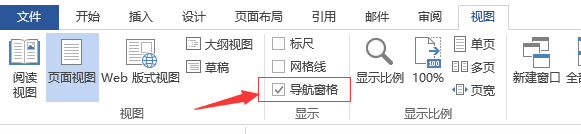
<前端入门文档>

OTT-终端组撰写

阅读建议

**方式一：**建议开启word导航窗口，利用一级&二级标题来快速查阅

导航窗口打开方式：视图 -> 导航窗格 打钩

**方式二：**建议使用阅读视图，在视图->阅读视图，或者右下角的书本按钮

声明

本文档整理自OTT终端组的前端分享系列课程，课程资料参考自以下来源：

* 《HTML5权威指南》
* 《JavaScript权威指南》
* 《ES6标准入门》
* 《Vue.js实战》
* MDN web docs : <https://developer.mozilla.org/zh-CN/docs/Web>
* Github: <https://github.com/smyhvae/Web>
* TypeScript 中文网: <https://www.tslang.cn/docs/handbook/basic-types.html>

作者信息

作者：苏晓强 文档创建时间：2018-07-24

# 目录

[目录 3](#_Toc533020427)

[学习路线 9](#_Toc533020428)

[相关概念 10](#_Toc533020429)

[HTML CSS JS 10](#_Toc533020430)

[基础 11](#_Toc533020431)

[HTML 11](#_Toc533020432)

[CSS 12](#_Toc533020433)

[JavaScript 15](#_Toc533020434)

[标签1-修饰文档结构 16](#_Toc533020435)

[标准HTML文档基本机构 16](#_Toc533020436)

[<!DOCTYPE> 17](#_Toc533020437)

[<html> 18](#_Toc533020438)

[<head> 19](#_Toc533020439)

[<body> 20](#_Toc533020440)

[<meta> 21](#_Toc533020441)

[<link> 22](#_Toc533020442)

[标签2-修饰文本内容 23](#_Toc533020443)

[该类别标签含义 23](#_Toc533020444)

[<a> 24](#_Toc533020445)

[<br> & <wbr> 25](#_Toc533020446)

[<p> 26](#_Toc533020447)

[<pre> & <code> 27](#_Toc533020448)

[<ol> & <li> 28](#_Toc533020449)

[<ul> & <li> 29](#_Toc533020450)

[<h1> ~ <h6> 30](#_Toc533020451)

[<table>表格 31](#_Toc533020452)

[<tr>&<th>&<td> 31](#_Toc533020453)

[<thead>&<tfoot>&<tbody> 32](#_Toc533020454)

[<caption> 33](#_Toc533020455)

[完整示例 33](#_Toc533020456)

[<form>表单 35](#_Toc533020457)

[<fieldset> 36](#_Toc533020458)

[<button> 37](#_Toc533020459)

[<input> 38](#_Toc533020460)

[<iframe> 42](#_Toc533020461)

[<img> 43](#_Toc533020462)

[标签3-容器类 44](#_Toc533020463)

[<span> & <div> 44](#_Toc533020464)

[<section> 45](#_Toc533020465)

[<header> & <footer> 46](#_Toc533020466)

[<nav> 47](#_Toc533020467)

[标签文档 48](#_Toc533020468)

[CSS结构和语法 49](#_Toc533020469)

[工作原理 49](#_Toc533020470)

[结构 49](#_Toc533020471)

[盒模型 50](#_Toc533020472)

[使用方式 51](#_Toc533020473)

[CSS选择器 52](#_Toc533020474)

[基本选择器 52](#_Toc533020475)

[多规则选择器 52](#_Toc533020476)

[伪选择器 55](#_Toc533020477)

[CSS层叠算法 58](#_Toc533020478)

[CSS属性 61](#_Toc533020479)

[单位 in cm mm px em % 61](#_Toc533020480)

[字体 font-xxx 61](#_Toc533020481)

[文本 text-xxx 61](#_Toc533020482)

[背景 background-xxx 62](#_Toc533020483)

[盒子 width height margin padding border&… 64](#_Toc533020484)

[阴影 box-shadow text-shadow 65](#_Toc533020485)

[显示模式 display 66](#_Toc533020486)

[浮动 float 68](#_Toc533020487)

[位置 position 71](#_Toc533020488)

[弹性布局 flex 74](#_Toc533020489)

[CSS预处理器 87](#_Toc533020490)

[CSS预处理 87](#_Toc533020491)

[Less 88](#_Toc533020492)

[Sass（Scss） 96](#_Toc533020493)

[JavaScript快速入门 102](#_Toc533020494)

[ES5 ES6 102](#_Toc533020495)

[语法 102](#_Toc533020496)

[字符串操作 105](#_Toc533020497)

[作用域 105](#_Toc533020498)

[null undefined 106](#_Toc533020499)

[类型隐式转换 106](#_Toc533020500)

[回调写法 106](#_Toc533020501)

[DOM 107](#_Toc533020502)

[概念 107](#_Toc533020503)

[document 108](#_Toc533020504)

[location 108](#_Toc533020505)

[window 109](#_Toc533020506)

[HTMLElement 111](#_Toc533020507)

[DOM模型示例 112](#_Toc533020508)

[DOM事件 113](#_Toc533020509)

[jQuery 117](#_Toc533020510)

[为什么使用jQuery 117](#_Toc533020511)

[API中文文档 117](#_Toc533020512)

[Ajax 117](#_Toc533020513)

[css 117](#_Toc533020514)

[class 118](#_Toc533020515)

[html 119](#_Toc533020516)

[attr 120](#_Toc533020517)

[h5微信授权接入 121](#_Toc533020518)

[JavaScript语法 122](#_Toc533020519)

[相关术语 122](#_Toc533020520)

[数据类型 127](#_Toc533020521)

[运算符 141](#_Toc533020522)

[对象 150](#_Toc533020523)

[数组 166](#_Toc533020524)

[函数 176](#_Toc533020525)

[JavaScript进阶 191](#_Toc533020526)

[原型 191](#_Toc533020527)

[继承 200](#_Toc533020528)

[原型链 210](#_Toc533020529)

[执行上下文和变量对象（EC&VO） 217](#_Toc533020530)

[作用域 220](#_Toc533020531)

[作用域链 226](#_Toc533020532)

[闭包 233](#_Toc533020533)

[异步回调执行的时机 238](#_Toc533020534)

[JavaScript ES6 247](#_Toc533020535)

[Symbol 247](#_Toc533020536)

[块作用域 let 和 const 248](#_Toc533020537)

[Module 模块机制 249](#_Toc533020538)

[变量解构 251](#_Toc533020539)

[字符串的扩展 252](#_Toc533020540)

[函数的扩展 253](#_Toc533020541)

[箭头函数 254](#_Toc533020542)

[class 类 256](#_Toc533020543)

[其他 259](#_Toc533020544)

[JavaScript模块化 261](#_Toc533020545)

[模块化历程 261](#_Toc533020546)

[CommonJS规范 265](#_Toc533020547)

[AMD规范 268](#_Toc533020548)

[CMD规范 272](#_Toc533020549)

[CommonJS, AMD, CMD 三者间区别 275](#_Toc533020550)

[ES6标准 277](#_Toc533020551)

[TypeScript 282](#_Toc533020552)

[为什么使用TypeScript 282](#_Toc533020553)

[WebStrom 配置 288](#_Toc533020554)

[tsconfig.json 289](#_Toc533020555)

[语法-数据类型 289](#_Toc533020556)

[语法-类型声明 290](#_Toc533020557)

[语法-接口 293](#_Toc533020558)

[语法-类机制 295](#_Toc533020559)

[语法-泛型 297](#_Toc533020560)

[语法-模块和命名空间 297](#_Toc533020561)

[其他 299](#_Toc533020562)

# 学习路线

一、学习 HTML、CSS基础

二、学习客户端 JavaScript、DOM概念，即掌握借助浏览器相关 API 操纵 DOM 树修改页面

三、jQuery 和 Ajax 相关

四、H5 小项目实战

五、学习 JavaScript 语法基础

六、深入学习 JavaScript 核心语法原理，包括作用域链原理、原型链、闭包等

七、学习 ES6 新增特性

八、了解 JavaScript 模块化技术的发展历程，掌握目前的模块化相关规范

九、了解预处理器 Less 和 Sass，响应式布局 BootStrap 框架

十、学习 TypeScript

十一、学习前端框架 Vue.js

十二、待续

# 相关概念

## HTML CSS JS

HTML: 内容层---表示某个标签在页面中是什么角色

CSS: 样式层---表示某个标签在页面中该呈现什么样式

JavaScript: 行为层---页面与用户的交互行为

通俗的讲：Html 只负责文档的语义和结构，它描述了网页的内容和结构；

内容的呈现则由应用于元素上的CSS样式控制，它描述了网页的表现与展示效果；

JavaScript 则是负责网页的功能与行为，如与用户的交互。

# 基础

## HTML

Html 是一种标记语言，不是编程语言，需要明确这点。

编程语言，通俗的理解就是向计算机发送指令，通过程序让计算机达到我们想要的功能。

而标记语言，可以通俗的理解成对文本增加一些标志信息，类似于批注信息来说明文本内容

更通俗点理解，一大段文本内容，毫无重点，毫无结构，让人不好理解。所以，Html就通过大伙约定俗成的规范，利用一些标签，来指明，这一段文本是标题，这一段文本是个表格，这一段文本是个列表，这一段文本是导航菜单，这里需要分段，这里需要分行，这几个词是关键词，需要重点标记一下，等等等等。

Html可以这么通俗的理解，那么这个大伙约定俗成的规范，其实也就是由W3C来作为官方发布的标准规范。

定了规范，自然是想要让人遵守才有意义。所以，在这里，W3C所定义的标准规范，遵守方其实也就是各大浏览器。不同的浏览器厂商，根据W3C发布的标准规范来解析每一份HTML文档，那么相同的HTML文档在不同的浏览器上才会有相同的作用。

基本概念

<a class="ddd" href="index.html">点击跳转</a>

**标签**：<> 带有这种符号的称为标签，跟Android中的标签一样，分开始标签，结束标签

**内容**：文本内容，上图中的点击跳转四字

**属性**：跟Android中的xml里的标签一样，每个标签有自己的一些属性，另外，Html有一些全局属性，比如上述的 class，这个属性所有标签都可以用。

**元素**：标签 + 内容

## CSS

CSS负责文本样式的呈现，既然将HTML 和 CSS 分离开，各自只负责各自的职责，那么肯定需要某种方式将两者连接在一起。

更准确的说，是在 HTML 文档中该如何使用 CSS，因为HTML 文档是互联网的基础，一个个网页就是一份份HTML文档，既然HTML文档是基础，那么就是要明确在HTML文档中该如何使用CSS。

总共有三种方式：全局属性style，style标签，link标签

CSS 最终作用的对象其实就是HTML文档中的每个元素

全局属性 style

因此，第一种方式：全局属性 style 是直接作用于指定的标签上了，用法就直接将需要的样式声明即可，如：

<a href="index.html" style="background: gray; color: #6a90d9">点击跳转</a>

剩余两种方式，都是集中将所有的 CSS 样式管理存放，因此如果需要作用到具体元素上，要借助选择器来实现，选择器后面再说，先看这两种方式的使用：

style标签内嵌方式：

<style type="text/css">  
 a {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

Link标签引用外部文件方式

<link type="text/css" rel="stylesheet" href="css/nms.css">

选择器

选择器很多，规则也很多，足够覆盖各种各样的场景，这里只列举几种常见的选择器：

元素选择器

<a class="myClass" href="index.html">点击跳转</a>

<span class="myClass">另一文本</span>

<style type="text/css">  
 a {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

作用于HTML文档中的所有 a 标签的元素上

Id选择器

<a id="myId" href="index.html">点击跳转</a>

<style type="text/css">  
 #myId {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

Id 在 HTML 文档中需唯一存在，所有id选择器只作用于单个元素

Class选择器

<a class="myClass" href="index.html">点击跳转</a>

<span class="myClass">另一文本</span>

<style type="text/css">  
 .myClass {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

HTML文档中可对多个元素应用相同class，所以class可同时作用于多个元素

属性选择器

<a class="myClass" href="index.html">点击跳转</a>

<span class="myClass">另一文本</span>

<style type="text/css">  
 [href] {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

作用于所有具有href属性的元素，不管有没有使用

组合选择器

<a class="myClass" href="index.html">点击跳转</a>

<span class="myClass">另一文本</span>

<style type="text/css">  
 a.myClass {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

元素选择器和class选择器组合使用，作用于 a 元素中有声明myClass类型的元素

通用选择器

<style type="text/css">  
 \* {  
 background: gray;  
 }  
</style>

作用到所有元素上

## JavaScript

Js 是脚本语言，可用于增加网页的交互功能，及各种行为功能。

既然是一门语言，那么学习这么脚本语言自然需要先掌握其语法，以及运行方式。

浏览器解析 HTML 文档是按顺序解析的，也就是说，当遇到脚本语言时，也会按顺序一条条的解释执行，直至将脚本语言执行结束再继续解析文档。这就解释了，为什么一份 HTML 文档里，<script> 标签经常是在各种各样的位置出现的。

而 HTML 文档使用 js 的方式是通过 <script> 标签来实现，如：

内嵌

<script>  
 *console*.log("hello world")  
</script>

引用外部js文件

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

# 标签1-修饰文档结构

## 标准HTML文档基本机构

<!DOCTYPE html> ------ 声明这是一份H5文档

<html > ------ HTML文档内容开始

<head>

… ------ 在<head>标签中声明文档的各种元数据，

… ------ 该部分内容是给浏览器看的

</head>

<body>

… -------<body>标签内为文档的文本内容

… ------ 该部分内容是给用户看的

</body>

</html>

## <!DOCTYPE>

准确的说，<!DOCTYPE>并不是HTML标签，它是声明web浏览器关于页面使用哪个 HTML 版本进行编写的指令。

在 HTML 4.01 中，<!DOCTYPE> 声明引用 DTD，因为 HTML 4.01 基于 SGML。DTD 规定了标记语言的规则，这样浏览器才能正确地呈现内容。

[HTML5](https://www.html5tricks.com/) 不基于 SGML，所以不需要引用 DTD。

如 H5中用法：

<!DOCTYPE html>

HTML 4 中用法：

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">

## <html>

每一份HTML文档的内容的根节点，表示文档内容的开始

文档内容包括两部分：头部声明<head>和文本内容<body>

## <head>

HTML文档的头部声明，用于声明该文档的一些属性，以及一些元数据，<head> 内部的内容是用于给浏览器看的，并不是用于给用户看的，浏览器通过<head>信息，知晓这份文档引用了哪些外部资源文件，使用的哪些属性。

可在<head>中使用的标签并不多：

<head>

<base href=”http://www.dasu.”> --设置基准URL

<meta charset=“UTF-8”> ---声明文档所使用的编码

<title>Title</title> ---声明文档的标题

<base href=“http://www.baidu.com”>---声明文档全局的基准URL

<style type=“text/css”></style> ---声明内嵌类型的css样式

<link type=“text/css” rel=“stylesheet” href=“css/nms.css”>--外部css

<script></script> --- JavaScript脚本

<noscript></noscript> ---浏览器不支持JS情况下的处理

</head>

## <body>

<body>标签用于声明文本内容，该标签内的内容都是用于展示给用户看的，所以基本所有标签都可以在 <body> 内，浏览器解析相应的标签，并根据 CSS 作用到每个对象上，生成网页呈现给用户。

## <meta>

元数据标签，可用于声明文档所使用的一些元数据，用途蛮多，如下：

<meta charset=”utf-8”/> ---声明文档编码格式

<meta http-equiv=”refresh” content=”5”/> --往Http头部中增加key-value

<meta http-equiv=”content-type” content=”text/html charset=UTF-8”/>

## <link>

<link>标签用于指定HTML文档使用了哪些外部资源文件，可以是外部的CSS文件，或者网页图标，或者说明文档等等，如下：

<link rel=”stylesheet” type=”text/css” href=”styles.css”/>--外部CSS

<link rel=”shortcut icon” href=”favicon.ico” type=”image/x-icon”/>-网页图标

<link rel=”prefetch” href=”/page2.html”>--预先加载page2.html文件

…

# 标签2-修饰文本内容

## 该类别标签含义

这是我自己为标签进行的划分，《HTML权威指南》中将标签类别划分成了很多种，比如：内容分组，文档分节，表单七七八八等等。

但我按照自己个人的理解习惯，对总的标签划分成三类：修饰文档结构的标签、修饰文本内容标签、容器类标签。

修饰文档结构的标签如上一节中描述的那些用于表示一份标准、完整的HTML文档的一些标签，以及可放于<header>内的一些标签。

修饰文本内容标签，大意是说，这些标签是直接用于标记文本内容，赋予文本内容某些语义行为，比如<a>赋予超链接语义，<h1>赋予了一级标题语义等等。这个类别有些类似于 Android 中表示某个 View 的标签，通俗来讲，这些标签可直接使用在文本内容上了。

容器类标签，并不是真正意义上的容器，而是说，这类标签主要的作用是用来包含其他标签的，但并不是说，只能用来包含其他标签，也可直接对文本内容标记。如<nav>,<header>这类表示某一块区域的标签。

## <a>

<a>标签的作用是引导用户从一张页面链接到另一张页面，互联网说到底就是一张张网页通过超链接<a>互相关联起来的。

所以，只要不是单个页面，只要页面支持跳转，那么HTML文档中就肯定有 <a> 标签的存在，用于指定下个页面的跳转。

用法

<**a href="http://www.baidu.com"**>百度</**a**>  
<**a href="index.html"**>首页</**a**>  
<**a href="#myId"**>标题5</**a**>

以上是<a>标签的三种用法，指定绝对路径的链接，指定相对路径的链接，指定文档内的链接。

也就是说，<a>标签既可以用于指定页面间的跳转关联，也可以指定页面内的跳转。

<base> 标签设置的基准url会影响到相对路径的拼接，默认以当前HTML文档的路径作为基准路径。

另外新页面的打开方式也支持多种形式配置，如：

<**a href="http://www.baidu.com" target="\_blank"**>百度</**a**>

通过 target 属性来声明新页面的打开方式，target可以取值如下：

|  |  |
| --- | --- |
| **\_blank** | **在新页面或标签页中打开文档** |
| **\_parent** | 在父窗框组（frameset）中打开文档 |
| **\_self** | 在当前窗口中打开文档（默认行为） |
| **\_top** | 在顶层窗口打开文档 |
| **<frame>** | 在指定窗框中打开文档 |

## <br> & <wbr>

换行标签

<br>表示将后续内容转移到新行上

<wbr> H5新增的，表示当长度超过当前浏览器窗口的内容适合在此换行。

两者都是换行，前者是强制换行，后者是建议在这里换行，但什么时候换行，由浏览器根据当前窗口大小决定。

用法

<**br**/>

## <p>

<p>标签用于表示段落，标签围起来的文本内容表示一个段落。

因为浏览器会忽略所有的空格、缩进、换行，所以，即使文本内容的段落结构很好，但经由浏览器解析出来后的文本内容全部都挤到一堆。

因此，<p>段落标签还是很有必要的。

可以用此来标记哪些文本内容作为一个段落。

用法

<p>。。。。。。。</p>

## <pre> & <code>

<pre>标签用于保留源文档中的格式。

<code>标签用于表示代码块。

由于浏览器会合并空白字符，忽略掉换行，导致如果文档中含有代码块时显示不符合开发工具下的代码格式风格。

此时，可以借助<pre>标签和<code>一起使用，来阻止浏览器合并空白字符，达到保留代码格式的目的。

用法

<pre><code>

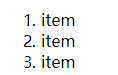
…..

</code></pre>

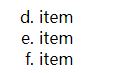
## <ol> & <li>

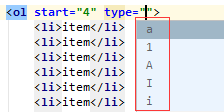
<ol>标签用于表示有序列表，<li>标签用于表示列表中的每一项。

用法

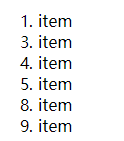
<**ol**>  
 <**li**>item</**li**>   
 <**li**>item</**li**>  
 <**li**>item</**li**>  
</**ol**>

既然是有序列表项，那么序号的起始以及样式是支持通过属性设定的，如下：

<**ol start="4" type="a"**>  
 <**li**>item</**li**>  
 <**li**>item</**li**>  
 <**li**>item</**li**>   
</**ol**>

通过 start 属性设置起始的编号，通过type 属性设置编号的样式，默认的type样式如下：

如果要实现编号是非连续的，那么可以借助<li>标签的 value 属性实现：

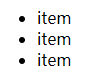
<**ol** >  
 <**li**>item</**li**>  
 <**li value="3"**>item</**li**>  
 <**li**>item</**li**>  
 <**li**>item</**li**>  
 <**li value="8"**>item</**li**>  
 <**li**>item</**li**>  
</**ol**>

注：每个ol列表项都是独立存在，编号默认都从0开始，如果想实现不同列表项的编号连贯，或者想以跨度2 或其他数递增，那么只用标签属性实现局限很多。这时，可考虑通过CSS的::before选择器实现，具体实现参考相应小节。

## <ul> & <li>

<ul>标签用于表示无序列表，<li>标签表示列表里的每一项。

用法

<**ul**>  
 <**li**>item</**li**>  
 <**li**>item</**li**>  
 <**li**>item</**li**>   
</**ul**>

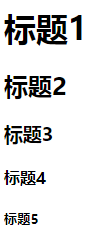
因为是无序列表，所以用法比起有序列表<ol>简单很多，无需使用任何属性，直接用无序标签<ul>包含一系列子项<li>即可。

至于，每一项前的样式，可通过CSS样式，通过 list-style-type 属性实现，以上样式对应的CSS：

**ul** { **list-style-type**: **disc**}

## <h1> ~ <h6>

标题标签，对应一级到六级标题。

用法

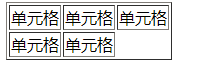
<**h1**>标题1</**h1**>  
<**h2**>标题2</**h2**>  
<**h3**>标题3</**h3**>  
<**h4**>标题4</**h4**>  
<**h5**>标题5</**h5**>

## <table>表格

HTML文档做一个表格挺复杂的，涉及的标签很多，如<thead>, <tfoot>, <tbody> 等等。

但根节点总是<table>，一份表格无外乎就是各种单元格组成，而单元格标签为<td>，另外，浏览器解析表格文本时，是以行为单位来构建表格，并不是列，所以每个单元格都需要指定位于哪一行中，行标签为<tr>。而所有行的单元格都是表格的主要内容，因此都在<tbody>标签中。

以上是表格的最基本要素，因此一张最简单的表格，至少需要<table>，<tbody>，<tr>，<td> 三种标签。

<**table**>  
 <**tbody**>  
 <**tr**>  
 <**td**>单元格</**td**><**td**>单元格</**td**><**td**>单元格</**td**>  
 </**tr**>  
 <**tr**>  
 <**td**>单元格</**td**><**td**>单元格</**td**>  
 </**tr**>  
 </**tbody**>  
</**table**>

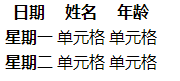
有时候，写表格标签时，如果没有其他表头<thead>部分，或者表脚<tfoot>时，会将<tbody>省略，但这并不是说就可以不用<table>标签，而是很多浏览器会自动将<tbody>填补上。

### <tr>&<th>&<td>

由于浏览器是以行为单位构建表格，所以一个表格有多少行就是通过<tr>标签来控制，哪些单元格属于哪一行，就放在那一行的<tr>标签中。

虽然说表格都是由一个个的单元格组成，但单元格之间还可以继续划分含义，有些单元格是表示内容，而有些单元格则是表示属性值，或者说列头或行头。

通常来说，这些标题类型的表格都是在第一行或第一列的单元格：



这是一个很常见的二维表格，通过<th>和<td>来将表格的单元格含义区分开。

<th>标签用于表示单元格的表头

<td>标签用于表格单元格的内容

既然是单元格，那么就会存在合并单元格的现象，通俗的讲也就是有些表格的大小并不是只占据一格，而是有可能多行多列。此时，可通过属性来实现。

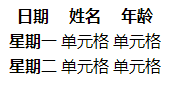
|  |  |
| --- | --- |
| **colspan** | **rowspan** |
| **单位数值，如1表示占据1列** | 单位数值，如2表示占据2行 |

### <thead>&<tfoot>&<tbody>

类似于HTML文档有一些专门用于表明文档结构的标签，表格同样有一些用于指示表格结构的标签。引入表格结构标签，是为了更好的区分出各个单元格的含义。

比如，<th>标签用来表示表头类型的单元格，但不管是第一行的表头，还是第一列的表头，用的都是<th>，那如果还想继续划分这个表头是属于第一行或者第一列时该怎么做呢？

所以，引入了一些结构性标签有便于对表格各个单元格更加具体的细分，以满足各种复杂场景。

<**table**>  
 <**thead**>  
 <**tr**>  
 <**th**>日期</**th**><**th**>姓名</**th**><**th**>年龄</**th**>  
 </**tr**>  
 </**thead**>  
  
 <**tbody**>  
 <**tr**>  
 <**th**>星期一</**th**><**td**>单元格</**td**><**td**>单元格</**td**>  
 </**tr**>  
 <**tr**>  
 <**th**>星期二</**th**><**td**>单元格</**td**><**td**>单元格</**td**>  
 </**tr**>  
 </**tbody**>  
</**table**>

<table>是表格的根节点

<thead>用来标记表格的标题行

<tfoot>用来标记组成脚的行。

不用<thead>，表格最终效果也一样，但用了<thead>之后，如果CSS想分别作用第一行，或者第一列，这时就可以很容易的通过 thread th 以及 tbody th 来达到目的了。

所以，结合表格结构性标签的使用，可以让表格的结构更加明确。

### <caption>

表格除了一张表格内容外，通常还会需要有表格的标题，此时用<caption>标签来标记。

### 完整示例

表格也有标题，而且是一个二维表格，表头也给出来了，其中，姓名一列有单元格合并。

<**table border="1"**>  
 <**caption**>工作记录表</**caption**>  
 <**thead**>  
 <**tr**>  
 <**th**>日期</**th**><**th**>姓名</**th**><**th**>记录</**th**>  
 </**tr**>  
 </**thead**>  
  
 <**tbody**>  
 <**tr**>  
 <**th**>2018-07-24</**th**><**td rowspan="2"**>suxq</**td**><**td**>单元格</**td**>  
 </**tr**>  
 <**tr**>  
 <**th**>2018-07-25</**th**><**td**>单元格</**td**>  
 </**tr**>  
 </**tbody**>  
</**table**>

## <form>表单

表单在网页中的角色很重要，因为表单是用来接收用户输入的信息并提交发送给服务器的中间角色。

表单并不是说，流程器就呈现给用户一张表单，反而通常呈现给用户的只是各自输入控件，比如输入框，或者勾选控件之类的。

表单，我的理解是，浏览器会将用户输入的这些数据收集起来生成一张表单提交给服务端。

<form>标签则是表单的根节点。

因为表单需要收集用户输入的信息，以及提交服务端的时机，因此，一般来说，表单还需要有<input>标签以及<button>标签。

一份基本的表单如下：

<**form method="post" action="form.html"**>  
 <**input name="key"**/>  
 <**button**>提交</**button**>  
</**form**>

<form>标签的属性 method 用来指明发送表单数据时使用哪种方式，有两种get, post。

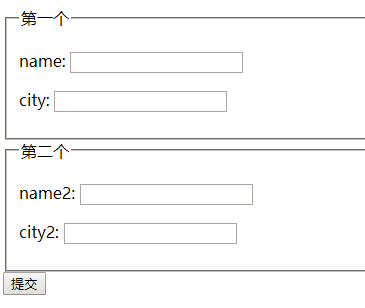
action 属性用于指明表单数据要发送到哪里，如果没有设置，则默认发送到所在HTML文档的地址。

### <fieldset>

如果表单过于复杂，需要将各个<input>收集的信息进行归类，可以使用<fieldset>标签。而<fieldset>有个子标签<legend>，是用来对这个分组提供相关说明使用。

用法

<**form method="get"**>  
 <**fieldset**>  
 <**legend**>第一个</**legend**>  
 <**p**>name: <**input name="name"**/></**p**>  
 <**p**>city: <**input name="city"**/></**p**>  
 </**fieldset**>  
  
 <**fieldset**>  
 <**legend**>第二个</**legend**>  
 <**p**>name2: <**input name="name1"**/></**p**>  
 <**p**>city2: <**input name="city1"**/></**p**>  
 </**fieldset**>  
  
 <**button**>提交</**button**>  
</**form**>



### <button>

<button>标签用来标记在表单中的按钮，但按钮的作用有三类，可通过属性值 type 来设置。如下：

|  |  |
| --- | --- |
| submit | 默认值，表示按钮的用途是提交表单 |
| reset | 表示按钮用途是重置表单 |
| button | 表示按钮是一个普通的按钮，没有任何语义 |

如果<button>标签不放在<form>标签内，那么它就需要指定绑定的是哪个<form>表单，通过 form 属性绑定 form表单的id，所以这种场景下，form表单必须设置id属性值。

### <input>

<input>标签是用于收集用户输入的标签，因此它的形态有各自各样，可通过属性type来设置。

另外，它有很多属性，每个属性都有各自的含义，一些基本的属性需要了解一下。

name 属性，用于设置该<input>的 key 值，value 值就是用户的输入，key 和 value 组合成表单中的一项用于发送给服务端。如：

<**input name="name"**/>  
<**input name="city"**/>

表明有两个<input>控件，一个 key 值为name，一个 key 值为city，收集用户输入后组成表单上传。

不同type的<input>作用不同，分别来看下：

type=”text”

默认的<input>的type值，在浏览器中呈现一个单行文本输入框。

这种类型下，还可以配合一些属性使用，如

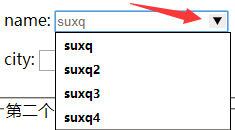
placeholder: 数据提示，类似于hint功能

list: 结合<datalist>标签使用，用于给出一系列输入提示

其他还有一些属于用于设置<input>是否可用，是否聚焦，宽度，输入最大长度等等。

示例：

<**input name="name" placeholder="suxq" list="namelist"**/>

<**datalist id="namelist"**>  
 <**option value="suxq"**/>  
 <**option value="suxq2"**/>  
 <**option value="suxq3"**/>  
 <**option value="suxq4"**/>  
</**datalist**>

type=”password”

这类型的<input>在浏览器上的呈现跟type=”text”类型一致，功能也基本一致，唯一的区别就是这是个密码框，也就是当用户输入数据时，在浏览器上是以掩饰字符替换，如···



type=”submit”

type=”reset”

type=”button”

这三种类型的用途跟<button>标签一样，所以想要设置按钮，用<input>也可以，唯一的区别就是，<input>是虚元素，也就是它没有标记任何文本内容，而<button>是可以标记文本内容的。

<**input value="提交" type="submit"**/>  
<**button**>提交</**button**>

type=”hidden”

该类型<input>浏览器会将其隐藏，不显示在网页上。通常是用于一些需要传给服务端的数据，但又没有必要让用户知道的场景下的使用。如：

<**input type="hidden" name="name\_id" value="123456"**/>

<**input name="name"**/>

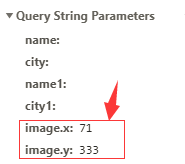
如上，用户只需输入名字信息，但提交给服务端时还需要一个name\_id信息，这个数据没必要让用户知道，此时可以通过hidden来实现。

type=”image”

该类型的<input>其实就是个图片按钮。跟<img>标签的区别也就是，这个是可点击的，<img>只是将图片嵌入进来

用法

<**input type="image" src="https://upload-images.jianshu.io/upload\_images/5687349-d9d7ce1fec758d8c.png"**/>



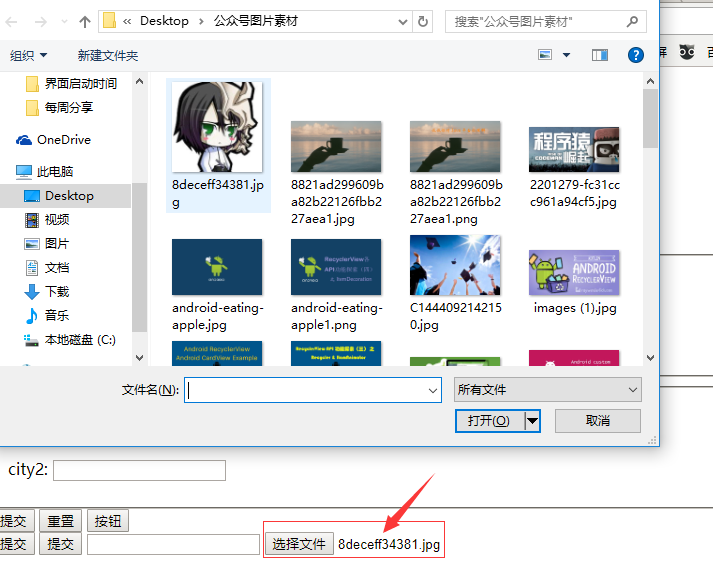
当在 form 表单中有 type=”image”类型的<input>时，点击这张图片，发送给服务端的信息是点击的坐标点。

type=”file”

该类型的<input>可以让用户选择本地文件。

用法

<**input type="file" name="filedata"**/>



type=”checkbox”

复选框

是<**input type="checkbox"**/>

type=”radio”

多选框，通常配合<fieldset>一起使用，将同类型的集合管理在一起。当然，不用也可以。

<**fieldset**>  
 星期一<**input type="radio" value="星期一" name="time"**/>  
 星期二<**input type="radio" value="星期二" name="time"**/>  
 星期三<**input type="radio" value="星期三" name="time"**/>  
</**fieldset**>

type=”number”

type=”email”

type=”tel”

type=”url”

type=”color”

…

这类type的<input>，用途只是用于限定用户的输入格式，比如number，这个输入框就只能输入数字。

## <iframe>

HTML文档中是可以嵌入其他HTML文档的，通过<iframe>标签标记。

用法

<**iframe src="index.html" width="500" height="500"**></**iframe**>

通过src属性设定目标HTML文档地址，width,height设置区域大小。



## <img>

基本每个网页都会有图片，在HTML中嵌入一张图片用<img>标签，跟Android中的ImageView控件很类似，同样需要指定图片来源，区域宽高。

用法

<**img src="https://upload-images.jianshu.io/upload\_images/5687349-d9d7ce1fec758d8c.png" width="300" height="300"**>

# 标签3-容器类

容器类标签是我自行对其进行的划分，并不是说，这类标签只能用于当容器使用，只能用于包含其他标签，而是说，常见的用法，这类标签基本都是表示具有某种含义的某一块区域，具体这块区域内既可以包含各种标签，也可以直接是文本内容。

## <span> & <div>

都是通用标签，没什么具体的语义

<span>标签通常用于标记段落中的某块文本内容，然后通过该标签，可以单独为这块文本内容增加CSS样式

<div>标签一般用于，将屏幕某块区域划分出来后，用该标签标记后可通过选择器作用自定义的CSS样式。

## <section>

<section>标签用于表示文档中的某一节，其实也就是某一块区域，这块区域有自己单独独立的含义。通俗的理解，有些类似于第一节，第二节的概念。

每一节<section>都是相互独立的，因此方便各节里面独自使用<headler>和<footer>



## <header> & <footer>

<header>标签表示某一节的首部，像某些网页通常会有一些 Logo 之类的首部信息。

<footer>标签表示某一节的尾部，最常见的基本每个网页尾部都会有版权信息，作者介绍，相关链接，免责声明等信息，这部分信息都适用于放在尾部标签<footer>里。

例如：<header>

<footer>



## <nav>

<nav>标签表示文档中某一个区域，它包含着到其他页面或者同一页面的其他部分的链接。

直译其实也就是导航的作用。



# 标签文档

<https://developer.mozilla.org/zh-CN/docs/Web/HTML>

# CSS结构和语法

首先需要清楚，CSS职责是控制HTEM文档里的文本内容在网页上样式呈现的效果，写的每一个样式最终是通过选择器作用到具体的元素上面。

## E:\我的图片\QQ图片20180720155515.png工作原理

互联网其实都是通过网页相互连接组成的，每个网页都是一份HTML文档，因此浏览器与后端的首次通信是以HTML文档为基准，那么整个流程的第一步也就是解析HTML文档。

如果在解析HTML文档过程中发现有通过<link>标签引用了外部的CSS文件时，那么浏览器会去下载相对应的CSS文件。

当HTML文档解析完毕后会生成一个DOM文档结构，DOM文档结构中记录着每个节点的元素，各元素之间的关系，有点类似于Android中的View树。

最后，通过CSS的选择器将相对应的样式作用到DOM中选择器找到的元素节点，然后浏览器渲染呈现在网页上。

## CSS 语法结构

以上是一个CSS的典型结构，总共由两部分组成：选择器 + 样式

{} 大括号内部的都是具体的某种样式，可用来控制元素的背景、大小、排版位置等样式效果。而 {} 左边的则是选择器，用来指定说后面跟随的样式列表要作用到HTML文档中的哪个元素上。

选择器类型很多，规则也很多，因此会有一种现象，就是某个元素被不止一个选择器匹配到，如果某个样式属性起冲突了，那么就会按照一定的优先级顺序处理。

样式属性也很多，具体也后面介绍，但有一点需要先明确一下，如果使用了未知的样式属性，或者给某个样式属性赋予了无效值，那么该样式属性会被视为无效，浏览器的CSS引擎会完全忽略它。

## https://mdn.mozillademos.org/files/13647/box-model-standard-small.png盒模型

跟Android很类似，每个元素在页面上都是占据一个矩形区域，也分margin和padding，唯一不同的就是这里的模型多了个border，因此，一个元素在页面上的宽高就是由四个部分共同影响的了：content区域，padding区域，border区域，margin区域。

需要注意的是，上图中的盒子模式的 box-sizing 属性为默认值 content-box 模式。

这种模式下，width 和 height 指的只是content区域的宽高。

box-sizing属性的取值有：content-box, padding-box, border-box, margin-box。四种，对应的就是指明 width 和 height 表示的是包含哪些区域的宽高。

## 使用方式

CSS基本结构是由选择器和样式属性列表组成，那么如何跟HTML文档关联起来使用呢？一共有三种方式：

内联样式

使用HTML元素的全局属性style声明，仅影响一个元素，除非工作环境 受限，比如只允许编辑HTML的body，否则强烈不推荐这种方式。

示例：

<**a href="index.html" style="background**: **gray**; **color**: **#6a90d9"**>点击跳转</**a**>

内部样式表

在HTML中的 <head> 内使用 <style> 标签，在某些情况下很有用，比如当你不能直接修改CSS文件时。

示例：

<style type="text/css">  
 a {  
 background: gray;  
 color: #6a90d9;  
 }  
</style>

外部样式表

将CSS 保存在一个独立的扩展名为 .css 的文件中，并在HTML的<head> 里使用 <link> 元素中引用它，这种方法可以说是最好的，因为你可以使用一个样式表来设置多个文档的样式，并且需要更新 CSS 的时候只要在一个地方更新。

<**link type="text/css" rel="stylesheet" href="css/nms.css"**>

# CSS选择器

## 基本选择器

基本选择器其实是一些比较常用、简单的选择器，包括：元素选择器、id选择器、class选择器、属性选择器、组合选择器。示例介绍见[基础-CSS一节](#_CSS)

## 多规则选择器

基本选择器规则很简单，选择器基本就是一两个条件，满足了即可匹配上，如 a.class，p#id等，即使有稍微经过组合，但仍旧不复杂，但有些应用场景下需要通过复杂的规则，即需要满足多个条件下才能匹配上。

属性选择器

在基本选择器一节中介绍的属性选择器其实是最基本用法的一种，而它还支持其他很多规则的用法，如下：

|  |  |
| --- | --- |
| [attr] | 选择定义attr属性的元素，不管属性的取值具体是什么 |
| [attr=”val”] | 选择定义attr属性，且属性值为val的元素 |
| [attr^=”val”] | 选择定义attr属性，且属性值以字符串val打头的元素 |
| [attr$=”val”] | 选择定义attr属性，且属性值以字符串val结尾的元素 |
| [attr\*=”val”] | 选择定义attr属性，且属性值包含字符串val的元素 |
| [attr~=”val”] | 选择定义attr属性，且属性值具有多个值，其中一个为字符串val的元素。 |
| [attr|=”val”] | 选择定义attr属性，且属性值为连字符(-)分割的多个值，其中第一个为字符串val的元素。 |

也就是说，属性选择器不仅可以用来匹配那些具有指定属性的元素，还可以进一步根据不同属性值来匹配。

并集选择器

结构：<选择器>, <选择器>

并集选择器是通过 ， 逗号将不同选择器组合使用的一种选择器，这种情况下，各个选择器之间是没有任何关系，都是相互独立的，就是他们具有相同的样式属性表而已。

这只是一种简便写法的用法而已，具有相同样式属性表的不同再复制粘贴，可以直接通过 ，逗号将不同选择器分开即可。只有HTML文档中的元素满足其中一个选择器即可。

示例：

**a**, **h1**, **span**, **div** {  
 **background-color**: **black**;  
}

上述示例中有四个选择器含有同样的样式属性，HTML文档中只要满足其中一个选择器即可被匹配到。

后代选择器

结构：<第一个选择器> <第二个选择器>

多个选择器之间通过”空格”分隔开的话表示这是一个后代选择器，也就是说，需要先满足第一个选择器的前提下，在第一个选择器匹配到的元素的后代元素中去匹配第二个选择器。

这里的后代包括了子孙后代。

示例：

**p span** {  
 **background-color**: **black**;  
}

<**span**>第一个span</**span**>  
<**p**>第二个<**span**>span</**span**></**p**>

以上示例中，选择器要匹配的元素是位于 p 元素的后代元素中的 span 元素，因此第一个 span 元素就不符合规则，而第二个 span 则会被匹配到。

儿子选择器

结构：<第一个选择器> > <第二个选择器>

儿子选择器是多个选择器之间通过”>”右箭头符号连接，表示的是在满足第一个选择器的前提下，从它匹配到的元素的直接子元素中寻找第二个选择器。

跟后代选择器的区别就在于它只能在直接子元素中匹配第二个选择器。

示例：

**p > span** {  
 **background-color**: **black**;  
}

<**span**>第一个span</**span**>  
<**p**>第二个<**span**>span</**span**></**p**>  
<**p**><**div**>第三个<**span**>span</**span**></**div**></**p**>

第一个span元素不是p元素的后代，第二个span元素是p元素的直接子元素，第三个span元素是p元素的孙子元素，因此只有第二个span元素满足规则被匹配到。

兄弟选择器

结构：<第一个选择器> + <第二个选择器>

兄弟选择器是多个选择器之间通过 “+” 加号连接。表示的是，在满足第一个选择器的前提下，从它匹配到的元素的紧跟着的位于同一层级的下一个元素，看该元素是否符合第二个选择器。

也就是说，兄弟选择器，两个选择器所匹配的元素要求，位于同一层级，且相邻。

示例：

**p** + **a** {  
 **background-color**: **#6a90d9**;  
}

<**a**>第一个a</**a**>  
<**p**>第二个<**a**>a</**a**></**p**>  
<**a**>第三个a</**a**>  
<**a**>第四个a</**a**>

上述示例中，同时满足位于同一层级，且相邻，且需要先满足第一个选择器的前提下，还满足第二个选择器这四个条件的 a 元素就是第三个 a 元素了。

普通兄弟选择器

结构：<第一个选择器> ~ <第二个选择器>

普通兄弟选择器，是多个选择器之间通过 “~” 波浪符号连接。表示的是满足第一个选择器的前提下，从它匹配到的元素后，去寻找位于同一层级，且在该元素后面的所有满足第二个选择器的元素。

兄弟选择器只匹配相邻的一个元素，而普通兄弟选择器则是可以匹配位于元素后面的所有符合第二个选择器的元素。

示例：

**p** + **a** {  
 **background-color**: **#6a90d9**;  
}

<**a**>第一个a</**a**>  
<**p**>第二个<**a**>a</**a**></**p**>  
<**a**>第三个a</**a**>  
<**a**>第四个a</**a**>

同时满足同层级，且在p元素后面的兄弟元素有两个，第三个a元素和第四个a元素，因此这里可以匹配到这两个元素。

## 伪选择器

选择器的目的就是为了匹配到HTML文档中的满足条件的元素，然后将样式属性作用在元素上。

元素是什么，在基础一节中有介绍过，元素其实就是包含了标签以及文本内容的整块内容。因此被选择器匹配到的元素，都是直接将CSS样式作用到整个元素上，也就是作用到整个文本内容上。

那么，如果有一些需求并不是直接去匹配HTML文档中的具体元素，而是指定了一些状态、行为，然后让浏览器动态去根据当前情况选择符合这些状态、行为的元素。

或者有一些需求是并不想将CSS样式作用到整个元素上，而是只作用到元素标记的文本内容的某一部分。

这个时候，这种选择器就称作**伪选择器**，因为它有区别于普通选择器的行为

伪选择器总共分成两种：伪元素选择器，伪类选择器

伪元素选择器

当伪选择器最终将CSS作用的对象并不是整个元素，而是满足条件的元素标记的文本内容的某一部分时，称伪元素选择器。

伪元素选择器不多，如下：

|  |  |
| --- | --- |
| ::first-line | 匹配满足条件的元素标记的文本内容的首行部分 |
| ::first-letter | 匹配满足条件的元素标记的文本内容的首字母部分 |
| :before | 在满足条件的元素之前**插入**生成的内容 |
| :after | 在满足条件的元素之后**插入**生成的内容 |

伪元素选择器的用法基本都是和其他选择器组合使用，比如 p::first-line 表示匹配p元素标记的文本内容的首行部分。

而 :before 和 :after 与之前的选择器都不大一样，因为之前介绍的选择器作用都只是用于匹配选择HTML文档中的元素或文本内容而已。但这两个伪元素选择器会生成内容，并插入到匹配到元素的文本内容中去。

因此，它们的基本用法通常都是这样：

**a**:**before** {  
 **content**: **"在文本内容之前插入"**;  
}  
**a**:**after** {  
 **content**: **"在文本内容之后插入"**;  
}

有一种应用场景很适合使用这两种伪元素选择器，当需要对列表动态的生成复杂的编号规则时，可以结合:before和counter()使用。在CSS中也是可以使用一些内置的方法功能。

伪类选择器

当不是通过HTML文档中元素的一些基本性质，比如id，class，标签名，属性这些基本特征来匹配这些元素时，就可以称作伪类选择器。

伪类选择器是通过满足一些指定状态、行为下来匹配元素的一种选择器，比如满足是否获取焦点等等。

伪类选择器相对来说，比较多，如下：

|  |  |
| --- | --- |
| :first-child | 选择元素的第一个子元素 |
| :last-child | 选择元素的最后一个子元素 |
| :only-child | 选取元素的唯一一个子元素 |
| :only-of-type | 选取属于父元素的特定类型的唯一子元素 |
| :nth-child(n) | 选取元素的第n个子元素 |
| :nth-last-child(n) | 选取元素的倒数第n个子元素 |
| :nth-of-type(n) | 选取属于父元素的特定类型的第n个子元素 |
| :nth-last-of-type(n) | 选取属于父元素的特定类型的倒数第n个子元素 |
| :enabled | 选取启用状态的元素 |
| :disable | 选取被禁用状态的元素 |
| :checked | 选取所有选中的复选框和单选按钮元素 |
| :default | 选取默认的元素 |
| :valid :invalid | 选取基于输入验证判定的有效或者无效的input元素 |
| :in-range :out-of-range | 选取被限定在指定范围之内或之外的input元素 |
| :required :optional | 根据是否允许使用required属性选取input元素 |
| :link | 选取未访问的链接元素 |
| :visited | 选取用户已访问的链接元素 |
| :hover | 选取鼠标指针悬停的元素 |
| :active | 选取当前被用户激活的元素，这通常意味着用户即将点击该元素 |
| :focus | 选取获得焦点的元素 |
| :not(<选择器>) | 否定选择，（如选择所有不匹配<选择器>的元素） |
| :empty | 选取不包含任何子元素或文本的元素 |
| :lang(<语言>) | 选取lang属性为指定值的元素 |
| :target | 选取URL片段标识符指向的元素 |

一些伪类选择器看下说明应该就清楚怎么使用，不明白的再具体去查找相关文档即可。

# CSS层叠算法

由于一个元素通常会被多个选择器命中，而这些选择器又有可能是通过不同方式作用到元素上，因此这里存在了两种场景下的优先级问题，但请记住，只有当作用到同一个元素上的样式属性起了冲突时才会存在优先级问题。

如果不同选择器作用到同一个元素上，但它们各自的样式属性列表中没有重复的，那就不存在冲突，也就不存在优先级问题，都会一起合并作用到元素上。

场景1：不同使用方式的优先级

CSS有三种使用方式，另外浏览器也有默认样式，因此这些构成了一个优先级顺序：

1. 元素内嵌样式（全局属性style定义的样式）

2. 文档内嵌样式（style标签定义的样式） 和 外部样式（link标签引入的外部CSS文件）

3. 浏览器中的用户样式

4. 浏览器中的默认样式

以上优先级从高到低，同层级之间，如果存在冲突的样式属性的话，以文档中最后出现的属性为准，采用覆盖规则。

场景2：不同选择器之间的优先级

当作用到同一个元素上的不同选择器存在样式属性冲突时，优先以场景1考虑优先级，如果不存在场景1的情况，即起冲突的选择器在场景1中处于同一层优先级，那么才会考虑不同选择器之间的优先级。

1. id选择器

2. class选择器，属性选择器，伪类选择器

3. 元素选择器，伪元素选择器

以上优先级从高到低，同层级之间，如果存在冲突的样式属性的话，以文档中最后出现的属性为准，采用覆盖规则。

但，有时候，使用的是组合选择器，那么这时候就需要依靠一定的算法来计算出谁的优先级高了，这个算法叫做层叠算法：

通过对以上不同选择器赋予某个数值来计算整个组合选择器的最终数值，然后比较大小。

比如，上面三个优先级的选择器中，1优先级的表示100，2优先级的表示10，3优先级的表示1，以此来计算一个组合选择器的数值大小。

通常来说，组合选择器不会过于离谱，长达十几个选择器的组合，因此以上述赋予每个优先级的数值足够覆盖绝大多数场景。

示例：

h1，和 #indentifier 都只是简单选择器，不通过这种层叠算法计算也行。

h1 + p::first-letter 和 li > a[href\*=”zh-CN”] > .inline-warning 这种比较复杂的组合选择器，就可以根据赋予每一层级优先级对应的数值来进行计算，最终根据数值大小比较谁的优先级高。

这种赋予不同优先级某个具体数值具现化的思想叫做层叠算法，通常是用于比较复杂的组合选择器时。

实操

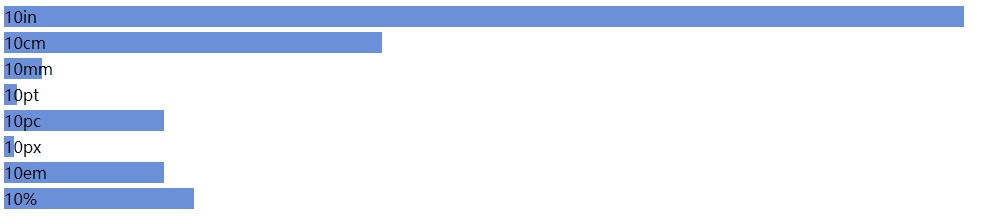
但实际开发中，很少会需要用到层叠算法，掌握场景1和场景2下简单的优先级分辨理论基础足够了。实际开发过程中，没必要这么复杂，借助开发工具或者运行查看下效果就可以确认谁的优先级高低了。

