JavaScript

初识 JavaScript

JavaScript 是什么

JavaScript (简称 JS)

- 是世界上最流行的编程语言之一
- 是一个脚本语言, 通过解释器运行
- 主要在客户端(浏览器)上运行, 现在也可以基于 node.js 在服务器端运行.

JavaScript 最初只是为了完成简单的表单验证(验证数据合法性), 结果后来不小心就火了.

当前 JavaScript 已经成为了一个通用的编程语言

JavaScript 的能做的事情:

- 网页开发(更复杂的特效和用户交互)
- 网页游戏开发
- 服务器开发(node.js)
- 桌面程序开发(Electron, VSCode 就是这么来的)
- 手机 app 开发

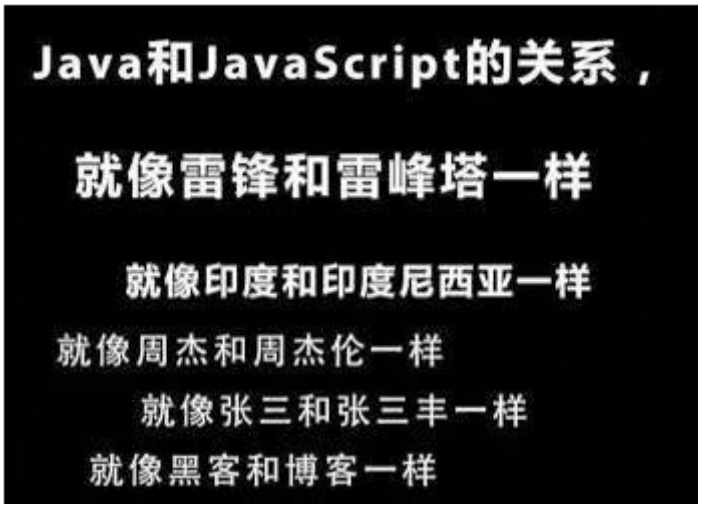
发展历史

JavaScript 之父 布兰登 * 艾奇 (Brendan Eich)

曾经的布兰登

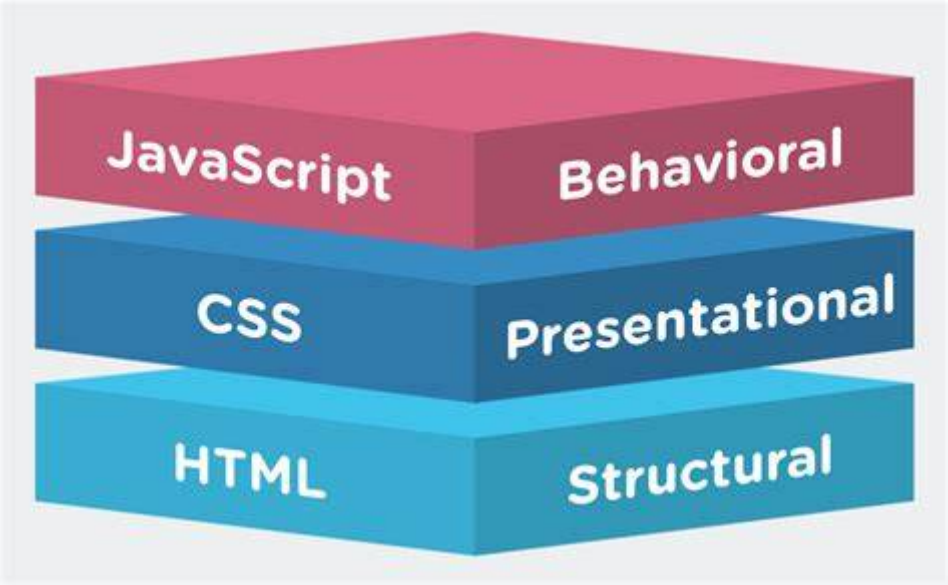
发明 JavaScript 之后的布兰登

> 1995 年, 用 10 天时间完成 JS 的设计 (由于设计时间太短，语言的一些细节考虑得不够严谨，导致后来很长一段时间，Javascript 写出来的程序混乱不堪)
>
> 最初在网景公司, 命名为 LiveScript,
>
> 一般认为，当时 Netscape 之所以将 LiveScript 命名为 JavaScript，是因为 Java 是当时最流行的编程语言，带有 "Java" 的名字有助于这门新生语言的传播。
>
> 其实 Java 和 JavaScript 之间的语法风格相去甚远.



> JavaScript 发展历史可以参考阮一峰大神的博客
>
> http://www.ruanyifeng.com/blog/2011/06/birth_of_javascript.html

