1.3 重复

在正则模式之后跟随用以指定字符重复的标记。

正则表达式的重复字符语法：

<code>{n, m}</code>	匹配前一项至少n次，至多m次
<code>{n, }</code>	匹配前一项n次或者更多次，也可以说至少n次
<code>{n}</code>	匹配前一项n次
<code>?</code>	匹配前一项0次或者1次，等价于 <code>{0, 1}</code>
<code>+</code>	匹配前一项1次或多次，等价于 <code>{1, }</code>
<code>*</code>	匹配前一项0次或多次，等价于 <code>{0, }</code>

注意：在使用 “” 和 “?” 时，由于这些字符可能匹配0个字符，因此它们允许什么都不匹配。比如：正则表达式/a/实际上与字符串 “bbb” 匹配，因为这个字符串含有0个a。

1.1.4 非贪婪的重复

默认情况下，匹配重复字符是尽可能多的匹配，而且允许后续的正则表达式继续匹配，即匹配直到下一个字符不满足匹配规则为止，这称为“贪婪的”匹配。

当然，我们也可以使用正则表达式进行非贪婪匹配，一旦条件满足，就不再往下匹配。只须在待匹配的字符后跟随一个问号即可：“??”、“+?”、“*?”或“{1,5}?”。

```
/a+/ .exec('aaa') //["aaa"]
```

```
/a+?/.exec('aaa') //["a"]
```

1.1.5 选择、分组和引用

正则表达式的语法还包括指定选择项、子表达式分组和引用前一子表达式的特殊字符。

字符“|”用于分隔供选择的字符，比如：/ab|cd|ef/可以匹配字符串“ab”，也可以匹配字符串“cd”，还可以匹配字符串“ef”。

注意：选择项的尝试匹配次序是从左到右，直到发现了匹配项。如果左边的选择项匹配，就会忽略右边的匹配项，即使它产生更好的匹配。比如：当正在表达式/a|ab/匹配字符串“ab”时，它只能匹配第一个字符。

圆括号“()”可以把单独的项组合成子表达式。

带圆括号的表达式的另一个用途是允许在同一正则表达式的后部引用前面的子表达式，这是通过在字符“\”后加一位或多位数字来实现的，这个数字指定了带圆括号的子表达式在正则表达式中的位置。比如：\1引用的是第一个带圆括号的子表达式，\3引用的是第三个带圆括号的子表达式。

```
/( . )b( . )\1b\2/.test('abcabc') //true
```

上面的代码中，\1表示第一个括号匹配的内容，即第一个(.)，匹配的是“a”；\2表示第二个括号(.)，匹配的是“b”。

注意：因为子表达式可以嵌套另一个子表达式，所以引用的位置是参与计数的左括号的位置。比如：(s(ss))，\2表示的是(ss)。

对正则表达式中前一个子表达式的引用，并不是指对子表达式模式的引用，而指的是那个模式相匹配的文本的引用。

```
/(a|b)c\1/.test('aca') //true
/(a|b)c\1/.test('acb') //false
```

上面的代码中，由于(a|b)匹配的是a，所以\1所对应的也应该是a。

正则表达式的选择、分组和引用字符

| 选择，匹配的是该符号左边的子表达式或右边的子表达式

(...) 组合，将几个项组合为一个单位，这个单位可通过“*”、“+”、“?”和“|”等符号加以修饰，而且可以记住和这个组合相匹配的字符串以供此后的引用使用。

(?:...) 只组合，把项组合到一个单位，但不记忆与该组相匹配的字符

\n 和第n个分组第一次匹配的字符相匹配，组是圆括号中的子表达式（也有可能是嵌套的），组索引是从左到右的左括号数，“(?:”形式的分组不编码

1.1.6 指定匹配位置

除了匹配字符串中的字符外，有些正则表达式的元素匹配的是字符之间的位置，亦可称为正则表达式的锚。比如：\b匹配一个单词的边界，即位于\w（ASCII单词）字符和\W（非ASCII单词）之间的边界，或位于一个ASCII

单词与字符串的开始或结尾之间的边界。

正则表达式中的锚字符：

- ^ 匹配字符串的开头，在多行检索中，匹配一行的开头
- \$ 匹配字符串的结尾，在多行检索中，匹配一行的结尾
- \b 匹配一个单词的边界
- \B 匹配非单词边界的位置
- (?=p) 零宽正向先行断言，要求接下来的字符都与p匹配，但不能包括匹配p的那些字符
- (?!p) 零宽负向先行断言，要求接下来的字符不与p匹配

1.1.7 修饰符

正则表达式的修饰符，用以说明高级匹配模式的规则。修饰符是放在“/”符号之外的，也就是说，它们不是出现在两条斜杠之间的，而是在第二条斜杠之后。

在JavaScript中，支持三个修饰符：

- i 执行不区分大小写的匹配
- g 执行一个全局匹配，即找到所有的匹配，而不是在找到第一个之后就停止
- m 多行模式匹配，在这种模式下，如果待检索的字符串包含多行，那么^和\$锚字符除了匹配整个字符串的开始和结尾之外，还能匹配每行的开始和结尾

这些修饰符可以任意组合。比如：

```
/test/ig
```

1.2 用于模式匹配的String方法

String支持4种使用正则表达式的方法。

- search()：按照给定的正则表达式进行搜索，返回一个整数，表示第一个与之匹配的字符串的起始位置，如果找不到匹配的子串，将返回-1。
- match()：返回一个数组，成员是所有匹配的子字符串。
- replace()：按照给定的正则表达式进行替换，返回替换后的字符串。
- split()：按照给定规则进行字符串分割，返回一个数组，包含分割后的各个成员。

1.2.1 search()

按照给定的正则表达式进行搜索，返回一个整数，表示第一个与之匹配的字符串的起始位置，如果找不到匹配的子串，将返回-1。

```
"javascript".search(/script/i);
```

上面的代码的返回值为4

如果search()的参数不是正则表达式，则首先会通过RegExp构造函数将它转换成正则表达式，search()方法不支持全局检索，因为它忽略正则表达式参数中的修饰符g。

1.2.2 match()

match()方法的唯一参数是一个正则表达式，返回的是一个由匹配结果组成的数组。如果该正则表达式设置了修饰符g，则返回的数组包含字符串中的所有匹配结果。

```
'1 plus 2 equals 3'.match(/\d+/g) //返回["1", "2", "3"]
```

返回来的数组还带有另外两个属性：index和input，分别表示包含发生匹配的字符位置和引用的正在检索的字符串。

1.2.3 replace()

replace()方法用以执行检索与替换操作。其中第一个参数是一个正则表达式，第二个参数是要进行替换的字符串。

如果replace()的第一个参数是字符串而不是正则表达式，则replace()将直接搜索这个字符串，而不会像search()一样首先通过RegExp()将它转换为正则表达式。

replace方法的第二个参数可以使用美元符号\$，用来指代所替换的内容。

\$& 指代匹配的子字符串。

- \$` 指代匹配结果前面的文本。
- \$' 指代匹配结果后面的文本。
- \$n 指代匹配成功的第n组内容，n是从1开始的自然数。
- \$\$ 指代美元符号\$。

比如：

```
'hello world'.replace(/(\w+)\s(\w+)/, '$2 $1')
// "world hello"
```

replace方法的第二个参数还可以是一个函数，将每一个匹配内容替换为函数返回值。

```
'abca'.replace(/a/g, function(match){
  return match.toUpperCase();
});
// "AbcA"
```

replace()方法的第二个参数可以接受多个参数。第一个参数是捕捉到的内容，第二个参数是捕捉到的组匹配（有多少个组匹配，就有多少个对应的参数）。

1.2.4 split()

split()方法用以将调用它的字符串拆分为一个子串组成的数组。

```
'123,456,789'.split(',') //返回["123","456","789"]
```

split()方法的参数也可以是一个正则表达式。

1.3 RegExp对象

RegExp()构造函数带有两个字符串参数，第二个参数是可选的，它指定正则表达式的修饰符（可传入修饰符g、i、m或者它们的组合）；第一个参数包含正则表达式的主体部分，也就是正则表达式直接量中两条斜线之间的文本。

```
var regexp = new RegExp('\\d{5}', 'g')
```

上面的代码表示会全局的查找5个数字。

1.3.1 RegExp对象的属性

每个RegExp对象都包含5个属性：

- source 只读字符串，包含正则表达式的文本
- global 只读布尔值，用以说明这个正则表达式是否带有修饰符g
- ignoreCase 只读布尔值，用以说明正则表达式是否带有修饰符i
- multiline 只读布尔值，用以说明正则表达式是否带有修饰符m
- lastIndex 可读写的整数，如果匹配模式带有g修饰符，这个属性存储在整个字符串中下一次检索的开始位置。

1.3.2 RegExp的方法

RegExp对象定义了两个用于执行模式匹配操作的方法。

(1) exec()

正则对象的exec方法，可以返回匹配结果。如果发现匹配，就返回一个数组，成员是每一个匹配成功的子字符串，否则返回null。

```
/a|b|c/.exec('abc') // ["a"]
```

```
/a|b|c/.exec('qwe') // null
```

(2) test()

正则对象的test方法返回一个布尔值，表示当前模式是否能匹配参数字符串。

```
var s = /a/g;
var a = 'baba';
```

```
s.lastIndex //0  
s.test(a); //true  
  
s.lastIndex; //2  
s.test(a); //true
```

注意：如果正则表达式带有g修饰符，则每一次test方法都从上一次结束的位置开始向后匹配，也可以通过正则对象的lastIndex属性指定开始搜索的位置。

JSON

JSON

JSON (JavaScript Object Notation , JavaScript对象表示法) , 它是JavaScript的一个严格子集。

JSON只是一种简单数据格式，并不是只有JavaScript拥有。

1、语法

JSON对值的类型和格式有严格的规定：

- 复合类型的值只能是数组或对象，不能是函数、正则表达式对象、日期对象。
- 简单类型的值只有四种：字符串、数值（必须以十进制表示）、布尔值和null（不能使用NaN, Infinity, -Infinity和undefined）。
- 字符串必须使用双引号表示，不能使用单引号。
- 对象的键名必须放在双引号里面。
- 数组或对象最后一个成员的后面，不能加逗号。

1.1 简值

最简单的JSON数据形式就是简单值。

```
5  
"tg"  
null  
true  
false
```

上面的都是有效的JSON数据。

1.2 对象

JSON中的对象与JavaScript中的对象字面量基本一样，除了JSON中的对象要求对象的属性一定要加上双引号（对象字面量的属性对引号可有可无）。

```
{  
  name: "tg"  
}  
{  
  'name' : "tg"  
}
```

上面两个对象对于JSON来说是错误的，正确的写法如下：

```
{
  "name": "tg"
}
```

当然，你可以使用嵌套对象：

```
{
  "name": "tg",
  "school": {
    "name": "tg",
    "location": "gz"
  }
}
```

可能你注意到了，上面有两个同名（ name ）属性，这是没问题的，因为它们属于不同的对象，但是请注意，同一对象内不能出现两个同名属性。

1.3 数组

JSON数组采用的是JavaScript中的数组字面量。

```
{
  "color": ["red", "blue"]
}
```

当然，你也可以在数组中放对象，对象中放数组。

2、解析与序列化

JSON在JavaScript中流行的原因是JSON数据结构可以解析成有用的JavaScript对象。

2.1序列化和反序列化

JSON对象有两个方法：`JSON.stringify()` 和 `JSON.parse()`，分别表示将JavaScript对象序列化为JSON字符串和把JSON字符串解析为原生的JavaScript值。

2.1.1 JSON.stringify()

(1) 基本用法

`JSON.stringify` 方法用于将一个Javascript对象序列化为字符串。该字符串应该符合JSON格式，并且可以被 `JSON.parse` 方法还原。

注意：在序列化JavaScript对象时，所有函数及原型成员都会被有意忽略，不体现在结果中，而且值为`undefined`的任何属性也会被跳过，还有正则对象会被转成空对象。, 最终返回来的值为有效的JSON格式。

```
var data = {
  name: 'tg',
  age: 1,
```

```

books: [ "Javascript", 2],
location:{
  name: "gz",
},
test: function(){}
sex: undefined
};
console.log(JSON.stringify(data));
//{"name":"tg", "age":1, "books":["Javascript",2], "location":{"name":"gz"}}

```

在上面的例子中，将data对象转换成JSON字符串，我们可以看到，键名都被双引号括起来了，而对于原始类型的字符串（name: 'tg'），也由单引号转成了双引号，这是因为将来还原的时候，双引号可以让JavaScript引擎知道，tg是一个字符串，而不是一个变量名。而且test和sex并没有在返回值中，这是因为test是函数，而sex的值是undefined。

(2) 过滤结果

JSON.stringify() 还可以接受第二个参数，指定需要转成字符串的属性。

当传入的参数是 数组 时，那么JSON.stringify()的结果中将只包含数组中列出的属性。

例子：

```

var data = {
  name: 'tg',
  age: 1,
  books: [ "Javascript", 2],
  location:{
    name: "gz",
  },
  test: function(){}
  sex: undefined
};
console.log(JSON.stringify(data, ["name", "books"]));
//{"name":"tg", "books":["Javascript",2]}

```

在上面的例子中，传给JSON.stringify()的第二个参数是一个数组，其中包含两个字符串：“name” 和 “books”，这两个属性与将要序列化的对象中的属性是对应的，因此在返回的结果字符串中，就只会包含这两个属性（如上结果）。

第二个参数还可以是函数，它接收两个参数：属性（键）名和属性值，返回的值就是相应键的值（它改变了序列化对象的结果）。

属性名只能是字符串，而在值并非键值对的值时，键名可以是空字符串。

```

var jsonText = JSON.stringify(data, function(key, value){
  if(key == 'books'){
    return value.join(',')
  }
})

```

```

} else{
    return value;
}
});
console.log(jsonText);
//{"name":"tg", "age":1, "books":"Javascript, 2", "location":{"name":"gz"}}

```

在上面的例子中，如果键名为"books"，就将数组连成一个字符串，否则，返回原值。

注意：如果函数返回了undefined，那么相应的属性会被忽略。

(3) 字符串缩进

JSON.stringify()方法还可以接收第三个参数，用于控制结果中的缩进和空白符。如果这个参数是数值，那它表示的是每个级别缩进的空格数。比如，要在每个级别缩进4个空格：

```

var data = {
    name: 'tg',
    age: 1,
    books: [ "Javascript", 2],
    location:{
        name: "gz",
    },
    test: function(){}
    sex: undefined
};
console.log(JSON.stringify(data, null, 4));
// 结果
{
    "name": "tg",
    "age": 1,
    "books": [
        "Javascript",
        2
    ],
    "location": {
        "name": "gz"
    }
}

```

当传入有效的控制缩进的参数值时，结果字符串就会自动包含换行符。不过要注意：最大缩进空格数为10，所有大于10的值都会自动转换为10.

如果缩进参数是一个字符串而非数值，则这个字符串将在JSON字符串中被用作缩进字符。

例子：

```

var data = {
    name: 'tg',

```

```

age: 1,
books: [ "Javascript", 2],
location:{
  name: "gz",
},
test: function(){}
sex: undefined
};
console.log(JSON.stringify(data, null, 4));
// 结果
{
--"name": "tg",
--"age": 1,
--"books": [
----"Javascript",
----2
--],
--"location": {
----"name": "gz"
--}
}

```

注意：缩进字符串最长不能超过10个字符长，如果超过了10个，结果中只会显示10个字符。

(4) toJSON()方法

如果JSON.stringify的参数对象有自定义的toJSON方法，那么JSON.stringify会使用这个方法的返回值作为参数，而忽略原对象的其他属性。

```

var data = {
  name: 'tg',
  age: 1,
  toJSON:function(){
    var data = {
      name: this.name
    };
    return data;
  }
};
console.log(JSON.stringify(data));
//{"name":"tg"}

```

在上面的代码中，我们给data对象定义了一个toJSON()方法，该方法返回name的值。

可以让toJSON()方法返回任何序列化的值。

假设将一个对象传入JSON.stringify()，序列化对象的顺序如下：

- 如果存在toJSON()方法而且能通过它取得有效的值，则调用该方法，否则，按默认顺序序列化。

- 如果提供了第二个参数，应用这个函数过滤器。传入函数过滤器的值是第（1）步返回的值
- 对第（2）步返回的每个值进行相应的序列化
- 如果提供了第三个参数，执行相应的格式化

2.1.2 JSON.parse()

`JSON.parse` 方法用于将JSON字符串转化成原生的JavaScript对象。

例子：

```
var jsonText = '{"name":"tg","age":1,"books":["Javascript",2],"location":{"name":"gz"}}';
console.log(JSON.parse(jsonText));

//返回对象
{
  name: 'tg',
  age: 1,
  books: [ "Javascript", 2 ],
  location:{
    name: "gz",
  }
};
```

如果传入的字符串不是有效的JSON格式，则会报错。

`JSON.parse()`也可以接收另一个参数，该参数是一个函数，将在每个键值对上调用。

例子：

```
var jsonText = '{"name":"tg","age":1,"books":["Javascript",2],"location":{"name":"gz"}}';

var o =JSON.parse(jsonText,function(key, value){
  if(key == "name"){
    return "Hello";
  }
  return value;
});
console.log(o.name);
// Hello
```

在上面的例子中，在解析时，当遇到键名是“name”时，就将相应的值替换成“Hello”，最终返回的name就是“Hello”。

如果函数返回`undefined`，则表示要从结果中删除相应的键，如果返回其他值，则将其放入到结果中。

AJAX

AJAX

超文本传输协议 (HyperText Transfer Protocol , HTTP) 是用于从WWW服务器传输超文本到本地浏览器的传输协议 (transport) 。它可以使浏览器更加高效 , 使网络传输减少。

Ajax (Asynchronous JavaScript and XML) 描述了一种主要使用脚本操纵HTTP的Web应用架构。

Ajax的主要特点是使用脚本操纵HTTP和Web服务器进行数据交换 , 不会导致页面重载。

AJAX技术的核心是XMLHttpRequest对象 (简称XHR) 。

一、 XMLHttpRequest对象

所有浏览器 (IE7之前除外) 都支持XMLHttpRequest对象 , 它定义了用脚本操纵HTTP的API。除了常用的GET请求 , 这个API还包含实现POST请求的能力 , 同时它能用文本或Document对象的形式返回服务器的响应。

浏览器在XMLHttpRequest类上定义了它们的HTTP API , 这个类的每个实例都表示一个独立的请求/响应对 , 并且这个对象的属性和方法允许指定请求细节和提取响应数据。

具体来说 , AJAX包括以下几个步骤 :

- 创建AJAX对象 (实例化XMLHttpRequest对象)
- 发起HTTP请求
- 接收服务器传回的数据
- 更新网页数据

1.1实例化

实例化XMLHttpRequest对象 :

```
var xhr = new XMLHttpRequest();
```

注意 : 你可以重用已存在的XMLHttpRequest , 但这将会终止之前通过该对象挂起的任何请求。

由于IE7之前的版本不支持非标准的XMLHttpRequest()构造函数 , 不过能如下模拟 :

```
function createXHR(){
    if ( typeof XMLHttpRequest != 'undefined' ){
        return new XMLHttpRequest();
    }else if(typeof ActiveXObject != 'undefined'){
        if(typeof arguments.callee.activeXString != 'string'){
            var versions = ['MSXML2.XMLHttp.6.0', 'MSXML2.XMLHttp.3.0', 'MSXML2.XMLHttp'];
            i, len;
            for(i = 0, len = versions.length; i < len; i++){
                try{
                    new ActiveXObject(versions[i]);
                }
```

```

        argumentscalleeactivexstring = versions[i];
    }catch(e){

    }
}
return new ActiveXObject(argumentscalleeactivexstring);
}else{
    throw new Error('No XHR object available');
}
}

```

一个HTTP请求由4部分组成：

- HTTP请求方法或动作 (verb)
- 正在请求的URL
- 一个可选的请求头集合，其中可能包括身份验证信息
- 一个可选的请求主体

服务器返回的HTTP响应包含3部分：

- 一个数字和文字组成的状态码，用来显示请求的成功和失败
- 一个响应头集合
- 响应主体

注意：使用XMLHttpRequest时，必须把文件放到Web服务器上。因为AJAX只能向同源网址（协议、域名、端口都相同）发出HTTP请求，如果发出跨源请求，就会报错

1.2 指定请求

1.2.1 open()

创建XMLHttpRequest对象后，就可以调用XMLHttpRequest对象的open()方法去指定这个请求的两个必需部分：方法和URL。

```
xhr.open('GET', 'example.php');
```

上面代码向指定的服务器网址，启动GET请求。

open()的第一个参数指定HTTP方法或动作（常用的是“GET”和“POST”），这个字符串不区分大小写，但通常使用大写字母来匹配HTTP协议。“DELETE”、“HEAD”、“OPTIONS”、“PUT”也可以作为open()方法的第一个参数。

第二个参数是URL（跨域的请求通常会报错），它是请求的主题。

还有第三个可选参数，表示是否异步发送请求的布尔值（true同步，false异步，默认false）

调用open()方法并不会真正发送请求，而只是启动一个请求以备发送。

1.2.2 HTTP头部信息

每个HTTP请求和响应都会带有相应的头部信息。

XHR对象也提供了操作这两种头部（即请求头部和响应头部）信息的方法。

默认情况下，在发送XHR请求的同时，还会发送下列头部信息：

```
Accept : 浏览器能够处理的内容类型
Accept-Charset : 浏览器能够显示的字符集
Accept-Encoding : 浏览器能够处理的压缩编码。
Accept-Language : 浏览器当前设置的语言。
Connection : 浏览器与服务器之间连接的类型
Cookie : 当前页面设置的任何Cookie
Host : 发出请求的页面所在的域。
Referer : 发出请求的页面的URI。
User-Agent : 浏览器的用户代理字符串
```

如果有请求头，我们可以设置它。例如，POST请求需要“Content-Type”头指定请求主题的MIME类型。

```
xhr.setRequestHeader('Content-Type', 'text/plain');
```

注意：如果对相同的头调用setRequestHeader()多次，新值不会取代之前指定的值，相反，HTTP请求将包含这个头的多个副本或这个头将指定多个值。

必须在调用open()方法只会且调用send()方法之前调用setRequestHeader()。

1.2.3 send()

使用XMLHttpRequest发起HTTP请求的最后一步是指定可选的请求主体的数据并向服务器发送它。

```
xhr.send(null);
```

GET请求没有主体，所以应该传递null或者省略这个参数。POST请求通常拥有主体，同时它应该匹配使用setRequestHeader()指定的“Content-Type”头。

HTTP请求的各部分有指定顺序：请求方法和URL首先到达，然后是请求头，最后是请求主体。

XMLHttpRequest实现通常直到调用send()方法才开始启动网络。

1.2.4 GET请求

GET是最常见的请求类型，必要时，可以将查询字符串参数追加到URL的末尾，以便将信息发送给服务器。

注意：对于XHR来说，传入open()方法的URL末尾的查询字符串必须经过正确的编码，否则可能出现字符串格式的错误。

查询字符串中每个参数的名称和值都必须使用encodeURIComponent()进行编码，然后才能放到URL的末尾，而且所有名值对都必须由和号（&）分隔。如下所示：

```
xhr.open('get','example.php?name1=value1&name2=value2',true);
```

下面这个函数可以辅助向现有的URL的末尾添加查询字符串参数：

```
function addURLParam(url, name, value){
    url += (url.indexOf('?') == -1) ? '?' : '&';
    url += encodeURIComponent(name) + '=' + encodeURIComponent(value);
    return url;
}
```

1.2.5 POST请求

对于POST请求，应该把数据作为请求的主题提交，它不是以地址形式传参，而是在send()中传参。

注意：使用POST请求时，如果不设置Content-Type头部信息，那么发送给服务器的数据就不会出现在
\$_POST 中。

1.3 取得响应

一个完整的HTTP响应由 状态码、响应头集合 和 响应主体 组成。这些都可以通过XMLHttpRequest对象的属性和方法使用：

- `status` 和 `statusText` 属性以数字和文本的形式返回HTTP状态码。这些属性保存标准的HTTP值。像200和“OK”表示成功请求，404和“Not Found”表示URL不能匹配服务器上的任何资源。
- 使用 `getResponseHeader()` 和 `getAllResponseHeaders()` 能查询/获取响应头。XMLHttpRequest会自动处理cookie：它会从`getAllResponseHeaders()`头返回集合中过滤cookie头，而如果给`getResponseHeader()`传递“Set-Cookie”和“Set-Cookie2”，则返回null。
- 响应主体可以从`responseText`属性得到文本形式的，从`responseXML`属性中得到Document形式的。

为了在响应准备就绪时得到通知，我们必须监听XMLHttpRequest对象上的 `readystatechange` 事件。

每当发生状态变化的时候，`readyState`属性的值就会发生改变。这个值每一次变化，都会触发`readyStateChange`事件。

```
xhr.onreadystatechange=function(){}
```

1.3.1 同步响应

XMLHttpRequest默认是异步的，当然，如果有需要，我们也可以设置成同步（较少用）：

```
xhr.open('GET','example.php',false);
```

上面的代码中，通过传递第三个参数为`false`实现同步。要注意的是：一旦设置为同步，那么`send()`方法将阻塞直到请求完成。

1.3.2 响应解码

如果服务器发送诸如对象或数组这样的结构化数据作为响应，它应该传输JSON编码的字符串数据。

下面是一个解析不同类型的响应值的方法：

```
function get(url,callback){
    var xhr = new XMLHttpRequest(); //创建新请求
    xhr.open('GET',url);
    xhr.onreadystatechange=function(){
        //如果请求完成且成功
        if(xhr.readyState === 4 && xhr.status === 200){
            //获得响应的类型
            var type = xhr.getResponseHeader('Content-type');
            if(type.indexOf('xml') !== -1 && xhr.responseXML){
                callback(xhr.responseXML); //Document对象响应
            }else if(type === 'application/json'){
                callback(JSON.parse(xhr.responseText)); //JSON响应
            }else{
                callback(xhr.responseText); //字符串响应
            }
        }
    };
    xhr.send(null); //立即发送请求
}
```

1.4 编码请求主体

HTTP POST请求包括一个请求主体，它包含客户端传递给服务器的数据。

1.4.1 表单编码的请求

默认情况下，HTML表单通过POST方法发送给服务器，而编码后的表单数据则用做请求主体。例如：

```
user=TG&age=18;
```

表单数据编码格式有一个正式的MIME类型：

```
application/x-www-form-urlencoded
```

当使用POST方法提交这种顺序的表单数据时，必须设置“Content-Type”请求头为这个值。

```
xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
```

注意：当使用form时，这不是必需值，因为这是默认方法。

当我们发送给服务器的是一个JavaScript对象时，我们也可以采取这种类型的编码。

下面的函数encodeFormData()就是将对象转换为表单编码。

```
/*
* {
*   name: 'TG',
*   age:18
* }
*/
function encodeFormData(data){
  if(!data) return '';
  var pairs = [];
  for(var name in data){
    if(!data.hasOwnProperty(name)) continue;
    if(typeof data[name] === 'function') continue;
    var value = data[name].toString();
    name = encodeURIComponent(name.replace('%20', '+'));
    value = encodeURIComponent(value.replace('%20', '+'));
    pairs.push(name + '=' + value);
  }
  return pairs.join('&');
}
```

1.4.2 JSON编码的请求

使用JSON编码主体来发起HTTP POST请求

```
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.send(JSON.stringify(data));
```

1.5.XMLHttpRequest实例的属性

(1) readyState属性

`readyState` 是一个整数，它指定了HTTP请求的状态。它可能的值如下：

- 0 , 对应常量UNSENT , 表示XMLHttpRequest实例已经生成 , 但是open()方法还没有被调用。
- 1 , 对应常量OPENED , 表示open()已调用 , 但send()方法还没有被调用 , 仍然可以使用setRequestHeader() , 设定HTTP请求的头信息。
- 2 , 对应常量HEADERS_RECEIVED , 表示send()方法已经执行 , 并且头信息和状态码已经收到。
- 3 , 对应常量LOADING , 表示正在接收服务器传来的主体 (body) 部分的数据 , 如果responseType属性是text或者空字符串 , responseText就会包含已经收到的部分信息。
- 4 , 对应常量DONE , 表示服务器数据已经完全接收 , 或者本次接收已经失败了。

在IE8之前并没有定义这些值 , 但使用硬编码值4来表示XMLHttpRequest.DONE。

(2) status

status属性为只读属性，表示本次请求所得到的HTTP状态码，它是一个整数。一般来说，如果通信成功的话，这个状态码是200。

```
200, OK, 访问正常
301, Moved Permanently, 永久移动
302, Move temporarily, 暂时移动
304, Not Modified, 未修改
307, Temporary Redirect, 暂时重定向
401, Unauthorized, 未授权
403, Forbidden, 禁止访问
404, Not Found, 未发现指定网址
500, Internal Server Error, 服务器发生错误
```

例子：

```
if(xhr.readyState === 4){
    //请求完成
    if(xhr.status === 200){
        //请求成功
    }
}
```

(3) statusText

statusText属性为只读属性，返回一个字符串，表示服务器发送的状态提示。不同于status属性，该属性包含整个状态信息，比如“200 OK”。

(4) timeout

timeout属性等于一个整数，表示多少毫秒后，如果请求仍然没有得到结果，就会自动终止。如果该属性等于0，就表示没有时间限制。

对应超时，还有个监听函数：

```
xhr.ontimeout = function(){
    //请求超时
}
```

(5) response

response属性为只读，返回接收到的数据体（即body部分）。它的类型可以是ArrayBuffer、Blob、Document、JSON对象、或者一个字符串，这由XMLHttpRequest.responseText属性的值决定。
如果本次请求没有成功或者数据不完整，该属性就会等于null。

