JavaScript

前言

> JavaScript之父 Brendan Eich（布兰登·艾奇）

在JavaScript诞生的前几年，有人说:

JavaScript是一门四不像的语言;JavaScript是一门没有规范的语言;JavaScript是一门兼容糟糕的语言;JavaScript是一门不精准的语言;JavaScript是一个半成品语言;JavaScript是一门糟糕的语言;JavaScript只是一个玩具胶水语言;

这些声音从JavaScript面世之初就一直伴随着她，声音的主人不乏已掌握多门语言的coding老兵，每一条负面都事实支撑。就连JavaScript之父也曾经说过:

> "与其说我爱Javascript，不如说我恨它。它是C语言和Self语言一夜情的产物。十八世纪英国文学家约翰逊博士说得好：'它的优秀之处并非原创，它的原创之处并不优秀。'（the part that is good is not original, and the part that is original is not good.）"

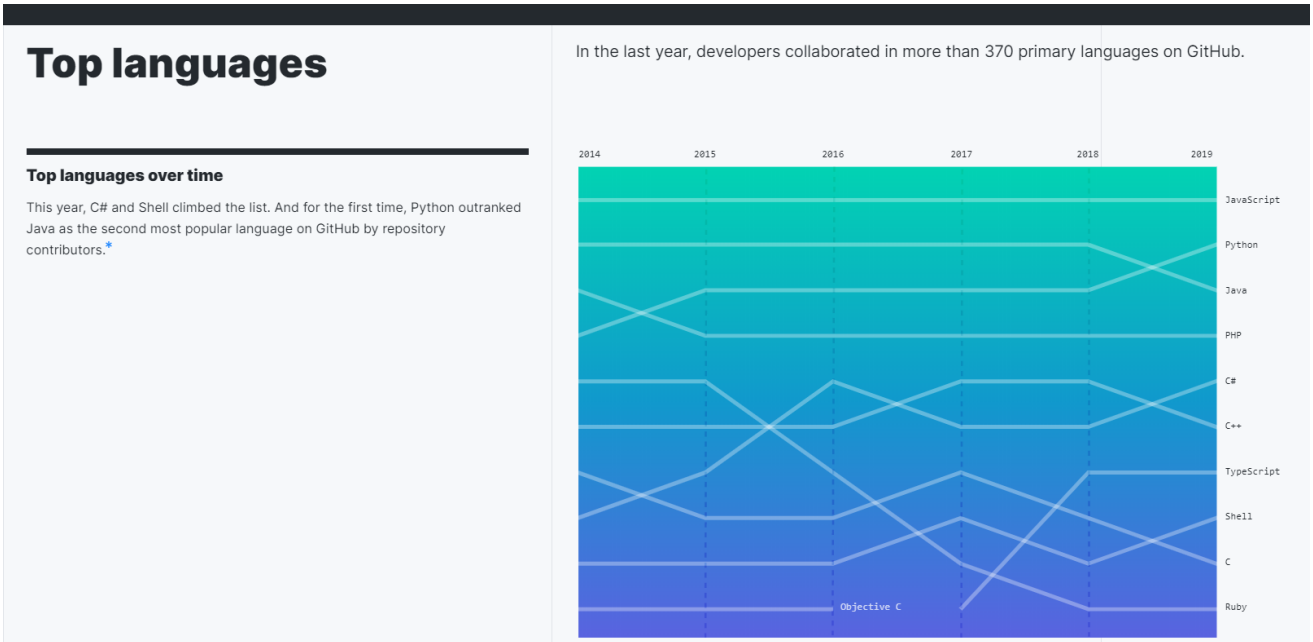
Ruby的设计者——松本行弘为此感叹：

> “这样的出身，得到这样的成功，还真让人出乎意料，……”，“但由于开发周期短，确实也存在着不足……”。

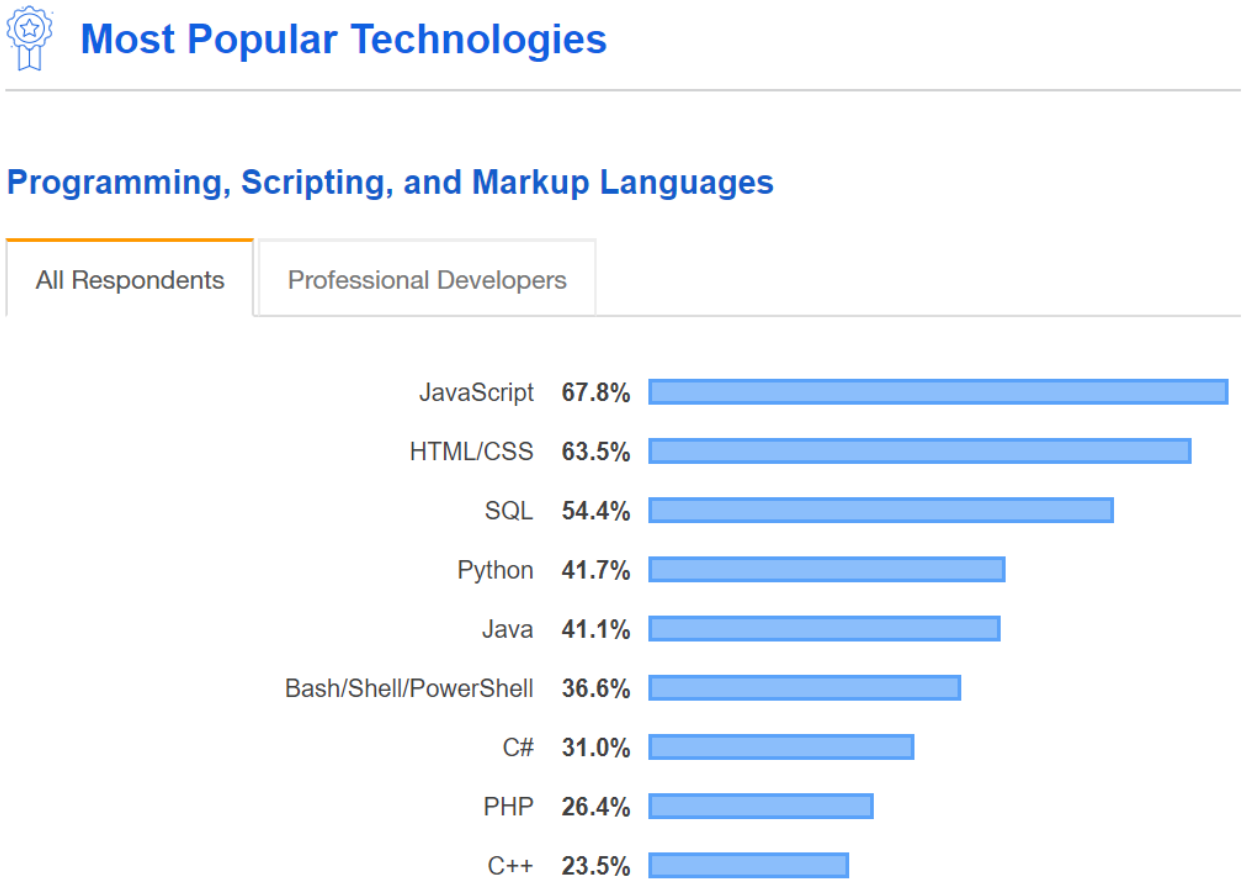
Douglas Crockford写了一本《JavaScript：The Good Parts》，在书中他这样写到：

> JavaScript建立在一些非常好的想法和少数非常坏的想法之上。
>
> 那些非常好的想法包括函数、弱类型、动态对象和一个富有表现力的对象字面量表示法，而那些坏的想法包括基于全局变量的编程模型、缺乏块作用域、“保留”了一堆根本没用到的保留字，不支持真正的数组（它所提供的类数组对象性能不好）等等。
>
> 还有一些是“鸡肋”，比如with语句，原始类型的包装对象，new,void等等

但如今，JavaScript已经成为大部分全球开发者与编程爱好者最常用/最喜欢的语言之一。



> [github Octoverse 调查报告][https://octoverse.github.com/] 合作开发中应用最多的语言排行



> [stackoverflow 2019年度调查报告][https://insights.stackoverflow.com/survey/2019#developer-profile-developer-roles-all-respondent] 最受欢迎的语言排行

这中间到底发生了什么，为什么会导致如此大的变化，让我们一起与JavaScript相关的历史年表；

时间	事件
1990年	万维网诞生
1992年	第一个浏览器诞生
1994年	Netscape（网景）成立开发第一代Netscape Navigator浏览器
1995年	Mocha诞生，之后改为LiveScript,最后与sun公司达成协议改为javascript
1996年	微软开发JScript,Netscape公司将JavaScript提交给国际标准化组织ECMA
1997年	ECMAscript1.0版发布 JavaScript进入标准化时代 ECMA-262发布
1998年	ECMAScript 2.0版发布。
1999年	ECMAScript 3.0版发布，成为JavaScript的通行标准，得到了广泛支持。
2004年	Gmail发布 Dojo诞生
2005年	Ajax 即“ **A**synchronous **J**avascript **A**nd **X**ML” AJAX 推广 CouchDB基于json格式的数据库
2006年	XMLHttpRequest被w3c纳入正式标准 同年 jQuery发布
2008年	V8引擎发布
2009年	ECMAScript 5.0发布 & node.js诞生
2010年	Express 发布 & angular发布 NPM、BackboneJS和RequireJS 诞生
2011年	React原型成立
2012年	Webpack诞生
2013年	mongodb 2.4* 开始支持JavaScript ELECTRON诞生 HTML5.1 发布
2014年	Vue.js 发布
2015年	ECMAScript 6正式发布，并且更名为 ECMAScript 2015 ， iotjs 发布
2016年	ECMAScript 2016发布
2017	ECMAScript 2017发布 主流浏览器全面支持 WebAssembly

个人观点: 真香
JavaScript的起步非常糟糕, 有着这样那样的问题让人诟病 , 但是他的灵魂足够有趣; 让人们乐于为他添砖加瓦,修枝剪叶,每一步都恰巧踩在了时代的脉搏上; The lucky JavaScript

JavaScript 的基本语法和对象体系，是模仿 Java 而设计的。但是，JavaScript 没有采用 Java 的静态类型。正是因为 JavaScript 与 Java 有很大的相似性，所以这门语言才从一开始的 LiveScript 改名为 JavaScript。基本上，JavaScript 这个名字的原意是“很像Java的脚本语言”。

JavaScript 语言的函数是一种独立的数据类型，以及采用基于原型对象（prototype）的继承链。这是它与 Java 语法最大的两点区别。JavaScript 语法要比 Java 自由得多。

另外，Java 语言需要编译，而 JavaScript 语言则是运行时由解释器直接执行。

总之，JavaScript 的原始设计目标是一种小型的、简单的动态语言，与 Java 有足够的相似性，使得使用者（尤其是 Java 程序员）可以快速上手。

什么是JavaScript?

> JavaScript 是一门弱类型的动态脚本语言，支持多种编程范式，包括面向对象和函数式编程，被广泛用于 Web 开发。
>
> JavaScript是一门基于原型的动态解释性脚本语言

一般来说，前端领域完整的JavaScript包括以下几个部分：

- ECMAScript，描述了该语言的语法和基本对象
- 文档对象模型（DOM），描述处理网页内容的方法和接口
- 浏览器对象模型（BOM），描述与浏览器进行交互的方法和接口

它的基本特点如下：

- 是一种解释性脚本语言（代码不进行预编译）。
- 主要用来向HTML页面添加交互行为。
- 可以直接嵌入HTML页面，但写成单独的js文件有利于结构和行为的分离。

JavaScript常用来完成以下任务：

- 嵌入动态文本于HTML页面
- 对浏览器事件作出响应
- 读写HTML元素
- 在数据被提交到服务器之前验证数据
- 检测访客的浏览器信息

BOM - 浏览器对象模型

一套操作浏览器功能的API

通过BOM可以操作浏览器窗口，比如：弹出框、控制浏览器跳转、获取分辨率等

DOM - 文档对象模型

一套操作页面元素的API

DOM可以把HTML看做是文档树，通过DOM提供的API可以对树上的节点进行操作

JavaScript 与 Java 的关系

> 本质上讲 JavaScript和 Java没有关系，只是JavaScript诞生之初 网景与sun合作想要推出 web端的脚本语言。JavaScript 的基本语法和对象体系，是模仿 Java 而设计的。但是，JavaScript 没有采用 Java 的静态类型。除此之外JavaScript和Java在语言层面上可以说是大相径庭。处于某种商业意图，这门语言才从一开始的 LiveScript 改名为 JavaScript。

JavaScript与ECMAScript的关系

> ECMAScript 只用来标准化 JavaScript 这种语言的基本语法结构，与部署环境相关的标准都由其他标准规定，比如 DOM 的标准就是由 W3C组织（World Wide Web Consortium）制定的。

ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现。在日常场合，这两个词是可以互换的。

JavaScript能做什么?

> 任何能够用JavaScript实现的应用系统，最终都必将用JavaScript实现　---Atwood定律

前端领域

ajax出现之后 JavaScript有了在WEB领域立足的土壤和根基，时至今日。JavaScript已经是WEB前端领域最重要的基石，一切现代化的WEB项目都离不开JavaScript。

- 数据交互
- UI管理
- 用户行为交互
- 数据校验
- 工程化/模块化
- MVVM

后端领域

V8 JIT NodeJS 让JavaScript可以在服务端崭露头角 打破了JavaScript只能寄生在浏览器上的魔咒

CouchDB mongodb等基于JSON格式的NoSQL类型的数据库诞生 让JavaScript也可以在DB操作上大展身手

- WEB服务框架: express/KOA
- NoSQL 数据库: mongodb CouchDB
- 博客系统 : Ghost/hexo
- 自动化构建领域: Gulp/Grunt

APP

> 目前来说主流的App开发方式有三种：Native App 、Web App、Hybird App，而3种方式下又分别有众多的框架可供选择。JavaScript 可以介入开发的有下面三种：

- Web App HTML5 APP 框架开发模式
- Hybrid App 混合模式移动应用
- PWA Progressive Web App **渐进式网页应用**

桌面应用

> JavaScript还可以介入的桌面应用开发，主流有 electron Node-webkit hex React Navite

electron代表作:

- vscode
- atom

Node-webkit代表作:

- teambition

hex代表作

- 有道词典

图形/游戏

> 世界上最流行的 2D 游戏引擎之一 Cocos2d 和最流行的 3D 游戏引擎之一 Unity3D 均支持 JS 开发游戏。
>
> 以及 Cocos2d-js 轻量型跨平台Web游戏引擎

嵌入式与IOT开发

> JavaScript 不只是 Web 时代的通用语言，如今还延伸到了使人难以置信的其它地方: 物联网;
>
> JavaScript for Microcontrollers and IoT

为什么要学JavaScript?

> JavaScript 的上手方便 表达灵活 社区支持度高 应用广泛，是当今最受欢迎 应用最广泛的语言之一....
>
> 现实点说，在现代互联网环境下，你很难找到不需要JavaScript开发岗位的企业。

####

学习JavaScript所需要的的环境与设备

- 一台可以运行浏览器的电脑
- 浏览器
- 开发工具 (推荐 vscode)

学习JavaScript所需要做的心里准备

> 不论你之前是否学习接触过其他编程语言，学习JavaScript你需要做好以下几点心里准备

- 不要用常规认知去理解JavaScript世界的规则 3<2<1 0.2 + 0.7
- 一切以事实为准，一切认知建立在实践基础上
- 多练 多练 多练 多练 多练 多练 多练 多练.....

JavaScript基础

在HTML中嵌入JavaScript代码

```
<html>
  <head>
      <script src="js/index.js"></script>
  </head>
  <body>
    <input type="button" value="按钮" onclick="alert('Hello World')" />
    <script>
      var str = 'hello world!';
       console.log(str);
    </script>
  </body>
</html>
```

第一段JavaScript代码

```
var str = 'hello world!';
console.log(str);
```

JavaScript基础名词概念

```
var num = 1 + 1;
```

语句

> JavaScript语句为由上至下 单行执行; 每一行都是一条语句, 每一条语句执行完成之后会进入下一行 语句由于 ; 结尾

表达式

> 上述代码中 `1 + 1` 就是一个表达式(expression) 指一个为了得到返回值的计算式。

语句可以理解为一条命令，并不一定需要的到一个具体的期望值。表达式的目的是为了得到一个值，方便后面利用值去做什么事情，在上述例子中 表达式 `1+1` 得到的值 **赋值** 给了 **变量** num;

变量

> 上述例子中 num 为我们 **声明** 的 **变量** , 变量可以理解为一个容器，用于存放各种各样的**值**。由于需要存放各种不同的值 所以我们要为 变量**命名** `num` 就是 上述例子中 的变量名

变量声明

> 上述例子中 我们通过表达式 `1 + 1` 得到的值, **赋值**给了 我们通过 **var**关键字 **声明**的 **变量** `str` ; 后续 num 所指向的内存地址中存储的值就是 `1 + 1` 的结果。 在这个过程中我们其实做了两件事情:

1. 我们声明变量 是通过 **关键字var** 创建了变量 num
2. 我们将 表达式 `1 + 1` 的值 **赋值**给了 变量 num

```
// 声明变量 变量名为 num 没有赋值 默认值为 undefined undefined也是一个 JavaScript 关键字，表示“无定义”。
var num;
// 这一步 将 1 + 1 的结果赋值给 变量 num 让str所代表的的内存地址所存储的内容为 1 + 1 的值, 后续我们想要使用 1 + 1的结果 可以直接通过 调用变量 num 调用
num = 1 + 1;
```

变量赋值

> 我们可以通过 `var` 关键字 创建一个变量 , 创建出现的变量默认值为 `undefined` 未定义, 我们可以通过赋值 表达式 `=` 来给变量存储值 格式如下

```
变量名称 = 值;
```

变量引用

> 我们学会了如何创建变量 以及如何给变量赋值, 那赋值后的变量我们如何使用呢?

```
//使用console.log方法 在控制台中 打印 num变量的值
console.log(num);
//将变量 num 带入表达式 num + 1 让表达式成为为 num变量的内容 + 1 并且赋值给 变量 count
var count = num + 1;
```

标识符 (变量名称)

> 标识符（identifier）指的是用来识别各种值的合法名称。最常见的标识符就是变量名，以及后面要提到的函数名。JavaScript 语言的标识符对大小写敏感，所以`a`和`A`是两个不同的标识符。

命名规则

> 必须遵守的命名规则 如果不遵守 就会报错

```
- 由字母、数字、下划线、$符号组成，不能以数字以及其他符号开头
- 不能是保留关键字，例如：for、while。
```

保留关键字:

arguments、break、case、catch、class、const、continue、debugger、default、delete、do、else、enum、eval、export、extends、false、finally、for、function、if、implements、import、in、instanceof、interface、let、new、null、package、private、protected、public、return、static、super、switch、this、throw、true、try、typeof、var、void、while、with、yield。

*合法标识符 中文为合法标识符任意 Unicode 字母（包括英文字母和其他语言的字母）

```
num
$con
_target
π
计数器
```

*非法标识符

```
1x
321
***
-x
undefined
for
```

命名规范

> 遵守规范能够给自己和他人带来更好的开发体验，不遵守并不会导致报错

- 具有语义性的 英文单词；
- 多个单词使用驼峰命名法
- 变量名称 为名词 可以使用形容词为前缀

*良好的命名

```
maxCount
petName
str
num
```

*不好的命名

```
max-count
maxcount
getName
```

变量规则

未声明变量直接使用

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

上面代码直接使用变量`x`，系统就报错，告诉你变量`x`没有声明。

省略 var 关键字

```
a = 30;
console.log(a); //30
```

> 在javascript中 变量可以省略 var关键字 直接调用或者赋值，解释器会帮我们 隐式声明 变量

> 但是，不写`var`的做法，不利于表达意图，而且容易不知不觉地创建全局变量，所以建议总是使用`var`命令声明变量。

重复赋值

```
var x = 10;
x = 20;
console.log(x); //20

解释为

var x;
x = 10;
x = 20;
console.log(x);
```

> `x` 一开始声明并且赋值为`10` 后面如果想要修改`x`的值 不需要重新声明 直接再次赋值`20` 覆盖之前`x`的值内容即可

重复声明

```
var x = 1;
var x;
console.log(x); //1

解释为

var x;
x = 1;
console.log(x);
```

> 对同一个变量进行二次声明 第二次声明是无效的操作 因为同一个 环境中 变量名是唯一的;

重复声明赋值

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

解释为

var x;
x = 1;
x = 2;
console.log(x);
```

> 结合上一个重复声明, 当重复声明且赋值的时候, 第二行的声明无效 但 赋值操作有效 所以 变量 `x` 的值 由1 覆盖为 2

批量声明

```
var a,b,c,d = 10;