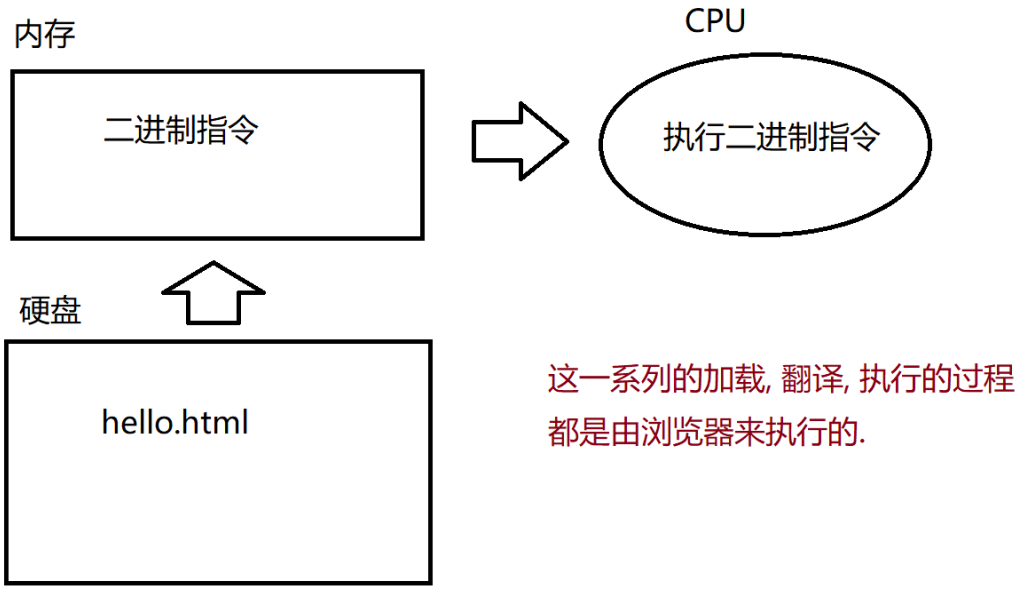
JavaScript 和 HTML 和 CSS 之间的关系



- HTML: 网页的结构(骨)
- CSS: 网页的表现(皮)
- JavaScript: 网页的行为(魂)

JavaScript 运行过程

- 编写的代码是保存在文件中的, 也就是存储在硬盘(外存上).
- 双击 .html 文件浏览器(应用程序)就会读取文件, 把文件内容加载到内存中(数据流向: 硬盘 => 内存)
- 浏览器会解析用户编写的代码, 把代码翻译成二进制的, 能让计算机识别的指令(解释器的工作)
- 得到的二进制指令会被 CPU 加载并执行(数据流向: 内存 => CPU)



浏览器分成渲染引擎 + JS 引擎.

- 渲染引擎: 解析 html + CSS, 俗称 "内核"
- JS 引擎: 也就是 JS 解释器. 典型的就是 Chrome 中内置的 V8

> JS 引擎逐行读取 JS 代码内容, 然后解析成二进制指令, 再执行.

JavaScript 的组成

- ECMAScript(简称 ES): JavaScript 语法
- DOM: 页面文档对象模型, 对页面中的元素进行操作
- BOM: 浏览器对象模型, 对浏览器窗口进行操作

光有 JS 语法, 只能写一些基础的逻辑流程.

但是要想完成更复杂的任务, 完成和浏览器以及页面的交互, 那么久需要 DOM API 和 BOM API.

> 这主要指在浏览器端运行的 JS. 如果是运行在服务端的 JS , 则需要使用 node.js 的 API,
>
> 就不太需要关注 DOM 和 BOM

重要概念: ECMAScript

这是一套 "标准", 无论是啥样的 JS 引擎都要遵守这个标准来实现.

> 啥叫标准? 车同轨, 书同文. 秦始皇最大的贡献之一, 就是制定了一套标准.
>
> 三流公司做产品, 一流公司做标准.

前置知识

第一个程序

```
<script>
    alert("你好!");
</script>
```

> - JavaScript 代码可以嵌入到 HTML 的 script 标签中.

JavaScript 的书写形式

1. 行内式

直接嵌入到 html 元素内部

```
<input type="button" value="点我一下" onclick="alert('haha')">
```

> 注意, JS 中字符串常量可以使用单引号表示, 也可以 使用双引号表示.
>
> HTML 中推荐使用双引号, JS 中推荐使用单引号.

2. 内嵌式

写到 script 标签中

```
<script>
    alert("haha");
</script>
```

3. 外部式

写到单独的 .js 文件中

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

```
alert("hehe");
```

> 注意, 这种情况下 script 标签中间不能写代码. 必须空着(写了代码也不会执行).
>
> 适合代码多的情况.

注释

单行注释 // [建议使用]

多行注释 /* */

```
// 我是单行注释
/*
    我是多行注释
    我是多行注释
    我是多行注释
*/
```

> 使用 ctrl + / 切换注释.
>
> 多行注释不能嵌套. 形如这种代码就会报错
>
> ```
> /*
> /*
>    我是多行注释
>    我是多行注释
>    我是多行注释
> */
> */
> ```

输入输出

输入: prompt

弹出一个输入框

```
// 弹出一个输入框
prompt("请输入您的姓名:");
```

输出: alert

弹出一个警示对话框, 输出结果

```
// 弹出一个输出框
alert("hello");
```

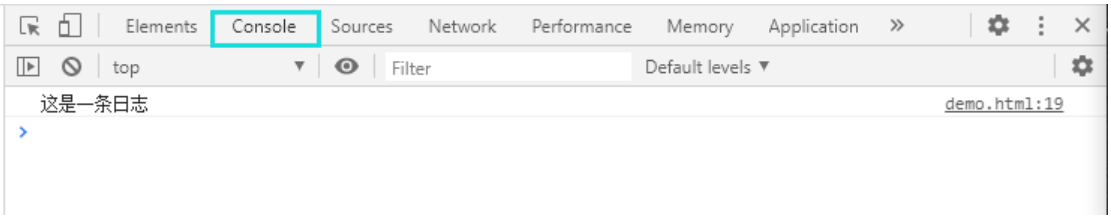
输出: console.log

在控制台打印一个日志(供程序员看)

```
// 向控制台输出日志
console.log("这是一条日志");
```

注意: 在 VSCode 中直接输入 "log" 再按 tab 键, 就可以快速输入 `console.log`

> 需要打开浏览器的开发者工具(F12) => Console 标签页 才能看到结果.



这样的输出一般不是给普通用户看的, 而是程序员来看的.