(6) responseType

`responseType`属性用来指定服务器返回数据（`xhr.response`）的类型。

```
"" : 字符串（默认值）
"arraybuffer" : ArrayBuffer对象
"blob" : Blob对象
"document" : Document对象
"json" : JSON对象
"text" : 字符串
```

`text`类型适合大多数情况，而且直接处理文本也比较方便，`document`类型适合返回XML文档的情况，`blob`类型适合读取二进制数据，比如图片文件。

(7) `responseText`

`responseText`属性返回从服务器接收到的字符串，该属性为只读。如果本次请求没有成功或者数据不完整，该属性就会等于`null`。

如果服务器返回的数据格式是JSON，就可以使用`responseText`属性。

```
var data = xhr.responseText;
data = JSON.parse(data);
```

(8) `responseXML`

`responseXML`属性返回从服务器接收到的`Document`对象，该属性为只读。如果本次请求没有成功，或者数据不完整，或者不能被解析为XML或HTML，该属性等于`null`。

返回的数据会被直接解析为DOM对象。

(9) 事件监听接口

`XMLHttpRequest`第一版，只能对 `onreadystatechange` 这一个事件指定回调函数。该事件对所有情况作出响应。`XMLHttpRequest`第二版允许对更多的事件指定回调函数。

```
onloadstart 请求发出
onprogress 正在发送和加载数据
onabort 请求被中止，比如用户调用了abort()方法
onerror 请求失败
.onload 请求成功完成
ontimeout 用户指定的时限到期，请求还未完成
onloadend 请求完成，不管成果或失败
```

1.6. XMLHttpRequest实例的方法

(1) `abort()`

`abort`方法用来终止已经发出的HTTP请求。

(2) `getAllResponseHeaders()`

getAllResponseHeaders方法返回服务器发来的所有HTTP头信息。格式为字符串，每个头信息之间使用CRLF分隔，如果没有受到服务器回应，该属性返回null。

(3) getResponseHeader()

getResponseHeader方法返回HTTP头信息指定字段的值，如果还没有收到服务器回应或者指定字段不存在，则该属性为null。

如果有多个字段同名，则它们的值会被连接为一个字符串，每个字段之间使用“逗号+空格”分隔。

(4) open()

XMLHttpRequest对象的open方法用于指定发送HTTP请求的参数，它的使用格式如下，一共可以接受五个参数。

```
void open( string method, string url, optional boolean async, optional string user, optional string password);
```

参数说明：

- method：表示HTTP动词，比如“GET”、“POST”、“PUT”和“DELETE”。
- url：表示请求发送的网址。
- async：格式为布尔值，默認為true，表示请求是否为异步。如果设为false，则send()方法只有等到收到服务器返回的结果，才会有返回值。
- user：表示用于认证的用户名，默認為空字符串。
- password：表示用于认证的密码，默認為空字符串。

(5) send()

send方法用于实际发出HTTP请求。如果不带参数，就表示HTTP请求只包含头信息，也就是只有一个URL，典型例子就是GET请求；如果带有参数，就表示除了头信息，还带有包含具体数据的信息体，典型例子就是POST请求。

(6) setRequestHeader()

setRequestHeader方法用于设置HTTP头信息。该方法必须在open()之后、send()之前调用。如果该方法多次调用，设定同一个字段，则每一次调用的值会被合并成一个单一的值发送。

```
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.setRequestHeader('Content-Length', JSON.stringify(data).length);
xhr.send(JSON.stringify(data));
```

在上面代码中，首先设置头信息Content-Type，表示发送JSON格式的数据；然后设置Content-Length，表示数据长度；最后发送JSON数据。

(6) overrideMimeType()

该方法用来指定服务器返回数据的MIME类型。该方法必须在send()之前调用。

```
// 强制将MIME改为文本类型
xhr.overrideMimeType('text/plain; charset=x-user-defined');
```

在XMLHttpRequest版本升级后，一般采用指定的responseType的方法。

```
xhr.responseType = 'text';
```

2、上传文件

讲到上传文件，我们最常用的就是使用 `<input type="file">` 元素选择文件，然后表单将在它产生POST请求主体中发送文件内容。

```
<form id="form" action="upload.php" method="POST">
  <input type="file" id="files" name="photos[]"/>
  <button type="submit" id="upload">上传</button>
</form>
```

如果要允许选择多个文件，可设置file控件的multiple属性。

```
<input type="file" multiple/>
```

1.5.1 files属性

每个 `<input type="file">` 元素都有一个files属性，返回一个FileList对象，它是File对象中的类数组对象，包含了用户选择的文件。

```
var fileInput = document.getElementById('files');
var files = fileInput.files;
```

通常情况下，对于文件上传元素，我们都会添加change事件处理程序，每次文件信息有变化，它都会触发。

```
fileInput.addEventListener('change', function(){
  var file = this.files[0];
  if(!file) return;
  var xhr = new XMLHttpRequest();
  xhr.open('POST', 'upload.php');
  xhr.send(file);
}, false);
```

文件类型是二进制大对象（Blob）类型中的一个子类型。XHR2允许向send()方法传入任何Blob对象。如果没有

显式设置Content-Type头，这个Blob对象的type属性用于设置待上传的Content-Type头。

也可以显式设置：

```
xhr.setRequestHeader('Content-Type', file.type);
```

1.5.2 multipart/form-data请求

当HTML表单同时包含文件上传元素和其他元素时，浏览器不能使用普通的表单编码而必须使用成为“multipart/form-data”的特殊Content-Type头来用POST方法提交表单。

1.5.3 FormData对象

XHR2定义了新的FormData API，它容易实现多部分请求主体。

首先，新建一个FormData对象的实例，用来模拟发送到服务器的表单数据，把选中的文件添加到这个对象上面。

```
var formdata = new FormData();
```

然后按需多次调用这个对象的append()方法把个体“部分”添加到请求中。

```
for(var i = 0; i < files.length; i++){
  var file = files[i];
  formdata.append('photos[]', file, file.name);
}
```

最后，把FormData对象传递给send()方法

```
xhr.send(formdata);
```

除了可以添加文件，还可以添加二进制对象（Blob）或者字符串。

```
// Files
formdata.append(name, file, filename);

// Blobs
formdata.append(name, blob, filename);

// Strings
formdata.append(name, value);
```

append方法的第一个参数是表单的控件名，第二个参数是实际的值，第三个参数是可选的，通常是文件名。

3、HTTP进度事件

除了使用readystatechange事件来探测HTTP请求的完成外，在XHR2中，还定义了多个有用的事件。

- abort事件：当进度事件被中止时触发。如果发生错误，导致进程中止，不会触发该事件。
- error事件：由于错误导致资源无法加载时触发。
- load事件：进度成功结束时触发。
- loadstart事件：进度开始时触发。
- loadend事件：进度停止时触发，发生顺序排在error事件\abort事件\load事件后面。
- progress事件：当操作处于进度之中，由传输的数据块不断触发。
- timeout事件：进度超过限时触发。

当调用send()时，触发单个loadstart事件。当正在加载服务器的响应时，XMLHttpRequest对象会发现progress事件，通常每隔50毫秒左右，可以使用这些事件给用户反馈请求的进度。当事件完成，会触发load事件。

HTTP请求无法完成有3种情况：

- 请求超时，会触发timeout事件
- 请求终止，会触发abort事件
- 请求发生错误，会触发error事件

注意：对于任何具体请求，浏览器将只会触发load、abort、timeout和error事件中的一个。一旦这些事件中的一个发生后，浏览器应该触发loadend事件。

要使用这些事件，有两种方式：

```
xhr.onload=function(){}
xhr.addEventListener('load',function(){})
```

3.1 progress事件

因为这些事件是XHR2中才定义的，所以有时需要检查浏览器是否支持progress事件：

```
if('onprogress' in (new XMLHttpRequest())){
  //支持progress事件
}
```

除了像type和timestamp这样的常用Event对象属性外，与progress事件相关联的事件对象有3个有用的属性：

- lengthComputable：返回一个布尔值，表示当前进度是否具有可计算的长度。如果为false，就表示当前进度无法测量。
- total：返回一个数值，表示当前进度的总长度。如果是通过HTTP下载某个资源，表示内容本身的长度，不

含HTTP头部的长度。如果lengthComputable属性为false，则total属性就无法取得正确的值。

- loaded：返回一个数值，表示当前进度已经完成的数量。该属性除以total属性，就可以得到目前进度的百分比。

我们可以利用total和loaded属性来获取当前进度：

```
xhr.onprogress = function(e){
  if(e.lengthComputable){
    var percentComplete = e.loaded / e.total;
  }
}
```

3.2 上传进度事件

在XHR2中，也提供了用于监控HTTP请求上传的事件。在实现这些特性的浏览器中，XMLHttpRequest对象有一个upload属性，upload属性值是一个对象，它定义了addEventListener()方法和整个progress事件集合，比如onprogress和onload。

```
xhr.upload.onprogress = function(e){
  var percentComplete = e.loaded / e.total;
}
```

3.3 中止请求和超时

我们可以通过调用XMLHttpRequest对象的abort()方法来取消正在进行的HTTP请求，调用abort()方法时，会触发abort事件。

在XHR2中，还定义了timeout属性来指定自动中止后的毫秒数。

```
xhr.timeout = 1000;
```

4、同源策略

同源策略是对JavaScript代码能够操作哪些Web内容的一条完整的安全限制。当Web页面使用多个`<iframe>`元素或打开其他浏览器窗口的时候，这一策略通常就会发挥作用。

所谓“同源”指的是“三个相同”。

- 协议相同
- 域名相同
- 端口相同

从不同Web服务器载入的文档具有不同的来源。通过同一主机的不同端口载入的文档具有不同的来源。使用http:协议载入的文档和使用https:协议载入的文档具有不同的来源，即使它们来自同一个服务器。

同源政策的目的，是为了保证用户信息的安全，防止恶意的网站窃取数据。

4.1 不严格的同源策略

有三种不严格的同源策略

(1) 使用Document对象的domain属性

在默认情况下，属性domain存放的是载入文档的服务器的主机名。

比如：来自home.example.com的文档里的脚本要合法的读取developer.example.com载入的文档的属性（默认情况下是不允许的，会受到同源策略的限制）。

```
document.domain = 'example.com';
```

一旦上面两个文档里包含的脚本把domain都设置成了上面相同的值，那么这两个文档就不会受同源策略的约束了，可以相互读取对方的属性。

(2) 跨域资源共享 (Cross-Origin Resource Sharing)

这种方式是应用到后台中：

```
Access-Control-Allow-Origin: *, //允许所有域名的脚本访问该资源。
```

```
Access-Control-Allow-Origin: http://www.example.com //允许特定的域名访问。
```

(3) 跨文档消息 (cross-document messaging)

跨文档消息允许来自一个文档的脚本可以传递文本消息到另一个文档里的脚本。

4.2 跨站脚本

跨站脚本 (Cross-site scripting , XSS)，这个术语用来表示一类安全问题，也就是攻击者向目标Web站点注入HTML标签或脚本。

通常，防止XSS攻击的方式是在使用任何不可信的数据来动态的创建文档内容之前，从中移除HTML标签。

5、Fetch API

Fetch API是一种新规范，用来取代XMLHttpRequest对象。

5.1 特性检测

```
if(self.fetch){
    //支持
}else{
    //不支持
}
```

在Fetch API中，最常用的就是fetch()函数，它接收一个URL参数（也可以是request对象），返回一个promise来处理response。response参数还带着一个Response对象。

```
fetch(url).then(function(response){
```

```
    console.log(response);
});
```

fetch方法还可以设置第二个参数，用来配置其他值，可选的参数有：

- method: 请求使用的方法，如 GET、POST。
- headers: 请求的头信息，形式为 Headers 对象或 ByteString。
- body: 请求的 body 信息：可能是一个 Blob、BufferSource、FormData、URLSearchParams 或者 USVString 对象。注意 GET 或 HEAD 方法的请求不能包含 body 信息。
- mode: 请求的模式，如 cors、no-cors 或者 same-origin。
- credentials: 请求的 credentials，如 omit、same-origin 或者 include。
- cache: 请求的 cache 模式: default, no-store, reload, no-cache, force-cache, or only-if-cached.

下面是发出POST请求

```
fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: 'name=TG&love=1'
}).then(function(response){})
```

注意：fetch() 方法的参数与 Request() 构造器是一样的。

```
fetch(input, init).then(function(response) { ... });

var myRequest = new Request(input, init);
```

下面是一个Fetch API完整请求的简单例子：

```
fetch(url).then(function (response) {
  return response.json();
}).then(function (jsonData) {
  console.log(jsonData);
}).catch(function () {
  console.log('出错了');
});
```

上面代码向指定的URL发出请求，得到回应后，将其转为JSON格式，输出到控制台。如果出错，则输出一条提示信息。

Fetch API引入三个新的对象（也是构造函数）：Headers, Request和Response。

5.2 Headers

Headers对象用来构造/读取HTTP数据包的头信息。

```
reqHeaders = new Headers({
  "Content-Type": "text/plain",
  "Content-Length": content.length.toString(),
  "X-Custom-Header": "ProcessThisImmediately",
});
```

我们还可以使用append方法：

```
var content = 'Hello World';
var headers = new Headers();
headers.append("Accept", "application/json");
headers.append("Content-Type", "text/plain");
headers.append("Content-Length", content.length.toString());
headers.append("X-Custom-Header", "ProcessThisImmediately");
Headers对象实例还提供了一些方法：
reqHeaders.has("Content-Type") // true
reqHeaders.has("Set-Cookie") // false
reqHeaders.set("Content-Type", "text/html")
reqHeaders.append("X-Custom-Header", "AnotherValue")

reqHeaders.get("Content-Length") // 11
reqHeaders.getAll("X-Custom-Header") // ["ProcessThisImmediately", "AnotherValue"]

reqHeaders.delete("X-Custom-Header")
reqHeaders.getAll("X-Custom-Header") // []
```

生成Header实例以后，可以将它作为第二个参数，传入Request方法。

```
var headers = new Headers();
headers.append('Accept', 'application/json');
var request = new Request(URL, {headers: headers});

fetch(request).then(function(response) {
  console.log(response.headers);
});
```

5.3 Request对象

Request对象用来构造HTTP请求。

```
var req = new Request("/index.html");
```

```
req.method // "GET"
req.url // "http://example.com/index.html"
```

Request对象的第二个参数，表示配置对象，

```
var uploadReq = new Request("/uploadImage", {
  method: "POST",
  headers: {
    "Content-Type": "image/png",
  },
  body: "image data"
});
```

Request对象实例的mode属性，用来设置是否跨域，合法的值有以下三种：same-origin、no-cors（默认值）、cors。当设置为same-origin时，只能向同域的URL发出请求，否则会报错。

5.4 Response对象

Fetch API 的Response接口呈现了对一次请求的响应数据

5.4.1 属性

(1) ok

如果ok属性返回的状态码在200到299之间（即请求成功），这个属性为true，否则为false。因此，我们可以这样判断请求是否成功：

```
fetch(url).then(function(response){
  if(response.ok){
    //请求成功
  }else{
    //请求失败
  }
});
```

(2) status、statusText

status属性返回HTTP的状态码；statusText返回一个字符串，表示服务器发送的状态提示。比如通信成功时，status是200，而statusText是“OK”

(3) url

返回完整的请求地址

(4) type

type属性表示HTTP回应的类型。合法的值有五个basic、cors、default、error、opaque。basic表示正常的同域请求；cors表示CORS机制的跨域请求；error表示网络出错，无法取得信息，status属性为0，如果需要在CORS机制下发出跨域请求，需要指明状态。

```
fetch(url, {mode: 'cors'}).then(function(response){})
```

(5) headers

Headers对象，表示HTTP回应的头信息

(6) body

表示请求的内容。

Request对象和Response对象都有body属性，表示请求的内容。body属性可能是以下的数据类型。

```
ArrayBuffer
ArrayBufferView (Uint8Array等)
Blob/File
string
URLSearchParams
FormData
```

注意：上面这些方法都只能使用一次，第二次使用就会报错，也就是说，body属性只能读取一次。Request对象和Response对象都有bodyUsed属性，返回一个布尔值，表示body是否被读取过。

如果希望多次使用body属性，可以使用Response对象和Request对象的clone方法。它必须在body还没有读取前调用，返回一个新的body，也就是说，需要使用几次body，就要调用几次clone方法。

response.clone()

(7) bodyUsed

bodyUsed属性，返回一个布尔值，表示body是否被读取过。

5.4.2 Response对象的方法

(1) text()、json()、FormData()、blob()、arrayBuffer()

在Fetch API中，数据传送是以数据流（stream）的形式进行的。对于大文件，数据是一段一段得到的。

而Fetch API提供了五个数据流读取器。

```
text() : 返回字符串
json() : 返回一个JSON对象
formData() : 返回一个FormData对象
blob() : 返回一个blob对象
arrayBuffer() : 返回一个二进制数组
```

简单例子：

```
response.json().then(function(json){
  console.log(json);
})
```

表单和富文本编辑器

[表单](#)

[富文本编辑器](#)

表单

表单

Web表单是开发人员与用户交互的重要控件。

1、form

1.1 form独有的属性和方法

在HTML中，表单是由 `<form>` 来表示的，而在JavaScript中，表单对应的是`HTMLFormElement`类型。

`HTMLFormElement`继承了`HTMLElement`，因此它跟其他HTML元素具有相同的默认属性。

`<form>` 也有自己独有的属性和方法：

```
acceptCharset : 服务器能够处理的字符集  
action : 接收请求的URL  
elements : 表单中所有控件的集合 (HTMLCollection)  
enctype : 请求的编码类型  
length : 表单中控件的数量  
method : 要发送的HTTP请求类型，通常是“get”或“post”  
name : 表单的名称  
reset() : 将所有表单域重置为默认值  
submit() : 提交表单  
target : 用于发送请求和接收响应的窗口名称
```

1.2 获取表单元素

获取表单元素一般有两种方式：

- 通过id来获取，比如获取一个id名为form1的表单元素：

```
document.getElementById('form1');
```

-通过`document.forms`获取name名为form1的表单元素：

```
document.forms['form1']
```

`document.forms` 可以获取到当前页面中所有的表单元素，我们又可以通过方括号表示法获取某个属性，传入数值索引或 `name` 值。

1.3 提交表单

提交表单也有两种方式：

- 通过 `<input>` 或 `<button>` 提交：

```
<input type="submit" />
<button type="submit"></button>
<input type="image" src="example.png" />
```

只要 `<form>` 内有上面这三种按钮，点击的时候就会提交表单，而且，当相应表单控件拥有焦点时，按回车键也会提交表单（焦点在`textarea`里例外，回车键是换行）。

- 通过JavaScript触发`submit()`提交：

```
var form1 = document.getElementById('form1');
form1.submit();
```

在请求发送给服务器之前，浏览器会触发`submit`事件，我们可以主动监听它：

```
var form1 = document.getElementById('form1');
form1.onsubmit = function(){
}
```

注意：在调用`submit()`方法提交表单时，不会触发`submit`事件。

1.4 重置表单

重置表单也有两种方式：

- 重置按钮

```
<input type="reset" />
<button type="reset"></button>
```

当点击重置按钮时，会触发`reset`事件：

```
form1.onreset = function(){
}
```

- 通过`reset()`方法

```
form1.reset();
```

与调用`submit()`不同，调用`reset()`方法时也会触发`reset`事件。

1.5 表单字段

除了使用原生DOM方法访问表单元素外，每一个表单都有elements属性，该属性是表单中所有元素的集合（比如 `<input>`、`<textarea>`、`<button>`、`<fieldset>` ），elements是一个有序列表，包含着表单中的所有字段。

```
<form id="form1">
  <input type="text" name="yourname" />
  <textarea name="intro"></textarea>
</form>

form1.elements[0] // 取得表单中的第一个字段
form1.elements['yourname']; // 取得name名为"yourname"的字段
```

如果表单内有多个同名（name）表单控件，那么用name取时，就会返回一个NodeList集合。

```
<form id="form1">
  <input type="radio" name="color" /> red
  <input type="radio" name="color" /> green
</form>

var colors = form1.elements['color']
```

colors是一个NodeList集合，包含了上面两个radio。

1.5.1 表单字段的共有属性

除了 `<fieldset>` 元素，所有表单字段都拥有一些相同的属性。

共有的属性和方法：

- `disabled`：布尔值，表单当前字段是否被禁用
- `form`：指向当前字段所属表单的指针，只读
- `name`：当前字段的名称
- `readonly`：布尔值，表示当前字段是否只读
- `tabIndex`：表单当前字段的切换（tab）序号
- `type`：当前字段的类型
- `value`：当前字段将被提交给服务器的值。

`<input>` 和 `<button>` 元素的type属性是可以修改，但 `<select>` 元素则是只读的。

1.5.2 表单字段的共有方法

每个表单字段都有两个方法：`focus()`和`blur()`。

- `focus()`方法用于让表单字段获取到焦点
- `blur()`方法用于让表单字段失去焦点

HTML5新增了一个autofocus属性，作用相当于focus()，只要设置了此属性，表单字段就会自动获取焦点。

1.5.3 表单字段的共有事件

所有表单都支持下列三个事件：

- blur：当前字段失去焦点时触发
- change：对于 `<input>`、`<textarea>` 元素，在它们失去焦点且value值改变时触发，对于 `<select>` 元素，在选项改变时触发
- focus：当前字段获得焦点时触发

如果你要实时的监听表单字段的值是否变化，可使用如下代码：

```
if('oninput' in document){
  input.addEventListener('input', function(){});
} else{
  input.onpropertychange = function(){};
}
```

当支持input事件时，就使用input，否则使用onpropertychange (IE特有)

onchange、oninput和onpropertychange三个事件的区别：

`onchange` 事件在内容改变（两次内容有可能还是相等的）且失去焦点时触发；

`oninput` 事件是IE之外的大多数浏览器支持的事件，在value改变时触发，实时的，即每增加或删除一个字符就会触发，然而通过js改变value时，却不会触发；

`onpropertychange` 事件是任何属性改变都会触发的，而oninput却只在value改变时触发，oninput要通过addEventListener()来注册。

失效情况：

- oninput事件：当脚本中改变value时，不会触发；从浏览器的自动下拉提示中选取时，不会触发。
- onpropertychange事件：当input设置为disabled=true后，onpropertychange不会触发。

2、文本框

文本框有两种：`<input>` 和 `<textarea>`。

```
<input type="text" />

<textarea cols ="10" rows ="5"></textarea>
```

2.1 文本框独有的属性

`<input>` 的独有属性：

`size`：指定文本框能够显示的字符数

`maxlength` : 指定文本框可以接受的最大字符数

对于 `<textarea>` , 它还有一些独有属性 :

`rows` : 指定的文本框的字符行数

`cols` : 指定的文本框的字符列数

注意 : `<textarea>` 的初始值是放在 `<textarea>` 和 `</textarea>` 之间的。

2.2 选择文本

文本框都支持`select()`方法 , 用于选择文本框中的所有文本。当调用`select()`方法时 , 大多数浏览器都会将焦点设置到文本框中。

当用户选择了文本框中的文本时 , 会触发`select`事件。

```
input.addEventListener('select', function(){});
```

2.2.1 取得选择的文本

在支持HTML5的浏览器中 , 我们可以获取到用户选择了什么文本 , 通过两个属性 : `selectionStart`和`selectionEnd` , 这两个属性保存的是基于0的数值 , 表示所选择文本的范围。

取得用户选择的文本 :

```
function getSelectedText(textbox){
    if(typeof textbox.selectionStart == 'number'){
        return textbox.value.substring(textbox.selectionStart, textbox.selectionEnd);
    }else if(document.selection){
        return document.selection.createRange().text;
    }
}
```

IE8之前不支持`selectionStart`。

2.2.2 选择部分文本

所有文本框都有一个 `setSelectionRange()` 方法 (支持HTML5的浏览器中) , 它接受两个参数 : 要选择的第一个字符的索引和要选择的最后一个字符的索引。

不过在IE8及之前的版本并不支持这个方法。当然 , 也有替代的方法。

在IE上 , 我们使用 `createTextRange()` 创建一个范围 , 然后使用 `moveStart()` 和 `moveEnd()` 方法将这个范围移动到需要获取文本的位置上。不过 , 在使用这两个方法之前 , 还必须使用 `collapse()` 将范围折叠到文本框的开始位置 , 此时 , `moveStart()` 将范围的起点和终点都移动到了相同的位置 , 只要再给 `moveEnd()` 传入要选择的字符总数即可。最后一步 , 就是使用范围的 `select()` 方法选择文本。

```

function selectText(textbox, startIndex, stopIndex){
    if(textbox.setSelectionRange){
        textbox.setSelectionRange(startIndex, stopIndex);
    }else if(textbox.createTextRange()){
        var range = document.createTextRange();
        range.collapse(true);
        range.moveStart('character', startIndex);
        range.moveEnd('character', stopIndex - startIndex);
        range.select();
    }
    textbox.focus();
}

```

`moveStart()` 和 `moveEnd()` 两个方法其实可用看做是 `setSelectionRange()` 的两个分解方法，获取开始点和获取结束点。

2.2.3 过滤输入

对于很多文本框来说，都不会任由用户输入文本，或多或少都会有所限制，这个时候，我们就需要监听 `keypress` 键盘事件，通过 `event` 事件对象中的字符编码 `keyCode` 来判断输入字符是否该被屏蔽

```

textbox.addEventListener('keypress', function(event){
    var keycode = event.keyCode;
});

```

2.2.4 操作剪贴板

剪切板事件：

- `beforecopy` : 在发生复制操作前触发
- `copy` : 在发生复制操作时触发
- `beforecut` : 在发生剪切操作前触发
- `cut` : 在发生剪切操作时触发
- `beforepaste` : 在发生黏贴操作前触发
- `paste` : 在发生黏贴操作时触发

要访问剪切板中的数据，可以使用 `clipboardData` 对象，在IE下，这个对象是 `window` 属性，在其他浏览器，这个对象是相应 `event` 对象的属性。

注意：在IE中，可以随时访问 `clipboardData` 对象；在其他浏览器中，只有在处理剪贴板事件期间 `clipboardData` 对象才有效。

`clipboardData` 对象有三个方法：

- `getData()`：用于从剪切板中取得数据，它接受一个参数，即要取得的数据的格式。在IE中，有两种数据格

式：“text” 和 “URL”；在其他浏览器中，这个参数是一种MIME类型，比如 “text/plain”，不过可以使用 “text” 代替 “text/plain”

- `setData()`：用于放置剪切板中的文本，它接受两个参数，第一个参数是数据格式（和`getData()`中的数据格式一样，IE支持 “text” 和 “URL”，其他浏览器只支持MIME类型），第二个参数是要放在剪切板中的文本。成功的将文本放到剪切板后，都会返回true。

`clearData()`：用于删除剪切板中指定格式的数据，它接受一个参数，即要删除数据的格式。

```
function getClipboardText(event){
  var clipboardData = (event.clipboardData || window.clipboardData);
  return clipboardData.getData('text');
}

function setClipboardText(event, value){
  if(event.clipboardData){
    return event.clipboardData.setData('text/plain', value);
  }else if(window.clipboardData){
    return window.clipboardData.setData('text', value);
  }
}
```

2.2.5 HTML5 约束验证 API

HTML5位表单字段提供了自动验证的功能，当然，要使用这些功能，开发者必须指定一些约束。

(1) 必填字段

可以使用 `required` 属性来将字段定位必填字段：

```
<input type="text" required />
```

只要设置了 `required` 属性，这个表单字段就不能为空。

当然，我们也可以使用JavaScript来获取或设置字段是否必填：

```
document.forms[0].elements['name'].required
```

由于是HTML5中定义的，我们有时需要检测浏览器是否支持：

```
if( 'required' in document.createElement('input') ){
}
```

(2) 输入模式

HTML5位文本字段新增了 `pattern` 属性，这个属性的值是一个正则表达式，用于匹配文本框中的值

```
<input type="text" pattern="\d+" />
```

(3) 检测有效性

使用 `checkValidity()` 方法可以检测表单中的某个字段是否有效。所有表单字段都有这个方法，如果字段的值有效，则返回`true`，否则返回`false`。

```
if( document.forms[0].elements[0].checkValidity() ){}
```

当然，如果要检测整个表单是否有效，可以在表单自身调用这个方法，当所有表单字段有效，才返回`true`，只要有一个字段无效，则返回`false`。

```
if( document.forms[0].checkValidity() ){}
```

每个表单字段还有一个更有效的属性：`validity`，它包含了一系列属性，每个属性会返回一个布尔值

`customError`：如果设置了`setCustomValidity()`，则为`true`，否则为`false`。

`patternMismatch`：如果值与指定的`pattern`属性不匹配，返回`true`

`rangeOverflow`：如果值比`max`值大，返回`true`

`rangeUnderflow`：如果值比`min`值小，返回`true`

`stepMismatch`：如果`min`和`max`之间的步长值不合理，返回`true`

`tooLong`：如果值的长度超过了`maxlength`属性指定的长度，返回`true`

`typeMismatch`：如果值不是“`mail`”或“`url`”要求的格式，返回`true`

`valid`：如果这里的其他属性都是`false`，返回`true`

`valueMissing`：如果标注为`required`的字段中没有值，返回`true`

例子：

```
if( input.validity && !input.validity.valid){  
}
```

(4) 禁用验证

通过设置 `novalidate` 属性，可以告诉表单不进行验证。

```
<form novalidate></form>
```

当然，也可以通过JavaScript设置：

```
document.forms[0].noValidate = true; // 禁用验证
```

我们也可以通过给提交按钮添加 `formnovalidate` 属性来禁用验证

```
<form>
  <input type="submit" formnovalidate />
</form>
```