解释为

var a;
var b;
var c;
var d;
d = 10;
```

> 在上面的代码中 我们可以通过 `,` 隔开多个变量, 通过一个 `var` 关键字进行批量声明 , 最后一个 变量 `d` 赋值为10;

变量提升

```
console.log(num); //undefined
var num = 10;
```

> 上面的这个代码中 我们书写语句的顺序是

1. 调用了 num 进行打印
2. 声明了变量 num 并且 赋值 10

> 实际在javascript引擎解释后 顺序为

```
var num;
console.log(num); //undefined
num = 10;
```

1. 先声明 num 这一步称为 **变量提升**
2. 调用console.log() 打印 num的值 这时因为没有给num赋值 num的值还是 初始默认值 undefined
3. 给num 赋值 为 10

注释

> 在javascript也会频繁用到注释 注释形式有 两种

```
// 这是单行注释 ctrl+/

/*
 这是
 多行
 注释
 ctrl+shift+/
*/
```

数据类型

> JavaScript 语言的每一个值，都属于某一种数据类型。JavaScript 的数据类型，共有七种。

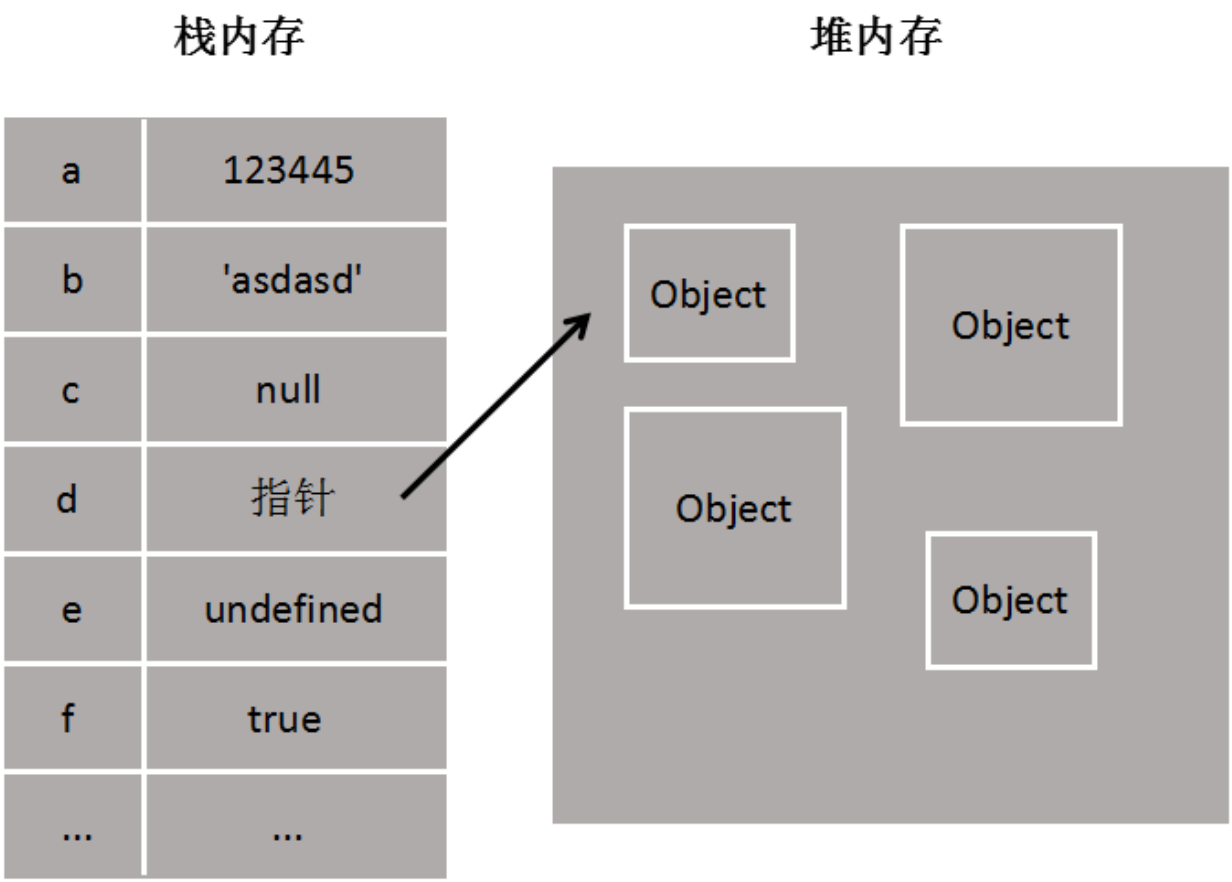
- 数值（number）：整数和小数（比如`1`和`3.14`）
- 字符串（string）：文本（比如`'Hello World'`）。
- 布尔值（boolean）：表示真伪的两个特殊值，即`true`（真）和`false`（假）
- `undefined`：表示"未定义"或不存在，即由于目前没有定义，所以此处暂时没有任何值
- `null`：表示空值，即此处的值为空。
- 对象（object）：各种值组成的集合。
- symbol (symbol) : 唯一标识符 //es6学习前不做讨论

通常，数值、字符串、布尔值这三种类型，合称为原始类型（primitive type）的值，即它们是最基本的数据类型，不能再细分了。对象则称为合成类型（complex type）的值，因为一个对象往往是多个原始类型的值的合成，可以看作是一个存放各种值的容器。至于undefined和null，一般将它们看成两个特殊值。

类型分类

> 传统分类通过存储位置 把数据类型分为 基础类型 和 引用类型

基础类型 存储在 栈内存 中
Undefined、Null、Boolean、Number、String和symbol



Number 类型

- 数值字面量：数值的固定值的表示法

  110 1024 60.5

- 进制

十进制
 var num = 9;
 进行算数计算时，八进制和十六进制表示的数值最终都将被转换成十进制数值。
十六进制
 var num = 0xA;
 数字序列范围：0~9以及A~F
八进制
  var num1 = 07; // 对应十进制的7
  var num2 = 019; // 对应十进制的19
  var num3 = 08; // 对应十进制的8
  数字序列范围：0~7
  如果字面值中的数值超出了范围，那么前导零将被忽略，后面的数值将被当作十进制数值解析

- 浮点数

- 浮点数的精度问题

```
浮点数
  var n = 5e-324;  // 科学计数法  5乘以10的-324次方
浮点数值的最高精度是 17 位小数，但在进行算术计算时其精确度远远不如整数
   var result = 0.1 + 0.2;   // 结果不是 0.3，而是：0.30000000000000004
   console.log(0.07 * 100);
   不要判断两个浮点数是否相等
```

- 数值范围

```
最小值：Number.MIN_VALUE，这个值为： 5e-324
最大值：Number.MAX_VALUE，这个值为： 1.7976931348623157e+308
无穷大：Infinity
无穷小：-Infinity
```

- 数值判断
  - NaN：not a number 表示“非数字”（Not a Number），主要出现在将字符串解析成数字出错的场合。
    - NaN 与任何值都不相等，包括他本身
  - isNaN: is not a number
  - NaN进行任何数学运算 结果也是 NaN

数值精度问题

根据国际标准 IEEE 754，JavaScript 浮点数的64个二进制位，从最左边开始，是这样组成的。

- 第1位：符号位，`0`表示正数，`1`表示负数
- 第2位到第12位（共11位）：指数部分
- 第13位到第64位（共52位）：小数部分（即有效数字）

符号位决定了一个数的正负，指数部分决定了数值的大小，小数部分决定了数值的精度。

指数部分一共有11个二进制位，因此大小范围就是0到2047。IEEE 754 规定，如果指数部分的值在0到2047之间（不含两个端点），那么有效数字的第一位默认总是1，不保存在64位浮点数之中。也就是说，有效数字这时总是`1.xx...xx`的形式，其中`xx..xx`的部分保存在64位浮点数之中，最长可能为52位。因此，JavaScript 提供的有效数字最长为53个二进制位。

```
(-1)^符号位 * 1.xx...xx * 2^指数部分
```

上面公式是正常情况下（指数部分在0到2047之间），一个数在 JavaScript 内部实际的表示形式。

精度最多只能到53个二进制位，这意味着，绝对值小于2的53次方的整数，即-2^53到2^53，都可以精确表示。

数值范围

根据标准，64位浮点数的指数部分的长度是11个二进制位，意味着指数部分的最大值是2047（2的11次方减1）。也就是说，64位浮点数的指数部分的值最大为2047，分出一半表示负数，则 JavaScript 能够表示的数值范围为21024到2-1023（开区间），超出这个范围的数无法表示。

如果一个数大于等于2的1024次方，那么就会发生“正向溢出”，即 JavaScript 无法表示这么大的数，这时就会返回`Infinity`。

String类型

'joker' "kyogre"

- 字符串字面量

  ‘海牙老师 真的帅’

- 转义符

字 面 量	含　义
\n	换行
\t	制表
\b	空格
\r	回车
\f	进纸
\\	斜杠
\'	单引号（'），在用单引号表示的字符串中使用。例如：'He said, \'hey.\''
\"	双引号（"），在用双引号表示的字符串中使用。例如："He said, \"hey.\""
\xnn	以十六进制代码*nn*表示的一个字符（其中*n*为0～F）。例如，\x41表示"A"
\unnnn	以十六进制代码*nnnn*表示的一个Unicode字符（其中*n*为0～F）。例如，\u03a3表示希腊字符Σ

- 字符串长度 (只读)

  length属性用来获取字符串的长度

```
var str = '海牙 Hello World';
console.log(str.length);
```

- 字符串拼接

  字符串拼接使用 + 连接

```
console.log(11 + 11);
console.log('hello' + ' world');
console.log('100' + '100');
console.log('11' + 32);
console.log('male:' + true);
```

1. 两边只要有一个是字符串，那么+就是字符串拼接功能
2. 两边如果都是数字，那么就是算术功能。

- 字符串换行

```
var longString = '第一行 '
  + '第二行 '
  + '第三行 '
  + '文本内容';


var longString = '第一行 \
第二行 \
第三行 \
文本内容';

longString
```

- 按位取值(只读)

```
var str = 'hello world!';
console.log(str[1]); //e
str[1] = 'x'; //无法改写
```

Boolean类型

> 布尔值代表“真”和“假”两个状态。“真”用关键字`true`表示，“假”用关键字`false`表示。布尔值只有这两个值。

- Boolean字面量： true和false，区分大小写
- 计算机内部存储：true为1，false为0

Undefined和Null

> `null`与`undefined`都可以表示“没有”，含义非常相似。将一个变量赋值为`undefined`或`null`，老实说，语法效果几乎没区别。

1. undefined表示一个声明了没有赋值的变量，变量只声明的时候值默认是undefined
2. null表示一个空，变量的值如果想为null，必须手动设置

在javascript设计初期 `null`就像在 Java 里一样，被当成一个对象。初像 Java 一样，只设置了`null`表示"无"。根据 C 语言的传统，`null`可以自动转为`0`。 但是javascript并没有完整的ERROR机制 `null`可以转换为0 对于javascript这种弱类型的语言来说 不利于发现bug 所以设计了 `undefined` 变量默认值也就成为了 `undefined`

所以在学习javascript的过程中 不能用java等编程语言的`null`来理解javascript中的`null`

在javascript中 null是 对象的延伸 是一个 '空' 对象。`var str = '';` 中 `''` 不能用null判断 他是有值的 内存中分配了空间来存储 `''` 一个空字符串值。`null`在javascript中常见于释放内存空间 `var str = null;`

类型判断

JavaScript 有三种方法，可以确定一个值到底是什么类型。

- `typeof`运算符
- `instanceof`运算符
- `Object.prototype.toString`方法

typeof

> 数值、字符串、布尔值分别返回`number`、`string`、`boolean`，undefined`返回`undefined null`返回`object

```
typeof 123 // "number"
typeof '123' // "string"
typeof false // "boolean"
typeof undefined //"undefined"
typeof null // "object"
```

typeof 针对未声明的变量

```
if (typeof v === "undefined" ) {
  console.log("变量 v 不存在")
}
```

isNaN

`isNaN`方法可以用来判断一个值是否为`NaN`。

```
isNaN(NaN) // true
isNaN(123) // false
```

但是，`isNaN`只对数值有效，如果传入其他值，会被先转成数值。比如，传入字符串的时候，字符串会被先转成`NaN`，所以最后返回`true`，这一点要特别引起注意。也就是说，`isNaN`为`true`的值，有可能不是`NaN`，而是一个字符串。

```
isNaN('Hello') // true
// 相当于
isNaN(Number('Hello')) // true
```

出于同样的原因，对于对象和数组，`isNaN`也返回`true`。

```
isNaN({}) // true
// 等同于
isNaN(Number({})) // true

isNaN(['xzy']) // true
// 等同于
isNaN(Number(['xzy'])) // true
```

但是，对于空数组和只有一个数值成员的数组，`isNaN`返回`false`。

```
isNaN([]) // false
isNaN([123]) // false
isNaN(['123']) // false
```

上面代码之所以返回`false`，原因是这些数组能被`Number`函数转成数值

因此，使用`isNaN`之前，最好判断一下数据类型。

```
function myIsNaN(value) {
  return typeof value === 'number' && isNaN(value);
}
```

判断`NaN`更可靠的方法是，利用`NaN`为唯一不等于自身的值的这个特点，进行判断。

```
function myIsNaN(value) {
  return value !== value;
}
```

isFinite

`isFinite`方法返回一个布尔值，表示某个值是否为正常的数值。

```
isFinite(Infinity) // false
isFinite(-Infinity) // false
isFinite(NaN) // false
isFinite(undefined) // false
isFinite(null) // true
isFinite(-1) // true
```

除了`Infinity`、`-Infinity`、`NaN`和`undefined`这几个值会返回`false`，`isFinite`对于其他的数值都会返回`true`。

操作符

运算符 operator

5 + 6

表达式 组成 操作数和操作符，会有一个结果

算术运算符

- **加法运算符**：`x + y`
- **减法运算符**：`x - y`
- **乘法运算符**：`x * y`
- **除法运算符**：`x / y`
- **指数运算符**：`x ** y`
- **余数运算符**：`x % y`
- **自增运算符**：`++x` 或者 `x++`
- **自减运算符**：`--x` 或者 `x--`
- **数值运算符**：`+x`
- **负数值运算符**：`-x`

一元运算符

一元运算符：只有一个操作数的运算符

1 + 2 两个操作数的运算符 二元运算符

++ 自身加1

-- 自身减1

- 前置++

```
var num1 = 5;
++ num1;

var num2 = 6;
console.log(num1 + ++ num2);
```

- 后置++

```
var num1 = 5;
num1 ++;
var num2 = 6
console.log(num1 + num2 ++);
```

- 猜猜看

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

总结
前置++：先加1，后参与运算
后置++：先参与运算，后加1
上面两个理解后，下面两个自通
前置-- ：先减1，后参与运算

后置-- ： 先参与运算，后减1

逻辑运算符(布尔运算符)

```
&& 与 两个操作数同时为true，结果为true，否则都是false
|| 或 两个操作数有一个为true，结果为true，否则为false
! 非 取反
```

关系运算符(比较运算符)

```
< > >= <= == != === !==
```

```
==与===的区别：==只进行值得比较，===类型和值同时相等，则相等

var result = '55' == 55;   // true
var result = '55' === 55;  // false 值相等，类型不相等
var result = 55 === 55;  // true
```

赋值运算符

```
=  +=  -=  *=  /=  %=
```

```
例如：
var num = 0;
num += 5; //相当于 num = num + 5;
```

二进制运算符 (了解)

> javascript 支持二进制运算 ~ | & >> << >>> ^

```
异或运算 ^
异或运算（^）在两个二进制位不同时返回1，相同时返回0。

一般可以用来做开关或者倒值
var a = 33;
var b = 66;

a ^= b, b ^= a, a ^= b;

a // 66
b // 33
```

```
否运算符 ~

一般用于 双否 取整
~~ 13.33    // 13
```

##

运算符的优先级

```
优先级从高到底
    1. () 优先级最高
    2. 一元运算符 ++  --  !
    3. 算数运算符 先* / %  后 + -
    4. 关系运算符 >  >=  <  <=
    5. 相等运算符  ==  !=  ===  !==
    6. 逻辑运算符 先&&  后||
    7. 赋值运算符
    8. 默认从左至右 除了 赋值运算 = 三目运算 ?: 指数运算 **
```

```
// 练习1:
4 >= 6 || '山海' != '琨' && !(12 * 2 == 144) && true
// 练习2:
var num = 10;
5 == num / 2 && (2 + 2 * num).toString() === '22'
```

注: Unicode 码表查询地址 https://www.ltool.net/characters-to-unicode-charts-in-simplified-chinese.php

数据类型转换

chrome浏览器中 不同类型的值 打印颜色不同

字符串的颜色是黑色的，数值类型是蓝色的，布尔类型也是蓝色的，undefined和null是灰色的

转换成字符串类型

- toString()

  ```
  var num = 5;
  console.log(num.toString());
  ```

- String()

  ```
  String()函数存在的意义：有些值没有toString()，这个时候可以使用String()。比如：undefined和null
  ```

- 拼接字符串方式

num + ""，当 + 两边一个操作符是字符串类型，一个操作符是其它类型的时候，会先把其它类型转换成字符串再进行字符串拼接，返回字符串

转换成数值类型

- Number(Obj)

  Number()可以把任意值转换成数值，如果要转换的字符串中有一个不是数值的字符，返回NaN

- parseInt(string,radix)

```
var num1 = parseInt("12.3abc"); // 返回12，如果第一个字符是数字会解析知道遇到非数字结束
var num2 = parseInt("abc123");  // 返回NaN，如果第一个字符不是数字或者符号就返回NaN
```

- parseFloat(string)

```
parseFloat()把字符串转换成浮点数
parseFloat()和parseInt非常相似，不同之处在与
    parseFloat会解析第一个. 遇到第二个.或者非数字结束
    如果解析的内容里只有整数，解析成整数
```

- +，-0等运算

```
var str = '500';
console.log(+str);   // 取正
console.log(-str);   // 取负
console.log(str - 0);
```

转换成布尔类型

- Boolean()
- !!

  ''(空字符串) null undefined NaN 0 会转换成false 其它都会转换成true

隐式转换

递增递减运算符(前置、后置)

1. 如果包含的是有效数字字符串或者是有效浮点数字符串，则会将字符串转换(Number())为数值，再进行加减操作，返回值的类型是：number类型。
2. 如果不包含有效数字字符串，则会将字符串的值转换为NaN,返回值的类型是：number类型。

3. 如果是boolean类型，则先会把true或者false转换为1或者0，再进行加减操作，返回值的类型是：number类型。
4. 如果是null类型，则先会把null转换为0，在进行加减操作，返回值的类型是：number类型。
5. 如果是undefined，则先会把undefined转换为NaN，再进行加减操作，返回值的类型是：number类型。
6. 如果是对象，则先会通过对象的valueOf()方法，进行转换，如果返回的是NaN，调用toString()方法，在进行前面的操作，返回值的类型是：number类型。（注：空数组[]会返回0，在进行加减操作，空对象则会返回NaN）。

逻辑操作符中的隐式转换规律

注：只有undefined、null、NaN、0、空字符串会被转换为false，其余都为true

逻辑操作符一般用于语句判断中。通过判断结果返回的值进行后面的语句操作。

1. 逻辑非(!)操作符：首先会通过Boolean()函数将其操作值转换为布尔值，然后求反。
2. 逻辑与(&&)操作符：如果第一个值经过Boolean()函数转换后为true，则返回第二个操作值，否则返回第一个操作值。如果有一个操作值为null这返回null，如果有一个操作值为undefined，则返回undefined，如果有一个值为NaN，则返回NaN。
3. 逻辑或(||)操作符：如果第一个值经过Boolean()函数转换为false，则返回第二个操作值，否则返回第一个操作值。
   （注：逻辑操作符的运算为短路逻辑运算：前一个条件已经能够得出结果后续条件不再执行！）

关系操作符的隐式转换规律

(关系操作符的操作值也可以是任意类型)：

1. 如果两个操作值都是数值，则直接比较大小。
2. 如果两个操作值都是字符串，则字符串进行其Unicode编码进行比较。
3. 如果一个操作值是数值，则另一个值转换为数值进行比较。
4. 如果一个操作值是对象，则调用对象的valueOf()和toString()方法，然后再进行上述比较。
5. 如果一个操作值是布尔值，则将布尔值转换为数值再进行比较。
   (注：NaN和任何值都不相等，包括自己，同时它与任何类型比较都会返回false。)

相等操作符==和===的隐式转换规律：

1. 布尔值、字符串和数值进行比较，会先将其转换为数值再进行比较。
2. null和undefined比较是相等的，但不是全等的。
3. NaN与任何值都不相等，都会返回false。

布尔类型的隐式转换

流程控制语句会把后面的值隐式转换成布尔类型

```
转换为true  非空字符串  非0数字  true 任何对象
转换成false  空字符串  0  false  null  undefined NaN
```