重要概念: 日志

日志是程序员调试程序的重要手段

> 去医院看病, 医生会让患者做各种检查, 血常规, 尿常规, B超, CT等... 此时得到一堆检测结果. 这些结果普通人看不懂, 但是医生能看懂, 并且医生要通过这些信息来诊断病情.
>
> 这些检测结果就是医生的 "日志"
>
> PS: 相比于医生来说, 程序猿多一个终极大招, "重启下试试".

重要概念: .

console 是一个 js 中的 "对象"

. 表示取对象中的某个属性或者方法. 可以直观理解成 "的"

console.log 就可以理解成: 使用 "控制台" 对象 "的" log 方法.

语法概览

JavaScript 虽然一些设计理念和 Java 相去甚远, 但是在基础语法层面上还是有一些相似之处的.

有了 Java 的基础很容易理解 JavaScript 的一些基本语法.

变量的使用

基本用法

创建变量(变量定义/变量声明/变量初始化)

```
var name = 'zhangsan';
var age = 20;
```

> var 是 JS 中的关键字, 表示这是一个变量.
>
> = 在 JS 中表示 "赋值", 相当于把数据放到内存的盒子中. = 两侧建议有一个空格
>
> 每个语句最后带有一个 ; 结尾. JS 中可以省略 ; 但是建议还是加上.
>
> 注意, 此处的 ; 为英文分号. JS 中所有的标点都是英文标点.
>
> 初始化的值如果是字符串, 那么就要使用单引号或者双引号引起来.

> 初始化的值如果是数字, 那么直接赋值即可.

使用变量

```
console.log(age); // 读取变量内容
age = 30;         // 修改变量内容
```

> 为啥动漫中的角色都是要先喊出技能名字再真正释放技能?
>
> 就是因为变量要先声明才能使用.

代码示例: 弹框提示用户输入信息, 再弹框显示.

```
var name = prompt("请输入姓名:");
var age = prompt("请输入年龄:");
var score = prompt("请输入分数");
alert("您的姓名是: " + name);
alert("您的年龄是: " + age);
alert("您的分数是: " + score);
```

也可以把三个输出内容合并成一次弹框

```
var name = prompt("请输入姓名:");
var age = prompt("请输入年龄:");
var score = prompt("请输入分数");
alert("您的姓名是: " + name + "\n" + "您的年龄是: " + age + "\n" + "您的分数是: " +
score + "\n");
```

> - `+` 表示字符串拼接, 也就是把两个字符串首尾相接变成一个字符串.
> - `\n` 表示换行

> JavaScript 中还支持使用 let 定义变量. 用法和 var 基本类似. 用法上的差异此处暂时不讨论.

理解 动态类型

1) JS 的变量类型是程序运行过程中才确定的(运行到 = 语句才会确定类型)

```
var a = 10;     // 数字
var b = "hehe"; // 字符串
```

2) 随着程序运行, 变量的类型可能会发生改变.

```
var a = 10;    // 数字
a = "hehe";    // 字符串
```

这一点和 C Java 这种静态类型语言差异较大.

> C, C++, Java, Go 等语言是静态类型语言. 一个变量在创建的时候类型就确定了, 不能在运行时发生改变.
>
> 如果尝试改变, 就会直接编译报错.

基本数据类型

JS 中内置的几种类型

- number: 数字. 不区分整数和小数.
- boolean: true 真, false 假.
- string: 字符串类型.
- undefined: 只有唯一的值 undefined. 表示未定义的值.
- null: 只有唯一的值 null. 表示空值.

number 数字类型

JS 中不区分整数和浮点数, 统一都使用 "数字类型" 来表示.

数字进制表示

计算机中都是使用二进制来表示数据, 而人平时都是使用十进制.

因为二进制在使用过程中不太方便(01太多会看花眼).

所以在日常使用二进制数字时往往使用 八进制 和 十六进制 来表示二进制数字.

```
var a = 07;      // 八进制整数, 以 0 开头
var b = 0xa;     // 十六进制整数, 以 0x 开头
var c = 0b10;    // 二进制整数, 以 0b 开头
```

注意:

- 一个八进制数字对应三个二进制数字
- 一个十六进制数字对应四个二进制数字. (两个十六进制数字就是一个字节)

> 各种进制之间的转换, 不需要手工计算, 直接使用计算器即可.

特殊的数字值

- Infinity: 无穷大, 大于任何数字. 表示数字已经超过了 JS 能表示的范围.
- -Infinity: 负无穷大, 小于任何数字. 表示数字已经超过了 JS 能表示的范围.
- NaN: 表示当前的结果不是一个数字.

```
var max = Number.MAX_VALUE;
// 得到 Infinity
console.log(max * 2);
// 得到 -Infinity
console.log(-max * 2);
// 得到 NaN
console.log('hehe' - 10);
```

> 注意:
>
> 1. 负无穷大 和 无穷小 不是一回事. 无穷小指无限趋近与 0, 值为 `1 / Infinity`
> 2. 'hehe' + 10 得到的不是 NaN, 而是 'hehe10', 会把数字隐式转成字符串, 再进行字符串拼接.
> 3. 可以使用 isNaN 函数判定是不是一个非数字.
>
> ```
> console.log(isNaN(10));  // false
> console.log(isNaN('hehe' - 10));  // true
> ```

string 字符串类型

基本规则

字符串字面值需要使用引号引起来, 单引号双引号均可.

```
var a = "haha";
var b = 'hehe';
var c = hehe;    // 运行出错
```

如果字符串中本来已经包含引号咋办?

```
var msg = "My name is "zhangsan"";    // 出错
var msg = "My name is \"zhangsan\"";  // 正确, 使用转义字符. \" 来表示字符串内部的引号.
var msg = "My name is 'zhangsan'";    // 正确, 搭配使用单双引号
var msg = 'My name is "zhangsan"';    // 正确, 搭配使用单双引号
```

转义字符

有些字符不方便直接输入, 于是要通过一些特殊方式来表示.