3、选择框

HTML中的选择框通过 `<select>` 和 `<option>` 元素创建。

`<select>` 属于HTMLSelectElement类型，有下列的独有属性和方法：

- `add(newOption, relOption)`：向控件中插入新的 `<option>` 元素，其位置在相关项 (`relOption`) 之前。
- `multiple`：布尔值，表示是否允许多项选择
- `options`：控件中所有 `<option>` 元素的HTMLCollection。
- `remove(index)`：移除给定位置的选项
- `selectedIndex`：基于0的选中项的索引，如果没有选中项，则值为-1；对于多选项来说，只保存选中项中第一项的索引
- `size`：选择框中可见的行数

选择框的`type`属性不是 “select-one” ，就是 “select-multiple” 。

选择框的`value`属性由当前选中项决定，相应规则如下：

- 如果没有选中的项，则选择框的`value`属性保存空字符串
- 如果有一个选中项，而且该项的`value`属性已经制定，则选择框的`value`等于选中项的`value`属性的值
- 如果有一个选中项，但该项的`value`属性没有指定，则选择框的`value`等于该项的文本
- 如果有多个选中项，则选择框的`value`将依据前两条规则取得第一个选中项的值。

在DOM中，每个 `<option>` 元素都有一个HTMLOptionElement对象，其有下列属性：

- `index`：当前选项在`options`集合中的索引
- `label`：当前选项的标签
- `selected`：布尔值，表示当前选项是否被选中。将其设置为`true`可以选中该项
- `text`：选项的文本
- `value`：选项的值

富文本编辑器

富文本编辑器

1、富文本编辑器简介

富文本编辑 (WYSIWYG (What You See Is What You GET , 所见即所得))。

最先的富文本编辑，就是在页面中嵌入一个包含空HTML页面的 `iframe`，然后通过设置 `designMode` 属性，这个空白的HTML页面就可以编辑了，编辑对象则是该页面 `<body>` 元素的HTML代码。

`designMode`属性有两个可能的值：“`off`” 和 “`on`”，默认为 “`off`”。

给`iframe`指定一个非常简单的HTML页面作为编辑框：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Edit</title>
  </head>
  <body></body>
</html>
```

要将其设置成可编辑，不是在`iframe`页面内，而是它嵌入的那个页面，而且必须等待页面完全加载后才能设置 `designMode` 属性：

```
<iframe name="richedit" style="width:100px;height:100px" src="blank.html" ></frame>

window.onload = function(){
  window.frames['richedit'].document.designMode = 'on';
}
```

等到上面的代码执行完，就可以使用可编辑区了。

1.1 使用`contenteditable`属性

可以给页面中的任何元素添加 `contenteditable` 属性，使其成为可编辑区：

```
<div id="richedit" contenteditable="true"></div>
```

当然，也可以通过JavaScript设置：

```
var richedit = document.getElementById('richedit');
richedit.contentEditable = 'true';
```

1.2 获取编辑区里的HTML内容

```
// iframe
frames['richedit'].document.body.innerHTML

// 设置contenteditable的元素
richedit.innerHTML
```

2 富文本选区

在富文本编辑器中，使用框架（iframe）的 `getSelection()` 方法，可以确定实际选择的文本，这个方法是 `window` 对象和 `document` 对象的属性，调用它会返回一个表示当前选择文本的 `Selection` 对象。

每个 `Selection` 对象可能包含一个或多个 `Range` 对象。

`Range` 对象表示文档的连续范围区域，简单的说就是高亮选区。一个 `Range` 的开始点和结束点位置任意，开始点和结束点也可以是一样的（空 `Range`）。最常见的就是用户在浏览器窗口中用鼠标拖动选中的区域。

不过，不同的浏览器，`Range` 对象是不一样的，在 Chrome、Mozilla、Safari 等主流浏览器上，`Range` 属于 `selection` 对象（表示 `range` 范围），而在 IE 下，`Range` 属于 `TextRange` 对象（表示文本范围）。

在下面的所有例子中，皆以 `selection` 对象为主，同时会加上 `textRange` 对象的兼容代码。

2.1 拖动选择获取

每一个浏览器窗口都有一个 `selection`（或 `text Range`）对象，我们可以通过 `window.getSelection()` 方法来获取：

```
var selectedRange;
function saveSelection(){
  if(window.getSelection){
    /*主流的浏览器，包括chrome、Mozilla、Safari*/
    return window.getSelection();
  }else if(document.selection){
    /*IE下的处理*/
    return document.selection.createRange();
  }
  return null;
};
```

注意：标准 `dom` 是从 `window` 中获取 `selection` 对象，而 `ie` 是从 `document` 对象中获取。

实例（可以试试在不同浏览器下的执行，点击里面的按钮 `btn1`）：[Demo](#)

2.2 优化获取代码

一个 `selection` 对象有可能不是只有一个 `Range` 对象，有可能有多个，每一个 `Range` 对象代表用户鼠标所选取范围内的一段连续区域。（在 Firefox 中，可以通过 `ctrl` 键可以选取多个连续的区域，因此在 Firefox 中一个 `selection` 对象有多个 `range` 对象，在其他浏览器中，用户只能选取一段连续的区域，因此只有一个 `range` 对象。）

如何获取某个 `Range` 对象呢？我们可以通过 `selection` 对象的 `getRangeAt` 方法来获取：

```
range = window.getSelection().getRangeAt(index)
```

`getRangeAt()` 方法接受一个参数，代表该Range对象的序列号，也可以说你拖动选择的顺序号，它的值有如下几种情况：

- 当用户没有按下鼠标时候，该参数的值为0.
- 当用户按下鼠标的时候，该参数值为1.
- 当用户使用鼠标同时按住ctrl键时选取了一个或者多个区域时候，该参数值代表用户选取区域的数量。
- 当用户取消区域的选取时，该属性值为1，代表页面上存在一个空的Range对象；

要获取Range对象，一般我们会判断是否有Range对象，我们可以通过selection对象的rangeCount属性（类似数组的length，返回Range对象的数量）来判断是否多个Range对象，然后再去调用getRangeAt()方法。

对于富文本编辑器来说，一般情况下，我们只需要一个选择区域（Range对象），优化后的代码如下：

```
function saveSelection(){
    if(window.getSelection){
        /*主流的浏览器，包括chrome、Mozilla、Safari*/
        var sel = window.getSelection();
        if(sel.rangeCount > 0){
            return sel.getRangeAt(0);
        }
    }else if(document.selection){
        /*IE下的处理*/
        return document.selection.createRange();
    }
    return null;
};
```

实例（可以试试在不同浏览器下的执行，点击里面的按钮btn2）：Demo

2、Range对象的属性和方法

2.1 创建Range对象（范围）

`document.createRange()` : 用于创建一个Range对象（范围）

在IE下：

`document.body.createTextRange()` : 用于创建一个textRange对象

2.2 Range对象的属性

`endContainer` : 返回范围的结束点的Document节点，通常是文本节点。

`endOffset` : 返回`endContainer`中的结束点位置。

`startContainer` : 返回范围的开始点中的Document节点，通常是文本节点。

`startOffset` : 返回`startContainer`中的开始点位置。

`collapsed` : 用于判断范围的开始点与结束点是否处于相同的位置, 如果相同, 该属性值返回`true`, 即范围是空的或折叠的。

`commonAncestorContainer` : 范围的开始点和结束点的(即它们的祖先节点)、嵌套最深的`Document`节点。

注意:所有属性都是只读的。如果范围内存在空格, 也会计算在偏移量内。

实例(点击里面的按钮`btn3`) : Demo

2.3 Range对象的方法

2.3.1 范围选择

`selectNode(node)` : 设置该范围的边界点, 使它包含指定节点和指定节点的所有子孙节点。

`selectNodeContents(node)` : 设置该范围的边界点, 使它包含指定节点的子孙节点, 但不包含指定节点本身。

2.3.2 操作范围

`deleteContents()` : 将Range对象中所包含的内容从页面中删除

`setStart(node, index)` : 将指定节点中的某处位置指定为Range对象所代表区域的起点位置

`setEnd(node, index)` : 将指定节点中的某处位置指定Range对象所代表区域的结束位置

`setStartBefore(node)` : 将指定节点的起点位置指定为Range对象所代表区域的起点位置。

`setStartAfter()` : 将指定节点的终点位置指定为Range对象所代表区域的起点位置。

`setEndBefore()` : 将指定节点的起点位置指定为Range对象所代表区域的终点位置。

`setEndAfter()` : 将指定节点的终点位置指定为Range对象所代表区域的终点位置。

`cloneRange()` : 对当前的Range对象进行复制, 该方法返回一个复制的Range对象

`cloneContents()` : 复制当前Range对象所代表区域中的HTML代码并返回新的`DocumentFragment`对象
。

`extractContents()` : 将Range对象所代表区域中的html代码克隆到一个`DocumentFragment`对象中, 然后从页面中删除这段HTML代码

`detach()` : 释放Range对象。

`insertNode(node)` : 将指定的节点插入到某个Range对象所代表的区域中, 插入位置为Range对象所代

表区域的起点位置，如果该节点已经存在于页面中，该节点将被移动到Range对象代表的区域的起点处。

实例（关于setStart和setEnd，点击里面的按钮btn6）：Demo

实例（关于deleteContents，点击里面的按钮btn7）：Demo

实例（关于extractContents，点击里面的按钮btn8）：Demo

2.3.3 其他方法

collapse(boolean) 用于使范围的边界点重合。当为true时，将范围的结束点设为与开始点相同的值；当为false时，将范围的开始点设为与结束点相同的值。

compareBoundaryPoints(how, sourceRange)：用来比较两个Range对象，返回1, 0, -1（0表示相等，等于1时，当前范围在sourceRange之后，等于-1时，当前范围在sourceRange之前）

toString()：返回该范围表示的文档区域的纯文本内容。

(1) compareBoundaryPoints()

how的常量：

START_TO_START 用指定范围的开始点与当前范围的开始点进行比较。

START_TO_END 用指定范围的开始点与当前范围的结束点进行比较。

END_TO_END 用指定范围的结束点与当前范围的结束点进行比较。

END_TO_START 用指定范围的结束点与当前范围的开始点进行比较。

4、selection对象

selection对象可看作是Range对象的集合，包含一个或多个Range对象。

4.1 属性

anchorNode：返回范围的开始点的Document节点，和range对象的endContainer作用一样。

anchorOffset：返回startContainer中的开始点位置，和range对象的startOffset作用一样。

focusNode：返回范围的结束点的Document节点，和range对象的endContainer作用一样。

focusOffset：返回endContainer中的结束点位置，和range对象的endOffset作用一样。

isCollapsed：范围的开始点与结束点是否重叠

这是新标准中selection的属性，通过这些属性，我们省却了先获取range对象再获取偏移量和节点的繁琐。

实例（点击里面的按钮btn4）：Demo

4.2 方法

`removeAllRanges()` : 删除selection中原有的所有range

`addRange(range)` : 将新的range添加到selection中

5、HTMLInputElement的属性方法

5.1 在IE下Range对象

** (1) 属性

`htmlText` : 返回字符串, 为textRange的HTML内容, 与innerHTML作用一样, 只读

`text` : 返回字符串, 为textRange的文本内容, 相当于innerText, 可读写。

(2) 方法

`moveStart("character", index)` : 选定范围的开始点向后移动index个字符

`moveEnd("character", index)` : 选定范围的结束点向后移动index个字符

`pasteHTML()` : 黏贴HTML到一个文本节点时, 该文本节点自动分隔。

5.2 在其他主流浏览器上

(1) 属性

`selectionStart` : 获取范围的开始点, 可读写

`selectionEnd` : 获取范围的结束点, 可读写

`selectionDiraction`

(2) 方法

`select()` : 在焦点状态下, 移动光标至第一个字符后面

`setSelectionRange(start, end)` : 设置范围的开始点和结束点。

注意 : selectionStart和selectionEnd会记录元素最后一次selection的相关属性, 意思就是当元素失去焦点后, 使用selectionStart和selectionEnd仍能够获取到元素失去焦点时的值。

`setSelectionRange(start, end)` :

如果textbox没有selection时, selectionStart和selectionEnd相等, 都是焦点的位置。

在使用`setSelectionRange()`时

如果end小于start, 会讲selectionStart和selectionEnd都设置为end ;

如果start和end参数大于textbox内容的长度 (`textbox.value.length`) , start和end都会设置为value属性的长度。

3 操作富文本

使用 `document.execCommand()` 方法是与富文本编辑器交互的重要方式，它可以对文档执行预定义的命令，而且可以应用大多数格式。

语法：

```
bool = document.execCommand(aCommandName, aShowDefaultUI, aValueArgument)
```

参数说明：

- `aCommandName` : String类型，命令名称
- `aShowDefaultUI` : Boolean类型，默认为`false`，是否展示用户界面，一般设为`false`。Mozilla没有实现。
- `aValueArgument` : String类型，一些命令需要额外的参数值（比如`insertHTML`需要提供HTML内容），默认为`null`，一般不需要参数时都使用`null`。

命令介绍：

`backColor` (用法：`document.execCommand('backColor', bfalse, Color);`)

改变文档的背景颜色。在`styleWithCss`模式，它影响的是包含元素的背景。这个命令要求提供一个颜色值`<color>`作为第三个参数（Internet Explorer 使用这个命令设置文本背景色）

`bold` (用法：`document.execCommand('Bold', 'false', null);`)

对选中文本或者插入元素设置、取消粗体显示。（Internet Explorer 使用``标签 而不是``标签。）

`contentReadOnly`

转化文档进入只读或者可编辑模式。这个命令要求提供给一个boolean值给第3个参数（ie不支持）。

`copy` 用法：`document.execCommand('copy', 'false', null);`

把当前选中区域复制到系统剪贴板。由于启用这个功能的条件因浏览器不同而不同，所以使用此功能时，请先检查浏览器是否支持。

`createLink`

当有选中区域的时候，将选中区域创建为一个锚点，需要提供一个URI给第3个参数。这个URI必须至少包含一个字符，空白字符也可。（Internet Explorer会创建一个URI为空的a标签）

`cut` 用法：`document.execCommand('cut', 'false', null);`

剪切选中文本到剪切板。同`copy`一样需要开启剪切板功能。

`decreaseFontSize`

给选中文本或者插入元素添加一个`small`标签。（Internet Explorer不支持）

`delete` 删除当前选中区域

`enableInlineTableEditing`

开启或禁用表的行和列的插入删除功能（Internet Explorer不支持）

enableObjectResizing

开启或禁用图片或者其他可调整元素大小的 (resize) 功能 (Internet Explorer不支持)

fontName 用法 : document.execCommand('fontName', 'false', sFontName);

改变选中文本或者插入元素的字体。需要给第3个参数提供一个字体值 (例如 : "Arial")

fontSize 用法 : document.execCommand('fontSize', 'false', sSize|iSize);

改变选中文本或者插入元素的字体大小。需要给第3个参数提供一个数字

foreColor

改变选中文本或者插入元素的字体颜色。需要给第3个参数提供一个颜色值

formatBlock

添加一个HTML块式标签在包含当前选择的行, 如果已经存在了, 更换包含该行的块元素 (在Firefox中的b lockquote例外, 它将包含任何包含块元素)。需要提供一个标签名称作为参数 (比如 : H1、P、DL、BLC OKQUOTE等) (IE仅支持标题标签H1-H6, ADDRESS和PRE, 使用时还必须包含标签分隔符<>, 比如<H1>)

forwardDelete

删除光标所在位置的字符, 和按下删除键一样。

heading

添加一个标题标签在光标处或所选文字上。需要提供标签名称字符串作为参数 (比如 : H1、H6) (IE和Safari不支持)

hiliteColor

更改选择或插入点的背景颜色。需要提供一个颜色值作为参数。 (IE不支持)

increaseFontSize

在选择或插入点周围添加一个BIG标签 (IE不支持)

indent

取消或缩进选择或插入点所在的行。

insertBrOnReturn

控制当按下Enter键时, 是插入br标签还是把当前块元素变成两个 (IE不支持)

insertHorizontalRule

在插入点插入一个水平线 (删除选中的部分)

insertHTML

在插入点插入一个HTML字符串 (删除选中部分)。需要提供一个HTML字符串作为参数 (IE不支持)

insertImage

在插入点插入一张图片 (删除选中部分)。需要提供一个image src URI作为参数。这个URI至少包含一个字符 (空字符也行)。 (IE会创建一个值为null的链接)

insertOrderedList

在插入点或选中文字上创建一个有序列表

insertUnorderedList

在插入点或选中文字上创建一个无序列表

insertParagraph

在选择或当前行周围插入一个段落。（IE会在插入点插入一个段落并删除选中部分）

insertText

在光标插入位置插入文本内容或覆盖所选的文本内容

italic

在光标插入点开启或关闭斜体字。（IE使用``标签，而不是`<i>`）

justifyCenter

对光标插入位置或所选内容进行文字居中

justifyFull

对光标插入位置或所选内容进行文本对齐（两端对齐）

justifyLeft

对光标插入位置或所选内容进行文本左对齐

justifyRight

对光标插入位置或所选内容进行文本右对齐

outdent

对光标插入行货所选行内容减少缩进量

paste

在光标位置黏贴剪贴板的内容。（对选中内容替换）

redo

重做被撤销的操作

removeFormat

对所选内容去除所有格式（比如：加粗）。

selectAll

选中编辑区里的全部内容

strikeThrough

在光标插入点开启或关闭删除线

subscript

在光标插入点开启或关闭下角标

superscript

在光标插入点开启或关闭上角标

underline

在光标插入点开启或关闭下划线

undo

撤销最近执行的命令

unlink去除所选的锚链接的标签**useCSS**

切换使用HTML tags还是CSS来生成标记。需要一个布尔值作为参数。该属性已废除，使用styleWithCSS代替。

styleWithCSS用这个取代useCSS命令。切换使用HTML tags还是CSS来生成标记。需要一个布尔值作为参数。（**false**使用CSS, **true**使用HTML）

这个方法适用于iframe类型的编辑器，也适用于将 contenteditable 设置为true的元素：

```
// iframe
frames['richedit'].document.execCommand('bold', false, null);

// 设置contenteditable为true的元素
document.execCommand('bold', false, null);
```

3.1 与命令相关的方法

- **queryCommandEnabled()**：可以用它来检测是否可以针对当前选择的文本，或者当前插入字符所在位置执行某个命令，它接受一个参数，即要检测的命令，如果单亲编辑区允许执行传入的命令，则返回true，否则返回false

```
document.queryCommandEnabled('bold') {}
```

- **queryCommandState()**：用于确定是否已将指定命令应用到了选择的文本。比如：判断当前选择的文本是否已将转换成了粗体：

```
isBold = document.queryCommandState('bold');
```

- **queryCommandValue()**：用于取得执行命名时传入的值（即document.execCommand()的第三个参数）

4、常用案例代码

6.1 针对div (contenteditable="true") 相关操作

(1) 获取用户选择内容

```
function saveSelection(){
```

```

if(window.getSelection){
    /*主流的浏览器，包括chrome、Mozilla、Safari*/
    var sel = window.getSelection();
    if(sel.rangeCount > 0){
        return sel.getRangeAt(0);
    }
}else if(document.selection){
    /*IE下的处理*/
    return document.selection.createRange();
}
return null;
};

var selectedRange = saveSelection(); // 保存获取到的Range对象

```

注意：如果是在IE下需要获取内容，需要使用selection.text来获取。

实例（可以试试在不同浏览器下的执行，点击里面的按钮btn2）：Demo

(2) 恢复光标位置

```

function restoreSelection() {
    var selection = window.getSelection();
    if (selectedRange) {
        try {
            selection.removeAllRanges(); /*清空所有Range对象*/
        } catch (ex) {
            /*IE*/
            document.body.createTextRange().select();
            document.selection.empty();
        };
        /*恢复保存的范围*/
        selection.addRange(selectedRange);
    }
}

```

实例（点击里面的按钮btn5）：Demo

(3) 将光标移至文本最后

```

function selectAllText(elem){
    if(window.getSelection){
        elem.focus();
        var range = window.getSelection();
        range.selectAllChildren(elem);
        range.collapseToEnd();
   }else if(document.selection){
        var range = document.selection.createTextRange();
        range.moveToElementText(elem);
    }
}

```

```

    range.collapse(false);
    range.select(); /*避免产生空格*/
}
}

```

6.2 表单元素 (input、textarea) 相关操作

(1) 将光标置于表单元素的最后

```

function toTextEnd(elem){
    if(window.getSelection){
        elem.setSelectionRange(elem.value.length, elem.value.length);
        elem.focus();
    }else if(document.selection){
        /*IE下*/
        var range = elem.createTextRange();
        range.moveStart('character', elem.value.length);
        range.collapse(true);
        range.select();
    }
}

```

实例 (点击里面的按钮btn10) : Demo

(2) 选中所有文字

```

function selectAllText(elem){
    if(window.getSelection){
        elem.setSelectionRange(0, elem.value.length);
        elem.focus();
    }else if(document.selection){
        var range = elem.createTextRange();
        range.select();
    }
}

```

实例 (点击里面的按钮btn11) : Demo

(3) 获取光标位置

```

function getCursorPosition(elem){
    if(window.getSelection){
        return elem.selectionStart;
    }else if(document.selection){
        elem.focus();
        var range = document.selection.createTextRange();
        range.moveStart('character', -elem.value.length);
        return range.text.length;
    }
}

```

```
    }
    return elem.value.length;
}
```

(4) 设置光标位置

```
function setCursorPosition(elem, position){
    if(window.getSelection){
        elem.focus();
        elem.setSelectionRange(position, position);
    }else if(document.selection){
        var range = elem.createTextRange();
        range.collapse(true);
        range.moveEnd('character', position);
        range.moveStart('character', position);
        range.select();
    }
}
```

一个简单的富文本编辑器：[富文本编辑器](#)

Canvas

Canvas

二、 canvas

HTML5 `<canvas>` 元素用于图形的绘制，通过脚本(通常是JavaScript)来完成。

`<canvas>` 标签只是图形容器，您必须使用脚本来绘制图形。

要使用Canvas API，首先需要新建一个 `<canvas>` 网页元素

```
<canvas id="canvas" width="400" height="200">  
    您的浏览器不支持canvas !  
</canvas>
```

width 和 height 属性定义的画布的大小。

接着，我们可以通过标签选择获取到

```
var canvas = document.getElementById('canvas');
```

然后，创建context对象：

```
if(canvas.getContext){  
    var ctx = canvas.getContext('2d');  
}
```

调用getContext()方法时，传递一个“2d”参数，会获得一个CanvasRenderingContext2D对象，使用该对象可以在画布上绘制二维图形。我们也可将CanvasRenderingContext2D简称为“上下文对象”。

2.1 canvas的绘图用法

canvas画布提供了一个用来作图的平面空间，该空间的每个点都有自己的坐标，x表示横坐标，y表示竖坐标。原点(0, 0)位于图像左上角，x轴的正向是原点向右，y轴的正向是原点向下。

(1) 绘制矩形

- `rect(x,y,width,height)`方法用来绘制矩形。
- `fillRect(x, y, width, height)`方法用来绘制矩形，它的四个参数分别为矩形左上角顶点的x坐标、y坐标，以及

矩形的宽和高。fillStyle属性用来设置矩形的填充色。

- strokeRect(x, y, width, height)方法同样是绘制矩形，但只画线不填充，也可以说是画空心矩形。
strokeStyle属性用来设置矩形的绘制色。
- clearRect(x, y, width, height)方法用来清除某个矩形区域的内容。

(2) 绘制圆形和扇形

- arc(x, y, radius, startAngle, endAngle, anticlockwise)方法用来绘制圆形和扇形。
arc方法的x和y参数是圆心坐标，radius是半径，startAngle和endAngle则是扇形的起始角度和终止角度（以弧度表示），anticlockwise表示做图时应该逆时针画（true）还是顺时针画（false）。
- arcTo(x1,y1,x2,y2,r)方法在画布上创建介于两个切线之间的弧/曲线。
参数：x1 弧的起点的 x 坐标，y1 弧的起点的 y 坐标，x2 弧的终点的 x 坐标，y2 弧的终点的 y 坐标，r 弧的半径。

(3) 绘制文本

- fillText(string, x, y) 用来绘制文本，它的三个参数分别为文本内容、起点的x坐标、y坐标。使用之前，需用font设置字体、大小、样式（写法类似与CSS的font属性）。
strokeText(string,x,y)方法，用来添加空心字。

```
// 设置字体
ctx.font = "Bold 20px Arial";
// 设置对齐方式, start、end、right、center
ctx.textAlign = "left";
// 设置填充颜色
ctx.fillStyle = "#008600";
//设置垂直对齐方式, top、hanging、middle、ideographic、bottom
ctx.textBaseline = top;
```

我们还可以计算字体宽度(px)：

```
var name = 'aaa';
ctx.measureText(name);
```

measureText() 方法返回包含一个对象，该对象包含以像素计的指定字体宽度。

```
ctx.measureText(name).width
```

(4) 绘制路径

beginPath() 方法表示开始绘制路径，moveTo(x, y) 方法设置线段的起点，lineTo(x, y) 方法

设置线段的终点， `stroke()` 方法用来给透明的线段着色。

```
ctx.beginPath(); // 开始路径绘制
ctx.moveTo(20, 20); // 设置路径起点, 坐标为(20,20)
ctx.lineTo(200, 20); // 绘制一条到(200,20)的直线
ctx.lineWidth = 1.0; // 设置线宽
ctx.strokeStyle = '#CC0000'; // 设置线的颜色
ctx.stroke(); // 进行线的着色, 这时整条线才变得可见
moveTo()和lineTo()方法可以多次使用。
```

注意：如果使用`closePath()`方法，会自动绘制一条当前点到起点的直线，形成一个封闭图形。

`fill()`和`stroke()`分别填充当前绘图（路径）和绘制已定义的路径。

`isPointInPath()`

```
isPointInPath(x, y);
```

如果指定的点位于当前路径中，`isPointInPath()` 方法返回 `true`，否则返回 `false`。

`clip()`

`clip()`方法从原始画布中剪切任意形状和尺寸。

注意：一旦剪切了某个区域，则所有之后的绘图都会被限制在被剪切的区域内（不能访问画布上的其他区域）。

您也可以在使用 `clip()` 方法前通过使用 `save()` 方法对当前画布区域进行保存，并在以后的任意时间对其进行恢复（通过 `restore()` 方法）。

`closePath()`方法创建当前点回到起始点的路径。

其他样式属性：

- `lineCap` 设置或返回线条的结束端点样式，可能值：`butt|round|square`（平直边缘|圆形线帽|正方形线帽）
- `lineJoin` 设置或返回两条线相交时，所创建的拐角类型，可能值：`bevel|round|miter`（斜角|圆角|尖角（默认值））
- `lineWidth` 设置或返回当前的线条宽度。
- `miterLimit` 正数，设置或返回最大斜接长度。如果斜接长度超过 `miterLimit` 的值，边角会以 `lineJoin` 的 "bevel" 类型来显示。
- `fillStyle` 设置填充色
- `strokeStyle` 设置绘制线色

颜色格式：

```
//可直接用颜色名称
'red' 'green'
//十六进制颜色
```

```
'#d9d9d9'
//rgb
rgb(0,0,0)
//rgba
rgba(0,0,0,1)
```

(5) 设置渐变色

线性渐变

createLinearGradient方法用来设置渐变色。

```
var gradient = ctx.createLinearGradient(0, 0, 0, 160);
gradient.addColorStop(0, "#BABABA");
gradient.addColorStop(1, "#636363");
```

createLinearGradient()方法的参数是(x1, y1, x2, y2) , 其中x1和y1是起点坐标 , x2和y2是终点坐标。通过不同的坐标值 , 可以生成从上至下、从左到右的渐变等等。

使用方法 :

```
ctx.fillStyle = myGradient;
ctx.fillRect(10,10,200,100);
```

环形渐变

createRadialGradient()方法的参数是(x0,y0,r0,x1,y1,r1) , 其中x0 渐变的开始圆的 x 坐标 , y0 渐变的开始圆的 y 坐标 , r0 开始圆的半径 , x1 渐变的结束圆的 x 坐标 , y1 渐变的结束圆的 y 坐标 , r1 结束圆的半径。