# CSS属性

## 单位 in cm mm px em %

绝对单位：1in（英寸） = 2.54cm（厘米） = 25.4mm（毫米） = 72pt（英镑points） = 6pc（皮卡picas）

相对单位：px（像素） em（相对于font-size的大小） %（百分比）



## 字体 font-xxx

|  |  |
| --- | --- |
| **font-size: 20px;** | **字体大小** |
| **line-height: 30px;** | 行高 |
| **font-family: 微软雅黑;** | 字体 |
| **font-style: italic;** | 斜体，normal正常字体 |
| **font-weight: bold;** | 粗体 |
| **font-variant: small-caps;** | 小写变大写 |

font-xxx, line-xxx 属性具有继承性，后代会继承祖先标签内的这些属性

## 文本 text-xxx

|  |  |
| --- | --- |
| **letter-spacing: 0.5cm;** | **单个字母之间的间距** |
| **word-spacing: 1cm;** | 单词之间的间距 |
| **text-decoration: none;** | 字体修饰，none:去掉下划线，underline下划线，line-through中划线，overline上划线 |
| **text-transform: lowercase;** | 单词小写，uppercase大写, capitalize首字母大写 |
| **color:red;** | 字体颜色 |
| **text-align: center;** | 在当前容器中对齐方式，left,right,justify |

text-xxx具有继承性，后代标签会继承祖先中声明的这些属性

想让文本居中显示时，如果属性不生效，可自行计算，如下：

## https://camo.githubusercontent.com/9ccf407b1df4e1abe207eaec71a655e99a21d5e0/687474703a2f2f696d672e736d79687661652e636f6d2f32303137303830385f323234302e706e67背景 background-xxx

background-color

设置元素背景颜色，属性值有三种方式：red, rgb(255, 0, 0), #ff0000

以上三种均表示红色。以下是几种常见的颜色：

#000（黑） #fff（白） #f00（红） #222（深灰）#333（灰）#ccc（浅灰）

background-repeat

设置背景图片是否重复，以及如何重复，默认平铺满。属性值如下：

no-repear 不要平铺

repeat-x 横向平铺

repeat-y 纵向平铺

应用场景：可以类似于Android中的.9图，设计一张1px的背景图，指定横向或者纵向平铺。

background-position

指定背景图位置，几种格式如下：

background-position:向右偏移量 向下偏移量;

属性值可以是正数，也可以是负数。比如：100px 200px，-50px -120px

background-position: 描述左右的词 描述上下的词;

描述左右的词：left、center、right

描述上下的词：top 、center、bottom

比如说，right center表示将图片放到右边的中间；center center表示将图片放到正中间。

这个属性既可以用于在一张包含各种icon种只显示指定区域的icon，也可用于在文本四周添加icon。

background-attachment

设置背景图片是否固定，属性值：

fixed //背景固定住，不会被滚动条滚走

scroll //默认属性，背景跟随滚动条滑动

background-clip

背景默认延伸到边界的外边缘，如果你只想背景扩展到内容区的边缘，可通过这个属性设置。

outline

设置盒子的轮廓，它看起来像边界，但不是盒模型的一部分，是画在边界框之外，外边距之内，但不会修改盒子的大小。

background

以上属性的综合属性，如果不想一个个属性的去写，可以用这个属性集中写在一起。格式如下：

background: -color –image –repeat –position –attachment

如：background:red url(1.jpg) no-repeat 100px 100px fixed;

等效于：

background-color:red;

background-image:url(1.jpg);

background-repeat:no-repeat;

background-position:100px 100px;

background-attachment:fixed;

另外，背景是可以设置多个的，多个背景间会自动重叠在一起，并不是像Android中只能设置一个背景。

## 盒子 width height margin padding border&…

width height

设置内容的宽高，并不是盒子的宽高，但可通过 box-sizing来更改宽高的作用域。

Android中必须给控件设置宽高，但在这里，宽高并不是必选像。当没有设置宽高时，会根据其显示模式display来决定其默认宽高。

比如，display: block块级元素默认高度会霸占父节点100%宽度，而高度默认会由子内容决定，类似于Android中的wrap\_content。

display: inline行内元素则是无法设置宽高，默认为文本内容的宽高。

除了正常宽高外，还有其他一些可选项配置：

min-width, min-height, max-width, max-height

借助这些配置，可以达到一些当窗口大小变化时，变化到一定程度改变原有元素的表现行为，比如某张图片本来居中显示，但当窗口缩小到比图片还小时，图片就根据窗口进行缩小，此时就可结合max-width来实现。

padding

padding：-top –right –bottom -left内边距

border

border：-width –style –color 边框

margin

margin：-top –right –bottom -left外边距

注意一点，margin跟Android中的margin有些不一样，对某个元素设置了margin后，margin区域的大小也算在这个元素盒子的大小中。

另外，默认的文档流中，上下方向会存在margin塌陷的情况，也就是上一个元素设置了margin-bottom: 10px，下一个元素设置了margin-top: 20px，那么最终这两个元素其实间距为20px，并不是叠加的。

另外，margin通常是用于兄弟节点间设置间距，如果要设置儿子和父亲间的间距，建议使用padding，否则如果父节点没有设置border时，可能效果并不是想要的。

margin: 0 auto; 可以达到水平居中的效果。

<body>标签，浏览器默认给了 margin: 8px

<ul>默认的margin-left 和 padding-top

## 阴影 box-shadow text-shadow

你会看到，我们在box-shadow属性值中有4个项：

第一个长度值是水平偏移量（horizontal offset ）——即向右的距离，阴影被从原始的框中偏移(如果值为负的话则为左)。

第二个长度值是垂直偏移量（vertical offset）——即阴影从原始盒子中向下偏移的距离(或向上，如果值为负)。

第三个长度的值是模糊半径（blur radius）——在阴影中应用的模糊度。

颜色值是阴影的基本颜色（base color）。

如果需要内部阴影，则是在上述属性值最前面加一个 inset 如：

box-shadow: inset 2px 2px 1px black

## 显示模式 display

标准文档流布局方式

当没有进行任何CSS控制元素的排版布局时（float, position, flex）默认就是按照标准的文档流布局方式进行排版布局绘制元素。

而这个标准的文档流布局方式就是按照解析HTML文档元素的顺序，从页面顶部开始，从上到下，从左到右，解析一个元素就绘制一个元素，解析时碰到的是行内元素，就忽略宽高，允许并排布局绘制，碰到的是块级元素，就另一起一行，让这个元素霸占父元素中的这一整行。

所以，清楚了浏览器默认的文档流布局方式后，得再了解下所说的行内元素，块级元素是什么，怎么切换。

inline（行内元素） block（块级元素）

display有两个很基本的属性值：inline（行内元素） block（块级元素）

通常，容器类的标签默认值都是block，而文本类的标签默认值是inline。

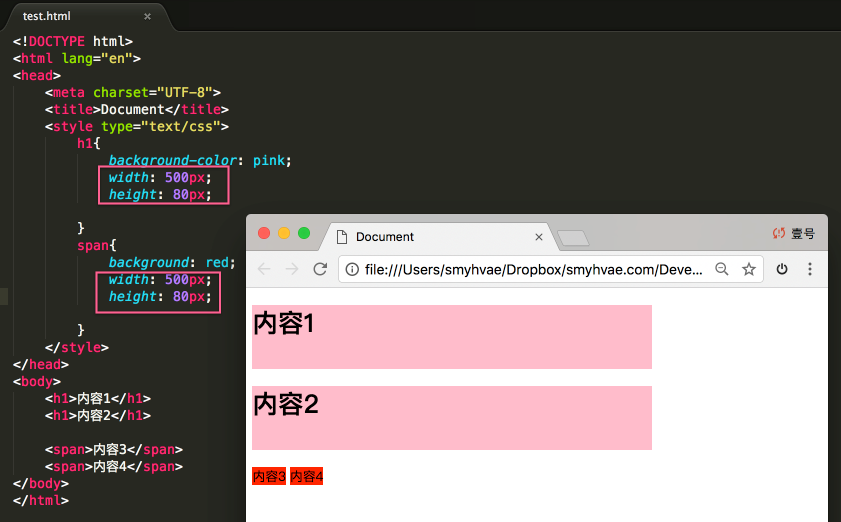
比如：div, header, main, li, h系列这类都是块级元素

p, span, I, a这类都是行内元素。

区分以及理解行内元素和块级元素对于写网页布局非常重要，因为浏览器是按照文档流从上到下，从左到右来进行绘制网页的。

对于行内元素（inline），浏览器绘制时会忽略对它设置的宽高，并且如果相邻两个元素都是行内元素，则直接以并排方式从左到右对其进行布局绘制。

但对于块级元素，浏览器会强制让其霸占一整行，也就是这一行内，只能有这个块级元素存在，其他元素不能与其并排。如果没有设置宽度，那么就是充满整行。

这个整行是想对于父元素的区域而言，并不是浏览器页面的整行。

这种默认的标准文档流的布局绘制方式有些类似于Android中的LinearLayout容器，inline就类似于水平方向，block就类似于垂直方向，同一个方向内只能有一个元素霸占，不允许并排。

所以，我们在写HTML，CSS时，脑中就要有个大概的蓝图，这些元素大概会呈现怎样的排版布局。

一个元素是行内元素还是块级元素，可以通过display来设置，如果没有设置，那么就是默认值，不同元素的默认值不同。

如果行内元素不支持设置宽高，块级元素又不允许并排存在，那么很显然，很多布局我们就实现不了了。是吧，如果既要能够并排，还要支持可以设置宽高，这是该怎么办呢？

解决方法有很多种，说白了就是一点：脱离默认的文档流布局方式

方式有以下几种：

* display:inline-block（行内块元素）
* 浮动 float: left/right
* 绝对定位 position: absolute
* 固定定位 position: fixed

inline-block（行内块元素）

当设置了display: inline-block 时，这时这个元素就不会霸占一整行了，而是根据设置的宽高来布局绘制，因此这种情况下，可以解决并排的场景。

但需要注意下，最好不要有这样的布局：

<div style=”display:inline-block”>

<div style=”display:block”>

<div style=”display:inline-block></div>

</div>

</div>

也就是行内块元素嵌套了块级元素内部又嵌套了行内块元素，这样的话，布局方面会有些问题，原因不清楚。但把中间的块级元素也设置成行内块元素就没问题了。

另外，并不是说，不允许行内块元素内嵌块级元素，也是可以的。

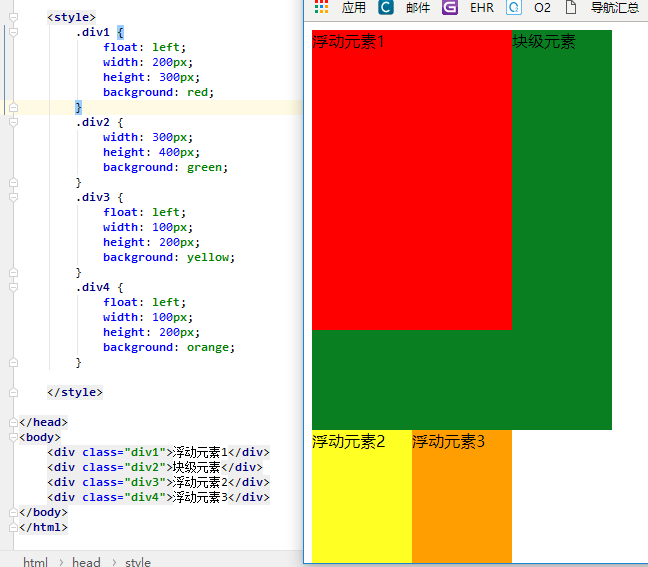
## 浮动 float

float属性值通常会用到这两个：left right

浏览器默认是按照标准文档流从上到下，从左到右布局绘制各个元素，此时这些元素可以说位于同一个层面，但当碰到元素设置了float属性时，会将这个元素以当前绘制的位置抽离到新的层面上进行布局。

就像单词直译过来：浮动

也就是让这个元素浮在标准文档流上面。

浏览器绘制这个网页时，按照文档流顺序，先在网页第一行左边开始处理div1元素，发现它的一个浮动元素，则将其抽离到另一个层面，浮动在文档流上面。

然后它继续处理div2元素，因为之前处理的div1元素是浮动元素，不占用文档流，所以此时仍旧是在第一行左边绘制div2元素，发现它是一个块级元素，所以让其霸占一整行。

所以，此时可以看到div1浮在div2上面的效果。

紧接着，继续处理div3元素，因此之前div2块级元素已经霸占了第一行了，所以此时是在第二行处理div3元素，发现它也是一个浮动元素，便以当前位置将其抽离到另一层面绘制。

所以此时看到的效果就是，浮动元素div3是在块级元素div2下。

继续往下处理，如果发现的还是浮动元素，因为所有的浮动元素都处理同一层面，所有的文档流元素都处于一个层面，所有浮动元素div4并不会跟div3发生重叠，而是贴着它。

如果接下去处理的元素是块级元素，那么此时的效果会是怎样的呢？

如果接下去是块级元素，那么它就会是绘制在浮动元素div3和dive4下面，呈现出重叠的效果。

但这里需要注意一点，**虽然浮动元素会造成重叠的现象，但这只是正常的文档流的元素盒子被浮动元素压住了，但文档流元素的文本内容会自动围绕在浮动元素周围，就像上图中块级元素四个字并没有被覆盖住**。

也就是说，浮动元素并不会造成文档流的内容被覆盖的情况，反而它会影响到它之后文档流中元素的内容区域的显示排版。如果不想让后面的元素受到浮动元素的影响，那么就要进行浮动清除处理。

另外，不考虑嵌套元素的话，兄弟元素之间，如果处于同一层面，是不会有重叠现象的。

浮动清除

由于浮动最初设计是为了让文字可以围绕在图片周围，因此，浮动元素后面的非浮动元素会自动围绕着浮动元素进行包装。而如果我们想让浮动元素之后的元素另起一行，从新的位置开始布局，那么就要进行浮动的清除。

另外，浮动元素之后的浮动元素也会受到它的影响，比如设置了float:left，那么这个元素要浮动的靠左的位置会受到前面已经靠左浮动的元素的影响。

如果不想让当前的浮动元素受到之前浮动元素的影响，那么也要进行浮动清除的处理，方式有几种，每种有自己的适用场景，了解下即可，后续写多了，自然就懂了。

* clear: both 表示当前元素不受之前浮动元素的影响
* 隔墙法（在两部分浮动元素中间，建一个墙。隔开两部分浮动，让后面的浮动元素，不去追前面的浮动元素。本质上也是clear:both）

浮动的不足

浮动虽然好用，既可以实现文字围绕的效果，又可以实现多列并排的布局，但它也存在一些不足的地方，上面说的浮动清除是其中一点，还有一些问题，如下：

1. 整个宽度可能难以计算

这是因为多个浮动元素之间并排显示的前提的有足够的空间让这些元素并排，所以**通常对于浮动元素的宽度设置是通过百分比来设置**，确保多个并排的元素即使窗口发生变化仍旧可以并排布局。

但，如果元素还需要进行内边距，外边距的设置，边框的设置，因为这些大小都算在盒子的总宽度中，那么最终盒子的大小就变得很难确定，有可能导致某个浮动元素被挤到下一行去。

有个方法可以解决，修改box-sizing:border-box，让width就是盒子总宽度，当设置了边距时，会自动减少相应的内容区域。但如果页面使用不同类型的box-sizing，会使CSS的代码阅读变得很杂乱。

2. 浮动元素之后的元素设置margin失效

非浮动元素的外边距不能用于它们和浮动元素之间来创建空间，通常我们再浮动元素之后的非浮动元素会进行浮动清除，顺便设置外边距来创建间隔空间，但这时会发现这个间隔空间失效。解决方法可以在这中间插一个空的元素，在这个元素里清除浮动，也就是所说的隔墙法。

## 位置 position

position 属性值有三个：absolute, relative, fixed

三种虽然是不同的定位模式，但其实说白了，就是参考点不同。

也都是通过 left, top, right, bottom 来根据参考掉调整位置。

releative 相对定位

相对定位并不是相对于父元素，而是相对于该元素原本所应该在的位置作为参考点。

这点跟Android中的ReleativeLayout布局不一样，需要注意一下。

另外，相对定位并不会改变元素在文档流中的位置，也就是这个元素原本占据哪个坑，通过相对定位微调之后，仍占据那个坑，只是视觉上它发生了移动而已。有点类似Android中的View动画。

应用场景：

* 微调
* 让后代元素可以使用绝对定位

通常都是用来给后代使用绝对定位的，因为固定定位和绝对定位都会导致该元素从文档流中脱离，就像浮动元素那样，不再占用文档流的坑位。

而相对定位并不会，所以通常父类元素设置了相对定位，而让后代元素使用绝对定位，这样可以让后代元素可以脱离文档流，达到压盖的效果，同时还可以受限于父类元素的范围管控。

absolute 绝对定位

参考的对象是父类或祖先元素中有使用了position的最近一个元素。也就是说在父类元素中，不管是使用了相对定位，绝对定位，固定定位之后，都可以用来作为后代绝对定位的参考元素。

但通常都是使用子绝父相的模式，其他模式并没有什么意义。

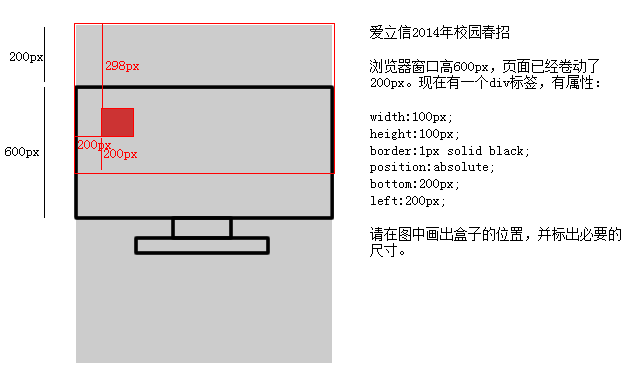
参考点

参考的元素决定了之后，参考点的选取还分两种情况，参考元素的左上角或者左下角。

如果使用了 top 来调整位置，那么参考点就是参考元素的左上角

如果使用了bottom来调整位置，那么参考点就是参考元素首屏页面的左下角

为什么强调首屏，因为参考元素的大小可能是超出一个屏幕的，bottom并不是说参考元素的左下角坐标，也不是当前页面参考元素的左下角坐标，而是首屏状态时，也可以理解成，没有发生滑动前，参考元素在当前页面的左下角坐标作为参考点。

举个例子：

另外，有点需要注意下，绝对定位的元素，因为其已经从文档流中抽离，因此就不存在什么行内元素、块级元素的限制了。大小就是根据设置的宽高、位置就是根据参考点进行调整后进行布局绘制。

而如果没有使用position属性，元素默认是文档流处理的，此时的布局绘制方式就只能按照文档流的规则来，即行内元素、块级元素这些特性。

因为绝对定位是将元素脱离出标准文档流，那么绝对定位的元素显示与否，并不会影响到原本的文档流元素，所以，通常一些弹窗框，弹窗控制面板，可在页面上任意拖放的元素等都会通过绝对定位来做。

应用：

* 弹窗框
* 压盖（角标之类）

fixed 固定定位

固定定位参考点就是浏览器的左上角，不管页面如何发生滑动，元素显示的位置都没有发生改变。

应用：

* 网页右下角的返回顶部按钮
* 顶部导航栏

z-index

这个属性只有当使用了position的元素才会生效，其他元素设置了这个属性没有什么意义。

这个属性其实就是用于当元素发生重叠时，决定由谁盖在上面，默认值为0，值越大，越上层。

而会发生元素重叠的现象也就只有使用了position调整了元素的位置，以及浮动元素两种场景。正常的文档流方式布局绘制元素是不会出现元素重叠，当然如果是嵌套的元素，层级关系也早就确定了，也没必要通过这个属性来调整了。

浮动元素造成的重叠只是盒子上的重叠，并不会造成元素内容上的重叠，那么也就没有使用z-index的必要，因为要呈现的内容并不会被覆盖。

总结一下，这个属性有几个特性：

* 属性值大的位于上层，属性值小的位于下层
* z-index值没有单位，就是一个正整数。默认的z-index值是0
* 如果大家都没有z-index值，或者z-index值一样，那么在HTML代码里写在后面，谁就在上面能压住别人。定位了的元素，永远能够压住没有定位的元素
* 只有定位了的元素，才能有z-index值。也就是说，不管相对定位、绝对定位、固定定位，都可以使用z-index值。而浮动的元素不能用
* 从父现象：父亲怂了，儿子再牛逼也没用。意思是，如果父亲1比父亲2大，那么，即使儿子1比儿子2小，儿子1也能在最上层。

## 弹性布局 flex

弹性布局的作用有点儿类似 Android 中 LinearLayout 和 RelativeLayout 两者的合成版，即支持横向布局，纵向布局，start，end，center布局，宽高按比例瓜分等等，当然它还有很多其他功能，比如自动换行，按指定 order 排列等。总之有了 Android 基础，理解弹性布局 Flex 蛮容易的。

可以这么的理解，传统的网页布局方式是通过 display 和 position 以及 float 三者完成的，借助块级元素，行内元素特性，结合 position 指定的相对布局、绝对布局、固定布局方式来实现各种排版效果。如果需要浮动，则借助 float。

但这种传统的方式，一来使用较复杂，二来某些排版效果不好实现，如列表、居中、响应式布局等效果。

而 flex 则能够很好的完成传统的布局工作，而且，它还可以支持响应式布局。

如何使用

在 Android 中，不同控件系统已经封装成不同的对象，但在前端里，大部分标签都可作为容器标签，那么，要使用弹性布局 flex 的特性，就只能通过属性来声明：

**display**: **flex**;

当对块级元素使用了这个 css 属性之后，那么这个块级元素将被作为弹性容器处理，既然是容器，自然有孩子，它的子孙后代的元素就都被作为弹性项目处理（flex item在很多地方被翻译成弹性项目，但如果有 Android 基础，我更倾向于将它翻译成弹性子项，类比 Android 来理解）。

在 Android 中对这些容器，子项排版都是通过各自属性来控制的。那么，一样的道理，当声明了某个标签作为弹性盒子后，就可以使用某些属性来作用于弹性盒子或者它的子项上面。

将属性之前，先来看张 flex 布局原理的图：

弹性盒子的属性：

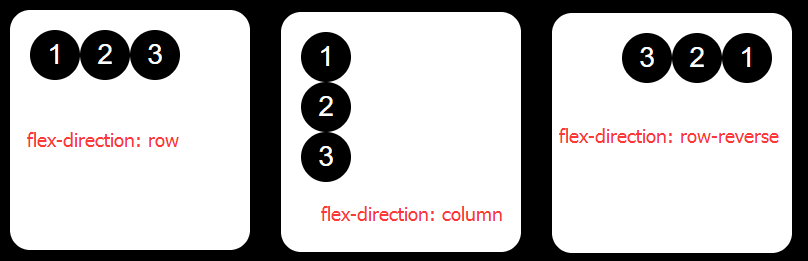
针对容器，弹性盒子的属性并不多，基本的大概就4个，分别用于设置主轴方向

flex-direction

语法格式：

flex-direction: row(default) | row-reverse | column | column-reverse

用于设置主轴的方向，flex 分主轴和交叉轴两个概念，items 布局时，默认延主轴方向进行，因此通过设置主轴是水平方向还是垂直方向就可以实现 items 的水平或垂直布局。

* row：默认值，设置主轴为水平方向
* column：设置主轴为垂直方向

其他属性就不介绍了，因为主轴方向就两个，要么水平，要么垂直，其他的区别仅在于水平时是从左到右，还是从右到左，所以这个属性的影响因素之一的 LTR 和 RTL，但没必要考虑这么多，这些场景应该不多，知道这个是用来设置主轴方向就够了，我觉得。

示例：

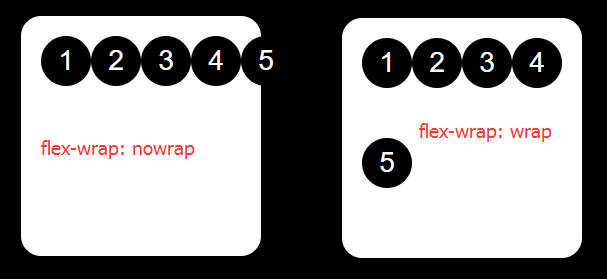
flex-wrap

语法格式：

flex-wrap: nowrap(default) | wrap | wrap-reverse

用于设置是否允许换行，默认值 nowrap。

当设置了 wrap 时，允许 items 在主轴方向溢出时自动进行换行布局，这点可以很好的用来实现响应式布局，比如当空间逐渐缩小时，原本水平排列的控件换成垂直方向排版。

示例：

flex-flow

语法格式：

flex-flow: <'flex-direction'> || <'flex-wrap'>

这个属性并没有另外的含义，它只是 flex-direction 和 flex-wrap 的简写用法而已。

如果你不想单独使用上述两个属性，可以将它们一起在 flex-flow 使用，如：

flex-flow: row wrap  
//等效于  
flex-direction: row;  
flex-wrap: wrap;

justify-content

语法格式：

justify-content: normal(default) | <content-distribution> | <overflow-position>? [ <content-position> | left | right ]  
where   
<content-distribution> = space-between | space-around | space-evenly | stretch  
<overflow-position> = unsafe | safe  
<content-position> = center | start | end | flex-start | flex-end

用于设置 items 在主轴方向上的对齐方式，可以靠左，靠右，居中或者按比例均分等效果。

需要先明确一点概念，对齐是指 items 在 flex 容器中的排版对齐方式，那么要想 flex 容器可以控制 items 的对齐方式，那主轴方向上自然就还需要有布局空白，如果都没有布局空白了，不就表明 items 已充满 flex 容器了，那谈何对齐。

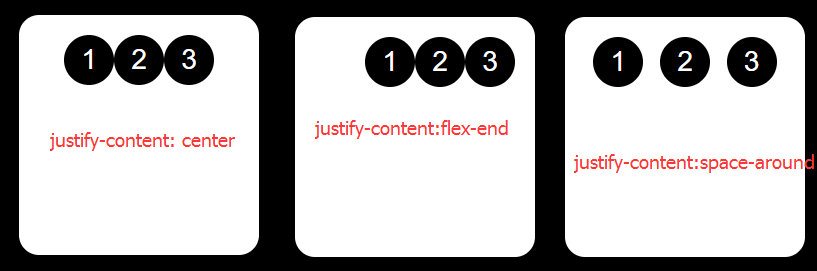
那么，如果存在至少一个 item，它的 flex-grow 属性不等于 0，justify-content 这个属性就失效了，因为 flex-grow 表示允许 item 按照比例瓜分布局空白，这样一来布局空白被瓜分完了，flex 容器在主轴方向上已被 items 充满， 也就没有对齐一说了。

所以要能够正确的使用该属性来控制 items 在主轴方向的对齐方式，那么就需要注意 item 中会影响布局空白的属性，如 flex-grow，flex-basis 这些的使用。

下面看看各属性值介绍（下面的介绍不考虑 RTL 的情况，默认都以 LTR 为主）：

* start：主轴是水平方向的话，左对齐方式排版；主轴是垂直方向的话，上对齐方式排版；
* end：主轴是水平方向的话，右对齐方式排版；主轴是垂直方向的话，下对齐方式排版；
* center：居中方式排版；
* space-between：等比例瓜分布局空白，每行首元素对齐，末元素对齐，每行各元素间距一致；
* space-around：与上述的类似效果，区别仅在于，每行首元素并不是在 flex 容器内容区域左边就开始布局，它距离 flex 容器左边的距离等于各个元素之间间距的一半。说白点，就是行首元素和末尾元素的周边有类型 margin 值存在。

剩余的属性值不介绍了，因为我也还没有搞懂它们的含义和应用场景。

示例：

(ps：flex 容器设置了 padding，所以 start 和 end 才没有贴靠边界 )

align-items

语法格式：

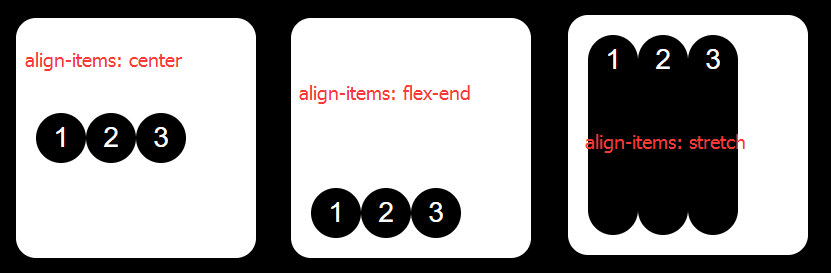
align-items: normal | stretch | <baseline-position> | [ <overflow-position>? <self-position> ]  
where   
<baseline-position> = [ first | last ]? baseline  
<overflow-position> = unsafe | safe  
<self-position> = center | start | end | self-start | self-end | flex-start | flex-end

用于控制 items 在交叉轴方向上的排版布局方式，justify-content 是能控制主轴上的排版，而这个属性则是用于控制交叉轴，通常两个都会一起使用，相互结合，可以达到一些类似页面居中效果。

看看属性值：

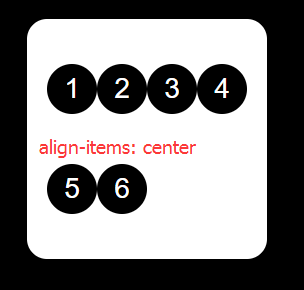
* flex-start：交叉轴方向，从起点开始布局排版
* flex-end：交叉轴方向，从末尾开始布局排版
* center：交叉轴方向，从中间开始布局排版
* stretch：交叉轴方向，如果 items 在交叉轴方向没有设置大小，那么让 items 在交叉轴的方向充满 flex 容器的高度。

其他属性不介绍了，不熟悉。

示例：

(ps：flex 容器设置了 padding，所以 start 和 end 才没有贴靠边界 )

stretch 要能够生效，需要在 items 在交叉轴方向的不设置大小，如上图中主轴是水平方向，那么 items 需要不设置 height，此时 stretch 才能够让 items 拉伸占据交叉轴的高度。

有一点需要注意，当 flex 容器的 items 在主轴方向上只有一行时，可以很明确的使用这个属性来控制在交叉轴的排版方式。但，如果 items 在主轴上超过一行，那么最终的效果可能就不是想要的了，比如下图：

如果是想实现多行的 items 都作为一个整体居中，那么用 align-items 就无法实现了，针对这种有多行的情况，需要用另外一个属性来控制：align-content。

align-content

语法格式：

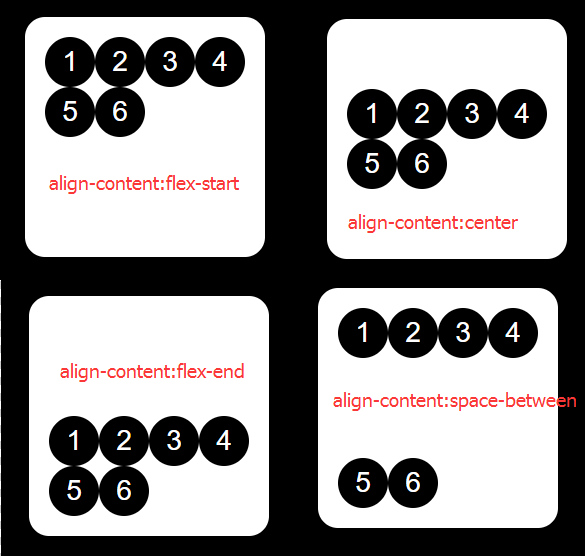
align-content: normal | <baseline-position> | <content-distribution> | <overflow-position>? <content-position>  
where   
<baseline-position> = [ first | last ]? baseline  
<content-distribution> = space-between | space-around | space-evenly | stretch  
<overflow-position> = unsafe | safe  
<content-position> = center | start | end | flex-start | flex-end

当 flex 容器的 items 设置了溢出换行属性，且当前在交叉轴方向上存在多行 item 的情况下，该属性才会生效。

网上有种翻译，说这个属性是用于轴对齐，我不是很理解，我自己粗俗的分两种情况理解：

当需要进行 start, center, end 这些排版时，是将这些多行的 items 都看成一个整体，然后进行交叉轴方向上的排版控制。此时，将多行 item 看成一行之后，那么这个 align-content 之后的排版布局就跟 align-items 一样的效果了。

其他的 space-around，space-between 究竟是如何计算排版的，不熟悉。

属性值含义不看了，跟 align-items 一样的效果，直接看示例：

(ps：flex 容器设置了 padding，所以 start 和 end 才没有贴靠边界 )

place-content

语法格式：

place-content: <'align-content'> <'justify-content'>?

这个属性并没有另外的含义，它只是 align-content 和 justify-content 的简写用法而已。

如果你不想单独使用上述两个属性，可以将它们一起在 place-content 使用，如：

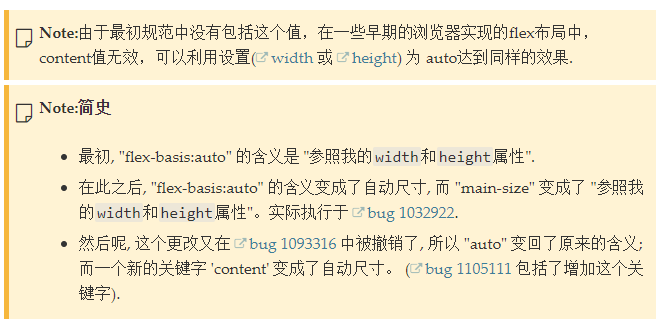
place-content: center center  
//等效于  
align-content: center;  
justify-content: center;

弹性盒子子项的属性：

flex-basis

语法格式：

flex-basis: content | <'width'>  
​  
where   
<'width'> = [ <length> | <percentage> ] && [ border-box | content-box ]? | available | min-content | max-content | fit-content | auto  
​

用于设置 items 在主轴方向的大小，如果主轴是水平方向，相当于设置 width，此时，该属性值会覆盖掉 width 设置的大小。

尝试了下，在 chorme 浏览器上 content 属性不生效，不清楚，可能不同浏览器行为还不一样，既然这样，就先暂时不深入了解这个属性了，大概知道用于设置主轴方向上的 item 大小即可。

就算要使用，先直接用指定数值大小的方式好了。

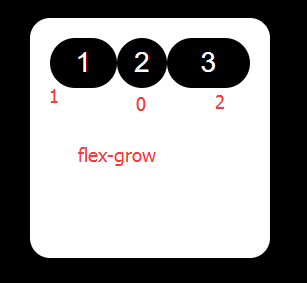
flex-grow

语法格式：

flex-grow: <number>

用于设置 item 在主轴方向上的拉伸因子，即如果 flex 容器还有剩余空间，会按照各 item 设置的拉伸因子比例关系分配。默认值为 0，即不拉伸。

作用很像 Andorid 中的 LinearLayout 的 child 里设置了 layout\_weight 用途一样。

示例：

flex-shrink

语法格式：

flex-shrink: <number>

用于设置 item 在主轴方向上的收缩因子，跟 flex-grow 刚好反着来。当 flex 容器空间不够，item 要溢出时，设置次属性可控制 item 按比例进行相应收缩，以便不让 item 溢出，默认 1，值越大收缩倍数越大，最后 item 就越小，0 表示不收缩，负值无效。

另外，如果设置了换行属性，那么这个就无效了。换行和收缩都是用于解决 item 在主轴方向上溢出的问题，既然是互斥，且换行优先级高，那么设置了换行，就不会再对 item 进行收缩了。

示例：

flex

语法格式：

flex: none | auto | initial | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]

这属性是 flex-grow，flex-shrink，flex-basis 三个属性的简化使用，有三种属性值：

* none：元素会根据自身宽高来设置尺寸。它是完全非弹性的：既不会缩短，也不会伸长来适应flex容器。相当于将属性设置为 flex: 0 0 auto。
* auto：元素会根据自身的宽度与高度来确定尺寸，但是会自行伸长以吸收flex容器中额外的自由空间，也会缩短至自身最小尺寸以适应容器。这相当于将属性设置为 flex: 1 1 auto.
* initial：属性默认值， 元素会根据自身宽高设置尺寸。它会缩短自身以适应容器，但不会伸长并吸收flex容器中的额外自由空间来适应容器 。相当于将属性设置为"flex: 0 1 auto"。

flex 属性可以指定 1 个，2 个或 3 个值。

**单值语法**: 值必须为以下其中之一:

* 一个无单位**数(<number>)**: 它会被当作 <flex-grow>的值。
* 一个有效的**宽度(width)**值: 它会被当作 <flex-basis>的值。
* 关键字 none, auto,或initial.

**双值语法**: 第一个值必须为一个无单位数，并且它会被当作<flex-grow>的值。第二个值必须为以下之一：

* 一个无单位数：它会被当作<flex-shrink>的值。
* 一个有效的宽度值: 它会被当作<flex-basis>的值。

**三值语法:**

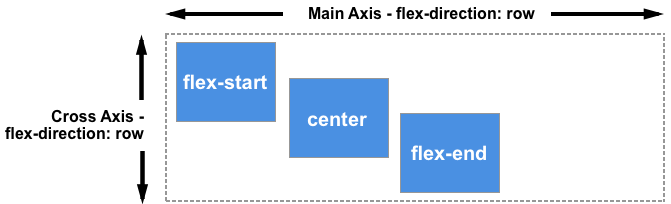
* 第一个值必须为一个无单位数，并且它会被当作<flex-grow>的值。
* 第二个值必须为一个无单位数，并且它会被当作 <flex-shrink>的值。
* 第三个值必须为一个有效的宽度值， 并且它会被当作<flex-basis>的值。

align-self

语法格式：

align-self: auto | normal | stretch | <baseline-position> | <overflow-position>? <self-position>  
where   
<baseline-position> = [ first | last ]? baseline  
<overflow-position> = unsafe | safe  
<self-position> = center | start | end | self-start | self-end | flex-start | flex-end

用于给单个 item 设置交叉轴方向上的排版布局方式，属性值和作用跟 align-items 一样，区别仅在于 align-items 是 flex 容器的属性，它会作用于所有的 items 上；而 align-self 允许对单个 item 设置，该值会覆盖 align-items 设置的属性值。

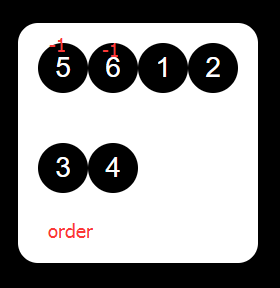
这样就可以实现控制交叉轴上的每个 item 的不同布局方式，如下：

order

语法格式：

order: <integer>

用于控制 items 的排版顺序，item 将按照 order 属性值的增序进行布局。拥有相同 order 属性值的元素按照它们在源代码中出现的顺序进行布局。默认值为 0，可设置负值，排序将在默认不设置的 item 前面。

示例：

小结

我觉得，这些属性大体记得每个属性的主要用途即可，至于每个属性值如何设置，如何相互结合使用可以达到什么样的效果，写代码的时候再调就是了。

# CSS预处理器

## CSS预处理

什么是 CSS 预处理？为什么要有 CSS 预处理？

这里就讲讲这两个问题，写过 CSS 应该就会比较清楚，虽然我才刚入门，但在写一些练手时就已经有点感觉了：写 CSS 后，很难维护，维护基本要靠注释来，而且由于 HTML 文档中标签的嵌套层次复杂，导致写 CSS 的选择器时也很费劲，尤其是在后期为某部分标签新增样式时，总会不知道到底应该在 CSS 文件中哪里写这个选择器，这个选择器是否会与前面冲突。

最有感觉的一点是，CSS 代码基本没法复用，一个页面一份 CSS，每次都需要重写，只是很多时候，可以直接去旧的里面复制粘贴。

有这么些问题是因为 CSS 本身并不是一门编程语言，它只是一个标记语言，静态语言，不具备编程语言的特性，所以写容易，但维护会比较难。

这个时候，CSS 预处理器就出现了，其实应该是说 Sass 和 Less 这类语言出现了。

Sass 和 Less 这类语言，其实可以理解成 CSS 的超集，也就是它们是基于 CSS 原本的语法格式基础上，增加了编程语言的特性，如变量的使用、逻辑语句的支持、函数等。让 CSS 代码更容易维护和复用。

但浏览器最终肯定是只认识 CSS 文件的，它并无法处理 CSS 中的那些变量、逻辑语句，所以需要有一个编译的过程，将 Sass 或 Less 写的代码转换成标准的 CSS 代码，这个过程就称为 CSS 预处理。

所以，CSS 预处理器其实只是一个过程的称呼，主要工作就是将源代码编译并输出标准的 CSS 文件，而这个源代码可以用 Sass 写，也可以用 Less，当然还有其他很多种语言。

那么，到底哪一种语言好，我还没能力来讨论，反正存在即有理，每种语言总它的优缺点、总有它的适用场景。

下面，主要就来介绍下其中的两种语言：Less 和 Sass。

我觉得，掌握 CSS 预处理的关键，其实也就两点，一是掌握语言的语法、二是清楚怎么编译成标准的 CSS 文件；语法基本都不会很难，编译一般需要配置一些环境，因为我使用的开发工具是 WebStrom，所以会介绍下 WebStrom 上的配置。

## Less

使用

Less 写的 CSS 文件后缀名为 .less，但浏览器并不认识 less 文件，所以最后需要转换成 css 文件，有两种方式：

* **借助 less.js**，程序运行期间，浏览器会自动借助 less.js 来解析 less 文件内的代码，转成 css 标准语法

这种方式，不需要配置任何其他环境，只需要下载 less.js，并在项目中使用即可，但有几点需要注意：

<head>  
    <!--link标签需要设置 rel=stylesheet/less, less.js的加载需要放在所有 link 标签后面-->  
    <link rel="stylesheet/less" type="text/css" href="css/src/main.less"/>  
    <link rel="stylesheet/less" type="text/css" href="css/src/test.less"/>  
    <script src="js/lib/less.js"></script>  
</head>

HTML 文档通过 link 标签引入 less 文件时，需要将 link 标签的 rel 属性设置为 stylesheet/less，否则无效；

而且，用 <script> 标签加载 less.js 的代码需要放在最后，即所有用 link 标签引入 less 文件的后面，因为 less.js 文件加载成功后就会去将 less 转成 css 标准样式，在 <script> 标签后面才用 link 标签引入的那些 less 文件就无法被转换了。

[less.js 下载地址：https://github.com/less/less.js/releases](https://github.com/less/less.js/releases)

* **借助 node.js 环境**，安装完 less 后，执行 lessc 命令

第一种方式，虽然便捷，但会让页面的渲染多了一个转换步骤，延长页面渲染时长，所以，还可以用第二种方式，直接在本地将 less 转成 css 文件后，项目直接使用转换后输出的 css 文件。

假设你已经在电脑上安装了 node.js 了，如果还没有，先去网上自行搜索下教程，很多，也很快。

首次使用需要先安装 less，打开终端，执行下述命令：

npm install -g less

安装完后，就可以使用 lessc 命令，如：

lessc main.less main.css

这是最简单的用法，将 main.less 文件编译输出 main.css；还有其他高级的用法，比如顺便压缩 css 文件，输出 .min.css 文件等等。

WebStorm配置

我是比较习惯使用第二种方式，也就是在本地就将 less 文件转成 css 文件，项目里是直接使用转换后输出的 css 文件了，而且我用的开发工具是 WebStrom，所以可以借助它，省去了每次自己输命令的操作：

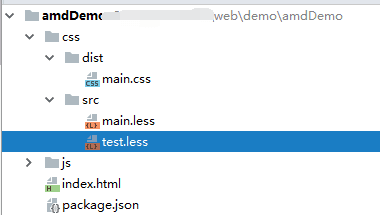
第一步：在项目根目录下执行 npm init -y 初始化项目，初始化完项目后，根目录会生成 package.json 文件；

第二步：打开 package.json，在里面的 scripts 字段，根据你的项目结构，输入脚本命令；

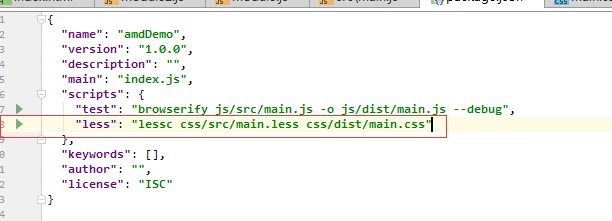
第三步：点击 scripts 旁边的三角形按钮，就可以自动执行脚本命令，完成转换工作；

第四步：（可选）如果嫌每次都需要自己手动点击按钮麻烦，可以将这项工作添加进 File Watcher 功能中，每次文件改动就会自动执行脚本命令，完成转换，具体参考[上一篇](https://www.cnblogs.com/dasusu/p/10105433.html)，或者自行搜索，很简单。

这是例子的项目结构：



src 目录中存放 less 文件，dist 目录中存放转换后输出的 css 文件，所以，我的 package.json 里的脚本命令如下：



具体的脚本命令可根据实际需求，实际项目结构，自行决定。

语法

好了，清楚了 less 文件的两种使用方式后，就可以来学习语法了，这样在学习语法过程中，就可以随时进行转换，查看 less 书写的代码，最终转换的 css 代码是什么样的，这样比较着学习比较容易掌握。

* **变量**

通过 @变量名: 来定义变量，通过 @变量名 使用变量，其实本质上是常量了，如：

@width: 10px;  /\*定义变量\*/  
@height: @width + 10px; /\*使用变量，和逻辑语句\*/  
​  
#header {  
  width: @width;  /\*使用变量\*/  
  height: @height;  
}

转换成 CSS 文件：

/\*定义变量\*/  
/\*使用变量，和逻辑语句\*/  
#header {  
  width: 10px;  
  /\*使用变量\*/  
  height: 20px;  
}

* **注释**

上个例子中，转换后的注释也还保留着，这是因为 less 和 css 都支持 /\* \*/ 的注释方式，所以这种会保留，但如果是 //，这种只有在 less 中支持，css 不支持，那么这种注释就不会保留，验证下：

/\*这是注释1\*/  
//这是注释2

转换成 css 文件：

/\*这是注释1\*/

所以，在 less 中使用注释时，需要注意下，哪些是想保留，哪些是不想保留的。

* **Mixins（混合）**

也有的文章里翻译成混入，还有的文章直接保留单词，不做翻译，可能是觉得中文翻译不能够很好的表达意思吧。

先说有这么种场景：有时候写在某个选择器中的属性样式，在其他选择器中也需要，所以通常是直接将那部分复制粘贴过来使用。

而 less 的 Mixins 允许你在某个选择器内，直接使用其他选择器内的属性样式，所以中文翻译才有混合，或混入之说，其实也就是将其他的属性样式混合到当前选择器中。

看个例子就明白了：

.class1 {   //类选择器 class1 的属性样式  
  width: 10px;  
  height: 20px;  
}  
​  
.class2(@color: #fff) {  //定义了一个模板样式，类似于函数的作用  
  background-color: @color;  
}  
​  
#id1() {  //定义了一个模板样式，类似于函数作用  
  border: 1px solid #ff22ff;  
}  
​  
.mian {   
  .class1; //直接使用其他选择器定义的属性样式  
  .class2(#f2f); //使用模板样式，传入参数  
  #id1; //使用模板样式，不传参时，括号可省略  
}

转换后的 CSS 文件：

.class1 {  
  width: 10px;  
  height: 20px;  
}  
.mian {  
  width: 10px;  
  height: 20px;  
  background-color: #f2f;  
  border: 1px solid #ff22ff;  
}

可以注意对比源代码和转换后的代码，原本就定义好的基本选择器，如 .class1，可直接在其他选择器内通过 .calss1 将它内部的属性样式都复制过来。

也可以在基本选择器后面加上 () 括号，这样一来，这个就会被当做模板处理，作用类似于函数，可接收参数，使用时就类似于调用函数那么使用，如果不传参，调用时也可以将括号省略。既然是作为函数使用，那么它们存在的意义就只是被调用，所以转换后的 CSS 中并不会存在这个函数。

另外，有的文章中，对 Mixins 的解释是说，在 class 中使用 class，但上面的例子中也测试了，class 中也是可以使用其他 id 选择器的属性样式的，所以应该不仅限于 class 类选择器，但不能用于组合选择器中。

* **命名空间**

我对于命名空间的理解：属于 less 自己的命名空间，也就是这些代码并不会在转换后的 CSS 文件中出现，因为它只属于 less。

所以，其实也就是上述例子中，类似函数存在的那些模板选择器，当在书写选择器时，在其后面增加 () 括号，则表示这个选择器只属于 less 的命名空间，转成 CSS 后并不会出现。k

* **嵌套**

在写 CSS 时，组合选择器经常写得很复杂，因为 HTML 里的标签嵌套层次本身就很复杂，而且组合选择器写完也不是能够很明显的表示出它的目的，所以 less 允许依据 HTML 中的嵌套层次来书写，不用去记那么多组合选择器的规则，如：

body {  
  font-size: 16px;  
  .content {  
    .banner {  
      ul {  
        img {  
          width: 700px;  
          height: 300px;  
          &:hover {  
            width: 800px;  
            height: 300px;  
         }  
       }  
     }  
   }  
 }  
}

转换成 CSS 文件：

body {  
  font-size: 16px;  
}  
body .content .banner ul img {  
  width: 700px;  
  height: 300px;  
}  
body .content .banner ul img:hover {  
  width: 800px;  
  height: 300px;  
}

也就是，子孙后代的组合选择器规则可以不用去记，直接根据 HTML 文档中标签的嵌套层次来书写即可，这样便于后期维护，如果要对某个标签新增某些样式，也知道该去哪里找。

有一点需要注意的是，类似 a:hover 这种伪类选择器，需要加一个 & 符号。

* **运算**

less 允许在代码中进行一些简单的加、减、乘、除基本运算，结合变量的使用，可进行一些字体、颜色等的动态运算效果。

@border: 1px;  
@color: #fff;  
#header {  
  color: @color \* 0.5;  
  border-width: @border @border\*2 @border\*3 @border\*4;  
}

转换成 CSS 后：

#header {  
  color: #808080;  
  border-width: 1px 2px 3px 4px;  
}

通常，变量和运算都是用于对颜色、大小等进行计算。

* **内置函数**

less 内置了一些基础函数，可用于转换颜色、处理字符串、算术运算等，这里列举一些函数：

color("#aaa");  //输出 #aaa， 将字符串的颜色值转换为颜色值  
image-size("file.png");  //输出 => 10px 10px，获取图片文件的宽高信息  
//image-with("file.png"); image-height("file.png"); 获取图片文件宽高  
convert(9s, "ms");  //输出 => 9000ms，单位换算，例如对 m,cm,mmin,pt,pc的换算  
@size: if((true), 1px, 0px);  //if函数，第一个参数为条件，满足则返回第二个参数，不满足返回第三个参数  
if(not (true), 1px, 0px);     //非语句， not  
if((true) and (true), 1px, 0px); //逻辑与语句， and   
if((false) or (true), 1px, 0px); //逻辑或语句， or   
​  
//处理数组  
@list: "banana", "tomato", "potato", "peach";  
length(@list);  // 输出 => 4  
extract(@list, 3);  //输出 => potato,注意第一个不是从 0 开始计算  
​  
//类型判断  
isnumber(#ff0);  // false  
isstring("234"); // true  
iscolor(#ff0);   // true  
isXXX  
​  
...