- `\n`
- `\\`
- `\'`
- `\"`
- `\t`

求长度

使用 String 的 length 属性即可

```
var a = 'hehe';
console.log(a.length);

var b = '哈哈';
console.log(b.length);
```

结果:

```
4
2
```

单位为字符的数量

字符串拼接

使用 + 进行拼接

```
var a = "my name is ";
var b = "zhangsan";
console.log(a + b);
```

注意, 数字和字符串也可以进行拼接

```
var c = "my score is ";
var d = 100;
console.log(c + d);
```

注意, 要认准相加的变量到底是字符串还是数字

```
console.log(100 + 100);     // 200
console.log('100' + 100);   // 100100
```

boolean 布尔类型

表示 "真" 和 "假"

> boolean 原本是数学中的概念 (布尔代数).
>
> 在计算机中 boolean 意义重大, 往往要搭配条件/循环完成逻辑判断.

Boolean 参与运算时当做 1 和 0 来看待.

```
console.log(true + 1);
console.log(false + 1)
```

> 这样的操作其实是不科学的. 实际开发中不应该这么写.

undefined 未定义数据类型

如果一个变量没有被初始化过, 结果就是 undefined, 是 undefined 类型

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

undefined 和字符串进行相加, 结果进行字符串拼接

```
console.log(a + "10");  // undefined10
```

undefined 和数字进行相加, 结果为 NaN

```
console.log(a + 10);
```

null 空值类型

null 表示当前的变量是一个 "空值".

```
var b = null;
console.log(b + 10);    // 10
console.log(b + "10");  // null10
```

> **注意:**
>
> null 和 undefined 都表示取值非法的情况, 但是侧重点不同.
>
> null 表示当前的值为空. (相当于有一个空的盒子)
>
> undefined 表示当前的变量未定义. (相当于连盒子都没有)

运算符

JavaScript 中的运算符和 Java 用法基本相同. 此处不做详细介绍了.

算术运算符

- +
- -
- *
- /
- %

赋值运算符 & 复合赋值运算符

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

自增自减运算符

- ++: 自增1
- --: 自减1

比较运算符

- `<`
- `>`
- `<=`
- `>=`
- `==` 比较相等(会进行隐式类型转换)
- `!=`
- `===` 比较相等(不会进行隐式类型转换)
- `!==`

逻辑运算符

用于计算多个 boolean 表达式的值.

- && 与: 一假则假
- || 或: 一真则真
- ！ 非

位运算

- & 按位与
- | 按位或
- ~ 按位取反
- ^ 按位异或

移位运算

- << 左移
- `>>` 有符号右移(算术右移)
- `>>>` 无符号右移(逻辑右移)

条件语句

if 语句

基本语法格式

条件表达式为 true, 则执行 if 的 { } 中的代码

```
// 形式1
if (条件) {
    语句
}

// 形式2
if (条件) {
    语句1
} else {
    语句2
}

// 形式3
if (条件1) {
    语句1
} else if (条件2) {
    语句2
} else if .... {
    语句...
} else {
    语句N
}
```

练习案例

代码示例1: 判定一个数字是奇数还是偶数

```
var num = 10;
if (num % 2 == 0) {
    console.log("num 是偶数");
} else {
    console.log("num 是奇数");
}
```

> 注意! 不能写成 num % 2 == 1 就是奇数. 负的奇数 % 2 结果可能是 -1.

代码示例2: 判定一个数字是正数还是负数

```
var num = 10;
if (num > 0) {
    console.log("num 是正数");
} else if (num < 0) {
    console.log("num 是负数");
} else {
    console.log("num 是 0");
}
```

代码示例3: 判定某一年份是否是闰年

```
var year = 2000;
```

```
if (year % 100 == 0) {
    // 判定世纪闰年
    if (year % 400 == 0) {
        console.log("是闰年");
    } else {
        console.log("不是闰年");
    }
} else {
    // 普通闰年
    if (year % 4 == 0) {
        console.log("是闰年");
    } else {
        console.log("不是闰年");
    }
}
```

三元表达式

是 if else 的简化写法.

```
条件 ? 表达式1 : 表达式2
```

条件为真, 返回表达式1 的值. 条件为假, 返回表达式2 的值.

> 注意, 三元表达式的优先级是比较低的.

switch

更适合多分支的场景.

```
switch (表达式) {
    case 值1:
        语句1;
        break;
    case 值2:
        语句2:
        break;
    default:
        语句N;
}
```

用户输入一个整数, 提示今天是星期几

```
var day = prompt("请输入今天星期几: ");
switch (parseInt(day)) {
    case 1:
        console.log("星期一");
        break;
    case 2:
        console.log("星期二");
```

```
        break;
    case 3:
        console.log("星期三");
        break;
    case 4:
        console.log("星期四");
        break;
    case 5:
        console.log("星期五");
        break;
    case 6:
        console.log("星期六");
        break;
    case 7:
        console.log("星期日");
        break;
    default:
        console.log("输入有误");
}
```

循环语句

重复执行某些语句

while 循环

```
while (条件) {
    循环体;
}
```

执行过程:

- 先执行条件语句
- 条件为 true, 执行循环体代码.
- 条件为 false, 直接结束循环

代码示例1: 打印 1 - 10

```
var num = 1;
while (num <= 10) {
    console.log(num);
    num++;
}
```