```
// 结果是什么?
var a = !!'123';
```

转换不同的数据类型时，相等和不相等操作符遵循下列基本规则：

- 如果有一个操作数是布尔值，则在比较相等性之前先将其转换为数值——false 转换为 0，而true 转换为 1；
- 如果一个操作数是字符串，另一个操作数是数值，在比较相等性之前先将字符串转换为数值；
- 如果一个操作数是对象，另一个操作数不是，则调用对象的 valueOf()方法，用得到的基本类型值按照前面的规则进行比较；

这两个操作符在进行比较时则要遵循下列规则。

- null 和 undefined 是相等的。
- 要比较相等性之前，不能将 null 和 undefined 转换成其他任何值。
- 如果有一个操作数是 NaN，则相等操作符返回 false，而不相等操作符返回 true。 **重要提示：即使两个操作数都是 NaN，相等操作符也返回 false；因为按照规则， NaN 不等于 NaN。**
- 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，则相等操作符返回 true；否则，返回 false。

表达式	值
null==undefined	true
“NaN”==NaN	false
5==NaN	false
NaN==NaN	false
false==0	true
true==1	true
true==2	false
undefined==0	false
null==0	false

null == undefined 会返回 true，因为它们是类似的值；但 null === undefined 会返回 false，因为它们是不同类型的值。

=== 转换

值	字符串操作环境	数字运算环境	逻辑运算环境	对象操作环境
undefined	"undefined"	NaN	false	Error
null	"null"	0	false	Error
非空字符串	不转换	字符串对应的数字值	True	
空字符串	不转换	0	false	String
0	"0"	不转换	false	Number
NaN	"NaN"	不转换	false	Number
Infinity	"Infinity"	不转换	true	Number
Number.POSITIVE_INFINITY	"Infinity"	不转换	true	Number
Number.NEGATIVE_INFINITY	"-Infinity"	不转换	true	Number
Number.MAX_VALUE	"1.7976931348623157e+308"	不转换	true	Number
Number.MIN_VALUE	"5e-324"	不转换	true	Number
其他所有数字	"数字的字符串值"	不转换	true	Number
true	"true"	1	不转换	Boolean
false	"false"	0	不转换	Boolean
对象	toString()	value()或toString()或NaN	true	不转换

测试:

```
1 + true;
1 + 'true';
1 + undefined;
1 + null;
NaN == NaN;
undefined == null;
null !== undefined;
2 + '5' - 3;
6 > '3' == 3;
undefined == '0';
null == 0;
null >= 0;
parseInt('13.33') === ~~'13.33';
false - 1 <= '0';
'Value is ' + (val != '0') ? 'define' : 'undefine';
```

关于 null 在关系运算和相等运算中的坑:

```
null > 0   // null 尝试转型为number , 则为0 . 所以结果为 false,
null >= 0  // null 尝试转为number ,则为0 , 结果为 true.
null == 0  // null在设计上，在此处不尝试转型. 所以 结果为false.
```

参考 : http://bclary.com/log/2004/11/07/#a-11.9.3

1. 关系运算符 和 相等运算符 并不是一个类别的.
2. 关系运算符,在设计上,总是需要运算元尝试转为一个number . 而相等运算符在设计上,则没有这方面的考虑.
3. 最重要的一点, 不要把 拿 a > b , a == b 的结果 想当然的去和 a >= b 建立联系. 正确的符合最初设计思想的关系是 a > b 与 a >= b是一组 . a == b 和其他相等运算符才是一组. 比如 a === b , a != b, a !== b .

控制流程

> 编程的三种基本结构

顺序结构

从上到下执行的代码就是顺序结构

程序默认就是由上到下顺序执行的

分支结构

根据不同的情况，执行对应代码

循环结构

循环结构：重复做一件事情

分支结构

f语句

语法结构

```
if (/* 条件表达式 */) {
  // 执行语句
}

if (/* 条件表达式 */){
  // 成立执行语句
} else {

  // 否则执行语句
```

```
}

if (/* 条件1 */){
  // 成立执行语句
} else if (/* 条件2 */){
  // 成立执行语句
} else if (/* 条件3 */){
  // 成立执行语句
} else {
  // 最后默认执行语句
}
```

案例：
求两个数的最大数
判断一个数是偶数还是奇数
分数转换，把百分制转换成ABCDE <60 E 60-70 D 70-80 C 80-90 B 90 - 100 A

判断四季 3-5 春 6-8夏 9-11 秋 12-2 冬

作业：
判断一个年份是闰年还是平年
判断一个人的年龄是否满18岁(是否成年)

挑战:

闰年：能被4整除，但不能被100整除的年份 或者 能被400整除的年份

三元运算符

```
表达式1 ? 表达式2 : 表达式3
是对if.....else语句的一种简化写法
```

案例：
是否年满18岁
从两个数中找最大值

```
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

switch语句

语法格式:

```
switch (expression) {
  case 常量1:
    语句;
    break;
  case 常量2:
    语句;
    break;
```

```
  case 常量3:
    语句;
    break;
  ...
  case 常量n:
    语句;
    break;
  default:
    语句;
    break;
}
```

break可以省略，如果省略，代码会继续执行下一个case
switch 语句在比较值时使用的是全等操作符, 因此不会发生类型转换（例如，字符串'10' 不等于数值 10）

作业：switch 打印星期几 0为星期日 1-6分别对应星期一到星期六

交互与写入：alert、prompt 和 confirm / write

由于我们将使用浏览器作为我们的演示环境，让我们看几个与用户交互的函数：`alert`，`prompt` 和 `confirm`。

alert

这个我们前面已经看到过了。它会显示一条信息，并等待用户按下 “OK”。

例如：

```
alert("Hello");
```

弹出的这个带有信息的小窗口被称为 **模态窗**。“modal” 意味着用户不能与页面的其他部分（例如点击其他按钮等）进行交互，直到他们处理完窗口。在上面示例这种情况下 —— 直到用户点击“确定”按钮。

prompt

pormpt有文本消息的模态窗口，还有 input 框和确定/取消按钮。

```
result = prompt(title, [default]);
```

浏览器会显示一个带有文本消息的模态窗口，还有 input 框和确定/取消按钮。

- `title`

  显示给用户的文本

- `default`

  可选的第二个参数，指定 input 框的初始值。

语法中的方括号 `[...]`

上述语法中 `default` 周围的方括号表示该参数是可选的，不是必需的。

访问者可以在提示输入栏中输入一些内容，然后按“确定”键。然后我们在 `result` 中获取该文本。或者他们可以按取消键或按 Esc 键取消输入，然后我们得到 `null` 作为 `result`。

`prompt` 将返回用户在 `input` 框内输入的文本，如果用户取消了输入，则返回 `null`。

举个例子：

```
var age = prompt('How old are you?', 100);

alert("You are" + age +"ars old!");
```

IE 浏览器会提供默认值

第二个参数是可选的。但是如果我们不提供的话，Internet Explorer 会把 `"undefined"` 插入到 prompt。

我们可以在 Internet Explorer 中运行下面这行代码来看看效果：

```
var test = prompt("Test");
```

所以，为了 prompt 在 IE 中有好的效果，我们建议始终提供第二个参数：

```
var test = prompt("Test", ''); // <-- 用于 IE 浏览器
```

confirm

confirm取消两个按钮的模态窗口。

```
result = confirm(question);
```

`confirm` 函数显示一个带有 `question` 以及确定和取消两个按钮的模态窗口。

点击确定返回 `true`，点击取消返回 `false`。

例如：

```
var isBoss = confirm("Are you the boss?");

alert( isBoss ); // 如果“确定”按钮被按下，则显示 true
```

document.write

在JavaScript中document.write()函数可以向文档写入HTML表达式或JavaScript代码，用法“document.write(exp1,exp2,exp3,....)”，该函数可接受任何多个参数，并将其写入文档中。

```
document.write('我被写入了BODY中','还有我');
```

循环结构

> 在javascript中，循环语句有三种，while、do..while、for循环。
>
> 我们经常需要重复执行一些操作。
>
> 例如，我们需要将列表中的商品逐个输出，或者运行相同的代码将数字 1 到 10 逐个输出。
>
> **循环** 是一种重复运行同一代码的方法。

while语句

基本语法：

```
// 当循环条件为true时，执行循环体，
// 当循环条件为false时，结束循环。
while (循环条件) {
  //循环体
}
```

当 `condition`(条件) 为 `true` 时，执行循环体的代码。

例如，以下将循环输出当 `i < 3` 时的 `i` 值：

```
var i = 0;
while (i < 3) { // 依次显示 0、1 和 2
  console.log(i);
  i++;
}

//递减操作
var i = 10;
while (i) { // 10 9 8 7 6 5 4 3 2 1
   console.log(i);
   i--;
}
```

案例：
　打印100以内 7的倍数
　打印100以内所有偶数
　打印100以内所有偶数的和
作业：

打印100以内的奇数
打印100以内的奇数的和

do...while语句

> do..while循环和while循环非常像，二者经常可以相互替代，但是do..while的特点是不管条件成不成立，都会执行一次。

基础语法：

```
do {
  // 循环体;
} while (循环条件);
```

代码示例：

```
// 初始化变量
var i = 1;
var sum = 0;
do {
  sum += i;//循环体
  i++;//自增
} while (i <= 100);//循环条件
```

案例：

```
求100以内所有3的倍数的和
使用do-while循环：输出询问confirm “起床没?”，选择"确定" 打印'继续睡吧,我就是跟你说一下你要迟到了', 选择"取消" 继续输出询问confirm “起床没?”。
```

for语句

> while和do...while一般用来解决无法确认次数的循环。for循环一般在循环次数确定的时候比较方便

for循环语法：

```
// for循环的表达式之间用的是;号分隔的，千万不要写成,
for (初始化表达式1; 判断表达式2; 自增or自减表达式3) {
  // 循环体4
}
```

执行顺序：1243 ---- 243 -----243(直到循环条件变成false)

1. 初始化表达式
2. 判断表达式
3. 自增表达式
4. 循环体

案例：

打印1-100之间所有数
求1-100之间所有数的和
求1-100之间所有数的平均值
求1-100之间所有偶数的和
同时求1-100之间所有偶数和奇数的和
打印正方形
// 使用拼字符串的方法的原因
// console.log 输出重复内容的问题
// console.log 默认输出内容介绍后有换行

```
var start = '';
for (var i = 0; i < 10; i++) {
  for (var j = 0; j < 10; j++) {
    start += '* ';
  }
  start += '\n';
}
console.log(start);
```

打印直角三角形

```
var start = '';
for (var i = 0; i < 10; i++) {
  for (var j = i; j < 10; j++) {
    start += '* ';
  }
  start += '\n';
}
console.log(start);
```

打印9*9乘法表

```
var str = '';
for (var i = 1; i <= 9; i++) {
  for (var j = i; j <=9; j++) {
    str += i + ' * ' + j + ' = ' + i * j + '\t';
  }
  str += '\n';
}
console.log(str);
```

作业：

打印9 9 乘法表
求1-100之间所有数的乘积
求1-100之间所有奇数的和
计算1-100之间能3整除的数的和
计算1-100之间不能被7整除的数的和
//选做
// 讲解思路。如果不会写程序，可以先把数学公式准备好
本金10000元存入银行，年利率是千分之三，每过1年，将 本金和利息 相加作为新的本金。计算5年后，获得的本金是多少?
大马驮2石粮食，中马驮1石粮食，两头小马驮一石粮食，要用100匹马，驮100石粮食，该如何 调配
五个小朋友排成一队。问第一个多大了，第一个说比第二个大两岁，问第二个，第二个说比第 三个大两岁，以此类推。问第五个小朋友几岁了，第五个小朋友说3岁了。问第一个小朋友几岁 ？

continue和break

> break:立即跳出整个循环，即循环结束，开始执行循环后面的内容（直接跳到大括号）
>
> continue:立即跳出当前循环，继续下一次循环（跳到i++的地方）

案例：

```
求整数1～100的累加值，但要求碰到个位为3的数则停止累加
求整数1～100的累加值，但要求跳过所有个位为3的数
```

作业：

求1-100之间不能被7整除的整数的和（用continue）
求200-300之间所有的奇数的和（用continue）
求200-300之间第一个能被7整数的数（break）

调试

- 过去调试JavaScript的方式
  - alert()
  - console.log()
- 断点调试

> 断点调试是指自己在程序的某一行设置一个断点，调试时，程序运行到这一行就会停住，然后你可以一步一步往下调试，调试过程中可以看各个变量当前的值，出错的话，调试到出错的代码行即显示错误，停下。

- 调试步骤

```
浏览器中按F12-->sources-->找到需要调试的文件-->在程序的某一行设置断点
```

- 调试中的相关操作

```
Watch: 监视，通过watch可以监视变量的值的变化，非常的常用。
F10: 程序单步执行，让程序一行一行的执行，这个时候，观察watch中变量的值的变化。
F8：跳到下一个断点处，如果后面没有断点了，则程序执行结束。
```

tips: ***监视变量，不要监视表达式，因为监视了表达式，那么这个表达式也会执行。***

1. 代码调试的能力非常重要，只有学会了代码调试，才能学会自己解决bug的能力。初学者不要觉得调试代码麻烦就不去调试，知识点花点功夫肯定学的会，但是代码调试这个东西，自己不去练，永远都学不会。
2. 今天学的代码调试非常的简单，只要求同学们记住代码调试的这几个按钮的作用即可，后面还会学到很多的代码调试技巧。

数组

为什么要学习数组

> 之前学习的数据类型，只能存储一个值 比如：Number/String。我们想存储班级中所有学生的姓名，此时该如何存储?

数组的概念

> 所谓数组，就是将多个元素（通常是同一类型）按一定顺序排列放到一个集合中，那么这个集合我们就称之为数组。

数组的定义

> 数组是一个有序的列表，可以在数组中存放任意的数据，并且数组的长度可以动态的调整。

创建方式

通过实例化创建

```
var arr = new Array(3); //只写一个参数,创建一个长度为3的数组 , 数组中每一项都为 空置empty [empty,empty,empty]
arr = new Array(1,2,3,4); //多个参数  会生成对应参数个数与内容的数组 [1,2,3,4]
```

通过数组字面量创建数组

```
// 创建一个空数组
var arr1 = [];
// 创建一个包含3个数值的数组，多个数组项以逗号隔开
var arr2 = [1, 3, 4];
// 创建一个包含2个字符串的数组
var arr3 = ['a', 'c'];
```

数组的属性

```
var arr = [1,2,3,4];
// 可以通过数组的length属性获取数组的长度
console.log(arr.length);
// 可以设置length属性改变数组中元素的个数
arr.length = 0;
```

获取数组元素

数组的取值

```
// 格式：数组名[下标] 下标又称索引
// 功能：获取数组对应下标的那个值，如果下标不存在，则返回undefined。
var arr = ['red', 'green', 'blue'];
arr[0]; // red
arr[2]; // blue
arr[3]; // 这个数组的最大下标为2,因此返回undefined
```

遍历数组

> 遍历：遍及所有，对数组的每一个元素都访问一次就叫遍历。

数组遍历的基本语法：

```
for(var i = 0; i < arr.length; i++) {
  // 数组遍历的固定结构
}
```

for in 遍历(不推荐)

```
for(var key in arr){
    console.log(key,arr[key]);  //key 为下标 arr[key]为对应key下标的值
}
```

使用for-in可以遍历数组，但是会存在以下问题：

1.index索引为字符串型数字（注意，非数字），不能直接进行几何运算。

2.遍历顺序有可能不是按照实际数组的内部顺序（可能按照随机顺序）。

3.使用for-in会遍历数组所有的可枚举属性，包括原型。例如上例的原型方法method和name属性都会被遍历出来，通常需要配合hasOwnProperty()方法判断某个属性是否该对象的实例属性，来将原型对象从循环中剔除。

for of 遍历

```
for(var key of arr){
    console.log(key);
}
```

相比 for-in 不会出现顺序错乱的问题 也不会遍历出所有可枚举属性

数组元素赋值

数组的赋值

```
var arr = ['red', 'green', 'blue'];

arr[2] = 'yellow'; //给下标为2的数组元素赋值 如果该元素本身有值会进行覆盖

arr[3] = '#368'; // 给下标为3的数组元素赋值 如果该元素不存在就新增

arr //["red", "green", "yellow", "#368"]

arr[5] = '#f60'; //如果跨位进行赋值 空位显示 empty (空置)

arr // ["red", "green", "yellow", "#368", empty, "#f60"]
```

多维数组

> 数组中包含数组的话称之为多维数组。 您可以通过将两组方括号链接在一起来访问数组内的另一个数组
>
> 使用最多的是二维数组

```
var arr = [[1,2,3],[4,5,6],[7,8,9]];
arr[2][1] //8
```

数组类型判定与隐式转换

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

typeof arr //'object'

Number(arr) //NaN

String(arr) // '1,2,3'

Bollean(arr) // true

[] == [] //false

arr + '海牙' //'1,2,3海牙'

arr / 2 // NaN

arr + [] // '1,2,3'

[] + [] //''

[2]  - 1 //1

[1,]  - 1 //0
```

```
[1,2]  - 1 // NaN

!![] // true
```

数组基础方法

> 数组的方法有很多，在我们学习完函数 作用域 对象之前我们先了解一些数组的基础方法

###

isArray(obj)

> 用于确定传递的值是否是一个 Array。

```
Array.isArray([1, 2, 3]);
// true
Array.isArray({foo: 123});
// false
Array.isArray("foobar");
// false
Array.isArray(undefined);
// false
```

参数

obj 需要检测的值

返回值

如果值是Array 则为true; 否则为false。

join(separator)

> 方法将一个数组的所有元素连接成一个字符串并返回这个字符串。如果数组只有一个项目，那么将返回该项目而不使用分隔符。

```
const elements = ['火', '空气', '水'];

console.log(elements.join());
// "火,空气,水"

console.log(elements.join(''));
// "火空气水"

console.log(elements.join('-'));
// "火-空气-水"
```

参数

separator 可选 指定分隔符号 该参数默认值为","

返回值

一个所有数组元素连接的字符串。如果 arr.length 为0，则返回空字符串。

注意

如果一个元素为 `undefined` 或 `null`，它会被转换为空字符串''。

push(element1,...,elementN)

> 方法将一个或多个元素添加到数组的末尾，并返回该数组的新长度。

```
var animals = ['猪', '狗', '牛'];

var count = animals.push('大象');
console.log(count);
// 4
console.log(animals);
// ['猪', '狗', '牛','大象']

animals.push('鸡', '鸭', '鹅');
console.log(animals);
//['猪', '狗', '牛', '大象', '鸡', '鸭', '鹅']
```

参数

elementN 可选多个参数
参数会添加到数组末尾

返回值

当调用该方法时，新的 length 属性值将被返回。

unshift(element1,...,elementN)

> 方法将一个或多个元素添加到数组的前端，并返回该数组的新长度。

```
var animals = ['猪', '狗', '牛'];

var count = animals.unshift('大象');
console.log(count);
// 4
console.log(animals);
// ['大象', '猪', '狗', '牛']

animals.unshift('鸡', '鸭', '鹅');
console.log(animals);
//[ '鸡', '鸭' , '鹅','大象','猪', '狗', '牛' ]
```

参数

elementN 可选多个参数
参数会添加到数组前端

返回值

当调用该方法时，新的 length 属性值将被返回。

pop()

> 方法从数组中删除最后一个元素，并返回该元素的值。此方法更改数组的长度。

```
var plants = ['西红柿', '黄瓜', '芹菜', '豆角', '土豆'];

console.log(plants.pop());
//  "土豆"

console.log(plants);
// ['西红柿', '黄瓜', '芹菜', '豆角']

plants.pop();

console.log(plants);
// ['西红柿', '黄瓜', '芹菜']
```

参数

无

返回值

从数组中删除的元素(当数组为空时返回undefined)。

shift()

> 方法从数组中删除**第一个**元素，并返回该元素的值。此方法更改数组的长度。

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

var firstElement = array.shift();

console.log(array);
//  Array [2, 3]

console.log(firstElement);
//  1
```

参数

无

返回值

从数组中删除的元素; 如果数组为空则返回undefined 。

slice()

> 切割方法返回一个新的数组对象，这一对象是一个由 `begin` 和 `end` 决定的原数组的**浅拷贝**（包括 `begin`(起始下标)，不包括`end`(结束下标)）。原始数组不会被改变。

```
var plants = ['西红柿', '黄瓜', '芹菜', '豆角', '土豆'];

console.log(plants.slice(2));
// ['芹菜', '豆角', '土豆']

console.log(plants.slice(2, 4));
// [ '芹菜', '豆角']

console.log(plants.slice(1, 5));
//  ['黄瓜', '芹菜', '豆角', '土豆']
```

参数

```
arr.slice([begin[, end]]) //begin 和end都是可选参数
如果不传参 默认切割整个数组
begin 可选
  提取起始处的索引（从 0 开始），从该索引开始提取原数组元素。
  如果该参数为负数，则表示从原数组中的倒数第几个元素开始提取，slice(-2) 表示提取原数组中的倒数第二 个元素到最后一个元素（包含最后一个元素）。
  如果省略 begin，则 slice 从索引 0 开始。
  如果 begin 大于原数组的长度，则会返回空数组。

end 可选
  提取终止处的索引（从 0 开始），在该索引处结束提取原数组元素。slice 会提取原数组中索引从 begin 到  end 的所有元素（包含 begin，但不包含 end）。
  slice(1,4) 会提取原数组中从第二个元素开始一直到第四个元素的所有元素 （索引为 1, 2, 3的元素）。
  如果该参数为负数， 则它表示在原数组中的倒数第几个元素结束抽取。 slice(-2,-1) 表示抽取了原数组中  的倒数第二个元素到最后一个元素（不包含最后一个元素，也就是只有倒数第二个元素）。
  如果 end 被省略，则 slice 会一直提取到原数组末尾。
  如果 end 大于数组的长度，slice 也会一直提取到原数组末尾。
```

返回值

一个含有被提取元素的新数组。

concat()

> 合并方法用于合并两个或多个数组。此方法不会更改现有数组，而是返回一个新数组。

```
var plants = ['西红柿', '黄瓜', '芹菜', '豆角', '土豆'];
var otherPlants = ['冬瓜', '韭菜']
var newPlants = plants.concat(otherPlants);

console.log(newPlants);
//['西红柿', '黄瓜', '芹菜', '豆角', '土豆', '冬瓜', '韭菜']
```

参数

```
var newArray = oldArray.concat(value1[, value2[, ...[, valueN]]]);
```

valueN可选
数组和/或值，将被合并到一个新的数组中。如果省略了所有 valueN 参数，则 concat 会返回调用此方法的现存数组的一个浅拷贝。

返回值

一个合并后的新数组。

indexOf()

> 方法返回在数组中可以找到一个给定元素的第一个索引，如果不存在，则返回-1。

```
var plants = ['西红柿', '黄瓜', '芹菜', '豆角', '土豆','黄瓜'];

console.log(plants.indexOf('黄瓜'));
//1

console.log(plants.indexOf('黄瓜',3));
//5

console.log(plants.indexOf('大象'));
//-1
```

参数

arr.indexOf(searchElement[, fromIndex])

searchElement
要查找的元素

fromIndex 可选
开始查找的位置。如果该索引值大于或等于数组长度，意味着不会在数组里查找，返回-1。如果参数中提供的索引值是一个负值，则将其作为数组末尾的一个抵消，即-1表示从最后一个元素开始查找，-2表示从倒数第二个元素开始查找 ，以此类推。 注意：如果参数中提供的索引值是一个负值，并不改变其查找顺序，查找顺序仍然是从前向后查询数组。如果抵消后的索引值仍小于0，则整个数组都将会被查询。其默认值为0.