内置函数很多，用途也很多，覆盖了基本算术运算、逻辑语句、颜色计算、字符串处理等等，需要用时再查手册吧。

* **作用域**

作用域很好理解，就是类似 JavaScript 中的变量作用域，因为在 less 中都是通过 @变量名: 来定义变量的，后定义的会覆盖掉前定义的，但当在不同嵌套层次中定义同一变量时，就存在局部变量和外部变量之分，内部变量并不会覆盖掉外部变量。

而且，less 的变量定义也有类似 JavaScript 中的提前特性，如：

@var: red;  
​  
#aaa {  
  color: @var;  // yellow，因为后面定义的 @var:yellow 将 @var:red 覆盖掉了  
}  
​  
#page {  
  #header {  
    color: @var; // white，内部变量不影响外部变量  
 }  
  @var: white;  
}  
​  
@var: yellow;  
​  
#ppp {  
  color: @var;  //yellow  
}

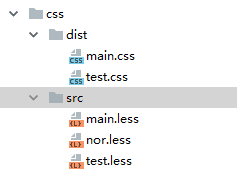
看看转换成 CSS 后：

#aaa {  
  color: yellow;  
}  
#page #header {  
  color: white;  
}  
#ppp {  
  color: yellow;  
}

* **import（导入）**

如果某份 less 文件是可以复用的，那么可以使用 @import 命令将其全部引入使用。

@import "main";



最后在 test.css 里会汇合 main.less 里的代码。

以上，只是介绍 less 的基础语法，还有更多详细、复杂的语法用途，需要时再翻阅文档吧。

## Sass（Scss）

Sass 相比于 Less 功能会更强大，但也更复杂。

Sass 和 Scss 本质是一家，Sass 早期版本的文件后缀名是 .sass，从 Sass 3 之后，因为修改了一些特性语法，Sass 更加强大且易使用，从这个版本之后的文件后缀名改成了 .scss，所以 Scss 其实 Sass 的新版本的称呼，但两者本质上没太大区别。

Scss 是基于 Sass 的语法基础上，修改了一部分的语法。比如早期的 Sass 是通过换行和缩进如：

#sidebar  
 width: 30%  
 background-color: #faa

这种语法格式跟 CSS 不一致，让使用者会很不习惯，Scss 之后就换成用分号和括号了：

#sidebar {  
  width: 30%;  
  background-color: #faa;  
}

使用

Sass 不像 Less 一样可以直接借助 less.js 来进行转换，它是基于 Ruby 运行环境，所以电脑上需要先安装 Ruby，然后才能有办法将 Sass 文件转成 CSS。

[Ruby 下载地址：https://rubyinstaller.org/downloads/](https://rubyinstaller.org/downloads/)

因为是 exe 文件，下载完直接按提示安装就可以了，安装后打开终端，执行 gem 命名安装 Sass：

gem install sass

安装完 Sass 后，就可以通过 scss 命令来进行转换工作了，如：

scss main.scss main.css

上述命令中，scss 换成 sass 也可以，但注意，scss 或 sass 命令是基于 Ruby 环境下运行的命令，因为电脑上已经安装过 Ruby 了，也通过 Ruby 安装了 Sass，所以才可以在终端里直接执行 scss 命令。

而类似于 Less 中说到的，WebStrom 可以借助 package.json 里的 scripts 来手动运行脚本命令，这有个前提，就这些脚本命令是运行在 node.js 环境上的，所以如果你直接将上述 scss 命令配置到 package.json 的 script 里，你会发现，是运行不了的。

要解决这个问题，让 WebStrom 能够运行 sass 命令来处理转换工作有两种方式：

* 直接去 WebStrom 配置 File Watcher，program 选择 Ruby 目录中的 sass.bat 或 scss.bat

这种方式下，每次配置的文件变动时，会自动生成对应的 css 文件，转换工作会自动实时进行。但如果不习惯这种方式，想要每次编写完 scss 代码后，手动来触发转换工作，那么可以选择第二种方式：

* 通过 npm 命令安装 sass

在终端里执行 npm install -g sass，这样就可以类似配置 less 那样的步骤来使用 sass 命令了，在 package.json 里配置相关命令，然后手动点击脚本的运行即可。

但 npm 安装的 sass 跟在 Ruby 下安装的 sass 是否有和区别，我不清楚，用段时间，如果有啥问题再来说说。

而且，对于选择使用 Sass，刚接触可能会有些困惑，是应该使用哪个后缀名的文件，命令是该用 sass 还是 scss 来进行转换，我也有这个困惑，但感觉好像并没有什么区别，先试着用段时间，以后熟悉了再来讲讲。

最后，Sass 虽然有 .sass 和 .scss 两种后缀名的文件，但建议使用 .scss，因为前者的语法跟 CSS 很不一样，使用起来会有些不习惯，当然如果你有 Ruby 基础的话，可能会比较喜欢这种。我个人会选择后者。

语法

语法方面，大部分类似于 Less，但就细节方面可能有些不一样，还有，支持更多更强大的功能吧。

上面介绍的 Less 的基础语法、基础功能，Sass 也基本全部支持，也差不了多少，所以下面就不一个个来介绍了，详细的到开头声明部分给的中文网链接中去查阅即可。

下面就主要列一些不同的地方：

* **变量**

Sass 中的变量用 $变量名: 定义，用 $变量名 使用，其余跟 Less 差不了多少。

* **作用域**

Less 中的变量分局部作用域和全局作用域，但在 Sass 中，不同版本，作用域范围并不一样，摘抄一段原文中描述：

Sass 中变量的作用域在过去几年已经发生了一些改变。直到最近，规则集和其他范围内声明变量的作用域才默认为本地。如果已经存在同名的全局变量，则局部变量覆盖全局变量。从 Sass 3.4 版本开始，Sass 已经可以正确处理作用域的概念，并通过创建一个新的局部变量来代替。

* **条件语句**

Less 中并不支持条件语句，当然，可以通过内置函数 if 以及 and，or，not 这些来模拟条件语句。

在 Sass 中是支持条件语句的，但也不是像其他编程语言直接 if 这样通过保留字来编写，需要加个 @ 符合，如：

@if $support-legacy {  
  // …  
} @else {  
  // …  
}

教程中给了几条准则要求：

* 除非必要，不然不需要括号；
* 务必在 @if 之前添加空行；
* 务必在左开大括号({)后换行；
* @else 语句和它前面的右闭大括号(})写在同一行；
* 务必在右闭大括号(})后添加空行，除非下一行还是右闭大括号(})，那么就在最后一个右闭大括号(})后添加空行。

另外，教程中也说了：

除非你的代码中有偏复杂的逻辑，否则没必要在日常开发的样式表中使用条件语句。实际上，条件语句主要适用于库和框架。

其他区别，等用段时间，熟悉了再来讲讲。

框架-[Compass](http://compass-style.org/)

Sass 有一点比 Less 有优势的就是，目前有很多稳定且热门的基于 Sass 编写的框架库，比如：[Compass](http://compass-style.org/)、[Bourbon](http://bourbon.io/) 和 [Susy](http://susy.oddbird.net/) 等。

这些框架库就类似于 jQurey 和 JavaScript 关系，对 Sass 进行了一层封装，让编写 Sass 代码的人，可以极为简便的开发，我还没用过，就不过多介绍了。

# JavaScript快速入门

## ES5 ES6

经常会看见一些网页或招聘要求上写着需要掌握ES5,ES6，这个是什么意思呢？

ES：ECMAScript

其实也就是制定和发布脚本语言的规范，这不是一门语言，是一个标准。

所以，ES5，ES6也就是类似于JDK8, JDK9的意思，指的是不同版本的JavaScript，每个版本有其自己的一些新特性。

## 语法

弱类型语言

声明变量类型时不必明确声明其类型，只需用var声明即可：

比如Java中需要定义如下变量：

int a;

float a;

double a;

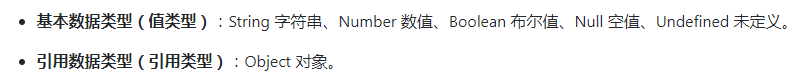
String a;

boolean a;

而JavaScript中，只用定义一个变量：

var a;

变量的数据类型

虽然声明变量时，不必指出变量的类型，但也要清楚下，JS中的基本数据类型：

在JavaScript中，只要是数，就是 number 数值型的。无论整浮、浮点数（即小数）、无论大小、无论正负，都是 number 类型的。

可以使用内置方法：isNaN()，来判断某个变量是否是数值类型。

关键字 typeof 可以打出变量的类型，如果需要查看某个变量的类型时。

类型的强制转换

转String:

* 变量 + “”
* 变量.toString()
* String(变量)

转Number

* parseInt/Float(变量)

函数

function sum(a, b){

return a+b;

}

函数的定义如上，不用声明是否有返回值，当需要有返回值时，直接在最后一行代码里加上return。函数名也可以省略，此时称匿名函数。

当定义了函数之后，需要调用函数的时候，直接用函数名()，如sum(1,2)

但如果只是想把函数跟某一事件绑定时，此时只需要函数名，如：

button.onclick() = sum;

如果此时sum也带上括号，sum()，那么函数就会被调用，这里需要注意。

对象

在js中不必像Java那样新建个类，然后从这个类new出对象。在js中，需要对象时，直接new Object()，然后赋予想要的属性和行为即可。

首先创建一个对象：

var obj = new Object();

向对象中添加属性：

obj.name = "孙悟空";

obj.age = 18;

对象的属性值可以是任何的数据类型，也可以是个**函数**：

obj.sayName = function () {

console.log('smyhvae');

};

console.log(obj.sayName); //没加括号，获取的是对象

console.log('-----------');

console.log(obj.sayName()); //加了括号，执行函数内容，并执行函数体的内容

创建对象的方式也可以用另外一种方式：

var o = {

name: "孙悟空",

age: 18,

sayName: function() {

console.log(this.name);

}

};

也可以使用类似于Java定义类，实例化对象的方式。

函数和方法的区别

在很多地方，都会看到有的人说方法，有的人说函数，其实这两个可以说没什么区别，可以认为是同一个事物的两种叫法，但如果真较真，想要区分的话，也可以。

如果一个函数作为一个对象的属性保存，那么我们称这个函数是这个对象的**方法**。调用这个函数就说调用对象的方法（method）。

如：

//调方法

obj.sayName();

//调函数

fun();

比较

=== !== 以及 == !=

前者为严格模式下的比较，比对的两者将从数据类型和数值上都进行比较，相同时才返回true。

后者为非严格模式下的比较，只比对两者的数值，即使数据类型不一致。

## 字符串操作

查找: indexOf()

'hello world'.indexOf('world');

截取字符串：slice()

'hello world'.slice(0, 5); // hello [0, 5)

'hello world'.slice(6); // world [6, end]

## 作用域

由于js中很少使用class概念，虽然也可以用，基本上都是引入一份份的js文件代码，而这些文件里的js代码基本都是一些function，因此有必要理清在js中变量的作用域。

作用域就分两个：全局作用域和局部作用域

全局作用域

只要不在任何 {} 代码块中声明的变量和函数，作用域都是全局，并且，变量会自动变成 window 的成员变量，函数自动变成 window 的成员方法，在任何地方都可以直接访问这些变量或者通过 window.变量名访问。

## null undefined

## 类型隐式转换

## 回调写法

function foo(callback) {

{

// do something

}

callback && callback() // 不传 callback 参数，则不会执行 callback() 函数

}

foo(); // 只执行do something中的代码

foo(callback);//callback是另一个函数，将此函数传入 foo，将会执行callback()

# DOM

DOM（document object model）：文档对象模型

了解JavaScript基本语法后，就要接着了解DOM概念。类似于CSS通过选择器来操作HTML文档中的元素。那么，同样的道理，js也需要有个中间媒介来操作HTML文档中的元素，这个媒介就是DOM。

## 概念

那么，什么是DOM呢，其实就是浏览器根据HTML文档构建出的一颗DOM树，树中每个节点对应着HTML文档中的每个元素标签，因此树的结构可以很好的表现出各个元素之间的层级关系。

另外，每个节点都携带着当前元素的所有信息，包括CSS作用的样式属性表，设置的类型，id等等，这些信息可以通过节点的各种属性方法获取到。

但有一点需要注意下，元素修饰的文本内容也会被创建成一个节点，作为这个元素的子元素加入DOM树中。

这种DOM树的概念跟Android中的视图树很类似。所以，每份HTML文档都会对应一颗DOM树。

JavaScript可以通过全局变量document拿到这个DOM树对象，那么之后就可以根据DOM提供的各种API接口来操纵这颗DOM树，包括获取指定节点的元素，动态修改该节点元素的信息，给这个节点元素绑定上事件操作等等。

兼容性

想想，这颗DOM树是谁来建模生成的呢？很明显，是浏览器吧，浏览器解析HTML文档以及CSS后，根据当前的视图建模出一颗DOM树出来。并提供了各种API接口供JavaScript来操纵。

那么，不同的浏览器厂商实现可能就会有所不同，比如W3C规定了一系列操纵DOM的API接口，但浏览器不想全部实现，就实现了其中核心部分，或者就算实现了，具体表现也有可能有所不同。

那么，这时就会存在一个问题了，也就是我们通过JavaScript，然后根据W3C规范的API接口来操纵DOM时，可能在不同浏览器上有不同的变现行为。所以，这时就需要考虑兼容性处理了。

但有一个更方便的解决方案，那就是使用**jQuery**，这是一个基于JavaScript的框架库，它封装了操纵DOM的各种功能，内部对不同浏览器进行了兼容性处理，那么我们使用的时候就可以不用再去考虑那么兼容性的处理了。

## document

document是内置的全局变量，在JavaScript可以直接通过该关键字使用，使用时会获取到当前HTML文档对应的 Document对象。

拿到这个对象后，就可以调用它的一些属性和方法来获取或修改我们想要的数据。

|  |  |
| --- | --- |
| **readyState** | **查看当前文档的被浏览器加载的状态（加载中等）** |
| **body/head/title** | 直接获取文档的相关元素标签信息 |
| **getElementByXXX()** | 根据id，class，tag等在文档中查找指定元素 |
| **createElement(tag)** | 创建指定标签的元素节点 |
| **craeteTextNode(text)** | 创建指定文本内容的Text对象 |
| **location** | 返回当前文档地址的Location对象 |

API很多，需要的时候再查就行，主要清楚下，document是JavaScript操纵DOM树的入口，从这里开始，可以获取一些关于文档的元数据方面的属性信息，也可以来查找指定的文档中某个节点的元素对象。

## location

Location对象提供了细粒度的文档地址信息，也支持导航到其他文档上。

当打开新文档在URL中有携带了一些信息时，可以通过这个来获取这些信息。

|  |  |
| --- | --- |
| **protocol** | **获取或设置文档URL的协议部分** |
| **host** | 获取或设置文档URL的主机和端口部分 |
| **href** | 获取或设置当前文档的地址 |
| **hostname** | 获取或设置文档URL的主机名部分 |
| **port** | 获取或设置文档URL的端口部分 |
| **pathname** | 获取或设置文档URL的路径部分 |
| **search** | 获取或设置文档URL的查询（问号串）部分 |
| **hash** | 获取或设置文档URL的锚（#号串）部分 |
| **assign(url)** | 导航到指定的URL |
| **replace(url)** | 清除当前文档并导航到新的URL |
| **reload()** | 重新载入当前文档 |
| **resolveURL(url)** | 将指定的相对URL解析成绝对URL |

## window

window直译过来就是窗口，其实也就是表示文档当前所显示的窗口对象，所以一些窗口性的功能都可以通过这个对象来调用。

比如：调用浏览器弹框、定时器的使用、获取窗口信息包括窗口宽高，屏幕宽高等等、窗口的滑动、操纵浏览器窗口的历史记录、向其他窗口发送消息等等。

窗口信息

|  |  |
| --- | --- |
| **innerHeight/Width** | **获取窗口内容区域的宽高** |
| **outerHeight/Width** | 获取窗口的宽高，包括边框和菜单栏等等 |
| **screen** | 获取描述屏幕的Screen对象 |
| **Screen.width/height** | Screen对象获取屏幕宽高 |
| **Screen.availWidth/Height** | 获取屏幕可用的宽高，去掉工具栏菜单栏 |
| **pageX/Yoffset** | 获取窗口在水平/垂直方向已滚动过的像素 |
| **document** | 获取次窗口关联的Document对象 |
| **history** | 访问浏览器历史 |
| **location** | 获取当前文档地址的详细信息 |

与窗口交互

|  |  |
| --- | --- |
| **blur()** | **让窗口失去键盘焦点** |
| **close()** | 关闭窗口（不是所有浏览器都允许js关闭窗口） |
| **focus()** | 让窗口获得键盘焦点 |
| **scrollBy(x, y)** | 让文档相对于当前位置进行滚动 |
| **scrollTo(x, y)** | 滚动到指定位置 |
| **alert(msg)** | 弹出一个对话框 |
| **confirm(msg)** | 弹出一个带有确认和取消的对话框 |
| **showModalDialog(url)** | 弹出窗口，显示指定的URL |
| **postMessage(msg, origin)** | 给另一个文档发送消息 |
| **set/clearInterval(fun, time)** | 创建/撤销周期性的任务 |
| **set/clearTimeout(fun, time)** | 创建/撤销延时任务 |

## HTMLElement

通过document获取到Document对象，以此来获取操纵DOM的入口，根据需要获取所需的文档相关信息，或者搜索指定的DOM中节点的元素，此时这个节点的元素对象就是HTMLElement对象。

所有的标签元素的基类对象都是HTMLElement，这个类定义的公共的、基础的操作元素节点的方法和属性。

但每个标签实际上都有具体的实现类，比如body对应HTMLBodyElement，script对应HTMLScriptElement，具体实现类由这个标签独有的属性和方法。

基类HTMLElement对象定义的基础的方法、属性包括：获取或修改元素的指定属性，添加或移除元素某个class，查看或修改该标签包装的内容等等。

元数据属性

|  |  |
| --- | --- |
| **classList** | **获取元素设置的class列表，返回DOMTokenList对象，可直接add,remove等操作** |
| **className** | 获取元素设置的class列表，返回字符串 |
| **disabled/hidden/id** | 获取或设置disable/hidden/id… |
| **attributes** | 获取元素设置的属性值列表，返回Attr[]对象 |
| **innerHTML** | 获取元素标签包装的内容，包括文本内容及子元素 |
| **outerHTML** | 获取元素整个内容 |

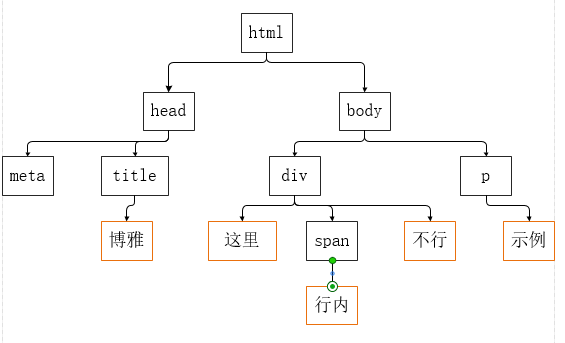
节点元素操纵

|  |  |
| --- | --- |
| **get/has/removeAttribute(name)** | **获取/判断/移除元素的某个属性** |
| **setAttribute(name, value)** | 设置元素的某个属性 |
| **appendChild(HTMLElement)** | 为当前元素添加子元素 |
| **cloneNode(boolean)** | 拷贝一份当前的元素，返回新的HTMElement对象，参数设置是否拷贝当前元素的子元素 |
| **isEqualNode(HTMLElement)** | 判断指定元素与当前是否相同，具有相同的class，相同的属性，相同的子元素 |
| **isSameNode(HTMLElement)** | 判断是否是同一个元素 |
| **removeChild(HTMLElement)** | 移除指定的子元素 |
| **replaceChild(HTMLElement, H.)** | 替换指定的子元素 |

## DOM模型示例

一颗DOM树究竟长什么样子呢？看个例子：

<!DOCTYPE **html**>  
<**html lang="en"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**title**>博雅</**title**>  
</**head**>  
<**body**>  
 <**div**>这里<**span**>行内</**span**>不行</**div**>  
 <**p**>示例</**p**>  
</**body**>  
</**html**>

上述是一份特别简单的HTML文档，看看它相对应的DOM树：

这就是对应的DOM树，层次分名，各节点都表示相对应的元素信息。但有一点需要注意下，橘色框都是相对应元素的文本内容，它也是DOM树中的节点，类型是Text对象。并且，并不是一个元素的所有文本内容作为一个Text对象，如果文本内容被其他元素标签分割开了，那么这些文本内容会被分割成多份节点，都作为元素的子元素拼接在DOM树中。

## DOM事件

通过上述一些方法，JavaScript可以定位找到所需的元素，然后也可以动态的修改相关数据，但通常，这些动态修改的操作都是用户操作了某些事件后去触发的。

所以，即使找到了元素后，还需要将元素与一些事件进行绑定，比如点击事件等等。

有两种方式让元素绑定事件和处理的方法：

addEventListener

**var** aElems = ***document***.getElementsByTagName(**"a"**);  
**for** (**var** i = 0; i < aElems.length; i++) {  
 aElems[i].addEventListener(**"click"**, **function** () {  
 ***console***.log(**"addEventListener:"** + **this**);  
 });}

调用HTMLElement对象的addEventListener()方法，第一个参数传入需要监听的事件名称，第二个参数为事件触发时的响应方法。

相对应的，还有一个removeEventListener()方法，同样接收这两个参数。

onXXX

**var** aElems = ***document***.getElementsByTagName(**"a"**);  
**for** (**var** i = 0; i < aElems.length; i++) {aElems[i].onclick = **function** () {  
 ***console***.log(**"onclick:"** + **this**);  
 }}

第二种方式，是在需要注册的事件类型前面加 on 作为元素的属性来注册事件的监听，这种比较常见。

所有事件类型

document的事件

|  |  |
| --- | --- |
| **readystatechange** | **readyState属性值发生变化时触发，也就是文档加载的不同阶段触发** |

window的事件

|  |  |
| --- | --- |
| **onabort** | **在文档或资源加载过程中被终止时触发** |
| **onerror** | 在文档或资源加载发生错误时触发 |
| **onhaschange** | 在锚部分发生变化时触发 |
| **onload** | 在文档或资源加载完成时触发 |
| **onresize** | 在窗口缩放时触发 |
| **onunload** | 在文档从窗口或浏览器中卸载时触发 |

鼠标事件

|  |  |
| --- | --- |
| **click** | **单击，释放时触发** |
| **dblclick** | 双击，释放时触发 |
| **mousedown** | 点击鼠标键时触发 |
| **mouseenter** | 在光标移入元素或某个后代元素所占据的屏幕区域时触发 |
| **mouseleave** | 在光标移出元素及所有后代元素所占据的屏幕区域时触发 |
| **mousemove** | 光标在元素上移动时触发 |
| **mouseout** | 与mouseleave基本相同，除了当光标仍然在某个后代元素上时也会触发 |
| **mouseenter** | 与mouseenter基本相同，除了当光标仍然在某个后代元素上时也会触发 |
| **mouseup** | 当释放鼠标时触发 |

鼠标事件被触发时，指定的处理方法都会传入一个MouseEvent对象，该对象携带一些额外的属性和方法供处理

MouseEvent

|  |  |
| --- | --- |
| **button** | **标明点击的是哪个键，0：鼠标主键，1：中键，2：次键** |
| **altkey** | 事件触发时是否有点击alt键 |
| **clientX** | 事件触发时鼠标相对于元素视口的X坐标 |
| **clientY** | 事件触发时鼠标相对于元素视口的Y坐标 |
| **screenX** | 事件触发时鼠标相对于屏幕坐标系的X坐标 |
| **screenY** | 事件触发时鼠标相对于屏幕坐标系的Y坐标 |
| **shiftKey** | 事件触发时是否有点击shift键 |
| **ctrlKey** | 事件触发时是否有点击ctrl键 |

键盘焦点事件

|  |  |
| --- | --- |
| **blur** | **在元素失去焦点时触发** |
| **focus** | 在元素获得焦点时触发 |
| **focusin** | 在元素即将获得焦点时触发 |
| **focusout** | 在元素即将失去焦点时触发 |

键盘焦点事件传入的是 FocusEvent对象

键盘点击事件

|  |  |
| --- | --- |
| **keydown** | **在用户按下某个键时触发** |
| **keypress** | 在用户按下并释放某个键时触发 |
| **keyup** | 在用户释放某个键时触发 |

键盘点击事件传入的是KeyboardEvent对象

# jQuery

## 为什么使用jQuery

类似于JVM隐藏了不同操作系统之间的差异，让开发能够更专注于功能的实现，而不必花费过多时间适配不同操作系统。jQuery隐藏了不同浏览器之间的差异，减少开发者花费在适配不同浏览器之间的精力。

举个例子：float属性

原生js的话，ie 需要通过 styleFloat 获取对象修改，W3C标准为 cssFloat，jQuery统一封装成 float，内部会自动根据不同浏览器的实现进行处理。

同时，它封装了很多基本实用的功能，如ajax，基本动画等。

## API中文文档

<http://www.css88.com/jqapi-1.9/css/>

## Ajax

$.**ajax**({  
 **url**: **"https://easy-mock.com/mock/5b592c01e4e04f38c7a55958/ywb/is/version/checkVersion"**,  
 **data**: **{"key": 122}**,  
 **type**: **"POST"**,  
 success: **function** (data) {  
 logUtils(data.content);  
 **var** con = ***JSON***.parse(data.content);  
 callback && callback(con);//通知回调  
 },  
 error: **function** (e) {  
 logUtils(e);  
 }  
});

## css

修改display样式

*//第一种方式*$(**"div"**).css(**"display"**, **"none"**);

$(**"span"**).css({**display**:**"block"**, **background**: **"#f2f"**});*//好用*

$(**"span"**).css({**"display"**:**"block"**, **"background"**: **"#f2f"**});*//这种也可以*  
*//第2种方式*$(**"div"**).each(**function** () {  
 **this**.**style**.**display** = **"none"**;  
});  
$(**"div"**)[0].**style**.**display** = **"none"**;  
*//第3种方式*$(**"div"**).show();*//等效于display:block(inline)*$(**"div"**).hide();*//等效于display:none*$(**"div"**).toggle();*//取相反的值*

读取样式时

$("div").css("display");

## class

添加class

$(selector).addClass("liItem"); //为指定元素添加类className

移除class

$(selector).removeClass("liItem"); //为指定元素移除类 className

$(selector).removeClass(); //不指定参数，表示移除被选中元素的所有类

判断有没有指定class

$(selector).hasClass("liItem"); //判断指定元素是否包含类 className

切换class

$(selector).toggleClass("liItem"); //为指定元素切换类 className，该元素有类则移除，没有指定类则添加

应用场景

当js动态修改的样式较少时，可直接通过.css()实现。

当js动态修改的样式比较多时，选择class操作较方便，事件将需要的样式写在css中，在js里直接添加或移除指定class实现。

如果考虑以后维护方便（把CSS从js中分离出来）的话，推荐使用类的方式来操作。

## html

创建元素 $(“”)

*//类似于js中: document.createElement("标签名")***var** node1 = $(**"<span>我是一个span元素</span>"**);*//返回的是jQuery对象*

添加子元素 append() prepend() html()

*//方式1:在.main元素的子元素末尾加入新的子元素*$(**".main"**).append(node1);  
$(**".main"**).append(**"<span>我是一个span元素</span>"**);  
*//方式2:在.main元素的子元素开头加入新的子元素*$(**".main"**).prepend(**"<span>我是第一个span元素</span>"**);  
*//方式3：替换掉所有子元素内容*$(**".main"**).**html**(**"<span>我把所有子元素都替换掉了</span>"**);

添加兄弟元素 after() before

$(**".main"**).after(**"<span>我是兄弟后span元素</span>"**);  
$(**".main"**).before(**"<span>我是兄弟前span元素</span>"**);

移除 html()

*//移除所有子元素*$(**".main"**).**html**(**""**);  
*//移除自已，自然子元素也被跟着移除*$(**".main"**).remove();

查看元素内容(包括标签)

***console***.log($(**".main"**).**html**());*//下面是元素标签和打出的日志*

$(**".main"**).prepend(**"<span>我是第<a>dsfds<span>23543</span></a>一个span元素</span>"**);



查看元素的纯文本内容

***console***.log($(**".main"**).**text**());*//下面是元素标签和打出的日志*

$(**".main"**).prepend(**"<span>我是第<a>dsfds<span>23543</span></a>一个span元素</span>"**);



text() 会返回当前元素内的所有文本内容，包括子孙后代元素所包装的文本内容。

小结

获取元素的内容（包括标签）可用html()，创建元素时用$ (“xxx”)，如果元素只有一个子元素，那么获取文本内容时可直接用text()，添加子元素时用append()。

## attr

设置属性 attr(“”,””)

$(selector).attr("title", "生命壹号");

获取属性 attr(“”)

$(selector).attr("title");

移除属性 removeAttr(“”)

$(selector).removeAttr("title");

# h5微信授权接入

官网：<https://mp.weixin.qq.com/wiki?t=resource/res_main&id=mp1421140842>

博客：

<https://blog.csdn.net/duotemplar/article/details/76018934>

# JavaScript语法

对于从 Java 或 C++ 转过来的开发人员来说 JavaScript 会有点让人困惑，因为它全部都是动态的，都是运行时，而且不存在类机制。所有的都是实例对象。所以，学习一些 JavaScript 的语言特性时，最好不要试图从 Java 中寻找相关概念硬套过来，类比可以。

## 相关术语

接下去一系列关于 JavaScript 语法的文章中，将会出现很多在 Java 里出现或者没出现过的专业术语，所以第一篇就先来罗列一下，每个术语先给一定的解释。

有的理解不了没关系，等这系列文章看完，再回过头来看这一张基本也都能理解了。先罗列出来，至少有个印象，不至于到时看到一脸懵逼。

直接量

可以直接使用的数据值，即在程序中直接出现的数据值，如：

“hello world”

123

{name:dasu}

[]

等等这些直接出现可直接使用的数据值称为直接量，不同数据类型有不同的叫法，常见的有：数字直接量、字符串直接量、正则表达式直接量、对象直接量、数组直接量。

字面量

跟直接量没什么区别，都是同一个单词 literals 的不同翻译而已，所以直接量也就是字面量，字面量就是直接量，不同书可能用的不同翻译而已。

原始值

原始类型的值称为原始值，原始值是不可变的。在 JavaScript 中，有两种数据类型：原始类型和引用类型，引用类型是对象。

原始类型包括：数字（Number）、字符串（String）、布尔（Boolean）、null、undefined。

表达式&语句

表达式可以理解成方程式，用于计算某个结果值或某种行为的代码可称为表达式，通常它们并不会改变程序状态，也不会影响程序后续执行流程，如算术表达式: x+1。

语句表示一个完整的行为，可以是多个表达式与关键字、运算符等的组合。

函数&方法

function 关键字声明的称为函数，但将函数置于对象内时，此时称它为对象的方法。

简单的理解，从不同的角度看待，如果是从函数本身，那么它就是个函数，如果是从对象角度看待，那么称某个函数是对象的方法，本质上没什么区别。但当调用时会有所区别。

另外，函数本质上也是一个对象。

构造函数

JavaScript 没有类似于 Java 的 class 机制，一个函数，当和 new 关键字一起使用时，此时称这个函数为构造函数。

也就是所有的函数都可以作为构造函数，当它和 new 关键字一起使用时，此时它的行为有区别于普通的函数调用。

全局对象

在前端里，当 js 的宿主是浏览器时，全局对象是 window。全局对象有几点特性：

1. js 文件中不在函数内声明的所有变量和函数都是作为全局对象的属性存在。

2. 全局对象的属性使用时，可以直接通过属性名访问，不必添加前缀，也就是不必像 window.name 这么使用。

3. js 文件中，函数外出现的 this 都指向全局对象 window。

全局属性

全局对象的属性就称全局属性，但这里的全局属性，更多的是在表示这种场景下为全局对象创建的属性：

<**script type="text/javascript"**>  
 **weixin** = dasuAndroidTv;  
</**script**>

当我们在函数外部直接对一个不存在的变量赋值操作时，此时等价于执行 this.weixin = dasuAndroidTv，而this指向全局对象，所以会自动为全局对象添加一个属性，属性名为 weixin。

所以，有时候，全局属性是特指这种场景下为全局对象创建的属性。之所以会用全局属性来称呼这种场景，是为了跟下面的全局变量和全局函数区分开。

全局变量

<**script type="text/javascript"**>  
 **var *weixin*** = dasuAndroidTv;  
</**script**>

像这种不在函数内声明的变量，称为全局变量。虽然，它们最终也都是作为全局对象 window 的属性而存在，但由于这种方式为 window 创建的属性和上述介绍的不加 var 关键字为全局对象添加属性的方式有本质上的区别。

所以，为了和上述介绍的全局属性概念区别开来，通常称这种通过 var 为全局对象创建的属性为全局变量。

既然全局对象的属性在任何地方都可以不加前缀的直接访问使用，所以全局变量在任何地方都可以被使用。

对于上述介绍的那种直接对不存在的属性进行赋值操作行为而创建的全局属性，是可通过 delete 动态删除的；

而通过 var 或 function 这种声明全局变量的方式为全局对象创建的属性，是不可通过 delete 删除的。

全局函数

<**script type="text/javascript"**>  
 **function** *add*() {}  
</**script**>

像这种不在函数内声明的函数，称为全局函数。它跟全局变量的特性、用意一模一样，区别只在于一个是变量、一个是函数而言。

包装对象

原始类型所对应的对象类型，类似于 Java 中的包装类。

因为原始类型不是对象，不可操作方法和属性，但可将其转换为对应的对象类型，此时称为包装对象，即可像操作对象一样操作这些转成包装对象的原始类型数据。

包装对象有：Number, Boolean, String

原型

由于在 JavaScript 中，除了原始类型，其余的皆为对象，所以它的继承双方只能都是对象，也就是说，对象也是继承自对象的，那么作为父类角色，作为被继承的那个对象，此时称它为原型。

所以，才说JavaScript是基于原型的继承语言。

原型就是类似于 Java 中父类的概念。

原型链

既然涉及继承，那么自然就有继承结构，这个结构在 JavaScript 中就称为原型链。

比如对象 a 继承自对象 b，对象 b 继承自对象 c，那么 a 的原型链就表示为：a -> b -> c（省略掉内置的继承关系）。

原型链用于当操作对象某个属性时，寻找该属性的来源。

作用域

作用域指的是变量和函数的作用域，下面统称变量，在 JavaScript 中，变量分两种：全局变量和局部变量。

全局变量指的是在函数外定义的变量，作用域是全局，在任何地方都可以使用，即使跨 js 文件中也可以使用，因为它们实际上是作为全局对象的属性存在，在前端里就是作为 window 的属性，而多个 <script> 中的不同 js 文件，都是共用同一个全局对象 window，自然就可以跨文件使用它的属性。

局部变量指函数内部定义的变量，或者函数形参，作用域为函数内。需要区别于Java中的局部变量，Java里的局部变量的作用域为块级作用域：只能在局部变量声明的代码块且声明位置之后使用；但在 JavaScript中，变量都有声明提前的特性，局部变量在函数内不管哪里都可以访问，即使声明的位置在末尾，或内嵌的代码块中。

作用域链

在 JavaScript 中，允许在函数内部继续定义函数，所以函数可以存在很深的嵌套层次，这里的嵌套层次不是指调用的嵌套，而是指函数声明的嵌套，A函数在B函数中定义，作为B函数的局部变量存在这种。

而内部函数是可以访问外部函数内的变量的，也可以访问全局的变量，那么当内部函数使用了某个外部变量，就会借助作用域链，沿着作用域链中寻找这个外部变量究竟是外部函数内的变量，还是全局变量。

从原理上解释，每个函数调用时，都会创建一个函数执行上下文，执行上下文中存储着当前上下文中的所有变量，作用域链，就是将具有嵌套层次的函数的上下文中的变量串接起来的存在。

还是要反复强调，上面的嵌套层次指的不是函数调用时的嵌套层次，而是函数定义时的嵌套层次。可以将这里有嵌套层次关系的函数理解成Java中的内部类。

闭包

MDN 中的文章对其的解释为：闭包是函数和声明该函数的词法环境的组合，这个环境包含了这个闭包创建时所能访问的所有局部变量。

某篇文章中看到过这么一种解释：闭包是代码块和创建该代码块的上下文中数据的结合。

闭包的概念不怎么好理解，有的文章里会说，函数就是闭包，有的说内嵌的函数是闭包，这里先不做评价，先尝试着从上面两种解释中去理解一下，看能否理解，后续会专门写一篇来讲讲。

## 数据类型

JavaScript 里有两种数据类型：原始类型和对象类型

原始类型

原始类型里包括：

* 数字（Number）
* 布尔（Boolean）
* 字符串（String）
* null
* undefined

布尔类型和字符串类型跟 Java 没多大区别，主要就讲一下数字类型、null 和 undefined。

数字

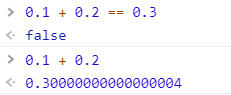
JavaScript 里不像 Java 一样会区分 int，float，long 等之类的数字类型，全部都归属于一个 Number 数字类型中。之所以不加区分，是因为，在 JavaScript 里，所有的数字，不管整数还是小数，都用浮点数来表示，采用的是 [IEEE 754标准定义的 64 位浮点格式](https://blog.csdn.net/abcdu1/article/details/75095781)表示数字。

那么，它所能表示的数值范围就是有限的，除了正常数值外，还有一些关键字表示特殊场景：

* Infinity（正无穷）
* -Infinity（负无穷）
* NaN（非数值）

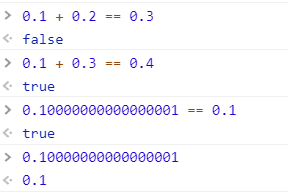
对于小数，支持的浮动小数表示法如下：

3.14        
-.2345789 // -0.23456789  
-3.12e+12  // -3.12\*1012  
.1e-23    // 0.1\*10-23=10-24=1e-24

另外，因为浮点表示法只能精确的表示如：1/2, 1/8, 1/1024 这类分数，对于 1/10 这种小数只能取近视值表示，因此在 JavaScript 里有个经典的有趣现象：

0.1 + 0.2 在 JavaScript 里是不等于 0.3 的，因为用浮点表示法，无法精确表示 0.1 和 0.2，所以会舍弃一些精度，两个近似值相加，计算结果跟实际算术运算结果自然有些偏差。

上图里也显示了，在 JavaScript 里，0.1 + 0.2 的运算结果是 0.30000000000000004。

那么，是否所有非 1/2, 1/4, 1/8 这类 1/2^n 小数的相加结果最后都不会等于实际运算结果呢？

0.1, 0.2, 0.3 都是浮点数无法精确表示的数值，所以在 JavaScript 里都是以近似值存储在内存中，那么，为何 0.1 + 0.2 ！= 0.3，但 0.1 + 0.3 == 0.4 ？

这是因为，JavaScript 里在处理这类小数时，允许一定程度的误差，比如 0.10000000000000001 在允许的误差中，所以在 JavaScript 里就将这个值当做 0.1 来看待处理。

所以如果两个是以近似值存储的小数运算之后的结果，在误差允许范围内，那么计算结果会按实际算术运算结果来呈现。

https://upload-images.jianshu.io/upload_images/1924341-bde7ab59fe415b0f.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240总之，不要用 JavaScript 来计算一些小数计算且有精度要求，如果非要不可，那么建议先将小数都按比例扩展到整数运算后，再按比例缩小，如：

还有另外一点，由于 JavaScript 的变量是不区分类型的，那么当有需要区分某个变量是不是数字时，可用内置的全局函数来处理：

* isNaN() -- 如果参数是 NaN 或者非数字值（如字符串或对象），返回 true
* isFinite() -- 如果参数不是 NaN，或 Infinity 或 -Infinity 时返回 true，通俗理解，参数是正常的数字

null

跟 Java 一样，JavaScript 里也有 null 关键字，但它的含义和用法却跟 Java 里的 null 不太一样。

在 Java 里，声明一个对象类型的变量后，如果没有对该变量进行赋值操作，默认值为 null，所以在程序中经常需要对变量进行判空处理，这是 Java 里 null 的场景。

但在 JavaScript 中，声明一个变量却没有进行赋值操作的话，默认值不是 null，而是 undefined。

那么，什么场景下，变量的值会是 null 呢？我可以告诉你，没有，没有任何场景下某个变量或某个属性的值默认会是 null，除非你在程序中手动将某个变量赋值为 null，那么此时这个变量的值才会是 null。

所以，才有些书本中会说，null 是表示程序级、正常的或在意料之中的值的空缺。意思就是说，null 是 JavaScript 设计出来的一个表示空值含义的数据类型，用来给你在程序中当有需要给某个变量手动设置为空值的场景时使用。

举个通俗的例子，对于数字类型变量，你可以用 0 表示它的初始值；对于字符串类型变量，你可以用 "" 表示它的初始值；那么对于对象类型，当你也需要给它一个表示空值无具体含义的初始值时，你就可以给它赋值为 null。

这也是为什么用 typeof 运算符获取 null 的数据类型时，会发现输出的是 Object。因为 null 实际上是个实际存在的数据值，只是它的含义是空值的意思，用于赋值给对象类型的变量。

那么，也就是说，不能沿用 Java 里使用 null 的思维应用到 JavaScript 中了，null 可以作为初始值赋值给变量，但变量如果没有进行初始化，默认值不再是 null 了，这点是 JavaScript 有区别于 Java 的地方，需要注意一下。

不然再继续挪用 Java 的使用 null 思维，可能在编程中，会遇到一些意料外，没想通的问题。

undefined

如果声明了一个变量，缺没有对这个变量进行赋值操作，那么这个值默认就是 undefined。

那么在 Java 中的判空操作来判断变量是否有进行初始化的行为在这里就是对应判断变量的值是否为 undefined 的，但实际上，在 JavaScript 里，由于 if 判断语句接收的为真值，而不像 Java 只支持布尔类型，所以基本没有类似 Java 的判空的编程场景。

undefined 还有另外一种场景：

当访问对象中不存在的属性时，此时会输出 undefined，表示这个属性并未在对象中定义。

针对这种场景，undefined 可用于判断对象中是否含有某些指定的属性。

总结一下 null 和 undefined：

* null 是用于在程序中，如果有场景需要，如某个变量在某种条件下需要有一个表示为空值含义的取值，此时，可手动为该变量赋值为 null；
* 当声明某个变量，却没有对其进行赋值初始化操作时，这个变量默认为 undefined
* 当访问对象某个不存在的属性时，会输出 undefined，可用于判断对象中是否含有指定属性

对象类型

除了原始类型外，其余都是对象类型，但有一些内置的对象类型，所以大概可以这么表示

* 对象类型（Object）
  + 函数（Function）
  + 数组（Array）
  + 日期（Date）
  + 正则（RegExp）
  + ...

也就是，在 JavaScript 里，函数和数组，本质上也是对象。

变量相关

由于我本身有 Java 的基础了，所以 JavaScript 一些很基础的语法我可能会漏掉了，但影响不大。

弱类型

虽然 JavaScript 中有原始类型和对象类型，而且每个分类下又有很多细分的数据类型，但它实际上是一门弱类型语言，也叫动态语言。也就是说，使用变量时，无需指明变量是何种类型，运行期间会自动确定。

变量声明

既然使用变量时不必指明变量的数据类型，那么自然没有类似于 Java 中那么多种的变量声明方式，在 JavaScript 中声明变量很简单，都是通过 var 来：

var name = dasu;

ES5 中，声明变量的方式就是通过 var 关键字，而且同一变量重复声明不会出问题，会以后面声明的为主。

变量的提前声明

先看段代码：

<script type="text/javascript">  
    console.log(a);  //输出 undefined  
    var a = 1;  
    console.log(a);  //输出 1  
    b();  
    function b() {  
        console.log(a); //输出 undefined  
        var a = 2;  
        console.log(a); //输出 2  
   }     
</script>

JavaScript 中有变量的提前声明特性，也就是在代码开始执行前，所有通过 var 或 function 声明的变量和函数都已经提前声明了（下面统称变量），所以在声明语句之前访问声明的这个变量并不会抛异常。

但提前的只有变量的声明，变量的赋值初始化操作并没有提前，所以第一行代码输出变量 a 的值时，因为变量已经被提前声明了，但没赋值，按照上面介绍的，此时变量 a 值为 undefined，当赋值语句执行完，输出自然就是赋值的 1 了。

同样，由于 b 函数已经被提前声明了，所以可以在声明它的位置之前就调用函数了，而函数调用后，开始执行函数内的代码时，也同样会有变量提前声明的特性。

因此，在执行函数内第一行代码时，输出的变量 a 是函数内声明的局部变量，而不是函数外部的变量，这点行为跟 Java 不一样，需要注意一下。

有些脚本语言并没有变量声明提前的特性，使用的变量或函数只能在声明了它的位置之后才能使用，这是 JavaScript 区别它们的一点。

全局属性

上面说过，声明变量时是通过 var 关键字声明，那如果漏掉 var 呢，看个例子：

<script type="text/javascript">  
    //console.log(a); //抛异常，因为没有找到a变量  
    a = 1;  
    b();  
    function b() {  
        console.log(a); //输出 1  
        a = 2;  
        console.log(a); //输出 2  
   }  
    console.log(a); //输出 2  
</script>

第一行代码如果不注释掉，那么它执行的结果会是抛出一个异常，因为没有找到 a 变量。

接着执行了 a = 1，a 是一个不存在的变量，直接对不存在的变量进行赋值语句，其实是会自动对全局对象 window 动态添加了一个 a 属性并赋值，所以后续调用了 b 函数，函数里操作的 a 其实都是来自全局对象 window 的属性 a，所以在函数内对 a 进行的操作结果，当函数执行结束后，最后再次输出 a 才会是 2。

这其实是因为对象的特性导致的，在对象一节会来讲讲，但这里要清楚一点，切记声明使用变量时，不要忘记在前面要使用 var。

另外，顺便提一下，第一行被注释掉的代码，如果换成输出 this.a，那么此时程序是不会抛异常的，而是输出 undefined，这是因为前面也有稍微提过，访问对象不存在的属性时，会输出 undefined，都是在讲对象时会来说说。

变量作用域

ES5 中，变量有两种作用域，全局作用域和函数内作用域。

在函数外声明的变量都具有全局作用域，即使跨 js 文件都能够访问；而在函数内声明的变量，不管声明变量的语句在哪个位置，整个函数内都可以访问该变量，因为有变量的提前声明特性，所以是函数内作用域。

由于在 JavaScript 中，同一变量的重复声明不会出问题，所以对于全局变量而言，在多人协作，多模块编程中，很容易造成全局变量冲突，即我在我写的 js 文件中声明的 a 全局变量，其他人在其他 js 文件中，又声明了 a 全局变量，对于浏览器而言，它就只是简单的以后声明的为主。

但对于程序而已，就会发生不可控的问题，而且极难排查，所以要慎用全局变量。当然针对这种情况也有很多解决方案，后续讲到函数一节中会来讲讲。

包装对象

JavaScript 里的对象具有很多特性，比如可以动态为其添加属性等等。但原始类型都不具有对象的这些特性，那么当需要对原始类型也使用类似对象的特性行为时，这时候包装对象就出现了。

包装对象跟 Java 中的包装类基本是类似的概念，原始数据类似对应的对象类型的值称为包装对象：

* 数字类型 -> Number 包装对象
* 布尔类型 -> Boolean 包装对象
* 字符串类型 -> String 包装对象
* null 和 undefined 没有包装对象，所以不允许对 null 和 undefined 的变量进行属性操作

接下来就讲讲原始类型和包装对象之间的转换，存在两种场景，程序运行期间自动转换，或者手动显示的进行转换。

隐式转换

因为属性是对象才有的特性，所以当对某个原始类型的变量进行属性操作时，此时会临时创建一个包装对象，属性操作结束后销毁包装对象。

看个例子：

var s = "test";   //创建一个字符串，s是原始类型的变量  
s.len = 4;   //对s动态添加一个属性len并赋值，执行这行代码时，会临时创建一个包装对象，所以这里的s已经不是上面的原生类型变量，进行了一次自动转换  
console.log(s.len);  //输出 undefined，上一行虽然进行了一次包装对象的自动转换，但是是临时的，那一行代码执行结束，包装对象就销毁了。所以这一行又对s原始类型变量进行属性操作，又再一次创建一个临时的包装对象

需要注意一点，当对原始类型的操作进行属性操作时，会创建一个临时的包装对象，注意是临时的，属性操作完毕，包装对象就销毁了。下一次再继续对原始类型进行属性操作时，创建的又是新的一个临时包装对象。

显示转换

除了隐式的自动转换外，也可以显示的手动转换。

如果是原始类型 -> 包装类型的转换，可使用相对应的包装对象的构造函数方式：

var a = new Number(123);  
var b = new Boolean(true);  
var s = new String("dasu");

此时，a, b, s 都是对象类型的变量了，可以对它们进行一些属性操作。

如果是包装类型 -> 原始类型的转换，使用不加 new 的调用全局函数的方式：

var aa = Number(a);  
var bb = Boolean(b);  
var ss = String(s);

在后续讲函数时会讲到，一个函数被调用的方式有多种：其中，有跟 new 关键字一起使用，此时叫这个函数为构造函数；如果只是简单的调用，此时叫函数调用；如果是作为对象的属性被调用，此时称方法调用；不同的调用方式会有一些区别。

所以，这里当包装对象使用构造函数方式使用时，可以显示的将原始类型数据转换为包装对象；但如果不作为构造函数，只是简单的函数调用，其实就是将传入的参数转换为原始类型，参数不单可以是包装对象类型，也可以是其他类型。

数据类型间相互转换

上面讲了原始类型与包装对象间的相互转换，其实本质上也就是不同数据类型间的相互转换。

按数据类型细分来讲的话，一共包括：数字、布尔、字符串、null、undefined、对象（函数、数组等），由于 JavaScript 是弱类型语言，运行期间自动确定变量类型，所以，其实这些不同数据类型之间都存在相互转换的规则。

先看个例子：

10 + " objects";    // => "10 objects",这里的 10 自动转换成 "10"  
"7" \* "4";          // => 28, 这里的两个字符串都自动转换为数字  
var n = 1 - "x";    // => NaN，字符串 "x" 无法转换为数字  
n + " objects";     // => "NaN objects", NaN 转换为字符串 "NaN"

数字可以转换成字符串，字符串也可以转换为数字，原始类型也可以转换为对象类型等等，反正不同类似之间都可以相互转换。

基本转换规则

具体的规则，可以参见下表：

| **待转换值** | **转换为字符串** | **转换为数字** | **转换为布尔值** | **转换为对象** |
| --- | --- | --- | --- | --- |
| undefined | "undefined" | NaN | false | throws TypeError |
| null | "null" | 0 | false | throws TypeError |
| true（布尔->其他） | "true" | 1 | -- | new Boolean(true) |
| false（布尔->其他） | "false" | 0 | -- | new Boolean(false) |
| ""（空字符串->其他） | -- | 0 | false | new String("") |
| "1.2"（字符串内容为数字->其他） | -- | 1.2 | true | new String("1.2") |
| "dasu"（字符串内容非数字->其他） | -- | NaN | true | new String("dasu") |
| 0（数字->其他） | "0" | -- | false | new Number(0) |
| -0（数字->其他） | "0" | -- | false | new Number(-0) |
| 1（数字->其他） | "1" | -- | true | new Number(1) |
| NaN | "NaN" | -- | false | new Number(NaN) |
| Infinity | "Infinity" | -- | true | new Number(Infinity) |
| -Infinity | "-Infinity" | -- | true | new Number(-Infinity) |
| {}（对象 -> 其他） | 单独讲 | 单独讲 | true | -- |
| [] （数组 -> 其他） | "" | 0 | true | -- |
| [1] （一个数字元素的数值 -> 其他） | "1" | 1 | true | -- |
| ['a'] （普通数组 -> 其他） | 使用join()方法 | NaN | true | -- |
| function(){} （函数 -> 其他） | 单独讲 | NaN | true | -- |

总之不同类型之间都可以相互转换，除了 null 和 undefined 不能转换为对象之外，其余都可以。

那么什么时候会进行这些转换呢？

其实在程序运行期间，就不断的在隐式的进行着各种类型转换，比如 if 语句中不是布尔类型时，比如算术表达式两边是不同类型时等等。

那么，如何进行手动的显示转换呢？

在上一小节中，其实有稍微提过了，就是使用：

* Number()
* String()
* Boolean()
* Object()

注意是以函数调用方式使用，即不加 new 关键字的使用方式。参数传入的值就是表示上表中第一列待转换的值，而四种不同的函数，就对应着上表中右边四列的转换规则。如

Number("dasu")  // => NaN，表示待转换值为字符串 "dasu"，需要转换为数字类型，按照上表规则，转换结果NaN  
String(true)    // => "true"，同理，将布尔类型true转为字符串类型  
Boolean([])     // => true，将空数组转为布尔类型  
Object(3)       // => new Number(3)，将数字类型转为包装对象

换句话说，这四个函数，其实就是用于将任意类型转换为函数对应的类型，比如 Number() 函数就是用于将任意类型转为数字类型，至于具体转换规则，就是按照表中的规则来进行转换。

一般来说，应该可以不用将表中所有的转换规则都详记，需要自己手动转换的场景应该也不多，记住一些常用基本的就行了，至于哪些是常见的，写多了就清楚了，比如数字类型 -> 布尔类型，对象类型 -> 布尔类型等。

对象转换为原始值规则

所有的数据类型之间的转换，就对象转换到原始值的规则会复杂点，其余的需要的时候，看一下表就行了。

* 对象 -> 布尔

首先，所有的对象，不管的函数、数组还是普通对象，只要这个对象是定义后存在的，那么它转换为布尔值都是 true，所以对象转布尔也很简单。反正就记住，对象存在，那么转布尔就为 true。

所以，即使一个布尔值 false，先转成包装对象 new Boolean(false)，再从包装对象转为布尔值，那么此时，包装对象转布尔后是 true，因为包装对象存在，就这么简单，不关心这个包装对象原本是从布尔 false 转来的。

* 对象 -> 字符串

对象转字符串，主要是需要借助两个方法：

1. 如果对象具有 toString()，则调用这个方法，如果调用后返回了一个原始值，那么就将这个原始值转为字符串，转换结束。
2. 如果对象没有 toString() 方法，或者调用该方法返回的并不是一个原始值，那么调用对象的 valueOf() 方法，同样，如果调用后返回一个原始值，那么将原始值转为字符串后，转换结束。
3. 否则，抛类型错误异常。

这就是对象转字符串的规则，有些内置的对象，比如函数对象，或数组对象就可能会对这两个方法进行重写，对于自定义的对象，也可以重写这两个方法，来手动控制它转成字符串的规则。

* 对象 -> 数字

对象转数字的规则，也是需要用到这两个方法，只是它将步骤替换了下：

1. 如果对象具有 valueOf() 方法，且调用后返回一个原始值，那么将这个原始值转为数字，转换结束。
2. 如果对象没有 valueOf() 方法，或者调用后返回的不是原始值，那么看对象是否具有 toSring() 方法，且调用它后返回一个原始值，那么将原始值转为数字，转换结束。
3. 否则，抛类型错误异常。

## 运算符

程序中的代码其实就是利用各种运算符来辅助完成各种指令功能，在 JavaScript 中，有一些不同于 Java 中的运算符处理，这次就来讲讲这些运算符。

由于我已经有了 Java 的基础了，本节不会讲基础的运算符介绍，比如算术表达式中的加减乘除取余等、关系表达式中的大于小于等、逻辑表示式中的自增、自减、移位等等，这些基础运算符的含义、用法、优先级这些跟 Java 基本没有区别，所以就不介绍了。

下面着重讲一些在 JavaScript 比较不同的行为的一些运算符：

"+" 运算符

任何数据类型的变量都可以通过 "+" 运算符来进行计算，所以它有一套处理规则，通常要么就是按数字的加法运算处理、要么就是按照字符串的拼接处理，处理规则如下：

1. 如果操作数中存在对象类型，先将其按照上节介绍的转换规则，转成原始值；
2. 如果操作数已经全部是原始值，此时如果有字符串类型的原始值，那么将两个原始值都转为字符串后，按字符串拼接操作处理；
3. 如果操作数已经全部是原始值且没有字符串类型的，那么将操作数都转为数字类型后，按数字的加法处理;
4. NaN 加上任意类型的值后都是 NaN.