代码示例2: 计算 5 的阶乘

```
var result = 1;
var i = 1;
while (i <= 5) {
    result *= i;
    i++;
}
console.log(result)
```

continue

结束这次循环

吃五个李子，发现第三个李子里有一只虫子, 于是扔掉这个, 继续吃下一个李子.

```
var i = 1;
while (i <= 5) {
    if (i == 3) {
        i++;
        continue;
    }
    console.log("我在吃第" + i + "个李子");
    i++;
}
```

```
我在吃第1个李子
我在吃第2个李子
我在吃第4个李子
我在吃第5个李子
```

代码示例: 找到 100 - 200 中所有 3 的倍数

```
var num = 100;
while (num <= 200) {
    if (num % 3 != 0) {
        num++;  // 这里的 ++ 不要忘记! 否则会死循环.
        continue;
    }
    console.log("找到了 3 的倍数, 为:" + num);
    num++;
}
```

break

结束整个循环

吃五个李子，发现第三个李子里有半个虫子, 于是剩下的也不吃了.

```
var i = 1;
while (i <= 5) {
    if (i == 3) {
        break;
    }
    console.log("我在吃第" + i + "个李子");
    i++;
}
```

```
我在吃第1个李子
我在吃第2个李子
```

代码示例: 找到 100 - 200 中第一个 3 的倍数

```
var num = 100;
while (num <= 200) {
    if (num % 3 == 0) {
        console.log("找到了 3 的倍数, 为:" + num);
        break;
    }
    num++;
}

// 执行结果
找到了 3 的倍数, 为:102
```

for 循环

```
for (表达式1; 表达式2; 表达式3) {
    循环体
}
```

- 表达式1: 用于初始化循环变量.
- 表达式2: 循环条件
- 表达式3: 更新循环变量.

执行过程:

- 先执行表达式1, 初始化循环变量
- 再执行表达式2, 判定循环条件
- 如果条件为 false, 结束循环
- 如果条件为 true, 则执行循环体代码.
- 执行表达式3 更新循环变量

代码示例1: 打印 1 - 10 的数字

```
for (var num = 1; num <= 10; num++) {
    console.log(num);
}
```

代码示例2: 计算 5 的阶乘

```
var result = 0;
for (var i = 1; i <= 5; i++) {
    result *= i;
}
console.log("result = " + result);
```

数组

创建数组

使用 new 关键字创建

```
// Array 的 A 要大写
var arr = new Array();
```

使用字面量方式创建 [常用]

```
var arr = [];
var arr2 = [1, 2, 'haha', false]; // 数组中保存的内容称为 "元素"
```

注意: JS 的数组不要求元素是相同类型.

> 这一点和 C, C++, Java 等静态类型的语言差别很大. 但是 Python, PHP 等动态类型语言也是如此.

获取数组元素

使用下标的方式访问数组元素(从 0 开始)

```
var arr = ['小猪佩奇', '小猪乔治', '小羊苏西'];
console.log(arr);
console.log(arr[0]);
console.log(arr[1]);
console.log(arr[2]);
arr[2] = '小猫凯迪';
console.log(arr);
```

如果下标超出范围读取元素, 则结果为 undefined

```
console.log(arr[3]);   // undefined
console.log(arr[-1]);  // undefined
```

注意: 不要给数组名直接赋值, 此时数组中的所有元素都没了.

> 相当于本来 arr 是一个数组, 重新赋值后变成字符串了.

```
var arr = ['小猪佩奇', '小猪乔治', '小羊苏西'];
arr = '小猫凯迪';
```

新增数组元素

1. 通过修改 length 新增

相当于在末尾新增元素. 新增的元素默认值为 undefined

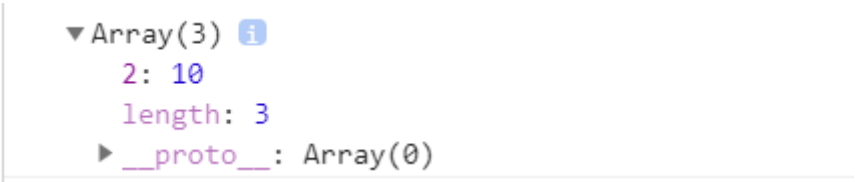
```
var arr = [9, 5, 2, 7];
arr.length = 6;
console.log(arr);
console.log(arr[4], arr[5]);
```

```
▶ (6) [9, 5, 2, 7, empty × 2]
undefined undefined
```

2. 通过下标新增

如果下标超出范围赋值元素, 则会给指定位置插入新元素

```
var arr = [];
arr[2] = 10;
console.log(arr)
```



> 此时这个数组的 [0] 和 [1] 都是 undefined

3. 使用 push 进行追加元素

代码示例: 给定一个数组, 把数组中的奇数放到一个 newArr 中.

```
var arr = [9, 5, 2, 7, 3, 6, 8];
var newArr = [];
for (var i = 0; i < arr.length; i++) {
    if (arr[i] % 2 != 0) {
        newArr.push(arr[i]);
    }
}
console.log(newArr);
```

删除数组中的元素

使用 splice 方法删除元素

```
var arr = [9, 5, 2, 7];
// 第一个参数表示从下表为 2 的位置开始删除. 第二个参数表示要删除的元素个数是 1 个
arr.splice(2, 1);
console.log(arr);

// 结果
[9, 5, 7]
```

> 目前咱们已经用到了数组中的一些属性和方法.
>
> arr.length, length 使用的时候不带括号, 此时 length 就是一个普通的变量(称为成员变量, 也叫属性)
>
> arr.push(), arr.splice() 使用的时候带括号, 并且可以传参数, 此时是一个函数 (也叫做方法)

函数

语法格式

```
// 创建函数/函数声明/函数定义
function 函数名(形参列表) {
    函数体
    return 返回值;
}

// 函数调用
函数名(实参列表)              // 不考虑返回值
返回值 = 函数名(实参列表)     // 考虑返回值
```