返回值

首个被找到的元素在数组中的索引位置; 若没有找到则返回 -1

排序

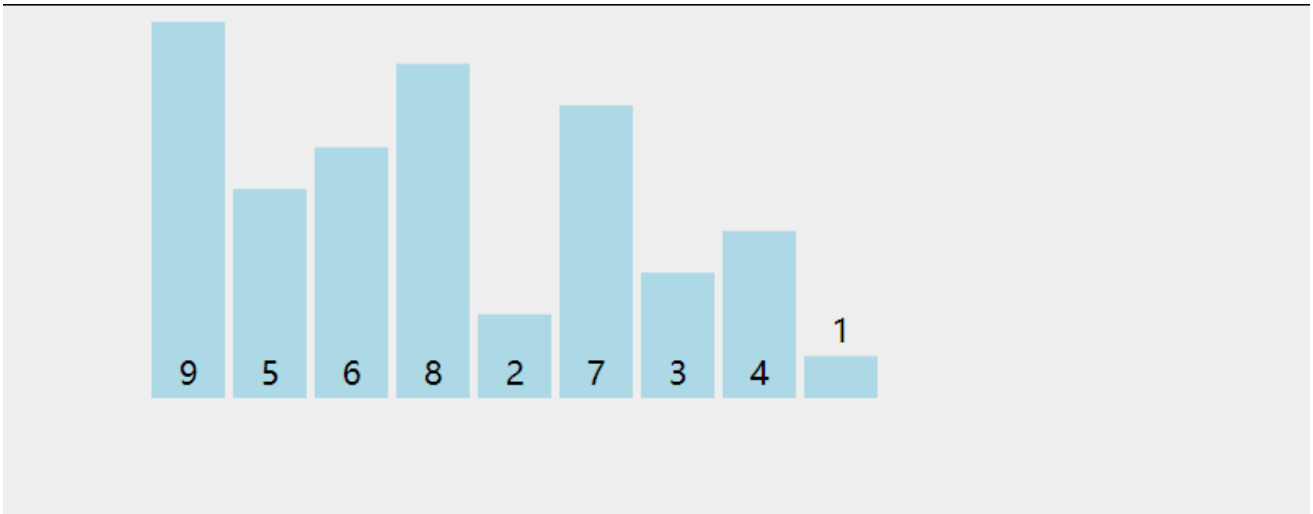
冒泡排序

> 冒泡排序是一种比较简单的排序算法，这种算法是让越小或者越大的元素经由交换慢慢“浮”到数列的顶端，就像水里面的泡泡向上浮动一样，所以叫“冒泡排序”。 冒泡排序算法的原理如下：

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2.对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

3. 针对所有的元素重复以上的步骤，除了最后一个.
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。



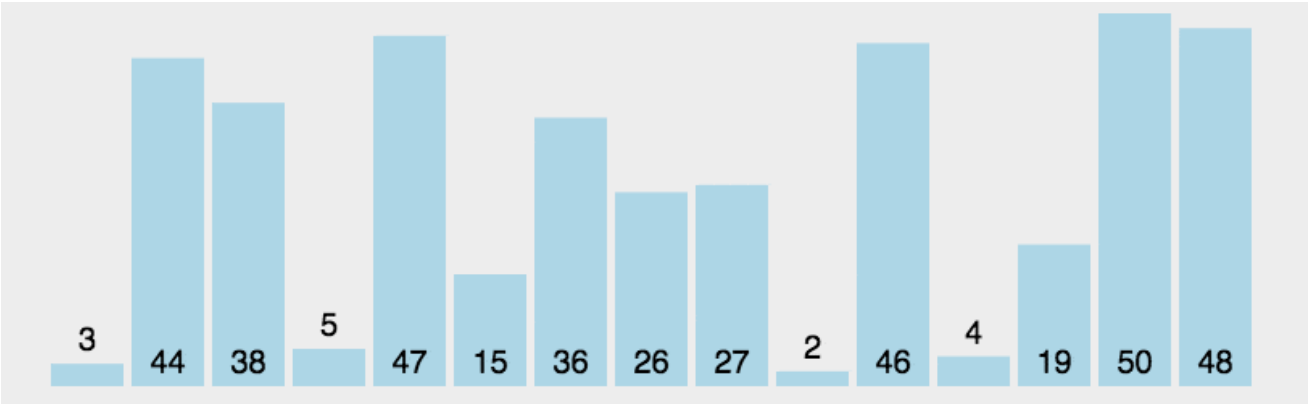
代码表现

```
var arr = [4,6,3,2,7,9,11,1];
```

```
var length = arr.length;
var temp;

for(var i = 0; i < length; i++) {
   for(var j = 0; j< length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
         temp = arr[j];
         arr[j] = arr[j + 1];
         arr[j + 1] = temp;
      }
   }
}
```

选择排序

> 选择排序是一种简单直观的排序算法，无论什么数据进去都是 O(n²) 的时间复杂度。所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

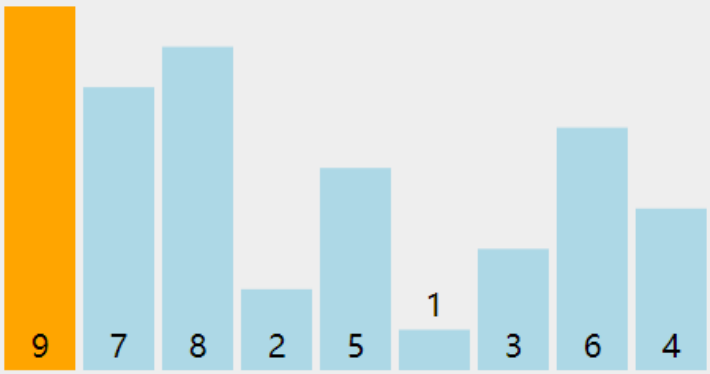


代码表现

```
var arr = [4,6,3,2,7,9,11,1];
var length = arr.length;
var minIndex, temp;
for (var i = 0; i < length - 1; i++) {
   minIndex = i;
   for (var j = i + 1; j < length; j++) {
     if (arr[j] < arr[minIndex]) {
        minIndex = j;
     }
   }
   temp = arr[i];
   arr[i] = arr[minIndex];
   arr[minIndex] = temp;
}
```

插入排序

> 插入排序的代码实现虽然没有冒泡排序和选择排序那么简单粗暴，但它的原理应该是最容易理解的了，因为只要打过扑克牌的人都应该能够秒懂。插入排序是一种最简单直观的排序算法，它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。



代码表现

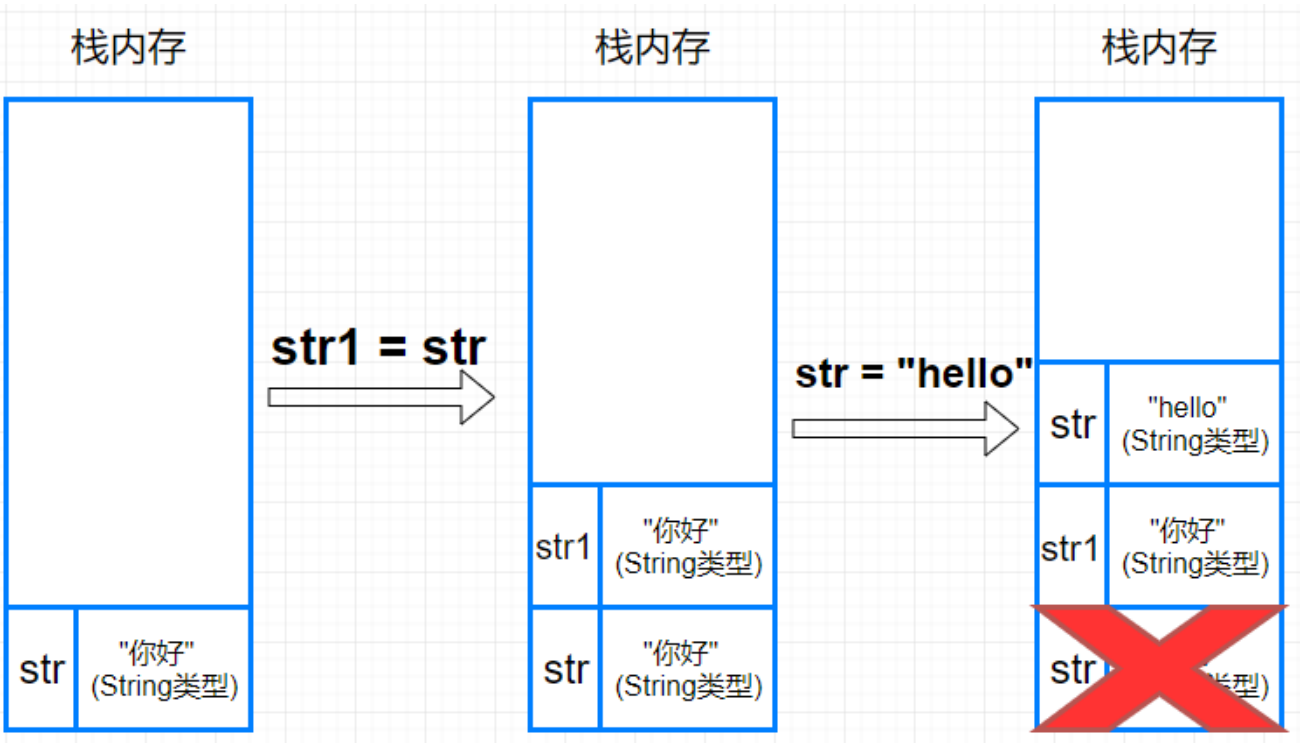
```
var arr = [4,6,3,2,7,9,11,1];
var length = arr.length;
for (var i = 1; i < length; i++) {
   var key = arr[i], j = i - 1;
   while (arr[j] > key) {
     arr[j + 1] = arr[j];
     j--;
   }
   arr[j + 1] = key;
 }
```

数组引用

> 数组属于对象object的一种，从数据类型上隶属于 **引用类型**

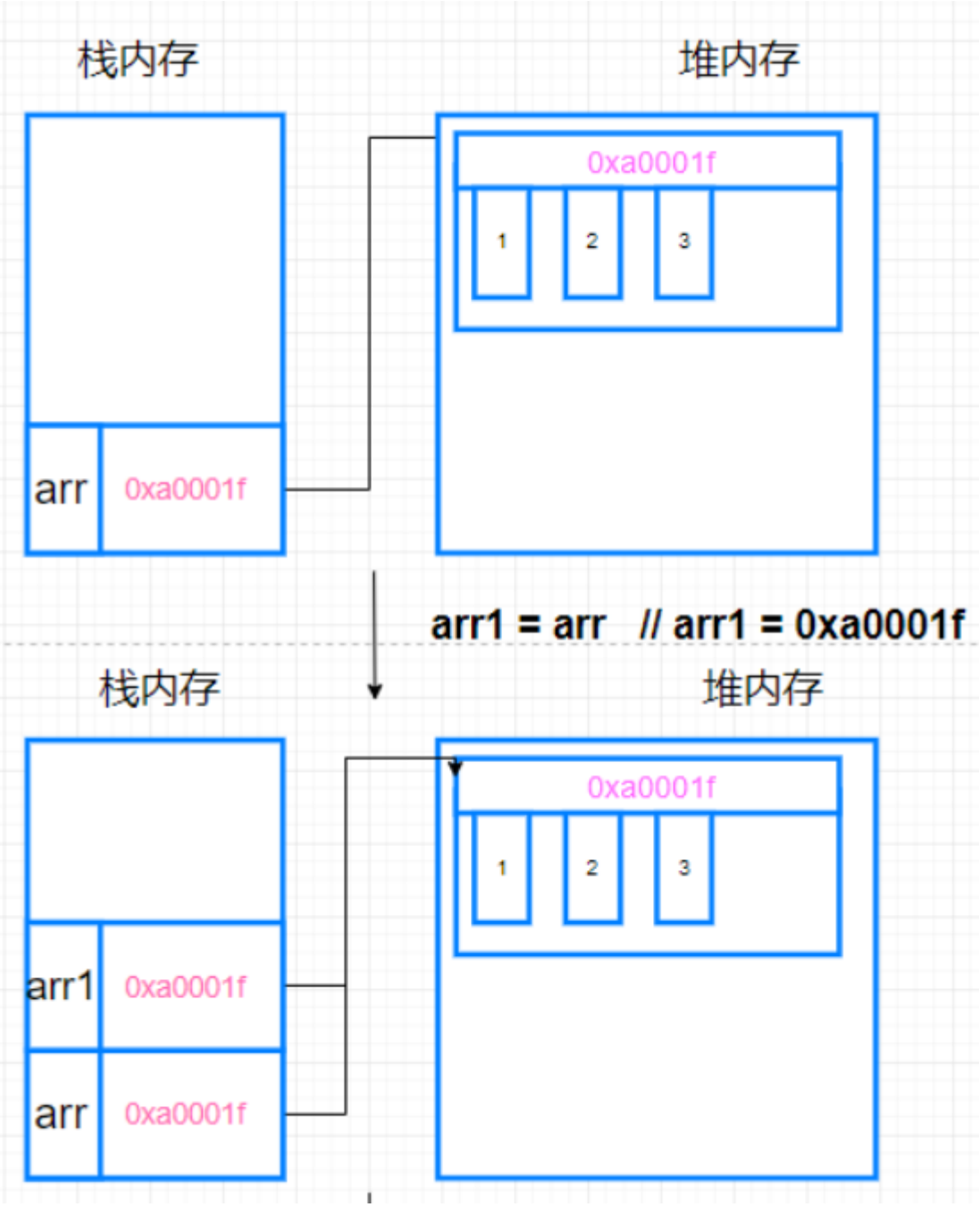
非引用关系的基础类型 在进行直接赋值的时候 会对赋值变量(右边)对应栈内存中空间所存储的值 赋值 给 被赋值变量(左边)

```
var str = '你好';
var str1 = str;
str = 'hello';
str // 'hello'
str1 // '你好'
```



引用类型进行直接变量赋值操作时 因为数组为引用类型 内容存储于堆内存中 栈内存只存储了数组的指针 所以数组变量直接作为赋值值时是将栈内存的指针赋值给了 接收变量

```
var arr = [1,2,3];
var arr1 = arr;
arr[2] = 5;
arr // [1,2,5]
arr1 //  [1,2,5]
```



为了切断引用关系 我们可以选择进行 按顺序给新数组push或者赋值 或者使用slice()进行浅拷贝 后续会学习深拷贝

```
var arr = [1,2,3];
var arr1;
for(var i = 0, len = arr.length; i < len; i++) {
 arr1[i] = arr[i];
 // arr1.push(arr[i]);
}

arr1 = arr.slice(); //浅拷贝
```

案例

求一组数中的所有数的和和平均值
求一组数中的最大值和最小值，以及所在位置
将字符串数组用|或其他符号分割
要求将数组中的0项去掉，将不为0的值存入一个新的数组，生成新的数组
查找数组[1,2,3,4,1,2,3,4,2,3,1,1,0]中所有值为1的项的下标 输出为新数组
翻转数组
冒泡排序，从小到大

作业

一、看数组 观察变化找规律
初始 ['a','b','c','d','e']
第一次 ['d','e','a','b','c']
第二次 ['b','c','d','e','a']
第三次 ['e','a','b','c','d']
...
请问第五次 第十五次数组是如何?