```
var grd=ctx.createRadialGradient(75,50,5,90,60,100);
grd.addColorStop(0,"red");
grd.addColorStop(1,"white");

ctx.fillStyle=grd;
ctx.fillRect(10,10,150,100);
```

(6) 设置阴影

我们还可以设置阴影

```
ctx.shadowOffsetX = 10; // 设置水平位移
ctx.shadowOffsetY = 10; // 设置垂直位移
ctx.shadowBlur = 5; // 设置模糊度
ctx.shadowColor = "rgba(0,0,0,0.5)"; // 设置阴影颜色

ctx.fillStyle = "#CC0000";
ctx.fillRect(10,10,200,100);
```

(7) 合成

`globalAlpha` 属性设置或返回绘图的当前透明值 (`alpha` 或 `transparency`)。

`globalAlpha` 属性值必须是介于 0.0 (完全透明) 与 1.0 (不透明) 之间的数字。

`ctx.globalAlpha = number`

`globalCompositeOperation` 属性设置或返回如何将一个源 (新的) 图像绘制到目标 (已有的) 的图像上。

源图像 = 您打算放置到画布上的绘图。

目标图像 = 您已经放置在画布上的绘图。

`source-over` 默认。在目标图像上显示源图像。

`source-atop` 在目标图像顶部显示源图像。源图像位于目标图像之外的部分是不可见的。

`source-in` 在目标图像中显示源图像。只有目标图像之内的源图像部分会显示，目标图像是透明的。

`source-out` 在目标图像之外显示源图像。只有目标图像之外的源图像部分会显示，目标图像是透明的。

`destination-over` 在源图像上显示目标图像。

`destination-atop` 在源图像顶部显示目标图像。目标图像位于源图像之外的部分是不可见的。

`destination-in` 在源图像中显示目标图像。只有源图像之内的目标图像部分会被显示，源图像是透明的。

`destination-out` 在源图像之外显示目标图像。只有源图像之外的目标图像部分会被显示，源图像是透明的。

`lighter` 显示源图像 + 目标图像。

`copy` 显示源图像。忽略目标图像。

`xor` 使用异或操作对源图像与目标图像进行组合。

2.2 绘制图像

Canvas API 允许将图像文件插入画布。

我们可以使用 `Image` 对象来加载图片，然后绘制：

```
var image = new Image();
image.onload = function(){
    ctx.drawImage(image, 0, 0);
};
image.src='new.jpg';
```

drawImage方法

```
drawImage(img,x,y)
drawImage(img,x,y,width,height)
drawImage(img,sx,sy,switch,sheight,x,y,width,height)
```

参数：

`img` 规定要使用的图像、画布或视频；`sx` 可选，开始剪切的 `x` 坐标位置；`sy` 可选，开始剪切的 `y` 坐标位置；`swidth` 可选，被剪切图像的宽度；`sheight` 可选，被剪切图像的高度；`x` 表示在画布上放置图像的 `x` 坐标位置；`y` 在画布上放置图像的 `y` 坐标位置；`width` 可选，表示要使用的图像的宽度（伸展或缩小图像）；`height` 可选，表示要使用的图像的高度（伸展或缩小图像）。

图像平铺

```
createPattern(image, type)
```

参数：

`type`: `no-repeat`:不平铺；`repeat-x`:横方向平铺；`repeat-y`:纵方向平铺；`repeat`:全方向平铺

2.3 像素处理

通过 `getImageData` 方法和 `putImageData` 方法，可以处理每个像素，进而操作图像内容。

`getImageData()` 方法可以用来读取Canvas的内容，返回一个对象，包含了每个像素的信息。

```
var imageData = ctx.getImageData(x, y, w, h)
```

参数：`x`是canvas的X轴坐标，`y`是canvas的Y轴坐标，`w`是宽度，`h`是高度。

`getImageData()`方法返回一个像素颜色数组（该数组的长度等于图像的像素宽度图像的像素高度4，每个值的范围是0-255，可读写），`imageData`的属性`data`就是指向它，顺序是所取像素范围的从左到右，从上到下，数组的元素是（所有图形，包括图片，和绘制的图形）每个像素的rgba：

```
[r1, g1, b1, a1, r2, g2, b2, a2...]
```

`putImageData()` 方法可将数组内容重新绘制到Canvas上。

```
putImageData(imgData, x, y, dirtyX, dirtyY, dirtyWidth, dirtyHeight);
```

参数：`imgData` 规定要放回画布的 `ImageData` 对象；`x` 是`ImageData` 对象左上角的 `x` 坐标，以像素计；`y` 是 `ImageData` 对象左上角的 `y` 坐标，以像素计；`dirtyX` 可选，水平值（`x`），以像素计，在画布上放置图像的位置；`dirtyY` 可选，垂直值（`y`），以像素计，在画布上放置图像的位置；`dirtyWidth` 可选，在画布上绘制图像所使用的宽度；`dirtyHeight` 可选，在画布上绘制图像所使用的高度。

当然，我们也可以创建一个空白的`ImageData`对象。

```
var imgData=context.createImageData(width,height);
```

也可创建与指定的另一个`ImageData`对象尺寸相同的新`ImageData`对象（其不会复制图像数据）：

```
var imgData=context.createImageData(imageData);
```

ImageData对象有data属性，它包含 color/alpha 信息的数组。

2.4 保存与恢复

save()方法用于保存上下文环境，restore()方法用于恢复到上一次保存的上下文环境。

2.5 保存图像

我们可以使用toDataURL()方法将Canvas数据重新转化成图像文件形式：

```
canvas.toDataURL('image/png')
```

上面的代码将Canvas数据，转化成PNG data URI。

2.6 转换

(1) scale()

scale() 方法缩放当前绘图至更大或更小。

```
scale(scalewidth, scaleheight)
```

参数：scalewidth 缩放当前绘图的宽度；scaleheight 缩放当前绘图的高度。

注意：如果您对绘图进行缩放，所有之后的绘图也会被缩放。定位也会被缩放。如果您 scale(2,2)，那么绘图将定位于距离画布左上角两倍远的位置。

(2) rotate()

rotate() 方法旋转当前的绘图。

```
rotate(angle)
```

参数：angle 旋转角度，以弧度计。

注意：旋转只会影响到旋转完成后的绘图。

(3) translate()

translate() 方法重新映射画布上的 (0,0) 位置。

```
translate(x, y)
```

参数：x 添加到水平坐标 (x) 上的值；y 添加到垂直坐标 (y) 上的值。

(4) transform()

```
transform(a, b, c, d, e, f)
```

参数：a 水平缩放绘图，b 水平倾斜绘图，c 垂直倾斜绘图，d 垂直缩放绘图，e 水平移动绘图，f 垂直移动绘图。

(5) setTransform()

setTransform() 方法把当前的变换矩阵重置为单位矩阵，然后以相同的参数运行 transform()。

```
setTransform(a, b, c, d, e, f)
```

参数：a 水平缩放绘图，b 水平倾斜绘图，c 垂直倾斜绘图，d 垂直缩放绘图，e 水平移动绘图，f 垂直移动绘图。

注意：该变换只会影响 setTransform() 方法调用之后的绘图。

2.7 动画

我们可以使用 `setInterval()` 和 `setTimeout()` 来产生动画效果。

还可以使用 `requestAnimationFrame()` 来制作动画。

`requestAnimationFrame()` 函数是全局函数。

考虑兼容，如下：

```
var requestAnimFrame = function() {
  return window.requestAnimationFrame || window.webkitRequestAnimationFrame ||
  window.mozRequestAnimationFrame || window.oRequestAnimationFrame || window.msRe
  questAnimationFrame || function(a) {window.setTimeout(a, 1e3 / 60, (new Date).g
  etTime());};
}();
```

使用方法：

```
function step(){
  requestAnimationFrame(step);
}
```

`requestAnimationFrame()` 方法会返回一个 `requestID`，是一个长整型非零值，作为一个唯一的标识符，可将该值作为参数传递给 `window.cancelAnimationFrame()` 来取消这个函数。

```
cancelAnimationFrame(requestID);
```

兼容性代码：

```
var cancelAnimFrame = function() {
  return window.cancelAnimationFrame || window.webkitCancelAnimationFrame || wi
  ndow.mozCancelAnimationFrame || function(id) { clearTimeout(id); };
```

canvas

```
}();
```

离线应用

离线应用

离线Web应用，是指在设备不能上网的情况下仍然可以运行的应用。

要开发离线Web应用，需要下列几个步骤：

- 检测设备是否能上网
- 应用能访问一定的资源（图像、JavaScript、CSS等）
- 有一块本地空间用于保存数据

1.1 离线检测

HTML5定义了一个 `navigator.onLine` 属性，用来检测设备是在线还是离线，为true时表示设备能上网，否则表示设备离线。

检测代码：

```
if (navigator.onLine){
    // 正常工作
} else {
    // 设备已离线
}
```

除了 `navigator.onLine` 属性，HTML5还为检测网络是否可用提供了两个事件：`online` 和 `offline`。

- `online`：当网络从离线变为在线时触发
- `offline`：当网络从在线变为离线时触发

```
window.addEventListener('online', function(){
    // 在线
});
window.addEventListener('offline', function(){
    // 离线
});
```

为了检测应用是否离线，在页面加载后，最好先通过 `navigator.onLine` 取得初始的状态，然后再通过 `online` 和 `offline` 两个事件检测网络连接状态是否变化。

1.2 应用缓存

HTML5的应用缓存（application cache），简称：appcache，是专门为开发离线Web应用而设计的。

Appcache就是从浏览器的缓存中分出来的一块缓存区。要想在这个缓存中保存数据，可以使用一个描述文件

(manifest file) , 列出要下载和缓存的资源。

`manifest` 文件可分为三个部分 :

- CACHE MANIFEST - 在此标题下列出的文件将在首次下载后进行缓存
- NETWORK - 在此标题下列出的文件需要与服务器的连接 , 且不会被缓存
- Fallback - 在此标题下列出的文件规定当页面无法访问时的回退页面 (比如 404 页面)

```
CACHE MANIFEST
```

CACHE :

```
# 需要缓存的列表
test.css
test.jpg
test.js
```

NETWORK :

```
# 不需要缓存的
test2.jpg
```

FALLBACK :

```
# 访问缓存失败后, 备用访问的资源, 第一个是访问源, 第二个是替换文件*.html /offline.html
2.jpg/3.jpg
```

是注释的意思。

注意 : 必须以 `CACHE MANIFEST` 开头

可以在 `<html>` 中的 `manifest` 属性中指定离线文件的路径 , 就可以将描述文件与页面关联起来了。

```
<html manifest="offline.manifest">
```

注意 : `manifest` 文件的 `MIME` 类型必须是 "text/cache-manifest"。

描述文件的扩展名以前推荐用 `manifest` , 现在推荐用 `appcache`

1.2.1 applicationCache 对象

JavaScript 提供了相应的 API 让我们去监控描述文件的状态 , 这个 API 的核心是 `applicationCache` 对象 (全局) , 这个对象有一个 `status` 属性 , 属性的值是常量 , 可能值如下 :

- 0 : 无缓存 (UNCACHE) , 即没有与页面相关的应用缓存
- 1 : 闲置 (IDLE) , 即应用缓存未得到更新
- 2 : 检查中 (CHECKING) , 即正在下载描述文件并检测更新
- 3 : 下载中 (DOWNLOADING) , 即应用缓存正在下载描述文件中指定的资源

- 4：更新完成（UPDATEREADY），即应用缓存已经更新了资源，而且所有资源都已下载完毕，可以通过 swapCache() 来使用了
- 5：废弃（IDLE），即应用缓存的描述文件已经不存在了。

1.2.2 应用缓存相关事件

应用缓存还有很多相关的事件，表示其状态的改变：

- checking：在浏览器为应用缓存查找更新时触发
- error：在检测更新或下载资源期间发生错误时触发
- noupdate：在检查描述文件发生文件无变化时触发
- downloading：在开始下载应用缓存资源时触发
- progress：在文件下载应用缓存的过程中持续不断的触发
- updateready：在页面新的应用缓存下载完毕且通过 swapCache() 使用时触发
- cached：在应用缓存完整可用时触发

```
applicationCache.addEventListener('updateready', function(){
  if(applicationCache == 4) { //4等于application.UPDATEREADY
    applicationCache.swapCache(); //使用新版本资源
    window.location.reload(); // 刷新页面
  }
}, false);
```

使用 on 方式也可以：

```
applicationCache.ondownloading = function(){}
```

我们可以通过 update() 方法来手动检查更新：

```
applicationCache.update();
```

1.3 更新问题

这里还要提一个关键问题，在线状态下，浏览器会检测描述文件是否有更新，而并不会检测描述文件内的相应文件是否更新。

比如有一个描述文件如下：

```
CACHE MANIFEST
# Version 1

test.js
```

当你修改了test.js里的内容时，浏览器并不会在检测描述文件时检测到这文件的变化，应该改成这样：

```
CACHE MANIFEST  
# Version 2 (更改这个数字让浏览重新下载描述文件)  
test.js
```

客户端存储 (Cookie、Storage、IndexedDB)

离线应用与客户端存储

Web应用允许使用浏览器提供的API实现将数据存储到用户的电脑上。

1、cookie

cookie是指Web浏览器存储的少量数据，每个Cookie的大小一般不能超过4KB。

Cookie保存以下几方面的信息。

- Cookie的名字
- Cookie的值
- 到期时间
- 所属域名（默认是当前域名）
- 生效的路径（默认是当前网址）

我们可以通过特殊的格式的字符串形式来读写Document对象的cookie属性。

```
document.cookie //读取当前页面的所有cookie
```

我们可以通过检查navigator.cookieEnabled属性来检测当前cookie是否启用。若值为true，表示cookie是启用的。

1.1 cookie属性：有效期和作用域

cookie默认的有效期很短暂，它只能持续在Web浏览器的会话期间，一旦用户关闭浏览器，cookie保存的数据就会丢失。

注意：cookie的有效期和sessionStorage的有效期是有区别的：cookie的作用域并不是局限在浏览器的单个窗口中，它的有效期和整个浏览器进程而不是单个浏览器窗口的有效期一致。

cookie的作用域是通过文档源和文档路径来确定的。

1.2 保存cookie

document.cookie属性是可写的，可以通过它为当前网站添加Cookie。

```
document.cookie = 'user=TG';
```

Cookie的值必须写成key=value的形式。注意，等号两边不能有空格。另外，写入Cookie的时候，必须对分号、逗号和空格进行转义（它们都不允许作为Cookie的值），这可以用encodeURIComponent方法进行编码，读取时再采用decodeURIComponent方法解码。

但是，document.cookie一次只能写入一个Cookie，而且写入并不是覆盖，而是添加。

```
document.cookie = 'test1=hello';
document.cookie = 'test2=world';
document.cookie
// test1=hello; test2=world
```

如果要延长cookie的有效期，就需要设置max-age属性来指定cookie的有效期（单位是秒）

```
document.cookie = 'user=TG;max-age=60*60*24'; //将有效期设置为一天
```

1.3 读取cookie

使用JavaScript表达式来读取cookie属性时，其返回的值是一个字符串，该字符串都是由一系列名/值对组成，不同名/值对之间通过“分号和空格”分开，其内容包含了所有作用在当前文档的cookie。一般情况下，我们会采用split()方法将cookie值中的名/值对都分离出来。当然，如果之前进行了一系列的编码，就需要先解码再分离。

```
function getCookie(cname)
{
    var name = cname + "=";
    var ca = document.cookie.split(';');
    for(var i=0; i<ca.length; i++)
    {
        var c = ca[i].trim();
        if (c.indexOf(name)==0)
        {
            return c.substring(name.length,c.length);
        }
    }
    return "";
}
```

1.4 其他属性

除了Cookie本身的内容，还有一些可选的属性也是可以写入的，它们都必须以分号开头。

```
Set-Cookie: value[; expires=date][; domain=domain][; path=path][; secure]
```

上面的Set-Cookie字段，用分号分隔多个属性：

(1) value属性

value属性是必需的，它是一个键值对，用于指定Cookie的值。

(2) expires属性

expires属性用于指定Cookie过期时间。它的格式采用Date.toUTCString()的格式。

```
expires=Thu, 01 Jan 1970 00:00:00 GMT
```

如果不设置该属性，或者设为null，Cookie只在当前会话（session）有效，浏览器窗口一旦关闭，当前Session结束，该Cookie就会被删除。

浏览器根据本地时间，决定Cookie是否过期，由于本地时间是不精确的，所以没有办法保证Cookie一定会在服务器指定的时间过期。

(3) domain属性

domain属性指定Cookie所在的域名，比如example.com或.example.com（这种写法将对所有子域名生效）、subdomain.example.com。

如果未指定，默认为设定该Cookie的域名。所指定的域名必须是当前发送Cookie的域名的一部分，比如当前访问的域名是example.com，就不能将其设为google.com。只有访问的域名匹配domain属性，Cookie才会发送到服务器。

(4) path属性

path属性用来指定路径，必须是绝对路径（比如/、/mydir），如果未指定，默认为请求该Cookie的网页路径。只有path属性匹配向服务器发送的路径，Cookie才会发送。这里的匹配不是绝对匹配，而是从根路径开始，只要path属性匹配发送路径的一部分，就可以发送。比如，path属性等于/blog，则发送路径是/blog或者/blogroll，Cookie都会发送。path属性生效的前提是domain属性匹配。

(5) secure

secure属性用来指定Cookie只能在加密协议HTTPS下发送到服务器。

该属性只是一个开关，不需要指定值。如果通信是HTTPS协议，该开关自动打开。

(6) max-age

max-age属性用来指定Cookie有效期，比如 $60 * 60 * 24 * 365$ （即一年 $31536e3$ 秒）。

2.5 cookie的限制

浏览器对Cookie数量的限制，规定不一样。目前，Firefox是每个域名最多设置50个Cookie，而Safari和Chrome没有域名数量的限制。

所有Cookie的累加长度限制为4KB。超过这个长度的Cookie，将被忽略，不会被设置。

由于Cookie可能存在数量限制，有时为了规避限制，可以将cookie设置成下面的形式。

```
name=a=b&c=d&e=f&g=h
```

上面代码实际上是设置了一个Cookie，但是这个Cookie内部使用&符号，设置了多部分的内容。因此，读取这个Cookie的时候，就要自行解析，得到多个键值对。这样就规避了cookie的数量限制。

2、localStorage和sessionStorage

`localStorage` 和 `sessionStorage` 这两个属性都代表同一个Storage对象（一个持久化关联数组，数组使用字符串来索引，存储的值都是字符串形式的）。

2.1 存储有效期和作用域

`localStorage` 和 `sessionStorage` 的区别在于存储的有效期和作用域的不同。

2.1.1 localStorage

通过 `localStorage` 存储的数据是永久性的，除非Web应用刻意删除存储的数据或用户通过设置浏览器设置来删除，否则数据将一直保留在用户的电脑里，永不过期。

`localStorage` 的作用域是限定在文档源 (document origin) 级别。

同源的文档间共享同样的 `localStorage` 数据。

2.1.2 sessionStorage

`sessionStorage` 的作用域同样是限定在文档源中，不过它被限定在窗口中。也就是说，如果同源的文档在不同的浏览器标签页中，那它们互相之间拥有的是各自的 `sessionStorage` 数据，无法共享。

注意：基于窗口作用域的 `sessionStorage` 指的窗口只是顶级窗口。如果一个浏览器标签页包含多个

`<iframe>` 元素，它们包含的文档是同源的，两者之间的`sessionStorage`是可共享的。

2.2 存储API

我们可以将 `localStorage` 和 `sessionStorage` 当做普通的JavaScript对象：通过设置属性来存储字符串值，查询该属性来读取该值。

```
localStorage.user = 'TG';
```

当然，这两个对象也提供了对应的存储 (`setItem()`) 和读取 (`getItem()`) 的方法。

```
localStorage.setItem('user', 'TG'); //存储一个以“user”的名字存储的数值。
```

```
localStorage.getItem('user'); //读取
```

同样，`sessionStorage` 也有这两个方法。

```
sessionStorage.setItem('user', 'TG');
sessionStorage.getItem('user');
```

还可以使用 `removeItem()` 和 `clear()` 方法来删除。

```
localStorage.removeItem('user'); //删除名为“user”的数据。
localStorage.clear(); //删除所有存储的数据
```

```
sessionStorage.removeItem('user');
sessionStorage.clear();
```

遍历存储数据。

```
for(var i=0; i < localStorage.length; i++){
    var name = localStorage.key(i); //获取第i对的名字
    console.log(localStorage.getItem(name)); //获取该对的值
}
```

其中的 `key` 方法，根据位置（从0开始）获得键值。

```
localStorage.key(1);
```

2.3 storage事件

当储存的数据发生变化时，会触发storage事件。我们可以指定这个事件的回调函数。

```
window.addEventListener("storage",onStorageChange);
```

回调函数接受一个event对象作为参数。这个event对象的key属性，保存发生变化的键名。

```
function onStorageChange(e) {
    console.log(e.key);
}
```

除了key属性，event对象的属性还有三个：

- `oldValue`：更新前的值。如果该键为新增加，则这个属性为null。
- `newValue`：更新后的值。如果该键被删除，则这个属性为null。
- `url`：原始触发storage事件的那个网页的网址。

值得特别注意的是，该事件不在导致数据变化的当前页面触发。如果浏览器同时打开一个域名下面的多个页面，当其中的一个页面改变`sessionStorage`或`localStorage`的数据时，其他所有页面的storage事件会被触发，而原生页面并不触发storage事件。可以通过这种机制，实现多个窗口之间的通信。所有浏览器之中，只有IE浏览器除外，它会在所有页面触发storage事件。

3、客户端数据库 (IndexedDB)

IndexedDB（对象数据库）可以说是浏览器端数据库，可以被网页脚本程序创建和操作。它允许储存大量数据，提供查找接口，还能建立索引。

在IndexedDB API中，一个数据库就是一个命名对象仓库（object store）的集合，对象存储区存储的是对象。

IndexedDB特点：

- 键值对存储：在对象仓库中，数据以“键值对”的形式保存，每一个数据都有对应的键名，键名是独一无二的，不能有重复，否则会抛出一个错误。
- 同域限制：IndexedDB数据库的作用域是限制在包含它们的文档源中，每一个数据库对应创建该数据库的域

名，两个同源的Web页面互相之间可以访问对方的数据，但非同源的页面就不行。

- 支持事务：IndexedDB支持事务，这就是说对数据库的查询和更新都是包含在一个事务（transaction）中，以此来确保这些操作要么是一起成功，要么是一起失败，并且永远不会让数据库出现更新到一半的情况。
- 异步：IndexedDB的操作不会阻塞浏览器的UI主线程。
- 储存空间大：IE的储存上限是250MB，Chrome和Opera是剩余空间的某个百分比，Firefox则没有上限。

3.1 检测浏览器是否支持IndexedDB API

```
if('indexedDB' in window){
    //支持
}else{
    //不支持
}
```

3.2 访问数据库

要异步访问数据库，就要调用 window 对象 indexedDB 属性的 open() 方法

```
var request = indexedDB.open(name[, version])
```

indexedDB.open方法可传输入两个参数：name是数据库名称，必填；version是数据库版本，是一个大于0的正整数（0将报错）。

open方法返回一个 IDBRequest 对象 (IDBOpenDBRequest)，

注意：如果数据库存在，将打开数据库，否则，则会新建该数据库。如果省略第二个参数，则会自动创建版本为1的该数据库。

当打开数据时，有可能触发4种事件：

success：打开成功。

error：打开失败。

upgradeneeded：第一次打开该数据库，或者数据库版本发生变化。

blocked：上一次的数据库连接还未关闭。

第一次打开数据库时，会先触发upgradeneeded事件，然后触发success事件。

```
request.onupgradeneeded = function(e){}

request.onsuccess = function(e){
    db = e.target.result;
}
```

回调函数接受一个事件对象event作为参数，它的target.result属性就指向打开的IndexedDB数据库。

3.3 IndexedDB实例对象的方法

3.3.1 createObjectStore()方法

createObjectStore()方法用于创建存放数据的“对象仓库”(object store)。

```
db.createObjectStore(name[, options]);
```

参数说明：

参数name是对象仓库的名字；options是可选参数，用来设置对象仓库的属性，可配置两个属性：keyPath和autoIncrement，分别表示每条记录的键名和是否使用自动递增的整数作为键名，默认为false。

```
db.createObjectStore('db1', {keyPath: 'user'});
db.createObjectStore('db2', {autoIncrement: true});
```

由于对象仓库的名字具有唯一性（当创建已存在的数据库时，会报错），所以在创建对象仓库时，我们有必要检测对象仓库是否已存在：

```
db.objectStoreNames.contains(name)
```

objectStoreNames属性返回一个DOMStringList对象，里面包含了当前数据库所有“对象仓库”的名称。可以使用DOMStringList对象的contains方法，检查数据库是否包含某个“对象仓库”。

3.3.2 transaction方法

创建了数据库，当然要使用它，不过数据库的更新、读取和删除是建立在事务的基础上的，所以我们首先要创建一个事务：

```
var t = db.transaction(array, type)
```

transaction()方法接受两个参数：参数array是一个数组，包含了所要使用的对象仓库，通常是一个；参数type是一个表示操作类型的字符串，目前只有两种类型：readonly（只读）和readwrite（读写）。

```
t = db.transaction(['db1', 'readwrite');
```

transaction()方法返回一个事务对象，该对象的objectStore()方法用于获取指定的对象仓库：

```
var store = t.objectStore('db1');
```

事务对象有三个监听事件：

abort：事务中断。

complete：事务完成。

error：事务出错。

假如事务完成时：

```
t.oncomplete =function(e){}
```

3.3.3 数据操作

下面的方法都是在事件对象上。

(1) add()方法

add()方法用来添加数据

```
var add = store.add(data, key)
```

参数说明：参数data是所要添加的数据；参数key是这条数据对应的键名（key）。

add()方法是异步的，有success和error事件：

```
add.onsuccess = funciton(e){}
add.onerror = function(e){}
```

(2) get()方法

get()方法用来读取数据，它的参数是键名

```
store.get(key)
```

get方法也是异步的，也有success和error事件。

(3) put()方法

put()方法用来更新数据，与add()方法类似：

```
var update = store.put(data, key)
```

(4) delete()方法

delete()方法用来删除数据，它的参数是键名：

```
var delete = store.delete(key)
```

delete方法也是异步的，也有success和error事件。

(5) openCursor()方法

openCursor()方法用来遍历数据：

```
var cursor = store.openCursor()
```

openCursor方法也是异步的，也有success和error事件。

```
cursor.onsuccess = function(e){
  var res = e.target.result;
  console.log('key', res.key);
  console.log('data', res.value);
  res.continue()
}
```

e.target.result属性指向当前数据对象。当前数据对象的key和value分别返回键名和键值（即实际存入的数据）。continue方法将光标移到下一个数据对象，如果当前数据对象已经是最后一个数据了，则光标指向null。openCursor方法还可以接受第二个参数，表示遍历方向，默认值为next，其他可能的值为prev、nextunique和prevunique。后两个值表示如果遇到重复值，会自动跳过。

3.3.4 createIndex()方法

createIndex()方法用来创建索引：

```
createIndex(index, name, options)
```

createIndex方法接受三个参数，第一个是索引名称，第二个是建立索引的属性名，第三个是参数对象，用来设置索引特性。unique表示索引所在的属性是否有唯一值。

3.3.5 index方法

index()方法用于从对象仓库返回指定的索引。

```
var index = store.index(index);
var data = index.get(name)
```

注意：get方法有可能取回多个数据对象，因为name属性没有唯一值。

4、同源策略

浏览器的同源政策规定，两个网址只要域名相同和端口相同，就可以共享Cookie。

注意：这里不要求协议相同。也就是说，

<http://example.com>设置的Cookie，可以被<https://example.com>读取。

HTML5 API

HTML5 API

在这一节中收集了HTML5 API，后续会持续添加。

Video/Audio

多媒体 (video和audio)

一、音频 (audio) 和视频 (video)

在HTML5中引入了 `<audio>` 和 `<video>` 元素，用来在HTML文档中嵌入音频和视频：

```
<audio src="m.mp3"/>
<video src="m.mp4" width=320 height=400 />
```

由于不同的浏览器对标准音频和视频的编解码支持上并不一致，所以通常需要使用 `<source>` 元素来为指定不同格式的媒体源：

```
<audio id="music">
  <source src="m.mp3" type="audio/mpeg">
  <source src="m.ogg" type="audio/ogg;codec='vorbis'">
</audio>
```

`<source>` 元素没有任何内容：没有闭合的 `</source>` 标签，也需要使用 “`/>`” 来结束它们。

我们可以在 `<audio>` 和 `</audio>` (或 `<video>` 和 `</video>`) 标签之间插入文本内容，如果浏览器支持 `<audio>` 和 `<video>` 元素，将不会渲染文本内容；而如果浏览器不支持时，则会将它们渲染出来。

```
<audio id="music">
  <source src="m.mp3" type="audio/mpeg">
  <source src="m.ogg" type="audio/ogg;codec='vorbis'">
  您的浏览器不支持audio标签。
</audio>
```

1.1 HTML中媒体的属性

(1) 视频 (`<video>`)

autoplay	autoplay	如果出现该属性，则视频在就绪后马上播放。
controls	controls	如果出现该属性，则向用户显示控件，比如播放按钮。
height	pixels	设置视频播放器的高度。
width	pixels	设置视频播放器的宽度。
loop	loop	如果出现该属性，则循环播放。
muted	muted	如果出现该属性，视频的音频输出为静音。
poster	URL	规定视频正在下载时显示的图像，直到用户点击播放按钮。 <code>preload</code>
<code>auto/metadata/none</code>		如果出现该属性，则视频在页面加载时进行加载，并预备播放。如果使用 “ <code>autoplay</code> ”，则忽略该属性。

src	URL	要播放的视频的 URL。
-----	-----	--------------

(2) 音频(`<audio>`)

autoplay	autoplay	如果出现该属性，则视频在就绪后马上播放。
controls	controls	如果出现该属性，则向用户显示控件，比如播放按钮。
loop	loop	如果出现该属性，则当媒介文件完成播放后再次开始播放。
muted	muted	如果出现该属性，视频的音频输出为静音。
preload	auto/metadata/none	如果出现该属性，则视频在页面加载时进行加载，并预备播放。如果使用 "autoplay"，则忽略该属性。
src	URL	要播放的视频的 URL。

1.2 用JavaScript操作音频和视频

audio可以通过new创建Audio对象