- 函数定义并不会执行函数体内容, 必须要调用才会执行. 调用几次就会执行几次.

```
function hello() {
    console.log("hello");
}
// 如果不调用函数, 则没有执行打印语句
hello();
```

- 调用函数的时候进入函数内部执行, 函数结束时回到调用位置继续执行. 可以借助调试器来观察.
- 函数的定义和调用的先后顺序没有要求. (这一点和变量不同, 变量必须先定义再使用)

```
// 调用函数
hello();
// 定义函数
function hello() {
    console.log("hello");
}
```

关于参数个数

实参和形参之间的个数可以不匹配. **但是实际开发一般要求形参和实参个数要匹配**

1) 如果实参个数比形参个数多, 则多出的参数不参与函数运算

```
sum(10, 20, 30);    // 30
```

2) 如果实参个数比形参个数少, 则此时多出来的形参值为 undefined

```
sum(10);    // NaN, 相当于 num2 为 undefined.
```

> JS 的函数传参比较灵活, 这一点和其他语言差别较大. 事实上这种灵活性往往不是好事.

函数表达式

另外一种函数的定义方式

```
var add = function() {
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
console.log(add(10, 20));            // 30
console.log(add(1, 2, 3, 4));        // 10

console.log(typeof add);             // function
```

此时形如 `function() { }` 这样的写法定义了一个匿名函数, 然后将这个匿名函数用一个变量来表示.

后面就可以通过这个 add 变量来调用函数了.

> JS 中函数是一等公民, 可以用变量保存, 也可以作为其他函数的参数或者返回值.

作用域

某个标识符名字在代码中的有效范围.

在 ES6 标准之前, 作用域主要分成两个

- 全局作用域: 在整个 script 标签中, 或者单独的 js 文件中生效.
- 局部作用域/函数作用域: 在函数内部生效.

```
// 全局变量
var num = 10;
console.log(num);

function test() {
    // 局部变量
```

```
    var num = 20;
    console.log(num);
}

function test2() {
    // 局部变量
    var num = 30;
    console.log(num);
}

test();
test2();

console.log(num);

// 执行结果
10
20
30
10
```

> 创建变量时如果不写 var, 则得到一个全局变量.
>
> ```
> function test() {
>     num = 100;
> }
> test();
> console.log(num);
>
> // 执行结果
> 100
> ```
>
> 另外, 很多语言的局部变量作用域是按照代码块(大括号)来划分的, JS 在 ES6 之前不是这样的.
>
> ```
> if (1 < 2) {
>     var a = 10;
> }
> console.log(a);
> ```

作用域链

背景:

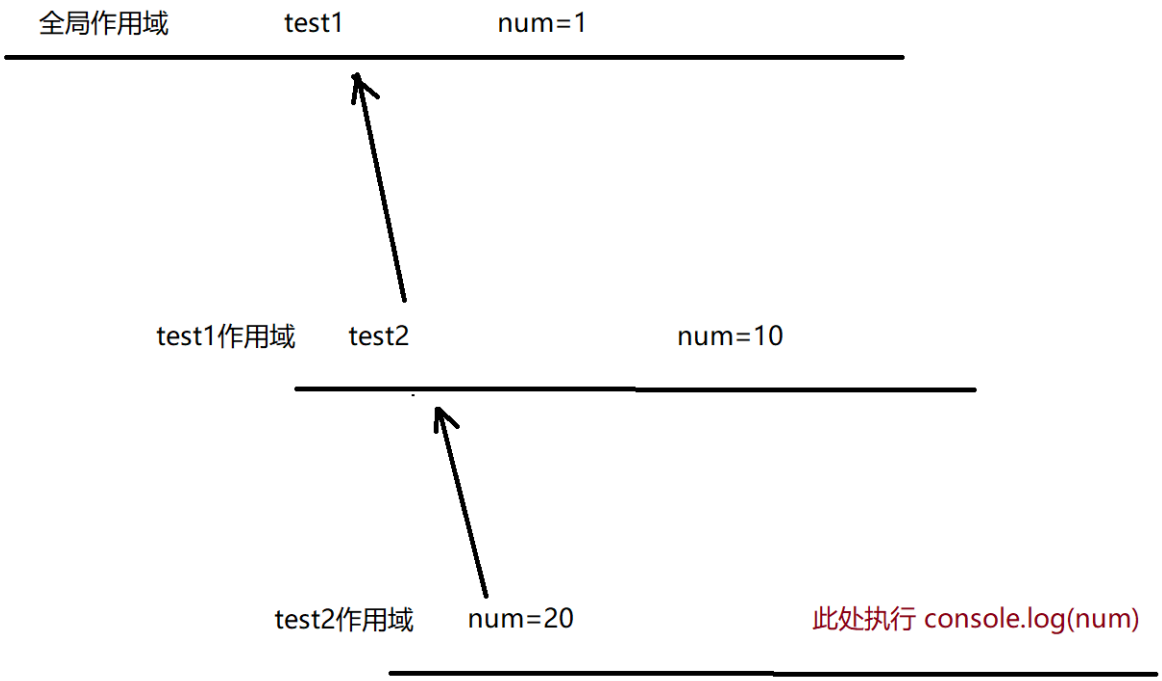
- 函数可以定义在函数内部
- 内层函数可以访问外层函数的局部变量.

内部函数可以访问外部函数的变量. 采取的是链式查找的方式. 从内到外依次进行查找.

```
var num = 1;
function test1() {
    var num = 10;

    function test2() {
        var num = 20;
```

```
        console.log(num);
    }

    test2();
}
test1();

// 执行结果
20
```

执行 console.log(num) 的时候, 会现在 test2 的局部作用域中查找 num. 如果没找到, 则继续去 test1 中查找. 如果还没找到, 就去全局作用域查找.