二、抽茧剥丝
var arr = ['鸡腿',101,'3','奥利奥',9,false,'33a',1.333,'巧克力'];
求数组中所有类型为number的整数之和

数组替换switch实现根据0-6打印星期一到星期日

字符串基础方法

charAt()

> 方法从一个字符串中返回指定的字符。

```
var str ='你好,我是海牙老师';
console.log(str.charAt(3));
//返回了 str[3]的值
```

参数

index
字符串中的字符从左向右索引，第一个字符的索引值为 0，最后一个字符（假设该字符位于字符串 stringName 中）的索引值为 stringName.length - 1。 如果指定的 index 值超出了该范围，则返回一个空字符串。

返回值

指定字符串

indexOf()

> 方法返回在字符串中可以找到一个给定字符的第一个首字对应索引，如果不存在，则返回-1。

```
var plants = '西红柿,黄瓜,芹菜,豆角,土豆,黄瓜'; //length 18

console.log(plants.indexOf('黄瓜'));
//4

console.log(plants.indexOf('黄瓜',5));
//16

console.log(plants.indexOf('黄'));
//4

console.log(plants.indexOf('黄瓶'));
//-1

console.log(plants.indexOf('大象'));
//-1

console.log(plants.indexOf(''));
//0


console.log(plants.indexOf('',5));
//5

console.log(plants.indexOf('',19));
//18 超出长度的参数会返回长度
```

参数

str.indexOf(searchValue[, fromIndex])

searchValue
要被查找的字符串值。
如果没有提供确切地提供字符串，searchValue 会被强制设置为 "undefined"， 然后在当前字符串中查找这个值。
举个例子：'undefined'.indexOf() 将会返回0，因为 undefined 在位置0处被找到，但是 'undefine'.indexOf() 将会返回 -1 ，因为字符串 'undefined' 未被找到。

fromIndex 可选
数字表示开始查找的位置。可以是任意整数，默认值为 0。
如果 fromIndex 的值小于 0，或者大于 str.length ，那么查找分别从 0 和str.length 开始。（译者注： fromIndex 的值小于 0，等同于为空情况； fromIndex 的值大于 str.length ，那么结果会直接返回 -1 。）
举个例子，'hello world'.indexOf('o', -5) 返回 4 ，因为它是从位置0处开始查找，然后 o 在位置4处被找到。另一方面，'hello world'.indexOf('o', 11) （或 fromIndex 填入任何大于11的值）将会返回 -1 ，因为开始查找的位置11处，已经是这个字符串的结尾了。

返回值

查找的字符串 searchValue 的第一次出现的索引，如果没有找到，则返回 -1。

split()

> 方法使用指定的分隔符字符串将一个String对象分割成子字符串数组，以一个指定的分割字串来决定每个拆分的位置。

```
var str ='你好,我是海牙老师';
var arr = str.split(',');

console.log(arr);
// ['你好','我是海牙老师']

var str ='你好,我是海牙老师';
var arr = str.split('');

console.log(arr);
// ["你", "好", ",", "我", "是", "海", "牙", "老", "师"]

var str ='你好,我是海牙老师';
var arr = str.split();

console.log(arr);
//["你好,我是海牙老师"]



var str ='你好,我,是海,牙老,师';
var arr = str.split(',',3);
```

```
console.log(arr);
// ["你好", "我", "是海"]
```

参数

str.split([separator[, limit]])
separator
指定表示每个拆分应发生的点的字符串。separator 可以是一个字符串或正则表达式。 如果纯文本分隔符包含多个字符，则必须找到整个字符串来表示分割点。如果在str中省略或不出现分隔符，则返回的数组包含一个由整个字符串组成的元素。如果分隔符为空字符串，则将str原字符串中每个字符的数组形式返回。
limit
一个整数，限定返回的分割片段数量。当提供此参数时，split 方法会在指定分隔符的每次出现时分割该字符串，但在限制条目已放入数组时停止。如果在达到指定限制之前达到字符串的末尾，它可能仍然包含少于限制的条目。新数组中不返回剩下的文本。

返回值

返回源字符串以分隔符出现位置分隔而成的一个 数组(Array)

slice()

> 方法提取某个字符串的一部分，并返回一个新的字符串，且不会改动原字符串。

```
var str = 'The quick brown fox jumps over the lazy dog.';

console.log(str.slice(31));
//  "the lazy dog."

console.log(str.slice(4, 19));
//  "quick brown fox"

console.log(str.slice(-4));
//  "dog."

console.log(str.slice(-9, -5));
//  "lazy"
```

参数

str.slice(beginIndex[, endIndex])

beginIndex
从该索引（以 0 为基数）处开始提取原字符串中的字符。如果值为负数，会被当做 strLength + beginIndex 看待，这里的 strLength 是字符串的长度（例如， 如果 beginIndex 是 -3 则看作是：strLength - 3)

endIndex
可选。在该索引（以 0 为基数）处结束提取字符串。如果省略该参数，slice() 会一直提取到字符串末尾。如果该参数为负数，则被看作是 strLength + endIndex，这里的 strLength 就是字符串的长度(例如，如果 endIndex 是 -3，则是, strLength - 3)。

返回值

返回一个从原字符串中提取出来的新字符串

trim()

> 方法会从一个字符串的两端删除空白字符。在这个上下文中的空白字符是所有的空白字符 (space, tab, no-break space 等) 以及所有行终止符字符（如 LF，CR等）。

```
var greeting = '   Hello world!   ';

console.log(greeting);
//  "   Hello world!   ";

console.log(greeting.trim());
//  "Hello world!";
```

返回值

一个依据调用字符串两端去掉空白的新字符串。

JavaScript 函数

> 函数是程序的主要“构建模块”。函数使该段代码可以被调用很多次，而不需要写重复的代码。
>
> 我们已经看到了内置函数的示例，如 `alert(message)`、`prompt(message, default)` 和 `confirm(question)`。但我们也可以创建自己的函数。
>
> 函数在javascript中是`一等公民`

函数定义

JavaScript 函数是被设计为执行特定任务的代码块。

JavaScript 函数会在某代码调用它时被执行。

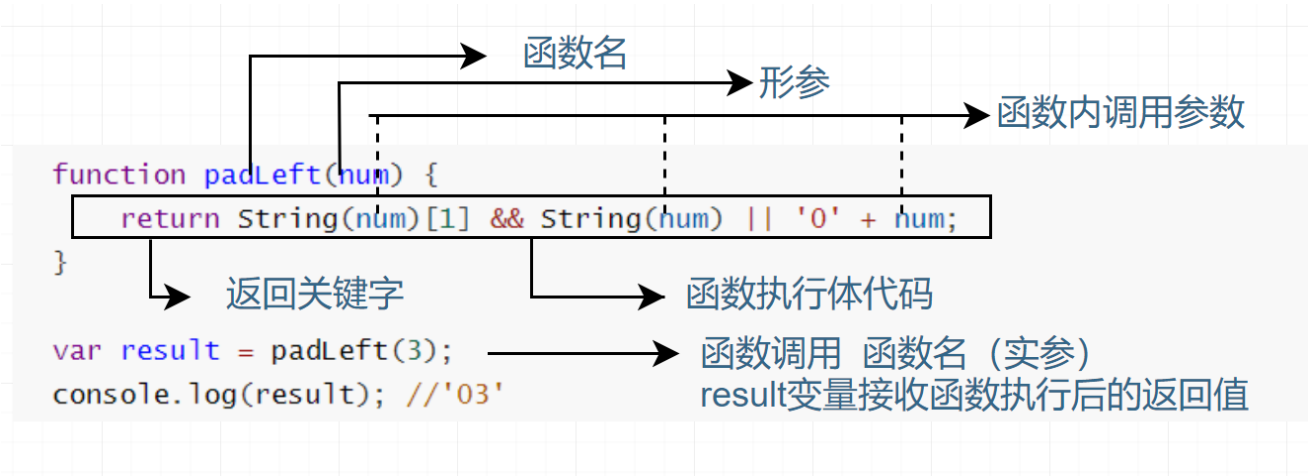
什么是函数?

> 把一段相对独立的具有特定功能的代码块封装起来，形成一个独立实体，就是函数，起个名字（函数名），在后续开发中可以反复调用
>
> 函数的基础作用就是封装一段代码，将来可以重复使用。我们称之为 **封装**

函数示例

```
function padLeft(num) {
   return String(num)[1] && String(num) || '0' + num;
}

var result = padLeft(3);
console.log(result); //'03'
```

分解图



函数的类型

> 函数拥有自己的数据类型 在某些老版本浏览器中会辨识为 object 需要用 Object.prototype.toString.call()来进行判断

```
function fn() {}
console.log(typeof fn);
//function

console.log(Object.prototype.toString.call(fn));
// "[object Function]"
```

函数的定义

> 函数的定义 有 **函数表达式** 和 **函数声明** 两种, 函数声明的时候，函数体并不会执行，只要当函数被调用的时候才会执行。
>
> JavaScript 函数通过 function 关键词进行定义，其后是*函数名*和括号 ()。
>
> 函数名可包含字母、数字、下划线和美元符号（规则与变量名相同）。

- 函数声明

```
function 函数名(){
  // 函数体
}
```

- 函数表达式

```
var 变量名 = function() {
  // 函数体
}
```

函数名:

函数一般都用来干一件事情，需用使用动词+名词 的驼峰结构命名，例如: `setValue` `clearInterval` 等。

函数调用

> 函数有声明就有调用，函数内代码只有在调用时才会执行。函数可以多次调用。

```
函数名(); //主动调用
```

```
//从指定数组中查询目标值
function findIdx(arr, target) {
    for (var i = 0, len = arr.length; i < len; i++) {
        if (arr[i] === target) {
            return i;

        }
```

```
    }
    return -1;
}

console.log(findIdx([1,2,3,4,5], 3)); //2

console.log(findIdx([1,2,5,4,5], 3)); // -1


// 求1-100之间所有数的和
function getSum() {
  var sum = 0;
  for (var  i = 1; i <= 100; i++) {
    sum += i;
  }
  console.log(sum);
}
// 调用
getSum(); //5050
```

函数的参数

> 函数的参数分为 **形参**(parameter) 和 **实参**(argument) , 当函数需求根据不同的值做出不同反馈时, 就需要通过使用参数。 **形参**为声明函数时预设的形势参数，我们可以自定义命名。 **实参**为调用函数时对应 **形参**所传递的实际参数，为所需要的原始值。

为什么要有参数

```
function getSum() {
  var sum = 0;
  for (var i = 1; i <= 100; i++) {
    sum += i;
  }
  console.log(sum);
}

// 虽然上面代码可以重复调用，但是只能计算1-100之间的值
// 如果想要计算n-m之间所有数的和，应该怎么办呢?
```

语法:

```
// 函数内部是一个封闭的环境，可以通过参数的方式，把外部的值传递给函数内部
// 带参数的函数声明
function 函数名(形参1, 形参2, 形参...){
  // 函数体
}

// 带参数的函数调用
函数名(实参1, 实参2, 实参3);
```

对比：

> 1. 形式参数：在声明一个函数的时候，为了函数的功能更加灵活，有些值是固定不了的，对于这些固定不了的值。我们可以给函数设置参数。这个参数没有具体的值，仅仅起到一个占位置的作用，我们通常称之为形式参数，也叫形参。
> 2. 实际参数：如果函数在声明时，设置了形参，那么在函数调用的时候就需要传入对应的参数，我们把传入的参数叫做实际参数，也叫实参。

```
var x = 5, y = 6;
fn(x,y);
function sum(num1, num2) {
  console.log(num1 + num2);
}
//x,y实参，有具体的值。函数执行的时候会把x,y复制一份给函数内部的a和b，函数内部的值是复制的新值，无法修改外部的x,y
```

注意:

如果函数调用时 没有传入实参 对应形参默认值为 `undefined`

JavaScript 参数通过*值*传递：函数只知道值，而不是参数的位置。

如果函数改变了参数的值，它不会改变参数的原始值。

参数的改变在函数之外是不可见的。

函数返回值

> 当 JavaScript 到达 return 语句，函数将停止执行。
>
> 如果函数被某条语句调用，JavaScript 将在调用语句之后“返回”执行代码。

返回值语法：

```
//声明一个带返回值的函数
function 函数名(形参1, 形参2, 形参...){
  //函数体
  return 返回值;
}

//可以通过变量来接收这个返回值
var 变量 = 函数名(实参1, 实参2, 实参3);
```

函数的调用结果就是返回值，因此我们可以直接对函数调用结果进行操作。

返回值详解：
　如果函数没有显示的使用 return语句 ，那么函数有默认的返回值：undefined
　如果函数使用 return语句，那么跟再return后面的值，就成了函数的返回值
　如果函数使用 return语句，但是return后面没有任何值，那么函数的返回值也是：undefined
　函数使用return语句后，这个函数会在执行完 return 语句之后停止并立即退出，也就是说return后面的所有其他代码都不会再执行。

arguments

JavaScript 函数有一个名为 arguments 对象的内置对象。

arguments 对象包含函数调用时使用的参数数组。

arguments为 (类数组)

这样，您就可以简单地使用函数来查找（例如）数字列表中的最高值：

实例

```
jsx = findMax(1, 123, 500, 115, 44, 88);
function findMax() {
   var i;
   var max = -Infinity;
   for (i = 0; i < arguments.length; i++) {
      if (arguments[i] > max) {
         max = arguments[i];
      }
   }
   return max;
}
```

或创建一个函数来总和所有输入值：

实例

```
jsx = sumAll(1, 123, 500, 115, 44, 88);
function sumAll() {
  var i, sum = 0;
  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

函数分类

> javascript非常适合函数式编程，这里简单介绍下几种基础函数方式的基础应用

纯函数

> 纯函数(Pure Function)
>
> 1. 如果函数的调用参数相同，则永远返回相同的结果。它不依赖于程序执行期间**函数**外部任何状态或数据的变化，必须只依赖于其**输入参数**。
> 2. 该函数不会产生任何可观察的**副作用**

```
function padLeft(num) {
  return String(num)[1] && String(num) || '0' + num;
}
```

非纯函数(函数副作用)

> 所谓"副作用"（side effect），指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

```
var str = '海牙';
function changeStr(){
 str = 'kyogre';
}
console.log(str); //海牙
change();
console.log(str); //kyogre
```

匿名函数

> 匿名函数：没有名字的函数

匿名函数如何使用：

```
将匿名函数赋值给一个变量，这样就可以通过变量进行调用
匿名函数自调用
```

关于自执行函数（匿名函数自调用）的作用：防止全局变量污染。

自调用函数

> 匿名函数(IIFE)不能通过直接调用来执行，因此可以通过匿名函数的自调用的方式来执行

```
(function () {
  alert(123);
})();
```

柯理化函数

> 柯理化(currying)
>
> 函数可以用来作为返回值
>
> 函数柯里化允许和鼓励你分隔复杂功能变成更小更容易分析的部分。这些小的逻辑单元显然是更容易理解和测试的，然后你的应用就会变成干净而整洁的组合，由一些小单元组成的组合。

```
//纯函数
function add(x, y) {
   return x + y;
}
console.log(add(1, 2)); // 3

//柯理化
function curryingOfAdd(x) {
   return function (y) {
      return x + y;
   }
}

console.log(curryingOfAdd(1)(2)); // 3

var tranCurry = add(3);
tranCurry(2); //5
tranCurry(8); //11
```

> `回调函数` `偏函数` `递归函数` `深度柯理化函数` `通道函数` `闭包` `高阶函数` 等函数式编程技巧会在后续学习中慢慢展开

作用域

作用域：变量可以起作用的范围

全局变量和局部变量

- 全局变量

  在任何地方都可以访问到的变量就是全局变量，对应全局作用域

- 局部变量

  只在固定的代码片段内可访问到的变量，最常见的例如函数内部。对应局部作用域(函数作用域)

```
不使用var声明的变量是全局变量，不推荐使用。
变量退出作用域之后会销毁，全局变量关闭网页或浏览器才会销毁
```

块级作用域

任何一对花括号（｛和｝）中的语句集都属于一个块，在这之中定义的所有变量在代码块外都是不可见的，我们称之为块级作用域。
在es5之前没有块级作用域的的概念,只有函数作用域，现阶段可以认为JavaScript没有块级作用域

词法作用域

变量的作用域是在定义时决定而不是执行时决定，也就是说词法作用域取决于源码，通过静态分析就能确定，因此词法作用域也叫做静态作用域。

在 js 中词法作用域规则:

- 函数允许访问函数外的数据.
- 整个代码结构中只有函数可以限定作用域.
- 作用域规则首先使用提升规则分析
- 如果当前作用规则中有名字了, 就不考虑外面的名字

```
var num = 123;
function foo() {
  console.log( num );
}
foo();
```

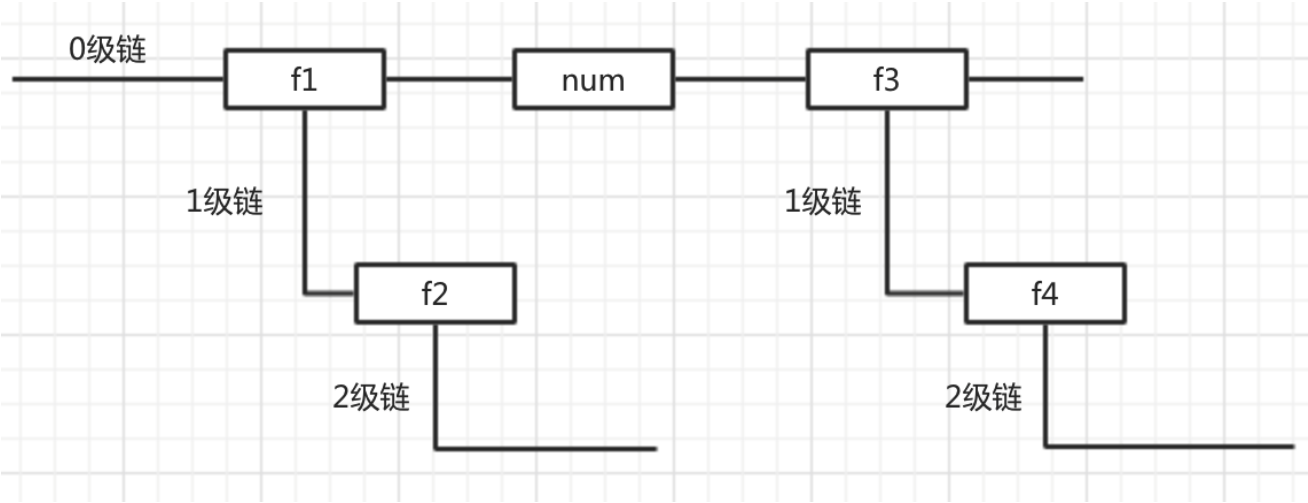
```
if ( false ) {
    var num = 123;
}
console.log(num); // undefiend
```

作用域链

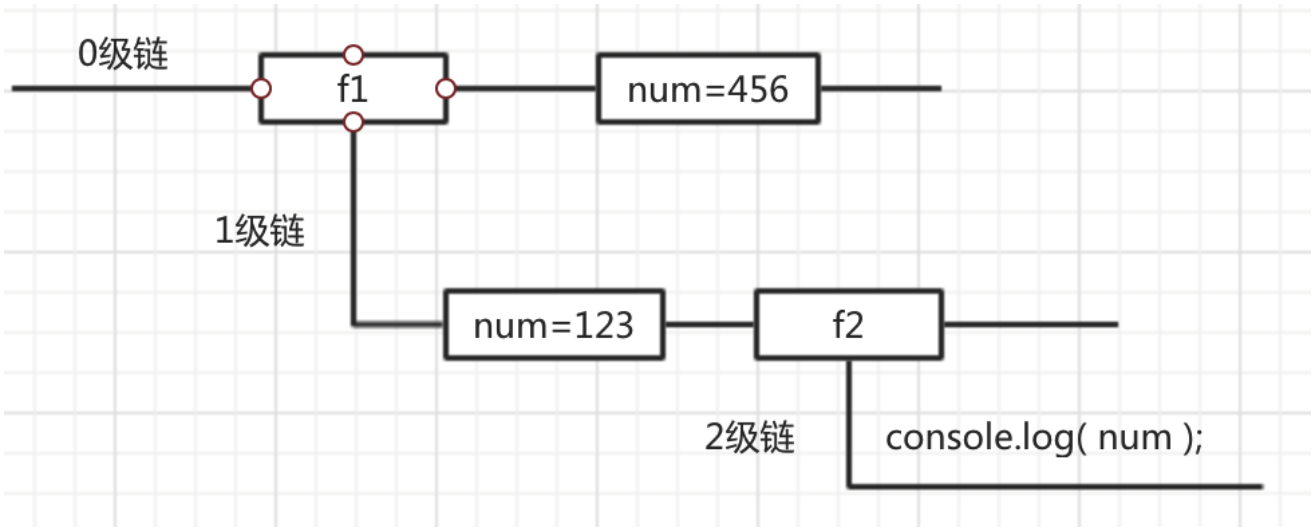
只有函数可以制造作用域结构， 那么只要是代码，就至少有一个作用域, 即全局作用域。凡是代码中有函数，那么这个函数就构成另一个作用域。如果函数中还有函数，那么在这个作用域中就又可以诞生一个作用域。

将这样的所有的作用域列出来，可以有一个结构: 函数内指向函数外的链式结构。就称作作用域链。

```
// 案例1:
function f1() {
    function f2() {
    }
}

var num = 456;
function f3() {
    function f4() {
    }
}
```




```
// 案例2
function f1() {
    var num = 123;
    function f2() {
        console.log( num );
    }
    f2();
}
var num = 456;
f1();
```



预解析

> JavaScript代码的执行是由浏览器中的JavaScript解析器来执行的。JavaScript解析器执行JavaScript代码的时候，分为两个过程：预解析过程和代码执行过程

预解析过程：

1. 把变量的声明提升到当前作用域的最前面，只会提升声明，不会提升赋值。
2. 把函数的声明提升到当前作用域的最前面，只会提升声明，不会提升调用。
3. 先提升var，在提升function

JavaScript的执行过程

```
var a = 25;
function abc (){
  alert(a);//undefined
  var a = 10;
}
abc();
// 如果变量和函数同名的话，函数优先
console.log(a);
function a() {
  console.log('aaaaa');
}
var a = 1;
console.log(a);
```

全局解析规则

函数内部解析规则

变量提升

- 变量提升

  定义变量的时候，变量的声明会被提升到作用域的最上面，变量的赋值不会提升。

- 函数提升

  JavaScript解析器首先会把当前作用域的函数声明提前到整个作用域的最前面