以上的处理规则是针对于通过 "+" 运算符处理两个操作数的场景，如果一个表达式中存在多个 "+" 运算符，那么分别以优先级计算过程中，每一次计算 "+" 运算符的两个操作数使用上述规则进行处理。

举个例子：

1 + 2    // => 3, 因为操作数都是数字类型的原始值  
1 + "2"  // => "12"，因为操作数中存在字符串类型的原始值，所以是按字符串拼接来处理  
1 + {}   // => "1[object Object]"，因为有操作是对象类型，先将其转为原始值，{} 转为原始值为字符串 "[object Object]"，所以将操作数都转为字符串后，按字符串拼接处理  
1 + true // => 2，因为两个都是原始值，且没有字符串类型，所以将 true 转为数字类型后是 1，按加法处理  
1 + undefined // => NaN，因为 undefined 转为数字类型后为 NaN，NaN 与任何数运算结果都为 NaN   
​  
1 + 2 + " dasu"  // => "3 dasu"， 因为先计算 1+2=3，然后再计算 3 + " dasu"，所以是 "3 dasu"  
1 + (2 + " dasu") // => "12 dasu"，因为先计算 2 + " dasu" = "2 dasu"，再计算 1 + "2 dasu" = "12 dasu"

因为 "+" 运算符在编程中很常见，也很常用，而 JavaScript 又是弱类型语言，变量无需声明类型，那么程序中，"+" 运算符的两个操作数究竟是哪两种类型在进行计算，结果又会是什么，这点在心里至少是要明确的。

"==" 和 "===" 相等运算符

"==" 和 "===" 都是用于判断两个操作数是否相等的运算符，但它们是有区别的。

"==" 比较相等的两个操作数会自动进行一些隐式的类型转换后，再进行比较，俗称不严格相等。

"===" 比较相等的两个操作数，不会进行任何类型转换，相等的条件就是类型一样，数值也一样，所以俗称严格相等。

而 "!=" 和 "!==" 自然就是这两个相等运算符的求反运算。下面分别来看看：

"==="

当通过这个运算符来比较两个操作数是否严格相等时，具体规则如下：

* 如果两个操作数的类型不相同，则它们不相等
* 如果其中一个操作数是 NaN 时，则它们不相等（因为 NaN 跟任何数包括它本身都不相等）
* 如果两个操作数都是对象类型，那么只有当两个操作数都指向同一个对象，即它们的引用一样时，它们才相等
* 如果两个操作数都是字符串类型时，当字符串一致时，在某些特殊场景下，比如具有不同编码的 16 位值时，它们也不相等，但大部分情况下，字符串一致是会相等，但要至少清楚不是百分百
* 如果两个操作数都是布尔类型、数字类型、null、undefined，且值都一致时，那它们相等

总之，这里的规则跟 Java 里的相等比较类似，Java 里没有严格不严格之分，它处理的规则就是按照 JavaScript 这里的严格相等来处理，所以大部分比较逻辑可参考 Java。

需要注意的就是，NaN 与任何数包括它本身也不相等、同一个字符串内容可能会有不同的编码值，所以并不是百分百相等。

"=="

这个通常称为不严格相等，当比较是否相等的两个操作数的数据类型不一样时，会尝试先进行转换，然后再进行比较，相比于上面的 "===" 严格相等运算符来说，它其实就是放宽了比较的条件，具体规则如下：

* 如果两个操作数的类型一样，那么规则跟 "===" 一样
* 如果一个类型是 null，另一个类型是 undefined，此时，它们也是相等的
* 如果一个类型是数字，另一个类型是字符串，那么先将字符串转为数字，再进行比较
* 如果一个类型是布尔，先将布尔转成 1（true）或 0（false），然后再根据当前两个类型是否需要再进一步处理再比较
* 如果一个类型是对象，那么先将对象转换成原始值，然后再根据当前两个类型是否需要再进一步处理再比较

总之，"==" 的比较相对于 "===" 会将条件放宽，下面可以看些例子：

null === undefined    // => false，两个类型不一样  
null == undefined     // => true，不严格情况下两者可认为相等     
​  
1 == "1"              // => true，"1" 转为数字 1 后，再比较  
1 == [1]              // => true，[1] 先转为字符串 "1"，此时等效于比较 1 == "1"，所以相等  
2 == true             // => false，因为 true 先转为数字 1，此时等效于比较 2 == 1

"&&” 逻辑与

逻辑与就是两个条件都要满足，这点跟 Java 里的逻辑与操作 && 没有任何区别。

但 JavaScript 里的逻辑与 && 操作会更强大，在 Java 里，逻辑与 && 运算符的两个操作数都必须是关系表达式才行，而且整个逻辑与表达式最终的结果只返回 true 或 false。

但在 JavaScript 里，允许逻辑与 && 运算符的两个操作数是任意的表达式，而且整个逻辑与 && 表达式最终返回的值并不是 true 或 false，而是其中某个操作数的值。

什么意思，来看个例子：

x == 0 && y == 0

这是最基本的用法，跟 Java 没有任何区别，当且仅当 x 和 y 都为 0 时，返回 true，否则返回 false。

上面那句话，是从这个例子以及延用 Java 那边对逻辑与 && 运算符的理解所进行的解释。

但实际上，在 JavaScript 里，它是这么处理逻辑与 && 运算符的：

* 如果左操作数的值是假值，那么不会触发右操作数的计算，且整个逻辑与 && 表达式返回左操作数的值
* 如果左操作数的值是真值，那么整个逻辑与 && 表达式返回右操作数的值
* 假值真值可以通俗的理解成，上节介绍各种数据类型间的转换规则中，各类型转换为布尔类型的值，转为布尔后为 true，表示这个值为真值。反之，为假值。

所以，按照这种理论，我们再来看看上面那个例子，首先左操作数是个关系表达式：x == 0，如果 x 为 0，这个表达式等于 true，所以它为真值，那么整个逻辑与 && 表达式返回右操作数的值。右操作数也是个关系表达式：y == 0，如果 y 也等于 0，右操作数的值就为 true，所以整个逻辑与 && 表达式就返回 true。

虽然结果一样，但在 JavaScript 里对于逻辑与 && 表达式的解释应该按照第二种，而不是按照第一种的 Java 里的解释。如果还不理解，那么再来看几个例子：

function getName() {  
 return "dasu"  
}  
​  
null && getName()   //输出 => null，因为左操作数 null 转成布尔是 false，所以它是假值，所以逻辑与 && 直接返回左操作数的值 null  
​  
getName && getName()  //输出 => "dasu"，因为左操作数是一个函数对象，如果该函数对象被声明定义了，那么转为布尔值就是 true，所以逻辑与 && 表达式返回右操作数的值，右操作数是 getName()，调用了函数，返回了 "dasu"，所以这个就是这个逻辑与 && 表达式的值。

第一个逻辑与表达式：null && getName() 会输出 null，是因为左操作数 null 转成布尔是 false，所以它是假值，所以逻辑与 && 直接返回左操作数的值 null。

第二个逻辑与表达式：getName && getName() 会输出 "dasu"，是因为左操作数是一个函数对象，如果该函数对象被声明定义了，那么转为布尔值就是 true，所以逻辑与 && 表达式返回右操作数的值，右操作数是 getName()，调用了函数，返回了 "dasu"，所以这个就是这个逻辑与 && 表达式的值。

所以 JavaScript 里的逻辑与 && 表达式会比 Java 更强大，它有一种应用场景：

* 应用场景

function queryName(callback) {  
    //...   
      
    //回调处理  
    callback && callback();  
}

在 Java 中，我们提供回调机制的处理通常是定义了一个接口，然后接口作为函数的参数，如果调用的时候，传入了这个接口的具体实现，那么在内部会去判断如果传入的接口参数不为空，就调用接口里的方法实现通知回调的效果。

在 JavaScript 里实现这种回调机制就特别简单，通过逻辑与 && 表达式，一行代码就搞定了，如果有传入 callback 函数，那么 callback 就会是真值，逻辑与 && 表达式就会去执行右操作数的 callback()。

当然，如果你想严谨点，你可以多加几个逻辑与 && 表达式来验证传入的 callback 参数是否是函数类型。

"||" 逻辑或

逻辑或 || 跟逻辑与 && 就基本是一个东西了，理解了上面讲的逻辑与 && 运算符的理论，那么自然也就能够理解逻辑或 || 运算符了。

它们的区别，仅在于对表达式的处理，逻辑或 || 表达式是这么处理的：

* 如果左操作数的值是真值，那么不会触发右操作数的计算，且整个逻辑或 || 表达式返回左操作数的值
* 如果左操作数的值是假值，那么整个逻辑或 || 表达式返回右操作数的值
* 假值真值可以通俗的理解成，上节介绍各种数据类型间的转换规则中，各类型转换为布尔类型的值，转为布尔后为 true，表示这个值为真值。反之，为假值。

这里就直接来说下它的一个应用场景了：

* 应用场景

function queryNameById(id) {  
    //参数的默认值  
    id = id || 10086;  
    //...  
}

处理参数的默认值，如果调用函数时，没有传入指定的参数时。

当然，还有其他很多应用场景。总之，善用逻辑与 && 和逻辑或 || 运算符，可以节省很多编程量，同时实现很多功能。

"," 逗号运算符

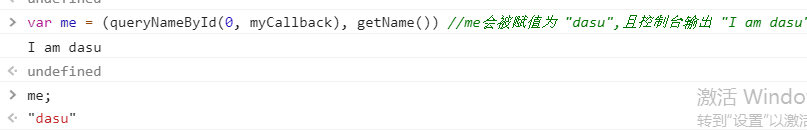
在 Java 中，"," 逗号只用于在声明同一类型变量时，可同时声明，如：

int a, b, c;

在 JavaScript 里，"," 逗号运算符同样具有这个功能，但它更强大，因为带有 "," 逗号运算符的表达式会有一个返回值，返回值是逗号最后一项操作数的值。

逗号运算符跟逻辑与和逻辑或唯一的区别，就在于：逗号运算符会将每一项的操作数都进行计算，而且表示式一直返回最后一项的操作数的值，它不管每个操作数究竟是真值还是假值，也不管后续操作数是否可以不用计算了。

举个例子：

  
function getName() {  
 return "dasu"  
}  
​  
function queryNameById(id, callback) {  
    id = id || 10086;  
    callback && callback();  
}  
​  
function myCallback() {  
    console.log("I am dasu");  
}  
​  
var me = (queryNameById(0, myCallback), getName()) //me会被赋值为 "dasu",且控制台输出 "I am dasu"

变量 me 会被赋值为 "dasu"，且控制台输出 "I am dasu"。

typeof 运算符

返回指定操作数的数据类型，例：



在 JavaScript 中数据类型大体上分两类：原始类型和引用类型。

原始类型对应的值是原始值，引用类型对应的值为对象。

对于原始值而言，使用 typeof 运算符可以获取原始值所属的原始类型，对于函数对象，也可以使用 typeof 运算符来获取它的数据类型，但对于其他自定义对象、数组对象、以及 null，它返回的都是 object，所以它的局限性也很大。

delete 运算符

delete 是用来删除对象上的属性的，因为 JavaScript 里的对象有个特性，允许在运行期间，动态的为对象添加某个属性，那么，自然也允许动态的删除属性，就是通过这个运算符来操作。

这个在对象一节还会拿出来讲，因为并不是所有的属性都可以成功被删除的，属性可以设置为不可配置，此时就无法通过 delete 来删除。

另外，之前也说过，在函数外声明的全局变量，本质上都是以属性的形式被存在在全局对象上的，但这些通过 var 或 function 声明的全局变量，无法通过 delete 来进行删除。

之前也说过，如果在声明变量时，不小心漏掉了 var 关键字，此时程序并不会出异常，因为漏掉 var 关键字对一个不存在的变量进行赋值操作，会被 js 解释器认为这行代码是要动态的为全局对象添加一个属性，这个动态添加的属性就可以通过 delete 来进行删除，因为动态添加的属性默认都是可配置的。

instanceof 运算符

在 Java 中，可以通过 instanceof 运算符来判断某个对象是否是从指定类实例化出来的，也可以用于判断一群对象是否属于同一个类的实例。

在 JavaScript 中有些区别，但也有些类似。

var b = {}  
function A() {}  
A.prototype = b;  
var a = new A();  
if (a instanceof A) { //符合，因为 a 是从A实例化的，继承自A.prototype即b  
    console.log("true");   
}  
​  
function B() {}  
B.prototype = b;  
var c = new B();  
if (c instanceof A) {//符合，虽然c是从B实例化的，但c也同样继承自b，而A.prototype指向b，所以满足  
    console.log("true");  
}  
if (c instanceof Object) {//符合，虽然 c 是继承自 b，但 b 继承自 Object.prototype，所以c的原型链中有 Object.prototype  
    console.log("true");  
}

在 JavaScript 中，instanceof 运算符的左侧是对象，右侧是构造函数。但他们的判断是，只要左侧对象的原型链中包括右侧构造函数的 prototype 指向的原型，那么条件就满足，即使左侧对象不是从右侧构造函数实例化的对象。

例子代码看不懂么事，这个在后续介绍原型时，还会再拿出来说，先清楚有这么个运算符，运算符大概的作用是什么就可以了。

## 对象

在 JavaScript 除了原始数据类型外，其余均是对象，函数是对象，数组也是对象；继承通过对象来实现，构造函数也通过对象来实现，所以对象在 JavaScript 里有着很重要的角色，理解和掌握对象的一些特性，对于掌握 JavaScript 这门语言有着很大的帮助。

Java 里对象有属性和方法之分，但在 JavaScript 中，只存在属性，变量是属性，方法也是属性，对于 JavaScript 来说，对象，其实只是一堆属性的无序集合而已，外部可通过对象来操作各种属性，只不过有的属性，它的值是函数类型，所以这时可叫它为对象的方法。

对象的每个属性，都是一个 key-value 的形式，属性名和属性值。而属性，又分自有属性和继承属性，自有属性是指对象本身自己拥有的属性，而继承属性是指继承的属性。

对象分类

一般来说，有三类对象，分别是内置对象、宿主对象、自定义对象：

* 内置对象：是指语法标准规范中内置实现的一些对象，例如函数、数组、正则、日期等这些内置对象；
* 宿主对象：是指 JavaScript 解释器所嵌入的宿主环境，在前端里，一般来说宿主环境就是浏览器，浏览器也会定义一些内置对象，比如 HTMLElement 等；
* 自定义对象：开发人员自行实现的对象。

创建对象

创建对象有三种方式：对象直接量、构造函数、Object.create()

* 对象直接量

这是最简单的一种创建对象的方式，在代码中，直接通过 {} 形式创建一个对象，如：

var book = {  
    "main title": "JavaScript",  
    'sub-title': "The Definitive Guide",  
    "pages": 900,  
    author: {  
        firstname: "David",  
        surname: "Flanagan"  
   }  
};

上述代码中，等号右侧 {} 代码块形式定义的对象方式，就叫对象直接量。代码中，每出现一次对象直接量，会直接创建一个新的对象，对象的属性就是对象直接量中定义的。

定义属性时，有几点需要注意一下，属性名也就是 key 值，可加引号也可不加引号，但如果属性名使用到一些保留字时，就肯定需要加引号。

属性值可以是 JavaScript 中的任意类型，原始类型、对象类型都可以。

* 构造函数

构造函数就是通过 new 关键字和函数一起使用时，此时的函数就称构造函数，用途在于创建一个新的对象。具体在后续讲函数时会详细讲解。

这里可以看个例子：

var o = new Object();  
var o1 = new Object;  
var a = new Array();  
​  
function Book() {}  
var b = new Book();

通过 new 关键字和函数一起使用时，就可以创建新对象，例子中的 Object 和 Array 是内置的构造函数，也可以自定义构造函数，其实就是自定义一个函数，让它和 new 关键字一起使用就可以了。

通过构造函数方式如何给新创建的对象添加一些初始化的属性，这些细节和原理在函数一节中再讲，这里需要注意一点的就是，当不往构造函数传参数时，此时括号是允许可以省略的。

另外，第一种对象直接量的方式创建对象，其实，本质上也是通过构造函数：

var o = {name:"dasu"}   
//等效于  
var o = new Object();  
o.name = "dasu";

对象直接量其实是一种语法糖，可以通俗的理解，JavaScript 为方便我们创建对象，封装的一层工具，其内部的本质实现也是通过构造函数。

* Object.create()

你可以把 Object.create() 理解成 Java 中的静态方法。

通过这种方式，可以创建一个新的对象，参数是指定对象要继承的目标对象，这个被继承的对象，在 JavaScript 里被称为原型。

举个例子：

var o = Object.create(new Object());  //创建一个新对象，让它继承自一个空对象

通过构造函数创建的对象，其继承关系是由构造函数决定的，而 Object.create() 方式，可自己手动指定继承关系。当然，并不是说，构造函数就无法自己指定继承关系。

原型

原型可以理解成 Java 中的父类概念。

在 JavaScript 中，对象也存在继承关系，继承的双方都是对象，对象是从对象上继承的，被继承的那个对象称作原型。所以，有一种描述说，JavaScript 是基于原型的继承。

在 Java 中，是通过 extends 关键字实现继承关系，那么在 JavaScript 里呢？

自然也有类似的用来指定对象的继承关系，这就取决于创建对象的方式，上面说过，创建对象有三种方式：对象直接量、构造函数、Object.create()，但由于对象直接量本质上也是通过构造函数，所以其实就两种。

对于构造函数创建的对象来说，因为每个函数都有一个 prototype 属性，prototype 是它的属性名，属性值是一个对象，这个对象就是原型，就是通过该构造函数创建出来的新对象的继承来源。

我们可以通过修改构造函数的 prototype 属性值来达到指定对象继承关系的目的，如果不修改，那么内置的构造函数如 Object 或 Array 这些都已经有默认指定的 prototype 属性值了，也就是创建内置对象时，这个对象已经具有一定的默认继承结构了。

对于 Object.create() 方式创建对象，参数传入的就是子对象的原型，想让创建出来的对象继承自哪里，就传入那个对象就可以了。这个方法必须传入一个参数，否则运行时会抛异常，但可以传入 null，表示不继承任何对象，所以，JavaScript 里，是允许对象没有原型，允许对象不具有继承结构的。

对于原型，在后续会专门有一篇来讲讲，这里大概清楚些概念即可。

添加属性

JavaScript 里的对象，其实可以通俗的理解成属性的集合，既然是作为容器的存在，那么其实创建完对象只是第一步，后续就是往这个集合中添加属性，所以 JavaScript 里，对象是允许在运行期间动态添加属性的。

添加属性的方式，可以通过对象直接量方式，在创建对象之时，就写在对象直接量中，或者运行期间动态添加，如：

https://upload-images.jianshu.io/upload_images/1924341-e6e42a025ff67fcf.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240  
var o = {name:"dasu"}  
o.age = 24;  
o.sex = "man";  
o.love = "girl";

但需要注意一点的是，不像 Java 中在编写类代码，为类添加成员变量时，可以只声明却初始化。在 JavaScript 中，是不允许这样的。

也就是说，为对象添加某个属性时，需要直接将属性名和属性值都设置好，其实想想也能明白，对象无非就是属性的集合，你见过对哪个集合进行添加数据时，是可以只对该集合设置一个 key 值的吗？

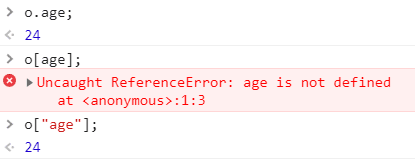
查询属性

访问对象的属性方式很简单，两种：. 运算符或 [] 运算符；

两种方式都可以访问对象的属性，但有一个区别：

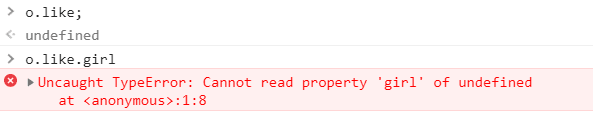
* . 运算符访问属性的话，后面跟着的是属性名
* [] 运算符访问属性的话，中括号内跟着的是属性名的**字符串**

仍旧以上面例子中的代码为例：



由于通过 [] 运算符访问对象的属性，需要的是一个属性名的字符串形式，所以这种方式会特别灵活，你可以再 [] 内放一个表达式也可以，只要表达式最后的结果是字符串，或者说可以自动类型转换为属性名的字符串即可，特别灵活。

而 . 运算符可能会比较习惯，但它就只能明明确确的通过属性名来访问了，如果你想通过某种拼接规则来生成属性名，就只能用 [] 不能使用 .。

如果访问对象中某个不存在的属性时，并不会抛异常，会输出 undefined，但如果继续访问不存在的属性的属性时，等价于访问 undefined 原始类型值的属性，这就会抛异常了：

ps：是不是发现，对对象的操作很像 Java 中对集合的操作？所以，有人解释说对象是属性的集合，这不是没根据的。

删除属性

delete 是用来删除对象上的属性的，但它只是将该属性名从对象上移除，并不删除属性值，举个例子：

var a = [1,2,3];  
var o = {name:"dasu", arr:a};  
​  
delete o.arr;  
console.log(a[0]);  //输出 => 1  
console.log(o.arr); //输出 => undefined

新键一个对象 o，它有个属性 aar 存储着数组 a，当通过 delete 删除对象 o 上的 aar 属性后，再去访问这个 aar 属性，获取的是 undefined，表明这个属性确实被删除了，但本质上只是将其与这个对象 o 的关联删除掉，并不会删除属性值，所以输出数组 a 的值时还是可以访问到的。

不过，delete 有一些局限，它并不是什么属性都可以删除：

* 只能删除自由属性，无法删除继承属性
* 不能删除那些可配置性为 false 的属性

属性拥有一些特性，在下面会讲，其中有一个是可配置性，当将这个特性设置为 false 时，就无法通过 delete 来删除。

而之前说过的，通过 var 声明的全局变量，虽然它最后是作为全局对象的属性存在，但它的可配置性被设为 false，所以这些全局变量才无法通过 delete 被删除。

尝试删除那些无法删除的属性，并不会让程序出问题，delete 表达式有一个返回值，true 表示删除成功，false 表示删除失败，仅此而已，没有其他什么副作用。

检测属性

因为 JavaScript 中对象的属性太过动态性了，在运行期间，都无法确定某个属性到底存不存在，某个到底是不是指定对象的属性，所以这种场景，一般都需要进行属性的检测。

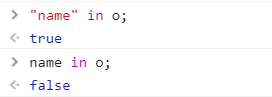
也就是检测对象内是否含有某个属性，有多种方式，下面分别来介绍：

* 查询属性的方式

之前说过，访问对象内不存在的属性时，会返回 undefined，可以利用这点来判断对象是否含有要访问的属性。

这种方式有个缺点，如果属性值刚好被人为的手动设置成 undefined 时，就无法区别对象究竟有没有这个属性。

* in 运算符方式

in 运算符左侧是属性名的字符串格式，右侧是对象，当右侧对象含有左侧字符串同名的属性时，返回 true，用这种方式就可以很好的判断对象是否含有某个属性。

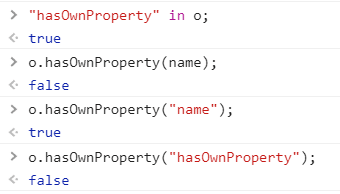
注意，左侧必须是属性名的字符串格式，跟 [] 运算符访问对象属性一样的限制要求。

但这种方式有个局限，就是无法区分这个属性究竟是自有属性还是继承属性，也就是说，继承自原型的属性通过该操作符同样会返回 true。

* hasOwnProperty()

上面说过，通过构造函数创建的对象，默认都会存在内置的继承结构，不管什么对象，这个默认的继承结构顶端都是构造函数 Object 的 prototype 属性值，由于它的属性值是一个内置的匿名对象，所以，通常都直接这么表达，对象都会继承自 Object.prototype，直接用 Object.prototype 的描述来代表这个属性所指向的具体对象。

所以，以后在看到诸如某某对象继承自 Function.prototype 或 Array.prototype 之类的描述，我们要能够清楚，它表示的是，对象的原型是 xxxx.prototype 这属性所指向的具体对象。

Object.prototype 属性值指向的对象中，定义了一个 hasOwnProperty() 方法，所以基本所有对象都可以使用，它是用来判断，对象是否含有指定的自有属性的。

首先利用上小节介绍的 in 方式来检测，o 对象的默认继承结构顶端是 Object.prototype，所以 o 对象继承了它的 hasOwnProperty 属性，第一行代码返回 true。

这个 hasOwnProperty 属性是个方法，调用它可以来检测对象是否含有指定的自有属性，参数也需要传入属性名的字符串格式，所以第二行代码返回 false，第三行返回 true。

hasOwnProperty 是继承自 Object.prototype 的属性，由于 hasOwnProperty() 方法只能检测自有属性，所以第四行返回 false。

* propertyIsEnumerable()

这个方法同样是 Object.prototype 中所定义的方法，所以，同样基本所有对象都能够使用。

它是 hasOwnProperty() 的增强版，也就是，用于检测对象的自有属性且该属性是可枚举性的，才会返回 true。

可枚举性是属性的另一个特性，用来标识该属性是否允许被遍历，下面会讲解。

因为有一些内置属性是不希望被枚举出来的，所以可通过该方法来判断。

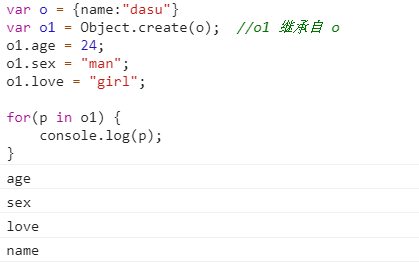
遍历属性

遍历属性也称枚举属性，也就是类似于对集合进行遍历操作，将其所含有的属性一个个读取出来。

遍历对象属性的方式也有多种，也一一来介绍：

* for-in 遍历

var o = {name:"dasu"}  
var o1 = Object.create(o);  //o1 继承自 o  
o1.age = 24;  
o1.sex = "man";  
o1.love = "girl";  
​  
for(p in o1) {  
 console.log(p);      
}

看看输出的结果：

o1 继承自 o，在 o1 内有三个自有属性，有一个继承属性，通过 for-in 方式遍历对象 o1 的属性时，不管是自有属性，还是继承属性，都会被输出。

同时，输出的是属性名，并不是属性值，所以 for-in 方式只是遍历对象的属性（包括继承属性），并返回属性名，注意是属性名。

通常 for-in 这种方式，可以结合 hasOwnProperty() 方法一起使用，来过滤掉继承的属性。

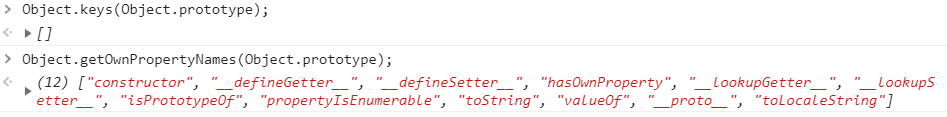
* Object.keys()

这又是一个类似静态方法的存在，注意这个方法跟上述 Object.create() 都是构造函数 Object 上的方法，而普通对象继承的是构造函数 Object.prototype 属性值所指向的那个原型对象，这是两个相互独立的对象，也就是说，通过构造函数创建出来的子对象并不是继承构造函数对象本身。

https://upload-images.jianshu.io/upload_images/1924341-8a1df1551b7080d0.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240所以在子对象中，无法使用 Object.keys() 这类构造函数对象本身的属性，这点需要注意一下，在后续专门讲继承时会再拿出来讲讲。

参数传入需要遍历属性的对象，通过该方法，可以获得一个数组对象，数组内就存储着参数传入的对象的自有属性且属性是可枚举性的，相当于 for-in 方式结合 hasOwnProperty() 的效果。

* Object.getOwnPropertyNames()

该方法也是遍历对象的自有属性，只是它是将参数传入的对象所拥有的所有属性都输出，包括那些被设置为不可枚举的属性，看个例子：

Object.prototype 指向了一个内置的对象，内置对象中定义了很多属性，继承这个原型的子对象们都可以使用这些属性，但这些属性都被设置为不可枚举性，所以通过 Object.keys() 遍历它时，得到的是一个空数组，子对象通过 for-in 方式遍历时也读取不到这些属性。

这种设计是好的，但考虑到如果有某些场景是需要读取对象自身的所有属性，包括那些不可枚举的，此时，就可通过 Object.getOwnPropertyNames() 来达到目的了。

属性的特性

上面介绍中，或多或少有提到属性的特性，属性特性是指，属性的一些特有行为。

属性的特性一共有三个：可写性、可配置性、可枚举性

* 可写性：表示这个属性是否允许被更改，当设置成 false 时，这就是一个只读属性
* 可配置性：表示这个属性是否允许被动态的添加或删除，当设置成 false 时，就不允许通过 delete 来删除
* 可枚举性：表示这个属性是否允许在遍历属性过程中被读取，当设置成 false 时，通过 for-in 或 Object.keys 都无法遍历到这个属性

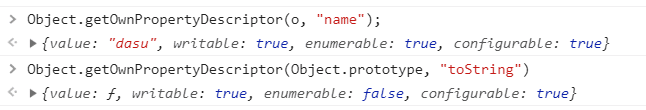
那么，如果知道对象的某个属性的这三种特性都是什么配置呢？

针对这种情况，内置了一个叫做属性描述符的对象，这个对象本身含有四个属性来描述属性：value、writable、enumerable、configurable。

* value：描述属性值，即 key-value 中的 value
* writable：描述属性的可写性
* enumerable：描述属性的可枚举性
* configurable：描述属性的可配置性

用来描述属性的数据结构有了，接下去就是如何操作了，先看一下，如果获取对象某个属性的描述信息：

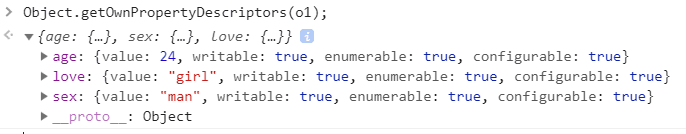
* Object.getOwnPropertyDescriptor()

还是通过Object 的一个方法，接收两个参数，第一个参数是对象，第二个参数是对象内的某个自有属性，将会返回一个属性描述符对象：

内置对象的很多属性都会针对属性的使用场景进行了不同的配置了，比如 Object.prototype 中所有属性的 enumerable 可枚举性都配置成 false。

但对于在代码中，通过对象直接量创建的对象，或者自定义构造函数创建的对象等，对这些非内置对象添加的属性，默认这三个特性都为 true，即对象添加的属性默认都是可写、可枚举、可配置的。

* Object.getOwnPropertyDescriptors()

这个方法也是用来获取对象属性的描述信息的，只是它只需一个参数即可，就是对象，然后会输出所有自有属性的描述信息：

这两个方法都是只能获取对象的自有属性的描述信息，如果想要获取继承属性的描述信息，需要先获取原型对象，再调用这两个方法处理原型。获取原型对象后续讲原型时会介绍，这里知道思路就可以了。

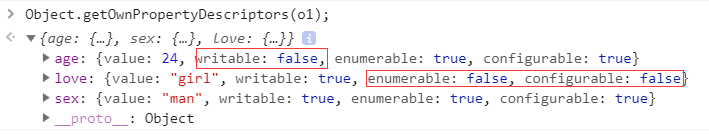
* Object.defineProperty()

有获取对象属性的描述信息的方法，自然有设置对象属性的描述信息方法，所以与上面两个方法相对应的就是 Object.defineProperty() 方法和 Object.definproperties()。

Object.defineProperty() 接收三个参数，第一参数是对象，第二个参数是需要修改属性描述信息的属性，第三个参数是含有属性描述符结构的对象：

var o = {name:"dasu"}  
var o1 = Object.create(o);  //o1 继承自 o  
o1.age = 24;  
o1.sex = "man";  
o1.love = "girl";  
​  
Object.defineProperty(o1, "age", {writable:false});  
Object.defineProperty(o1, "love", {enumerable:false, configurable:false});

第三个参数，你可以将四个属性值都指定，没指定的仍旧会使用默认的配置，再用 Object.getOwnPropertyDescriptors() 看下修改后的配置：



* Object.defineProperties()

这方法作用跟上面一样，只是它是批量处理的方法，接收两个参数，第一个是对象，第二个是需要修改的属性集合，如：

var o = {name:"dasu"}  
var o1 = Object.create(o);  //o1 继承自 o  
o1.age = 24;  
o1.sex = "man";  
o1.love = "girl";  
​  
Object.defineProperties(o1, {  
    age: {writable:false},  
    love: {enumerable:false, configurable:false}  
})

* 规则

有一些规则需要注意一下：

* 如果属性是不可配置的，那么不能修改它的可配置性和可枚举性，对于可写性，只能将 true 改为 false，不能将 false 改为 true
* 如果属性是不可配置且不可写的，那么不能修改这个属性的值
* 如果属性是可配置但不可写的，那么可以先将属性修改成可写，这时就可以修改属性值

属性的setter和getter

正常来说，对象的属性由属性的三种特性来控制属性的操纵限制，但有一种情况是例外的，那就是通过 setter 和 getter 添加的属性，这类属性通常叫做存取器属性，为了区分，将正常使用的那些属性叫做数据属性。

之所以叫做存取器属性，是因为，通过这种方式添加的属性，它的读写是交由 setter 和 getter 控制，并不是由属性描述符的三种特性控制。

先来看下，如何定义一个存取器属性：

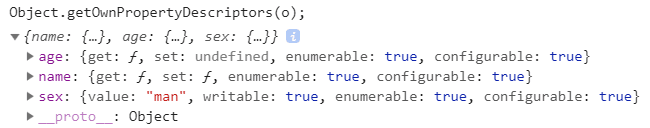
var o = {  
    set name(value) {},  
    get name() {return "dasu"},  
      
    get age() {return 24}  
}

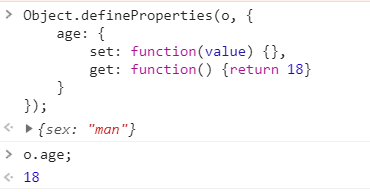
虽然看起来有点像 Java 中的 set 方法和 get 方法，但完全是两种不一样的东西，首先，这里的 set 和 get 虽然类似方法，但外部是不能通过方法来调用，第二，外部访问这些存取器属性，仍旧是使用 . 或 [] ，如 o.age 或 o["name"]。

相比于数据属性，存取器属性的区别就在于，读和写是通过 set 和 get 控制，在定义存取器属性时，如果没有定义 get，那么这个属性就是无法读取的，如果没有定义 set，那么这个属性就是不可写的。其余的，可枚举性和可配置性都跟数据属性一样。

也一样是通过 Object.defineProperty() 和 Object.getOwnPropertyDescriptor() 来设置或查看存取器属性的描述信息，唯一需要注意的是，对于数据属性，描述符对象有四个属性：value，writable，enumerable，configurable；但对于存取器属性来说，没有 value 和 writable 属性，与之替换的是 get 和 set 两个属性，所以看个例子：

var o = {  
    set name(value) {},  
    get name() {return "dasu"},  //存取器属性，可读，可写，但读写逻辑得自己实现  
      
    get age() {return 24},  //存取器属性，只读，读的逻辑得自己写  
      
    sex: "man"  //数据属性  
}

假设定义了这么个对象，有两个存取器属性，一个数据属性，通过 Object.getOwnPropertyDescriptors() 看一下这些属性的描述：

所以存取器属性和数据属性就在于读和写这两方面的不同，看下修改描述的方式：

存取器属性是可以换成数据属性，同样，数据属性也是可以换成存取器属性的，通过 Object.defineProperty() 在修改属性描述信息时，使用的如果是 set 和 get，那就将数据属性换成存取器属性了，使用的如果是 value 和 writable，原本如果是存取器属性，就将存取器属性转换成数据属性了。

另外，它也有一些规则需要注意一下：

* 如果存取器属性是不可配置的，则不能修改 set 和 get 方法，也不能将它转换为数据属性
* 如果数据属性是不可配置的，则不能将它转换为存取器属性

对象的特性

对象的属性有它的几种特性，而对象本身也有一些特性，主要是三个：原型属性、类属性、可扩展性

原型属性：表示对象继承自哪个对象，被继承的对象称为子对象们的原型

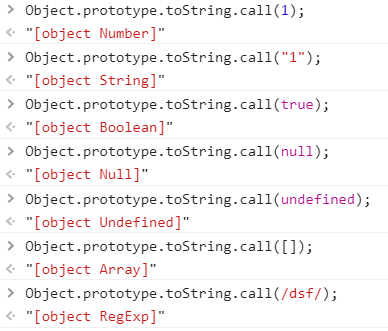
类属性：表示对象的类型信息，是一个字符串，比如数字的类属性为 Number

可扩展性：表示是否允许对象可动态的添加属性

原型留着后续讲原型时再来细讲，大概清楚对象是有继承结构，被他继承的对象称作它的原型，所以通常说 JavaScript 是基于原型的继承这些概念即可。

* 类属性

类属性，本质上就是通过调用 Object.toString() 方法来输出对象的一些信息，这些信息中，包括了对象所属类型的信息，对这串文本信息进行截取处理，就可以只获取对象所属类型的信息，所以称这些信息为对象的类属性。

类属性所呈现的信息很类似于 typeOf 运算符所获取的信息，只是类属性会比 typeOf 更有用一些，它能够区分所有的内置对象，以及区分 null，这些是 typeOf 所做不到的，如：

* 可扩展性

类似于属性有可配置性、可写性、可枚举性来控制属性的操纵限制，对象也具有可扩展性来限制对象的一些行为。

当将对象的可扩展性设置为 false 时，就无法再动态的为对象添加属性。默认创建的新对象，都是具有可扩展性的。

不像属性的特性那样，还专门定义了一个属性描述符对象来控制属性的特性，对于对象的可扩展性，操作很简单：

* Object.isExtensible()

使用 Object.isExtensible() 来获取对象的可扩展性描述，返回 true，表示对象是可扩展的，即可动态添加属性。

* Object.preventExtensions()

同样，可使用 Object.preventExtensions() 来设置对象的不可扩展，参数传入对象即可。这样，这个对象就不可动态添加属性了。

但有几点需要注意：

* 一旦将对象设置为不可扩展，就无法再将其转换回可扩展了
* 可扩展性只限制于对象本身，对对象的原型并不影响，在原型上添加的属性仍可动态同步到子对象上

针对于对象的可扩展性，对象属性的可写性、可配置性、可枚举性这些操作，Object 内封装了一些便捷的方法，如：

* Object.seal()：将对象设置为不可扩展，同时，将对象所有自有属性都设置为不可配置，通常称封闭对象。可用 Object.isSealed() 来检测对象是否被封闭。
* Object.freeze()：将对象设置为不可扩展，同时，将对象所有自有属性不可配置且只读，通常称冻结对象。可用 Object.isFrozen() 来检测对象是否被冻结。

## 数组

数据的有序集合称为数组。

其实也就是个容器，但与 Java 中数组不同的是，JavaScript 中的数组不限制元素类型、本身就是个对象，因此不管在使用方面、语法方面、概念上都会一些区别。

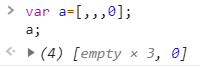
那么本章其实也就是学习 JavaScript 中数组的用法：

相关术语

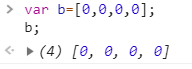
* 稀疏数组

稀疏数组就是指不连续索引的数组，数组容器中某些索引是空的，无值。相反，正常的连续索引的数组就是非稀疏数组，容器中各元素紧密堆放。

如：

稀疏数组例子：

通过 delete 删除某个元素后也会让数组变成稀疏数组，直接通过 a[index] 以不连续形式为数组添加元素也会变成稀疏数组。

非稀疏数组例子：

数组内每个元素紧密排列。

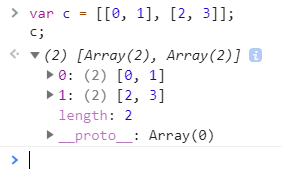
* **多维数组**

JavaScript 不支持真正的多维数组，但可以用数组的数组来近似。

以二维数组举例，在 Java 中可直接声明：

**int**[][] a = **new int**[][]{};

但在 JavaScript 中无法定义二维数据，会报语法错误：

但由于数组在 JavaScript 中也是对象，数组中的元素也可以是数组，因此可以用数组的数组来实现多维数组：

* 类数组对象

理解类数组对象概念可以将这个词的补充解释完整，即：类似数组的对象。

所以，这个概念的主语是对象，而对象如果是通过 [] 来操作它的属性时，属性值可以很灵活，不必要必须满足标识符规定。因此，当如果定义了某个对象，其属性值是非负整数：0,1,2,3…，此外再给这个对象定义了一个 length 属性，那么此时就可称这个对象为类数组对象。

因为对这种对象的操作，跟数组很类似，而且 Array.prototype 中提供的很多操作数组的方法都可以直接用来操作这些类数组对象。

数组属性

* length

每个数组都有一个 length 属性，这个属性是使数组区别于常规的 JavaScript 对象的关键。

需要注意，length 并不是表示数组的元素个数。

length 表示的是数组里最大索引 + 1，因为 JavaScript 分稀疏数组和非稀疏数组。

如果是非稀疏数组，各元素都紧密堆放，那么此时 length = 最大索引 + 1，能够表示数组元素的个数。

但如果是稀疏数组，由于中间有些索引位置其实是空的，并没有元素，索引 length = 最大索引 + 1，此时并不表示数组元素个数。

而且，数组索引是基于 32 位数值的，所以 length 的取值范围：

0 ~ 2^32 - 1 🡺 0 ~ 4294967295

另外，由于 length 属性默认是可读可写的，所以它有一些特殊功能：

* 当添加或删除数组元素时，length 会自动更新。
* 并不是所有删除数组元素的操作都会让 length 更新，有些删除操作只是移除索引里保存数据，并不移除数组这个索引所占的坑位。
* length 可写性，当设置 length 比当前数组长度小的值时，会自动删除那些索引值大于等于 length 的元素。
* 反过来将 length 设置比当前数组长度大，会让数组变成稀疏数组，并不会实际添加一些元素进去。

数组特性

虽然数组也是对象，但它有一些特性是其他对象所没有的：

* 当有新元素添加到数组中时，自动更新 length 属性
* 设置 length 为一个较小值将截断数组
* 继承了 Array.prototype 一些操作数组的便捷方法
* 类属性为 “Array”
* 不限制元素类型，一个数组中可以同时存储各种类型的数据

创建数组

数组的创建，或者说定义数组，初始化数组一共有两种方式：

* 数组直接量

**var *a*** = []; *//空数组***var *b*** = [1, **"2"**, **true**, [1+2, {}]]; *//不同类型的数组元素，数组直接量中甚至可以是表达式***var *c*** = [,,,3]; *//省略的索引，读取时为 undefined*

* 构造函数 Array()

**var *a*** = **new *Array***(); *//通过构造函数创建数组*  
***a***[0] = 1;

数组元素的读写

跟 Java 语言的数组读写一样，同是通过 [] 中括号来操作。

但 JavaScript 更灵活，[] 里可以是任何表达式，不限制于非负整数，如：

***a***[2] = 0; *//常规操作****a***[**"23"**] = 0; *//自动将 "23" 字符串转成数值类型 23，等效于 a[23]=0****a***[-23] = 0; *//当[]中不是非负整数时，此操作变成对象的属性读写，因为数组也是对象****a***[5+6]; *//[] 中可以是表达式，先计算表达式值后，再操纵数组，等效于 a[11]*

因为数组也是对象，所以 JavaScript 中的数组操作不存在越界的场景，当试图查询不存在的属性时，只会返回 undefinded。

数组元素的添加

添加元素都数组最简单的方式是通过 [] 操作符，另外也可借助 Array.prototype 的一些方法：

**var *a*** = []; *//a 是空数组****a***[0] = 0; *//指定索引位置添加元素****a***.push(1); *//等效于 a[length] = 1，在数组末尾添加元素****a***.unshift(-1); *//在数组头部添加元素，原本数组中的元素依次向后移****a***.splice(0, 0, **"0"**, **"1"**); *//插入删除操作通用的方法，这里等效于 a.unshift("0", "1");*

[] 方式来添加元素的前提是，中括号里的索引位置原先并没有元素存在，如果索引位置有元素存在，则该操作变成赋值操作。

如果想在末尾添加元素，直接使用 push 即可；

如果想在开头添加元素，并让其他元素自动后移，可用 unshift；

splice 是个通用的方法，可在数组指定的任何位置添加元素，并让这位置之后的元素自动后移，同时它也可用来删除指定位置元素，并让后续元素自动前移补上被删除的位置。具体参数含义后面介绍。

数组元素的删除

数组元素的删除分两种场景：

1. 单纯将指定位置的元素删除，不会触发高索引元素往下移的填充行为，也不会触发 length 的长度减少；

2. 删除指定位置的元素，同时后面元素会往前移来填充被删除元素留下的空白位置，同时 length 会跟随着减少。

所以，当有涉及数组元素删除操作时，需特别注意下，根据自己的需求场景，选择对应的方法进行操作。

场景1对应的操作：

**var *a*** = [1,2,3]; *//数组 length = 3;***delete *a***[1]; *//此时数组：[1,,3]，length 仍旧=3*

场景2对应的操作：

**var *a*** = [1,2,3,4,5,6,7,8]; *//数组 length = 8;****a***.pop(); *//数组：[1,2,3,4,5,6,7] length = 7;****a***.shift(); *//数组：[2,3,4,5,6,7] length = 6****a***.splice(2, 2); *//数组：[2,3,6,7] length = 4****a***.length = 2; *//数组：[2,3] length = 2*

除了使用方法来删除元素，对 length 的赋值操作也可以达到删除末尾的多个元素，超过 length 的索引位置的元素就都被清空掉。

数组遍历

* for 循环语句

数组的遍历也是很常见的场景，常规的用法类似 Java 的 for 循环语句：

**var *a*** = [1,2,,,,6,7,8]; *//数组 length = 8;***for** (**var *i*** = 0; ***i*** < ***a***.length; ***i***++) {  
 ***console***.log(***a***[***i***]);  
}