```
var music = new Audio('m.mp3');
```

还可以通过标签获取 `<audio>` 或 `<video>` 元素，即获得HTMLVideoElement和HTMLAudioElement对象：

```
var music = document.getElementById('music');
```

检测浏览器是否支持

```
var hasAudio = !! (music.canPlayType); //通过!!运算符将结果转换成布尔值
```

(1) 属性

只读属性：

duration	整个媒体文件的播放时长，以秒为单位。如果无法获取时长，则返回NaN
paused	如果媒体文件当前被暂停，则返回 <code>true</code> 。如果还未开始播放，默认返回 <code>true</code>
ended	如果媒体文件已经播放完毕，则返回 <code>true</code>
startTime	返回最早的播放起始时间，一般是 <code>0.0</code> ，除非是缓冲过的媒体文件，并且一部分内容已经不在缓冲区
error	在发生了错误的情况下返回的错误代码
currentSrc	以字符串形式返回当前正在播放或已加载的文件。对于浏览器在source元素中选择的文件
seeking	如果播放器正在跳到一个新的播放点，那 <code>seeking</code> 的值为 <code>true</code> 。
initialTime	指定了媒体的开始时间，单位为秒

可读写属性：

autoplay	将媒体文件设置为创建后自动播放，或者查询是否已设置为autoplay
loop	返回是否循环播放，或设置循环播放（或者不循环播放）
currentTime	指定了播放器应该跳过播放的时间（单位为秒）。在播放过程中，可设置currentTime属性来进行定点播放。
controls	显示或隐藏用户控制界面，或者查询用户控制界面当前是否可见
volume	在0.0到1.0之间设置音频音量的相对值，或者查询当前音量相对值
muted	布尔值，设置静音或者消除静音，或者检测当前是否为静音
autobuffer	通知播放器在媒体文件开始播放前，是否惊醒缓冲加载。如果已设置为autoplay，则忽略此特性
playbackRate	用于指定媒体播放的速度。1.0表示正常速度，大于1则表示“快进”，0~1之间表示“慢放”，负值表示回放。

三个特殊属性：

played	返回已经播放的时间段
buffered	返回当前已经缓冲的时间段
seekable	返回当前播放器需要跳到的时间段
played、buffered和seekable	都是TimeRanges对象，每个对象都有一个length属性以及start(index)和end(index)方法，前者表示当前一个时间段，后者分别返回当前时间段的起始时间点和结束时间点（单位为秒）。

还有另外三个属性：

readyState、networkState和error，它们包含 `<audio>` 和 `<video>` 元素的一些状态细节。每个属性都是数字类型的，而且为每个有效值都定义了对应的常量。

readyState属性

`readyState` 属性指定当前已经加载了多少媒体内容，只读属性

HAVE NOTHING (数字值为0)	：没有获取到媒体的任何信息，当前播放位置没有可播放数据。
HAVE_METADATA (数字值为1)	：已经获取到足够的媒体数据，但是当前播放位置没有有效的媒体数据（也就是说，获取到的媒体数据无效，不能播放）。
HAVE_CURRENT_DATA (数字值为2)	：当前播放位置已经有数据可以播放，但没有获致到可以让播放器前进的数据。当媒体为视频时，意思是当前帧的数据已获取，但没有获取到下一帧的数据，或者当前帧已经是播放的最重一帧。
HAVE_FUTURE_DATA (数字值为3)	：当前播放位置已经有数据可以播放，而且也获取到了可以让播放器前进的数据。当媒体为视频时，意思是当前帧的数据已获取，而且也获取到了下刺目版数据，当前帧是播放的最后一帧时，readyState属性不可能为HAVE_FUTURE_DATA。 HAVE_ENOUGH_DATA (数字值为4) : 当前播放位置已经有数据可以播放，同时也获取到了可以让播放器前进的数据，而且浏览器确认媒体以某一种速度进行加载，可以保证有足够的后续数据进行播放。

networkState属性

networkState属性读取当前的网络状态，共有如下所示的4个可能值：

- NETWORK_EMPTY (数字值为0) : 元素牌初始状态。

- NETWORK_IDLE (数字值为1) : 浏览器已选择好用什么编码格式来播放媒体 , 但尚未建立网络连接。
- NETWORK_LOADING (数字值为2) : 媒体数据加载中。
- NETWORK_NO_SOURCE (数字值为3) : 没有支持的编码格式 , 不执行加载。

error属性

当在加载媒体或者播放媒体过程中发生错误时 , 浏览器就会设置

- MEDIA_ERR_ABORTED(数字值为1) : 媒体数据的下载过程由于用户的操作原因而被中止。
- MEDIA_ERR_NETWORK(数字值为2) : 确认媒体资源可用 , 但是在下载时出现网络错误 , 媒体数据下载过程被中止。
- MEDIA_ERR_DECODE(数字值为3) : 确认媒体资源可能 , 但是解码时发生错误。
- MEDIA_ERR_SRC_NOT_SUPPORTED (数字值为4) : 媒体格式不被支持。

video的额外属性 :

poster 在视频加载完成之前 , 代表视频内容的图片的URL地址。该特性可读可修改
width、height 读取或设置显示尺寸。如果大小不匹配视频本身 , 会导致边缘出现黑色条状区域
videoWidth、videoHeight 返回视频的固有或自使用宽度和高度。只读

(2) 方法

canPlayType(type) 方法将媒体的MIME类型作为参数 , 用来测试浏览器是否支持指定的媒体类型。如果它不能播放该类型的媒体文件 , 将返回一个空的字符串 ; 反之 , 它会返回一个字符串 : “maybe” 或 “probably” 。

```
var a = new Audio();
if(a.canPlayType('audio/wav')){
  a.src = 'm.wav';
  a.play();
}
```

其他方法 :

- play() 控制媒体开始播放
- pause() 暂停媒体播放
- load() 重新加载src指定的资源

(3) 事件

audio元素和video元素加载音频和视频时 , 以下事件按次序发生。

loadstart : 开始加载音频和视频。

durationchange : 音频和视频的duration属性（时长）发生变化时触发，即已经知道媒体文件的长度。如果没有指定音频和视频文件，duration属性等于NaN。如果播放流媒体文件，没有明确的结束时间，duration属性等于Inf (Infinity)。

loadedmetadata : 媒体文件的元数据加载完毕时触发，元数据包括duration（时长）、dimensions（大小，视频独有）和文字轨。

loadeddata : 媒体文件的第一帧加载完毕时触发，此时整个文件还没有加载完。

progress : 浏览器正在下载媒体文件，周期性触发。下载信息保存在元素的buffered属性中。

canplay : 浏览器准备好播放，即使只有几帧，readyState属性变为CAN_PLAY。

canplaythrough : 浏览器认为可以不缓冲（buffering）播放时触发，即当前下载速度保持不低于播放速度，readyState属性变为CAN_PLAY_THROUGH。

除了上面这些事件，audio元素和video元素还支持以下事件。

abort 播放中断
emptied 媒体文件加载后又被清空，比如加载后又调用load方法重新加载。
ended 播放结束
error 发生错误。该元素的error属性包含更多信息。
pause 播放暂停
play 暂停后重新开始播放
playing 开始播放，包括第一次播放、暂停后播放、结束后重新播放。
ratechange 播放速率改变
seeked 搜索操作结束
seeking 搜索操作开始
stalled 浏览器开始尝试读取媒体文件，但是没有如预期那样获取数据
suspend 加载文件停止，有可能是播放结束，也有可能是其他原因的暂停
timeupdate 网页元素的currentTime属性改变时触发。
volumechange 音量改变时触发（包括静音）。
waiting 由于另一个操作（比如搜索）还没有结束，导致当前操作（比如播放）不得不等待。

Geolocation API

Geolocation API (地理位置)

1、地理位置 (Geolocation API)

Geolocation接口用于获取用户的地理位置。它使用的方法基于GPS或者其他机制（比如IP地址、Wifi热点、手机基站等）。

1.1 检测地理位置是否可用

```
if('geolocation' in navigator){
    //地理位置可用
} else{
    //地理位置不可用
}
```

1.2 获取当前定位

getCurrentPosition()函数可用来获取设备当前位置：

```
navigator.geolocation.getCurrentPosition(success,error,options);
```

参数说明：

- success：成功得到位置信息时的回调函数，使用Position对象作为唯一的参数
- error：获取位置信息失败时的回调函数，使用PositionError对象作为唯一的参数，可选项
- options：一个可选的PositionOptions对象，可选项

注意：使用它需要得到用户的授权，浏览器会跳出一个对话框，询问用户是否许可当前页面获取他的地理位置。如果同意授权，就会调用success；如果用户拒绝授权，则会抛出一个错误，调用error。

1.2.1 授权成功

```
function success(position){
    //成功
}
```

`position` 参数是一个Position对象。其有两个属性：`timestamp` 和 `coords`。`timestamp` 属性是一个时间戳，返回获得位置信息的具体时间。`coords` 属性指向一个对象，包含了用户的位置信息，主要是以下几个值：

- `coords.latitude`：纬度

- coords.longitude : 经度
- coords.accuracy : 精度
- coords.altitude : 海拔
- coords.altitudeAccuracy : 海拔精度 (单位 : 米)
- coords.heading : 以360度表示的方向
- coords.speed : 每秒的速度 (单位 : 米)

1.2.2 授权失败

```
function error(PositionError){
    //用户拒绝授权
}
```

PositionError 接口表示当定位设备位置时发生错误的原因。

`PositionError.code` 返回无符号的、简短的错误码：

- 1 相当于PERMISSION_DENIED 地理位置信息的获取失败，因为该页面没有获取地理位置信息的权限。
- 2 相当于POSITION_UNAVAILABLE
地理位置获取失败，因为至少有一个内部位置源返回一个内部错误。
- 3 相当于TIMEOUT
获取地理位置超时，通过定义PositionOptions.timeout 来设置获取地理位置的超时时长。

1.2.3 options参数

用来设置定位行为

```
var option = {
    enableHighAccuracy : true,
    timeout : Infinity,
    maximumAge : 0
};
```

参数说明：

- `enableHighAccuracy` : 如果设为true，就要求客户端提供更精确的位置信息，这会导致更长的定位时间和更大的耗电，默认设为false。
- `Timeout` : 等待客户端做出回应的最大毫秒数，默认值为Infinity (无限)。
- `maximumAge` : 客户端可以使用缓存数据的最大毫秒数。如果设为0，客户端不读取缓存；如果设为infinity，客户端只读取缓存。

1.3 监视定位

`watchPosition()` 方法可以用来监听用户位置的持续改变。它与 `getCurrentPosition()` 接受相同的参数，但回调函数会被调用多次。错误回调函数与 `getCurrentPosition()` 中一样是可选的，也会被多次调用。

```
var watchID = navigator.geolocation.watchPosition(success, error, options);
```

一旦用户位置发生变化，就会调用回调函数`success`。这个回调函数的事件对象，也包含`timestamp`和`coords`属性。

`watchPosition()` 函数会返回一个 ID，唯一地标记该位置监视器。您可以将这个 ID 传给 `clearWatch()` 函数来停止监视用户位置。

```
navigator.geolocation.clearWatch(watchID);
```

1.4 完整例子

```
<div id="myLocation"></div>

var ml=document.getElementById("myLocation");
function getUserLocation(){
    if("geolocation" in navigator){
        var options={
            enableHighAccuracy: true,
            maximumAge: 30000,
            timeout: 27000
        };
        navigator.geolocation.getCurrentPosition(success,error,options);
        var watchID = navigator.geolocation.watchPosition(success,error, options);
    }else{
        ml.innerHTML="您的浏览器不支持定位！";
    }
}
function success(position){
var coords=position.coords;
var lat=coords.latitude;
var lng=coords.longitude;
ml.innerHTML="您当前所在的位置：经度"+lat+"；纬度："+lng;
//只有firefox支持address属性
if(typeof position.address !== "undefined"){
    var country = position.address.country;
    var province = position.address.region;
    var city = position.address.city;
    ml.innerHTML += "您的地址" + country + province + city;
}
}
function error(error){
```

```
switch(error.code){  
    case error.TIMEOUT:  
        ml.innerHTML="连接超时, 请重试";break;  
    case error.PERMISSION_DENIED:  
        ml.innerHTML="您拒绝了使用位置共享服务, 查询已取消";break;  
    case error.POSITION_UNAVAILABLE:  
        ml.innerHTML="亲, 非常抱歉, 我们暂时无法为您提供位置服务";break;  
}  
ml.style.color="red";  
}  
window.onload=function(){  
    getUserLocation();  
}
```

requestAnimationFrame

requestAnimationFrame

`requestAnimationFrame` 是浏览器用于定时循环操作的一个接口，类似于`setTimeout`，主要用途是按帧对网页进行重绘。

可以使用 `requestAnimationFrame()` 来制作动画。

如果你想做逐帧动画的时候，你应该用这个方法。这就要求你的动画函数执行会先于浏览器重绘动作。通常来说，被调用的频率是每秒60次，但是一般会遵循W3C标准规定的频率。如果是后台标签页面，重绘频率则会大大降低。

回调函数只会被传入一个`DOMHighResTimeStamp`参数，这个参数指示当前被`requestAnimationFrame`序列化的函数队列被触发的时间。因为很多个函数在这一帧被执行，所以每个函数都将被传入一个相同的时间戳，尽管经过了之前很多的计算工作。这个数值是一个小数，单位毫秒，精确度在 10 μs。

`requestAnimationFrame()` 函数是全局函数。

考虑兼容，如下：

```
var requestAnimFrame = function() {
  return window.requestAnimationFrame || window.webkitRequestAnimationFrame ||
  window.mozRequestAnimationFrame || window.oRequestAnimationFrame || window.msRequestAnimationFrame ||
  function(a) {window.setTimeout(a, 1e3 / 60, (new Date).getTime());};
}();
```

使用方法：

```
function step(){
  requestAnimationFrame(step);
}
```

`requestAnimationFrame()`方法会返回一个`requestID`，是一个长整型非零值，作为一个唯一的标识符，可将该值作为参数传递给`window.cancelAnimationFrame()`来取消这个函数。

```
cancelAnimationFrame(requestID);
```

兼容性代码：

```
var cancelAnimFrame = function() {
  return window.cancelAnimationFrame || window.webkitCancelAnimationFrame || wi
ndow.mozCancelAnimationFrame || function(id) { clearTimeout(id); };
}();
```

File API

File API

3、文件系统API

3.1 File API

在HTML5中新增了File API，可以让网页要求用户选择本地文件，并且读取这些文件的信息。选择的方式可以是HTML `<input>` 元素，也可以是拖拽。

```
<input type="file" id="photo">

var selectedFile = document.getElementById('photo');
var file = selectedFile.files[0];
//或者
file = selectedFile.files.item(0)
```

`selectedFile.files` 返回一个FileList对象（有一个属性 `length`，表示文件（File对象）个数），包含了一个或多个File对象，每个File对象都有自己的属性：

- `file.name`：文件名，该属性只读。
- `file.size`：文件大小，单位为字节，该属性只读。
- `file.type`：文件的MIME类型，如果分辨不出类型，则为空字符串，该属性只读。
- `file.lastModified`：文件的上次修改时间，格式为时间戳。
- `file.lastModifiedDate`：文件的上次修改时间，格式为Date对象实例。

注意：如果要允许用户选取多个文件，需要加上 `multiple` 属性

```
<input type="file" multiple />
```

一般情况下，我们会为input注册 `change` 事件，当文件被选择时，触发change。

```
selectFile.addEventListener('change', function(){
  var fileList = this.files;
  for(var i = 0; i < fileList.length; i++){
    var file = fileList[i]; //或者 fileList.item(0);
  }
}, false);
```

3.1.1 拖拽文件

前面也说过，我们也可以通过拖拽方式选择文件。

```
<div id="dropbox"></div>

dropbox = document.getElementById('dropbox');
dropbox.addEventListener('dragenter', dragenter, false);
dropbox.addEventListener('dragover', dragover, false);
dropbox.addEventListener('drop', drop, false);
```

在上面的代码中，ID为dropbox的div就是我们拖放目的区域。

拖放事件：

```
function dragenter(e){
  e.stopPropagation();
  e.preventDefault();
}

function dragover(e){
  e.stopPropagation();
  e.preventDefault();
}

function drop(e){
  e.stopPropagation();
  e.preventDefault();

  var dt = e.dataTransfer;
  var files = dt.files;
}
```

在上面的代码中，参数e是一个事件对象，该参数的dataTransfer.files属性就是一个FileList对象，里面包含了拖放的文件。

注意：使用拖放事件时，必须阻止dragenter和dragover事件的默认行为，才能触发drop事件。

3.1.2 FileReader API

在上面我们知道如何获取文件信息，如何使用呢？

这时我们就要用到 `FileReader API` 了，此API用于读取文件，即把文件内容读入内存。它的参数是File对象或Blob对象。

首先，我们需要实例化FileReader对象：

```
var reader = new FileReader();
```

对于不同类型的文件，FileReader提供了不同的方法来读取文件：

- `readAsBinaryString(Blob|File)` : 返回二进制字符串，该字符串每个字节包含一个0到255之间的整数。

- `readAsText(Blob|File, opt_encoding)` : 返回文本字符串。默认情况下，文本编码格式是'UTF-8'，可以通过可选的格式参数，指定其他编码格式的文本。
 - `readAsDataURL(Blob|File)` : 返回一个基于Base64编码的数据-uri对象。
 - `readAsArrayBuffer(Blob|File)` : 返回一个ArrayBuffer对象，即固定长度的二进制缓存数据。
- 我们来看一个显示用户所选图片的缩略图的例子：

```
<input type="file" onchange="handleFiles(this.files)" />

function handleFiles(files){
  for(var i = 0; i < files.length; i++){
    var file = files[i];
    var imageType = /^image\//;
    if(!imageType.test(file.type)) continue;
    var img = document.createElement('img');
    img.file = file;
    document.body.appendChild(img);

    var reader = new FileReader();
    reader.onload = function(e){
      img.src=e.target.result;
    };
    reader.readAsDataURL(file);
  }
}
```

在上面的代码中，我们通过onchange去监听input内文件信息的变化，通过file.type判断用户选择的是是否是图片，这里使用File对象的readAsDataURL()方法来返回一个data URL，然后使用onload事件监听文件是否读取完毕，如果读取完毕，我们就可以事件对象e来读取文件内容，也就是e.target.result；

`readAsDataURL()` 方法用于读取文本文件，它的第一个参数是File或Blob对象，第二个参数是前一个参数的编码方法，如果省略就默认为UTF-8编码。该方法是异步方法，一般监听onload事件，用来确定文件是否加载结束，方法是判断FileReader实例的result属性是否有值。其他三种读取方法，用法与readAsDataURL方法类似。注意：如果浏览器不支持FileReader，你也可以使用URL.createObjectURL(file)方法来创建一个data URL来显示图片缩略图。

FileReader API还有一个 `abort` 方法，用于中止文件上传。

FileReader API的其他监听事件

- `onabort`方法：读取中断或调用reader.abort()方法时触发。
- `onerror`方法：读取出错时触发。
- `onload`方法：读取成功后触发。
- `onloadend`方法：读取完成后触发，不管是否成功。触发顺序排在 `onload` 或 `onerror` 后面。
- `onloadstart`方法：读取将要开始时触发。

- `onprogress`方法：读取过程中周期性触发。

FullScreen API

全屏操作 (FullScreen API)

全屏API可以控制浏览器全屏显示。

8.1 requestFullscreen()

Element节点的requestFullscreen方法，可以使得这个节点全屏。

```
function openFullscreen(elem){
  if (elem.requestFullscreen) {
    elem.requestFullscreen();
  } else if (elem.mozRequestFullScreen) {
    elem.mozRequestFullScreen();
  } else if (elem.webkitRequestFullscreen) {
    elem.webkitRequestFullscreen();
  }
}

openFullscreen(document.documentElement); //整个页面全屏
openFullscreen(document.getElementById('videoElement')); //使播放器全屏
```

运行到这里，Gecko 与 WebKit 两个实现中出现了一个值得注意的区别：

Gecko 会为元素自动添加 CSS 使其伸展以便铺满屏幕：“width: 100%; height: 100%”。WebKit 则不会这么做；它会让全屏的元素以原始尺寸居中到屏幕中央，其余部分变为黑色。

所以为了在 WebKit 下也达到与 Gecko 同样的全屏效果，你需要手动为元素增加 CSS 规则“width: 100%; height: 100%;”：

```
/* html */
:-webkit-full-screen {}
:-moz-fullscreen {}
fullscreen {}

:-webkit-full-screen video {
  width: 100%;
  height: 100%;
}
```

8.2 exitFullscreen()

Document对象的exitFullscreen方法用于取消全屏。

```

function closeFullscreen(){
  if (document.exitFullscreen) {
    document.exitFullscreen();
  } else if (document.mozCancelFullScreen) {
    document.mozCancelFullScreen();
  } else if (document.webkitCancelFullScreen) {
    document.webkitCancelFullScreen();
  } else if (document.msExitFullscreen) {
    document.msExitFullscreen();
  }
}
closeFullscreen()

```

用户手动按下ESC键或F11键，也可以退出全屏键。此外，加载新的页面，或者切换tab，或者从浏览器转向其他应用（按下Alt-Tab），也会导致退出全屏状态。

8.3 全屏属性和事件

8.3.1 属性

`document.fullscreenElement`: 当前处于全屏状态的元素 element. `document.fullscreenEnabled`: 标记 `fullscreen` 当前是否可用.

```

var fullscreenElement = document.fullscreenElement || document.mozFullscreenElement || document.webkitFullscreenElement || document.msFullscreenElement;

var fullscreenEnabled = document.fullscreenEnabled || document.mozFullscreenEnabled || document.webkitFullscreenEnabled || document.msFullscreenEnabled;

```

8.3.2 全屏事件

`fullscreenchange`事件：浏览器进入或离开全屏时触发。

`fullscreenerror`事件：浏览器无法进入全屏时触发，可能是技术原因，也可能是用户拒绝。

```

document.addEventListener('fullscreenchange', function(){
  if(document.fullscreenElement){
    //进入全屏
  }else{
    //退出全屏
  }
}, false);

```

IndexedDB

IndexedDB

4、客户端数据库 (IndexedDB)

IndexedDB (对象数据库) 可以说是浏览器端数据库 , 可以被网页脚本程序创建和操作。它允许储存大量数据 , 提供查找接口 , 还能建立索引。

在IndexedDB API中 , 一个数据库就是一个命名对象仓库 (object store) 的集合 , 对象存储区存储的是对象。

IndexedDB特点 :

- 键值对存储 : 在对象仓库中 , 数据以 “键值对” 的形式保存 , 每一个数据都有对应的键名 , 键名是独一无二的 , 不能有重复 , 否则会抛出一个错误。
- 同域限制 : IndexedDB数据库的作用域是限制在包含它们的文档源中 , 每一个数据库对应创建该数据库的域名 , 两个同源的Web页面互相之间可以访问对方的数据 , 但非同源的页面就不行。
- 支持事务 : IndexedDB支持事务 , 这就是说对数据库的查询和更新都是包含在一个事务 (transaction) 中 , 以此来确保这些操作要么是一起成功 , 要么是一起失败 , 并且永远不会让数据库出现更新到一半的情况。
- 异步 : IndexedDB的操作不会阻塞浏览器的UI主线程。
- 储存空间大 : IE的储存上限是250MB , Chrome和Opera是剩余空间的某个百分比 , Firefox则没有上限。

(1) 检测浏览器是否支持IndexedDB API

```
if('indexedDB' in window){
    //支持
}else{
    //不支持
}
```

(2) 访问数据库

要异步访问数据库 , 就要调用 window 对象 indexedDB 属性的 open() 方法

```
var request = indexedDB.open(name[, version])
```

indexedDB.open方法可传输入两个参数 : name是数据库名称 , 必填 ; version是数据库版本 , 是一个大于0的正整数 (0将报错) 。

open方法返回一个 IDBRequest 对象 (IDBOpenDBRequest) ,

注意 : 如果数据库存在 , 将打开数据库 , 否则 , 则会新建该数据库。如果省略第二个参数 , 则会自动创建版本为1的该数据库。

当打开数据时 , 有可能触发4种事件 :

success : 打开成功。

error : 打开失败。

upgradeneeded : 第一次打开该数据库 , 或者数据库版本发生变化。

blocked : 上一次的数据库连接还未关闭。

第一次打开数据库时 , 会先触发upgradeneeded事件 , 然后触发success事件。

```
request.onupgradeneeded = function(e){}

request.onsuccess = function(e){
  db = e.target.result;
}
```

回调函数接受一个事件对象event作为参数 , 它的target.result属性就指向打开的IndexedDB数据库。

(3) IndexedDB实例对象的方法

3.1 createObjectStore()方法

createObjectStore()方法用于创建存放数据的 “对象仓库” (object store) 。

```
db.createObjectStore(name[, options]);
```

参数说明 :

参数name是对象仓库的名字 ; options是可选参数 , 用来设置对象仓库的属性 , 可配置两个属性 : keyPath和autoIncrement , 分别表示每条记录的键名和是否使用自动递增的整数作为键名 , 默认为false。

```
db.createObjectStore('db1', {keyPath: 'user'});
db.createObjectStore('db2', {autoIncrement: true});
```

由于对象仓库的名字具有唯一性 (当创建已存在的数据库时 , 会报错) , 所以在创建对象仓库时 , 我们有必要检测对象仓库是否已存在 :

db.objectStoreNames.contains(name)

objectStoreNames属性返回一个DOMStringList对象 , 里面包含了当前数据库所有 “对象仓库” 的名称。可以使用DOMStringList对象的contains方法 , 检查数据库是否包含某个 “对象仓库” 。

3.2 transaction方法

创建了数据库 , 当然要使用它 , 不过数据库的更新、读取和删除是建立在事务的基础上的 , 所以我们首先要创建一个事务 :

```
var t = db.transaction(array, type)
```

transcation()方法接受两个参数 : 参数array是一个数组 , 包含了所要使用的对象仓库 , 通常是一个 ; 参数type是

一个表示操作类型的字符串，目前只有两种类型：readonly（只读）和readwrite（读写）。

```
t = db.transaction(['db1', 'readwrite']);
```

transaction()方法返回一个事务对象，该对象的objectStore()方法用于获取指定的对象仓库：

```
var store = t.objectStore('db1');
```

事务对象有三个监听事件：

abort：事务中断。

complete：事务完成。

error：事务出错。

假如事务完成时：

```
t.oncomplete =function(e){}
```

3.2.1 数据操作

下面的方法都是在事件对象上。

(1) add()方法

add()方法用来添加数据

```
var add = store.add(data, key)
```

参数说明：参数data是所要添加的数据；参数key是这条数据对应的键名（key）。

add()方法是异步的，有success和error事件：

```
add.onsuccess = funciton(e){}
add.onerror = function(e){}
```

(2) get()方法

get()方法用来读取数据，它的参数是键名

```
store.get(key)
```

get方法也是异步的，也有success和error事件。

(3) put()方法

put()方法用来更新数据，与add()方法类似：

```
var update = store.put(data, key)
```

(4) delete()方法

delete()方法用来删除数据，它的参数是键名：

```
var delete = store.delete(key)
```

delete方法也是异步的，也有success和error事件。

(5) openCursor()方法

openCursor()方法用来遍历数据：

```
var cursor = store.openCursor()
```

openCursor方法也是异步的，也有success和error事件。

```
cursor.onsuccess = function(e){
  var res = e.target.result;
  console.log('key', res.key);
  console.log('data', res.value);
  res.continue()
}
```

e.target.result属性指向当前数据对象。当前数据对象的key和value分别返回键名和键值（即实际存入的数据）。continue方法将光标移到下一个数据对象，如果当前数据对象已经是最后一个数据了，则光标指向null。openCursor方法还可以接受第二个参数，表示遍历方向，默认值为next，其他可能的值为prev、nextunique和prevunique。后两个值表示如果遇到重复值，会自动跳过。

3.3 createIndex()方法

createIndex()方法用来创建索引：

```
createIndex(index, name, options)
```

createIndex方法接受三个参数，第一个是索引名称，第二个是建立索引的属性名，第三个是参数对象，用来设置索引特性。unique表示索引所在的属性是否有唯一值。

3.3.1 index方法

index()方法用于从对象仓库返回指定的索引。

```
var index = store.index(index);
var data = index.get(name)
```

注意：get方法有可能取回多个数据对象，因为name属性没有唯一值。

检测设备方向

检测设备方向

7、手机设备方向

7.1 方向

Orientation API用于检测手机的摆放方向（竖放或横放）。

(1) 检测浏览器是否支持

```
if(window.DeviceOrientationEvent){
    //支持
}else{
    //不支持
}
```

(2) 监听方向变化

一旦设备的方向发生变化，会触发deviceorientation事件，可以对该事件指定回调函数。