```
// 1、-----------------------------------
var num = 10;
fun();
function fun() {
  console.log(num);
  var num = 20;
}
//2、-----------------------------------
var a = 18;
f1();
function f1() {
  var b = 9;
  console.log(a);
  console.log(b);
  var a = '123';
}
// 3、-----------------------------------
f1();
console.log(c);
console.log(b);
console.log(a);
function f1() {
  var a = b = c = 9;
  console.log(a);
  console.log(b);
  console.log(c);
}
```

对象 Object

> 什么是对象？咱们要说的对象可不是 `女朋友`，在这个世界上 任何**具体事物**都可以看做 `对象` 因为他们都有自己的特征、行为。

车子 手机 猫 是对象吗?
这些都是一类事物, 只有具体的事物才是对象 , 比如我家的小黄猫。

小黄猫
 特征:
      颜色：黄色
      年龄：1岁
      体重：5kg
      最爱：小鱼干
      名字: 橘子
  行为:
    吃饭 睡觉 伸懒腰 喵喵喵

> JavaScript是一门基于对象的语言。
>
> javascript中 我们称Object为 **对象** 对象的概念也分广义和狭义：广义上javascript中处处是对象，狭义指的是我们通过{}**字面量**创建的对象。

JavaScript的对象是无序属性的集合。
 其属性可以包含基本值、对象或函数。对象就是一组没有顺序的值。我们可以把JavaScript中的对象想象成键值对，其中值可以是数据和函数。
对象的行为和特征
 特征---属性
 行为---方法
事物的特征在对象中用属性来表示。
事物的行为在对象中用方法来表示。

字面量

> 字面量(literal)是用于表达源代码中一个固定值的表示法(notation).

> 几乎所有计算机编程语言都具有对基本值的字面量表示, 诸如: 整数, 浮点数以及字符串; 而有很多也对布尔类型和字符类型的值也支持字面量表示; 还有一些甚至对枚举类型的元素以及像数组, 记录和对象等复合类型的值也支持字面量表示法.
>
> 字面量（literal），在高级语言中 我们可以通过更直观更高效的方式直接赋予变量 具体 `值`，当需要使用值得时候 才会去根据值得类型和内容进行包装解析; 先存储 后解释 。

javascript中字面量包括

1. 字符串字面量（String Literal）

```
var str = '张晓华'; //张晓华 就是字符串字面量
```

2.数组字面量(array literal)

```
var arr = [1,2,3,4,5]; //[1,2,3,4,5] 就是数组字面量
```

3.对象字面量(object literal)

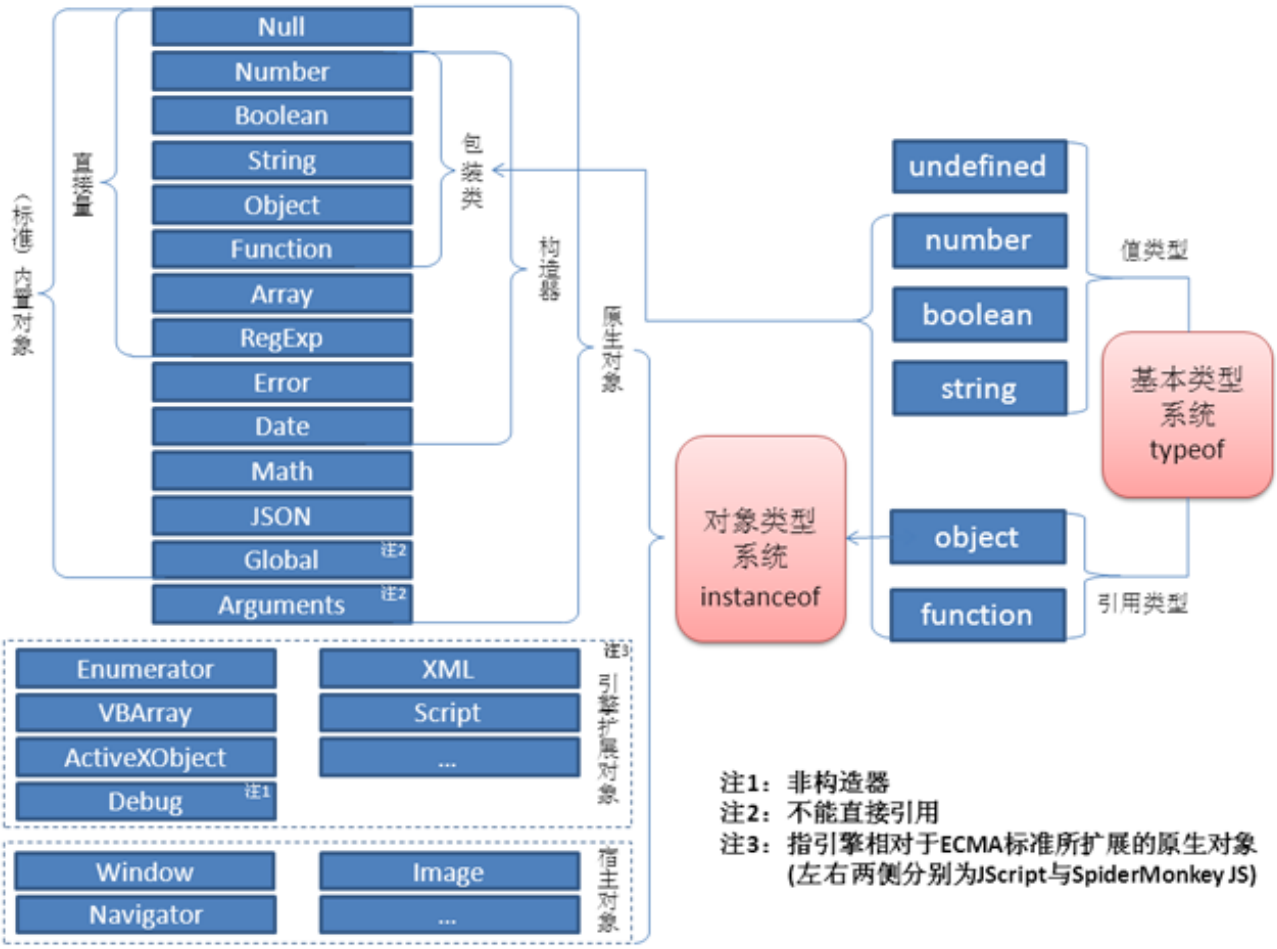
```
var obj = {
   name:'橘子',
   age: 1,
   favorite: '小鱼干'
}

/*
{
   name:'橘子',
   age: 1,
   favorite: '小鱼干'
}

就是对象字面量


*/
```

4.函数字面量(function literal)

```
var fn = function(){
   alert('你好');
}

/*
 function(){
   alert('你好');
}
就是函数字面量

*/
```

javascript中的内置对象

> JavaScript中的对象分为3种：内置对象、浏览器对象、自定义对象
>
> javascript是基于对象的语言, javascript不可以自己创建一个原生对象,只能访问内置对象属性和调用已有的内置对象方法。但是可以通过基于Object创建狭义概念上的`对象`。



```
Number //数字
Array  //数组
Boolean // 布尔
String //字符串
Object //对象
Function //函数
Date //时间
Math //数学
Null //空
RegExp //正则对象
```

创建对象方式(自定义)

> 创建对象有三种基础方式：对象字面量 原生对象实例化 自定义构造函数

- 字面量对象

```
var myCat = {
   name: '橘子',
   color: 'orange',
   age: 1,
   favorite: '小鱼干',
   speak: function(){
     console.log('喵~~喵~喵~~~');
   }
}
```

- 原生对象实例化 new Object()

```
var myCat = new Object();
myCat.name = '橘子';
myCat.color = 'orange';
myCat.age = 1;
myCat.favorite = '小鱼干';
myCat.speak = function(){
   console.log('喵~~喵~喵~~~');
}
```

- 工厂函数创建对象

```
function createCat(name,age,color){
    var cat = new Object();
    cat.name = name;
    cat.color = color;
    cat.age = age;
    cat.favorite = '小鱼干';
    cat.speak = function(){
        console.log('喵~~喵~喵~~~');
    }
    return cat;
}

var myCat = createCat('橘子',1,'orange');
```

- 自定义构造函数

```
function Cat(name,age,color,favorite){
    this.name = name;
    this.age = age;
    this.color = color;
    this.favorite = favorite;
    this.speak = function(){
        console.log('喵~~喵~喵~~~');
    }
}
var myCat = new Cat('橘子',1,'orange','小鱼干');
```

属性 方法

如果一个变量属于一个对象所有，那么该变量就可以称之为该对象的一个属性，属性一般是名词，用来描述事物的特征
如果一个函数属于一个对象所有，那么该函数就可以称之为该对象的一个方法，方法是动词，描述事物的行为和功能

instanceof

在 JavaScript 中，判断一个变量的类型尝尝会用 typeof 运算符，在使用 typeof 运算符时采用引用类型存储值会出现一个问题，无论引用的是什么类型的对象，它都返回 “object”。ECMAScript 引入了另一个 Java 运算符 instanceof 来解决这个问题。instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。例如：

```
var oStringObject = new String("hello world");
console.log(oStringObject instanceof String);     // 输出 "true"
```

构造函数

> 面向对象编程'的第一步，就是要生成对象。而js中面向对象编程是基于构造函数（constructor）和原型链（prototype）的。
>
> 前面说过，“对象”是单个实物的抽象。通常需要一个模板，表示某一类实物的共同特征，然后“对象”根据这个模板生成。
>
> js语言中使用构造函数（constructor）作为对象的模板。所谓构造函数，就是提供一个生成对象的模板，并描述对象的基本结构的函数。一个构造函数，可以生成多个对象，每个对象都有相同的结构。

```
function Person(name,age,sex){ //Person就是构造函数
    this.name = name;
    this.age = age;
    this.sex = sex;
    this.speak = function(){
        console.log('我叫' + this.name + ',今年:' + this.age + '岁,性别:' + this.sex);
    }
}
```

```
    a：构造函数的函数名的第一个字母通常大写。

    b：函数体内使用this关键字，代表所要生成的对象实例。

    c：生成对象的时候，必须使用new命令来调用构造函数。
```

new关键字

> 构造函数 ，是一种特殊的函数。主要用来在创建对象时初始化对象， 即为对象成员变量赋初始值，总与new运算符一起使用在创建对象的语句中。

1. 构造函数用于创建一类对象，首字母要大写。
2. 构造函数要和new一起使用才有意义。

new在执行时会做四件事情

```
1 创建一个空对象，作为将要返回的对象实例。

2 将空对象的原型指向了构造函数的prototype属性。

3 将空对象赋值给构造函数内部的this关键字。

4 开始执行构造函数内部的代码。
```

this详解

```
JavaScript中的this指向问题，有时候会让人难以捉摸，随着学习的深入，我们可以逐渐了解
现在我们需要掌握函数内部的this几个特点
    1. 函数在定义的时候this是不确定的，只有在调用的时候才可以确定
    2. 一般函数直接执行，内部this指向全局window
    3. 函数作为一个对象的方法，被该对象所调用，那么this指向的是该对象
    4. 构造函数中的this其实是一个隐式对象，类似一个初始化的模型，所有方法和属性都挂载到了这个隐式对象身上，后续通过new关键字来调用，从而实现实例化
```

对象的使用

遍历对象的属性

> 通过for..in语法可以遍历一个对象

```
var obj = {};

obj.name = 'object';
obj['name'] = 'object';

var key = 'name';

console.log(obj[key]);

for (var i = 0; i < 10; i++) {
  obj[i] = i * 2;
}

for(var key in obj) {
  console.log(key + "==" + obj[key]);
}
```

删除对象的属性

```
function fun() {
  this.name = 'mm';
}
var obj = new fun();
console.log(obj.name); // mm
delete obj.name;
console.log(obj.name); // undefined
```

JSON格式对象

> JSON即Javascript对象表示方法 (Javascript Object Notation) ,也就是通过字面量来表示一个对象：

JSON 英文全称 JavaScript Object Notation
JSON 是一种轻量级的数据交换格式。
它基于 ECMAScript (欧洲计算机协会制定的js规范)的一个子集

JSON 语法：

数据使用键名/值对表示，键名是字符串，值没有限定；
例如 “language”:”Java”
每个数据之间由逗号分隔；
使用大括号保存对象，对象可以包含若干个数据；
使用方括号保存数组，数组值使用“,”分割；

JSON数据使用”’键名”：”值”’的形式，其中键名要求是字符串，而值 可以是以下任意类型：
1. 数值（整数，浮点数）
2. 字符串（在双引号中）
3. 逻辑值（true/false）
4. 数组（在方括号中）
5. 对象（在花括号中）

JSON 示例:

```
var nodeDate = {
  "NodeName" : "P",
    "NodeType" : "Node.ELEMENT_NODE",
    "NodeId" : "",
    "NodeClassName" : "des",
    "NodeStyle": "text-align:center;color:blue;font-size:22px;text-indent:2em;",
    "NodeContent":"哈哈哈哈",
    "NodeChildElement" : []
}
```

```
var studentList = [
    { "name": "张三", "age": 18, "sex": 1 },
    { "name": "李思", "age": 19, "sex": 0 },
    { "name": "王武", "age": 98, "sex": 1 },
    { "name": "赵柳", "age": 77, "sex": 0 }
]
```

```
var chinaMap = [
    {
      "code": "33",
      "name": "浙江省",
      "children": [
        {
          "code": "3301",
          "name": "杭州市"
        },
        {
          "code": "3302",
          "name": "宁波市"
        },
        {
          "code": "3303",
          "name": "温州市"

        }
```

```
    ]
   },
   {
    "code": "32",
    "name": "湖南省",
    "children": [
     {
      "code": "3201",
      "name": "长沙市"
     },
     {
      "code": "3202",
      "name": "株洲市"
     },
     {
      "code": "3203",
      "name": "湘潭市"
     }
    ]
   }
  ]
```

JSON方法API:

JSON.parse() 反序列化

> 将JSON数据解析为Javascript对象
>
> 方法用来解析JSON字符串，构造由字符串描述的JavaScript值或对象。提供可选的 **reviver** 函数用以在返回之前对所得到的对象执行变换(操作)。

```
var json = '{"result":true, "count":42}';
var obj = JSON.parse(json);

console.log(obj.count);
//42

console.log(obj.result);
// true
```

参数

JSON.parse(text[, reviver])

text
要被解析成 JavaScript 值的字符串
reviver 可选
转换器, 如果传入该参数(函数)，可以用来修改解析生成的原始值，调用时机在 parse 函数返回之前。

返回值

Object 类型, 对应给定 JSON 文本的对象/值。

异常

若传入的字符串不符合 JSON 规范，则会抛出 SyntaxError 异常

JSON.stringify() 序列化

> 方法将一个 JavaScript 对象或值转换为 JSON 字符串，如果指定了一个 replacer 函数，则可以选择性地替换值，或者指定的 replacer 是数组，则可选择性地仅包含数组指定的属性。

```
console.log(JSON.stringify({ x: 5, y: 6 }));
//  "{"x":5,"y":6}"

console.log(JSON.stringify([new Number(3), new String('false'), new Boolean(false)]));
//"[3,"false",false]"

console.log(JSON.stringify({ x: [10, undefined, function(){}, Symbol('')] }));
//  "{"x":[10,null,null,null]}"

console.log(JSON.stringify(new Date(2006, 0, 2, 15, 4, 5)));
//  ""2006-01-02T15:04:05.000Z""
```

参数

JSON.stringify(value[, replacer [, space]])

value
将要序列化成 一个 JSON 字符串的值。
replacer 可选
如果该参数是一个函数，则在序列化过程中，被序列化的值的每个属性都会经过该函数的转换和处理；如果该参数是一个数组，则只有包含在这个数组中的属性名才会被序列化到最终的 JSON 字符串中；如果该参数为 null 或者未提供，则对象所有的属性都会被序列化。
space 可选
指定缩进用的空白字符串，用于美化输出（pretty-print）；如果参数是个数字，它代表有多少的空格；上限为10。该值若小于1，则意味着没有空格；如果该参数为字符串（当字符串长度超过10个字母，取其前10个字母），该字符串将被作为空格；如果该参数没有提供（或者为 null），将没有空格。

返回值

一个表示给定值的JSON字符串。

异常

当发现循环引用时，抛出类型错误TypeError("循环对象值")异常。
当试图将BigInt (BigInt 为javascript中Number能表示的最大数 2^53 - 1)值字符串化时，会抛出类型错误 TypeError("BigInt值不能在JSON中序列化")。

执行上下文

1. 什么是执行上下文

简而言之，执行上下文就是当前 JavaScript 代码被解析和执行时所在环境的抽象概念， JavaScript 中运行任何的代码都是在执行上下文中运行

2. 执行上下文的类型

执行上下文总共有三种类型：

- 全局执行上下文： 这是默认的、最基础的执行上下文。不在任何函数中的代码都位于全局执行上下文中。它做了两件事：1. 创建一个全局对象，在浏览器中这个全局对象就是 window 对象。2. 将 this 指针指向这个全局对象。一个程序中只能存在一个全局执行上下文。
- 函数执行上下文： 每次调用函数时，都会为该函数创建一个新的执行上下文。每个函数都拥有自己的执行上下文，但是只有在函数被调用的时候才会被创建。一个程序中可以存在任意数量的函数执行上下文。每当一个新的执行上下文被创建，它都会按照特定的顺序执行一系列步骤，具体过程将在本文后面讨论。
- Eval 函数执行上下文： 运行在 eval 函数中的代码也获得了自己的执行上下文，但由于eval是魔鬼 我们一般不通过eval进行开发操作。

执行上下文的生命周期

执行上下文的生命周期包括三个阶段：**创建阶段** → **执行阶段** → **回收阶段**，本文重点介绍创建阶段。

1. 创建阶段

当函数被调用，但未执行任何其内部代码之前，会做以下三件事：

- 创建变量对象：首先初始化函数的参数 arguments，提升函数声明和变量声明。下文会详细说明。
    - 创建作用域链（Scope Chain）：在执行期上下文的创建阶段，作用域链是在变量对象之后创建的。作用域链本身包含变量对象。作用域链用于解析变量。当被要求解析变量时，JavaScript 始终从代码嵌套的最内层开始，如果最内层没有找到变量，就会跳转到上一层父作用域中查找，直到找到该变量。
- 确定 this 指向：包括多种情况，下文会详细说明

在一段 JS 脚本执行之前，要先解析代码（所以说 JS 是解释执行的脚本语言），解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来。变量先暂时赋值为 undefined，函数则先声明好可使用。这一步做完了，然后再开始正式执行程序。

另外，一个函数在执行之前，也会创建一个函数执行上下文环境，跟全局上下文差不多，不过 函数执行上下文中会多出 this arguments 和函数的参数。

2. 执行阶段

执行变量赋值、代码执行

3. 回收阶段

执行上下文出栈等待虚拟机回收执行上下文

变量提升和 this 指向的细节

1. 变量声明提升

大部分编程语言都是先声明变量再使用，但在 JS 中，事情有些不一样：

```
console.log(a); // undefined
var a = 10;
```

上述代码正常输出`undefined`而不是报错`Uncaught ReferenceError: a is not defined`,这是因为声明提升（hoisting），相当于如下代码：

```
var a; //声明 默认值是undefined “准备工作”
console.log(a);
a = 10; //赋值
```

2. 函数声明提升

我们都知道，创建一个函数的方法有两种，一种是通过函数声明`function foo(){}`另一种是通过函数表达式`var foo = function(){}`,那这两种在函数提升有什么区别呢?