当数组是稀疏数组时，那些索引位置没有元素存在的也仍旧需要遍历，读取的值是 undefined，所以需要根据需要做相应判断处理：

**for** (**var *i*** = 0; ***i*** < ***a***.length; ***i***++) {  
 **if** (!***a***[***i***]) **continue**; *//跳过 null，undefined 和不存在的元素  
 //...*}  
**for** (**var *i*** = 0; ***i*** < ***a***.length; ***i***++) {  
 **if** (***a***[***i***] === **undefined**) **continue**; *//跳过undefined 和不存在的元素  
 //...*}  
**for** (**var *i*** = 0; ***i*** < ***a***.length; ***i***++) {  
 **if** (!(***i* in *a***)) **continue**; *//跳过不存在的元素  
 //...*}

* for-in 循环语句

除了使用常规的 for 循环外，还可以使用 for-in 的方式：

**var *a*** = [1,2,,,,6,7,8]; *//数组 length = 8;*

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

因为数组实际上也是对象，数组的索引从对象角度来看，其实也就是属性，那么就可以用 for-in这种方式遍历属性，这种方式可以跳过稀疏数组中那些不存在的元素，但有个缺点，它也会遍历那些继承属性，所以如果需要，可做一些过滤判断：

**for**(**var *i* in *a***) {  
 **if** (!***a***.hasOwnProperty(***i***)) **continue**; *//跳过继承的属性   
 //...*}

注意：虽然 for-in 也可以达到遍历的效果，但不建议使用在遍历数组的场景，因为遍历顺序并不一定按照索引顺序。

* forEach 方法

那么，有没有一种方便一点的遍历方法，有的：forEach

***a***.forEach(**function** (x) { *//x即数组a中存在的元素*  
 ***console***.log(x);   
});

这种方式可以遍历数组中存在的元素，不需做额外的判断处理。如果函数中需要数组元素的索引信息、数组本身的对象引用信息，此时，可增加额外参数实现:

*//x:数组元素, i:元素的索引, a:数组的引用*

***a***.forEach(**function** (x, i, a) {   
 ***console***.log(a[i] + **" = "** + x);   
});

数组方法

Array.prototype 中定义了一些很有用的操作数组的函数，可用于作为数组的方法调用，足够满足开发中所需的数组相关的操作需求，列举一些常见的，更多可参考 API：

* join()

将数组各元素按照指定字符串拼接起来后输出字符串：

**var *a*** = [1,,2,3];  
***a***.join(); *//输出：1,,2,3 没有参数默认以逗号,拼接****a***.join(**" "**) *//输出：1 2 3 以空格拼接*

不存在的元素也会占据一个拼接符，所以可以结合其他方法过滤使用，后续介绍。

* reverse()

颠倒数组，将原数组进行逆序操作：

**var *a*** = [1,,2,3];  
***a***.reverse();  
***a***.join(); *//输出：3,2,,1 原数组被逆序*

* sort()

将原数组按照指定规则对元素进行排序，默认以字母表顺序排序：

**var *a*** = [22,,3,0,1];  
***a***.sort();  
***a***.join(); *//输出：0,1,22,3,,*

注意：默认排序行为是将所有元素按照字符串形式处理，一个字符一个字符的排序，所有 22 的首字符 2 在 3 前面，排序结果才会是 22 在 3 前面，因为它并不是按照数值的大小来排序。

另外，不存的元素都排在末尾。

所以可以自行指定排序规则，如从小大到排序：

**var *a*** = [22,,3,0,1];  
***a***.sort(**function** (a, b) {  
 **return** a - b; *//根据顺序返回：负数，0，正数*  
});  
***a***.join(); *//输出：0,1,3,22,*

* concat()

将参数传入的数值拼接到数组末尾，但不是在原数组上操作，而是会新建一个数组，此方法的拼接操作不会影响到原数组内容。

**var *a*** = [1,2,3];  
***a***.concat(4,5); *//返回 [1,2,3,4,5]****a***.concat([4,5]); *//返回 [1,2,3,4,5] 因为上述操作没有影响到原数组****a***.concat([4,5], [6,[7,8]]); *//返回 [1,2,3,4,5,6,[7,8]]*

注意：如果传入的参数是数组，那么会解析一层数组，拼接数组的元素内容，那如果数组是多维数组，也只拼接第一维的数组元素，不会进一步解析数组。

* slice()

截取原数组的某个片段，返回一个子数组，不会在原数组上操作，返回的是新数组：

**var *a*** = [1,2,3,4,5];  
***a***.slice(0, 3); *//返回 [1,2,3] 两个参数指定起始和终点位置，关系是[),即左包含右不包含****a***.slice(3); *//返回 [4,5] 只有一个参数时，表示指定起点到末尾****a***.slice(1, -1);*//返回 [2,3,4] 负数表示倒数第n个元素，*

* splice()

通用的在数组的指定位置插入或删除元素，插入会让后面的元素自动往后移空出位置，删除会让后面的元素自动往前移填补空白，length 会跟随着变化。

如果有包含删除操作，那么删除的数组会被返回，否则返回空数组；

**var *a*** = [1,2,3,4,5,6,7,8]; *//第一个参数选择操作的起始位置，第二个参数指定要删除的个数****a***.splice(4); *//返回 [5,6,7,8] 原数组 a：[1,2,3,4]****a***.splice(1,2); *//返回 [2,3] 原数组 a：[1,4]****a***.splice(1,1); *//返回 [4] 原数组 a：[1]*

第一个参数选择要操作的起始位置，第二个参数指定要删除的元素个数，如果只有一个参数，那么就删除从起始位置到末尾的元素。被删除的元素会组成新数组返回，删除操作是直接在原数组上进行的。

**var *a*** = [1,2,3,4,5]; *//第三个参数开始之后的任意参数都会被插入到指定的位置****a***.splice(2,0,**'a'**,**'b'**); *//返回 [] 原数组 a：[1,2,'a','b',3,4,5]****a***.splice(2,2,[1,2],3); *//返回 ['a','b'] 原数组 a：[1,2,[1,2],3,3,4,5]*

第三个参数开始之后的任意数量的参数都会被插入的第一个参数指定的位置，先进行删除操作，再进行添加操作。

* push() 和 pop()

在数组末尾添加或移除元素，pop() 时，被移除的元素会返回。

* unshift() 和 shift()

在数组开头添加或移除元素，都会触发数组元素进行迁移行为；

* toString() 和 toLocaleString()

数组的 toString() 行为跟 join() 输出的一致，toLocaleString() 说是使用本地化方式来生成字符串，没理解它的应用场景。

* forEach()

遍历数组内每个元素，每遍历一个元素，会调用一次指定的函数，并将元素的相关信息通过参数传入函数内。

* map()

原数组按照指定规则映射到新数组的操作，跟 forEach() 很类似，遍历数组内的每个元素时，都会调用一次指定的函数，并将元素相关信息通过参数传入函数内。但这个函数需要有一个返回值，用于生产新的数组的元素。

**var *a*** = [1,2,3];  
**var *b*** = ***a***.map(**function** (value, index, array) {  
 **return** value + index;  
}); *//b：[1,3,5]*

新数组与原数组的映射关系为：新数组元素 = 原数组元素 + 元素索引；

当有需要对原数组根据某种规则换算出新数组时，可用此方法。

* filter()

原数组元素根据某种规则进行过滤操作，过滤完后的元素作为新数组返回。跟 forEach() 也类似，都一样是在遍历每个元素时调用指定的方法，并将元素进行传入。这个方法需要一个 boolean 返回值，用来表示是否可以加入新数组。

**var *a*** = [1,2,3,4,5];  
**var *b*** = ***a***.filter(**function** (x) {  
 **return** x > 3;  
}); *//b：[4,5]*

用此方法也将稀疏数组转成非稀疏数组，函数内默认返回 true 即可，因为这些方法的遍历是只遍历数组内存在的元素。

* every() 和 some()

用于检测数组的元素是否满足指定的条件，这两个方法都返回 boolean 值。检测行为就命名表示的意思，every() 表示数组里每个元素都需要满足条件，最终才会返回 true，一旦某个元素不满足，后续元素不会再进行检测，方法直接返回 false。some() 则刚好相反，只要有一个元素满足条件，后续元素不用检测，方法直接返回 true。

**var *a*** = [1,2,3,4,5];  
***a***.some(**function** (value, index, array) {  
 **return** value > 3; *//返回 true，因为存在大于3的元素*});  
***a***.every(**function** (value, index, array) {  
 **return** value > 3; *//返回false，因为不是所有元素都大于3*});

* reduce() 和 reduceRight()

依次对数组里每个元素按照指定规则进行计算，计算之后的结果继续跟下一个元素按照规则计算，常用于求和，最大值之类的场景。

**var *a*** = [1,2,3,4,5];  
***a***.reduce(**function** (x,y) { *//数组求和* **return** x + y;  
}, 0);

区别上述几个方法，这个的参数需要有两个参数，第一参数是函数，用于指定按照某种规则计算，这个函数也需要有两个参数，以及返回值，它的返回值会和下一个元素再一次传入该函数中计算。reduce 的第二个参数会和数组第一个元素被传入函数内计算，这里是求和，所以初始值传 0，求积可以传1，以此类推。

如果不传第二个参数，那么默认以数组第一个元素的值作为第二个参数的值。

reduceRight 和 reduce 用途，用法一致，唯一的区别，它是从数组的末尾往前一个个处理元素的。一个左到右处理数组，一个右到左。

* indexOf() 和 lastIndexOf()

在数组内搜索指定元素，返回找到的第一个元素的索引位置，没有找到返回 -1

两个方法，一个从左往右寻找，一个从右往左寻找。

* Array.isArray()

用于判断某个对象是否是数组类型。

## 函数

相关术语

* function

定义函数的关键字

* 函数名

function 定义函数时，可以紧跟着函数名，它其实是一个变量，需要符合标识符规则，这个函数名变量指向这个函数对象，通过它即可调用函数。

* 函数体

定义函数时 {} 大括号中的语句代码的集合。

* 形参和实参

形参是指定义函数时指定的参数列表，表示期望调用时传入哪些参数；

实参是指具体调用时，实际传入的参数值。

在函数调用的时候，会将形参跟实参以key-value形式关联起来，函数体的语句代码即可通过形参访问传入的这些实参的具体值。

* 普通函数和构造函数

当普通函数跟 new 关键字一起使用时，此时这个函数就被称为构造函数。

函数定义

函数的定义大体上包含以下几部分：function 关键字、函数对象的变量标识符、形参列表、函数体、返回语句。

如果函数没有 return 语句，则函数返回的是 undefined。

函数定义有三种方式：

* 函数声明式

*add*(1,2); *//由于函数声明被提前了，不会出错*

**function** *add*(x, y) {  
 *//函数体*}

add 是函数名，由于 JavaScript 有声明提前的处理，以这种方式定义的函数，可以在它之前调用。

* 函数定义表达式

**var** *add* = **function** (x, y) {  
 *//函数体*}

这种方式其实是定义了匿名函数，然后将函数对象赋值给 add 变量，JavaScript 的声明提前处理只将 add 变量的声明提前，赋值操作仍在原位置，因此这种方式的声明，函数的调用需要在声明之后才不会报错。

* Function

**var *add*** = **new *Function***(**"x"**, **"y"**, **"return x\*y;"**);

*//基本等价于***var** *add* = **function** (x, y) {  
 **return** x\*y;  
}

Function 构造函数接收不定数量的参数，最后一个参数表示函数体，前面的都作为函数参数处理。

注意：以这种方式声明的函数作用域是全局作用域，即使这句代码是放在某个函数内部，相当于全局作用域下执行 eval()，而且对性能有所影响，不建议使用这种方式。

函数调用

跟 Java 不一样的地方，在 JavaScript 中函数也是对象，既然是对象，那么对于函数对象这个变量是可以随意使用的，比如作为赋值语句的右值，作为参数等。

当被作为函数对象看待时，函数体的语句代码并不会被执行，只有明确是函数调用时，才会触发函数体内的语句代码的执行。

例如：

**var** *a* = **function** () {  
 **return** 2;  
}

**var *b*** = *a*; *//将函数对象a的引用赋值给b***var *c*** = *a*(); *//调用a函数，并将返回值赋值给c*

函数的调用可分为四种场景：

* 作为函数被调用
* 作为对象的方法被调用
* 作为构造函数被调用
* 通过 call() 或 apply() 间接的调用

不同场景的调用所造成的区别就是，函数调用时的上下文（this）区别、作用域链的区别；

* 作为函数被调用

通常来说，直接使用函数名+() 的形式调用，就可以认为这是作为函数被调用。如果有借助 bind() 时会是个例外的场景，但一般都可以这么理解。

如果只是单纯作为函数被调用，那么通常是不用去考虑它的上下文、它的this值，因为这个时候，函数的用途倾向于处理一些通用的工作，而不是特定对象的特定行为，所以需要使用 this 的场景不多。

普通函数被调用时的作用域链的影响因素取决于这个函数被定义的位置，作用域链是给变量的作用域使用的，变量的作用域分两种：全局变量、函数内变量，作用域链决定着函数内的变量取值来源于哪里；

普通函数被调用时的上下文在非严格模式下，一直都是全局对象，不管这个函数是在嵌套函数内被调用或定义还是在全局内被定义或调用。但在严格模式下，上下文是 undefined。

* 作为对象的方法被调用

普通的函数如果挂载在某个对象内，作为对象的属性存在时，此时可从对象角度称这个函数为对象的方法，而通过对象的引用访问这个函数类型的属性并调用它时，此时称为方法调用。

方法调用的上下文（this）会指向挂载的这个对象，作用域链仍旧是按照函数定义的位置生成。

**var *a*** = {  
 **b**: 1,  
 c: **function** () {  
 **return this**.**b**;  
 }  
}  
***a***.c(); *//输出1，a.c() 称为对象的方法调用*

***a***[**"c"**](); *//对象的属性也可通过[]访问，此种写法也是调用对象a的c方法*

只有明确通过对象的引用访问函数类型的属性并调用它的行为才称为对象的方法调用，并不是函数挂载在对象上，它的调用就是方法调用，需要注意下这点，看个例子：

**var *d*** = ***a***.c;  
***d***(); *//将对象的c函数引用赋值给d，调用d，此时d()是普调的函数调用，上下文在非严格模式下是全局对象，不是对象a*

下面通过一个例子来说明普通函数调用和对象的方法调用：

**var *a*** = 0;  
**var *o*** = {  
 **a**:1,  
 m: **function** () {  
 ***console***.log(**this**.**a**);   
 *f*(); *//f() 是函数调用*  
 **function** *f*() {  
 ***console***.log(**this**.**a**);  
 }  
 }  
}

***o***.m(); *//输出 1 0，因为0.m()是方法调用，m中的this指向对象o，所以输出*

输出1 0，因为o.m() 是方法调用，m 中的 this 指向对象 o，所以输出的 a 是对象 o 中 a 属性的值 1；

而 m 中虽然内嵌了一个函数 f，它并不挂载在哪个对象像，f() 是对函数 f 的调用，那么它的上下文 this 指向的是全局对象。

所以，对于函数的不同场景的调用，重要的区别就是上下文。

* 作为构造函数被调用

普通函数挂载在对象中，通过对象来调用称方法；而当普通函数结合 new 关键字一起使用时，被称为构造函数。

构造函数的场景跟其他场景比较不同，区别也比较大一些，除了调用上下文的区别外，在实参处理、返回值方面都有不同。

如果构造函数没有参数，是可以省略圆括号的，如：

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

对于方法调用或函数调用圆括号是不能省略的。

构造函数调用时，是会创建一个新的空对象，继承自构造函数的 prototype 属性，并且这个新创建的空对象会作为构造函数的上下文，如：

**var *o*** = {  
 **a**:1,  
 f:**function** () {  
 ***console***.log(**this**.**a**);  
 }  
}

***o***.f(); *//输出1***new *o***.f(); *//输出undefined*

如果是 o.f() 时，此时是方法调用，输出 1;

而如果是 new o.f() 时，此时 f 被当做构造函数处理，this 指向的是新创建的空对象，空对象没有 a 这个属性，所以输出 undefined。

构造函数通常不使用 return 语句，默认会创建继承自构造函数 prototype 的新对象返回。但如果硬要使用 return 语句时，如果 return 的是个对象类型，那么会覆盖掉构造函数创建的新对象返回，如果 return 的是原始值时，return 语句无效。

**var *o*** = {  
 f:**function** () {  
 **return** [];  
 }  
}  
  
**var *b*** = **new *o***.f(); *//b是[] 空数组对象，而不是f*

* 间接调用

call() 和 apply() 是 Function.prototype 提供的函数，所有的函数对象都继承自 Function.prototype，所有都可以使用这两个函数。它们的作用是可以间接的调用此函数。

什么意思，也就是说，任何函数可以作为任何对象的方法来调用，即使这个函数并不是那个对象的方法。

**var *o*** = {  
 **a**:1,  
 f:**function** () {  
 ***console***.log(**this**.**a**);  
 }  
}  
***o***.f(); *//输出1***var *o1*** = {  
 **a**:2  
}  
***o***.f.call(***o1***); *//输出2*

函数 f 原本是对象 o 的方法，但可以通过 call 来间接让函数 f 作为其他对象如 o1 的方法调用。

所以间接调用本质上也还是对象的方法调用。应用场景可以是子类用来调用父类的方法。

那么函数的调用其实按场景来分可以分为三类：作为普通函数被调用，作为对象方法被调用，作为构造函数被调用。

普通函数和对象方法这两种区别在于上下文不一样，而构造函数与前两者区别更多，在参数处理、上下文、返回值上都有所区别。

如果硬要类比于 Java 的函数方面，我觉得可以这么类比：

* 普通函数的调用 VS 公开权限的静态方法
* 对象方法的调用 VS 对象的公开权限的方法
* 构造函数的调用 VS 构造函数的调用

左边 JavaScript，右边 Java，具体实现细节很多不一样，但大体上可以这么类比理解。

函数参数

参数分形参和实参两个概念，形参是定义时指定的参数列表，期望调用时函数所需传入的参数，实参是实际调用时传入的参数列表。

在 JavaScript 中，不存在 Java 里方法重载的场景，因为 JavaScript 不限制参数的个数，如果实参比形参多，多的省略，如果实参比形参少，少的参数值就是 undefined。

这种特性让函数的用法变得很灵活，调用过程中，根据需要传入所需的参数个数。但同样的，也带来一些问题，比如调用时没有按照形参规定的参数列表来传入，那么函数体内部就要自己做相对应的处理，防止程序因参数问题而异常。

同样需要处理的还有参数的类型，因为 JavaScript 是弱类型语言，函数定义时无需指定参数类型，但在函数体内部处理时，如果所期望的参数类型与传入的不一致，比如希望数组，传入的是字符串，这种类型不一致的场景JavaScript虽然会自动根据类型转换规则进行转换，但有时转换结果也不是我们所期望的。

所以，有些时候，函数体内部除了要处理形参个数和实参个数不匹配的场景外，最好也需要处理参数的类型检查，来避免因类型错误而导致的程序异常。

* arguments

函数也是个对象，当定义了一个函数后，它继承自 Function.prototype 原型，在这个原型中定义了所有函数共有的基础方法和属性，其中一个属性就是 arguments。

这个属性是一个类数组对象，按数组序号顺序存储着实参列表，所以在函数内使用参数时，除了可以使用形参定义的变量，也可以使用 arguments。

**var** *a* = **function** (x, y) {  
 *//x 和 arguments[0]等效* ***console***.log(x);  
 ***console***.log(***arguments***[0]);

***console***.log(***arguments***[1]);

***console***.log(***arguments***[2]);  
}  
  
*a*(5); *//输出 5 5 undefined undefined*

*a*(5, 4, 3); *//输出 5 5 4 3*

所以，虽然函数定义时声明了三个参数，但使用的时候，并不一定需要传入三个，当传入的实参个数少于定义的形参个数时，相应形参变量对应的值为 undefined；

相反，当传入实参个数超过形参个数时，可用 arguments 来取得这些参数使用。

* 参数处理

因为函数不对参数个数、类型做限制，使用时可以传入任意数量的任意类型的实参，所以在函数内部通常需要做一些处理，大体上从三个方面进行考虑：

* 形参个数与实参个数不符时处理
* 参数默认值处理
* 参数类型处理

**形参个数与实参个数不符时处理**

通过 argument.length 可以获取实参的个数，通过函数属性 length 可以获取到形参个数，知道形参个数和实参个数就可以做一些处理。如：

**var** *a* = **function** (x) {  
 **if** (***arguments***.length !== ***arguments***.callee.length) {  
 **throw *Error***(**"..."**);  
 }  
}

上述代码表示当传入的实参个数不等于形参个数时，抛异常。

形参个数用：arguments.callee.length 获取，callee 是一个指向函数本身对象的引用。这里不能直接用 length 或 this.length，因为在函数调用一节说过，当以不同场景使用函数时，上下文 this 的值是不同的，不一定指向函数对象本身。

在函数体内部要获取一个指向函数本身对象的引用有三种方式：

* 函数名
* arguments.callee
* 作用域下的一个指向该函数的变量名

**参数默认值处理**

通常是因为实参个数少于形参的个数，导致某些参数并没有被定义，函数内使用这些参数时，参数值将会是 undefined，为了避免会造成一些逻辑异常，可以做一些默认值处理。

**var** *a* = **function** (x) {  
 *//根据形参实参个数做处理* **if** (***arguments***.length !== ***arguments***.callee.length) {  
 **throw *Error***(**"..."**);  
 }  
 *//处理参数默认值* x = x || **"default"**; *// 等效于 if(x === undefined) x = "default";*}

**参数类型处理**

**var** *a* = **function** (x) {  
 *//根据形参实参个数做处理* **if** (***arguments***.length !== ***arguments***.callee.length) {  
 **throw *Error***(**"..."**);  
 }  
 *//处理参数默认值* x = x || **"default"**; *// 等效于 if(x === undefined) x = "default";  
 //参数类型处理* **if** (***Array***.isArray(x)) {  
 *//...* }  
 **if** (x **instanceof *Function***) {  
 *//...* }   
 *//...*}

参数类型的处理可能比较常见，通过各种辅助手段，确认所需的参数类型究竟是不是期望的类型。

* 多个参数时将其封装在对象内

当函数的形参个数比较多的时候，对于这个函数的调用是比较令人头疼的，因为必须要记住这么多参数，每个位置应该传哪个。这个时候，就可以通过将这些参数都封装到对象上，函数调用传参时，就不必关心各个参数的顺序，都添加到对象的属性中即可。

*//函数用于复制原始数组指定起点位置开始的n个元素到目标数组指定的开始位置***function** *arrayCopy*(fromArray, fromStart, toArray, toStart, length) {  
 *//...*}  
  
*//外部调用时，传入对象内只要有这5个属性即可，不必考虑参数顺序，同时这种方式也可以实现给参数设置默认值***function** *arrayCopyWrapper*(args) {  
 *arrayCopy*(args.fromArray,  
 args.fromStart || 0,   
 args.toArray,  
 args.toStart || 0,  
 args.**length**);  
}

*arrayCopyWrapper*({**fromArray**:[1,2,3], **fromStart**:0, **toArray**:***a***, **length**:3});

第二种方式相比第一种方式会更方便使用。

函数特性

函数既是函数，也是对象。它拥有类似其他语言中函数的角色功能，同时，它本身也属于一个对象，同样拥有对象的相关功能。

当作为函数来对待时，它的主要特性也就是函数的定义和调用：如何定义、如何调用、不同定义方式有和区别、不同调用方式适用哪些场景等等。

而当作为对象来看待时，对象上的特性此时也就适用于这个函数对象，如：动态为其添加或删除属性、方法，作为值被传递使用等。

所以，函数的参数类型也可以是函数，函数对象也可以拥有类型为函数的属性，此时称它为这个对象的方法。

如果某些场景下，函数的每次调用时，函数体内部都需要一个唯一变量，此时通过给函数添加属性的方式，可以避免在全局作用域内定义全局变量，这是 Java 这类语言做不到的地方。

类似需要跟踪函数每次的调用这种场景，就都可以通过对函数添加一些属性来实现。

**function** *uniqueCounter*() {  
 **return** *uniqueCounter*.**counter**++;  
}  
*uniqueCounter*.**counter** = 0;  
  
**var *a*** = *uniqueCounter*(); *//a = 0;***var *b*** = *uniqueCounter*(); *//b = 1;***var *c*** = *uniqueCounter*(); *//c = 2;*

虽然定义全局变量的方式也可以实现，但容易污染全局空间的变量。

函数属性

除了可动态对函数添加属性外，由于函数都是继承自 Function.prototype 原型，因此每个函数其实已经自带了一些属性，包括常用的方法和变量，比如上述介绍过的 arguments。

这里就来学下，一个函数本身自带了哪些属性，不过函数比较特别，下面介绍的一些属性并没有被纳入标准规范中，但各大浏览器却都有实现，不过使用这类属性还是要注意下：

* arguments

上述介绍过，这个属性是个类数组对象，用于存储函数调用时传入的实参列表。

但有一点需要注意，在严格模式下，不允许使用这个属性了，这个变量被作为一个保留字了。

* length

上述也提过，这个属性表示函数声明时的形参个数，也可以说是函数期望的参数个数。

有一点也需要注意，在函数体内不能直接通过 length 或 this.length 来访问这个属性，因为函数会跟随着不同的调用方式有不同的上下文 this，并不一定都指向函数对象本身。

而 arguments 对象中还有一个属性 callee，它指向当前正在执行的函数，在函数体内部可以通过 arguments.callee 来获取函数对象本身，然后访问它的 length 属性。

在函数外部，就可以直接通过访问函数对象的属性方式直接获取 length。如：

**var** *a* = **function** (x, y) {  
 ***console***.log(***arguments***.length);  
 ***console***.log(***arguments***.callee.length);  
}  
  
*a*(1); *// 输出 1 2，实参个数1个，形参个数2个  
a*.length; *//2*

但需要注意一点，在严格模式下，函数体内部就不能通过 arguments.callee.length 来使用了。

* caller

caller 属性表示指向当前正在执行的函数的函数，也就是当前在执行的函数是在哪个函数内执行的。这个是非标准的，但大多浏览器都有实现。

在严格模式下，不能使用。

还有一点需要注意的是，有的书里是说这个 caller 属性是函数的参数对象 arguments 里的一个属性，但某些浏览器中，caller 是直接作为函数对象的属性。

总之，arguments，caller，callee 这三个属性如果要使用的话，需要注意一下。

* name

返回函数名，这个属性是 ES6 新增的属性，但某些浏览器在 ES6 出来前也实现了这个属性。即使不通过这个属性，也可以通过函数的 toSring() 来获取函数名。

* bind()

用于将当前函数绑定至指定对象，也就是作为指定对象的方法存在。同时，这个函数会返回一个函数类型的返回值，所以通过 bind() 方式，可以实现以函数调用的方式来调用对象的方法。

**function** *f*(y) {  
 **return this**.**x** + y;  
}  
**var *o*** = {**x**:1}  
  
**var *g*** = *f*.bind(***o***);  
***g***(2); *//输出 3*

此时 g 虽然是个函数，但它表示的是对象 o 的方法 f，所以 g() 这种形式虽然是函数调用，但实际上却是调用 o 对象的方法 f，所以方法 f 函数体中的 this 才会指向对象 o。

另外，如果调用 bind() 时传入了多个参数，第一个参数表示需要到的对象，剩余参数会被使用到当前函数的参数列表。

* prototype

该属性直译就是原型，通常是在构造函数中使用，当某个对象是从构造函数实例化出来的，那么这个对象会继承自这个构造函数的 prototype 所指向的原型。

在 prototype 指向的原型对象中添加的属性，会被所有从它关联的构造函数创建出来的对象所继承。所有，数组内置提供的一些属性方法、函数内置提供的相关属性方法，实际上都是在 Array.prototype 或 Function.prototype 中定义的。

而 var a = []; //实际上是 var a = new Array();的语法糖，所以代码中所使用的数组对象，都是默认继承自 Array.prototype，其他引用类型也是类似的道理。

* call() 和 apply()

这两个方法在函数调用一小节中介绍过了，因为在 JavaScript 中的函数的动态的，任意函数都可以作为任意对象的方法被调用，即使这个函数声明在其他对象中。此时，就需要通过间接调用实现，也就是通过 call() 和 apply()。

一种很常见的应用场景，就是用于调用原型中的方法，类似于 Java 中的 super 调用父类的方法。因为子类可能重写了父类的方法，但有时又需要调用父类的方法，那么可通过这个实现。

* toString()

Function.prototype 重写了 Object.prototype 中提供的 toString 方法，自定义的函数会通常会返回函数的完整源码，而内置的函数通常返回 [native code] 字符串。

借助这个可以获取到函数名。

嵌套函数

嵌套函数就是在函数体中继续定义函数，需要跟函数的方法定义区别开来。

函数的方法定义，是将函数看成对象，定义它的属性，类型为函数，这个函数只是该函数对象的方法，并不是它的嵌套函数。

而嵌套函数需要在函数体部分再用 function 定义的函数，这些函数称为嵌套函数。

**var *x*** = 0;  
**var** *a* = **function** () {  
 **var** x = 1;  
 **function** *b*() {  
 ***console***.log(x);  
 }  
  
 **var** *c* = **function** () {  
 ***console***.log(x);  
 }  
  
 *b*(); *//输出：1  
 c*(); *//输出：1  
 a*.d();*//输出：0*}  
  
*a*.d = **function** () {  
 ***console***.log(***x***);  
}

函数 b 和 c 是嵌套在函数 a 中的函数，称它们为嵌套函数。其实本质就是函数体内部的局部变量。

函数 d 是函数 a 的方法。

嵌套函数有些类似于 Java 中的非静态内部类，它们都可以访问外部的变量，Java 的内部类本质上是隐式的持有外部类的引用，而 JavaScript 的嵌套函数，其实是由于作用域链的生成规则形成了一个闭包，以此才能嵌套函数内部可以直接访问外部函数的变量。

闭包涉及到了作用域链，而继承涉及到了原型链，这些概念后面会专门来讲述。

这里稍微提下，闭包通俗点理解也就是函数将其外部的词法作用域包起来，以便函数内部能够访问外部的相关变量。

通常有大括号出现都会有闭包，所以函数都会对应着一个闭包。

高级应用场景

利用函数的特性、闭包特性、继承等，能够将函数应用到各种场景。

* 使用函数作为命名空间

JavaScript 中的变量作用域大概就两种：全局作用域和函数内作用域，函数内定义的变量只能内部访问，外部无法访问。函数外定义的变量，任何地方均能访问。

基于这点，为了保护全局命名空间不被污染，常常利用函数来实现一个临时的命名空间，两种写法：

**var *a***;  
(**function** () {  
 **var** a = 1;  
 ***console***.log(a); *//输出1*})();  
***console***.log(***a***); *//输出undefined*

简单说就是定义一个函数，定义的同时末尾加上 () 顺便调用执行函数体内容，那么这个函数的作用其实也就是创建一个临时的命名空间，在函数体内部定义的变量不用担心与其他人起冲突。

(**function** () {  
 *//...*}());

外层括号不能漏掉，末尾函数调用的括号也不能漏掉，这样就可以了，至于末尾的括号是放在外层括号内，还是外都可以。

* 使用函数封装内部信息

闭包的特性，让 JavaScript 虽然没有类似 Java 的权限控制机制，但也能近似的模拟实现。

因为函数内的变量外部访问不到，而函数又有闭包的特性，嵌套函数可以包裹外部函数的局部变量，那么外部函数的这些局部变量，只有在嵌套函数内可以访问，这样就可以实现对外隐藏内部一些实现细节。

**var** *a* = **function** () {  
 **var** b = 1;  
 **return** {  
 getB: **function** () {  
 **return** b;  
 }  
 }  
}  
***console***.log(***c***.**b**); *//输出 undefined***var *c*** = *a*(); *//输出 1*

函数式编程

…

# JavaScript进阶

<https://github.com/goddyZhao/Translation/tree/master/JavaScript>

## 原型

JavaScript 中并没有Java中的类，但它有构造函数，也有继承，只是它是动态的基于原型的继承。所以，原型有点类似于 Java 中父类的概念。

但是，JavaScript中的关于类、实例、继承等这些跟Java还是有很大的区别。

先来说说在 Java 里面：

类是静态的，类是可继承的，是对象的抽象模型的表现，每个具体的对象都是从类上实例化出来的，一个类中定义了这类对象的属性和行为，一旦定义完了运行时就无法改变了。

但对于JavaScript来说，它并没有类的存在，在JavaScript里，除了原始类型外，其余皆是对象。

它是动态的基于原型的继承机制，原型本质上也是对象，也就是说对象是继承自对象而来的。

而对象这个概念是实例化后的每一个具体个体代表，它是运行期动态生成的，再加上JavaScript里对象的特性，如可动态添加属性，这就让JavaScript里的继承机制非常强大，因为这样一来，它是可动态继承的，原型对象上发生的变化能够同步让继承它的子对象都跟随着变化。

原型概念

函数和构造函数的区别就在于，所有的函数，当和 new 关键字一起使用时，此时称它为构造函数。类似的关系，所有的对象，当它被设置为某个构造函数的 prototype 属性值时，此时称它为原型。

也就是说，任何对象都可以当做其他对象的原型。

在 Java 中，对象一般通过 super 关键字指向它的父类，而在 JavaScript 中，对象可通过 \_\_proto\_\_ 来指向它的原型对象，或者通过构造函数的 prototype 指向对象的原型。

prototype & \_\_proto\_\_

这两个虽然指向的是同一个原型对象，但它们的宿主却不一样，需要区分一下，prototype 是构造函数的属性，\_\_proto\_\_ 是通过构造函数创建出来的对象的属性。

\_\_proto\_\_ 属性并不在ES5标准规范中，但基本大部分浏览器都为引用类型实现了这么一个属性，用于查看当前对象所继承的原型，它的值等于该对象的构造函数的 prototype 属性值。

prototype 是每个函数对象的一个属性，其他对象并没有这个属性，因为基本所有的对象其实都是通过构造函数创建出来的，所以也只有函数才能来实现继承的机制。这个属性值表示着从这个构造函数创建的对象的原型是什么。

对象一节学习过，创建一个对象的三种方式：

* 对象直接量：

**var *a*** = {};*//其实是 var a = new Object(); 的语法糖***var *a*** = [];*//其实是 var a = new Array(); 的语法糖*

* 构造函数
* Object.crate()

所以，对象直接量的方式本质上也是通过构造函数的方式创建对象。

这也是为什么会在对象一节中说，所有通过直接量方式创建的对象都继承自 Object.prototype 的理由。

而通过 Object.create() 方式创建的对象，其原型就是参数指定的对象，可手动传入 null，表示创建的对象没有原型。

所以，在 JavaScript 中，绝大部分的对象都有原型，即使不手动指定，也会有默认的内置原型对象。之所以说绝大部分，是因为原型链顶部的 Object.prototype 对象的原型是 null，或者通过 Object.create() 创建对象时手动指定 null。

* 默认的继承关系

如果不手动指定继承关系，默认的几种引用类型的继承关系（原型链）如下：

* 声明的每个函数 -> Function.prototype –> Object.prototype -> null
* 数组对象 -> Array.prototype -> Object.prototype -> null
* 对象直接量创建的对象 -> Object.prototype -> null
* 自定义构造函数创建的对象 -> {} -> Object.prototype -> null

所有对象继承的顶层原型是 Object.prototype。

这也是为什么函数对象、数组对象、普通对象都可以使用一些内置的方法，因为创建这些对象的时候，默认就会有一些继承关系，跟 Java 中所有的类都继承自 Object 的机制类似。

构造函数和原型的关系

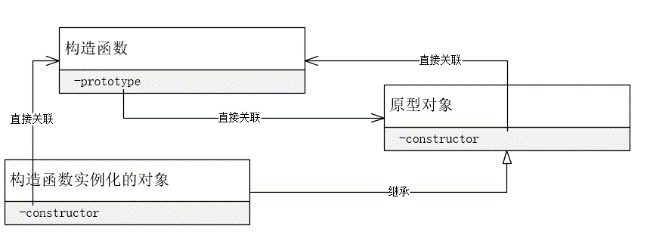
构造函数本身是一个函数对象，它的属性 prototype 指向的是另一个对象，所以这两个概念本身就是两个不同的东西。

通过一个构造函数创建一个新的对象，不能说，这个对象继承自构造函数，而是应该说，这对象继承自构造函数的属性 prototype 指向的对象。

所以，可以通俗的理解，构造函数只是作为第三方类似于工具的角色，用来创建一个新对象，然后让这个新对象继承自 prototype 属性指向的对象。

不过构造函数和原型之间是相互引用的关联关系，构造函数有个属性 prototype 指向原型，而原型也有一个属性 constructor 指向构造函数。

所以，所有从这个构造函数创建的新对象，都继承了原型的属性，那么这些新对象也就可以通过继承而来的 constructor 的属性访问构造函数。

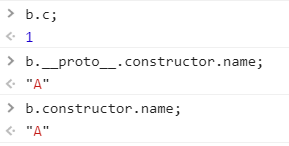
如果不手动破坏原型链，那么通过构造函数创建新对象时，三者间的关系：

而更多的时候，我们需要借助原型来让对象继承一些公有行为，有两种做法，一种是通过直接在原型对象上动态添加相关属性，这种方式不破坏原型链，比较推荐。

还有一种，定义一个新的原型对象，然后重新赋值构造函数的 prototype 属性值，将它指向新的原型对象。但这种方式会破坏默认的原型链，同时也会破坏构造函数、原型、实例化对象三者间的默认关联关系。

举个例子：

**function** *A*(){} *//定义构造函数A  
A*.**prototype**.**c** = 1;**var *b*** = **new** *A*(); *//通过构造函数创建对象b*

通过构造函数创建一个新对象b，且在构造函数的 prototype 上手动添加新的属性c，会被 b 继承，由于这种方式是没有破坏原型链的，所以三者间关系如下：

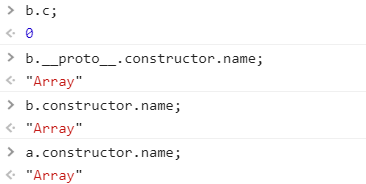
b.\_\_proto\_\_ 表示 b 的原型，原型对象的 constructor 属性指向构造函数 A，name 是函数对象的属性，用于输出函数名。

而且对象 b 由于继承自原型 A.prototype，所以也继承它的 constructor 属性，所以也指向构造函数 A。

此时对象 b 的继承关系：b -> {} -> Object.prototype

以上是默认的不破坏原型链下三者的关系，但如果手动破坏了原型链呢：

**function** *A*(){} *//定义构造函数A  
A*.**prototype**.**c** = 1;  
**var *a*** = []; *//创建数组对象a****a***.**c** = 0;  
*A*.**prototype** = ***a***; *//手动修改构造函数A的prototype，让其指向 a***var *b*** = **new** *A*(); *//通过构造函数创建对象b，b继承自原型a*

上面的代码手动修改了 A.prototype 的属性值，让 b 是继承自手动创建的对象 a，所以这里就破坏了默认的原型链，同时，三者间的关系也被破坏了：

首先，c 属性验证了 b 是继承自对象 a了。

而我们说过，b.\_\_proto\_\_ 指向 b 的原型，在这里，b 的原型就是对象 a了。而对象 a 是手动创建的，所以它的 constructor 属性是继承自它的原型对象。数组直接量创建的数组对象，本质上是通过 new Array()，所以a的构造函数是 Array()，对象 a 继承自 Array.prototype。

对于对象 a，我们创建它的方式并没有手动去修改它的原型链，所以按默认的三者间的关系，Array.prototype 的 constructor 属性指向构造函数 Array()，这就是为什么 b.\_\_proto\_\_.constructor.name 的值会是 Array 了。

而，对象 b 继承自对象 a，所以 b.constructor 的值也才会是 Array。

此时，对象 b 的继承关系： b-> a -> Array.prototype -> Object.prototype

所以，在这个例子中，虽然对象 b 是从构造函数 A 创建的，但它的 constructor 其实并不指向 A，这点也可以稍微说明，构造函数的作用其实更类似于作为第三方协调原型和实例对象两者的角色。

通常是不建议通过这种方式来实现继承，因为这样会破坏默认的三者间的联系，除非手动修复，手动对 a 的 constructor 属性赋值为 A，这样可以手动修复三者间默认的关联。

小结

因为原型本质上也是对象，所以它也具有对象的特性，同时它也有自己的一些特性，总结下：

* 所有的引用类型（数组、对象、函数），都具有对象特性，都可以自由扩展属性，null除外。
* 所有的引用类型（数组、对象、函数），都有一个 \_\_proto\_\_ 属性，属性值的数据类型是对象，含义是隐式原型，指向这个对象的原型。
* 所有的函数（不包括数组、对象），都有一个 prototype 属性，属性值的数据类型是对象，含义是显式原型。因为函数都可以当做构造函数来使用，当被用于构造函数创建新对象时，新对象的原型就是指向构造函数的 prototype 值。
* 所有的内置构造函数（Array、Function、Object…），它的 prototype 属性值都是定义好的内置原型对象，所以从这些内置构造函数创建的对象都默认继承自内置原型，可使用内置的属性。
* 所有的自定义函数，它的 prototype 属性值都是 new Object()，所以所有从自定义构造函数创建的对象，默认的原型链为 （空对象）{} ---- Object.prototype。
* 所有的引用类型（数组、对象、函数），\_\_proto\_\_ 属性指向它的构造函数的prototype值，不手动破坏构造函数、原型之间的默认关系时
* 所有的引用类型（数组、对象、函数），如果不手动破坏原型链，构造函数、原型、实例对象三者之间有默认的关联。

对象的标识

在 Java 中，由于对象都是从对应的类实例化出来的，因此类本身就可以做为对象的标识，用于区分不同对象是否同属一个类的实例。运算符是 instanceof。

在 JavaScript 中，虽然也有 instanceof 运算符，但由于并没有类的概念，虽然有类似的构造函数、原型的概念存在，但由于这些本质上也都是对象，所以很难有某个唯一的标识可以来区分 JavaScript 的对象。

下面从多种思路着手，讲解如何区分对象。

* instanceof

在 Java 中，可以通过 instanceof 运算符来判断某个对象是否是从指定类实例化出来的，也可以用于判断一群对象是否属于同一个类的实例。

在 JavaScript 中有些区别，但也有些类似。

**var *b*** = {}  
**function** *A*() {}  
*A*.**prototype** = ***b***;  
**var *a*** = **new** *A*();  
**if** (***a* instanceof** *A*) { *//符合，因为 a 是从A实例化的，继承自A.prototype即b* ***console***.log(**"true"**);   
}  
  
**function** *B*() {}  
*B*.**prototype** = ***b***;  
**var *c*** = **new** *B*();  
**if** (***c* instanceof** *A*) {*//符合，虽然c是从B实例化的，但c也同样继承自b，而A.prototype指向b，所以满足* ***console***.log(**"true"**);  
}

**if** (***c* instanceof** *Object*) {*//符合，虽然 c 是继承自 b，但 b 继承自 Object.prototype，所以c的原型链中有 Object.prototype*

***console***.log(**"true"**);  
}

在JavaScript中，instanceof 运算符的左侧是对象，右侧是构造函数。但他们的判断是，只要左侧对象的原型链中包括右侧构造函数的prototype指向的原型，那么条件就满足，即使左侧对象不是从右侧构造函数实例化的对象。

也就是说，在JavaScript中，判断某些对象是否属于同一个类的实例，不是根据他们是否是从同一个构造函数实例化的，而是根据他们的构造函数的prototype指向是不是相同的。

通过这种方式来区分对象有点局限是：在浏览器中多个窗口里，每个窗口的上下文都是相互独立的，无法相互比较。

* isPrototypeOf

instanceof 是判断的对象和构造函数两者间的关系，但本质上是判断对象与原型的关系，只是刚好通过构造函数的 prototype 属性值做中转。

那么，是否有可以直接判断对象和原型两者的操作呢？

这个就是 isPrototypeOf 的用法了：左边是原型对象，右边是实例对象，用于判断左边的原型是否在右边实例对象的原型链当中。

***Object***.prototype.isPrototypeOf(***b***);

但它跟 instanceof 有个本质上的区别，instanceof 是运算符，而 isPrototypeOf 是 Object.prototype 中的方法，由于基本所有对象都继承自这个，所以基本所有对象都可以使用这个方法。

instanceof 和 isPrototypeOf 更多使用的场景是用于判断语句中，如果需要主动对某个对象获取它的一些标识，可以使用接下来介绍的几种方式：

* typeof

在 JavaScript 中数据类型大体上分两类：原始类型和引用类型。

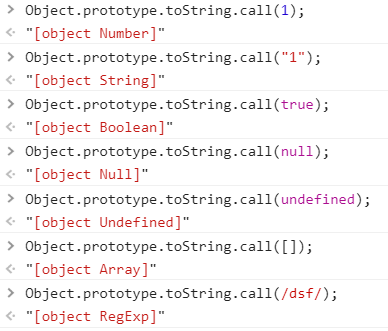
原始类型对应的值是原始值，引用类型对应的值为对象。

对于原始值而言，使用 typeof 运算符可以获取原始值所属的原始类型。

对于函数对象，也可以使用 typeof 运算符来区分：

所以它的局限也很大，基本只能用于区分原始值的标识，对于对象，自定义对象，它的结果都是 object，无法进行区分。

* class 类属性

在对象一节中，介绍过，对象有一个类属性，其实也就是通过 Object.prototype.toString() 方法可以获取包含原始类型和引用类型名称的字符串，对其进行截取可以获取类属性。

相比于 typeof，它的好处在于可以区别所有的数据类型的本质，包括内置引用对象（数组、函数、正则等），也可以区分 null。

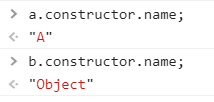
局限在于，需要自己封装个工具方法获取类属性，但这不是难点，问题在于，对于自定义的构造函数，都是返回 Function，而很多对象其实是通过构造函数创建出来的，所以无法区分不同的构造函数所创建的对象。

* constructor 的 name 属性

constructor 是对象的一个属性，它的值是继承自原型的取值。而原型该属性的取值，在不手动破坏对象的原型链情况下，为创建对象的构造函数。

即，默认情况下，构造函数的 prototype 指向原型，原型的 constructor 指向构造函数，那么从该构造函数创建的对象都继承了原型的这个属性可指向构造函数。

所以，在这些场景下，可用对象的 constructor.name 来获取构造函数的函数名，用函数名作为对象的标识。

**function** *A*(){} *//定义构造函数A***var *a*** = **new** *A*();  
**var *b*** = {};

这种方式有个局限，如果手动修改构造函数的 prototype，破坏了对象的原型链，那么此时，新创建的对象的 constructor 就不是指向创建它的构造函数了，此时，这种方式就无法处理了。

由于 JavaScript 不像 Java 这种静态的类结构语言，所以没有一种完美的方式适用于各自场景中来区分对象的标识，只能是在适用的场景选择适合的方式。

所以，在 JavaScript 有一种编程理念：鸭式辩型

鸭式辩型

我不是很理解中文翻译为什么是这个词，应该是某个英文词直译过来的。

它的理念是：像鸭子一样走路、游泳、嘎嘎叫的鸟就称它为鸭子。

通俗点说，编程时，不关心对象所属的标识，不关心对象继承自哪个原型、由哪个构造函数创建，只要这个对象含有相同的属性、行为，那么就认为它们归属于同一类。

有个例子就是：类数组对象，它本质并不是数组对象，但由于具有数组对象的特征，所以基本上可以把它当做数组来使用。

对应到编程中，不应用判断对象是否拥有相同的标识来区分对象，而是应该判断对象是否含有期望的属性即可。

## 继承

继承是面向对象编程语言中一大特性，Java 中的继承是静态的，通过在编写 class 代码过程中指定，一旦继承关系确定了，就无法在运行期间去修改了。

子类默认继承父类的所有非私有的属性和方法。

但在 JavaScript 中，由于并不存在类的机制，而且它是动态的基于原型的继承，所以在很多方面与 Java 的继承并不一样。

下面从多个方面来进行比较：

用法

* Java

**class** MyTask **extends** Thread {}

* JavaScript

**var *a*** = ***Object***.create({**b**:1});*//Object.create方式指定继承的对象***function** *A*() {}  
*A*.**prototype**.**b** = 2; *//构造函数的prototype方式指定继承的对象***var *a*** = **new** *A*();

在 Java 中只能通过 extends 关键字声明继承的关系。

在 JavaScript 中有两种方式指定继承的原型对象，一种用 Object.create()，一种通过构造函数的 prototype 属性。

当在声明一个自定义的构造函数时，内部会自动创建一个空的对象（new Object()），然后赋值给构造函数的 prototype 属性，之后通过该构造函数创建的对象，就都默认继承自 prototype 指向的空对象，所以可在这个空对象上直接动态的添加属性，以便让创建的对象都可以继承这些属性。

继承的内容

* Java

在 Java 中，存在：类，实例对象两种概念。

因此，也就有了类属性、类方法、对象属性、对象方法的说法，这些的区别在于是否有 static 关键字声明。

**public class** Animal {  
 **public int age**; *//对象属性*  
 **public void** eat(){}*//对象方法*  
 **public static void** dead(){}*//类方法*  
}

**public class** Dog **extends** Animal {  
 **public void** growUp(){

eat();*//子类可直接使用父类的非私有方法*  
 *dead*();*//包括类方法*  
 }  
}

Dog dog = **new** Dog();  
dog.**age** = 15; *//对象属性和方法需通过实例对象才可进行操作*  
dog.eat();

dog.dead();*//类属性和类方法不实例化对象也可使用，通过对象也可使用*

Dog.dead();

对象属性和对象方法必须经过类的实例化操作，创建出一个对象来时，才可以通过对象操作这些属性和方法。

而类属性和类方法在子类中可以直接使用，子类实例化的对象也可直接调用。

* JavaScript

在 JavaScript 中只有对象的概念，被继承的对象称为原型。

**function** *Animal*() {}  
*Animal*.prototype.**age** = 0; *//为原型添加属性*  
*Animal*.prototype.eat = **function** () {***console***.log(**"eat"**)}  
  
**function** *Dog*() {}

*//Dog构造函数的prototype继承自Animal.prototype*  
*Dog*.**prototype** = ***Object***.create(*Animal*.prototype);

*Dog*.**prototype**.constructor = *Dog*; *//由于手动修改了原型链，破坏了默认的三者关联，手动修补*  
*Dog*.**prototype**.growUp = **function** () {***console***.log(**"growUp"**)}

*//dog 对象的原型链：dog -> Dog.prototype -> Animal.prototype -> Object.prototype -> null*  
**var *dog*** = **new** *Dog*();  
***dog***.**age**;  
***dog***.eat();  
***dog***.growUp();

先定义个 Animal 构造函数，然后注意，JavaScript 是基于原型继承的，此时如果要定义一些可继承的属性，需要在 Animal.prototype 对象上添加，不可在构造函数本身上添加。

然后再定义一个 Dog 构造函数，让它继承自 Animal.prototype，注意，因为在这里手动修改了原型链，所以最好手动补上 Dog.prototype.constructor = Dog 这行代码，让构造函数、实例对象、原型三者间仍旧可以保持默认的关联关系。

最后，通过构造函数 Dog 创建的对象，就可使用继承而来的属性。

还有另一种写法：

**function** *Animal*() {  
 **this**.**age** = 0;  
 **this**.eat = **function** () {***console***.log(**"eat"**)}  
}  
**function** *Dog*() {}  
*Dog*.**prototype** = ***Object***.create(**new** *Animal*());

可以直接在构造函数 Animal 中添加相关属性，但涉及要继承时，需要使用 new Animal() 作为原型参数，如果直接使用 Animal，那么将会误认将函数对象本身作为原型。