对象

基本概念

对象是指一个具体的事物.

> "电脑" 不是对象, 而是一个泛指的类别. 而 "我的联想笔记本" 就是一个对象.

在 JS 中, 字符串, 数值, 数组, 函数都是对象.

每个对象中包含若干的属性和方法.

- 属性: 事物的特征.
- 方法: 事物的行为.

> 例如, 你有一个女票.
>
> 她的身高体重三围这些都是属性.
>
> 她的唱歌, 跳舞, 暖床都是方法.

对象需要保存的属性有多个, 虽然数组也能用于保存多个数据, 但是不够好.

> 例如表示一个学生信息. (姓名蔡徐坤, 身高 175cm, 体重 170斤)
>
> var student = ['蔡徐坤', 175, 170];
>
> 但是这种情况下到底 175 和 170 谁表示身高, 谁表示体重, 就容易分不清

JavaScript 的对象 和 Java 的对象概念上基本一致. 只是具体的语法表项形式差别较大.

1. 使用 字面量 创建对象 [常用]

使用 { } 创建对象

```
var a = {};  // 创建了一个空的对象

var student = {
    name: '蔡徐坤',
    height: 175,
    weight: 170,
    sayHello: function() {
        console.log("hello");
    }
};
```

- 使用 { } 创建对象
- 属性和方法使用键值对的形式来组织.
- 键值对之间使用 , 分割. 最后一个属性后面的 , 可有可无
- 键和值之间使用 : 分割.
- 方法的值是一个匿名函数.

使用对象的属性和方法:

```
// 1. 使用 . 成员访问运算符来访问属性 `.` 可以理解成 "的"
console.log(student.name);
// 2. 使用 [ ] 访问属性, 此时属性需要加上引号
console.log(student['height']);
// 3. 调用方法, 别忘记加上 ()
student.sayHello();
```

2. 使用 new Object 创建对象

```
var student = new Object(); // 和创建数组类似
student.name = "蔡徐坤";
student.height = 175;
student['weight'] = 170;
student.sayHello = function () {
    console.log("hello");
}

console.log(student.name);
console.log(student['weight']);
student.sayHello();
```

注意, 使用 { } 创建的对象也可以随时使用 `student.name = "蔡徐坤";` 这样的方式来新增属性.

3. 使用 构造函数 创建对象

前面的创建对象方式只能创建一个对象. 而使用构造函数可以很方便 的创建 多个对象.

例如: 创建几个猫咪对象

```
var mimi = {
    name: "咪咪",
    type: "中华田园喵",
    miao: function () {
        console.log("喵");
    }
};

var xiaohei = {
    name: "小黑",
    type: "波斯喵",
    miao: function () {
        console.log("猫呜");
    }
}

var ciqiu = {
    name: "刺球",
    type: "金渐层",
    miao: function () {
        console.log("咕噜噜");
    }
}
```

此时写起来就比较麻烦. 使用构造函数可以把相同的属性和方法的创建提取出来, 简化开发过程.

基本语法

```
function 构造函数名(形参) {
    this.属性 = 值;
    this.方法 = function...
}

var obj = new 构造函数名(实参);
```

注意:

- 在构造函数内部使用 this 关键字来表示当前正在构建的对象.
- 构造函数的函数名首字母一般是大写的.
- 构造函数的函数名可以是名词.
- 构造函数不需要 return
- 创建对象的时候必须使用 new 关键字.

> this 相当于 "我"

使用构造函数重新创建猫咪对象

```
function Cat(name, type, sound) {
    this.name = name;
    this.type = type;
    this.miao = function () {
        console.log(sound); // 别忘了作用域的链式访问规则
    }
}

var mimi = new Cat('咪咪', '中华田园喵', '喵');
var xiaohei = new Cat('小黑', '波斯喵', '猫鸣');
var ciqiu = new Cat('刺球', '金渐层', '咕噜噜');

console.log(mimi);
mimi.miao();
```

理解 new 关键字

new 的执行过程:

1. 先在内存中创建一个空的对象 { }
2. this 指向刚才的空对象(将上一步的对象作为 this 的上下文)
3. 执行构造函数的代码, 给对象创建属性和方法
4. 返回这个对象 (构造函数本身不需要 return, 由 new 代劳了)

参考 https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new

JavaScript 的对象和 Java 的对象的区别

1. JavaScript 没有 "类" 的概念

对象其实就是 "属性" + "方法" .

类相当于把一些具有共性的对象的属性和方法单独提取了出来, 相当于一个 "月饼模子"

在 JavaScript 中的 "构造函数" 也能起到类似的效果.

而且即使不是用构造函数, 也可以随时的通过 { } 的方式指定出一些对象

> 在 ES6 中也引入了 class 关键字, 就能按照类似于 Java 的方式创建类和对象了.

2. JavaScript 对象不区分 "属性" 和 "方法"

JavaScript 中的函数是 "一等公民", 和普通的变量一样. 存储了函数的变量能够通过 ( ) 来进行调用执行.

3. JavaScript 对象没有 private / public 等访问控制机制.

对象中的属性都可以被外界随意访问.

4. JavaScript 对象没有 "继承"

继承本质就是 "让两个对象建立关联". 或者说是让一个对象能够重用另一个对象的属性/方法.