```
window.addEventListener("deviceorientation", listener, false);

function listener(event){
    var alpha = event.alpha;
    var beta = event.beta;
    var gamma = event.gamma;
}
```

上面代码中，event事件对象有alpha、beta和gamma三个属性，它们分别对应手机摆放的三维倾角变化。要理解它们，就要理解手机的方向模型。当手机水平摆放时，使用三个轴表示它的空间位置：x轴代表横轴、y轴代表竖轴、z轴代表垂直轴。event对象的三个属性就对应这三根轴的旋转角度。

- alpha：表示围绕z轴的旋转，0~360(度)。当设备水平摆放时，顶部指向地球的北极，alpha此时为0。
- beta：表示围绕x轴的旋转，-180~180(度)，由前向后。当设备水平摆放时，beta此时为0。
- gamma：表示围绕y轴的选择，-90~90(度)，从左到右。当设备水平摆放时，gamma此时为0。

7.2 移动 (motion)

(1) 检测是否支持

```
if(window.DeviceMotionEvent){
    //支持
}else{
    //不支持
}
```

}

(2) 和方向事件一样，移动也有监听事件：devicemotion

```
window.addEventListener('devicemotion', listener, true)

function listener(event){
  var acceleration = event.acceleration;
  var accelerationIncludingGravity = event.accelerationIncludingGravity;
  var rotationRate = event.rotationRate;
  var interval = event.interval;
}
```

上面代码中，event事件对象有acceleration、accelerationIncludingGravity、rotationRate和interval四个属性。

属性说明：

(1) acceleration、accelerationIncludingGravity

acceleration和accelerationIncludingGravity属性都包含三个轴：

x轴：西向东 (acceleration.x)

y轴：南向北 (acceleration.y)

z轴：垂直地面 (acceleration.z)

(2) rotationRate

rotationRate有三个值：

alpha: 设备沿着垂直屏幕的轴的旋转速率 (桌面设备相对于键盘)

beta: 设备沿着屏幕左至右方向的轴的旋转速率(桌面设备相对于键盘)

gamma: 设备沿着屏幕下至上方向的轴的旋转速率(桌面设备相对于键盘)

(3) interval

interval 表示的是从设备获取数据的频率，单位是毫秒。

更多：检测设备方向

简单的摇一摇功能：

```
var SHAKE_THRESHOLD = 3000;
var last_update = 0;
var x = y = z = last_x = last_y = last_z = 0;

if (window.DeviceMotionEvent) {
  window.addEventListener('devicemotion', deviceMotionHandler, false);
}

function deviceMotionHandler(eventData) {
  var acceleration = eventData.accelerationIncludingGravity;
```

```
var curTime = new Date().getTime();
var diffTime = curTime - last_update;
if (diffTime > 100) {
    last_update = curTime;
    x = acceleration.x;
    y = acceleration.y;
    z = acceleration.z;
    var speed = Math.abs(x + y + z - last_x - last_y - last_z) / diffTime * 100
00;
    if (speed > SHAKE_THRESHOLD) {
        //dosomething
    };
    last_x = x;
    last_y = y;
    last_z = z;
}
}
```

Blob

Blob

2、 Blob

Blob (Binary Large Object) 对象代表了一段二进制数据，提供了一系列操作接口。比如通过new Blob()创建的对象就是Blob对象。又比如，在XMLHttpRequest里，如果指定responseType为blob，那么得到的返回值也是一个blob对象。

2.1 生成Blob对象

生成Blob对象有两种方法：一种是使用Blob构造函数，另一种是对已有的Blob对象使用slice()方法切出一段。

(1) Blob构造函数

```
var blob = new Blob(data, type)
```

Blob构造函数接受两个参数：

参数data是一组数据，所以必须是数组，即使只有一个字符串也必须用数组装起来。

参数type是对这一Blob对象的配置属性，目前也只有一个type也就是相关的MIME需要设置 type的值：

'text/csv;charset=UTF-8' 设置为csv格式，并设置编码为UTF-8

'text/html' 设置成html格式

注意：任何浏览器支持的类型都可以这么用

```
var blob = new Blob(['我是Blob'], {type: 'text/html'});
```

2.2 属性

```
blob.size    //Blob大小（以字节为单位）
blob.type    //Blob的MIME类型，如果是未知，则是“ ”（空字符串）
```

2.3 slice()

slice()返回一个新的Blob对象，包含了源Blob对象中指定范围内的数据。

```
blob.slice(
  optional long long start,
  optional long long end,
  optional DOMString contentType  );
```

参数说明：

start 可选，开始索引,可以为负数,语法类似于数组的slice方法.默认值为0.

end 可选，结束索引,可以为负数,语法类似于数组的slice方法.默认值为最后一个索引.

contentType可选，新的Blob对象的MIME类型,这个值将会成为新的Blob对象的type属性的值,默认为一个空字符串.

2.4 Blob的使用

使用Blob最简单的方法就是创建一个URL来指向Blob：

```
<a download="data.txt" id="getData">下载</a>

var data= 'Hello world!';
var blob = new Blob([data], {
  type: 'text/html;charset=UTF-8'
});
window.URL = window.URL || window.webkitURL;
document.querySelector("#getData").href = URL.createObjectURL(blob);
```

上面的代码将Blob URL赋值给a，点击后提示下载文本文件data.txt，文件内容为“Hello World”。

2.5 URL.createObjectURL()

```
objectURL = URL.createObjectURL(blob);
```

使用URL.createObjectURL()函数可以创建一个Blob URL，参数blob是用来创建URL的File对象或者Blob对象，返回值格式是：blob://URL。

注意：在每次调用 createObjectURL() 方法时，都会创建一个新的 URL 对象，即使你已经用相同的对象作为参数创建过。当不再需要这些 URL 对象时，每个对象必须通过调用 URL.revokeObjectURL() 方法传入创建的URL为参数，用来释放它。浏览器会在文档退出的时候自动释放它们，但是为了获得最佳性能和内存使用状况，你应该在安全的时机主动释放掉它们。

2.6 乱码问题

当数据中包含汉字时，导出的文件可能会出现乱码，不过我们可以这样解决：

```
var data = "\ufffeff" + "汉字";
```

vibrate

vibrate

5、设备震动 (Vibration API)

Vibration接口用于在浏览器中发出命令，使得设备振动。

(1) 检测是否可用

目前，只有Chrome和Firefox的Android平台最新版本支持它。

```
navigator.vibrate = navigator.vibrate || navigator.webkitVibrate || navigator.mozVibrate || navigator.msVibrate;

if (navigator.vibrate) {
    // 支持
}
```

(2) 振动

```
navigator.vibrate(1000);
```

vibrate()方法的参数就是振动持续的毫秒数，除了单个数值外，还可以接受一个数组作为参数，表示振动的模式。偶数位置的数组成员表示振动的毫秒数，奇数位置的数组成员表示等待的毫秒数。

```
navigator.vibrate([200, 100, 300])
```

上面代码表示，设备先振动200毫秒，然后等待100毫秒，再接着振动300毫秒。

注意：vibrate是一个非阻塞式的操作，即手机振动的同时，JavaScript代码仍然继续向下运行。要停止振动，只有将0毫秒或者一个空数组传入vibrate方法。

Luminosity API

Luminosity

6、屏幕亮度 (Luminosity API)

Luminosity API用来屏幕亮度调节，不过目前只有Firefox支持。

当移动设备的亮度传感器感知外部亮度发生显著变化时，会触发devicelight事件。

```
window.addEventListener('devicelight', function(event) {  
    console.log(event.value + ' lux');  
});
```

上面代码表示，devicelight事件的回调函数，接受一个事件对象作为参数。该对象的value属性就是亮度的流明值。

WebRTC

WebRTC

WebRTC (Web Real-Time Communication , 网页实时通信) , 是一个支持网页浏览器进行实时语音对话或视频对话的API。

1、getUserMedia

要播放摄像头的影像，首先需要一个video标签：

```
<video id="video"></video>
```

获取摄像头影像主要是通过 `navigator.getUserMedia` 这个接口，这个接口的支持情况已经逐渐变好了（[点击这里](#)），不过，使用的时候还是要加上前缀的。

兼容代码：

```
navigator.getUserMedia = navigator.getUserMedia || navigator.webkit GetUserMedia  
|| navigator.mozGetUserMedia;
```

语法：

```
navigator.getUserMedia(constraints, successCallback, errorCallback);
```

参数说明：

- constraints : Object类型，指定了请求使用媒体的类型
- successCallback : 启用成功时的函数，它传入一个参数，为视频流对象，可以进一步通过 `window.URL.createObjectURL()` 接口把视频流转换为对象URL。
- errorCallback : 启动失败时的函数。它传入一个参数，为错误对象 (chrome) 或错误信息字符串 (Firefox) ，可能值：

`PERMISSION_DENIED` : 用户拒绝提供信息。

`NOT_SUPPORTED_ERROR` : 浏览器不支持硬件设备。

`MANDATORY_UNSATISFIED_ERROR` : 无法发现指定的硬件设备。

例如要启用视频设备（摄像头），可这样：

```
navigator.getUserMedia({  
  video: true
```

```
});
```

如果要同时启用视频设备和音频设备，可这样：

```
navigator.getUserMedia({
  video: true,
  audio: true
});
```

1.1 获取摄像头完整实例

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title></title>
  </head>
  <body>
    <video id="video" width="400" height="300"></video>
    <button id="live">直播</button>
  <script>
    var video = document.getElementById('video');

    function liveVideo(){
      // 获取到window.URL对象
      var URL = window.URL || window.webkitURL;
      navigator.getUserMedia({
        video: true
      }, function(stream){
        video.src = URL.createObjectURL(stream);
        video.play();
      }, function(error){
        console.log(error.name || error);
      });
    }
    document.getElementById("live").addEventListener('click', function(){
      liveVideo();
    })
  </script>
  </body>
</html>
```

当点击直播按钮时，电脑会提升用户是否允许使用摄像头，允许之后，网页上就可以实时显示摄像头影像了。如果不允许，就会触发错误事件。

1.1.1 截图

除了实时直播外，我们还可以做实时截图效果，这时我们需要利用 `<canvas>` 元素来画图，代码如下：

```
<!DOCTYPE html>
<html>

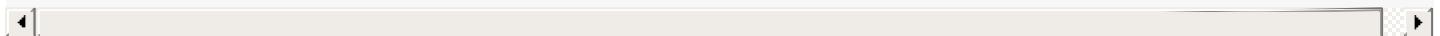
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        #canvas, #video{
            float:left;
            margin-right:10px;
            background:#fff;
        }
        .box{
            overflow:hidden;
            margin-bottom:10px;
        }
    </style>
</head>

<body>
    <div class="box">
        <video id="video" width="400" height="300"></video>
        <canvas id="canvas"></canvas>
    </div>
    <button id="live">直播</button>
    <button id="snap">截图</button>
<script>
    var video = document.getElementById('video');
    var canvas = document.getElementById('canvas');
    var ctx = canvas.getContext('2d');
    var width = video.width;
    var height = video.height;
    canvas.width = width;
    canvas.height = height;
    function liveVideo(){
        // 获取到window.URL对象
        var URL = window.URL || window.webkitURL;
        navigator.getUserMedia({
            video: true
        }, function(stream){
            video.src = URL.createObjectURL(stream);
            video.play();
            //点击截图
            document.getElementById("snap").addEventListener('click',function(){

                ctx.drawImage(video, 0, 0, width, height);
            });
        });
    }
</script>
</body>

```

```
    }, function(error){
      console.log(error.name || error);
    });
}
//开始直播
document.getElementById("live").addEventListener('click',function(){
  liveVideo();
})
</script>
</body>
</html>
```



1.1.2 保存图片

当然，截图后，你也可以保存下来：

```
<a download='snap.png' id="download">下载图片</a>

var url = canvas.toDataURL('image/png');
document.getElementById('download').src = url;
```

1.2 捕获麦克风声音

要通过浏览器捕获声音，需要借助Web Audio API。

```
window.AudioContext = window.AudioContext || window.webkitAudioContext;

var context = new AudioContext();

function onSuccess(stream){
  var audioInput = context.createMediaStreamSource(stream);
  audioInput.connect(context.destination);
}

navigator.getUserMedia({audio: true} ,onSuccess);
```

Page Visibility API

PageVisibility

1、PageVisibility

PageVisibility API是用于判断页面是否处于浏览器的当前窗口，即是否可见。

这API是部署在 `document` 对象上的。

1.1 属性

它有两个属性：

- `document.hidden`：返回一个布尔值，表示当前是否被隐藏，只读
- `document.visibilityState`：只读，表示页面当前的状态，有四个可能值：`visible`（页面可见）、`hidden`（页面不可见）、`prerender`（页面正处于渲染之中，不可见）、`unloaded`（如果文档被卸载了）

在使用这两个属性时，要加上不同浏览器的私有前缀：

获取 `hidden` 属性值：

```
function getHiddenProp() {
  var prefixes = ['webkit', 'moz', 'ms', 'o'];

  if ('hidden' in document) return 'hidden';

  for(var i = 0; i < prefixes.length; i++) {
    var hidden = prefixes[i] + 'Hidden';
    if( hidden in document) {
      return hidden;
    }
  };

  return null;
}
```

获取 `visibilityState` 属性

```
function getVisibilityState() {
  var prefixes = ['webkit', 'moz', 'ms', 'o'];

  if ('visibilityState' in document) return 'visibilityState';

  for(var i = 0; i < prefixes.length; i++) {
    var visibilityState = prefixes[i] + 'VisibilityState';
```

```
if( hidden in document) {  
    return visibilityState;  
}  
};  
  
return null;  
}
```

1.2 事件

当页面的可见状态发生变化时，会触发 `visibilityChange` 事件（也是需要加上私有前缀）

```
var hidden = getHiddenProp();  
  
var visibilityChange = hidden.split(/[H|h]/, '')[0] + 'visibilitychange';  
  
document.addEventListener(visibilityChange, function(){  
    console.log( document[getVisibilityState] );  
});
```

Performance API

Performance API 高精度时间戳

Performance API是ECMAScript 5才引入的，它的精度可达到1毫秒的千分之一。

目前，所有主要浏览器都已经支持 `performance` 对象，包括Chrome 20+、Firefox 15+、IE 10+、Opera 15+。

1.1 `performance.timing`对象

`performance` 对象是全局的，它的 `timing` 属性是一个对象，它包含了各种与浏览器性能有关的时间数据，提供浏览器处理网页各个阶段的耗时。

`performance.timing` 对象包含下列属性（全部只读）：

`navigationStart`：当前浏览器窗口的前一个网页关闭，发生`unload`事件时的Unix毫秒时间戳。如果没有前一个网页，则等于`fetchStart`属性。

`unloadEventStart`：如果前一个网页与当前网页属于同一个域名，则返回前一个网页的`unload`事件发生时的Unix毫秒时间戳。如果没有前一个网页，或者之前的网页跳转不是在同一个域名内，则返回值为`0`。

`unloadEventEnd`：如果前一个网页与当前网页属于同一个域名，则返回前一个网页`unload`事件的回调函数结束时的Unix毫秒时间戳。如果没有前一个网页，或者之前的网页跳转不是在同一个域名内，则返回值为`0`。

`redirectStart`：返回第一个HTTP跳转开始时的Unix毫秒时间戳。如果没有跳转，或者不是同一个域名内部的跳转，则返回值为`0`。

`redirectEnd`：返回最后一个HTTP跳转结束时（即跳转回应的最后一个字节接受完成时）的Unix毫秒时间戳。如果没有跳转，或者不是同一个域名内部的跳转，则返回值为`0`。

`fetchStart`：返回浏览器准备使用HTTP请求读取文档时的Unix毫秒时间戳。该事件在网页查询本地缓存之前发生。

`domainLookupStart`：返回域名查询开始时的Unix毫秒时间戳。如果使用持久连接，或者信息是从本地缓存获取的，则返回值等同于`fetchStart`属性的值。

`domainLookupEnd`：返回域名查询结束时的Unix毫秒时间戳。如果使用持久连接，或者信息是从本地缓存获取的，则返回值等同于`fetchStart`属性的值。

`connectStart`：返回HTTP请求开始向服务器发送时的Unix毫秒时间戳。如果使用持久连接（`persistent connection`），则返回值等同于`fetchStart`属性的值。

`connectEnd`：返回浏览器与服务器之间的连接建立时的Unix毫秒时间戳。如果建立的是持久连接，则返回值等同于`fetchStart`属性的值。连接建立指的是所有握手和认证过程全部结束。

`secureConnectionStart`：返回浏览器与服务器开始安全链接的握手时的Unix毫秒时间戳。如果当前网页不要求安全连接，则返回`0`。

`requestStart` : 返回浏览器向服务器发出HTTP请求时（或开始读取本地缓存时）的Unix毫秒时间戳。

`responseStart` : 返回浏览器从服务器收到（或从本地缓存读取）第一个字节时的Unix毫秒时间戳。

`responseEnd` : 返回浏览器从服务器收到（或从本地缓存读取）最后一个字节时（如果在此之前HTTP连接已经关闭，则返回关闭时）的Unix毫秒时间戳。

`domLoading` : 返回当前网页DOM结构开始解析时（即`Document.readyState`属性变为“loading”、相应的`readystatechange`事件触发时）的Unix毫秒时间戳。

`domInteractive` : 返回当前网页DOM结构结束解析、开始加载内嵌资源时（即`Document.readyState`属性变为“interactive”、相应的`readystatechange`事件触发时）的Unix毫秒时间戳。

`domContentLoadedEventStart` : 返回当前网页`DOMContentLoaded`事件发生时（即DOM结构解析完毕、所有脚本开始运行时）的Unix毫秒时间戳。

`domContentLoadedEventEnd` : 返回当前网页所有需要执行的脚本执行完成时的Unix毫秒时间戳。

`domComplete` : 返回当前网页DOM结构生成时（即`Document.readyState`属性变为“complete”，以及相应的`readystatechange`事件发生时）的Unix毫秒时间戳。

`loadEventStart` : 返回当前网页`load`事件的回调函数开始时的Unix毫秒时间戳。如果该事件还没有发生，返回`0`。

`loadEventEnd` : 返回当前网页`load`事件的回调函数运行结束时的Unix毫秒时间戳。如果该事件还没有发生，返回`0`。



1.2 performance中的方法

(1) `performance.now()`

`performance.now` 方法返回当前网页自从 `performance.timing.navigationStart` 到当前时间之间的微秒数（毫秒的千分之一）。也就是说，它的精度可以达到100万分之一秒。

(2) `performance.mark()`

`mark`方法用于为相应的视点做标记。

`clearMarks`方法用于清除标记，如果不加参数，就表示清除所有标记。

(3) `performance.getEntries()`

浏览器获取网页时，会对网页中每一个对象（脚本文件、样式表、图片文件等等）发出一个HTTP请求。

`performance.getEntries` 方法以数组形式，返回这些请求的时间统计信息，有多少个请求，返回数组就会有多少个成员。

1.3 `performance.navigation`对象

除了时间信息，`performance`还可以提供一些用户行为信息，主要都存放在`performance.navigation`对象上面。

它有两个属性：

(1) `performance.navigation.type`

该属性返回一个整数值，表示网页的加载来源，可能有以下4种情况：

0：网页通过点击链接、地址栏输入、表单提交、脚本操作等方式加载，相当于常数`performance.navigation.TYPE_NAVIGATENEXT`。

1：网页通过“重新加载”按钮或者`location.reload()`方法加载，相当于常数`performance.navigation.TYPE_RELOAD`。

2：网页通过“前进”或“后退”按钮加载，相当于常数`performance.navigation.TYPE_BACK_FORWARD`。

255：任何其他来源的加载，相当于常数`performance.navigation.TYPE_UNDEFINED`。

(2) `performance.navigation.redirectCount`

该属性表示当前网页经过了多少次重定向跳转。

Web Speech

Web Speech 语音输入

老版本中，如果我们要部署一个带有语音搜索功能的input，可用如下代码：

```
<input id="speech" type="search"
      x-webkit-speech speech />
```

然后通过webkitspeechchange事件来监听用户输入：

```
var speech = document.getElementById('speech');
speech.onwebkitspeechchange = funciton(){
}
```

注意：目前只有Chrome支持这种带语音的输入框。

在新版本中，我们还可以用JavaScript操作语音输入，就是通过Web Speech这个API。（也是只有Chrome支持）

主要有两部分：

- 语言识别（将所说的转换成文本文字）
- 语言合成（将文本文字读出来）

1、语言识别

1.1 SpeechRecognition对象

这个API部署在 `SpeechRecognition` 对象之上。

1.1.1 检测是否支持

```
if ('webkitSpeechRecognition' in window) {
  // 支持
}
```

确定支持后，新建一个 `SpeechRecognition` 的实例对象：

```
if ('webkitSpeechRecognition' in window) {
  var recognition = new webkitSpeechRecognition();
}
```

1.1.2 属性和方法

属性

- continuous : 是否让浏览器始终进行语言识别，默认为false，也就是说，当用户停止说话时，语音识别就停止了。这种模式适合处理短输入的字段。
- maxAlternatives : 设置返回的最大语音匹配结果数量，默认为1
- lang : 设置语言类型，默认值就继承自HTML文档的根节点或者是祖先节点的语言设置。

方法

- start() : 启动语音识别
- stop() : 停止语音识别
- abort() : 中止语音识别

1.1.3 事件

浏览器会询问用户是否许可浏览器获取麦克风数据

这个API提供了11个事件。

- audiostart : 当开始获取音频时触发，也就是用户允许时。
- audioend : 当获取音频结束时触发
- error : 当发生错误时触发
- nomatch : 当找不到与语音匹配的值时触发
- result : 当得到与语音匹配的值时触发，它传入的是一个SpeechRecognitionEvent对象，它的results属性就是语音匹配的结果数组，最匹配的结果排在第一位。该数组的每一个成员是SpeechRecognitionResult对象，该对象的transcript属性就是实际匹配的文本，confidence属性是可信度（在0到1之间）
- soundstart
- soundend
- speechstart
- speechend
- start : 当开始识别语言时触发
- end : 当语音识别断开时触发

看个实例：

```
<input id="speech" type="search"
      x-webkit-speech speech />

var speech = document.getElementById('speech');
if ('webkitSpeechRecognition' in window) {
  var recognition = new webkitSpeechRecognition();
```

```

recognition.onaudiostart = function(){
    speech.value = '开始录音';
};

recognition.ononmatch = function(){
    speech.value = '没有匹配结果，请再次尝试';
};

recognition.onerror = function(){
    speech.value = '错误，请再次尝试';
};

// 如果得到与语音匹配的值，则会触发result事件。

recognition.onresult = function(event){
    if(event.results.length > 0) {
        var results = event.results[0];
        var topResult = results[0];

        if(topResult.confidence > 0.5) {

        } else {
            speech.value = '请再次尝试';
        }
    }
}
}

```

2、语音合成

语音合成只有Chrome和Safari支持。

2.1 SpeechSynthesisUtterance对象

打开控制台，黏贴下面的函数，然后调用：

```

funciton speak(textToSpeak){
    var newUtterance = new SpeechSynthesisUtterance();

    newUtterance.text = textToSpeak;

    window.speechSynthsisi.speak(newUtterance);
}

```

2.1 SpeechSynthesisUtteranc的实例对象的属性

- text : 识别的文本
- volume : 音量 (0~1)

- rate : 发音速度
- pitch : 音调
- voice : 声音
- lang : 语言类型

2.2 window.speechSynthesis.getVoices()

通过API提供给用户的声音在很大程度上取决于操作系统。谷歌有自己的一套给Chrome的默认声音，可以在Mac OS X, Windows和Ubuntu上使用。Mac OS X的声音也可用，所以和OSX的Safari的声音一样。你可以通过开发者工具的控制台看有哪种声音可用。

`window.speechSynthesis.getVoices()` 用来获取发音列表，返回一个数组，每个元素的name属性表示声音名称

2.3 window.speechSynthesis.speak()

`window.speechSynthesis.speak()` 方法是用于发音的。

Notification

Notification

Notifications API 的通知接口用于向用户配置和显示桌面通知。

语法：

```
let notification = new Notification(title, options)
```

参数：

- title : 一定会被显示的通知标题
- options 可选，一个被允许用来设置通知的对象。它包含以下属性：
 - dir : 文字的方向；它的值可以是 auto (自动), ltr (从左到右), or rtl (从右到左)
 - lang: 指定通知中所使用的语言。这个字符串必须在 BCP 47 language tag 文档中是有效的。
 - body: 通知中额外显示的字符串
 - tag: 赋予通知一个ID，以便在必要的时候对通知进行刷新、替换或移除。
 - icon: 一个图片的URL，将被用于显示通知的图标。

属性

静态属性

`Notification.permission`

只读，一个用于表明当前通知显示授权状态的字符串。可能的值包括：denied (用户拒绝了通知的显示), granted (用户允许了通知的显示), 或 default (因为不知道用户的选择，所以浏览器的行为与 denied 时相同).

实例属性

属性	描述
<code>Notification.title</code>	在构造方法中指定的 title 参数
<code>Notification.dir</code>	通知的文本显示方向
<code>Notification.lang</code>	通知的语言
<code>Notification.body</code>	通知的文本内容
<code>Notification.tag</code>	通知的 ID
<code>Notification.icon</code>	通知的图标图片的 URL 地址

事件

事件	描述
<code>Notification.onclick</code>	处理 click 事件的处理

Notification.onshow	处理 show 事件的处理
Notification.onerror	处理 error 事件的处理
Notification.onclose	处理 close 事件的处理

方法

静态方法

```
Notification.requestPermission()
```

用于当前页面想用户申请显示通知的权限。这个方法只能被用户行为调用（比如：onclick 事件），并且不能被其他的方式调用。

实例方法

方法	描述
Notification.close()	用于关闭通知

简单实例

```
// 申请权限
function requestNotification(title, options) {
    // 先检查浏览器是否支持
    if (!('Notification' in window)) {
        alert('This browser does not support desktop notification');
    }
    // 检查用户是否同意接受通知
    else if (Notification['permission'] === 'granted') {
        this.openNotification(title, options);
    }
    // 否则我们需要向用户获取权限
    else if (Notification['permission'] !== 'denied') {
        Notification.requestPermission((permission) => {
            if (permission === 'granted') {
                // 如果用户同意，就可以向他们发送通知
                openNotification(title, options);
            }
        });
    }
}

function openNotification(title, options) {
    const notification = new Notification(title, options);
    notification.onclose = (event) => {
        // 当关闭时
    }
}

requestNotification('您有一条消息通知', {
```

```
body: '来自XX',
requireInteraction: true,
data: {},
icon: 'notification-icon.jpg'
});
```

面向对象的程序设计

面向对象的程序设计

概述

对象概述

面向对象（Object-Oriented，OO）的语言有一个标志，那就是它们都有类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。但是，ECMAScript中并没有类的概念，所以它的对象也有有所不同。

ECMAScript对象是一个无序属性的集合，其属性可以包含基本值、对象或函数。

对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。

每个对象都是基于一个引用类型创建的。

1. 对象

1.1 创建对象

(1) 创建自定义对象的最简单方式就是创建一个Object的实例，然后给其添加属性和方法：

```
var person = new Object();
person.name = 'tg';
person.age = 10;
person.say = function(){
    console.log(this.name);
}
```

上面的例子创建了一个名为person的对象，并为它添加了两个属性（name、age）和一个方法（say()）。

(2) 对象字面量

```
var person = {
    name: 'tg',
    age: 10,
    say: function(){
        console.log(this.name);
    }
}
```

这个person对象和上面例子是等价的。

1.2 属性类型

ECMA-262第5版在定义只有内部才用的特性（attribute）时，描述了属性（property）的各种特征。ECMA-262定义这些特性是为了实现JavaScript引擎用的，因此在JavaScript中不能直接访问它们。该规范将它们放在了两对方括号中，表示特性是内部值，如[[Enumerable]]。

ECMAScript中有两种属性：数据属性和访问权属性

1.2.1 数据属性

数据属性包含一个数据值的位置，在这个位置可以读取和写入值。

数据属性有4个描述特性：

- [[Configurable]]：表示能否通过delete删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器属性。比如直接在对象上定义的属性，它们的这个特性默认值为true。
- [[Enumerable]]：表示能否通过for-in循环返回属性。对于直接在对象上定义的值，默认为true。
- [[Writable]]：表示能否修改属性的值。
- [[Value]]：包含这个属性的数据值。读取属性值时，从这个位置读；写入属性值时，把新值保存在这个位置。默认值为undefined。

要修改属性默认的特性，必须使用ECMAScript 5的 `Object.defineProperty()` 方法，这个方法接收三个参数：属性所在的对象、属性的名字和一个描述符对象。其中，描述符对象的属性必须是：configurable、enumerable、writable和value。

```
var person = {};
Object.defineProperty(person, 'name', {
  writable: false,
  value: 'tg'
});

console.log(person.name); // "tg"
person.name = 'tg2';
console.log(person.name); // "tg"
```

在上面的例子中，我们将person对象中的名为name的属性的 `writable` 设置为false，也就是不可修改，所以即使后面执行了`person.name='tg2'`，最后person对象的name值依旧是原始值。

注意：在严格模式下，如果对一个不可修改的属性执行赋值操作，会抛出错误；非严格模式下则忽略赋值操作。一旦将configurable特性设置为false后，就不能再把它变回可配置的了，如果再修改除writable之外的特性，都会导致错误。

1.2.2 访问器属性

访问器属性不包含数据值，它们包含一对getter和setter函数（非必需）。在读取访问器属性时，会调用getter函数，返回有效的值；在写入访问器属性时，会调用setter函数并传入新值，它负责决定如何处理数据。