不过这种方式，需要注意，当涉及多个对象需要继承自同一个原型时，原型对象的实例应该只有一个，这样才能保证对原型对象动态修改的属性能同步到所有继承的子对象上。

权限修饰符

Java 中有权限修饰符，子类可以使用父类中非私有的属性和方法。

但在 JavaScript 中，没有公有、私有权限之说，所有定义在原型中的属性，子对象中都可以使用。但可以利用对象属性的特性，在原型中控制它的属性的可枚举性、可配置性、可写性，以此来达到控制子对象访问原型属性的一些限制。

修改对象属性的特性用：Object.defineProperty()

同理，对象本身也有一些特性可利用，比如 Object.freeze()，Object.seal()这类方法可以限制对原型对象进行扩展等操作。

动态同步

Java 中，每个从类实例化出来的对象之间都是相互独立的，不会相互影响，而类属性，类方法只是它们可以用来共享、通信的渠道而已。

而且，类机制是静态的，在Java中，并不会存在在运行期，修改类相关属性而影响子类的场景。

但在 JavaScript 中，由于继承的两者都是对象，而 JavaScript 的对象又具有运行期动态添加属性等特性，所以，如果修改原型上的属性，是会同步到继承该原型的子对象上的。

**function** *A*() {}  
*A*.**prototype**.**num** = 1;  
**var *a*** = **new** *A*();  
**var *b*** = **new** *A*();  
***a***.**num**; *//输出1****b***.**num**; *//输出1，因为都是继承的 A.prototype  
A*.**prototype**.**num** = 5;  
***a***.**num**; *//输出5，原型的属性动态的变化可同步到子对象上****b***.\_\_proto\_\_.**num** = 0;  
***a***.**num**; *//输出0，因为可通过b对象获取原型对象，对原型的操作会同步到子对象上*

以上代码，首先定义了一个构造函数A，通过它创建了两个新的子对象a,b，这两个子对象都继承自A.prototype，所以当访问 a.num 时会输出 1。

然后动态修改 A.prototype 对象的 num 属性，将其改成5，这时会发现，子对象 a 和 b 访问 num 时都输出 5 了，也就是说对原型对象的动态修改属性可同步到它的子对象上。

而子对象又可以通过 \_\_proto\_\_ 属性或者符合默认关系下 constructor.prototype 来获取原型对象，之后对原型对象的操作也可影响到所有继承该原型的子对象。

这点就是 JavaScript 与 Java 这种有类机制语言的很大不同之处。

另外，对原型对象的修改之所以可以同步到子对象上，其实是因为原型链的原理。a，b对象虽然继承自 A.prototype，但其实它们两内部中并没有 num 这个属性，而当访问 num 属性时，在它们内部没找到时，会去沿着原型链中寻找，所以原型对象的属性发生变化时才会影响到子对象。

清楚这点原理后，应该就能理解，有些文章说，原型对象的属性只有读操作会同步到子对象上，写操作无效的原因了吧。

看个例子：

**function** *A*() {}  
*A*.**prototype**.**num** = 1;  
**var *a*** = **new** *A*();  
**var *b*** = **new** *A*();

***a***.**num** = 5;

***b***.**num**; *//输出1*

上面说过，虽然 a 对象继承自 A.prototype，但其实 a 对象内部并没有 num 属性，使用 a.num 时其实会去原型链上寻找这个 num 属性是否存在。

现在，执行了 a.num = 5，因为 a 对象内部没有这个 num 属性，所以这行代码作用等效于动态给 a 对象添加了 num 属性，那这个属性自然也就只属于 a 对象，自然不会对 b 对象造成任何影响，b.num 还是去b的原型链上寻找 num。

改变继承关系

Java 中，类是继承结构一旦编写完毕，在运行期间是不可改变的了。

但在 JavaScript 中，由于对象的属性是可运行期间动态添加、修改的，所以在运行期间是可改变对象的继承结构的。

有两种不同的场景，一是修改构造函数的 prototype 属性，二是修改对象的 \_\_proto\_\_ 属性。

* 修改构造函数 prototype

对象的创建大部分都是通过构造函数，所以，在构造函数创建这个对象时，它的继承关系就确定了。

看个例子：

**var *B*** = []; *//定义一个数组对象***function** *A*() {} *//定义构造函数***var *a*** = **new** *A*(); *//创建一个对象，该对象继承自 A.prototype****a***.\_\_proto\_\_.constructor.**name**; *//应该输出什么*

默认不手动破坏原型链的话，构造函数、原型两者间是相关关联的关系，所以通过实例对象 a 的原型 \_\_proto\_\_ 访问与它关联的构造函数，输出函数名，这里就应该是 “A”。

这也是之前讲过，可用来获取对象的标识—构造函数名的方法，但有前提，就是构造函数、原型、实例对象三者关系满足默认的关联关系。

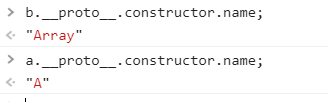
那么，如果这个时候再手动修改 A 的 prototype 属性呢？

举个例子，在上面代码基础上，继续执行下述代码：

**var *C*** = *A*.**prototype**; *//先将 A.prototype 保存下来*

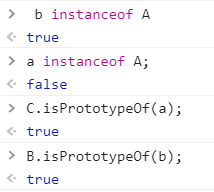
*A*.**prototype** = ***B***; *//手动修改A.prototype*  
**var *b*** = **new** *A*();  
***b***.\_\_proto\_\_.constructor.**name**; *//应该输出什么*  
***a***.\_\_proto\_\_.constructor.**name**; *//应该输出什么*

手动修改了构造函数的 prototype 属性，然后又新创建了 b 对象，那么此时 a 对象和 b 对象都是通过构造函数 A 创建的。

但 a 对象创建时是继承自 A.prototype，这是一个继承自 Object.prototype 的空对象，后续手动修改了构造函数 A 的 prototype，会让 a 对象的继承关系自动跟随着发生变化吗？

我们看一下输出，a 对象仍旧是之前的继承结构，它的原型链并没有因为构造函数的 prototype 发生变化而跟随着变化。

而 b 对象则是在修改了构造函数 prototype 属性后创建的，所以它的原型链就是新的结构了，跟 a 就会有所不同了。这里之所以会输出 Array，是因为 b 的原型是数组对象 B，而数组对象 B 是由 new Array() 创建的，所以 B 继承了 Array.prototype 的 constructor 属性指向了 Array。这也是之前有说过，不建议手动修改原型链结构，否则会破坏默认的构造函数、原型、实例对象三者间的关系。

如果对原型和构造函数的概念还不是很理解，那么我们换个方式验证：

instanceof 表示如果左边的对象是继承自右边构造函数的 prototype 的话，表达式为 true。

isPrototypeOf 表示，左边的对象如果在右边对象的原型链上的话，表达式为 true。

所以，修改构造函数的 prototype 属性，并不会对原本从构造函数创建的对象的原型链，继承结构有所影响。这其实也再次验证，构造函数在 JavaScript 中的角色类似于作为第三方牵手原型和实例对象，修改原型会影响实例对象，但修改构造函数并不会对原本的实例对象有何影响。

但构造函数之后创建的对象，新对象的继承结构跟之前的就不一样了。

* 修改对象的 \_\_proto\_\_ 属性

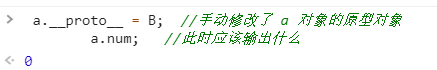
对象有办法直接获取到它的原型对象，一种是通过 \_\_proto\_\_，这是通用方式，所有对象都有，唯一的弊端在于 ES5 中并不是标准规范中的属性，虽然基本所有浏览器中都有实现，所以在一些开发工具中可能不会提示对象含有这个属性。

另一种获取对象原型的方式是，通过 constructor 的 prototype，这也是通用方式，弊端在于，对象的 constructor 属性可能指向的并不是创建它的构造函数，因为这个属性其实是继承自原型对象的属性，所以关键还取决于原型和构造函数之间是否满足默认的相互引用关系。另外，有些对象可能并没有 constructor 属性。

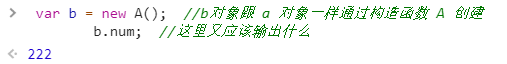
既然对象有属性是指向它的原型，那么手动修改这个属性的指向，会有怎样的影响？

**var *B*** = {**num**:0} *//定义一个对象，含有 num 属性***function** *A*() {} *//定义一个构造函数  
A*.**prototype**.**num** = 222; *//为构造函数prototype添加一个 num 属性***var *a*** = **new** *A*();  
***a***.**num**; *//应该输出什么****a***.**\_\_proto\_\_** = ***B***; *//手动修改了 a 对象的原型对象****a***.**num**; *//此时应该输出什么***var *b*** = **new** *A*(); *//b对象跟 a 对象一样通过构造函数 A 创建****b***.**num**; *//这里又应该输出什么*

a 对象刚被创建来时，是继承的构造函数 A.prototype，所以第一次 a.num 输出 A.prototype.num 的值：222，这里应该没疑问。

然后手动修改了对象 a 的原型，让它的原型指向了 B 对象，那么此时对象 a 的原型链会发生变化吗？它的继承结构会发生变化吗？测试一下：

所以，手动修改对象的 \_\_proto\_\_ 属性是会影响到对象的原型链的，虽然对象在创建时会根据构造函数的 prototype 生成一条原型链，但运行期间，手动修改对象的原型指向，会重新让对象推翻原本原型链，再重新生成一条新的原型链的。

那么，会影响到之后构造函数创建的新对象的原型链吗？测试一下：

所以，手动修改某个对象的原型指向，只会让这个对象的原型链重建，并不会影响到创建它的构造函数之后创建的新对象的继承关系。

* 小结
* 在 JavaScript 中，由于对象继承自原型，但原型本质上也是对象，所以，如果在运行期间动态修改原型对象上的属性，会影响到继承它的子对象们读取相关原型属性的结果。
* 由于继承关系通常是在构造函数创建新对象时，由构造函数的 prototype 属性值决定，而构造函数本质上也是对象，也可在运行期间，动态修改属性值。但如果运行期间，手动修改构造函数的 prototype 属性值，并不会影响到原先通过该构造函数创建的对象的继承结构（原型链），但之后通过该构造函数创建的新对象的继承结构（原型链）就跟之前的不一样了。
* 也就是即使同一个构造函数，但如果有修改过构造函数的 prototype 指向，那么该构造函数前后创建的对象的继承结构（原型链）也是会不一样的。
* 对象有相关的属性指向它的原型，比如 \_\_proto\_\_ ，当运行期间，手动修改对象的原型指向，那么会让这个对象的继承结构（原型链）重建，但不会影响到创建该对象的构造函数原本的行为。
* 总之，对象的继承结构（原型链）可动态发生变化。

重写

重写：子类覆盖父类的同名方法称为重写。

在JavaScript中，重写跟 Java 很类似，使用某个属性时，先在当前对象内部寻找，如果没找到，才往它的原型链上寻找。

但 JavaScript 中并没有 Java 中的类静态机制，所以定义对象的某个属性时，通常都是动态的写操作来进行，一旦在对象中出现对某个原型属性的写操作，那么就会在该对象内部创建一个同名的属性，之后对这个属性的读写，都是对对象内部这个属性的操作，原型上的同名属性的变化也不会影响到它了。

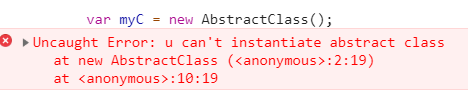
**function** *A*() {} *//定义一个构造函数  
A*.**prototype**.**num** = 222; *//为构造函数prototype添加一个 num 属性***var *a*** = **new** *A*();  
***a***.**num**; *//输出222,****a***.**num** = 0;  
*A*.**prototype**.**num** = 5;  
***a***.**num**; *//输出0, 因为num属性已经被重写了*

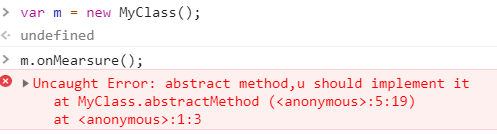
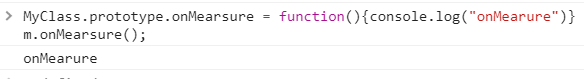
抽象方法

在 Java 中可以定义抽象类，接口，在其中定义一些抽象的方法，子类必须实现这些抽象方法。

但在 JavaScript 中并没有相关的机制，但可以自己通过 throw Error 抛异常形式来模拟这种机制。

比如：

*//不允许使用该构造函数创建对象，来模拟抽象类***function** *AbstractClass*() {  
 **throw new *Error***(**"u can't instantiate abstract class"**);  
}  
*//没有实现的抽象方法，通过抛异常来模拟***function** *abstractMethod*() {  
 **throw new *Error***(**"abstract method,u should implement it"**);  
}  
*//定义抽象方法，子类继承之后，如果不自己实现，直接使用会抛异常  
AbstractClass*.prototype.onMearsure = *abstractMethod*;  
*AbstractClass*.prototype.onLayout = *abstractMethod*;

**function** *MyClass*() {}  
*MyClass*.**prototype** = ***Object***.create(*AbstractClass*.prototype);

子类继承后，如果不实现直接调用这些方法，会抛异常。

说白了，就是通过抛异常方式来模拟 Java 中的抽象方法机制，这种方式无法让开发工具在编写代码期间就检测出来，需要代码实际运行期间才能发现。

## 原型链

原型链也就是对象的继承结构，举个例子：

**var *a*** = [];

那么 a 对象的原型链：

a -> Array.prototype -> Object.prototype -> null

基本所有对象的原型链顶部都是 Object.prototype，而 Object.prototype 没有原型，手动通过 Object.create(null) 创建的对象也没有原型。但这两点是特例。

原型的用途在于让对象可继承原型上的属性，达到功能复用、代码复用的目的。

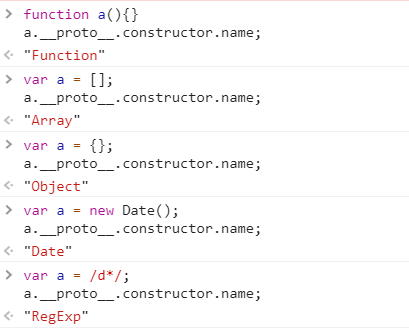
面向对象的编程语言中，继承是一大特性，所以在编写 JavaScript 代码时，要能够很明确所创建的对象的一个原型链结构，这样才便于更好的设计，更好的编写代码。

在编写代码过程中，使用的无非就是内置对象，或者自定义对象，所以下面来看看两者的原型链结构：

内置对象的原型链结构

其实也就是之前有讲过的默认的原型链结构：

* 声明的每个函数 -> Function.prototype –> Object.prototype -> null
* 数组对象 -> Array.prototype -> Object.prototype -> null
* 对象直接量创建的对象 -> Object.prototype -> null
* 日期对象 -> Date.prototype -> Object.prototype -> null
* 正则对象 -> RegExp.prototype -> Object.prototype -> null

可以用对象的 \_\_proto\_\_.constructor.name 来测试：

Object.prototype 已经内置定义了一些属性，如：toString()，isPrototypeOf()，hasOwnProperty() 等等；

同样，Array.prototype 内置了如：forEach()，map() 等等。

其他内置原型也都有相对应的一些属性。

所以使用内置对象时，才可以直接使用内置提供的一些属性。

* 自定义对象的原型链结构

不手动修改自定义构造函数的 prototype 属性的话，默认创建的对象的原型链结构：

* 自定义构造函数创建的对象 -> {} -> Object.prototype -> null

比如：

**function** *A*() {}

**var *a*** = **new** *A*();

在首次使用构造函数 A 时，内部会去对 prototype 属性赋值，所进行的工作类似于：A.prototype = new Object();

所以 A.prototype 会指向一个空对象，但这个空对象继承了 Object.prototype。

那么不修改这条原型链的话，默认通过自定义构造函数创建的对象的继承结构也就是：{} –> Object.prototype –> null。

虽然这条原型链也可以这么表示：A.prototype –> Object.prototype -> null

a 虽然确实继承自 A.prototype，但我不倾向于这种写法来表示，因为自定义构造函数的 prototype 属性值会有很大的可能性被修改掉，当它的属性值重新指向另一个对象后，此时也仍旧可以说 a 对象继承自 A.prototype，个人感觉理解上会有点别扭，无法区别前后原型的不同，毕竟 A.prototype 只是一个 key 值，所以我倾向于直接说 a 继承的实际对象，也就是 key 值对应的 value 值。

虽然 Object.prototype 也是一个 key 值，实际指向的一个内置的对象，但手动修改这些内置构造函数的 prototype 的可能性不高，所以个人觉得对于内置构造函数，可以直接用类似 Object.prototype 来表示。

那么这个时候，如果为这个构造函数的 prototype 添加一些属性：

**function** *A*() {}

*A*.**prototype**.**num** = 0;

**var *a*** = **new** *A*();

那么，对于对象 a 而言，它的原型链：

a -> {num:0} -> Object.prototype -> null

这是不修改原型链的场景，那么如果手动破坏了默认的原型链呢？

**var *B*** = [];  
***B***.**num** = 0;   
**function** *A*() {}   
*A*.**prototype**.**num** = 222;

**var *a*** = **new** *A*(); *//a 的原型链*  
*A*.**prototype** = ***B***;  
**var *b*** = **new** *A*(); *//b 的原型链*

此时对象 b 的原型链又是什么呢？

首先看看对象 B，是一个数组对象，所以 B 对象的原型链：

B –> Array.prototype -> Object.prototype -> null

再来看看对象 a，创建它时，还并没有修改构造函数的 prototype，所以它的原型链：

a -> {num:222} -> Object.prototype -> null

那么这个时候，手动修改掉了构造函数的 prototype 指向，这之后再通过构造函数 A 创建的对象的原型链也就会跟随着变化，所以对象 b 的原型链：

b -> B –> Array.prototype -> Object.prototype -> null

所以，修改构造函数的 prototype，其实相当于将另外一条原型链拿来替换掉原本的原型链。

* 原型链用途

对于对象，它的本质其实也就是一堆属性的集合，所以对象的用途是用来操作对象内的属性的，而当操作对象的属性时，会有一种类似于作用域链机制来寻找属性。

操作无非分两种场景，一是读取对象属性，二是写对象属性，两种所涉及的处理不一样。

当读取对象属性时，是依靠对象的原型链来辅助工作，如果对象内部含有该属性，则直接读取，否则沿着原型链去寻找这个属性。

也就是说，对象继承原型的机制，并不是说，将原型的所有属性拷贝一份到对象内部，而只是简单对对象建立一条原型链而已。这条原型链中保存着各个原型对象的引用，当读取继承的属性时，就可以根据这条原型链上的引用访问到其他原型对象内的属性了。

因为读取继承属性，本质上是读取其他对象的属性，那么，这些原型属性发生变化时，也才会影响到继承他们的子对象。

那么，对于写对象属性的操作：

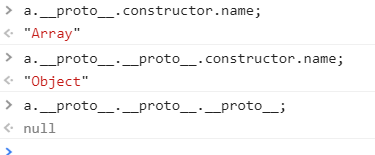
这点就由对象的特性决定了：当对一个对象的属性进行赋值操作时，如果对象内没有该属性，那么会动态为该对象添加一个属性，如果对象内部有该属性，那么修改属性值。

对象的属性写操作会影响到后续的读操作，因为如果是读取对象的某个继承属性，本来对象内部没有该属性，所以是去读取的原型内的属性值。但经过写操作后，对象内部创建了同名的内部属性，之后再读取时，发现内部已经有了，自然不会再去原型链中读取。

* 获取对象的原型链

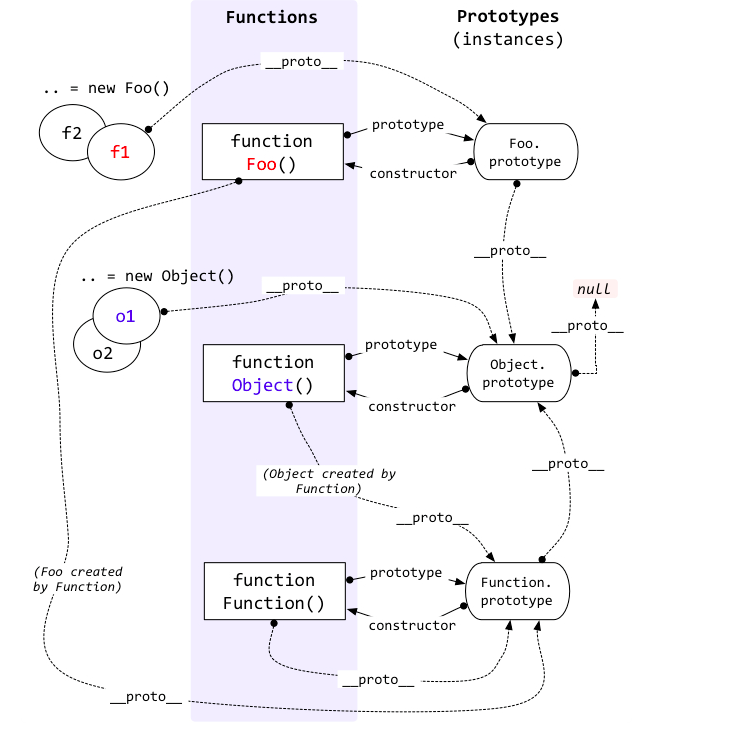
掌握了原型链的相关理论，对于代码中某个对象的原型链也就能够很清楚的知道了。无外乎内置对象的默认原型链，或者自定义构造函数手动修改的原型链。

但，初学阶段，如果想借助浏览器的开发者工具的 console 来测试、查看对象的原型链以便验证猜想，可以这么处理：

**var *a*** = [];

虽然 \_\_proto\_\_ 可以获取原型，但拿到的是对象，所以可以借助对象的某些标识，比如原型的 constructor 的 name 函数名属性标识。

* 实例

网上关于原型链的文章经常会出现这么一张图片，首先我承认，这张图很高级，也基本把原型链的相关理论表示出来了，但我很不喜欢它。因为对于新手来说，很难看懂这张图，我第一次看到也一脸懵逼。

就算现在能够看懂了，我也还是不喜欢它，因为这张图表达的内容太多了：它不仅表示了某个对象的原型链结构，同时，也表示出了实例对象、原型、构造函数三者间的函数，而构造函数本质上也是对象，所以也顺便表示它的原型链结构。

我们一步步来看，它首先定义了一个构造函数 Foo，然后通过它创建了 f1,f2对象，然后从 f1,f2开始出发，先求他们的原型链。

用代码来说，其实也就是：

**function** *Foo*() {}  
**var *f1*** = **new** *Foo*();  
*//求f1对象的原型链*

根据我们上述梳理的理论，很简单了吧，原型链其实也就是：

f1 -> {} -> Object.prototype -> null

接着，它表达了可以用 \_\_proto\_\_ 获取对象的原型，然后每个原型、构造函数、实例对象三者间的关系它也表达出来了，原型的constructor指向构造函数，而构造函数的prototype指向原型。

而这三个角色本质上也都是对象，既然是对象，那么它们本身也有原型，所以也再顺便画出它们的原型链。

总之，就是从 f1 实例对象出发，先找它的原型，通过原型再找构造函数，然后再分别将原型和构造函数看成实例对象，重复之前f1的工作。

另外，又通过 new Object() 创建了对象 o1，求它的原型链。

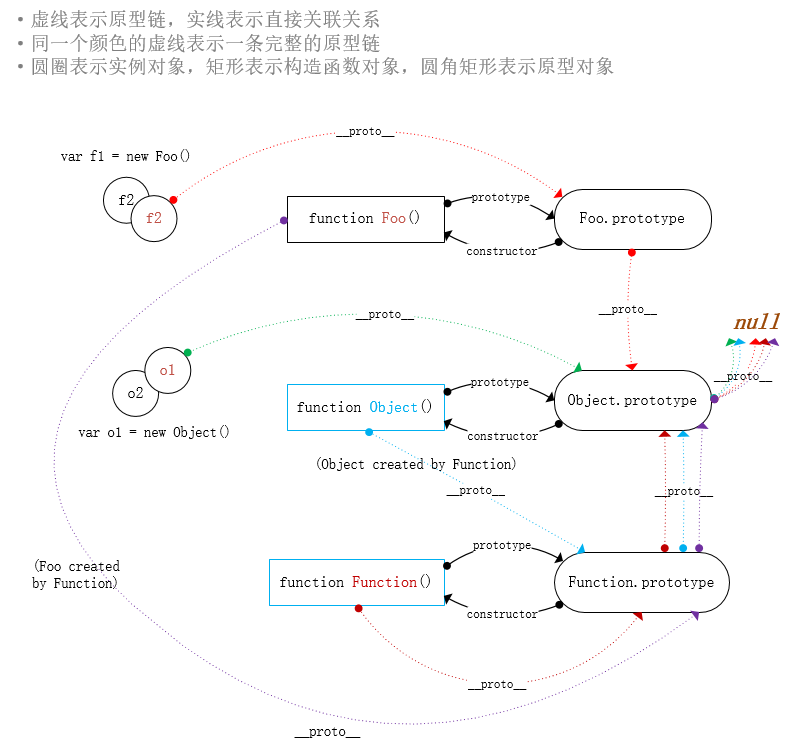
所以，这张图上，其实表达了一共 5 条原型链，分别是:

* f1 的原型链
* f1 的原型的constructor指向的构造函数Foo对象的原型链
* 函数对象Foo的原型的constructor指向的构造函数Function对象的原型链
* f1 的原型的原型即Object.prototype的constructor指向的构造函数Object 对象的原型链。
* o1 的原型链

如果你能从这张图看出这5条原型链，那么原型链的理论你就基本掌握了。

而且，建议看这张图时，每次都将某条原型链跟踪到底，再去看另一条，这过程不要过多关注在分支上，否则很容易混乱。

对于新手，如果能够对这张稍作备注，而不是直接将这张图放出来，我觉得会更好，如下：



## 执行上下文和变量对象（EC&VO）

EC：Execution Context，中文翻译执行上下文，也有翻译成执行环境的。

VO：Variable object，中文翻译变量对象。

这两个概念很重要，涉及到作用域以及作用域链的原理。

执行上下文 EC

* Android

先说说Android中的上下文：

Android 中也有上下文：Context，四大组件都是上下文，还有一个全局的 Application上下文。在Android中基本是以四大组件为界限，每创建一个四大组件，都会产生一个上下文，比如每个 Activity 都是独立的上下文。

在 Android 中，上下文 Context 的作用大体上用于标识各种资源的所属，要加载一张图片、创建一个 View、弹一个 Dialog等等，你需要告诉系统，这些是谁发出的指令，要挂载到哪个上下文，这些资源依赖于上下文的生命周期。

所以才会出现，有时弹Dialog或者更新某个 View时抛异常说Context已销毁，因为它需要挂载的上下文已经销毁了，那么就没有上下文来统筹管理这些资源了，自然会抛异常。

* JavaScript

在 JavaScript中，上下文是指执行上下文，通俗点理解，代码执行的上下文，所以也有翻译成执行环境，可以通俗的把它理解成一个对象，对象名EC，表示代码的执行上下文。

既然理解成一个对象，那么就有它创建的时机，在JavaScript中，每当要执行不同类型的代码时，就会创建一个执行上下文EC。

而代码的类型分三种：

* 全局代码
* 函数代码
* eval()执行的代码

最后一种不讨论，全局代码就是指写在函数外的代码，在前端里，当HTML加载一个 js 文件时，全局代码就会被执行，那么在全局代码执行前就会先创建一个全局的执行上下文EC，之后每调用一次函数，要执行函数内的代码时，会再创建一个函数执行上下文EC。

也就是说，不讨论eval的话，那么在 JavaScript有两种执行上下文，一种是全局执行上下文，一种是函数执行上下文。

而每次创建一个执行上下文时，都会将其放入一个栈结构，这个栈就称为执行上下文栈（ECS），也有翻译成执行环境栈。

所以执行js文件代码期间，这个栈底一直是全局执行上下文，直到js文件代码执行结束。全局代码执行过程中，每调用一次函数，新创建一个函数执行上下文，就放入栈内。

因此，栈顶就表示当前执行的代码，如果栈顶是全局执行上下文，表示正在执行全局代码；如果栈顶是函数执行上下文，表示正在执行函数内的代码。当函数执行结束时，这个函数执行上下文就从栈中移出。

那么执行上下文（EC）有什么用呢？

用途可多了，跟Android不一样，Android里由于是各种资源的组合使用，但在JavaScript中更多的是嵌套函数的变量使用。所以，用途之一就是保存各个变量。

将EC理解成一个对象的话，它有两个属性，一个是变量对象（VO），另一个是作用域链（Scope Chain）。对于全局执行上下文，当HTML加载一个js文件时，就会创建一个全局EC，此时会创建它的两个属性：变量对象和作用域链。之后，每调用一次函数，创建这次函数执行的上下文，函数内部的变量的使用就依赖于这个函数执行上下文中的变量对象和作用域链。

也就是说，内部函数之所以可以使用外部函数内的变量，之所以可以使用全局变量，都是依赖于当前这个内部函数的执行上下文。

而且，变量之所以会提前声明也是因为执行上下文的因素。这些当讲解了执行上下文EC的创建过程就清楚了。

变量对象（VO）

变量对象只是一个抽象的概念，可以通俗的理解成保存当前上下文所有变量的对象。

在不同的执行上下文中，它有不同的具体表现。

在全局执行上下文中，变量对象VO的具体表现就是全局对象，因为所有的全局变量其实都是全局对象的属性，而变量对象VO的作用是要保存当前上下文中的所有变量，所以此时的变量对象VO实际上是指向的全局对象。

尤其在前端中，全局对象就是 window，所以全局执行上下文的变量对象VO = window。

在函数执行上下文中，因为变量对象VO是要保存当前上下文中所有的变量，一个函数内的变量包括：形参变量、局部变量、自身函数对象引用变量、arguments、this。

针对函数执行上下文，为了保存这些变量，特意创建了一个对象，称它为活动对象AO，函数内所需的变量就都保存在AO中，所以在函数执行上下文中，变量对象VO的具体表现也就是AO。

小结：变量对象VO是一个抽象概念，用于保存当前执行上下文中所有的变量。所以在全局执行上下文中，因为全局对象已经保存着当前上下文所有的变量，所以VO在这里的具体实现就是全局对象。在函数执行上下文中，由于要保存函数形参、局部变量、自身函数对象引用变量、arguments、this，所以新创建了一个叫活动对象AO来保存，此时VO的具体实现就是AO。

作用域链（Scope chain）

每次函数调用时，都会创建一个函数执行上下文EC，但其中的变量对象VO只保存着当前上下文中的变量而已，那么函数内如果需要使用到外部函数的变量，甚至是使用全局的变量时，此时就需要依赖于执行上下文的另一个属性：作用域链。

作用域链本质上，其实是将有嵌套层次关系的执行上下文的VO拼接起来。

所以大部分场景作用域链只有两个节点：当前函数执行上下文的VO –> 全局执行上下文的VO。所以函数内才可以根据作用域链访问全局内的变量。

当出现函数内再嵌套函数时，此时作用域链就会比较长：

内层函数执行上下文的VO –> 外层函数执行上下文的VO –> 全局执行上下文 VO。

至于作用域链是如何将有嵌套层次的执行上下文的VO拼接起来，需要借助函数对象的内部属性[[Scope]]，[[]]表示执行引擎为对象创建的内部属性，我们访问不了，也操作不了。具体原理在作用域链一节中讲解。

## 作用域

在 ES5 中，变量的作用域只有两类：

* 全局作用域
* 函数作用域

只要不是在函数内部定义的变量，作用域都是全局的，全局的变量在哪里都可以被访问到，即使跨 js 文件。

函数作用域是指在函数体定义的变量，不管有没有在函数体的开头定义，在函数体的任何地方都可以被使用，因为JavaScript中的变量有声明提前的行为。

函数作用域需要区别于 Java 语言中的块级作用域：

**var *i*** = 0;

**function** *A*() {  
 ***console***.log(i); *//输出undefined* **for** (**var** i = 0; i < 1; i++) {}  
 ***console***.log(i); *//输出1*}

在 Java 中，类似的代码，在 for 循环前后输出的 i 都会是 0，因为都会使用成员变量 i，for循环内定义的 i 由于块级作用域限制，只在for 循环的 {} 大括号中的代码有效。

但在 JavaScript 中，变量作用域只分函数作用域，而且变量有声明提前的特性，所以在函数体内部第一次输出 i 时，此时变量已经提前声明，但还没初始化，所以会是 undefined。而函数内定义的变量的作用域或者说生命周期是整个函数内，所以即使for循环体语句结束，仍旧可以访问到 i 变量。

由于允许变量的重复定义，所以全局变量很容易起冲突，因为无法确保多份 js 文件中是否已经在全局中定义了该变量，一旦起冲突，浏览器行为仅仅是将后定义的覆盖掉前定义的而已，这对于浏览器角度没什么大问题，但对于程序而已，很容易出现不可控的问题。而且，极难排查。

所以，实际编程中，建议不要过多的使用全局变量，有多种方法可以避免：

* 使用一个全局对象来作为命名空间，将其余不在函数体内部定义的变量，作为该全局对象的属性来定义使用。
* 使用一个立即执行的函数来作为临时命名空间，函数执行结束释放临时命名空间。
* 如果临时命名空间内的部分变量需要供外部使用，一可以将这部分变量添加到作为命名空间的全局对象上的属性，二可以利用闭包的特性，返回一个新建的对象，为该对象添加一些接口可访问这部分变量。

全局对象作为命名空间

**var *DASU*** = {};  
***DASU***.**num** = 1;  
**function** *a*() {  
 ***console***.log(***DASU***.**num**);  
}

这里的全局对象意思是说，数据类型为对象的全局变量，简称全局对象，与前端里说的全局对象window是两个不同概念，区分一下。

其实也就是一种思想，将所有函数外需要定义的变量，都替换成对指定对象的属性来操作。

立即执行的函数作为临时命名空间

(**function** () {  
 **var** num = 1;  
 **function** *a*() {  
 ***console***.log(num);  
 }  
 *a*();  
}())

当引入 js 文件到 HTML 时，js 文件中的代码就会被执行，或者声明了 <script> 标签后，在标签内的代码也会立马被执行。但函数只有被调用的时候才会执行，所以，如果我们使用一个立即执行的函数，那这个函数体内部的代码行为就跟正常的 js 文件代码被执行的行为一致了。

而且，还可以利用函数内作用域这一特点，来保证，在这个立即执行的函数内部定义的变量不会影响到全局变量。

缺点就是函数内部代码执行结束后，这些在函数内定义的变量就被回收了。所以，如果有些信息需要跨 js 文件通信，此时要么通过全局对象方式，要么通过闭包特性来辅助实现。

临时命名空间内的变量共享方式

全局变量可以在任何地方被访问，所以可以将那些需要共享给外部使用的临时命名空间内的变量赋值给全局对象的属性，即结合第一种：全局对象做命名空间方式。

或者，通过闭包的特性，作为临时命名空间的立即执行的函数需要有一个返回值，当外部持有这个返回值时，这个函数内的变量就不会被回收。

然后，返回值可以是一个对象，公开一些接口来获取这些需要共享的变量，如：

**var *model*** = (**function** () {  
 **var** num = 1;  
 **function** *a*() {  
 ***console***.log(num);  
 }  
 **return** {  
 getNum: **function** () {  
 **return** num;  
 }  
 }  
}());  
***model***.getNum();

或者：

**var *model*** = (**function** () {  
 **var** num = 1;  
 **function** *a*() {  
 ***console***.log(num);  
 }  
 **return** {  
 **num**:num  
 }  
}());  
***model***.num;

变量声明提前原理

看个例子：

**var *i*** = 0;  
**function** *A*() {  
 ***console***.log(i); *//输出undefined* **for** (**var** i = 0; i < 1; i++) {}  
 ***console***.log(i); *//输出1*}

*A*();

函数内第一个输出 undefined 是因为变量的声明提前，第二个输出 1 是因为变量作用域为函数作用域，而不是块级作用域。

那么，有想过，这些似乎理所当然的基础常识原理是什么吗？

我们先来看些理论，再结合理论返回来分析这个例子，但只分析变量的声明提前原理，至于作用域的原理留着作用域链一节分析。

* 理论

我们之前有介绍过执行上下文EC，和变量对象VO，执行上下文分全局执行上下文和函数执行上下文。在全局执行上下文中，VO的具体表现是全局对象；在函数执行上下文中，VO的具体表现是AO，AO存储着函数内的变量：形参、局部变量、函数自身引用、this、arguments。

不管是执行函数代码还是全局代码，js解释器会分两个过程，有的文章翻译成：进入执行上下文阶段、执行代码阶段（我不怎么喜欢这个翻译）。

进入执行上下文阶段：其实本质上就是创建一个执行上下文，这个阶段会解析当前上下文内的代码，将声明的变量都保存到VO对象上。

执行代码阶段：就是代码实际运行期，当运行到相对应的变量的赋值语句时，就会将具体的属性值写入VO对象上保存的对应变量。

也就是说，在执行代码阶段，代码实际运行时，js解释器已经解析了一遍上下文内的代码，并创建了执行上下文，且为其添加了一个VO属性，在VO对象上添加了上下文内声明的所有变量，这就是变量的声明提前行为。而之后函数体内对各变量的操作，其实是对VO对象上保存的变量进行操作了。

我看过一篇文章对这两个过程的翻译是：解析阶段、执行阶段。

我比较喜欢这种翻译，解析阶段主要的工作就是解析上下文内的代码，创建执行上下文，创建变量对象VO等，为执行阶段做准备；而执行阶段就是代码实际运行过程。

* 分析

**var *i*** = 0;   
**function** *A*() {  
 ***console***.log(i); *//输出undefined* **for** (**var** i = 0; i < 1; i++) {}  
 ***console***.log(i); *//输出1*}

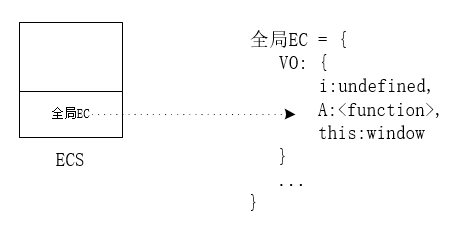
*A*();

再回过头来看这个简单的例子，假设这段代码放在一份单独的 js 文件中，解释器第一次执行这份代码，那么当执行全局代码时，首先进入全局执行上下文的解析阶段：

1. 解析代码创建全局执行上下文

2. 创建VO，并为其添加属性 i、A

3. 省略该过程其他工作

4. 将创建的全局EC放入ECS栈内。

当实际开始执行第一行全局代码时，js解释器经过了解析阶段已经做了如上的工作，得到了一些基本的信息。之后便是执行全局代码，如果执行的代码是访问全局变量，那么直接读取全局EC中VO里的对应变量；如果是对全局变量赋值操作，那么写入全局EC中的VO里对应变量的属性值。

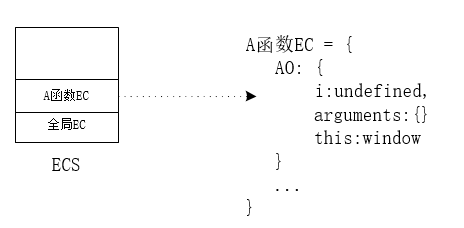
如果执行的代码是调用某个函数，此时就会为这个函数的执行创建一个函数执行上下文，那么这个过程同样需要两个阶段：解析阶段和执行阶段。

所以当代码执行到最后一行 A() 时，此时新的函数执行上下文的解析阶段做的工作：

1. 解析A() 函数内代码，并创建函数执行上下文 A函数EC

2. 创建 AO，并为其添加属性：i

3. 省略其他工作介绍

4. 将创建的A函数EC放入ECS栈内

所以当执行函数A内的代码时，第一行输出才会输出 undefined，因为变量的声明提前特性在调用函数时创建函数执行上下文的过程中，已经解析了函数内的声明语句，并将这些变量添加到函数上下文EC的AO中了。

AO就是变量对象VO在函数执行上下文中的具体表现。

而当执行完 for 循环语句，A函数EC中的AO里的i属性已经被赋值为 1了，而A函数EC是直到函数执行结束才销毁，所以即使在for语句内定义的 i 变量也可以在后面继续使用。

以上，就是变量声明提前的原理，当然，创建执行上下文的过程中，还涉及到其他很多工作，用来实现例如作用域链等机制，留待后续来说。

## 作用域链

作用域一节中，我们介绍了变量的作用域分两种：全局和函数内，且函数内部可以访问外部函数和全局的变量。

我们也介绍了，每个函数被调用时，会创建一个函数执行上下文EC，EC里有个变量对象VO属性，函数内部操作的局部变量就是来源于VO，但VO只保存当前上下文的变量，那么函数内部又是如何可以访问到外部函数的变量以及全局变量的呢？

本篇就是来讲讲作用域链的原理，理清楚这些理所当然的基础知识的底层原理。

先来看个例子，再看些理论，最后结合理论再回过头分析例子。

实例

**var *num*** = 0;  
**var *sum*** = -1;  
**function** *a*() {  
 ***console***.log(num); *//1. 输出什么* **var** *b* = **function** () {  
 ***console***.log(num++);  
 }  
 **var** num = 1;  
 *b*(); *//2. 输出什么* ***console***.log(***sum***); *//3. 输出什么* **return** *b*;  
}  
  
**var *c*** = **function**(num) {  
 **var** d = *a*();  
 d(); *//4. 输出什么*}  
  
*c*(10);

当执行了最后一行代码时，会有四次输出，每次都会输出什么，可以先想想，然后再继续看下去，对比下你的答案是否正确。

理论

作用域链的原理还是跟执行上下文EC有关，执行上下文EC有个作用域链属性（Scope chain），作用域链是个链表结构，链表中每个节点是一个VO，在函数内部嵌套定义新函数就会多产生一个节点，节点越多，函数嵌套定义越深。

由于作用域链本质上类似于VO，也是执行上下文的一个属性，那么，它的创建时机自然跟EC是一样的，即：全局代码执行时的解析阶段，或者函数代码执行时的解析阶段。

每调用一次函数执行函数体时，js解释器会经过两个阶段：解析阶段和执行阶段；

调用函数进入解析阶段时主要负责下面的工作：

1. 创建函数上下文

2. 创建变量对象

3. 创建作用域链

创建变量对象的过程在作用域一节中讲过了，主要就是解析函数体中的声明语句，创建一个活动对象AO，并将函数的形参列表、局部变量、arguments、this、函数对象自身引用添加为活动对象AO的属性，以便函数体代码对这些变量的使用。

而创建作用域链的过程，主要做了两件事：

1. 将当前函数执行上下文的VO放到链表头部

2. 将函数的内部属性[[Scope]]存储的VO链表拼接到VO后面

两个步骤创建了当前函数的作用域链，而当函数体的代码操作变量时，优先到作用域链的表头指向的VO寻找，找不到时，才到作用域链的每个节点的VO中寻找。

那么，函数的内部属性[[Scope]]存储的VO链表是哪里赋值的？

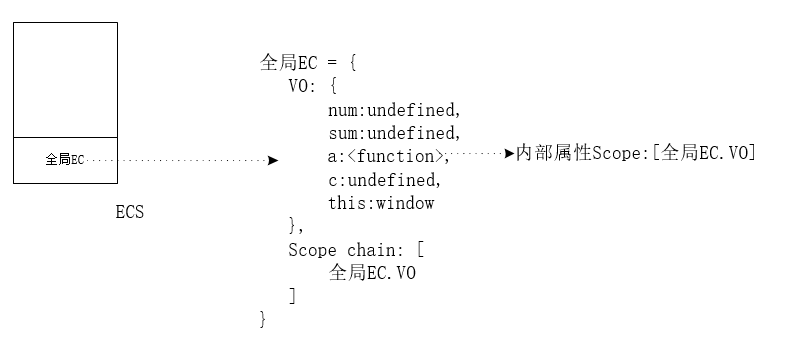
这部分工作也是在解析阶段进行的，只不过是外层函数被调用时的解析阶段。解析阶段会去解析当前上下文的代码，如果碰到是变量声明语句，那么将该变量添加到上下文的VO对象中，如果碰到的是函数声明语句，那么会将当前上下文的作用域链对象引用赋值给函数的内部属性[[Scope]]。但如果碰到是函数表达式，那[[Scope]]的赋值操作需要等到执行阶段。

所以，函数的内部属性[[Scope]]存储着外层函数的作用域链，那么当每次调用函数时，创建函数执行上下文的作用域链属性时，直接拼接外层函数的作用域链和当前函数的VO，就可以达到以函数内部变量优先，依照嵌套层次寻找外层函数变量的规则。

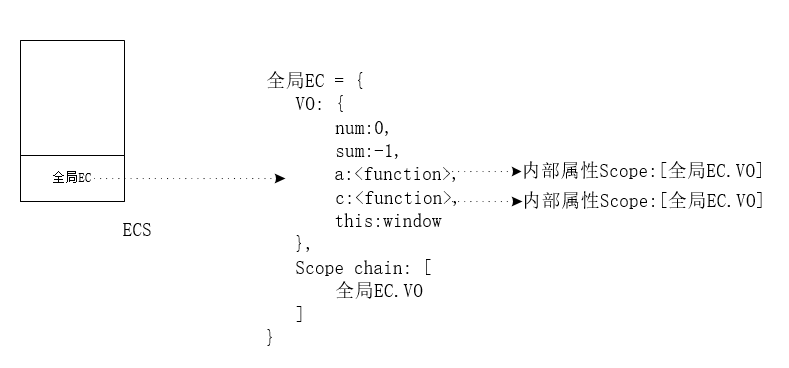
这也是为什么，明明函数的作用域链是当函数调用时才创建，但却依赖于函数定义的位置的原因。因为函数调用时，创建的只是当前函数执行上下文的VO。而函数即使没被调用，只要它的外层函数被调用，那么外层函数创建执行上下文的阶段就会顺便将其作用域链赋值给在它内部定义的函数。

分析

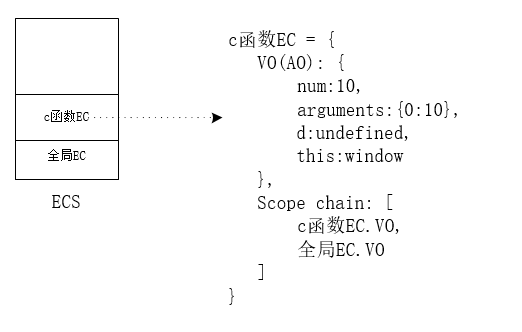
**var *num*** = 0;  
**var *sum*** = -1;  
**function** *a*() {  
 ***console***.log(num); *//1. 输出：undefined* **var** *b* = **function** () {  
 ***console***.log(num++);  
 }  
 **var** num = 1;  
 *b*(); *//2. 输出：1* ***console***.log(***sum***); *//3.输出：-1* **return** *b*;  
}  
  
**var *c*** = **function**(num) {  
 **var** d = *a*();  
 d(); *//4. 输出：2*}  
  
*c*(10);

1. 当第一次执行全局代码时，首先创建全局执行上下文EC：

所以，当进入执行阶段，开始执行全局代码时，全局变量已经全部添加到全局EC的VO里的，这也就是变量的提前声明行为，而且对于全局EC来说，它的作用域链就是它的VO，同时，因为解析过程中遇到了函数声明语句，所以在解析阶段就创建了函数a对象（a:<function> 表示 a 是一个函数对象），也为函数 a 的内部属性 [[Scope]] 赋值了全局EC的作用域对象。

2. 全局代码执行到 **var *c*** = **function**(num) 语句时：

相应的全局变量在执行阶段进行了赋值操作，那么，赋值操作实际操作的变量就是对全局EC的VO里的相对应变量的操作。

3. 当全局代码执行到 c(10)，调用了函数c时：

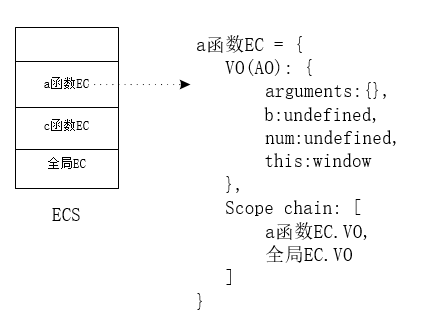
也就是说，在c函数内部代码执行之前，就为c函数的执行创建了c函数执行上下文EC，这个过程中，会将形参变量，函数体声明的变量都添加到AO中（在函数执行上下文中，VO的具体表现为AO），同时创建 arguments 对象，确定函数内 this 的指向，由于这里的普通函数调用，所以 this 为全局对象。

最后，会创建作用域链，赋值逻辑用伪代码表示：

Scope chain = c函数EC.VO -> c函数内部属性[[Scope]]

= c函数EC.VO -> 全局EC.VO

图中用数组形式来表示作用域链，实际数据结构并非数组，所以，对于函数c内部代码来说，变量的来源依照优先级在作用域链中寻找。

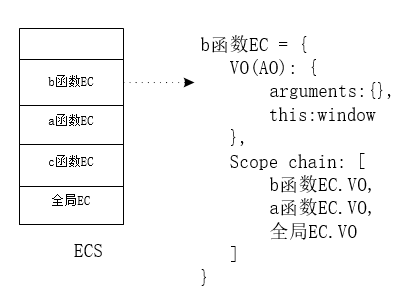
4. 当函数c内部执行到 **var** d = *a*(); 调用了a函数时：

同样，调用 a 函数时，也会为函数a的执行创建一个函数执行上下文，a 函数跟 c函数一样定义在全局代码中，所以在全局EC的创建过程中，已经为 a 函数的内部属性[[Scope]]赋值了全局EC.VO，所以a函数EC的作用域链同样是：a函数EC.VO -> 全局EC.VO。

也就是作用域链跟函数在哪被调用无关，只与函数被定义的地方有关。

5. 接下去开始执行a函数内部代码，所以第一行执行 console.log(num) 时，需要访问到 num 变量，去作用域链中依次寻找，首先在 a函数EC.VO 中找到 num:undefined，所以直接使用这个变量，输出就是 undefined。

6. 接下去执行了 var b = function ()，创建了一个函数对象赋值给b，同时对 b 函数的内部属性[[Scope]] 赋值为当前执行上下文的作用域链，所以b函数的内部属性[[Scope]]值为：a函数EC.VO -> 全局EC.VO

7. 接下去执行到 b()，调用了b函数，所以此时：

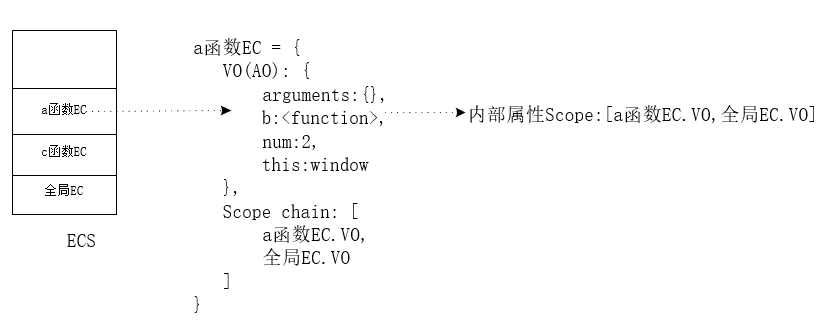
同样，也为b函数的执行创建了函数执行上下文，而作用域链的取值为当前上下文的VO 拼接上当前函数的内部属性[[Scope]]值，这个值在第6步中计算出来。所以，最终b函数EC的作用域：

b函数EC.VO -> a函数EC.VO -> 全局EC.VO

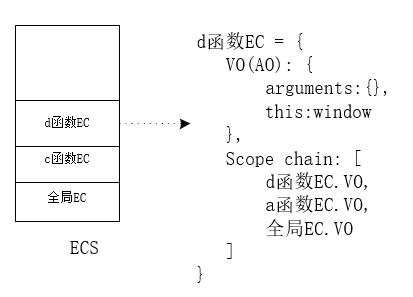
8. 接下去开始执行函数b的内部代码：console.log(num++);

由于使用到num变量，开始从作用域链中寻找，首先在 b函数EC.VO 中寻找，没找到；接着到下个作用域节点a函数EC.VO中寻找，发现存在 num 这个变量，所以b函数内使用的num变量是来自于a函数内部，而这个变量的取值在上述介绍的第7步时已经被赋值为 1 了，所以这里输出1。

同时，它还对 num 进行累加1操作，所以当这行代码执行结束，a函数EC.VO中的num变量已经被赋值为2了。

9. b函数执行结束，将b函数EC移出ECS栈，继续执行栈顶a函数的代码：console.log(sum);，看看当前的ECS情况：

所以这里需要使用sum变量，同样去作用域链中寻找，首先在 a函数EC.VO 中并没有找到，继续去全局EC.VO中寻找，发现sum变量取值为-1，所以这里输出-1.