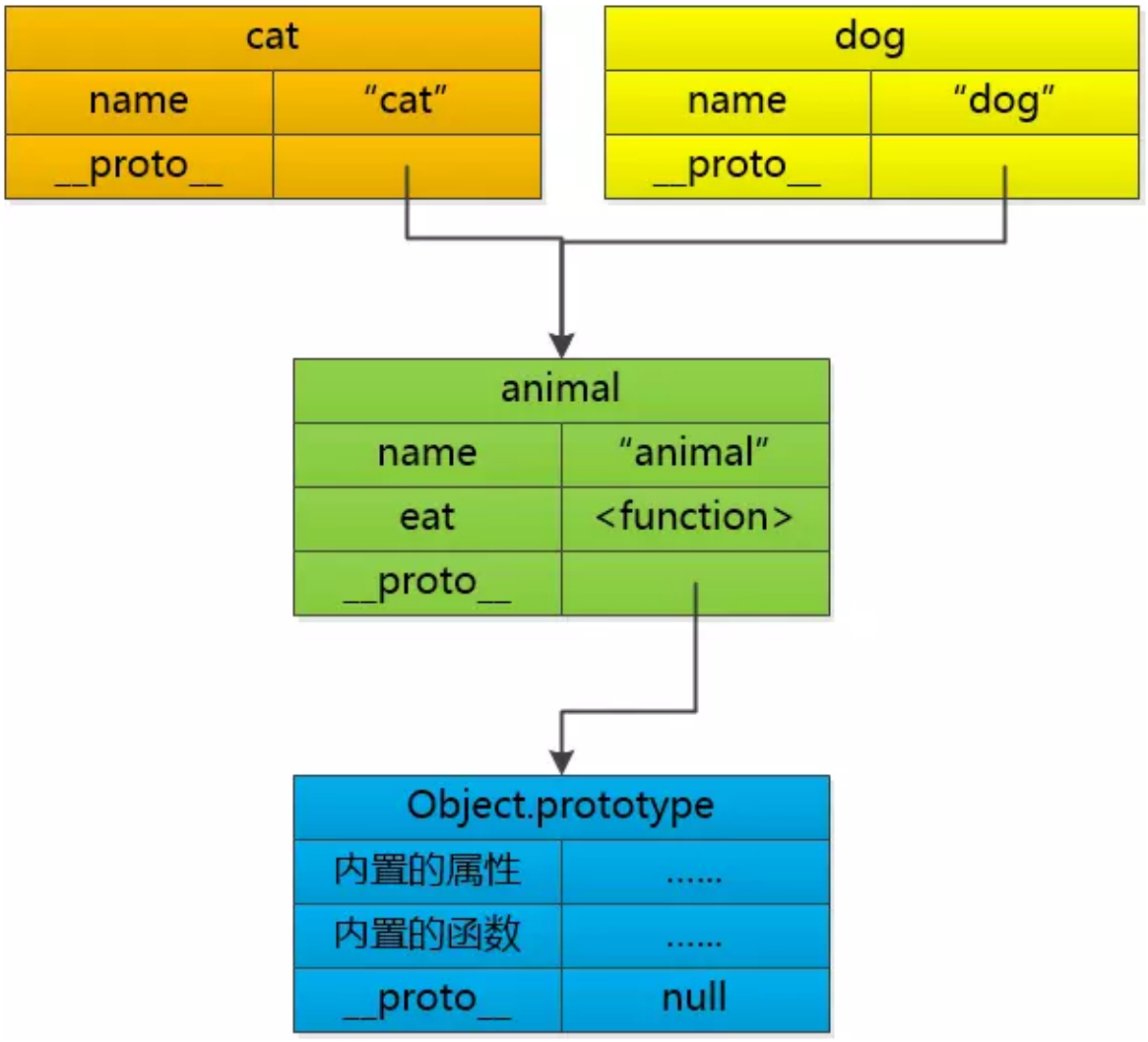
JavaScript 中使用 "原型" 机制实现类似的效果.

例如: 创建一个 cat 对象和 dog 对象, 让这两个对象都能使用 animal 对象中的 eat 方法.

通过 `__proto__` 属性来建立这种关联关系 (proto 翻译作 "原型")

```
var dog = {
  name: "dog",
  __proto__: animal  //指向animal对象
};

var cat = {
  name: "cat",
  __proto__: animal // 指向animal对象
};

dog.eat(); // dog is eating
cat.eat(); // cat is eating
```

当eat方法被调用的时候，先在自己的方法列表中寻找， 如果找不到，就去找原型中的方法， 如果原型中找不到， 就去原型的原型中去寻找..... 最后找到Object那里， 如果还找不到， 那就是未定义了。



> 关于原型链的内容比较复杂, 此处不做过多讨论

5. JavaScript 没有 "多态"

多态的本质在于 "程序猿不必关注具体的类型, 就能使用其中的某个方法".

C++ / Java 等静态类型的语言对于类型的约束和校验比较严格. 因此通过 子类继承父类, 并重写父类的方法的方式 来实现多态的效果.

但是在 JavaScript 中本身就支持动态类型, 程序猿在使用对象的某个方法的时候本身也不需要对对象的类型做出明确区分. 因此并不需要在语法层面上支持多态.

> 例如:
>
> 在 Java 中已经学过 ArrayList 和 LinkedList. 为了让程序猿使用方便, 往往写作
>
> `List<String> list = new ArrayList<>()`
>
> 然后我们可以写一个方法:
>
> ```
> void add(List<String> list, String s) {
>     list.add(s);
> }
> ```

> 我们不必关注 list 是 ArrayList 还是 LinkedList, 只要是 List 就行. 因为 List 内部带有 add 方法.
>
> 当我们使用 JavaScript 的代码的时候
>
> ```
> function add(list, s) {
>     list.add(s)
> }
> ```
>
> add 对于 list 这个参数的类型本身就没有任何限制. 只需要 list 这个对象有 add 方法即可. 就不必像 Java 那样先继承再重写绕一个圈子.