访问器属性：

- [[Configurable]]：表示能否通过delete删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器属性。比如直接在对象上定义的属性，它们的这个特性默认值为true。
- [[Enumerable]]：表示能否通过for-in循环返回属性。对于直接在对象上定义的值，默认为true。
- [[Get]]：在读取属性时调用的函数，默认为undefined
- [[Set]]：在写入属性时调用的函数，默认为undefined

访问器属性不能直接定义，也是要使用 `Object.defineProperty()` 方法来定义。

```

var person = {
  _age: 20,
  intro: ''
};

Object.defineProperty(person, 'age', {
  get: function() {
    return this._age;
  },
  set: function(newValue){
    if(newValue < 18) {
      this.intro = '装嫩';
      this._age = newValue;
    }
  }
});

console.log(person.age); // 20
person.age = 17;
console.log(person.intro); // "装嫩"

```

在上面的例子中，访问器属性age有一个getter函数和一个setter函数。getter函数返回 `_age` 的值，setter函数通过判断 `age` 的设置值来改变其他属性值。

1.2.3 定义多个属性

ECMAScript 5提供的 `Object.defineProperties()` 方法可以通过描述符一次定义多个属性，这个方法接收两个对象参数，第一个对象是要添加和修改其属性的对象，第二个对象的属性与第一个对象要添加或修改的属性一一对应。

```

var person = {};
Object.defineProperties(person, {
  _age: {
    value: 20
  },
  intro: {
    value: ''
  },
  age: {
    get: function(){
      return this._age;
    },
    set: function(newValue){
      if(newValue < 18) {
        this.intro = '装嫩';
        this._age = newValue;
      }
    }
}

```

```
});
```

1.2.4 读取属性的特性

ECMAScript 5提供的 `Object.getOwnPropertyDescriptor()` 方法可以取得给定属性的描述符，它接受两个参数：属性所在的对象和要读取其描述符的属性名称，返回来的是一个对象，如果是访问器属性，这个对象的属性有`configurable`、`enumerable`、`get`和`set`；如果是数据属性，这个对象的属性有`configurable`、`enumerable`、`writable`和`value`。

```
var person = {
  _age: 20,
  intro: ''
};

Object.defineProperty(person, 'age', {
  get: function() {
    return this._age;
  },
  set: function(newValue){
    if(newValue < 18) {
      this.intro = '装嫩';
      this._age = newValue;
    }
  }
});

var descriptor = Object.getOwnPropertyDescriptor(person, 'age');

console.log(descriptor.enumerable); // false
console.log(typeof descriptor.get); // "function"
```

在JavaScript中，可以针对任何对象--包括DOM和BOM对象，使用

`Object.getOwnPropertyDescriptor()` 方法。

this关键字

this关键字

JavaScript代码都存在于一定的 上下文对象 中。上下文对象通过 `this` 关键字来动态指定，它永远指向当前对象。简单的说，就是返回属性或方法“当前”所在的对象。

this的工作原理

在5种不同的情况下，`this` 指向的各不相同。

(1) 全局范围内

```
this
```

当在全局范围内使用 `this`，它将指向全局对象

(2) 函数调用

```
function test(){
  console.log(this);
}
test();
```

这里的 `this` 也指向全局对象。

(3) 方法调用

```
test.foo();
```

这个例子中，`this` 指向test对象。

(4) 调用构造函数

```
new test();
```

如果函数倾向于 `new` 关键词一块使用，则我们称这个函数是构造函数。在函数内部，`this` 指向新创建的实例对象。

(5) 显式的设置this

```
function test(a, b) {}

var bar = [];
test.apply(bar, [1, 2]); //数组将会会被扩展
```

this关键字

```
test.call(bar , 1, 2); // 传递到test的参数是： a = 1, b = 2
```

当使用 `Function.prototype` 上的 `call` 或 `apply` 方法时，函数内的 `this` 将会被显式设置为函数调用的第一个参数。

改变this指向的方法

JavaScript提供了`call`、`apply`、`bind`这三个方法，来切换/固定`this`的指向。

(1) `call()`

语法：

```
call([thisObj[, arg1[, arg2[, [, .argN]]]]])
```

定义：调用一个对象的一个方法，以另一个对象替换当前对象。

说明： `call` 方法可以用来代替另一个对象调用一个方法。`call` 方法可将一个函数的对象上下文从初始的上下文改变为由 `thisObj` 指定的新对象。

(2) `apply()`

语法：

```
apply([thisObj[, argArray]])
```

定义：应用某一对象的一个方法，用另一个对象替换当前对象。

说明： 如果 `argArray` 不是一个有效的数组或者不是 `arguments` 对象，那么将导致一个 `TypeError`。如果没有提供 `argArray` 和 `thisObj` 任何一个参数，那么 `Global` 对象将被用作 `thisObj`，并且无法被传递任何参数。

`bind()`方法 `bind()`方法是在ECMAScript 5中新增的方法。`toString()`方法

函数的`toString`方法返回函数的源码。

```
function f(){
  return 1;
}
f.toString()
//function f(){
//  return 1;
//}
```

(3) `bind()`

`bind()`方法会创建一个新函数，称为绑定函数，当调用这个绑定函数时，绑定函数会以创建它时传入 `bind()`方法的第一个参数作为 `this`，传入 `bind()` 方法的第二个以及以后的参数加上绑定函数运行时本身的参数按照顺序作为原函数的参数来调用原函数。

```
var bar=function(){
  console.log(this.x);
}
var foo={
  x:3
}
bar();
bar.bind(foo)();
/*或*/
var func=bar.bind(foo);
func();
```

输出：

```
undefined
3
```

注意：bind()返回的是函数。

原型链

原型链

3.1 原型

每一个JavaScript对象（null除外）都和另一个对象相关联，也可以说，继承另一个对象。另一个对象就是我们熟知的“原型”（prototype），每一个对象都从原型继承属性。只有null除外，它没有自己的原型对象。

我们可以通过`__proto__`（首尾都是双下划线）来获取实例的原型对象。

注意：`__proto__`连接的是实例与构造函数的原型对象之间，而不是实例与构造函数之间。

所有通过对象直接量创建的对象都具有同一个原型对象，并可以通过JavaScript代码`Object.prototype`获得对原型对象的引用。

通过关键字new和构造函数调用创建的对象的原型就是构造函数的`prototype`属性的值。比如：通过`new Object()`创建的对象继承自`Object.prototype`；通过`new Array()`创建的对象的原型就是`Array.prototype`。

没有原型的对象为数不多，`Object.prototype`就是其中之一，它不继承任何属性。

所有的内置构造函数都具有一个继承自`Object.prototype`的原型。

3.1 原型链

在JavaScript中，原型链是实现继承的主要方式。

原型链的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

可用`__proto__`属性来获取或设置对象的原型。

每个构造函数都有一个原型对象（prototype），原型对象本身也是对象，所以它也有自己的原型，而它本身的原型对象也有自己的原型，这样层层递进，就形成了一条链，这个链就是原型链。

JavaScript引擎在访问对象的属性时，会先在对象本身中查找，如果没有找到，则会去原型链中查找，如果找到，则返回值，如果整个原型链中都没有找到这个属性，则返回`undefined`。

```
var person = {
  name: 'human'
};

var person1 = {
  name: 'tg',
  __proto__: person
};

var person2 = {
  __proto__: person
};

console.log(person1.name); // "tg"
console.log(person2.name); // "human"
console.log(person1.id); // undefined
```

从上面例子的运行结果，我们也可以看出原型链的运行机制：从对象本身出发，沿着 `__proto__` 查找，直到找到属性名称相同的值（没有找到，则返回`undefined`）。

所有对象都继承自 `Object`，而这个继承也是通过原型链实现的。所有函数的默认原型都是`Object`的实例，因此默认的原型都会包含一个内部指针，指向 `Object.prototype`。而 `Object.prototype` 的 `__proto__` 的属性值为`null`，标志着原型链的结束。

3.2 原型和实例

有两种方式来确定原型和实例之间的关系：

- 使用 `instanceof` 操作符

```
function Person(){}
var person = new Person();
console.log(person instanceof Object); // true
console.log(person instanceof Person); // true
```

- 使用 `isPrototypeOf()` 方法。

```
console.log(Object.prototype.isPrototypeOf(person)); // true
console.log(Person.prototype.isPrototypeOf(person)); // true
```

作用域

作用域

1.1 作用域

几乎所有的编程语言都有作用域的概念，简单的说，作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。

作用域有全局作用域和局部作用域（一般是在函数内）之分。

1.1.1 全局作用域

在代码中任何地方都能访问到的对象拥有全局作用域。

一般来说，拥有全局作用域有以下几种情况：

(1) 在最外层的函数和在最外层函数外面定义的变量拥有全局作用域

例子：

```
var name = 'tg';
function test(){
    var name2 = 'tg';
}
```

在上面的例子中，变量name和函数test()就拥有全局作用域

(2) 所有未定义而直接赋值（不用var关键字声明）的变量被自动声明为拥有全局作用域

```
funciton test(){
    var age = 10;
    name = 'tg';
}
test();
console.log(age); // 会报错undefined
console.log(name); // "tg"
```

变量name拥有全局作用域，而age在函数外面是无法访问到的

(3) 所有window对象的属性拥有全局作用域

window对象的内置属性都由拥有全局作用域，比如window.location、Date对象等

1.1.2 局部作用域

局部作用域不像全局作用域，它被限定在一个范围内，最常见的局部作用域就是在函数内部，我们也称为函数作用域。

函数作用域是指变量能够被使用的代码区间。超出作用域的变量值一般为undefined，或者被其他同名变量值所覆盖。

```
funciton test(){
  var age = 10;
  console.log(age);
}
test(); // 10
console.log(age); // 会报错undefined
```

在上面的例子中，在函数外面是无法访问到变量age的，但在函数内部是可以访问的。

要记住，对于变量或函数，它的作用域取决于定义时的作用域，而不是在调用它的作用域中。

1.2 作用域链

在ES 5中，将作用域链称为词法环境。

当代码在一个作用域中执行时，会创建变量对象的一个作用域链（scope chain）。

作用域链的用途：保证对执行环境有权访问的所有变量和函数的有序访问。

作用域链的前端，始终都是当前执行的代码所在作用域的变量对象。如果这个环境是函数，则将其活动对象作为变量对象，然后对于每一个函数的形参，都命名为该活动对象的命名属性。

活动对象在最开始只包含一个变量，即arguments对象（在全局环境中是不存在的）。

看一个例子：

```
var name = 'p';
function test(){
  var name = 'tg';
  function get(){
    console.log(name);
  }
  get();
}
test(); // "tg"
```

当执行get()时，将创建函数get()的执行环境（调用对象），并将该对象置于作用域链的前端（开头），然后将函数test()的调用对象链接在之后，最后是全局对象，然后从作用域链的前端开始查找标识符name，很显然，变量name的值是"tg"；

作用域链：get() -> test() -> window

每次进入一个新的作用域，都会创建一个用于搜索变量和函数的作用域链。

函数的局部变量不仅有权访问函数作用域中的变量，而且有权访问其他包含（父）环境，乃至全局作用域。但是全局作用域只能访问在全局作用域中定义的变量和函数，不能直接访问局部作用域中的任何数据。

变量的作用域有助于确定应该何时释放内存。

1.3 JavaScript没有块级作用域

JavaScript中并没有块级作用域。也就是说，对于if、for、while、switch等块结构是没有作用域的。

```
if(true){
  var name = 'tg';
}
console.log(name); // "tg"
```

在if语句里定义了一个name变量，但它并不会像在函数内那样，但函数执行结束后就销毁，而是会一直存在，因为它被添加到了当前的执行环境（也就是全局环境）中。

对于for循环也是一样：

```
for(var i = 0; i < 10; i++){
  //循环体
}
console.log(i); // 10
```

(1) 声明变量

使用var声明的变量会自动被添加到最接近的环境中，比如在函数内部，最接近的环境就是函数的局部环境。如果初始化变量没有使用var声明，会自动被添加到全局环境中。

(2) 查询标识符

标识符的查询规则是：逐级向上查询，如果在局部环境中找到了该标识符，搜索过程就停止，变量就绪，否则，会继续向上查询，直到全局环境的变量对象，如果在全局环境中也没有找到这个标识符，就表示该变量尚未声明（通常会报错）。

```
var name = 'tg2';
function test(){
  var name = 'tg';
  console.log(name); // "tg"
}
test();
```

在上面的例子中，在函数内部已经找到了name标识符，所以返回的是"tg"

常用API合集

常用API合集

一、节点

1.1 节点属性

```

Node.nodeName    //返回节点名称，只读
Node.nodeType   //返回节点类型的常数值，只读
Node.nodeValue  //返回Text或Comment节点的文本值，只读
Node.textContent //返回当前节点和它的所有后代节点的文本内容，可读写
Node.baseURI    //返回当前网页的绝对路径

Node.ownerDocument //返回当前节点所在的顶层文档对象，即document
Node.nextSibling //返回紧跟在当前节点后面的第一个兄弟节点
Node.previousSibling //返回当前节点前面的、距离最近的一个兄弟节点
Node.parentNode  //返回当前节点的父节点
Node.parentElement //返回当前节点的父Element节点
Node.childNodes  //返回当前节点的所有子节点
Node.firstChild   //返回当前节点的第一个子节点
Node.lastChild   //返回当前节点的最后一个子节点

//parentNode接口
Node.children   //返回指定节点的所有Element子节点
Node.firstElementChild //返回当前节点的第一个Element子节点
Node.lastElementChild //返回当前节点的最后一个Element子节点
Node.childElementCount //返回当前节点所有Element子节点的数目。

```

1.2 操作

```

Node.appendChild(node) //向节点添加最后一个子节点
Node.hasChildNodes() //返回布尔值，表示当前节点是否有子节点
Node.cloneNode(true); // 默认为false(克隆节点)，true(克隆节点及其属性，以及后代)
Node.insertBefore(newNode,oldNode) // 在指定子节点之前插入新的子节点
Node.removeChild(node) //删除节点，在要删除节点的父节点上操作
Node.replaceChild(newChild,oldChild) //替换节点
Node.contains(node) //返回一个布尔值，表示参数节点是否为当前节点的后代节点。
Node.compareDocumentPosition(node) //返回一个7个比特位的二进制值，表示参数节点和当前
//ChildNode接口
Node.remove() //用于删除当前节点

```

```
Node.before() //  
Node.after()  
Node.replaceWith()
```

1.3 Document节点

1.3.1 Document节点的属性

```
document.doctype //  
document.documentElement //返回当前文档的根节点  
document.defaultView //返回document对象所在的window对象  
document.body //返回当前文档的<body>节点  
document.head //返回当前文档的<head>节点  
document.activeElement //返回当前文档中获得焦点的那个元素。  
  
//节点集合属性  
document.links //返回当前文档的所有a元素  
document.forms //返回页面中所有表单元素  
document.images //返回页面中所有图片元素  
document.embeds //返回网页中所有嵌入对象  
document.scripts //返回当前文档的所有脚本  
document.styleSheets //返回当前网页的所有样式表  
  
//文档信息属性  
document.documentElementURI //表示当前文档的网址  
document.URL //返回当前文档的网址  
document.domain //返回当前文档的域名  
document.lastModified //返回当前文档最后修改的时间戳  
document.location //返回location对象，提供当前文档的URL信息  
document.referrer //返回当前文档的访问来源  
document.title //返回当前文档的标题  
document.characterSet属性返回渲染当前文档的字符集，比如UTF-8、ISO-8859-1。  
document.readyState //返回当前文档的状态  
document.designMode //控制当前文档是否可编辑，可读写  
document.compatMode //返回浏览器处理文档的模式  
document.cookie //用来操作Cookie
```

1.3.2 Document节点的方法

(1) 读写方法

```
document.open() //用于新建并打开一个文档  
document.close() //不关闭open方法所新建的文档  
document.write() //用于向当前文档写入内容  
document.writeln() //用于向当前文档写入内容，尾部添加换行符。
```

(2) 查找节点

```

document.querySelector(selectors) //接受一个CSS选择器作为参数，返回第一个匹配该选择器的元素节点。
document.querySelectorAll(selectors) //接受一个CSS选择器作为参数，返回所有匹配该选择器的元素节点。
document.getElementsByTagName(tagName) //返回所有指定HTML标签的元素
document.getElementsByClassName(className) //返回包括了所有class名字符合指定条件的元素
document.getElementsByName(name) //用于选择拥有name属性的HTML元素（比如<form>、<radio>、<img>、<frame>、<embed>和<object>等）
document.getElementById(id) //返回匹配指定id属性的元素节点。
document.elementFromPoint(x,y) //返回位于页面指定位置最上层的Element子节点。

```

(3) 生成节点

```

document.createElement(tagName) //用来生成HTML元素节点。
document.createTextNode(text) //用来生成文本节点
document.createAttribute(name) //生成一个新的属性对象节点，并返回它。
document.createDocumentFragment() //生成一个DocumentFragment对象

```

(4) 事件方法

```

document.createEvent(type) //生成一个事件对象，该对象能被element.dispatchEvent()方法使用
document.addEventListener(type, listener, capture) //注册事件
document.removeEventListener(type, listener, capture) //注销事件
document.dispatchEvent(event) //触发事件

```

(5) 其他

```

document.hasFocus() //返回一个布尔值，表示当前文档之中是否有元素被激活或获得焦点。
document.adoptNode(externalNode) //将某个节点，从其原来所在的文档移除，插入当前文档，并返回插入后的新节点。
document.importNode(externalNode, deep) //从外部文档拷贝指定节点，插入当前文档。

```

1.4 Element节点

1.4.1 Element节点的属性

(1) 特性属性

```

Element.attributes //返回当前元素节点的所有属性节点
Element.id //返回指定元素的id属性，可读写
Element.tagName //返回指定元素的大写标签名
Element.innerHTML //返回该元素包含的HTML代码，可读写
Element.outerHTML //返回指定元素节点的所有HTML代码，包括它自身和包含的所有子元素，可读写

```

```
Element.className //返回当前元素的class属性，可读写
Element.classList //返回当前元素节点的所有class集合
Element.dataset //返回元素节点中所有的data-*属性。
```

(2) 尺寸属性

```
Element.clientHeight //返回元素节点可见部分的高度
Element.clientWidth //返回元素节点可见部分的宽度
Element.clientLeft //返回元素节点左边框的宽度
Element.clientTop //返回元素节点顶部边框的宽度
Element.scrollHeight //返回元素节点的总高度
Element.scrollWidth //返回元素节点的总宽度
Element.scrollLeft //返回元素节点的水平滚动条向右滚动的像素数值,通过设置这个属性可以改变
元素的滚动位置
Element.scrollTop //返回元素节点的垂直滚动向下滚动的像素数值
Element.offsetHeight //返回元素的垂直高度(包含border,padding)
Element.offsetWidth //返回元素的水平宽度(包含border,padding)
Element.offsetLeft //返回当前元素左上角相对于Element.offsetParent节点的垂直偏移
Element.offsetTop //返回水平位移
Element.style //返回元素节点的行内样式
```

(3) 节点相关属性

```
Element.children //包括当前元素节点的所有子元素
Element.childElementCount //返回当前元素节点包含的子HTML元素节点的个数
Element.firstElementChild //返回当前节点的第一个Element子节点
Element.lastElementChild //返回当前节点的最后一个Element子节点
Element.nextElementSibling //返回当前元素节点的下一个兄弟HTML元素节点
Element.previousElementSibling //返回当前元素节点的前一个兄弟HTML节点
Element.offsetParent //返回当前元素节点的最靠近的、并且CSS的position属性不等于static
的父元素。
```

1.4.2 Element节点的方法

(1) 位置方法

```
getBoundingClientRect()
// getBoundingClientRect返回一个对象，包含top,left,right,bottom,width,height // wi
dth、height 元素自身宽高
// top 元素上外边界距窗口最上面的距离
// right 元素右外边界距窗口最上面的距离
// bottom 元素下外边界距窗口最上面的距离
// left 元素左外边界距窗口最上面的距离
// width 元素自身宽(包含border,padding)
// height 元素自身高(包含border,padding)
```

```
getClientRects() //返回当前元素在页面上形参的所有矩形。  

// 元素在页面上的偏移量  

var rect = el.getBoundingClientRect()  

return {  

  top: rect.top + document.body.scrollTop,  

  left: rect.left + document.body.scrollLeft  

}
```

(2) 属性方法

Element.getAttribute() : 读取指定属性
Element.setAttribute() : 设置指定属性
Element.hasAttribute() : 返回一个布尔值, 表示当前元素节点是否有指定的属性
Element.removeAttribute() : 移除指定属性

(3) 查找方法

```
Element.querySelector()  

Element.querySelectorAll()  

Element.getElementsByTagName()  

Element.getElementsByClassName()
```

(4) 事件方法

Element.addEventListener() : 添加事件的回调函数
Element.removeEventListener() : 移除事件监听函数
Element.dispatchEvent() : 触发事件

```
//ie8
Element.attachEvent(eventName, listener)
Element.detachEvent(eventName, listener)

// event对象
var event = window.event || event;

// 事件的目标节点
var target = event.target || event.srcElement;

// 事件代理
ul.addEventListener('click', function(event) {
  if (event.target.tagName.toLowerCase() === 'li') {
    console.log(event.target.innerHTML)
  }
});
```

(5) 其他

```

Element.scrollIntoView() //滚动当前元素，进入浏览器的可见区域

//解析HTML字符串，然后将生成的节点插入DOM树的指定位置。
Element.insertAdjacentHTML(where, htmlString);
Element.insertAdjacentHTML('beforeBegin', htmlString); // 在该元素前插入
Element.insertAdjacentHTML('afterBegin', htmlString); // 在该元素第一个子元素前插入

Element.insertAdjacentHTML('beforeEnd', htmlString); // 在该元素最后一个子元素后面
插入
Element.insertAdjacentHTML('afterEnd', htmlString); // 在该元素后插入

Element.remove() //用于将当前元素节点从DOM中移除
Element.focus() //用于将当前页面的焦点，转移到指定元素上

```



二、CSS操作

(1) 类名操作

```

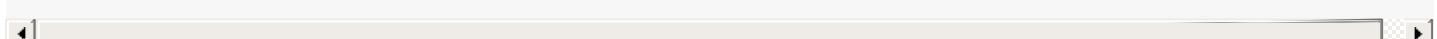
//ie8以下
Element.className //获取元素节点的类名
Element.className += ' ' + newClassName //新增一个类名

//判断是否有某个类名
function hasClass(element, className){
    return new RegExp(className, 'gi').test(element.className);
}

//移除class
function removeClass(element, className){
    element.className = element.className.replace(new RegExp('(^|\\b)' + className
    .split(' ').join('|') + '(\\b|$)', 'gi'), '');
}

//ie10
element.classList.add(className) //新增
element.classList.remove(className) //删除
element.classList.contains(className) //是否包含
element.classList.toggle(className) //toggle class

```



(2) style操作

```

element.setAttribute('style', '')
element.style.backgroundColor = 'red'

```

```

element.style.cssText //用来读写或删除整个style属性

element.style.setProperty(propertyName,value) //设置css属性
element.style.getPropertyValue(property) //获取css属性
element.style.removeProperty(property) //删除css属性
操作非内联样式
//ie8
element.currentStyle[attrName]
//ie9+
window.getComputedStyle(el,null)[attrName]
window.getComputedStyle(el,null).getPropertyValue(attrName)
//伪类
window.getComputedStyle(el,':after')[attrName]

```

三、对象

3.1 Object对象

(1) 生成实例对象

```
var o = new Object()
```

(2) 属性

```
Object.prototype //返回原型对象
```

(3) 方法

```

Object.keys(o) //遍历对象的可枚举属性
Object.getOwnPropertyNames(o) //遍历对象不可枚举的属性

```

对象实例的方法

valueOf() : 返回当前对象对应的值。
toString() : 返回当前对象对应的字符串形式。
toLocaleString() : 返回当前对象对应的本地字符串形式。
hasOwnProperty() : 判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。
isPrototypeOf() : 判断当前对象是否为另一个对象的原型。
propertyIsEnumerable() : 判断某个属性是否可枚举。

3.2 Array对象

(1) 生成实例对象

```
var a = new Array()
```

(2) 属性

```
a.length //长度
```

(3) Array.isArray()

```
Array.isArray(a) //用来判断一个值是否为数组
```

(4) Array实例的方法

`a.valueOf()` //返回数组本身

`a.toString()` //返回数组的字符串形式

`a.push(value, vlaue...)` //用于在数组的末端添加一个或多个元素，并返回添加新元素后的数组长度。

`pop()` //用于删除数组的最后一个元素，并返回该元素

`join()` //以参数作为分隔符，将所有数组成员组成一个字符串返回。如果不提供参数，默认用逗号分隔。

`concat()` //用于多个数组的合并。它将新数组的成员，添加到原数组的尾部，然后返回一个新数组，原数组不变。

`shift()` //用于删除数组的第一个元素，并返回该元素。

`unshift(value)` //用于在数组的第一个位置添加元素，并返回添加新元素后的数组长度。

`reverse()` //用于颠倒数组中元素的顺序，返回改变后的数组

`slice(start_index, upto_index);` //用于提取原数组的一部分，返回一个新数组，原数组不变。第一个参数为起始位置（从0开始），第二个参数为终止位置（但该位置的元素本身不包括在内）。如果省略第二个参数，则一直返回到原数组的最后一个成员。负数表示倒数第几个。

`splice(index, count_to_remove, addElement1, addElement2, ...);` //用于删除原数组的一部分成员，并可以在被删除的位置添加入新的数组成员，返回值是被删除的元素。第一个参数是删除的起始位置，第二个参数是被删除的元素个数。如果后面还有更多的参数，则表示这些就是要被插入数组的新元素。

`sort()` //对数组成员进行排序，默认是按照字典顺序排序。排序后，原数组将被改变。如果想让`sort`方法按照自定义方式排序，可以传入一个函数作为参数，表示按照自定义方法进行排序。该函数本身又接受两个参数，表示进行比较的两个元素。如果返回值大于0，表示第一个元素排在第二个元素后面；其他情况下，都是第一个元素排在第二个元素前面。

`map()` //对数组的所有成员依次调用一个函数，根据函数结果返回一个新数组。

`map(elem, index, arr)` //`map`方法接受一个函数作为参数。该函数调用时，`map`方法会将其传入三个参数，分别是当前成员、当前位置和数组本身。

`forEach()` //遍历数组的所有成员，执行某种操作，参数是一个函数。它接受三个参数，分别是当前位置的值、当前位置的编号和整个数组。

`filter()` //参数是一个函数，所有数组成员依次执行该函数，返回结果为`true`的成员组成一个新数组返回。该方法不会改变原数组。

`some()` //用来判断数组成员是否符合某种条件。接受一个函数作为参数，所有数组成员依次执行该函数，返回一个布尔值。该函数接受三个参数，依次是当前位置的成员、当前位置的序号和整个数组。只要有一个数组成员的返回值是`true`，则整个`some`方法的返回值就是`true`，否则`false`。

```
every() //用来判断数组成员是否符合某种条件。接受一个函数作为参数，所有数组成员依次执行该函数，返回一个布尔值。该函数接受三个参数，依次是当前位置的成员、当前位置的序号和整个数组。所有数组成员的返回值都是true，才返回true，否则false。
reduce() //依次处理数组的每个成员，最终累计为一个值。从左到右处理（从第一个成员到最后一个成员）
reduceRight() //依次处理数组的每个成员，最终累计为一个值。从右到左（从最后一个成员到第一个成员）
indexOf(s) //返回给定元素在数组中第一次出现的位置，如果没有出现则返回-1。可以接受第二个参数，表示搜索的开始位置
lastIndexOf() //返回给定元素在数组中最后一次出现的位置，如果没有出现则返回-1。
```



3.3 Number对象

(1) 生成对象

```
var n = new Number()
```

(2) Number对象的属性

Number.POSITIVE_INFINITY：正的无限，指向Infinity。
 Number.NEGATIVE_INFINITY：负的无限，指向-Infinity。
 Number.NaN：表示非数值，指向NaN。
 Number.MAX_VALUE：表示最大的正数，相应的，最小的负数为-Number.MAX_VALUE。
 Number.MIN_VALUE：表示最小的正数（即最接近0的正数，在64位浮点数体系中为5e-324），相应的，最接近0的负数为-Number.MIN_VALUE。
 Number.MAX_SAFE_INTEGER：表示能够精确表示的最大整数，即9007199254740991。
 Number.MIN_SAFE_INTEGER：表示能够精确表示的最小整数，即-9007199254740991。

(4) Number对象实例的方法

```
toString() //用来将一个数值转为字符串形式。可以接受一个参数，表示输出的进制。如果省略这个参数，默认将数值先转为十进制，再输出字符串；否则，就根据参数指定的进制，将一个数字转化成某个进制的字符串。
toFixed() //用于将一个数转为指定位数的小数，返回这个小数对应的字符串。
toExponential() //用于将一个数转为科学计数法形式。可传入一个参数，参数表示小数点后有效数字的位数，范围为0到20，超出这个范围，会抛出一个RangeError。
toPrecision() //用于将一个数转为指定位数的有效数字。
```