```
console.log(f1); // function f1(){}
function f1() {} // 函数声明
console.log(f2); // undefined

var f2 = function() {}; // 函数表达式
```

接下来我们通过一个例子来说明这个问题：

```
function test() {
   foo(); // 未捕获类型错误“foo不是函数”
   bar(); // "this will run!"
   var foo = function() {
      console.log("this won't run!");
   };
   function bar() {
      ale console.logrt("this will run!");
   }
}
test();
```

在上面的例子中，foo()调用的时候报错了，而 bar 能够正常调用。

我们前面说过变量和函数都会上升，遇到函数表达式 `var foo = function(){}` 时，首先会将 `var foo` 上升到函数体顶部，然而此时的 foo 的值为 undefined,所以执行 `foo()` 报错。

而对于函数 `bar()`, 则是提升了整个函数，所以 `bar()` 才能够顺利执行。

有个细节必须注意：**当遇到函数和变量同名且都会被提升的情况，函数声明优先级比较高，因此变量声明会被函数声明所覆盖，但是可以重新赋值。**

```
alert(a); //输出：function a(){ alert('我是函数') }
function a() {
   alert("我是函数");
} //
var a = "我是变量";
alert(a); //输出：'我是变量'
```

function 声明的优先级比 var 声明高，也就意味着当两个同名变量同时被 function 和 var 声明时，function 声明会覆盖 var 声明

这代码等效于：

```
function a() {
   alert("我是函数");
}
var a; //hoisting
alert(a); //输出：function a(){ alert('我是函数') }
a = "我是变量"; //赋值
alert(a); //输出：'我是变量'
```

最后我们看个复杂点的例子：

```
function test(arg) {
   // 1. 形参 arg 是 "hi"

   // 2. 因为函数声明比变量声明优先级高，所以此时 arg 是 function
```

```
    console.log(arg);
    var arg = "hello"; // 3.var arg 变量声明被忽略， arg = 'hello'被执行
    function arg() {
        console.log("hello world");
    }
    console.log(arg);
}
test("hi");
/* 输出：
function arg(){
    console.log('hello world')
    }
hello
*/
```

这是因为当函数执行的时候,首先会形成一个新的私有的作用域，然后依次按照如下的步骤执行：

- 如果有形参，先给形参赋值
- 进行私有作用域中的预解释，函数声明优先级比变量声明高，最后后者会被前者所覆盖，**但是可以重新赋值**
- 私有作用域中的代码从上到下执行

3. 确定 this 的指向

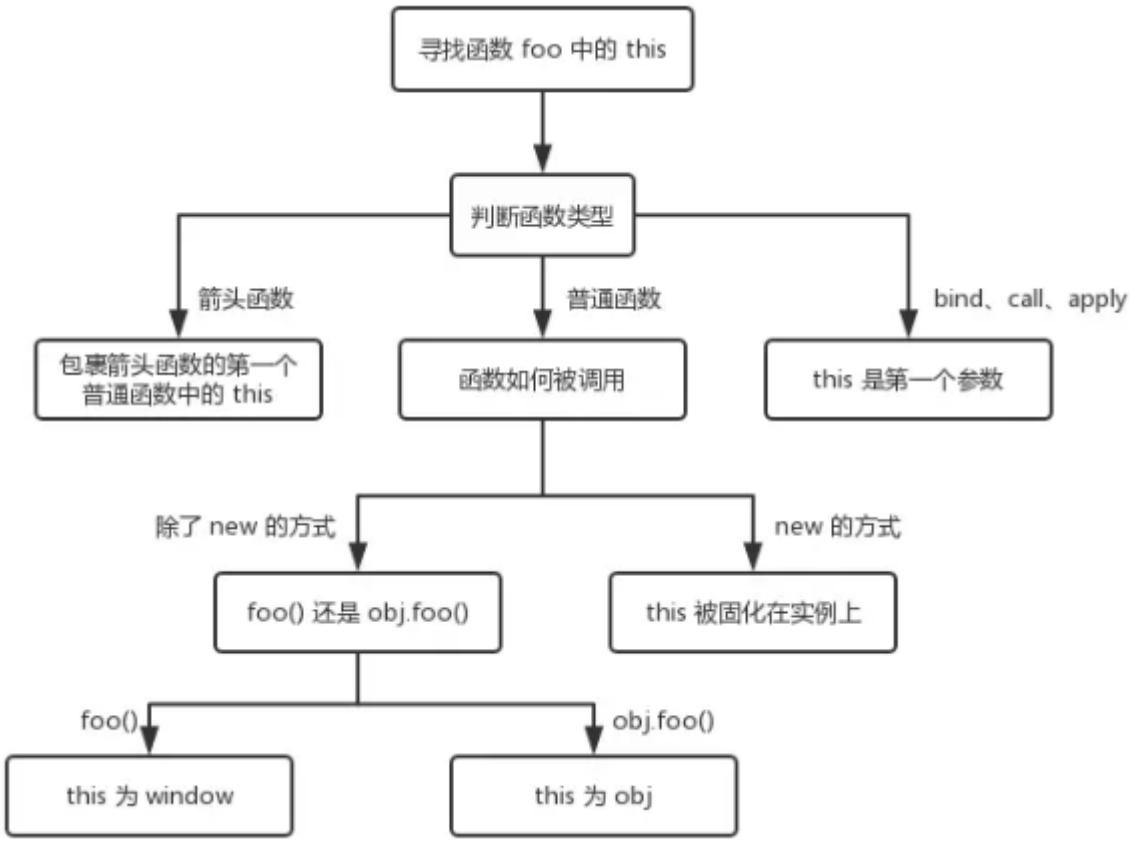
先搞明白一个很重要的概念 —— **this 的值是在执行的时候才能确认，定义的时候不能确认！** 为什么呢 —— 因为 this 是执行上下文环境的一部分，而执行上下文需要在代码执行之前确定，而不是定义的时候。看如下例子：

```
// 情况1
function foo() {
  console.log(this.a) //1
}
var a = 1
foo()

// 情况2
function fn(){
  console.log(this);
}
var obj={fn:fn};
obj.fn(); //this->obj

// 情况3
function CreateJsPerson(name,age){
//this是当前类的一个实例p1
this.name=name; //=>p1.name=name
this.age=age; //=>p1.age=age
}
var p1=new CreateJsPerson("海牙",18);

//情况4
var obj = {
    name:'海牙',
    showName:function(){
        console.log(this.name); // 海牙 this => obj
```

```
        (function(){
            console.log(this.name); //undefined this=>window
        })();
    }
}
obj.showName();

</script>
```

接下来我们逐一解释上面几种情况

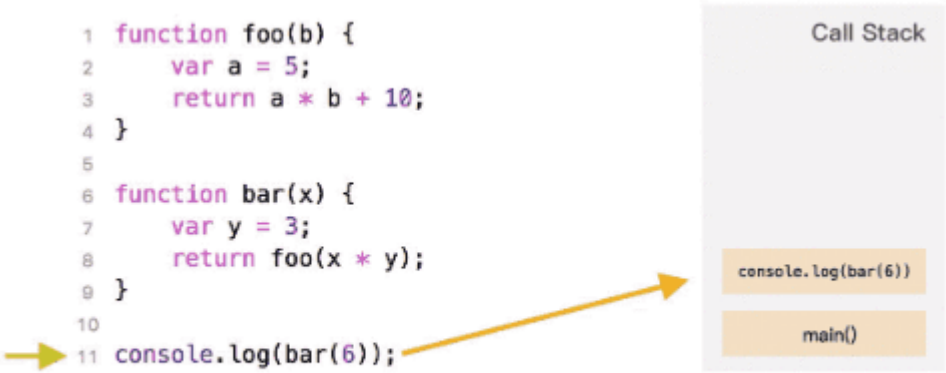
- 对于直接调用 foo 来说，不管 foo 函数被放在了什么地方，this 一定是 window
- 对于 obj.foo() 来说，我们只需要记住，谁调用了函数， this指向谁，所以在这个场景下 foo 函数中的 this 就是 obj 对象
- 在构造函数模式中，类中(函数体中)出现的 this.xxx=xxx 中的 this 是当前类的一个实例
- IIFE匿名函数自调用时 因为没有明确调用主体 this指向 window。



执行上下文栈（Execution Context Stack）

函数多了，就有多个函数执行上下文，每次调用函数创建一个新的执行上下文，那如何管理创建的那么多执行上下文呢?

JavaScript 引擎创建了执行上下文栈来管理执行上下文。**可以把执行上下文栈认为是一个存储函数调用的栈结构，遵循先进后出的原则。**



输出：

从上面的流程图，我们需要记住几个关键点：

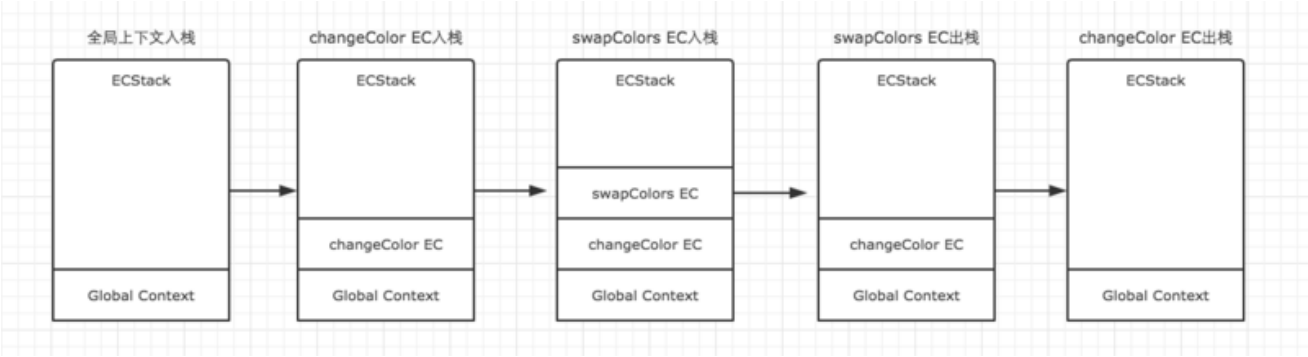
- JavaScript 执行在单线程上，所有的代码都是排队执行。
- 一开始浏览器执行全局的代码时，首先创建全局的执行上下文，压入执行栈的顶部。
- 每当进入一个函数的执行就会创建函数的执行上下文，并且把它压入执行栈的顶部。当前函数执行完成后，当前函数的执行上下文出栈，并等待垃圾回收。
- 浏览器的 JS 执行引擎总是访问栈顶的执行上下文。
- 全局上下文只有唯一的一个，它在浏览器关闭时出栈。

我们再来看个例子：

```
var color = "blue";
function changeColor() {
  var anotherColor = "red";
  function swapColors() {
    var tempColor = anotherColor;
    anotherColor = color;
    color = tempColor;
  }
  swapColors();
}
changeColor();
```

上述代码运行按照如下步骤：

- 当上述代码在浏览器中加载时，JavaScript 引擎会创建一个全局执行上下文并且将它推入当前的执行栈
- 调用 changeColor 函数时，此时 changeColor 函数内部代码还未执行，js 执行引擎立即创建一个 changeColor 的执行上下文（简称 EC），然后把这执行上下文压入到执行栈（简称 ECStack）中。
- 执行 changeColor 函数过程中，调用 swapColors 函数，同样地，swapColors 函数执行之前也创建了一个 swapColors 的执行上下文，并压入到执行栈中。
- swapColors 函数执行完成，swapColors 函数的执行上下文出栈，并且被销毁。
- changeColor 函数执行完成，changeColor 函数的执行上下文出栈，并且被销毁。



#包装对象

> 基础类型undefined、null、boolean、number、string和symbol 都是非引用类型(非对象)
>
> 之前我们学习了对象 了解到 对象具有属性和方法 也学习了字面量值 我们一起来看一个案例

```
var str = '你好';
var arr = str.split('');
console.log(arr);
// ["你", "好"]
```

问题:

一个以string基础类型的字面量为值的变量str 是如何能够通过 . 操作执行方法的呢? str到底是对象 还是字符串呢?

```
var str = '你好';
//var newStr = new String('你好'); 包装 实例化对象
var arr = str.split('');
//var arr = newStr.split(''); 通过包装对象调用对象方法
// newStr = null; 销毁包装对象
console.log(arr);
// ["你", "好"]
```

#常用内置对象方法

> javascript内置对象都提供了基本方法, 学习方法需要注意 **参数 返回值 是否改变原对象** 三个点。
>
> MDN 官方文档 : https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects

Object方法

Object.is()

> 方法判断两个值是否为**同一个值**

Object.is() 方法判断两个值是否为同一个值。如果满足以下条件则两个值相等:

都是 undefined
都是 null
都是 true 或 false
都是相同长度的字符串且相同字符按相同顺序排列
都是相同对象（意味着每个对象有同一个引用）
都是数字且
都是 +0
都是 -0
都是 NaN
或都是非零而且非 NaN 且为同一个值
与== 运算不同。 == 运算符在判断相等前对两边的变量(如果它们不是同一类型) 进行强制转换 (这种行为的结果会将 "" == false 判断为 true), 而 Object.is不会强制转换两边的值。

与=== 运算也不相同。 === 运算符 (也包括 == 运算符) 将数字 -0 和 +0 视为相等 ， 而将Number.NaN 与NaN视为不相等.

参数

Object.is(value1, value2);

value1
被比较的第一个值。
value2
被比较的第二个值。

返回值

一个 Boolean 类型标示两个参数是否是同一个值。 true or false

Object.freeze()

> 方法可以**冻结**一个对象。一个被冻结的对象再也不能被修改；冻结了一个对象则不能向这个对象添加新的属性，不能删除已有属性，不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。此外，冻结一个对象后该对象的原型也不能被修改。`freeze()` 返回和传入的参数相同的对象。

```
var obj = {
  prop: 42
};

Object.freeze(obj);

obj.prop = 33;
// Throws an error in strict mode

console.log(obj.prop);
// 42
```

参数

```
Object.freeze(obj)
```

obj
要被冻结的对象。

返回值

被冻结的对象。

Object.assign()

> 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

```
var target = { a: 1, b: 2 };
var source = { b: 4, c: 5 };

var returnedTarget = Object.assign(target, source);

console.log(target);
//  { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
//  { a: 1, b: 4, c: 5 }
```

如果目标对象中的属性具有相同的键，则属性将被源对象中的属性覆盖。后面的源对象的属性将类似地覆盖前面的源对象的属性。

参数

Object.assign(target, ...sources)

target
目标对象。

sources
源对象。

返回值

目标对象。

Object.keys()

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

```
Object.keys({a:1,b:2,c:3,d:5});
// ["a", "b", "c", "d"]

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

Object.keys 返回一个所有元素为字符串的数组，其元素来自于从给定的object上面可直接枚举的属性。这些属性的顺序与手动遍历该对象属性时的一致。

参数

Object.keys(obj)

obj
要返回其枚举自身属性的对象。

返回值

一个表示给定对象的所有可枚举属性的字符串数组。

Array方法

forEach()

> 方法对数组的每个元素执行一次给定的函数。

```
var items = ['item1', 'item2', 'item3'];
var copy = [];

// for写法
for (var i=0; i<items.length; i++) {
  copy.push(items[i]);
}

// forEach方法
items.forEach(function(item){ //function匿名函数作为实参
  copy.push(item);
});
```

参数

```
arr.forEach(callback(currentValue [, index [, array]])[, thisArg])

callback
为数组中每个元素执行的函数，该函数接收一至三个参数：
    currentValue
    数组中正在处理的当前元素。
    index 可选
    数组中正在处理的当前元素的索引。
    array 可选
    forEach() 方法正在操作的数组。
thisArg 可选
可选参数。当执行回调函数 callback 时，用作 this 的值。
```

返回值

```
undefined
```

sort()

> 方法对数组的元素进行排序，并返回数组。默认排序顺序是在将元素转换为字符串，然后比较它们的UTF-16代码单元值序列时构建的

```
const months = ['March', 'Jan', 'Feb', 'Dec'];
months.sort();
console.log(months);
//  ["Dec", "Feb", "Jan", "March"]

const array1 = [1, 30, 4, 21, 100000];
array1.sort();
console.log(array1);
//  [1, 100000, 21, 30, 4]

var numbers = [4, 2, 5, 1, 3];
numbers.sort(function(a, b) {
  return a - b;
});
console.log(numbers);
// [1, 2, 3, 4, 5]

var numbers = [4, 2, 5, 1, 3];
numbers.sort(function(a, b) {
  return b - a;
});
console.log(numbers);
// [5, 4, 3, 2, 1]
```

如果没有指明 排序函数 ，那么元素会按照转换为的字符串的诸个字符的Unicode位点进行排序。例如 "Banana" 会被排列到 "cherry" 之前。当数字按由小到大排序时，9 出现在 80 之前，但因为（没有指明 排序函数），比较的数字会先被转换为字符串，所以在Unicode顺序上 "80" 要比 "9" 要靠前。

参数

```
arr.sort( [,compareFunction(firstEl,secondEl)])

compareFunction 可选
用来指定按某种顺序进行排列的函数。如果省略，元素按照转换为的字符串的各个字符的Unicode位点进行排序。
   firstEl
   第一个用于比较的元素。
   secondEl
   第二个用于比较的元素。
```

返回值

排序后的数组。请注意，数组已原地排序，不会产生新数组。

map()

> 方法创建一个新数组，其结果是该数组中的每个元素是调用一次提供的函数后的返回值。

```
var originArr =  [1,2,3,4,5];
var newArr = originArr.map(function(current,index,array){
   return current * 2;
});
console.log(newArr);
//[2,4,6,8,10]

----------------------------------------------------
var kvArray = [{key: 1, value: 10},
               {key: 2, value: 20},
               {key: 3, value: 30}];

var reformattedArray = kvArray.map(function(obj) {
  var rObj = {};
  rObj[obj['key']] = obj.value;
  return rObj;
});
console.log(reformattedArray);
// [{1: 10}, {2: 20}, {3: 30}],
```

map 方法会给原数组中的每个元素都按顺序调用一次 callback 函数。callback 每次执行后的返回值（包括 undefined）组合起来形成一个新数组。 callback 函数只会在有值的索引上被调用；那些从来没被赋过值或者使用 delete 删除的索引则不会被调用。

callback 函数会被自动传入三个参数：数组元素，元素索引，原数组本身。

map 不修改调用它的原数组本身（当然可以在 callback 执行时改变原数组）

参数

```
var new_array = arr.map(function callback(currentValue[, index[, array]]) {
 // Return element for new_array
}[, thisArg])
```

callback
生成新数组元素的函数，使用三个参数：
  currentValue
  callback 数组中正在处理的当前元素。
  index可选
  callback 数组中正在处理的当前元素的索引。
  array可选
  map 方法调用的数组。

thisArg可选

执行 callback 函数时值被用作this。

返回值

一个由原数组每个元素执行回调函数的结果组成的新数组。

filter()

> 方法创建一个新数组, 其包含通过所提供函数实现的测试的所有元素。

```
var arr = [
  { id: 15 },
  { id: -1 },
  { id: 0 },
  { id: 3 },
  { id: 12.2 },
  { },
  { id: null },
  { id: NaN },
  { id: 'undefined' }
];

function isNumber(obj) {
  return obj !== undefined && typeof(obj) === 'number' && !isNaN(obj);
}

var arrByID = arr.filter(function(item) {
    return isNumber(item.id) && item.id !== 0;
});

console.log(arrByID);
// [{ id: 15 }, { id: -1 }, { id: 3 }, { id: 12.2 }]
```

filter 为数组中的每个元素调用一次 callback 函数，并利用所有使得 callback 返回 true 或等价于 true 的值的元素创建一个新数组。callback 只会在已经赋值的索引上被调用，对于那些已经被删除或者从未被赋值的索引不会被调用。那些没有通过 callback 测试的元素会被跳过，不会被包含在新数组中。

callback 被调用时传入三个参数：

元素的值
元素的索引
被遍历的数组本身
如果为 filter 提供一个 thisArg 参数，则它会被作为 callback 被调用时的 this 值。否则，callback 的 this 值在非严格模式下将是全局对象，严格模式下为 undefined。callback 函数最终观察到的 this 值是根据通常函数所看到的 "this"的规则确定的。

filter 不会改变原数组，它返回过滤后的新数组。

filter 遍历的元素范围在第一次调用 callback 之前就已经确定了。在调用 filter 之后被添加到数组中的元素不会被 filter 遍历到。如果已经存在的元素被改变了，则他们传入 callback 的值是 filter 遍历到它们那一刻的值。被删除或从来未被赋值的元素不会被遍历到。

参数

```
var newArray = arr.filter(callback(element[, index[, array]])[, thisArg])

callback
用来测试数组的每个元素的函数。返回 true 表示该元素通过测试，保留该元素，false 则不保留。它接受以下三个参数:
  element
  数组中当前正在处理的元素。
  index可选
  正在处理的元素在数组中的索引。
  array可选
  调用了 filter 的数组本身。
thisArg可选
执行 callback 时，用于 this 的值
```

返回值

一个新的、由通过测试的元素组成的数组，如果没有任何数组元素通过测试，则返回空数组。

reduce()

> 方法创建一个新数组, 其包含通过所提供函数实现的测试的所有元素。

```
var originArr = [3,4,5,6,7,8];
var sum = originArr.reduce(function(acc,current,idx,arr){
  return acc + current;
});
console.log(sum);
//33 数组中每项累加和

var str = 'jfkldsajgklasjkhlgjefaklhjaerkl';
var statistics = str.split('').reduce(function(acc,current){
  if(acc[current]){
    acc[current]++;
  }else {
    acc[current] = 1;
  }
  return acc;
},{});
```

```
console.log(statistics);
// {a: 4, d: 1, e: 2, f: 2, g: 2, h: 2, j: 5, k: 5, l: 5, r: 1, s: 2} 字符串统计后的对象结果
```

reduce为数组中的每一个元素依次执行callback函数，不包括数组中被删除或从未被赋值的元素，接受四个参数：

accumulator 累计器
currentValue 当前值
currentIndex 当前索引
array 数组
回调函数第一次执行时，accumulator 和currentValue的取值有两种情况：如果调用reduce()时提供了initialValue，accumulator取值为initialValue，currentValue取数组中的第一个值；如果没有提供 initialValue，那么accumulator取数组中的第一个值，currentValue取数组中的第二个值。

参数

```
var newArray = arr.filter(callback(element[, index[, array]])[, thisArg])
```

callback
执行数组中每个值 (如果没有提供 initialValue则第一个值除外)的函数，包含四个参数：
accumulator
累计器累计回调的返回值; 它是上一次调用回调时返回的累积值，或initialValue（见于下方）。
currentValue
数组中正在处理的元素。
index 可选
数组中正在处理的当前元素的索引。 如果提供了initialValue，则起始索引号为0，否则从索引1起始。
array可选
调用reduce()的数组

initialValue可选
作为第一次调用 callback函数时的第一个参数的值。 如果没有提供初始值，则将使用数组中的第一个元素。 在没有初始值的空数组上调用 reduce 将报错。

返回值

函数累计处理的结果

lastIndexOf()