10. a函数也执行结束，将a函数EC移出ECS栈，继续执行c函数内的代码：d()，由于a函数将函数b作为返回值，所以d() 执行是调用的 b 函数。此时：

这里又为d函数创建了执行上下文，所以到执行阶段执行代码：console.log(num++); 用到的 num 变量沿着作用域链寻找，最后发现是在a函数EC.VO 中找到，且此时 num 的值为第8步结束后的值 2，这里就输出 2.

到这里你可能会疑惑，此时ECS栈内，a函数EC不是被移出掉了吗，为何d函数创建EC的作用域链中还包括了a函数EC。

这里就涉及到闭包的概念了，留待下节闭包讲解。

总结

如果要从原理角度理解：

* 变量的作用域机制依赖于执行上下文，全局代码对应全局执行上下文，函数代码对应函数执行上下文
* 每调用一次函数，会创建一次函数执行上下文，这过程中，会解析函数代码，创建活动对象AO，将函数内声明的变量、形参、arguments、this、函数自身引用都添加到AO中
* 函数内对各变量的操作实际上是对上个步骤添加到AO对象内的这些属性的操作
* 创建执行上下文阶段中，还会创建上下文的另一个属性：作用域链。对于函数执行上下文，其值为当前上下文的VO拼接上当前函数的内部属性[[Scope]]，对于全局执行上下文，其值为上下文的VO。
* 函数内部属性[[Scope]]存储着它外层函数的作用域链，是在外层函数创建函数对象时，从外层函数的执行上下文的作用域链复制过来的值。
* 总之，JavaScript中的变量之所以可以在定义后被使用，是因为定义的这些变量都被添加到当前执行上下文EC的变量对象VO中了，而之所以有全局和函数内两种作用域，是因为当前执行上下文EC的作用域链属性的支持。也可以说一切都依赖于执行上下文机制。

那么，如果想通俗的理解：

* 函数内操作的变量，如果在其内部没定义，那么在其外层函数内寻找，还没有，往外层的外层函数内寻找，直到外层是全局对象为止。
* 这里的外层函数，指的是针对于函数声明位置的外层函数，而不是函数调用位置的外层函数。作用域链只与函数声明的位置有关系。

## 闭包

在作用域链那篇中，稍微留了个闭包的念想，那么这篇就来讲讲什么是闭包。

概念

这个闭包的概念蛮不好理解的，我在阮一峰的某篇文章中看过大概这么句话，闭包是对英文单词的直译，在中文里没有与之对应的句子解释，因此很难理解闭包究竟指的是什么。

看过很多解释，有说闭包就是函数；也有说闭包就是代码块；还有说函数内的函数就称闭包；还有说当函数返回内部某个函数时，返回的这个函数叫闭包，也有说闭包就是能够读取其他函数内部数据（变量/函数）的函数。

MDN网站里不同文章里出现过多种解释：

1. 一个闭包是一个可以自己拥有独立的环境与变量的表达式（通常是函数）

2. 闭包是函数和声明该函数的词法环境的组合，这个环境包含了这个闭包创建时所能访问的所有局部变量。

2009年发布了ECMAScript-262-5th第五版，不同的是取消了变量对象和活动对象等概念，引入了词法环境（Lexical Environments）、环境记录（EnviromentRecord）等新的概念

所以如果对词法环境这个词不理解的，可以将其理解成执行上下文，或者作用域链。

<https://github.com/goddyZhao/Translation/tree/master/JavaScript>

这里有几篇很早很早之前大佬们翻译的国外文章，它对闭包的解释刚好和MDN的解释很类似，它们都是基于闭包的特性上来解释的：

闭包是代码块和创建该代码块的上下文中数据的结合。

如果这个代码块是函数，那么利用作用域链那篇中的原理：

函数代码，和函数的内部属性[[Scope]]两者的结合可称为闭包。

闭包意义

先看个例子：

**var *num*** = 0;  
**function** *a*() {  
 **var** num = 1;  
 **function** *b*() {  
 ***console***.log(num);  
 }  
 **return** *b*;  
}  
**var *c*** = *a*();  
***c***();

调用 c() 输出的是 1，这点在作用域链那节已经讲解过了，这里再稍微说下。

调用 c()，会为c函数创建一个函数执行上下文，其中作用域链为：c函数EC.VO –> a函数EC.VO -> 全局EC.VO。

VO是变量对象，表示存储着当前上下文中所有变量的对象，所以如果以VO的实际对象表示作用域链：

c函数{} –> a函数{num:1} -> 全局{num:0}

（忽略VO中其他与此例无关变量）

所以，函数c内的代码输出num时，到作用域链上寻找时，发现最后使用的是a 函数内部的num 变量，最终输出1。

但当时也提了个疑问，当代码执行到 c() 时，a函数已经执行结束，那么a函数的EC已经从执行环境栈ECS中被移出了，c函数的EC里的作用域链为何还会有a函数EC.VO存在？

这就是闭包的典型场景了，闭包的意义之一就是解决这种场景。

通过作用域链一篇后，我们知道，函数内的变量依赖于函数执行上下文EC，一般来说，当调用函数时，创建函数执行上下文EC，并入栈ECS，当函数执行结束时，就将EC从ECS中移出，并释放内存空间。

通常函数的行为的确是这样，但当函数如果有返回值时，情况就不一样了。虽然函数执行结束后它的EC确实被移出ECS，但并没有被回收，JavaScript解释器的垃圾回收机制也有引用计数的处理。

既然内存没被回收，那么EC就还存在，那么当调用c() 时，虽然C的函数执行上下文是新创建的，上下文的作用域链也是新创建的，但作用域链的取值是当前执行上下文的VO拼接上函数对象的内部属性[[Scope]]。

这个函数对象的内部属性[[Scope]]存储的就是这个函数的外层函数的执行上下文里的作用域链，它的值并不是新创建的，一直保存着外层函数调用时生成的外层函数上下文中的作用域链。通过它可以访问到外层函数变量。

再谈闭包概念

所以，实际上，网络上这么多文章里对闭包的各种解释，其实都没错。如果对作用域链的原理理解清楚后，你会发现，其实函数就是闭包，因为由于作用域的机制，让函数内部也持有创建函数的上下文的数据集合，所以函数符合闭包的特性。

只是在大部分场景下，函数执行结束，函数的EC就可以被回收，那么这种场景闭包并没有什么实际应用意义。

除了函数，如果你可以让某部分代码块持有创建它的上下文的数据集合，那么这也可以称为闭包。

常见的一种就是在函数内返回一个对象，对象的某些属性使用了对象外层的数据，如：

**var *model*** = (**function** () {  
 **var** num = 1;  
 **return** {  
 **num**:num  
 }  
}());  
***model***.num;

此时，也可以称返回的这个对象是闭包。

对于闭包，我对它的理解，更倾向于，闭包并不是一种机制，也不是一种具体的事物（如执行上下文），反而，闭包是对原本存在的事物满足某种场景下的一种称呼。

也就是说，闭包，它其实是在原有机制，原有事物上的另一种称呼。所以，网上也才有人会说，闭包是函数、闭包是内嵌的函数等等说法。其实，也不是说这是错的，他们有的是从闭包特性角度解释，有的是从闭包现象。

只是，这原本就存在的事物，你本可以就用它原本的称呼，既然想要闭包来称呼它，那么自然是这个时候，称呼它为闭包有区别于原本事物的实际意义，所以也才有人会说当函数返回内部函数时，称为闭包，因为这种时候，返回的这个函数就是用到闭包的特性来解决某些问题，所以称这种现象为闭包当然就有实际应用场景意义了。

所以，我对闭包的理解，它并不是某个固定不变的东西，也不是某个具体的事物，只要符合闭包特性的原有事物，你都可以称它为闭包。所以，对于网上那些对闭包的解释，我的建议是，主谓互换一下，不要说闭包是函数，闭包是内嵌的函数等等，我们可以说，函数是闭包，内嵌的函数也是闭包。只要符合闭包特性的我们都可以称它为闭包，当然如果还有闭包的实际应用意义，那么称它为闭包更可以被人接受。

闭包的应用

* 作为外部和函数内部变量通信的桥梁

**var *model*** = (**function** () {  
 **var** num = 1;  
 **function** *a*() {  
 ***console***.log(num);  
 }  
 **return** {  
 **num**:num  
 }  
}());  
***model***.num;

外部是访问不了函数内部的信息，而闭包是指代码块持有创建它的上下文的数据集合。那么，如果在函数内部创建一个闭包，将这个闭包返回给外部，外部是否就可以通过这个闭包作为桥梁来间接与函数内部通信了。

* 封装

**var *Counter*** = (**function**() {  
 **var** privateCounter = 0;  
 **function** *changeBy*(val) {  
 privateCounter += val;  
 }  
 **return** {  
 increment: **function**() {  
 *changeBy*(1);  
 },  
 decrement: **function**() {  
 *changeBy*(-1);  
 },  
 value: **function**() {  
 **return** privateCounter;  
 }  
 }  
})();

还是同样的原因，外部是访问不了函数内部的信息，而闭包是指代码块持有创建它的上下文的数据集合。

那么，是否就可以借助闭包的特性，将一些实现封装在函数内部，通过闭包给外部提供有限的接口使用。

但要注意，函数本来执行结束，它的EC从ECS栈内移出时，通常就可被回收了，但如果用到了闭包的特性，导致外部持有着函数内部某个引用，此时函数的EC就不会被回收，那么就会占用着内存，使用不当，还会有可能造成内存泄漏。

## 异步回调执行的时机

本篇会讲到一个单线程事件循环机制，但并不是网络上对于 js 执行引擎介绍中的单线程机制，也没有涉及宿主环境浏览器的各种线程，如渲染线程、js 引擎执行线程、后台线程等等这些内容。

严谨来讲，应该不属于 JavaScript 自身的单线程机制，而是宿主对象，如浏览器处理执行 js 代码的单线程事件循环机制。

回到正题，本篇所要讲的，就是**类比于 Android 中的主线程消息队列循环机制，来讲讲在 JavaScript 中，如果设置了某个异步任务后，当异步任务执行完成需要回调通知时，这个回调任务的执行时机**。

如果还不清楚要讲的是什么，那么先来看个问题：

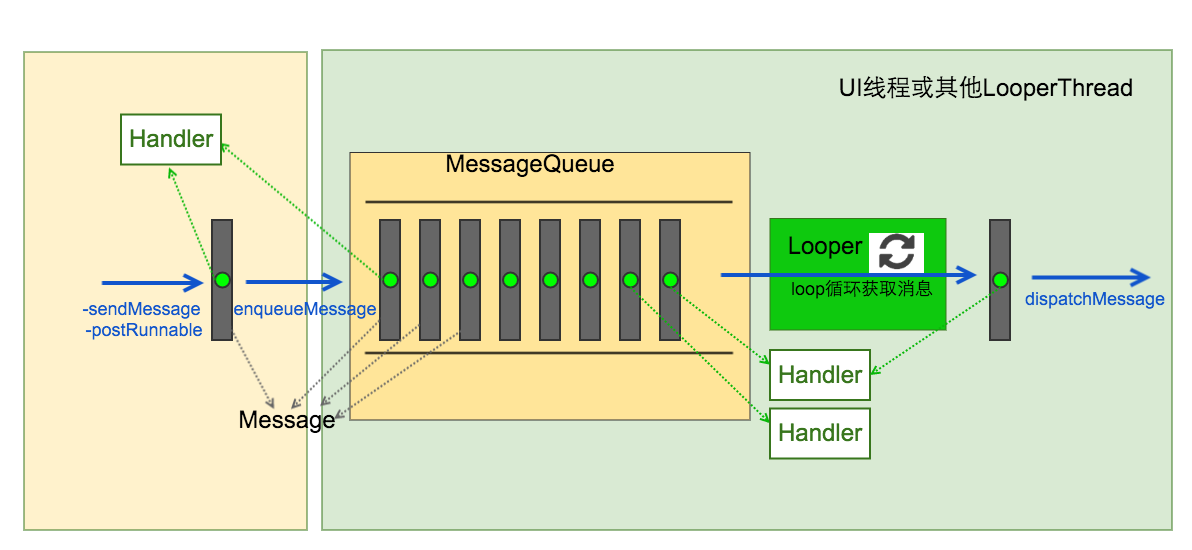
<script type="text/javascript">  
    $.ajax({  
        url: "https://easy-mock.com/mock/5b592c01e4e04f38c7a55958/ywb/is/version/checkVersion",  
        data: {"key": 122},  
        type: "POST",  
        success: function (data) {  
            console.log("----------success-----------");  //什么时候会执行回调  
       },  
        error: function (e) {  
            console.log("----------error-----------");  
       }  
   });  
​  
 //...  
</script>

这是用 jQuery 写的 ajax 网络请求的示例，这条请求自然是异步进行的，但当请求结果回来后，会去触发 success 或 error 回调，那么，问题来了：

**Q：想过没有，如果请求结果回来后，这个回调的代码是在什么时机会被执行的？是立马就执行吗，不管当前是否正在执行某个函数内的代码？还是等当前的函数执行结束？又或者是？**

也许你还没看懂这个问题要问的是什么，没关系，下面举例分析时，会讲得更细，到时你就知道这个问题要问的是什么了。

Android 消息队列循环机制

先来看看 Android 中的主线程消息队列循环机制，当然如果你不是从 Android 转前端，那可以跳过这趴：

这张图来自 [Android消息机制（一）：概述设计架构](https://www.jianshu.com/p/8656bebc27cb) 这篇文章中，我懒得自己画了，借大佬图片一用，如果不允许使用，麻烦告知下，我再来自己画。

在 Android 里有个主线程，因为只能在主线程中进行 UI 操作，所以也叫 UI 线程，这个主线程在应用启动时就进入一个死循环中，类似于执行了 while(true){...} 这样的代码，等到应用退出时，退出该死循环。而死循环之所以不会卡死 CPU，是因为利用了 Linux 的 epoll 机制，通俗的来将，就是，主线程会一直循环往消息队列中取消息执行，如果队列中没有消息，那么会进入阻塞状态，等有新的消息到来时，唤醒继续处理。而阻塞和唤醒就是利用了 Linux 的 epoll 机制。

所以，在 Android 中，打开页面是一个 message，触摸屏幕也是一个 message，message 中指示着当前应该执行的代码段，只有当前的 message 执行结束后，下会轮到下个 message 执行。

所以，在 Android 中的异步任务的回调工作，比如同样异步发起一个网络请求，请求结果回来后，需要回调到主线程中处理，那么这个回调工作的代码段会被封装到 message 中，发送到消息队列中排队，直到轮到它来执行。

而 message 发送到消息队列是基于 Handler 来传输，所以，在 Android 中，如果想要查看 message 是以什么为粒度，查找在哪里通过 Handler 发送了 message 即可。

JavaScript 中的单线程事件循环机制

那么，在 JavaScript 中，又是如何处理异步工作的回调任务的呢？

查了一些相关的资料，发现讲的都是 JavaScript 的单线程，事件循环机制等之类理论，但却没看到，事件的粒度是什么？

看完我能理解，JavaScript 也是类似 Android，一样执行了某段类似 while(true){...} 的代码来循环处理事件，但看完我仍旧无法理解，这个事件的粒度是什么，怎么查看事件的粒度？

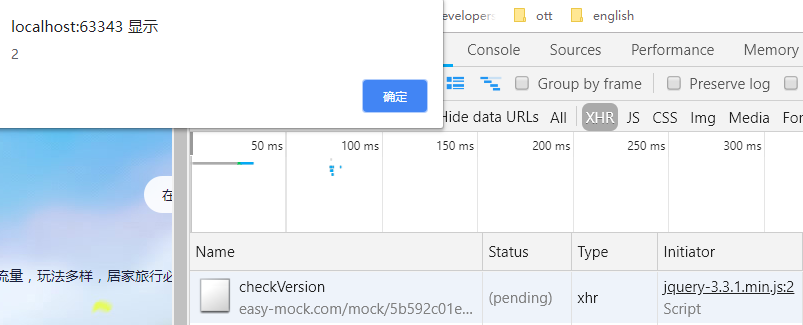
再举个例子来说明我的疑问好了：

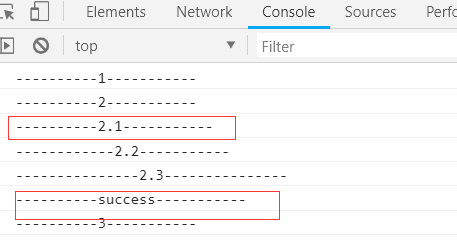
<script type="text/javascript">  
    console.log("----------1-----------");  
    $.ajax({  
        url: "https://easy-mock.com/mock/5b592c01e4e04f38c7a55958/ywb/is/version/checkVersion",  
        data: {"key": 122},  
        type: "POST",  
        success: function (data) {  
            console.log("----------success-----------");  //什么时候会执行回调  
       },  
        error: function (e) {  
            console.log("----------error-----------");  
       }  
   });  
​  
    console.log("----------2-----------");  
    alert("2");  //第一个卡点  
    console.log("----------2.1-----------");  
    function A() {  
        console.log("------------2.2-----------");  
        alert("A");  //第二个卡点  
   }  
    A();  
    console.log("---------------2.3---------------")  
</script>  
​  
<script type="text/javascript">  
    console.log("----------3-----------");  
</script>

alert() 会阻塞当前程序，当 js 执行到 alert() 的代码时卡在这里，后续代码不会被执行，直到取消弹窗。所以，我们可以通过注释上例中相对应的 alert() 来模拟异步请求的结果在什么时候接收到，而这个回调任务又是在哪个时机被执行的。

好，那么疑问来了：

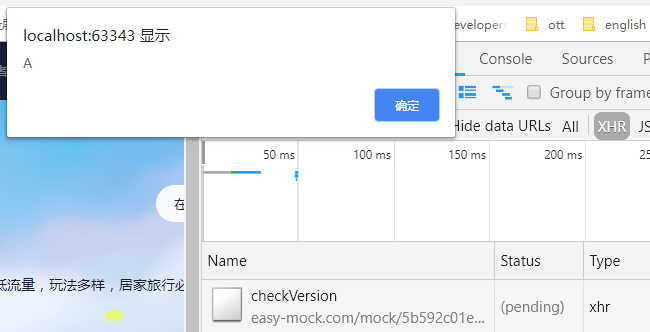
假设，程序卡在 alert("2") 这里，这时候，异步的请求结果回来了，那么回调任务是会被接到哪个时机执行？等我取消 alert 的弹窗后就先执行回调任务然后再继续处理 alert("2") 后的代码吗？

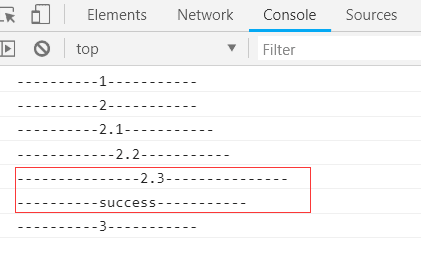
我们将 alert("A") 注释掉，运行一下，测试看看：

当前程序确实卡在 alert("2")，而且我们等到请求结果回来了，这时，我们把 alert 弹窗取消掉，看看日志：

回调任务中输出的 success 在 alert("2") 后续代码输出的 2.1 下面，那么就是先继续执行 alert("2") 后面的代码，然后才会执行回调任务的代码了，那么这个后面的代码究竟包括哪些代码？

好，这个时候，我们把 alert("2") 代码注释掉，让程序卡在 alert("A") 这行代码。

假设，当前程序正在执行某个函数内的代码，这个时候异步请求的结果回来了，那么这个回调任务会接在这个函数执行结束后吗？也就是，我们现在来验证下事件的粒度是否是以函数为粒度？

程序确实卡在函数 A 内部的代码 alert("A")，输出的日志上也能看到现在已经输出到 2.2，且异步请求的结果也回来了，那么这个回调任务的代码会在函数调用执行结束后，就被处理吗？如果是的话，那么日志 2.2 接下去应该要输出 success 才对，如果不是，那么就会输出 2.3，看看日志：

也就是说，即使异步请求结果回来了，回调任务也不能在当前函数执行完后立马被处理，它还是得继续等待，等到函数后面的代码也执行完了，那这个后面的代码到底是什么呢？也就是事件的粒度到底是什么呢？

我们试过了以每行代码为粒度做测试，也试过了以函数为粒度做测试，那还能以什么作为粒度呢？或者是以 <script> 为粒度，只有等当前 <script> 标签内的代码都执行完，才轮到下个代码段执行？

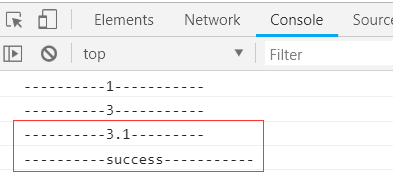
从上面两种场景下，所得到的日志来看，似乎确实也是这么个结论，success 的日志都是在 2.3 和 3 之间输出，2.3 表示当前 <script> 标签里的最后一行代码，而 3 表示下个 <script> 标签内的第一行代码。

既然这样，我们再来做个测试：

<script type="text/javascript">  
    console.log("----------1-----------");  
    $.ajax({  
        url: "https://easy-mock.com/mock/5b592c01e4e04f38c7a55958/ywb/is/version/checkVersion",  
        data: {"key": 122},  
        type: "POST",  
        success: function (data) {  
            console.log("----------success-----------");  //什么时候会执行回调  
       },  
        error: function (e) {  
            console.log("----------error-----------");  
       }  
   });  
    /\*  
   console.log("----------2-----------");  
   alert("2"); //第一个卡点  
   console.log("----------2.1-----------");  
   function A() {  
       console.log("------------2.2-----------");  
       alert("A"); //第二个卡点  
   }  
   A();  
   console.log("---------------2.3---------------") \*/  
</script>  
​  
<script type="text/javascript">  
    console.log("----------3-----------");  
    alert("3"); //第三个卡点  
    console.log("----------3.1---------")  
</script>

我们把第一个 <script> 标签内那些用于上面两种场景测试的代码注释掉，只留一个异步请求的代码，然后在第二个 <script> 标签内，加个 alert("3") 来模拟程序是在第一个 <script> 中发起异步请求，但直到程序运行到第二个 <script> 时，异步请求结果才回来，这种场景下回调任务的执行时机会是在哪？

如果当程序卡在 alert("3")，异步请求结果回来了，这时候还没有取消 alert 弹窗，或者一取消的时候，就先输出 success，再输出 3.1，则表示，回调任务的代码块是被安排到发起异步请求的这个 <script> 里代码都执行结束就去处理。

如果 success 是在 3.1 之后才输出，那么，就可以说明，浏览器处理 js 代码，是以 <script> 作为事件粒度，放入事件循环队列中去处理。看看日志：

好了，现在可以确认了，success 是在 3.1 之后才输出的，那么来整理下结论吧。

结论

之后问了一些前端同学，然后我基于对 Android 那边的类似理解，我自行梳理了下面的这些结论，因为涉及底层运行机制、浏览器行为的这些知识我还没开始去看，所以下面结论不保证正确，只能说是，基于我目前的能力，针对于做实验所得到的现象，我梳理出一些可以解释得通的结论。

* 浏览器解析 html 文档时，是按顺序一行一行进行解析，当处理到 <script> 标签时，会暂停当前页面的渲染，进入 js 代码的执行。
* 在执行当前 <script> 标签内的代码时，是以整个标签内的代码块作为事件粒度，放入事件队列中进行处理。
* 如果在当前 <script> 标签里的代码发起了某些异步工作，如异步网络请求，并设置了回调，那么回调任务的代码块会被单独作为一个事件，等到异步工作结束后，插入当前事件队列中。
* 所以，如果回调任务在执行当前 <script> 标签内的代码时就已经加入队列了，那么等到当前 <script> 里的代码都执行结束后，就可以轮到回调任务的执行。
* 如果回调任务直到当前 <script> 里的代码都执行结束也还没被加入事件队列，那么这时浏览器会接着去解析 html 文档，如果又碰到下个 <script> 标签，那么会将这个 <script> 标签内的代码块放入事件队列中处理。
* 所以，如果这时候第一个 <script> 标签内的代码发起的异步任务才结束，才将回调工作加入事件队列中，那么这个回调工作的代码只能等到第二个 <script> 标签内的代码都执行结束后才会被处理。

碰到的问题

为啥会想要梳理这个结论呢，是因为我碰到这么一种场景：

<script type="text/javascript">  
    document.location.href = "http://www.baidu.com"  
    //...  
</script>

之前有个 h5 项目中，有类似的代码，就是满足一定条件下，需要将页面跳转至其他页面。

修改 location.href 貌似不是同步操作，我猜测应该是这行跳转代码会告诉浏览器，当前页面准备跳转，这时候，浏览器再生成一个跳转事件，接入事件队列中等待执行的吧。

因为，最初我以为这是个同步操作，所以我认为当程序执行到 document.location.href = xx 这行代码之后，页面就会发生跳转，然后这行代码下面的那些代码都不会被执行，但最后实际运行时，却发现，这行代码下面的代码也都被执行了。

后来经过测试，发现，跳转语句这行代码所在的 <script> 里的代码会被全部执行完，然后才发起页面跳转，下个 <script> 里的代码不会被执行，所以，那个时候，就有个疑惑了，在 js 中发起一个异步操作的话，这个异步工作的回调任务的执行时机到底在哪里？

后来稍微查了相关资料，发现了个词说 JavaScript 是单线程机制，联想到 Android 中的主线程消息循环机制，这才想来理一理。

# JavaScript ES6

<http://es6.ruanyifeng.com/#README>

<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla>

S6 新标准规范相比于 ES5 旧标准规范中，无非就三个方面的改动：新增、更新、废弃。

由于更新和废弃需要考虑到兼容性问题，所以这两方面的内容应该并不多，那么大部分基本都是新增的特性。对于新增的特性来说，大体上也还可以再分两类：完全新增的特性和基于旧标准扩展的特性。

下面就大概来过一下，ES6 中新增的特性。

## Symbol

这是新增的一种原始数据类型，ES5 中原始类型有 5 种，在 ES6 中新引入了一种后，现在就是有 6 种原始数据类型了：Number、String、Boolean、null、undefined、Symbol

这个单词中文直译是：符号、标志等，但好像并没有在书中有这种叫法，书中都是直接使用 Symbol 来描述，可能中文翻译不能够很好的表示出这种原始数据类型的含义吧。

背景

之所以新增了这种原始数据类型，是为了解决：

ES5 的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。

基础用法

var s1 = Symbol('s1'); //参数只是用来描述当前 s1 变量，跟其他 Symbol 数据类型的变量相区分，在调用 toString 或 typeOf 时会输出当前数据类型跟描述信息  
​  
//第一种  
var o = {};  
o[s1] = "dasu";  
​  
//第二种  
var o = {  
   [s1]: "dasu"  
};  
​  
//第三种  
var o = {}  
Object.defineProperty(o, s1, {value: "dasu"});

给一个对象定义一个属性名用 Symbol 数据类型来表述的方法有上述三种，如果使用 o.s1 = "dasu" 这种方式，是给 o 对象定义了一个属性名为 s1 且数据类型为字符串的属性，字符串就存在相等与否的场景。而使用上述三种方式，是给对象定义了一个属性名为 s1 但数据类型为 Symbol 的属性，如果后期又定义了另一个属性名为 s1 的 Symbol 原始值的属性，并不会覆盖之前定义的属性。

其他内容，包括关于 Symbol 属性的遍历、Symbol 自带的方法、Symbol 应用场景等见：[Symbol](http://es6.ruanyifeng.com/#docs/symbol)

## 块作用域 let 和 const

背景

ES5 中变量的作用域只分全局作用域和函数内作用域，且全局变量本质上是全局对象的属性，书中是这么评价的：

顶层对象的属性与全局变量挂钩，被认为是 JavaScript 语言最大的设计败笔之一。这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，window对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。

为了解决这些问题，ES6 中新增了块级作用域的变量，通过关键字：let 和 const 声明的变量，作用域是块级作用域，同时，这些变量不属于顶层对象的属性。

const 定义的变量是常量，除此外，这个变量跟用 let 定义的变量没有其他方面的区别了。而通过 let 和 const 定义的变量行为、作用域类似于 Java 语言中定义的变量行为。

换句话说，let 和 const 定义的变量已经不包含 var 定义的变量的各种特性行为了，比如：没有变量的声明提前特性、存在暂时性死区（在定义之前不能使用，只能在定义变量位置之后使用）、不允许重复声明、作用域只有块级作用域等。

更多的参考：[let 和 const 命令](http://es6.ruanyifeng.com/#docs/let)

## Module 模块机制

背景

ES5 中并没有模块机制，常见的方式是：前端里通过 <script> 加载各种不同的 js 文件代码，在 js 文件代码内部中提供一些全局变量或全局函数供其他 <script> 使用。

说白了，就是不同 js 文件通过全局对象作为通信的桥梁来相互访问。而为了解决不同 js 文件共享全局对象造成的变量冲突问题，通常作为模块的 js 里的代码都会放在一个立即执行的函数内。

而 ES6 中，引入了模块的机制。

基本使用

当在 HTML 文档中，通过指定 <script> 标签的 type 属性为 module 时，如：

<script type="module" src="./foo.js"></script>

浏览器会按照模块的处理方式来加载这份 js 文件，与模块脚本的处理方式与正常 js 脚本文件处理方式最不同的地方在于，模块内的代码都是在模块作用域中执行，也就是在模块 js 文件中的全局变量这些并不会被添加到全局对象的属性中，其他 js 文件无法访问。

那么，其他 js 文件如何使用这份模块 js 文件呢？

有三点要求：

1. 模块 js 文件内，需要通过 export 声明该模块对外开放的接口
2. 当前 js 文件内，需要通过 import 来引入模块对外开放的接口，想使用哪些接口，就需要引入哪些
3. 当前 js 也必须是模块脚本文件，即 type=module 才行

import 命令和 export 命令是模块机制的关键，但只两个命令只能在 type=module 的 js 文件内生效，如果某个 js 文件声明了 type=text/javascript，然后代码里又使用到 import 或 export，那么运行期间会报错。

* 对外提供功能的模块

//module.js  
//...  
//以上省略模块内部功能代码  
​  
//一般对外的接口可放在文件底部，方便一览无余  
//第一种：  
export var name = "dasu";  
export function getName() {return "dasuAndroidTv"};  
​  
//第二种：  
var name = "dasu"  
var getName = function() {return "dasuAndroidTv"};  
export {name, getName}

* 使用其他模块功能的当前脚本

//第一种：from后跟js的相对路径或绝对路径  
import {name, getName} from 'module.js'  
​  
//第二种：  
import \* as Module from 'module.js'  
//使用  
Module.name;  
Module.getName();  
//...

最后，记住，模块脚本文件中，自动以严格模式运行，限制也很多，更多用法、细节说明参考：

[module 的语法](http://es6.ruanyifeng.com/#docs/module)

[module 的加载实现](http://es6.ruanyifeng.com/#docs/module-loader)

## 变量解构

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

var a = 1;  
var b = 2;  
var c = 3;

ES6 允许写成下面这样。

var [a, b, c] = [1, 2, 3];

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。

这种特性可以带来很多便捷：

* 函数可以返回多个值

// 返回一个数组  
function example() {  
  return [1, 2, 3];  
}  
let [a, b, c] = example();  
​  
// 返回一个对象  
function example() {  
  return {  
    foo: 1,  
    bar: 2  
 };  
}  
let { foo, bar } = example();

* 对象取值时很方便

//解析服务端返回的数据对象，提取其中的各字段值  
var o = {  
    code: 1,  
 msg: "success",  
    content: {...}  
};  
​  
var {code, msg, content} = o;

解构时，还可以设置默认值，更多用法，参考：[变量的解构赋值](http://es6.ruanyifeng.com/#docs/destructuring)

## 字符串的扩展

ES6 中对字符串的处理扩展了很多新特性，让字符串的处理更加强大，下面看一个很强大的特性：

模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的（下面使用了 jQuery 的方法）。

$('#result').append(  
  'There are <b>' + basket.count + '</b> ' +  
  'items in your basket, ' +  
  '<em>' + basket.onSale +  
  '</em> are on sale!'  
);

上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

$('#result').append(`  
  There are <b>${basket.count}</b> items  
   in your basket, <em>${basket.onSale}</em>  
  are on sale!  
`);

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

更多字符串扩展的特性介绍参见：[字符串的扩展](http://es6.ruanyifeng.com/#docs/string)

## 函数的扩展

ES6 中，对于函数的处理也增加了很多新特性，让函数变得更强大。

参数默认值

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

function log(x, y) {  
    x = x || 'dasu';  
 y = y || 'Android';  
 console.log(x, y);  
}

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

function log(x="dasu", y="Android") {  
 console.log(x, y);  
}

当然，参数默认值还有更多细节，比如默认参数的作用域是单独的、有默认参数值的函数 length 属性含义会变化等等。

剩余参数 reset

ES6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

function add(...values) {  
  let sum = 0;  
​  
  for (var val of values) {  
    sum += val;  
 }  
​  
  return sum;  
}  
​  
add(2, 5, 3) // 10

## 箭头函数

ES6 允许使用“箭头”（=>）定义函数。

var f = v => v;  
​  
// 等同于  
var f = function (v) {  
  return v;  
};

箭头前面是参数，后面的函数体，如果超过 1 个参数，那么用圆括号将多个参数圈起，如果没有参数，就用一个空圆括号 () 表示，如果函数体超过 1 行语句，那么用 {} 大括号包住。

var f = () => 5;  
// 等同于  
var f = function () { return 5 };  
​  
var sum = (num1, num2) => num1 + num2;  
// 等同于  
var sum = function(num1, num2) {  
  return num1 + num2;  
};

当函数作为另一个函数的参数时，箭头函数的写法会让代码变得很便捷，如：

* 箭头函数的一个用处是简化回调函数。

// 正常函数写法  
[1,2,3].map(function (x) {  
  return x \* x;  
});  
​  
// 箭头函数写法  
[1,2,3].map(x => x \* x);

* 嵌套的箭头函数

箭头函数内部，还可以再使用箭头函数。下面是一个 ES5 语法的多重嵌套函数。

function insert(value) {  
  return {into: function (array) {  
    return {after: function (afterValue) {  
      array.splice(array.indexOf(afterValue) + 1, 0, value);  
      return array;  
   }};  
 }};  
}  
​  
insert(2).into([1, 3]).after(1); //[1, 2, 3]

上面这个函数，可以使用箭头函数改写。

let insert = (value) => ({into: (array) => ({after: (afterValue) => {  
  array.splice(array.indexOf(afterValue) + 1, 0, value);  
  return array;  
}})});  
​  
insert(2).into([1, 3]).after(1); //[1, 2, 3]

但使用箭头函数有一些注意点：

* 函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象。
* 不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误。
* 不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
* 不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。

其他还有很多扩展，包括在 ES5 中，函数有个 name 属性并在标准规范中，但在 ES6 中加入了标准规范中，还有其他新增的一些特性，具体参考：[函数的扩展](http://es6.ruanyifeng.com/#docs/function)

## class 类

JavaScript 语言中，生成实例对象的传统方法是通过构造函数。下面是一个例子。

function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
​  
Point.prototype.toString = function () {  
  return '(' + this.x + ', ' + this.y + ')';  
};  
​  
var p = new Point(1, 2);

上面这种写法跟传统的面向对象语言（比如 C++ 和 Java）差异很大，很容易让新学习这门语言的程序员感到困惑。

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。

基本上，ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用 ES6 的class改写，就是下面这样。

//定义类  
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
 }  
​  
  toString() {  
    return '(' + this.x + ', ' + this.y + ')';  
 }  
}

上面代码定义了一个“类”，可以看到里面有一个constructor方法，这就是构造方法，而this关键字则代表实例对象。也就是说，ES5 的构造函数Point，对应 ES6 的Point类的构造方法。

Point类除了构造方法，还定义了一个toString方法。注意，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。

ES6 的类，完全可以看作构造函数的另一种写法。

class Point {  
  // ...  
}  
​  
typeof Point // "function"  
Point === Point.prototype.constructor // true

上面代码表明，类的数据类型就是函数，类本身就指向构造函数。

使用的时候，也是直接对类使用new命令，跟构造函数的用法完全一致。

class Bar {  
  doStuff() {  
    console.log('stuff');  
 }  
}  
​  
var b = new Bar();  
b.doStuff() // "stuff"

构造函数的prototype属性，在 ES6 的“类”上面继续存在。事实上，类的所有方法都定义在类的prototype属性上面。

class Point {  
  constructor() {  
    // ...  
 }  
​  
  toString() {  
    // ...  
 }  
​  
  toValue() {  
    // ...  
 }  
}  
​  
// 等同于  
​  
Point.prototype = {  
  constructor() {},  
  toString() {},  
  toValue() {},  
};

在类的实例上面调用方法，其实就是调用原型上的方法。

class B {}  
let b = new B();  
​  
b.constructor === B.prototype.constructor // true

更多细节内容参考：

[class 的基本语法](http://es6.ruanyifeng.com/#docs/class)

[class 的继承](http://es6.ruanyifeng.com/#docs/class-extends)

## 其他

阮一峰好歹专门为 ES6 新增的特性写了一整本书，想用一个章节来介绍太不可能了，很多我也没有去细看。

本篇的主旨就在于大体上列出一些新特性，知道原来 ES6 新增了这些东西，后续有时间再去细看这本书，或者当用到的时候再去查。

最后，就列举下，其他上面没有讲到的特性吧：

* ArrayBuffer：大概同于操作二进制数据
* set 和 map：新增是数据结构，类似于数组，可以类比 Java 中的 Set 和 Map
* Proxy：修改某些默认行为，等同于在语言层面做出修改，所以属于"元编程"，即对编程语言进行编程。
* Reflect：同样是用于操作对象，修改默认行为。Proxy 更多是在原本行为上增加新的行为，而 Reflect 则是直接修改原本行为。
* Iterator 和 for..of 循环：为不同的数据结构提供各自的遍历访问操作
* Promise：异步编程的解决方案，比传统的解决方案（回调和事件）更合理强大
* Generator 函数：异步编程的解决方案
* async：Generator 函数的语法糖
* 正则的扩展、数组的扩展、数值的扩展、对象的扩展等

# JavaScript模块化

## 模块化历程

在声明部分中的第二、第三链接里那两篇，以时间线介绍了模块化的相关技术的发展历程，我觉得很有必要一看，对于掌握和理解目前的模块化技术挺有帮助的。

这里，就稍微摘抄其中一些内容，详细内容还是需要到原文阅读。

1. **全局变量、全局函数（1999年）**

这时候的多个 js 脚本文件之间，直接通过全局变量或全局函数的方式进行通信，这种方式叫做：直接定义依赖。

虽然做的好一些的会对这些 js 文件做一些目录规划，将资源归类整理，但仍无法解决全局命名空间被大量污染，极其容易导致变量冲突问题。

1. **对象作为命名空间（2002年）**

为了解决遍地的全局变量问题，这时候提出一种命名空间模式的思路，即将本要定义成全局变量、全局函数的这些全都作为一个对象的属性存在，这个对象的角色充当命名空间，不同模块的 JS 文件中通过访问这个对象的属性来进行通信。

1. **立即执行的函数作为临时命名空间 + 闭包（2003年）**

虽然提出利用一个对象来作为命名空间的思路，一定程度解决了大量的全局变量的问题，但仍旧存在很多局限，比如没有模块的隐藏性，所以针对这些问题，这时候又新提出一种思路：利用立即执行的函数来作为临时命名空间，这样就可以避免污染全局命名空间，同时，结合闭包特性，来实现隐藏细节，只对外暴露指定接口。

虽然这种思路，解决了很多问题，但仍旧有一些局限，比如，缺乏管理者，什么意思，就是说，在前端里，开发人员得手动将不同 JS 脚本文件按照它们之间的依赖关系，以被依赖在前面的顺序来手动书写 <script> 加载这些文件。

也就是不同 <script> 的前后顺序实际上表示着它们之间的依赖关系，一旦文件过多，将会很难维护，这是上述方案都存在的问题。

1. **动态创建 <script> 工具（2009年）**

针对上述问题，也就衍生出了一些加载 js 文件的工具，先看个例子：

$LAB.script("greeting.js").wait()  
   .script("x.js")  
   .script("y.js").wait()  
   .script("run.js");

LAB.js 这类加载工具的原理实际上是动态的创建 <script>，达到作为不同 JS 脚本文件的管理者作用，来决定 JS 文件的加载和执行顺序。

虽然我没用过这种工具，但我觉得它的局限还是有很多，其实就是将开发人员本来需要手动书写在 HTML 文档里的 <script> 代码换成写在 JS 文件中，不同 JS 文件之间的依赖关系仍旧需要按照前后顺序手动维护。

1. **CommonJS规范&node.js（2009年）**

中间跳过了一些过程，比如 YUI 的沙箱模式等，因为不熟，想了解更详细的可以去原文阅读。

当 CommonJS 规范出来时，模块化算是进入了真正的革命，因为在此之前的探索，都是基于语言层面的优化，也就是利用函数特性、对象特性等来在运行期间模拟出模块的作用，而从这个时候起，模块化的探索就大量的使用了预编译。

CommonJS 模块化规范有如下特点：

* 所有代码都运行在模块作用域，不会污染全局作用域。
* 模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
* 模块加载的顺序，按照其在代码中出现的顺序。

不同的模块间的依赖关系通过 require 来控制，而每个模块需要对外暴露的接口通过 exports 来决定。

由于 CommonJS 规范本身就只是为了服务端的 node.js 而考虑的，node.js 实现了 CommonJS 规范，所以运行在 node.js 环境中的 js 代码可以使用 require 和 exports 这两个命令，但在前端里，浏览器的 js 执行引擎并不认识 require 这些命令，所以需要进行一次转换工作，后续介绍。

再来看看 require 命令的工作原理：

require 命令是 CommonJS 规范之中，用来加载其他模块的命令。它其实不是一个全局命令，而是指向当前模块的 module.require 命令，而后者又调用 Node 的内部命令 Module.\_load。

Module.\_load = function(request, parent, isMain) {  
  // 1. 检查 Module.\_cache，是否缓存之中有指定模块  
  // 2. 如果缓存之中没有，就创建一个新的Module实例  
  // 3. 将它保存到缓存  
  // 4. 使用 module.load() 加载指定的模块文件，  
  //   读取文件内容之后，使用 module.compile() 执行文件代码  
  // 5. 如果加载/解析过程报错，就从缓存删除该模块  
  // 6. 返回该模块的 module.exports  
};  
​  
Module.prototype.\_compile = function(content, filename) {  
  // 1. 生成一个require函数，指向module.require  
  // 2. 加载其他辅助方法到require  
  // 3. 将文件内容放到一个函数之中，该函数可调用 require  
  // 4. 执行该函数  
};

所以，其实 CommonJS 的模块化规范之所以可以实现控制作用域、模块依赖、模块通信，其实本质上就是将模块内的代码都放在一个函数内来执行，这过程中会创建一个对象 Module，然后将模块的相关信息包括对外的接口信息都添加到对象 Module 中，供其他模块使用。

1. [**AMD规范**](https://github.com/amdjs/amdjs-api)**&**[**Require.js**](https://requirejs.org/)**（2009年）**

CommonJS 规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。由于 Node.js 主要用于服务器编程，模块文件一般都已经存在于本地硬盘，所以加载起来比较快，不用考虑非同步加载的方式，所以CommonJS 规范比较适用。

但是，如果是浏览器环境，这种同步加载文件的模式就会导致浏览器陷入卡死状态，因为网络原因是不可控的。所以，针对浏览器环境的模块化，新提出了一种规范：AMD（Asynchronous Modules Definition）异步模块定义。

也就是说，对于 Node.js，对于服务端而言，模块化规范就是按照 CommonJS 规范即可。

但对于浏览器，对于前端而言，CommonJS 不适用，需要看看 AMD 规范。

AMD 规范中定义：

* 定义一个模块时通过 define 命令，通过 return 声明模块对外暴露的接口
* 依赖其他模块时，通过 require 命令

而规范终归只是规范，使用时还是需要有规范的具体实现，针对 AMD 规范，具体的实现是 Require.js，在前端里，如果基于 Require.js 来使用 AMD 规范的模块化技术，后续介绍。

1. [**CMD规范**](https://github.com/cmdjs/specification)**&**[**Sea.js**](https://seajs.github.io/seajs/docs/)**（2013年）**

CMD（Common Module Definition）也是专门针对浏览器、针对前端而提出的一种模块化规范。它跟 AMD 规范都是用途都是一样，用途解决前端里的模块化技术。

但两种规范各有各的优缺点，各有各的适用场景和局限性吧，我还没使用过这两种规范，所以无从比较，但网上关于这两种规范比较的文章倒是不少。

CMD 规范中定义：

* 使用 define 命令定义一个模块，使用 exports 来暴露模块对外的接口
* 使用 require 来同步加载依赖的模块，但也可使用 require.async 来异步加载依赖的模块

总之，虽然两种规范都是用于解决前端里的模块化技术，但实现的本质上还是有些不同，后续介绍。

对于 CMD 规范的具体实现是 Sea.js，前端里如果想使用 CMD 规范的模块化技术，需要借助 Sea.js。

1. **ES6标准（2015年）**

2015 年发布的 ES6 标准规范中，新增了 Module 特性，也就是官方在 ES6 中，在语言本身层面上，添加了模块的机制支持，让 JavaScript 语言本身终于可以支持模块特性了。

在 ES6 之前的各种方案，包括 CommonJS，AMD，CMD，本质上其实都是利用函数具有的本地变量的特性进行封装从而模拟出模块机制。也就是说，这些方案都是需要在运行期间，才会动态的创建出某个模块，并暴露模块的相关接口。这种方案其实也存在一些局限性。

而 ES6 新增了模块的机制后，在代码的解析阶段，就能够确定模块以及模块对外的接口，而不用等到运行期。这种本质上的区别，在借助开发工具写代码阶段的影响很明显就是，终于可以让开发工具智能的提示依赖的模块都对外提供了哪些接口。

但 ES6 由于是新增的特性，在支持方面目前好像还不是很理想，并不是所有环境都支持 ES6 的模块机制好像，所以会看到某些大佬的文章里会介绍一些诸如：Babel、Browserify。

Babel 用于将 ES6 转换为 ES5 代码，好让不支持 ES6 特性的环境可以运行 ES5 代码。

Browserify 用于将编译打包 js，处理 require 这些浏览器并不认识的命令。

上面只是简单介绍了下模块化的发展历程，而且中间略过一些阶段，想看详细的可以去原文阅读。下面就具体介绍下，目前四个比较稳定的模块技术：

## CommonJS规范

由于 CommonJS 是针对服务端设计的模块化规范，对于 Node.js 来说，一个 JS 文件就是一个模块，所以并不需要类似 AMD 或 CMD 中的 define 来声明一个模块。这是我的理解。

exports

既然在 Node.js 中，每个 JS 文件就是一个模块，那么这个模块文件内的变量都是对外隐藏的，外部无权访问，只能访问 exports 对外暴露的接口，如：

//module.js  
var name = "dasu";  
var wx = "dasuAndroidTv";  
var getBlog = function() {  
    return "http://www.cnblogs.com/dasusu/";  
}  
//以上变量和函数，外部都无权访问,外部只能访问到下面通过 exports 暴露的接口  
module.exports.name = name;  
exports.getWx = () => wx;

模块内，可以通过对 module.exports 或 exports 添加属性来达到对外暴露指定接口的目的，当然，从程序上来说，你可以直接改变 module.exports 的指向，让它指向一个新对象而不是在原本对象上添加属性，这个就类似于对构造函数 prototype 属性的修改。

但建议使用方式还是尽可能在 exports 对象上添加属性。

如果有想探究它的原理的话，可以尝试利用 Browserify 来转换这段模块代码，看看最后生成的是什么：

function(require,module,exports){  
    //module.js  
    var name = "dasu";  
    var wx = "dasuAndroidTv";  
    var getBlog = function() {  
        return "http://www.cnblogs.com/dasusu/";  
   }  
    //以上变量和函数，外部都无权访问,外部只能访问到下面通过 exports 暴露的接口  
    module.exports.name = name;  
    exports.getWx = () => wx;  
}

虽然，对于 Node.js 来说，它其实对待每个 JS 文件，本质上，会将文件内的代码都放于一个函数内，如果该模块首次被其他模块引用了，那么就会去执行这个函数，也就是执行模块内的代码，由于函数本身有三个参数，其中有两个分别是：module 和 exports，这也是内部为什么可以直接通过 module.exports 或 exports 来操作的原因。

Module 对象是 Node.js 会为每个模块文件创建的一个对象，模块之间的通信，其实就是通过访问每个 Module 对象的属性来实现。

所以，说白了，CommonJS 模块化技术的本质，其实就是利用了函数的局部作用域的特性来实现模块作用域，然后结合一个对象作为命名空间的方式，来保存模块内部需要对外暴露的信息方式。最后，通过 require 命令来组织各模块之间的依赖关系，解决以前方案没有管理者角色的局限，解决谁先加载谁后加载的问题。

require

每个 JS 文件其实都被当做一个模块处理，也就是文件内的代码都会被放入到一个函数内，那这个函数什么时候执行呢？也就是说，模块什么时候应该被加载呢？

这就是由 require 命令决定，当某个模块内使用了 require 命令去加载其他模块，那么这时候，被加载的模块如果是首次被调用，它是没有对应的 Module 对象的，所以会去调用它的函数，执行模块内代码，这个过程是同步的，这期间会完善这个模块的 Module 对象信息。之后，其他模块如果也引用了这个模块，因为模块内代码已经被执行过了，已经存在对应的 Module 对象，所以此时就不会再重复去加载这个模块了，直接返回 Module 对象。

以上，基本也就是模块间依赖的加载处理过程。而 require 命令用法很简单：

//main.js  
var module1 = require("./module");  
​  
module1.name;  //输出=> dasu  
module1.getWx(); //输出 => dasuAndroidTv  
//module1.getBlog(); //没有权限访问