3.4 String 对象

(1) 生成实例对象

```
var s = new String()
```

(2) String对象的属性

```
s.length //返回字符串的长度
```

(3) 方法

```
s.charAt(index) //返回指定位置的字符
s.fromCharCode() //该方法的参数是一系列Unicode码点，返回对应的字符串。
s.charCodeAt(index) //返回给定位置字符的Unicode码点（十进制表示）
s.concat(s2) //用于连接两个字符串
s.slice(start,end) //用于从原字符串取出子字符串并返回，不改变原字符串。第一个参数是子字符串的开始位置，第二个参数是子字符串的结束位置（不含该位置）。如果参数是负值，表示从结尾开始倒数计算的位置，即该负值加上字符串长度。
s.substring(start,end) //用于从原字符串取出子字符串并返回，不改变原字符串。第一个参数表示子字符串的开始位置，第二个位置表示结束位置。
s.substr(start,length) //用于从原字符串取出子字符串并返回，不改变原字符串。第一个参数是子字符串的开始位置，第二个参数是子字符串的长度。如果第一个参数是负数，表示倒数计算的字符位置。如果第二个参数是负数，将被自动转为0，因此会返回空字符串。
s.indexOf(s) //返回给定元素在字符串中第一次出现的位置，如果没有出现则返回-1。可以接受第二个参数，表示搜索的开始位置
s.lastIndexOf() //返回给定元素在字符串中最后一次出现的位置，如果没有出现则返回-1。
s.trim() //用于去除字符串两端的空格，返回一个新字符串
s.toLowerCase() //用于将一个字符串全部转为小写，返回一个新字符串，不改变原字符串。
s.toUpperCase() //全部转为大写
s.localeCompare(s2) //用于比较两个字符串。它返回一个整数，如果小于0，表示第一个字符串小于第二个字符串；如果等于0，表示两者相等；如果大于0，表示第一个字符串大于第二个字符串。
s.match(regexp) //用于确定原字符串是否匹配某个子字符串，返回一个数组，成员为匹配的第一个字符串。如果没有找到匹配，则返回null。
s.search() //返回值为匹配的第一个位置。如果没有找到匹配，则返回-1。
s.replace oldValue,newValue //用于替换匹配的子字符串，一般情况下只替换第一个匹配（除非使用带有g修饰符的正则表达式）。
s.split() //按照给定规则分割字符串，返回一个由分割出来的子字符串组成的数组。还可传入第二个参数，决定了返回数组的成员数。
```

3.5 Math对象

(1) 属性

Math.E : 常数e。
 Math.LN2 : 2的自然对数。
 Math.LN10 : 10的自然对数。
 Math.LOG2E : 以2为底的e的对数。
 Math.LOG10E : 以10为底的e的对数。
 Math.PI : 常数Pi。
 Math.SQRT1_2 : 0.5的平方根。
 Math.SQRT2 : 2的平方根。

(2) 数学方法

```

Math.abs() : 返回参数的绝对值
Math.ceil() : 向上取整，接受一个参数，返回大于该参数的最小整数。
Math.floor() : 向下取整
Math.max(n, n1, ...) : 可接受多个参数，返回最大值
Math.min(n, n1, ...) : 可接受多个参数，返回最小值
Math.pow(n, e) : 指数运算，返回以第一个参数为底数、第二个参数为幂的指数值。
Math.sqrt() : 返回参数值的平方根。如果参数是一个负值，则返回NaN。
Math.log() : 返回以e为底的自然对数值。
Math.exp() : 返回e的指数，也就是常数e的参数次方。
Math.round() : 四舍五入
Math.random() : 返回0到1之间的一个伪随机数，可能等于0，但是一定小于1。

```

(3) 三角函数方法

```

Math.sin() : 返回参数的正弦
Math.cos() : 返回参数的余弦
Math.tan() : 返回参数的正切
Math.asin() : 返回参数的反正弦（弧度值）
Math.acos() : 返回参数的反余弦（弧度值）
Math.atan() : 返回参数的反正切（弧度值）

```

3.6 JSON对象

(1) 方法

```

JSON.stringify()
//用于将一个值转为字符串。该字符串应该符合JSON格式，并且可以被JSON.parse方法还原。
// (JSON.stringify(obj, selectedProperties)) 还可以接受一个数组，作为第二个参数，指定需要转成字符串的属性。
//还可以接受第三个参数，用于增加返回的JSON字符串的可读性。如果是数字，表示每个属性前面添加的空格（最多不超过10个）；如果是字符串（不超过10个字符），则该字符串会添加在每行前面。

JSON.parse() //用于将JSON字符串转化成对象。

```

3.7 console对象

(1) 方法

```

console.log(text, text2, ...) //用于在console窗口输出信息。它可以接受多个参数，将它们的结果连接起来输出。如果第一个参数是格式字符串（使用了格式占位符），console.log方法将依次用后面的参数替换占位符，然后再进行输出。
console.info() //在console窗口输出信息，同时，会在输出信息的前面，加上一个蓝色图标。
console.debug() //在console窗口输出信息，同时，会在输出信息的前面，加上一个蓝色图标。
console.warn() //输出信息时，在最前面加一个黄色三角，表示警告；

```

```
console.error() //输出信息时，在最前面加一个红色的叉，表示出错，同时会显示错误发生的堆栈
console.table() //可以将复合类型的数据转为表格显示。
console.count() //用于计数，输出它被调用了多少次。
console.dir() //用来对一个对象进行检查（inspect），并以易于阅读和打印的格式显示。
console.dirxml() //用于以目录树的形式，显示DOM节点。
console.assert() //接受两个参数，第一个参数是表达式，第二个参数是字符串。只有当第一个参数为false，才会输出第二个参数，否则不会有任何结果。

//这两个方法用于计时，可以算出一个操作所花费的准确时间。
console.time()
console.timeEnd()
//time方法表示计时开始，timeEnd方法表示计时结束。它们的参数是计时器的名称。调用timeEnd方法之后，console窗口会显示“计时器名称：所耗费的时间”。

console.profile() //用来新建一个性能测试器（profile），它的参数是性能测试器的名字。
console.profileEnd() //用来结束正在运行的性能测试器。

console.group()
console.groupEnd()
//上面这两个方法用于将显示的信息分组。它只在输出大量信息时有用，分在一组的信息，可以用鼠标折叠/展开。
console.groupCollapsed() //用于将显示的信息分组，该组的内容，在第一次显示时是收起的（collapsed），而不是展开的。

console.trace() //显示当前执行的代码在堆栈中的调用路径。
console.clear() //用于清除当前控制台的所有输出，将光标回置到第一行。
```

SVG

SVG：可伸缩的矢量图形

这里只是简要的介绍一下SVG，如要深入，推荐看《SVG经典入门》

1、SVG

1.1 概述

SVG是一种用于描述图形的XML语法。由于结构是XML格式，使得它可以插入HTML文档，成为DOM的一部分，然后用JavaScript和CSS进行操作。

一个简单的SVG文件如下：

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 1000 1000" id="mySvg">
  <rect x="100" y="200" width="800" height="600" stroke="black" stroke-width="2
  5" fill="red"/>
</svg>
```

1.2 使用方法

要使用SVG有很多方法，最简单的就是直接将SVG代码嵌入到HTML中：

```
<body>
  <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 1000 1000" id="mySvg">
    <rect x="100" y="200" width="800" height="600" stroke="black" stroke-width="2
    5" fill="red"/>
  </svg>
</body>
```

SVG代码也可以单独写在一个文件中，后缀是 ".svg"，然后用在

`、<object>、<embed>、<iframe>` 等标签，以及CSS的background-image属性，将这个文件插入网页。

```


<object data="example.svg" type="image/svg+xml"></object>

<embed src="example.svg" type="image/svg+xml">

<iframe src="example.svg"></iframe>
```

1.3 基本图形

(1) 矩形

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="10" y="20" rx="5" ry="5" width="200" height="100"
    style="fill:rgb(0,0,255);stroke-width:1;stroke:rgb(0,0,0);fill-opacity:0.1;opacity:0.5" />
</svg>
```

属性说明：

rect 元素的 width 和 height 属性可定义矩形的高度和宽度

style 属性用来定义 CSS 属性

- fill 属性定义矩形的填充颜色 (rgb 值、颜色名或者十六进制值)
- fill-opacity 属性定义填充颜色的透明度
- stroke-width 属性定义矩形边框的宽度
- stroke 属性定义矩形边框的颜色
- stroke-opacity 属性定义笔触颜色的透明度
- opacity 属性定义元素的透明度
- rx 和 ry 属性定义矩形圆角

下面的其他图形都支持 style 属性，而且可以单独作为属性添加。

(2) 圆形

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <circle cx="100" cy="50" r="40" stroke="black"
    stroke-width="2" fill="red"/>
</svg>
```

属性说明：

- cx 和 cy 属性定义圆点的 x 和 y 坐标，如果省略，则默认为 (0,0)
- r 属性定义圆的半径

(3) 椭圆

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <ellipse cx="300" cy="80" rx="100" ry="50"
    style="fill:yellow;stroke:purple;stroke-width:2"/>
</svg>
```

属性说明：

- cx 和 cy 属性定义椭圆的中心的 x 和 y 坐标

- rx定义水平半径
- ry定义垂直半径

(4) 直线

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <line x1="0" y1="0" x2="200" y2="200"
    style="stroke:rgb(255,0,0);stroke-width:2"/>
</svg>
```

属性说明：

- x1和y1表示起始点的x和y坐标
- x2和y2表示结束点的x和y坐标

(5) 多边形

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <polygon points="200,10 250,190 160,210"
    style="fill:lime;stroke:purple;stroke-width:1"/>
</svg>
```

属性说明：

- points属性定义多边形每个角的x和y坐标，x和y坐标之间用逗号隔开，每个坐标点之间用空格隔开

(6) 曲线

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <polyline points="20,20 40,25 60,40 80,120 120,140 200,180"
    style="fill:none;stroke:black;stroke-width:3" />
</svg>
```

- points属性定义多边形每个角的x和y坐标，x和y坐标之间用逗号隔开，每个坐标点之间用空格隔开

(7) 文本

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <text x="0" y="15" fill="red">I love SVG</text>
</svg>
```

(8) 路径

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <path d="M150 0 L75 200 L225 200 Z" />
</svg>
```

属性说明：

- d属性定义一组路径数据，一对坐标值的x和y坐标之间可以用空格或逗号隔开，但坐标对与坐标对之间只能用空格分隔。
- pathLength属性定义路径总长度，不允许负值

绘图指令：

```
M    moveto, 表示移动，后跟着坐标点，比如150 0
L    lineto, 表示连接（也可以说是绘制直线），后跟着坐标点，比如75 200
H    horizontal lineto, 绘制水平线段
V    vertical lineto, 绘制垂直线段
C    curveto, 绘制普通的三次贝塞尔曲线 (C x1 y1 x2 y2 destx desty)
S    smooth curveto : 绘制光滑的三次贝塞尔曲线 (S x2 y2 destx desty)
Q    quadratic Bézier curve : 绘制普通的二次贝塞尔曲线
T    smooth quadratic Bézier curveto : 绘制光滑的二次贝塞尔曲线
A    elliptical Arc : 绘制椭圆弧
Z    closepath, 表示完成闭合路径，即将路径首尾相连，构成闭合图形
```

大写表示绝对定位，小写表示相对定位。

1.4 装饰SVG

1.4.1 stroke属性

- stroke：定义一条线，文本或元素轮廓颜色
- stroke-width：定义了一条线，文本或元素轮廓厚度
- stroke-linecap：定义不同类型的开放路径的终结（可能值：butt、round、square）
- stroke-dasharray：用于创建虚线

1.4.2 滤镜

(1) 模糊效果

所有的SVG滤镜都定义在 `<defs>` 元素中，

`<filter>` 标签用来定义SVG滤镜

例子：`<feGaussianBlur>` 元素是用于创建模糊效果

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <filter id="f1" x="0" y="0">
      <feGaussianBlur in="SourceGraphic" stdDeviation="15" />
```

```

</filter>
</defs>
<rect width="90" height="90" stroke="green" stroke-width="3"
      fill="yellow" filter="url(#f1)" />
</svg>

```

(2) 阴影

<feOffset> 元素是用于创建阴影效果。

```

<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <filter id="f1" x="0" y="0" width="200%" height="200%">
      <feOffset result="offOut" in="SourceGraphic" dx="20" dy="20" />
      <feBlend in="SourceGraphic" in2="offOut" mode="normal" />
    </filter>
  </defs>
  <rect width="90" height="90" stroke="green" stroke-width="3"
        fill="yellow" filter="url(#f1)" />
</svg>

```

1.4.3 渐变

(1) 线性渐变

<linearGradient> 元素用于定义线性渐变。

```

<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <linearGradient id="grad1" x1="0%" y1="0%" x2="100%" y2="0%">
      <stop offset="0%" style="stop-color:rgb(255,255,0);stop-opacity:1" />
      <stop offset="100%" style="stop-color:rgb(255,0,0);stop-opacity:1" />
    </linearGradient>
  </defs>
  <ellipse cx="200" cy="70" rx="85" ry="55" fill="url(#grad1)" />
</svg>

```

(2) 放射性渐变

<radialGradient> 元素用于定义放射性渐变。

```

<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <radialGradient id="grad1" cx="50%" cy="50%" r="50%" fx="50%" fy="50%">
      <stop offset="0%" style="stop-color:rgb(255,255,255);stop-opacity:0" />
      <stop offset="100%" style="stop-color:rgb(0,0,255);stop-opacity:1" />
    </radialGradient>
  </defs>

```

```
</defs>
<ellipse cx="200" cy="70" rx="85" ry="55" fill"url(#grad1)" />
</svg>
```

错误处理机制

错误处理机制

1、try-catch语句

ECMA-262第3版引入了 `try-catch` 语句，作为JavaScript中处理异常的一种标准方式。

语法：

```
try{
    // 可能会导致错误的代码
}catch (error){
    // 在错误发生时怎么处理
}
```

也就是说，我们应该把所有可能会抛出错误的代码都放在try语句块中，而把那些用于错误处理代码放在catch块中。

`try-catch` 语句的逻辑是：如果try块中的任何代码发生了错误，就会立即退出代码执行过程，然后接着执行catch块。此时，catch块会接收到一个包含错误信息的对象。

注意：即使你不想使用这个错误对象，也要给它起个名字。

虽然这个对象在不同浏览器中可能包含不同信息，但是都有一个保存着错误消息的`message`属性，还有一个保存错误类型的`name`属性（并不是所有浏览器都有）。

```
try{

}catch (error){
    console.log(error.message);
}
```

在跨浏览器编程时，最好还是只使用`message`属性。

1.1 finally子句

当使用 `finally` 子句时，其代码无论如何都会执行，也就是说，不管是正常执行还是出错了， `finally` 子句都会执行。甚至 `return` 语句，也不会阻止 `finally` 子句的执行。

看下面的例子：

```
function test(){
    try{
        console.log('a');
        return 2;
    }catch(error){
        console.log('b');
    }
}
```

```

}finally{
    console.log('c');
}
console.log(test());
//结果
a
c
2

```

从运行结果，我们可以看到，`return` 语句并没有阻止 `finally` 子句的执行，而且是在 `finally` 子句执行后才会返回 `return` 语句的值。

2、错误类型

执行代码期间可能会发生的错误有多种类型。每种错误都有对应的错误类型，而当错误发生时，会抛出相应类型的错误对象。

ECMA-262定义了下列7中错误类型：

```

Error
EvalError
RangeError
ReferenceError
SyntaxError
TypeError
URIError

```

`Error`是基类型，其他错误类型都继承自该类型。所有错误类型共享了一组相同的属性。

(1) `EvalError`类型

`EvalError`类型的错误会在使用`eval()`函数而发生异常时抛出。简单的说，如果没有把`eval()`当成函数调用，就会抛出异常。比如：

```

new eval() // 抛出EvalError
eval = foo; // 抛出EvalError

```

注意：在ES5中已经不在出现了。

(2) `RangeError`类型

`RangeError`类型的错误会在数值超出相应范围时触发。主要有几种情况，一是数组长度为负数，二是`Number`对象的方法参数超出范围，以及函数堆栈超过最大值。

```

var item = new Array(-20); // 抛出RangeError异常

```

(3) `ReferenceError`类型

在找不到对象的情况下，会发生ReferenceError。通常，在访问不存在的变量时，就会发生这种错误。

```
var obj = x; // 在x并未声明的情况下抛出ReferenceError
```

(4) SyntaxError类型

SyntaxError是解析代码时发生的语法错误。

```
var 1a; // 变量名错误，抛出SyntaxError
```

(5) TypeError类型

在变量中保存着意外的类型，或在访问不存在的方法时，都会导致这种错误。

```
var o = new 10; // 抛出TypeError
```

(6) URIError类型

URIError是URI相关函数的参数不正确时抛出的错误，主要涉及encodeURI()、decodeURI()、encodeURIComponent()、decodeURIComponent()、escape()和unescape()这六个函数。

2.2 抛出错误

与try-catch语句相配的还有一个throw操作符，用于随时抛出自定义错误。抛出错误时，必须要给throw操作符指定一个值，这个值可以是任何类型。

```
throw 1;
throw 'tg';
throw true;
throw {name: 'tg'};
```

上面的代码都是有效的。

在遇到throw操作符时，代码会立即停止运行。仅当有try-catch语句捕获到被抛出的值时，代码才会继续执行。

2.3 Error对象

所有抛出的错误都是Error构造函数的实例。Error构造函数接受一个参数，表示错误提示，可以从实例的message属性读到这个参数。

在JavaScript中，Error对象的实例必须有message属性，表示出错时的提示信息。在大多数JavaScript引擎中，Error实例还可能有name和stack属性，分别表示错误的名称和错误的堆栈。

```
var err = new Error('出错了');
err.message; // "出错了"
```

2.4 自定义错误

我们还可以创建自定义错误消息，最常用的错误类型是Error、RangeError、ReferenceError和TypeError。

```
throw new Error('报错了');
throw new RangeError('数组长度错误');
```

另外，利用原型链还可以通过继承Error来创建自定义错误类型：

```
function CustomError(message){
  this.name = 'CustomError';
  this.message = message;
}

CustomError.prototype = new Error();
throw new CustomError('我的错误信息');
```

3、错误 (error) 事件

任何没有通过try-catch处理的错误都会触发window对象的error事件。

onerror事件处理程序不会创建event对象，但它接受三个参数：错误消息、错误所在的URL和行号。

```
window.onerror = function(message, url, line){
};
```

当你在事件处理程序中返回false，可以阻止浏览器报告错误的默认行为

```
window.onerror = function(message, url, line){
  return false;
};
```

4、调试技术

4.1 alert方法

在以前，大多数都是在要调试的代码中插入alert()函数，看是否执行到这一步来判断哪里出错，这种方式比较麻烦，因为alert()会阻止后续代码的执行（除非你关闭了alert弹窗），而且调试后还要清理。

4.2 console

随着浏览器的不断改善，现在的浏览器都有JavaScript控制台，我们可以向这些控制台输出消息，比如最常用的console对象，它的常用方法如下：

- error(message)：将错误消息记录到控制台
- info(message)：将信息性消息记录到控制台
- log(message)：将一般消息记录到控制台

- warn(message) : 将警告消息记录到控制台

```
function test(){  
    console.log('结果：' + (1 + 2));  
}
```

4.3 throw

使用throw抛出错误。

JavaScript开发技巧合集

开发技巧

1、使用var声明变量

如果给一个没有声明的变量赋值，默认会作为一个全局变量（即使在函数内赋值）。要尽量避免不必要的全局变量。

2、行尾使用分号

虽然JavaScript允许省略行尾的分号，但是有时不注意的省略，会导致不必要的错误。建议在可用可不用行尾分号的地方加上分号。

3、获取指定范围内的随机数

```
var getRandom = function(max, min) {
    min = arguments[1] || 0;
    return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

上面的函数接受一个你希望的随机最大数和一个你希望的随机最小数。

4、打乱数字数组的顺序

```
var sortArray = array.sort(function(){
    return Math.random() - 0.5;
});
```

5、取出数组中的随机项

```
var ran = array[Math.floor(Math.random() * array.length)];
```

6、去除字符串的首尾空格

```
var s = string.trim();
```

7、类数组对象转为数组

比如：类数组对象遍历：

```
Array.prototype.forEach.call(argumens, function(value){
})
```

DOM的 NodeList 和 HTMLCollection 也是类数组对象

8、获取数组中的最大值和最小值

```
var max = Math.max.apply(Math, array);
var min = Math.min.apply(Math, array);
```

9、清空数组

array.length = 0;

array = [];

10、保留指定小数位

```
var num = num.toFixed(2);
```

返回字符串，保留两位小数

11、使用for-in循环来遍历对象的属性

```
for(var key in object) {
    // object[key]
}
```

不要用for-in来遍历数据

12、获取某月天数

```
function getMonthDay(date){
    date = date || new Date();
    if(typeof date === 'string') {
        date = new Date(date);
    };
    date.setDate(32);
    return 32 - date.getDate();
}
```

传入date参数，可以是字符串、日期对象实例；为空表示当月天数

13、浮点数问题

```
0.1 + 0.2 = 0.3000000000000004 != 0.3
```

JavaScript的数字都遵循IEEE 754标准构建，在内部都是64位浮点小数表示

14、JSON序列化和反序列化

使用 `JSON.stringify()` 来将JavaScript对象序列化为有效的字符串。

使用 `JSON.parse()` 来将有效的字符串转换为JavaScript对象。

在AJAX传输数据时很有用

15、使用 “`==`” 替换 “`=`”

相等运算符 (`==`) 在比较时会将操作数进行相应的类型转换，而全等运算符 (`==`) 不会进行类型转换。

16、避免使用with()

使用with()可以把变量加入到全局作用域中，因此，如果有其它的同名变量，一来容易混淆，二来值也会被覆盖。

17、不要使用eval()或函数构造器

eval()和函数构造器 (Function constructor) 的开销较大，每次调用，JavaScript引擎都要将源代码转换为可执行的代码。

18、简化if语句

```
if (condition) {
  fn();
}
```

可替换成：

```
condition && fn();
```

19、给可能省略的参数赋默认值

```
function test(a, b){
  a = a || '1';
}
```

20、给数组循环中缓存length的值

如果你确定循环中数组的长度不会变化，那么你可以这样：

```
var length = array.length;
for(var i = 0; i < length; i++) {
}
```

可以避免在每次迭代都将会重新计算数组的大小，提高效率

21、合并数组

对于小数组，我们可以这样：

```
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

var arr3 = arr1.concat(arr2); // [1, 2, 3, 4, 5, 6]
```

不过，concat()这个函数并不适合用来合并两个大型的数组，因为其将消耗大量的内存来存储新创建的数组。在这种情况下，可以使用 `Array.prototype.push.apply(arr1, arr2)` 来替代创建一个新数组。这种方法不是用来创建一个新的数组，其只是将第一个第二个数组合并在一起，同时减少内存的使用：

```
Array.prototype.push.apply(arr1, arr2);

console.log(arr1); // [1, 2, 3, 4, 5, 6]
```

22 枚举对象“自身”的属性

`for...in`除了枚举对象“自身”的属性外，还会枚举出继承过来的属性。

```
var hasOwn = Object.prototype.hasOwnProperty;

var obj = {name: 'tg', age: 24};

for(var name in obj) {
  if (hasOwn.call(obj, name)) {
    console.log(name + ': ' + obj[name]);
  }
}

// name : tg
// age : 24
```

编程风格

编风格程

作为前端开发人员，我相信每一个人都或多或少的用到原生的JavaScript，也正是因为用的人多，导致编码风格也是多种多样的，而不规范的编码风格，不仅会导致一些奇怪的问题出现，而且不利于后期维护和提高执行效率。

基于本人也在开发中因为规范而出现各种问题，我特意的整理了一下JavaScript编码规范（并不强制，只是推荐）。

1、变量

声明变量必须加上 `var` 关键字，而且每个 `var` 只声明一个变量，增加可读写。

推荐：

```
var name = 'TG';
var sex = 'man';
```

不推荐：

```
var name = 'TG', sex = 'man';
```

2、常量

常量的命名方式：用大写字符，并用下划线分隔。尽量不要使用`const`关键词。

```
var MY_NAME = 'TG';
```

原因：IE并不支持`const`

3、行尾分号

总是使用分号，对于可用可不用的行尾分号，使用分号。

4、嵌套函数

可以使用，可以减少重复代码，隐藏帮助函数等好处

5、块内函数声明

不要在块内声明一个函数

不推荐：

```
if(a){
  function foo(){}
}
```

推荐：

```
if(a){  
    var foo = function(){  
}
```

6、异常处理

可以使用

```
try{}catch(e){}  
  
throw exception
```

7、eval()

只用于解析序列化串

原因：eval()会让程序执行的比较混乱。

8、with(){}

尽量少用。

9、缩进

用4个空格作为一缩进，而不是使用tab

原因：因为在不同浏览器上，tab的长度不一。

10、字符串过长截取

每行代码不超过80个字符。如代码过长，可使用+运算符拼接。

原因：过长会导致需要拖动横向滚动条才看得到后面的代码，降低开发效率，而且在复制黏贴时有可能错漏。

11、大括号

区块起首的大括号，不要另起一行

推荐：

```
if (true) {  
}
```

不推荐：

```
if (true)  
{
```

```
}
```

12、构造函数

对于构造函数，命名采用首字母大写，其他函数一律小写。

原因：可读性，区分构造函数和普通函数

13、注释

合理的加上注释，有利于后期维护，提高可读性。

14、{}和[]

使用{}代替new Object()，使用[]代替new Array()

15、单引号(')'

尽量使用单引号(')，只在JSON文件中使用双引号。

16、变量和函数声明

变量名和函数名在JavaScript机制下会发生声明提升（也就是会提前到顶部声明），所以建议变量和函数应该在使用前声明。

17、使用==和!=代替==和!=

在JavaScript中，比较运算符进行计算时会进行强制转换，==和!=会产生一些意想不到的结果，所以应该用“严格相等” ===。

18、换行

在语句块和下一个语句之间留一个空行，提高可读性。

19、命名

构造函数或类名使用驼峰式命名

20、嵌入规则

JavaScript程序应该尽量放在js的文件中。

21、命名规则

JavaScript 中的标识符的命名规则：

以字母、下划线''或美元符号'\$'开头

允许名称中包含字母，数字，下划线''和美元符号'\$'

区分大小写

变量、属性和函数名应该用驼峰式：

```
var isLogin = false;
```

私有函数用下划线开头：

```
function getFirstName(){
    function _getName(){}
}
```

```
}
```

构造函数和类名应该首字母大写。

对象中私有变量和函数以下划线开头。

22、语句

对于复合语句，if, for, while, do, switch, try ... catch 等代码体，函数定义的函数体，对象的定义等都需要放在花括号'{}'里面。

'{' 应在行末，标志代码块的开始。

}'' 应在一行开头，标志代码块的结束，同时需要和'{'所在行的开始对齐，以表明一个完整的复合语句段。这样可以极大地提高代码的可阅读性，控制逻辑能清晰地表现出来。

被包含的代码段应该再缩进 4 个空格。

即使被包含的代码段只有一句，也应该用花括号'{}'包含。尽管不用花括号代码也不会错，但如若需要增加语句的话，则较容易因花括号遗漏而引起的编译错误或逻辑错误。

return语句在使用时也需注意，如果用表达式的执行作为返回值，应该把表达式和 return 放在同一行中，以免换行符被误解析为语句的结束而引起返回错误。return 关键字后若没有返回表达式，则返回 undefined。构造器的默认返回值为 this。

```
return a + b;
```

23、方法链（调用链）

如果使用方法链，应该每行只调用一个方法：

Animal

```
.getName()  
.getFirstName()
```

24、使用三元运算符

三元运算符不应该用在一行，应该分割成多行替代。

推荐：

```
var foo = (a === b)  
? 1  
: 2;
```

不推荐：

```
var foo = (a === b) ?1 : 2;
```

25、逗号

对于数组和对象不要使用多余的","

不推荐：

```
var arr = [1, 2, ]  
var person = {  
    name: 'TG'  
};
```

原因：IE不兼容

26、for-in

对于数组，尽量避免使用for-in

垃圾回收机制

##垃圾收集##

3. 垃圾收集

JavaScript具有自动垃圾收集机制，也就是说，执行环境会负责管理代码执行过程中使用的内存。

垃圾收集机制的原理：找出那些不再继续使用的变量，然后释放其占用的内存。

垃圾收集器是按固定的时间间隔，周期性地执行回收操作。

垃圾收集器如何判断哪些变量不需要使用了呢？常见的有两种方式

(1) 标记清除

当变量进入环境（比如在函数中声明一个变量）时，就将这个变量标记为“进入环境”。

标记变量的方式可以是任意的。

垃圾收集器在运行时会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记，而在此之后再被标记的变量将被视为待删除的变量，原因是环境中的变量以及无法访问到这些变量了，最后完成内存清除工作，销毁那些带标记的值并回收它们占用的内存空间。

(2) 引用计数

引用计数是不太常见的垃圾收集策略。引用计数可以说是跟踪记录每个值被引用的次数。比如：当声明了一个变量并将一个引用类型值赋给变量时，则这个值的引用次数是1，当同一个值又被赋给另一个变量，则该值的引用次数加1，相反，当包含对这个值引用的变量取得了另外一个值（也就是不指向前面的值）时，这个值的引用次数减1，当这个值的引用次数为0时，就会被销毁并释放内存空间了。

解除引用：一旦数据不再有用，最好通过将其值设置为null来释放其引用。