> 方法返回指定元素（也即有效的 JavaScript 值或变量）在数组中的最后一个的索引，如果不存在则返回 -1。从数组的后面向前查找，从 `fromIndex` 处开始。

```
var array = [2, 5, 9, 2];
var index = array.lastIndexOf(2);
// index is 3

index = array.lastIndexOf(7);
```

```
// index is -1
index = array.lastIndexOf(2, 3);
// index is 3
index = array.lastIndexOf(2, 2);
// index is 0
index = array.lastIndexOf(2, -2);
// index is 0
index = array.lastIndexOf(2, -1);
// index is 3
```

lastIndexOf 使用严格相等（strict equality，即 ===）比较 searchElement 和数组中的元素。

参数

```
arr.lastIndexOf(searchElement[, fromIndex])
```

searchElement
被查找的元素。
fromIndex 可选
从此位置开始逆向查找。默认为数组的长度减 1(arr.length - 1)，即整个数组都被查找。如果该值大于或等于数组的长度，则整个数组会被查找。如果为负值，将其视为从数组末尾向前的偏移。即使该值为负，数组仍然会被从后向前查找。如果该值为负时，其绝对值大于数组长度，则方法返回 -1，即数组不会被查找。

返回值

数组中该元素最后一次出现的索引，如未找到返回-1。

reverse()

> 方法将数组中元素的位置颠倒，并返回该数组。数组的第一个元素会变成最后一个，数组的最后一个元素变成第一个。该方法会改变原数组。

```
var array1 = ['one', 'two', 'three'];
console.log( array1);
//  ["one", "two", "three"]

var reversed = array1.reverse();
console.log( reversed);
// ["three", "two", "one"]

console.log(array1);
//  ["three", "two", "one"]
```

reverse会对原数组造成修改影响

参数

```
arr.reverse()
```

返回值

颠倒后的数组。

splice()

> 方法通过删除或替换现有元素或者原地添加新的元素来修改数组,并以数组形式返回被修改的内容。此方法会改变原数组。

```
var months = ['Jan', 'March', 'April', 'June'];
months.splice(1, 0, 'Feb');
// 在下标1的位置新增了1项
console.log(months);
// ["Jan", "Feb", "March", "April", "June"]

months.splice(4, 1, 'May');
// 在下标4的位置 修改了 1项
console.log(months);
//["Jan", "Feb", "March", "April", "May"]

splice会修改原数组
```

参数

```
array.splice(start[, deleteCount[, item1[, item2[, ...]]]])

start
  指定修改的开始位置（从0计数）。如果超出了数组的长度，则从数组末尾开始添加内容；如果是负值，则表示从数组末位开始的第几位（从-1计数，这意味着-n是倒数第n个元素并且等价于array.length-n）；如果负数的绝对值大于数组的长度，则表示开始位置为第0位。

deleteCount 可选
  整数，表示要移除的数组元素的个数。
  如果 deleteCount 大于 start 之后的元素的总数，则从 start 后面的元素都将被删除（含第 start 位）。
  如果 deleteCount 被省略了，或者它的值大于等于array.length - start(也就是说，如果它大于或者等于start之后的所有元素的数量)，那么start之后数组的所有元素都会被删除。
 如果 deleteCount 是 0 或者负数，则不移除元素。这种情况下，至少应添加一个新元素。

item1, item2, ... 可选
 要添加进数组的元素,从start 位置开始。如果不指定，则 splice() 将只删除数组元素。
```

返回值

由被删除的元素组成的一个数组。如果只删除了一个元素，则返回只包含一个元素的数组。如果没有删除元素，则返回空数组。

includes()

> 方法用来判断一个数组是否包含一个指定的值，根据情况，如果包含则返回 true，否则返回false。

```
var array1 = [1, 2, 3];

console.log(array1.includes(2));
// expected output: true

var pets = ['cat', 'dog', 'bat'];

console.log(pets.includes('cat'));
// expected output: true

console.log(pets.includes('at'));
// expected output: false
```

参数

valueToFind
需要查找的元素值。

Note: 使用 includes()比较字符串和字符时是区分大小写。

fromIndex 可选
从fromIndex 索引处开始查找 valueToFind。如果为负值，则按升序从 array.length + fromIndex 的索引开始搜 （即使从末尾开始往前跳 fromIndex 的绝对值个索引，然后往后搜寻）。默认为 0。

返回值

返回一个布尔值 Boolean ， 如果在数组中找到了（如果传入了 fromIndex ，表示在 fromIndex 指定的索引范围中找到了）则返回 true 。

flat()

> 方法会按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并为一个新数组返回。

```
var arr1 = [1, 2, [3, 4]];
arr1.flat();
// [1, 2, 3, 4]

var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();
// [1, 2, 3, 4, [5, 6]]

var arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);
// [1, 2, 3, 4, 5, 6]

//使用 Infinity，可展开任意深度的嵌套数组
var arr4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
arr4.flat(Infinity);
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


//二维展开替代方案
var arr = [1, 2, [3, 4]];

var flatArr = arr.reduce(function(acc,curr){
   return acc.concat(curr);
},[]);

console.log(flatArr);
//[1, 2, 3, 4]
```

参数

```
var newArray = arr.flat([depth])

depth 可选
指定要提取嵌套数组的结构深度，默认值为 1。
```

返回值

```
一个包含将数组与子数组中所有元素的新数组。
```

方法汇总

```
toString()          把数组转换成字符串，逗号分隔每一项
valueOf()       返回数组对象值本身
// 1 栈操作(先进后出)
push()
pop()           //取出数组中的最后一项，修改length属性
// 2 队列操作(先进先出)
```

```
push()
shift()          //取出数组中的第一个元素，修改length属性
unshift()        //在数组最前面插入项，返回数组的长度
// 3 排序方法
reverse()        //翻转数组
sort();     //即使是数组sort也是根据字符，从小到大排序
// 带参数的sort是如何实现的?
// 4 操作方法
concat()         //把参数拼接到当前数组
slice()     //从当前数组中截取一个新的数组，不影响原来的数组，参数start从0开始,end从1开始
splice()    //删除或替换当前数组的某些项目，参数start, deleteCount, options(要替换的项目)
// 5 位置方法
indexOf()、lastIndexOf()   //如果没找到返回-1
// 6 迭代方法 不会修改原数组(可选)
every()、filter()、forEach()、map()、reduce()、some()
// 7 方法将数组的所有元素连接到一个字符串中。
join()
```

String方法

charCodeAt()

> 方法返回 `0` 到 `65535` 之间的整数，表示给定索引处的 UTF-16 代码单元

```
var str = '如果';
var u16 = str.charCodeAt(0);
console.log(u16); //22914

console.log(u16.toString(16)); //5982

UTF-16 编码单元匹配能用一个 UTF-16 编码单元表示的 Unicode 码点。如果 Unicode 码点不能用一个 UTF-16 编码单元表示（因为它的值大于0xFFFF），则所返回的编码单元会是这个码点代理对的第一个编码单元) 。如果你想要整个码点的值，使用 codePointAt()。
```

参数

```
str.charCodeAt(index)

index
一个大于等于 0，小于字符串长度的整数。如果不是一个数值，则默认为 0
```

返回值

指定 index 处字符的 UTF-16 代码单元值的一个数字；如果 index 超出范围，charCodeAt() 返回 NaN。

replace()

> 方法返回一个由替换值（replacement）替换部分或所有的模式（pattern）匹配项后的新字符串。模式可以是一个字符串或者一个正则表达式，替换值可以是一个字符串或者一个每次匹配都要调用的回调函数。如果pattern是字符串，则仅替换第一个匹配项。

```
var str = '如果你是春天，就请融化冰雪，让小草破土而出。如果你是夏天，就请用用各种各样的风，谱写快乐的乐章如果你是秋天，就让枫叶似火，洋溢丰收的喜悦；，如果你是冬天，就请做一位时装设计师，为大地披上雪白霓裳。';

var regex = /\u5982\u679c/gi;

console.log(str.replace(regex, '假如'));


console.log(str.replace('如果', '假如'));


/* 汉子转换 unicode编码方法 */
function toUnicode(str){
   return escape(str).toLocaleLowerCase().replace(/%u/gi,'\\u');
}

function string2unicode(str){
      var ret ="";
      for(var i=0; i<str.length; i++){
     ret += "\\u" + str.charCodeAt(i).toString(16);
    }
   return ret;
}
```

参数

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

regexp (pattern)
一个RegExp 对象或者其字面量。该正则所匹配的内容会被第二个参数的返回值替换掉。
substr (pattern)
一个将被 newSubStr 替换的 字符串。其被视为一整个字符串，而不是一个正则表达式。仅第一个匹配项会被替换。
newSubStr (replacement)
用于替换掉第一个参数在原字符串中的匹配部分的字符串。该字符串中可以内插一些特殊的变量名。参考下面的使用字符串作为参数。
function (replacement)
一个用来创建新子字符串的函数，该函数的返回值将替换掉第一个参数匹配到的结果。参考下面的指定一个函数作为参数。

返回值

一个部分或全部匹配由替代模式所取代的新的字符串。

slice()

> 方法提取某个字符串的一部分，并返回一个新的字符串，且不会改动原字符串。

```
const str = 'The quick brown fox jumps over the lazy dog.';

console.log(str.slice(31));
// "the lazy dog."

console.log(str.slice(4, 19));
// "quick brown fox"

console.log(str.slice(-4));
//  "dog."

console.log(str.slice(-9, -5));
// "lazy"
```

参数

```
str.slice(beginIndex[, endIndex])
```

beginIndex
从该索引（以 0 为基数）处开始提取原字符串中的字符。如果值为负数，会被当做 strLength + beginIndex 看待，这里的 strLength 是字符串的长度（例如， 如果 beginIndex 是 -3 则看作是：strLength - 3）

endIndex
可选。在该索引（以 0 为基数）处结束提取字符串。如果省略该参数，slice() 会一直提取到字符串末尾。如果该参数为负数，则被看作是 strLength + endIndex，这里的 strLength 就是字符串的长度(例如，如果 endIndex 是 -3，则是，strLength - 3)。

返回值

返回一个从原字符串中提取出来的新字符串

方法汇总

```
// 1 字符方法
charAt()        //获取指定位置处字符
charCodeAt()        //获取指定位置处字符的ASCII码
str[0]          //HTML5，IE8+支持 和charAt()等效
// 2 字符串操作方法
concat()            //拼接字符串，等效于+，+更常用
slice()         //从start位置开始，截取到end位置，end取不到
substring()     //从start位置开始，截取到end位置，end取不到
substr()            //从start位置开始，截取length个字符
// 3 位置方法
indexOf()       //返回指定内容在元字符串中的位置
lastIndexOf()       //从后往前找，只找第一个匹配的
// 4 去除空白
trim()          //只能去除字符串前后的空白
// 5 大小写转换方法
to(Locale)UpperCase()       //转换大写
to(Locale)LowerCase()       //转换小写
// 6 其它
search()
replace()
split()
fromCharCode() //unicode转字符串
// String.fromCharCode(101, 102, 103);
```

案例

- 截取字符串"我爱中华人民共和国"，中的"中华"

```
var s = "我爱中华人民共和国";
s = s.substr(2,2);
console.log(s);

----------------------

function padLeft(num){
   num = num || 0;
   return ('00'+num).substr(-2);
}
```

- "abcoefoxyozzopp"查找字符串中所有o出现的位置

```
var s = 'abcoefoxyozzopp';
var array = [];
do {
  var index = s.indexOf('o', index + 1);
  if (index != -1) {
    array.push(index);
  }
} while (index > -1);
console.log(array);
```

- 把字符串中所有的o替换成!

```
var s = 'abcoefoxyozzopp';
do {
  s = s.replace('o', '');
} while (s.indexOf('o') > -1);
console.log(s);

console.log(s.replace(/o/ig, ''));
```

- 判断一个字符串中出现次数最多的字符，统计这个次数

```
var s = 'abcoefoxyozzopp';
var o = {};

for (var i = 0; i < s.length; i++) {
  var item = s.charAt(i);
  if (o[item]) {
    o[item] ++;
  }else{
    o[item] = 1;
  }
}

var max = 0;
var char ;
for(var key in o) {
```

```
  if (max < o[key]) {
    max = o[key];
    char = key;
  }
}

console.log(max);
console.log(char);
```

Math对象方法

> Math对象不是构造函数，它具有数学常数和函数的属性和方法，都是以静态成员的方式提供
>
> 跟数学相关的运算来找Math中的成员（求绝对值，取整）

Math静态属性

```
Math.E  //属性表示自然对数的底数（或称为基数），e，约等于 2.718。
Math.LN10 //属性表示 10 的自然对数，约为 2.302：
Math.LN2 // 属性表示 2 的自然对数，约为 0.693：
Math.LOG10E //属性表示以 10 为底数，e 的对数，约为 0.434：
Math.LOG2E //属性表示以 2 为底数，e 的对数，约为 1.442：
Math.PI //表示一个圆的周长与直径的比例，约为 3.14159：
Math.SQRT1_2 // 属性表示 1/2 的平方根，约为 0.707：
Math.SQRT2 //属性表示 2 的平方根，约为 1.414：
```

Math.abs()

> 函数返回指定数字 “x“ 的绝对值。如下：

{\mathtt{\operatorname{Math.abs}(x)}} = {|x|} = \begin{cases} x & \text{if} \quad x \geq 0 \\ -x & \text{if} \qu

```
Math.abs(x);  //x为数值参数
//传入一个非数字形式的字符串或者 undefined/empty 变量，将返回 NaN。传入 null 将返回 0。


Math.abs('-1');     // 1
Math.abs(-2);       // 2
Math.abs(null);     // 0
Math.abs("string"); // NaN
Math.abs();         // NaN
```

Math.cos()

> 函数返回一个数值的余弦值。

x
一个以弧度为单位的数值。

cos 方法返回一个 -1 到 1 之间的数值，表示角度（单位：弧度）的余弦值。

由于 cos 是 Math 的静态方法，所以应该像这样使用：Math.cos()，而不是作为你创建的 Math 实例的方法。

```
Math.cos(0);       // 1
Math.cos(1);       // 0.5403023058681398

Math.cos(Math.PI);    // -1
Math.cos(2 * Math.PI); // 1
```

Math.sin()

> 函数返回一个数值的正弦值。

x
一个数值（以弧度为单位）。
sin 方法返回一个 -1 到 1 之间的数值，表示给定角度（单位：弧度）的正弦值。

由于 sin 是 Math 的静态方法，所以应该像这样使用：Math.sin()，而不是作为你创建的 Math 实例的方法。

```
Math.sin(0);       // 0
Math.sin(1);       // 0.8414709848078965

Math.sin(Math.PI / 2); // 1
```

Math.tan()

> 方法返回一个数值的正切值。

```
Math.tan(x);
x
一个数值，表示一个角（单位：弧度）。
tan 方法返回一个数值，表示一个角的正切值。

由于 tan 是 Math 的静态方法，所以应该像这样使用 Math.tan()，而不是作为你创建的 Math 实例的方法。

Math.tan(1);        //1.5574077246549023
```

Math.max()

> 函数返回一组数中的最大值。

```
console.log(Math.max(1, 3, 2));
// expected output: 3

console.log(Math.max(-1, -3, -2));
// expected output: -1


Math.max(value1[,value2, ...])

参数
value1, value2, ...
一组数值
```

返回值

返回给定的一组数字中的最大值。如果给定的参数中至少有一个参数无法被转换成数字，则会返回 NaN。

Math.min()

> 函数返回一组数中的最大值。

```
console.log(Math.min(1, 3, 2));
// expected output: 1

console.log(Math.min(-1, -3, -2));
// expected output: -3


Math.min([value1[,value2, ...]])

参数
value1, value2, ...
一组数值
```

返回值

给定数值中最小的数。如果任一参数不能转换为数值，则返回NaN。

Math.pow()

> 函数返回基数（`base`）的指数（`exponent`）次幂，即 `baseexponent`。

```
console.log(Math.pow(7, 3));
// expected output: 343


Math.pow(base, exponent)

base
基数
exponent
指数
```

返回值

返回base的exponent次幂结果

Math.random()

> 函数返回一个浮点，伪随机数在范围从**0到**小于**1**，也就是说，从0（包括0）往上，但是不包括1（排除1），然后您可以缩放到所需的范围。实现将初始种子选择到随机数生成算法;它不能被用户选择或重置。

```
function getRandomInt(max) {
  return Math.floor(Math.random() * Math.floor(max));
}

console.log(getRandomInt(3));
// expected output: 0, 1 or 2

console.log(getRandomInt(1));
// expected output: 0

console.log(Math.random());
// expected output: a number between 0 and 1


Math.random()
```

返回值

一个浮点型伪随机数字，在0（包括0）和1（不包括）之间。

Math.round()

> 函数返回一个数字四舍五入后最接近的整数

```
Math.round(20.49) //20
Math.round(20.5) //21
Math.round(-20.5) //-20
Math.round(-20.51) //-21
```

返回值

四舍五入后的值

Math.floor()

> 返回小于或等于一个给定数字的最大整数 : 向下取整

```
Math.floor(x);

Math.floor( 45.95);
// 45
Math.floor( 45.05);
// 45
Math.floor( 4 );
// 4
Math.floor(-45.05);
// -46
Math.floor(-45.95);
// -46


/* 与 ~~ 去除小数位有区别 */
~~(-45.95) //-45
```

返回值

一个表示小于或等于指定数字的最大整数的数字。

Math.ceil()

> 函数返回大于或等于一个给定数字的最小整数

```
Math.ceil(x);
console.log(Math.ceil(.95));
//  1

console.log(Math.ceil(4));
//  4

console.log(Math.ceil(7.004));
//  8

console.log(Math.ceil(-7.004));
//  -7
```

返回值

大于或等于给定数字的最小整数。

方法汇总

```
Math.PI                         // 圆周率
Math.random()                   // 生成随机数
Math.floor()/Math.ceil()      // 向下取整/向上取整
Math.round()                    // 取整，四舍五入
Math.abs()                      // 绝对值
Math.max()/Math.min()           // 求最大和最小值

Math.sin()/Math.cos()         // 正弦/余弦
Math.power()/Math.sqrt()    // 求指数次幂/求平方根
```

Date对象方法

> 创建 `Date` 实例用来处理日期和时间。Date 对象基于1970年1月1日（世界标准时间）起的毫秒数。

```
// 获取当前时间，UTC世界时间，距1970年1月1日（世界标准时间）起的毫秒数
var now = new Date();
console.log(now.valueOf()); // 获取距1970年1月1日（世界标准时间）起的毫秒数

Date构造函数的参数
1. 毫秒数 1498099000356    new Date(1498099000356)
2. 日期格式字符串  '2015-5-1'   new Date('2015-5-1')
3. 年、月、日.....          new Date(2015, 4, 1)   // 月份从0开始
```

- 获取日期的毫秒形式

```
var now = new Date();
// valueOf用于获取对象的原始值
console.log(date.valueOf())

// HTML5中提供的方法，有兼容性问题
var now = Date.now();

// 不支持HTML5的浏览器，可以用下面这种方式
var now = + new Date();    // 调用 Date对象的valueOf()
```

- 日期格式化方法

```
toString()   // 转换成字符串
valueOf()  // 获取毫秒值
// 下面格式化日期的方法，在不同浏览器可能表现不一致，一般不用
toDateString()
toTimeString()
```

- 获取日期指定部分

```
getTime()   // 返回毫秒数和valueOf()结果一样，valueOf()内部调用的getTime()
getMilliseconds()
getSeconds()  // 返回0-59
getMinutes()  // 返回0-59
getHours()   // 返回0-23
getDay()    // 返回星期几 0周日  6周6
getDate()   // 返回当前月的第几天
getMonth()   // 返回月份，***从0开始***
getFullYear() //返回4位的年份  如 2016
```

案例

- 写一个函数，格式化日期对象，返回yyyy-MM-dd HH:mm:ss的形式

```
function formatDate(d) {
   //如果date不是日期对象，返回
   if (!date instanceof Date) {
      return;
   }
   var year = d.getFullYear(),
      month = d.getMonth() + 1,
      date = d.getDate(),
      hour = d.getHours(),
      minute = d.getMinutes(),
      second = d.getSeconds(),
      week = ['日', '一', '二', '三', '四', '五', '六'][d.getDay()]; // 1234560 => 一 二 三 四 五 六 七
   return year + '年' + padLeft(month) + '月' + padLeft(date) + '日 ' + padLeft(hour) + ':' + padLeft(minute) + ':' +
padLeft(second) + ' 星期' + week;
}

function padLeft(num){
   return String(num)[1] && String(num) || '0' + num;
}
```

- 计算时间差，返回相差的天/时/分/秒

```
function getInterval(start, end) {
  var day, hour, minute, second, interval;
  interval = end - start;
  interval /= 1000;
  day = ~~(interval / 60 /60 / 24);
  hour = ~~(interval / 60 /60 % 24);
```

```
  minute = ~~(interval / 60 % 60);
  second = ~~(interval % 60);
  return {
    day: day,
    hour: hour,
    minute: minute,
    second: second
  }
}
```