其实，Node.js 对 main.js 的处理也是认为它是个模块，所以文件内的代码也都放入一个函数内，还记得函数的第一个参数就是 require 么，这也就是为什么模块内可以直接使用 require() 的原因，require 其实本质上是一个函数，具体的实现是 Node.js 的一个内置方法：Module.\_load()，主要工作上一节有介绍过了。

说得稍微严谨点，Node.js 其实才是作为各模块之间的管理者，由它来管控着哪个模块先加载，哪个后加载，维护着各模块对外暴露的信息。

到这里再来理解，有些文章中对 Module 对象的介绍：

* module.id 模块的识别符，通常是带有绝对路径的模块文件名。
* module.filename 模块的文件名，带有绝对路径。
* module.loaded 返回一个布尔值，表示模块是否已经完成加载。
* module.parent 返回一个对象，表示调用该模块的模块。
* module.children 返回一个数组，表示该模块要用到的其他模块。
* module.exports 表示模块对外输出的值。

这时，对于 Module 对象内的各属性用途，理解应该会比较容易点了。

最后说一点，CommonJS 只是一种模块化的规范，而 Node.js 才是这个规范的具体实现者，但 Node.js 通常用于服务端的运行环境，对于前端而言，对于浏览器而言，因为不存在 Node.js 这东西，所以 require 或 exports 这类在前端里是无法运行的，但可以借助 Browerify 来进行代码转换。

## AMD规范

[AMD规范](https://github.com/amdjs/amdjs-api)和规范实现者：[Require.js](https://requirejs.org/)

前端里没有 Node.js 的存在，即使有类似的存在，但由于 CommonJS 的模块化规范中，各模块的加载行为是同步的，也就是被依赖的模块必须执行结束，当前模块才能继续处理，这对于前端而言，模块的加载就多了一个下载的过程，而网络是不可靠的，所以 CommonJS 并不适用于前端的场景。

所以，针对前端，提出了另一种模块化规范：AMD，即异步模块加载，通过增加回调处理的方式，来移除被依赖模块和当前模块的前后关联，两个模块可同时下载，执行。当前模块内，需要依赖于其他模块信息的代码放置于回调函数内，这样就可以先行处理当前模块内其他代码。

define

前端里是通过 <script> 来加载 JS 文件代码的，不能像 Node.js 那样从源头上处理 JS 文件，所以它多了一个 define 来定义模块，如：

//module.js  
define(function(){  
    var name = "dasu";  
    var wx = "dasuAndroidTv";  
    var getBlog = function() {  
        return "http://www.cnblogs.com/dasusu/";  
   }  
      
    return {  
        name: name,  
        getWx: function() {  
            return wx;  
       }  
   }  
})

如果定义的模块又依赖了其他模块时，此时 define 需要接收两个参数，如：

//两个参数，第一个参数是数组，数组里是当前模块依赖的所有模块，第二个参数是函数，函数需要参数，参数个数跟数组个数一直，也跟数组里依赖的模块一一对应，该模块内部就是通过参数来访问依赖的模块。  
define(['module2'], function(module2){  
    //module2.xxx 使用模块 module2 提供的接口  
      
    //本模块的内部逻辑  
      
    return {  
        //对外暴露的接口  
   }  
})

define 有两个参数，第一个参数是数组，数组里是当前模块依赖的所有模块，第二个参数是函数，函数需要参数，参数个数跟数组个数一直，也跟数组里依赖的模块一一对应，该模块内部就是通过参数来访问依赖的模块。

require

如果其他地方有需要使用到某个模块提供的功能时，此时就需要通过 require 来声明依赖关系，但声明依赖前，需要先通过 requirejs.config 来配置各个模块的路径信息，方便 Require.js 能够获得正确路径自动去下载。

requirejs.config({  
    paths: {  
        module1: './module'  
   }  
})  
​  
var module1 = require(['module1'], function(module1){  
    console.log(module1.name);    //访问模块内的 name 接口  
    console.log(module1.getWx()); //访问模块内的 getWx 接口  
});  
​  
//其他不依赖于模块的代码

这种方式的话，require 的第一个数组参数内的值就可以模块的别称，而第二个参数是个函数，同样，函数的参数就是加载后的模块，该 JS 文件内需要依赖到模块信息的代码都可以放到回调函数中，通过回调函数的参数来访问依赖的模块信息。

使用步骤

1. 下载 Require.js，并放到项目中

[Require.js:https://requirejs.org/docs/download.html#requirejs](https://requirejs.org/docs/download.html#requirejs)

1. 新建作为模块的文件，如 module.js，并通过 define 定义模块

//module.js  
define(function(){  
    var name = "dasu";  
    var wx = "dasuAndroidTv";  
    var getBlog = function() {  
        return "http://www.cnblogs.com/dasusu/";  
   }  
      
    return {  
        name: name,  
        getWx: function() {  
            return wx;  
       }  
   }  
})

1. 在其他 js 文件内先配置所有的模块来源信息

//main.js  
requirejs.config({  
    paths: {  
        module1: './module'  
   }  
})

1. 配置完模块信息后，通过 require 声明需要依赖的模块

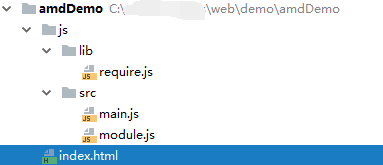
//main.js  
var module1 = require(['module1'], function(module1){  
    console.log(module1.name);    //访问模块内的 name 接口  
    console.log(module1.getWx()); //访问模块内的 getWx 接口  
});  
​  
//其他不依赖于模块的代码

1. 最后也最重要的一步，在 html 中声明 require.js 和 入口 js 如 main.js 的加载关系

<script src="js/lib/require.js" data-main="js/src/main.js"></script>

当然，这只是基础用法的步骤，其中第 3 步的模块初始化步骤也可通过其他方式，如直接利用 require 的不同参数类型来实现等等，但大体上需要这几个过程，尤其是最后一步，也是最重要一步，因为 AMD 在前端的具体实现都依靠于 Require.js，所以必须等到 Require.js 下载并执行结束后会开始处理其他 js。

以上例子的项目结构如图：



小结

最后小结一下，AMD 规范的具体实现 Require.js 其实从使用上来看，已经比较容易明白它的原理是什么了。

本质上，也还是利用了函数的特性，作为模块存在的那些代码本身已经通过 define 规范被定义在函数内了，所以模块内的代码自然对外是隐藏的，外部能访问到的只有函数内 return 的接口，那么这里其实也就利用了闭包的特性。

所以，模块化的实现，无非就是让函数作为临时命名空间结合闭包或者对象作为命名空间方式， 这种方式即使没有 CommonJS 规范，没有 AMD 规范，自己写代码很可以容易的实现。那么为什么会有这些规范技术的出现呢？

无非就是为了引入一个管理者的角色，没有管理者的角色，模块之间的依赖关系，哪个文件先加载，哪个后加载，<script> 的书写顺序都只能依靠人工来维护、管理。

而这些模块化规范，其实目的就在于解决这些问题，CommonJS 是由 Node.js 作为管理者角色，来维护、控制各模块的依赖关系，文件的加载顺序。而 AMD 则是由 Require.js 来作为管理者角色，开发者不用在 HTML 里写那么多 <script>，而且也没必要去关心这些文件谁写前谁写后，Require.js 会自动根据 require 来维护这些依赖关系，自动根据 requirejs.config 的配置信息去决定先加载哪个文件后加载哪个文件。

## CMD规范

CMD 规范，类似于 AMD，同样也是用于解决前端里的模块化技术问题。而有了 AMD 为什么还会有 CMD，我个人的理解是，AMD 的适用场景并没有覆盖整个前端里的需求，或者说 AMD 本身也有一些缺点，导致了新的 CMD 规范的出现。

比如说，从使用方式上，AMD 就有很多跟 CommonJS 规范不一致的地方，对于从 CommonJS 转过来的这部分人来说，可能就会很不习惯；再比如说，AMD 考虑的场景可能太多，又要适配 jQurey，又要适配 Node 等等；

总之，CMD 规范总有它出现和存在的理由，下面就大概来看看 CMD 的用法：

define&exports

类似于 AMD，CMD 规范中，也是通过 define 定义一个模块：

//module.js  
define(function (require, exports, module) {  
    var name = "dasu";  
    var wx = "dasuAndroidTv";  
    var getBlog = function() {  
        return "http://www.cnblogs.com/dasusu/";  
   }  
      
    exports.name = name;  
    exports.getWx = () => wx;  
})

跟 AMD 不一样的地方是，CMD 中 define 只接收一个参数，参数类型是一个函数，函数的参数也很重要，有三个，按顺序分别是 require，exports，module，作用就是 CommonJS 规范中三个命令的用途。

如果当前模块需要依赖其他模块，那么在内部，使用 require 命令即可，所以，函数的三个参数很重要。

当前模块如果不依赖其他模块，也没有对外提供任何接口，那么，函数可以没有参数，因为有了内部也没有使用。

而如果当前模块需要依赖其他模块，那么就需要使用到 require，所以函数第一个参数就是必须的；如果当前模块需要对外暴露接口，那么后两个参数也是需要的；

总之，建议使用 define 定义模块时，将函数三个参数都带上，用不用再说，规范一点总没错。

require

在有需要使用某个模块提供的功能时，通过 require 来声明依赖关系：

//main.js  
define(function (require, exports, module) {  
    console.log("=====main.js=======");  
​  
    var module1 = require("./module");//同步加载模块  
    console.log(module1.name);  
​  
    require.async("./module2", function (module2) {//异步加载模块  
        console.log(module2.wx);  
   })  
})

require 默认是以同步方式加载模块，如果需要异步加载，需要使用 require.async

使用步骤

1. 下载 Sea.js，并放到项目中

[Sea.js:https://github.com/seajs/seajs/releases](https://github.com/seajs/seajs/releases)

1. 新建作为模块的文件，如 module.js，并通过 define 定义模块

//module.js  
define(function (require, exports, module) {  
    var name = "dasu";  
    var wx = "dasuAndroidTv";  
    var getBlog = function() {  
        return "http://www.cnblogs.com/dasusu/";  
   }  
      
    exports.name = name;  
    exports.getWx = () => wx;  
})

1. 其他需要依赖到该模块的地方通过 require 声明

//main.js  
define(function (require, exports, module) {  
    console.log("=====main.js=======");  
​  
    var module1 = require("./module");//同步加载模块  
    console.log(module1.name);  
​  
    require.async("./module2", function (module2) {//异步加载模块  
        console.log(module2.wx);  
   })  
})

1. 最后也最重要的一步，在 html 中先加载 sea.js 并指定主模块的 js

<script src="js/lib/require.js"></script>  
<script>  
     seajs.use("./js/src/main.js");  
</script>

使用步骤跟 AMD 很类似，首先是需要依赖于 Sea.js，所以必须先下载它。

然后定义模块、依赖模块、使用模块的方式就跟 CommonJS 很类似，这几个操作跟 AMD 会有些不同，也许这点也正是 CMD 存在的原因之一。

最后一步也是最重要一步，需要在 HTML 文档中加载 sea.js 文档，并指定入口的 js，注意做的事虽然跟 AMD 一样，但实现方式不一样。

小结

其实，CMD 跟 CommonJS 很类似，甚至在模块化方面的工作，可以很通俗的将 sea.js 理解成 node.js 所做的事，只是有些 node.js 能完成但却无法通过 sea.js 来负责的工作需要开发人员手动处理，比如定义一个模块、通过 <script> 加载 sea.js 和指定主入口的 js 的工作。

## CommonJS, AMD, CMD 三者间区别

下面分别从适用场景、使用步骤、使用方式、特性等几个方面来对比这些不同的规范：

适用场景

CommonJS 用于服务端，基于 Node.js 的运行环境；

AMD 和 CMD 用于前端，基于浏览器的运行环境；

使用方式

CommonJS 通过 require 来依赖其他模块，通过 exports 来为当前模块暴露接口；

AMD 通过 define 来定义模块，通过 requirejs.config 来配置模块路径信息，通过 require 来依赖其他模块，通过 retrurn 来暴露模块接口；

CMD 通过 define 来定义模块，通过 require 来依赖其他模块，通过 exports 来为当前模块暴露接口；

使用步骤

CommonJS 适用的 Node.js 运行环境，无需其他步骤，正常使用模块技术即可；

AMD 适用的前端浏览器的运行环境没有 Require.js，所以项目中需要先加载 Require.js，然后再执行主入口的 js 代码，需要在 HTML 中使用类似如下命令：

<script src="js/lib/require.js" data-main="js/src/main.js"></script>

CMD 适用的前端浏览器的运行环境也没有 Sea.js，所以项目中也需要先加载 Sea.js，然后再执行主入口的 js 代码，需要在 HTML 中使用类似如下命令：

<script src="js/lib/sea.js"></script>  
<script>  
    seajs.use("./js/src/main.js");  
</script>

特性

AMD：依赖前置、提前执行，如：

require(['module1','module2'], function(m1, m2){  
    //...  
})  
define(['module1','module2'], function(m1, m2){  
    //...  
})

需要先将所有的依赖的加载完毕后，才会去处理回调中的代码，这叫依赖前置、提前执行；

CMD：依赖就近、延迟执行，如：

define(function(require, exports, module){  
    //...  
    var module1 = require("./module1");  
    //...  
    require("./module2", function(m2){  
        //...  
   });  
})

等代码执行到 require 这行代码时才去加载对应的模块

## ES6标准

ES6 中新增的模块特性，在上一篇中已经稍微介绍了点，这里也不具体展开介绍了，需要的话开头的声明部分有给出链接，自行查阅。

这里就简单说下，在前端浏览器中使用 ES6 模块特性的步骤：

1. 定义模块，通过指定 <script type="module"> 方式
2. 依赖其他模块使用 import，模块对外暴露接口时使用 export；

需要注意的一点是，当 JS 文件内出现 import 或者 export 时，这份 JS 文件必须声明为模块文件，即在 HTML 文档中通过指定 <script> 标签的 type 为 module，这样 import 或 export 才能够正常运行。

也就是说，使用其他模块的功能时，当前的 JS 文件也必须是模块。

另外，有一点，ES6 的模块新特性，所有作为模块的文件都需要开发人员手动去 HTML 文档中声明并加载，这是与其他方案不同的一点，ES6 中 import 只负责导入指定模块的接口而已，声明模块和加载模块都需要借助 <script> 实现。

这里不详细讲 ES6 的模块特性，但想来讲讲，一些转换工作的配置，因为：

* 有些浏览器不支持 ES6 的语法，写完 ES6 的代码后，需要通过 Babel 将 ES6 转化为 ES5。
* 生成了 ES5 之后，里面仍然有 require 语法，而浏览器并不认识 require 这个关键字。此时，可以用 Browserify 编译打包 js，进行再次转换。

而我是选择使用 WebStrom 作为开发工具的，所以就来讲讲如何配置

WebStrom 的 Babel 配置

教程部分摘抄自：[ES6的介绍和环境配置](https://github.com/smyhvae/Web/blob/master/10-ES6/03-ES6%E7%9A%84%E4%BB%8B%E7%BB%8D%E5%92%8C%E7%8E%AF%E5%A2%83%E9%85%8D%E7%BD%AE.md)

1. 新建项目
2. 通过 npm 初始化项目

在安装 Babel 之前，需要先用 npm 初始化我们的项目。打开终端或者通过 cmd 打开命令行工具，进入项目目录，输入如下命令： npm init -y，命令执行结束后，会在项目根目录下生成 package.json 文件

1. （首次）全局安装 Babel-cli

在终端里执行如下命令：

npm install -g babel-cli

1. 本地安装 babel-preset-es2015 和 babel-cli

npm install --save-dev babel-preset-es2015 babel-cli

本地是指在项目的根目录下打开终端执行以上命令，执行结束，项目根目录的 package.json 文件中会多了 devDependencies 选项

1. 新建 .babelrc 文件

在根目录下新建文件 .babelrc，输入如下内容：

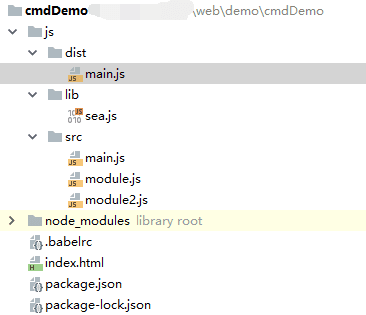
{  
    "presets":[  
        "es2015"  
   ],  
    "plugins":[]  
}

1. 执行命令转换

babel js/src/main.js -o js/dist/main.js

-o 前是原文件，后面是转换后的目标文件

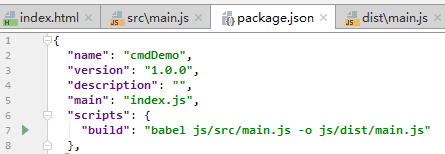
这是我的项目结构：



.json 文件和 node\_modules 文件夹都是操作完上述步骤后会自动生成的，最后执行完命令后，会在 dist 目录下生成目标文件。

1. （可选）如果嫌每次执行的命令太过复杂，可利用 npm 脚本

将 babel js/src/main.js -o js/dist/main.js 这行代码复制到 package.json 里的 scripts 字段中：



以后每次都点一下 build 左边的三角形按钮运行一下脚本就可以了，省去了手动输命令的时间。

1. （可选）如果还嫌每次手动点击按钮运行脚本麻烦，可配置监听文件改动自动进行转换

打开 WebStrom 的 Setting -> Tools -> File Watchers，然后点击 + 按钮，选择 Babel 选项，然后进行配置：



1. 最后，以后每次新的项目，除了第 3 步不用了之外，其余步骤仍旧需要进行。

WebStrom 的 Browserify 配置

步骤跟上述很类似，区别仅在于一个下载 babel，这里下载的是 browserify，以及转换的命令不同而已：

1. 新建项目
2. 通过 npm 初始化项目

打开终端，进入到项目的根目录，执行 npm init -y，执行结束后会在根目录生成 package.json 文件

1. （首次）全局安装 browserify

在终端里执行如下命令：

npm install browserify -g

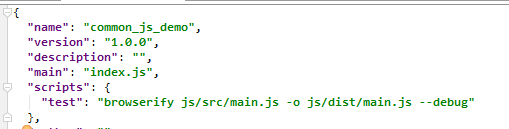
1. 执行命令转换

browserify js/src/main.js -o js/dist/main.js --debug

-o 前是原文件，后面是转换后的目标文件

1. （可选）如果嫌每次执行的命令太过复杂，可利用 npm 脚本

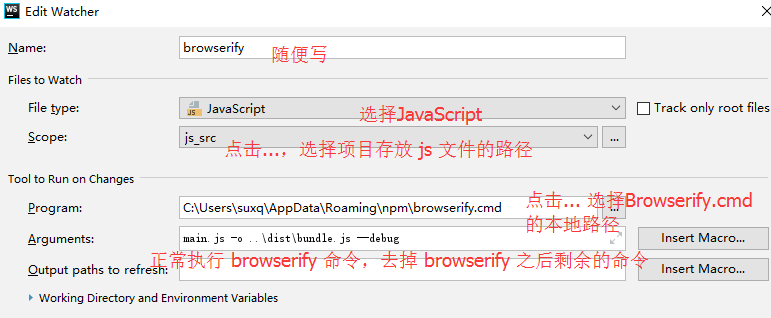
将 browserify js/src/main.js -o js/dist/main.js --debug 这行代码复制到 package.json 里的 scripts 字段中：



以后每次都点一下 build 左边的三角形按钮运行一下脚本就可以了，省去了手动输命令的时间。

1. （可选）如果还嫌每次手动点击按钮运行脚本麻烦，可配置监听文件改动自动进行转换

打开 WebStrom 的 Setting -> Tools -> File Watchers，然后点击 + 按钮，选择 <custom> 选项，然后进行配置：



1. 最后，以后每次新的项目，除了第 3 步不用了之外，其余步骤仍旧需要进行。

# TypeScript

## 为什么使用TypeScript

如果学习 JavaScript 之前已经有了 Java 的基础，那么学习过程中肯定会有很多不习惯的地方，因为 JavaScript 不管是在语法上面、还是编程思想上与 Java 这类语言都有一些差异。

下面就大概来看几个方面的差异：

变量声明

JavaScript 是弱语言，声明变量时无需指明变量的数据类型，运行期间会自动推断，所以声明方式很简单：

**var *a*** = 1;

Java 是强类型语言，声明变量时必须明确指出变量数据类型：

int ***a*** = 1;

弱类型语言虽然比较灵活，但也很容易出问题，而且需要一些额外的处理工作，比如函数期待接收数组类型的参数，但调用时却传入了字符串类型，此时 js 引擎并不会报错，对于它来说，这是合理的行为，但从程序、从功能角度来看，也许就不会按照预期的执行，所以通常需要在函数内部进行一些额外处理，如果没有额外处理，那么由于这种参数类型导致的问题也很难排查。

变量作用域

JavaScript 的变量在 ES5 只有全局作用域和函数内作用域，ES6 新增了块级作用域。

Java 的变量分类变量和实例变量，属于类的变量如果是公开权限，那么所有地方都允许访问，属于实例的变量，分成员变量和局部变量，成员变量在实例内部所有地方都可以访问，在实例外部如果是公开权限，可通过对象来访问，局部变量只具有块级作用域。

因为 JavaScript 在 ES5 时并没有块级作用域，有些场景下会导致变量被覆盖的情况，由于这种情况造成的问题也很难排查，比如：

**function** *aaa*() {  
 **var** i = -1;  
 **for** (**var** i = 0; i < 1; i++) {  
 **for** (**var** i = 1; i < 2; i++) {  
  
 }  
 ***console***.log(i);  
 }  
 ***console***.log(i);  
}

在 Java 中，两次 i 的输出应该 0, -1，因为三个地方的 i 变量并不是同一个，块级作用域内又生成一个新的局部 i 变量，但在 JavaScript 里，ES5 没有块级作用域，函数内三个 i 都是同一个变量。

而且，JavaScript 的全局变量会被作为全局对象的属性存在，而在 JavaScript 里对象的属性是允许动态添加的，这就会导致一个问题：当使用某变量，但拼写错误时，js 引擎并不会报错，对它来说，会认为新增了一个全局对象的属性；但从程序，从功能角度来看，常常就会导致预期外的行为，而这类问题也很难排查，比如：

**var** main = **"type-script"**;  
**function** modify(pre) {  
 mian = **`**${pre}**-script`**;  
}  
modify(123);

在 Java 里会找不到 mian 变量报错，但在 JavaScript 里 mian 会被当做全局对象的属性来处理。

而且，JavaScript 的变量允许重复申请，这样一来，全局变量一旦多了，很容易造成变量冲突问题，这类问题即使在运行期间也很难被发现和排查，比如：

*//a.js***var *a*** = 1;  
  
*//b.js***var *a*** = **"js"**;

在不同文件中，如果全局变量命名一样，会导致变量冲突，但浏览器不会有任何报错行为，因为对它来说，这是正常的行为，但对于程序来说，功能可能就会出现预期外的行为。

继承

JavaScript 是基于原型的继承，原型本质上也是对象，所以 JavaScript 中对象是从对象上继承的，同时对象也是由对象创建的，一切都是对象。

Java 中有 class 机制，对象的抽象模板概念，用于描述对象的属性和行为以及继承结构，而对象是从类实例化创建出来的。

正是因为 JavaScript 中并没有 class 机制，所以有 Java 基础的可能会比较难理解 JavaScript 中的继承、实例化对象等原理。

那么在面向对象的编程中，自定义了某个对象，并赋予它一定的属性和行为，这样的描述在 Java 里很容易实现，但在 JavaScript 里却需要通过定义构造函数，对构造函数的 prototype 操作等处理，语义不明确，不怎么好理解，比如定义 Dog 对象：

**function** *Dog*() {}  
Book.**prototype**.eat = **function** () {  
 *//...*}   
Book.**prototype**.**name** = **"dog"**;

Class 机制

JavaScript 虽然在 ES6 中加入了 class 写法，但本质上只是语法糖，而且从使用上，仍旧与 Java 的 class 机制有些区别，比如：

**class** Animal {  
 constructor(theName) {  
 **this**.name = theName;  
 **this**.ll = 23;  
 }  
 move(distanceInMeters = 0) {  
 console.log(**`**${**this**.name} **moved** ${distanceInMeters}**m.`**);  
 }  
}

以上是 JavaScript 中 ES6 自定义某个类的用法，与 Java 的写法有如下区别：

* 类的属性只能在构造函数内声明和初始化，无法像 Java 一样在构造函数外面先声明成员变量的存在；
* 无法定义静态变量或静态方法，即没有 static 语法；

权限控制

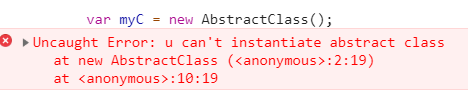
JavaScript 里没有 public 这些权限修饰符，对于对象的属性，只能通过控制它的可配置性、可写性、可枚举性来达到一些限制效果，对于对象，可通过控制对象的可扩展性来限制。

Java 里有 package 权限、publick 权限、protection 权限、private 权限之分，权限修饰符可修饰类、变量、方法，不同权限修饰符可以让被修饰的具有不一样的权限限制。

在 JavaScript 如果要实现对外部隐藏内部实现细节，大多时候，只能利用闭包来实现。

抽象类

JavaScript 虽然在 ES6 中引入了 class 的写法，但本质上只是语法糖，并没有类似 Java 中抽象类、抽象方法的机制存在，即使要模拟，也只能是定义一些抛异常的方法来模拟抽象方法，子类不实现的话，那么在运行期间就会抛异常，比如：

*//不允许使用该构造函数创建对象，来模拟抽象类***function** *AbstractClass*() {  
 **throw new *Error***(**"u can't instantiate abstract class"**);  
}  
*//没有实现的抽象方法，通过抛异常来模拟***function** *abstractMethod*() {  
 **throw new *Error***(**"abstract method,u should implement it"**);  
}  
*//定义抽象方法，子类继承之后，如果不自己实现，直接使用会抛异常  
AbstractClass*.prototype.onMearsure = *abstractMethod*;  
*AbstractClass*.prototype.onLayout = *abstractMethod*;

相比于 Java 的抽象类的机制，在编译期间就可以报错的行为，JavaScript 的运行期抛异常行为效果可能没法强制让所有开发者都能正确实现抽象方法。

对象标识

JavaScript 由于没有 class 机制，又是基于原型的继承，运行期间原型还可动态变化，导致了在 JavaScript 里没有一种完美的方式可以用来获取对象的标识，以达到区分不同对象的目的。

Java 中的对象都是从类实例化创建出来的，因此通过 instanceof 即可判断不同对象所属类别是否一致。

在 JavaScript 中，只能根据不同使用场景，选择 typeof、instanceof、isPrototypeOf()、对象的类属性、对象的构造函数名等方式来区别不同对象所属类别。

鸭式辩型

正是由于 JavaScript 里没有 class 机制，没有哪种方式可以完美适用所有需要区分对象的场景，因此在 JavaScript 中有一种编程理念：鸭式辩型（只要会游泳且嘎嘎叫的鸟，也可以认为它是鸭子）

意思就是说，编程中不要从判断对象是否是预期的类别角度出发，而是从判断对象是否具有预期的属性角度出发。

小结

所以，对于如果有 Java 基础的，JavaScript 学习过程可能会有些不习惯，那么如果是 TypeScript 的话，可以说是个福利，因为 TypeScript 很多语法和编程思想上都跟 Java 很类似，很容易就理解。

那么，来认识下，TypeScript 是什么？

TypeScript 是 JavaScript 的超集，超集是什么意思，就是说，JavaScript 程序可以不加修改就运行在 TypeScript 的环境中，TypeScript 在语法上是基于 JavaScript 进行扩展的。

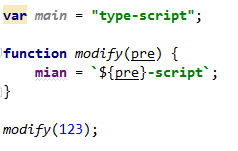
那么，TypeScript 在 JavaScript 语法基础上做了哪些扩展呢？其实就是加入了各种约束性的语法，比如加入了类似强类型语言的语法。

比如说，声明变量时，需要指定变量的数据类型的约束，以此来减少类型错误导致的问题。

**let *a***:**number** = 1;

其实，本质上是因为 JavaScript 是解释型语言，因为没有编译阶段，很多问题只能是运行期才可能被发现，而运行期暴露的问题也不是可以很好的排查出来。

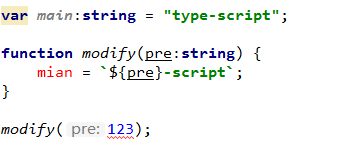
而 TypeScript 语法编写的 ts 文件代码，浏览器并不认识，所以需要经过一个编译阶段，编译成 js 文件，那么 TypeScript 就提供了一个编译过程，加上它语法上的支持，在编译期间编译器就可以帮助开发者找出一些可能出错的地方。

举个例子：

这个例子中，定义了一个全局变量和一个函数，函数本意是接收一个字符串类型的值，然后修改这个全局变量的值，但开发者可能由于粗心，将全局变量的变量名拼写错误了，而且调用方法时并没有传入字符串类型，而是数字类型。

如果是在 JavaScript 中，这段代码运行期间并不会报错，也不会导致程序异常，js 解释器会认为它是合理的，它会认为这个函数是用来增加全局对象的 mian 属性，同时函数参数它也不知道开发者希望使用的是什么类型，它所有类型都接受。

由于程序并没有出现异常，即使运行期间，开发者也很难发现这个拼写错误的问题，相反，程序由于拼写错误而没有执行预期的功能时，反而会让开发者花费很多时间来排查原因。

但这段代码如果是用 TypeScript 来写：

这些基础的语法错误，编译器甚至不用进入编译阶段，在开发者刚写完这些代码就能给出错误提示。而且，一些潜在的可能造成错误的代码，在编译阶段也会给出错误提示。

虽然 TypeScript 语法上支持了很多类似于 Java 语言的特性，比如强类型约束等，但 JavaScript 本质上并不支持，可以看看上面那段代码最后编译成的 js 代码：

**var** main = **"type-script"**;  
**function** modify(pre) {  
 mian = **`**${pre}**-script`**;  
}  
modify(123);

发现没有，编译后的代码其实也就是上述举例的 js 代码段，也就是说，你用 JavaScript 写和用 TypeScript 写，最后的代码都是一样的，区别在于，TypeScript 它有一个编译阶段，借助编译器可以在编译期就能发现可能的语法错误，不用等到运行期。

## WebStrom 配置

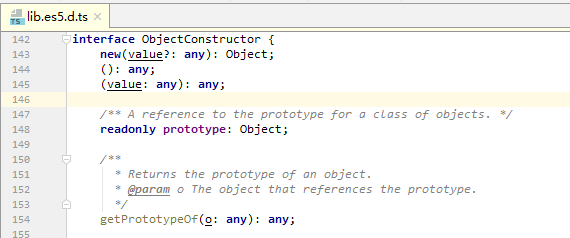
将 TypeScript 编写的 ts 文件编译成 js 文件有两种途径，一是借助命令，二是借助开发工具。

如果电脑已经有安装了 node.js 环境，那么可以直接执行下述命令：

> npm install -g typescript

然后打开终端，在命令行执行:

> tsc xxx.ts

我选择的开发工具是 WebStrom，这个开发工具本身就是支持 TypeScript 的了，如果你有尝试过查看 ES5、ES6 相关 api，你可能会发现：

.d.ts 文件就是用 TypeScript 编写的，所以如果你熟悉 TypeScript 的语法，这些代码就能很清楚了，.d.ts 是一份声明文件，作用类似于 C++ 中的 .h 文件。

在 WebStrom 中右键 -> 新建文件中，可以选择创建 TypeScript 的文件，可以设置 FileWatcher 来自动编译，也可以将项目初始化成 node.js 项目，利用 package.json 里的 scripts 脚本命令来手动触发编译。

我选择的是后者，如果你对 package.json 或 FileWatcher 配置不熟悉，可以参考之前模块化那篇最后对这些配置的介绍。

而编译器在编译过程，类似于 Android 里的 Gradle，可以设置很多配置项，进行不同的编译，而 TypeScript 编译过程对应的配置文件是 tsconfig.json

## tsconfig.json

<https://www.tslang.cn/docs/handbook/tsconfig-json.html>

* 不带任何输入文件的情况下调用tsc，编译器会从当前目录开始去查找tsconfig.json文件，逐级向上搜索父目录。
* 不带任何输入文件的情况下调用tsc，且使用命令行参数--project（或-p）指定一个包含tsconfig.json文件的目录。
* 当命令行上指定了输入文件时，tsconfig.json文件会被忽略。
* {  
   **"compilerOptions"**: {  
   **"module"**: **"commonjs"**, *//编译输出的 js 以哪种模块规范实现，有 commonjs,amd,umd,es2015等等* **"target"**: **"es5"**, *//编译输出的 js 以哪种 js 标准实现，有 es3,es5,es6,es2015,es2016,es2017,es2018等* **"sourceMap"**: **false**, *//编译时是否生成对应的 source map 文件* **"removeComments"**: **false**, *//编译输出的 js 文件删掉注释* **"outDir"**: **"./js/dist"** *//编译输出的 js 路径* },  
   **"exclude"**: [ *//编译时排除哪些文件* **"node\_modules"** ]  
  }

## 语法-数据类型

ES6 中的数据类型是：number，string，boolean，symbol，null，undefined，object

TypeScript 在此基础上，额外增加了：any，void，enum，never

any：表示当前这个变量可以被赋予任何数据类型使用；

void：表示当前这个变量只能被赋予 null 或 undefined，通常用于函数的返回值声明；

enum：枚举类型，弥补 JavaScript 中无枚举的数据类型；

never：表示永不存在的值，常用于死循环函数，抛异常函数等的返回值声明，因为这些函数永远也不会有一个返回值。

TypeScript 中的数据类型是用于类型声明服务的，类似于 Java 中定义变量或声明方法的返回值时必须指定一个类型。

## 语法-类型声明

ES5 中声明变量是通过 var，而 ES6 中引入块级作用域后新增了 let 和 const 的声明方式，TypeScript 建议声明变量时，都使用 let，因为 var 会很容易造成很多问题，不管是全局变量还是函数作用域的局部变量。

原始类型的声明：

**let *num***:**number** = 1;  
**let *str***:**string** = **"ts"**;  
**let *is***:**boolean** = **true**;  
**function** *f*(name: **string**, age: **number**):**void** {  
 *//...*  
}

声明一个变量时，就可以在变量名后面跟 : 冒号来声明变量的数据类型，如果赋值给变量声明的数据类型之外的类型，编译器会有错误提示；函数的返回值的类型声明方式类似。

如果某个变量的取值可以是任意类型的，那么可以声明为 any：

**let *variable***:**any** = 1;  
***variable*** = **true**;*//此时赋值其他类型都不会报错*

如果某个变量取值只能是某几个类型之间，可以用 | 声明允许的多个类型：

**let *numStr***:**number**|**string** = 1;  
***numStr*** = **"str"**;  
***numStr*** = **true**;*// 报错*

如果变量是个数组：

**let *numArr***:**number**[] = [1, 2]; *//纯数字数组，如果某个元素不是数字类型，会报错***let *anyArr***:**any**[] = [1, **"tr"**, **true**]; *//数组元素类型不限制***let *numStrArr***:(**number**|**string**)[] = [1, **"tr"**, 2, 4]; *// 数组元素类型限制在只能是 number 和 string*

如果变量是个对象：

**let *obj***:**object** = {};

但这通常没有什么意义，因为函数，数组，自定义对象都属于 object，所以可以更具体点，比如声明变量是个函数：

**let** *fun*:(a:**number**)=>**string** = **function** (a:**number**):**string** {  
 **return ""**;  
}

声明 fun 变量是一个函数类型时，还需要将函数的结构声明出来，也就是函数参数，参数类型，返回值类型，通过 ES6 的箭头函数语法来声明。

但赋值的时候，赋值的函数参数类型，返回值类型可以不显示声明，因为编译器可以根据函数体来自动推断，比如：

**let** *fun*:(a:**number**)=>**string** = **function** (a) {  
 **return ""**;  
}

如果变量是某个自定义的对象：

**class** Dog {  
 **name**:**string**;  
 **age**:**number** = 0;  
}  
  
**let *dog***:Dog = **new** Dog();

定义类的语法后面介绍，在 JavaScript 里，鸭式辩型的编程理念比较适用，也就说，判断某个对象是否归属于某个类时，并不是看这个对象是否是从这个类创建出来的，而是看这个对象是否具有类的特征，即类中声明的属性，对象是否拥有，有，则认为这个对象是属于这个类的。如：

**let *dog***:Dog = {**name**:**"dog"**, **age**:123}; *//可以赋值成功，因为对象直接量具有 Dog 类中的属性***let *dog1***:Dog = {**name**:**"dog"**, **age**:1, **sex**:**"male"**}; *//错误，多了个 sex***let *dog2***:Dog = {**name**:**"dog"**}; *//错误，少了个 age***let *dog3***:Dog = {**name**:**"dog"**, **age**:**"12"**}; *//错误，age 类型不一样*

以上例子中：

**let *dog1***:Dog = {**name**:**"dog"**, **age**:1, **sex**:**"male"**};

从鸭式辩型角度来说，这个应该是要可以赋值成功的，因为目标对象拥有类指定的特征行为了，TypeScript 觉得额外多出的属性可能会造成问题，所以会给一个错误提示。

针对这种因为额外多出的属性检查而报错的情况，如果想要绕开这个限制，有几种方法：

* 类型断言

**let *dog1***:Dog = <Dog>{**name**:**"dog"**, **age**:1, **sex**:**"male"**};  
**let *dog1***:Dog = {**name**:**"dog"**, **age**:1, **sex**:**"male"**} **as** Dog;

类型断言就是类似 Java 里的强制类型转换概念，通过 <> 尖括号或者 as 关键字，可以告诉编译器这个值的数据类型。

类型断言常用于开发者明确知道某个变量的数据类型的情况下。

* 用变量做中转赋值

如果赋值语句右侧是一个变量，而不是对象直接量的话，那么只会检查变量是否拥有赋值语句左侧所声明的类型的特征，而不会去检查变量额外多出来的属性，如：

**let *o*** = {**name**:**"dog"**, **age**:1, **sex**:**"male"**}; **let *dog1***:Dog = ***o***;

* 剩余属性

这种方式是最佳的方式，官网中对它的描述是字符串索引签名，但我觉得这个描述很难理解，而且看它实现的方式，有些类似于 ES6 中的函数的剩余参数的处理，所以我干脆自己给它描述成剩余属性的说法了。

方式是这样的，在类中定义一个用于存储其他没有声明的属性数组：

**class** Dog {  
 **name**:**string**;  
 **age**:**number** = 0;  
 [**propName**:**string**]:**any**;  
}

最后一行 [propName:string]:any，就表示，具有 Dog 特征的对象除了需要包含 name 和 age 属性外，还可以拥有其他任何类型的属性。所以：

**let *dog1***:Dog = {**name**:**"dog"**, **age**:1, **sex**:**"male"**, **s**:**true**};

这样就是被允许的了。

当然，这三种可以绕开多余属性的检查手段，应该适场景而使用，不能滥用，因为，大部分情况下，当 TypeScript 检查出你赋值的对象多了某个额外属性时，程序会因此而出问题的概念是比较大的。

鸭式辩型在 TypeScript 里更常用的是利用接口来实现，后续介绍。

## 语法-接口

鸭式辩型其实严格点来讲就是对具有结构的值进行类型检查，而具有结构的值也就是对象了，所以对对象的类型检查，其实也就是在对对象进行类别划分。

既然是类别划分，那么不同类别当然需要有个标识来相互区分，在 TypeScript 里，接口的作用之一也就是这个，作为不同对象类别划分的依据。

比如：

**interface** Dog {  
 **name**:**string**;  
 **age**:**number**;  
 **eat**:()=>**any**;  
}

上述就是定义了，对象如果拥有 name, age属性和 eat 行为，那么就可以将这个对象归类为 Dog，即使创建这个对象并没有从实现了 Dog 接口的类上实例化，如：

**let *dog***:Dog = {  
 **name**: **"小黑"**,  
 **age**:1,  
 eat: **function** () {  
   
 }  
}

上述代码声明了一个 Dog 类型的变量，那么什么对象才算是 Dog 类型，只要拥有 Dog 中声明的属性和行为就认为这个对象是 Dog，这就是鸭式辩型。（属性和行为是 Java 里面向对象常说的概念，属性对应变量，行为对应方法，在 JavaScript 里变量和方法都属于对象的属性，但既然 TypeScript 也有类似 Java 的接口和类机制，所以这里我习惯以 Java 那边的说法来描述了，反正能理解就行）

当然，也可以通过定义一个 Dog 类来作为变量的类型声明，但接口相比于类的好处在于，接口里只能有定义，一个接口里具有哪些属性和行为一目了然，而类中常常携带各种逻辑。

既然接口作用之一是用来定义对象的类别特征，那么，它还有很多其他的用法，比如：

**interface** Dog {  
 **name**:**string**;  
 **age**:**number**;  
 **eat**:()=>**any**;  
  
 **master**?:**string**; *//狗的主人属性，可有可无* **readonly variety**:**string**; *//狗的品种，一生下来就固定了*}  
  
**let *dog1***:Dog = {**name**:**"dog1"**, **age**:1, eat:()=>**""**, **variety**:**"柯基"**};  
***dog1***.**age** = 2;  
***dog1***.**variety** = **"中华犬"**;*//报错，variety声明时就被固定了，无法更改***let *dog2***:Dog = {**name**:**"dog2"**, **age**:1, eat:()=>**""**, **master**: **"me"**,**variety**:**"柯基"**};

在接口里声明属性时，可用 ? 问号表示该属性可有也可没有，可用 readonly 来表示该属性为只读属性，那么在定义时初始化后就不能再被赋值。

? 问号用来声明该项可有可无不仅可以用于在定义接口的属性时使用，还可以用于声明函数参数时使用。

在类型声明一节中说过，声明一个变量的类型时，也可以声明为函数类型，而函数本质上也是对象，所以，如果有需求是需要区分多个不同的函数是否属于同一个类别的函数时，也可以用接口来实现，如：

**interface** Func {  
 (name:**string**):**boolean**;  
}  
  
**let** *func*:Func = **function** (name) {  
 **return true**;  
}

这种使用接口的方式称为声明函数类型的接口，可以简单的理解为，为 Func 类型的变量定义了 () 运算符，需传入指定类型参数和返回指定类型的值。

如果想让某个类型既可以当做函数被调用，又可以作为对象，拥有某些属性行为，那么可以结合上述声明函数类型的接口方式和正常的接口定义属性行为方式一起使用。

当对象或函数作为函数参数时，通过接口来定义这些参数的类型，就特别有用，这样可以控制函数调用时传入了预期类型的数据，如果类型不一致时，编译阶段就会报错。

当然，接口除了用来在鸭式辩型中作为值类型的区分外，也可以像 Java 里的接口一样，定义一些行为规范，强制实现该接口的类的行为，如：

**interface** Dog {  
 **name**:**string**;  
 **age**:**number**;  
 **eat**:()=>**any**;  
  
 **master**?:**string**; *//狗的主人属性，可有可无* **readonly variety**:**string**; *//狗的品种，一生下来就固定了*}

**class** ChinaDog **implements** Dog{  
 **age**: **number** = 0;  
 **eat**: () => **any**;  
 **master**: **string**;  
 **name**: **string**;  
 **readonly variety**: **string**;  
  
}

ChinaDog 实现了 Dog 接口，那么就必须实现该接口所定义的属性行为，所以，ChinaDog 创建的对象明显就属于 Dog：

**let *dog3***:Dog = **new** ChinaDog();

除了这些基本用法外，TypeScript 的接口还有其他很多用法，比如，定义构造函数：

**interface** Dog {  
 **new** (name:**string**): Dog;  
}

再比如接口的继承：接口可继承自接口，也可继承自类，继承的时候，可同时继承多个等。

更多高级用法，等有具体的使用场景，碰到的时候再深入去学习，先了解到这程度吧。

## 语法-类机制

习惯 Java 代码后，首次接触 ES5 多多少少会很不适应，因为 ES5 中都是基于原型的继承，没有 class 概念，自定义个对象都是写构造函数，写 prototype。

后来 ES6 中新增了 class 语法糖，可以类似 Java 一样通过 class 自定义对象，但还是有很多区别，比如，ES6 中的 class 语法糖，就无法声明成员变量，成员变量只能在构造函数内定义和初始化；而且，也没有权限控制、也没有抽象方法机制、也不能定义静态变量等等。

然而，这一切问题，在 TypeScript 中都得到了解决，TypeScript 的 class 语法基本跟 Java 一样，有 Java 基础的，学习 TypeScript 的 class 语法会很轻松。

看个例子：

**abstract class** Animal {  
 **age**:**number**;  
 **protected abstract** eat():**void**;  
}  
  
**class** Dog **extends** Animal{  
 **public static readonly** *TAG*:**string** = **"Dog"**;  
 **public name**:**string**;  
 **private isDog**:**boolean** = **true**;  
  
 **constructor**(name:**string**) {  
 **super**();  
 **this**.**name** = name;  
 **this**.age = 0;  
 }  
  
 **protected** *eat*:()=>**any** = **function** () {  
 **this**.**isDog**;  
 }  
  
 **get** age():**number** { *//将 age 定义为存取器属性* **return this**.age;  
 }  
  
 **set** age(age:**number**) { *//将 age 定义为存取器属性* **if** (age > 0) {  
 **this**.age = age;  
 } **else** {  
 age = 0;  
 }  
 }  
}  
  
**let *dog***:Dog = **new** Dog(**"小黑"**);

大概有几个地方跟 Java 有些许小差别：

* 变量类型的声明
* 构造函数不是用类名表示，而是使用 constructor
* 如果有继承关系，则构造函数中必须要调用super
* 不手动使用权限修饰符，默认是 public 权限

其余方面，不管是权限的控制、继承的写法、成员变量的定义或初始化、抽象类的定义、基本上都跟 Java 的语法差不多。

所以说 TypeScript 的 class 语法比 ES6 的 class 语法糖要更强大。

还有很多细节的方面，比如在构造函数的参数前面加上权限修饰符，此时这个参数就会被当做成员变量来处理，可以节省掉赋值的操作；

比如在 TypeScript 里，类还可以当做接口来使用。

## 语法-泛型

Java 里在操作实体数据时，经常会需要用到泛型，但 JavaScript 本身并不支持泛型，不过 TypeScript 支持，比如：

**interface** Adapter<T> {  
 **data**:T;  
}  
  
**class** StringAdapter **implements** Adapter<**string**>{  
 **data**: **string**;  
}  
  
**function** *f1*<Y **extends** Animal>(arg:Y):Y {  
 **return**;  
}  
  
*f1*(**new** Dog(**"小黑"**));

用法基本跟 Java 类似，函数泛型、类泛型、泛型约束等。

## 语法-模块和命名空间

JavaScript 跟 Java 很不一样的一点就是，Java 有 class 机制，不同文件都需要有一个 public class，每个文件只是用于描述一个类的属性和行为，类中的变量不会影响其他文件内的变量，即使有同名类，只要类文件路径不一致即可。

但 JavaScript 所有的 js 文件都是运行在全局空间内，因此如果不在函数内定义的变量都属于全局变量，即使分散在多份不同文件内，这就很容易造成变量冲突。

所以也才有那么多模块化规范的技术。

虽然 TypeScript 的 class 语法很类似于 Java，但 TypeScript 最终仍旧是要转换成 JavaScript 语言的，因此即使用 TypeScript 来写 class，只要有出现同名类，那么即使在不同文件内，仍旧会造成变量冲突。

解决这个问题的话，TypeScript 也支持了模块化的语法。

而且，TypeScript 模块化语法有一个好处是，你只需掌握 TypeScript 的模块化语法即可，编译阶段可以根据配置转换成 commonJs, amd, cmd, es6 等不同模块化规范的实现。

模块

TypeScript 的语法跟 ES6 中的模块语法很类似，只要 ts 文件内出现 import 或 export，该文件就会被当做模块文件来处理，即整个文件内的代码都运行在模块作用域内，而不是全局空间内。

* 使用 export 暴露当前模块对外接口

*//module.ts***export interface** StringValidator {  
 isAcceptable(s: **string**): **boolean**;  
}  
  
**export const** numberRegexp = /^[0-9]+$/;  
  
**export class** ZipCodeValidator **implements** StringValidator {  
 isAcceptable(s: **string**) {  
 **return** s.**length** === 5 && numberRegexp.test(s);  
 }  
}

**class** AarCodeValidator **implements** StringValidator {  
 isAcceptable(s: **string**) {  
 *//...* }  
}  
**export** { AarCodeValidator };

export 的语法基本跟 ES6 中 export 的用法一样。

如果其他模块需要使用该模块的相关接口：

* 使用 import 依赖其他模块的接口

**import** { ZipCodeValidator } **from "./module"**;  
  
**let** myValidator = **new** ZipCodeValidator();

如果想描述非TypeScript编写的类库的类型，我们需要声明类库所暴露出的API。通常需要编写 .d.ts 声明文件，类似于C++ 中的 .h文件。

.d.ts 声明文件的编写，以及引用时需要用到三斜杠指令：

*/// <reference path="./m2.d.ts"/>*

这部分内容我还没理解清楚，后续碰到实际使用掌握后再来说说。

命名空间

命名空间与模块的区别在于，模块会涉及到 import 或 export，而命名空间纯粹就是当前 ts 文件内的代码不想运行在全局命名空间内，所以可以通过 命名空间的语法，让其运行在指定的命名空间内，防止污染全局变量。

语法：

**namespace** Validation {  
 *//...*}

## 其他

本篇只讲了 TypeScript 的一些基础语法，还有其他更多知识点，比如引入三方不是用 TypeScript 写的库时需要编写的 .d.ts 声明文件，比如编译配置文件的各种配置项，比如枚举，更多更多的内容，请参考：

<https://www.tslang.cn/docs/handbook/type-checking-javascript-files.html>

# Vue.js

组件的单向通信：props 结合 v-bind

反向通信：自定义事件，$emit() 触发，$on() 监听

语法糖双向通信：v-model，相当于封装了 v-bind:value 和 v-on:input，所以组件内需要使用 $on(‘input’) 来将信息通知给外部，如果组件内刚好有 input 标签，那么组件内需要手动声明 value 的 prop

以上都是父子间的通信，如果是跨级通信即爷孙，或者兄弟之间的通信：

v-bind 来动态绑定 class 或 style，:class，:style

# iVue

# Angular

angular cli 项目辅助工具：

ng new dirName 新创建项目

ng generate subDre 新建组件

{name | uppercase} 竖号右边可用于对左侧字符串、数字、日期等做格式化操作，可使用内置的格式化指令操作或自定义。

app.module.ts 中引入和声明各种依赖库、组件，包括内置和自定义的组件

组件里双向数据绑定：[(ngModule)] = “name”

ngModule 必须在 app.module.ts 中声明依赖 @angular/froms 才能正常使用

\*ngFor = “let i of list” 循环，类似 Vue 中的 v-for

\*ngIf = “表达式”，控制元素是否需要渲染，类似 Vue 中 v-if

（click）=”handleClick(x)”，监听点击事件，类似 Vue 中 @click，但括号不能省略，VUE 中可以省略

[class.xxx-class] = “a === b”，当 a === b 时，添加 xxx-class 的样式，否则移除

@input() variable: XXX; 用于定义组件接收的参数，外部使用时，可通过：

[variable]=”xxx” 来传入参数，类似于 Vue 中的 props 和 v-bind 用法

Angular 只会绑定到组件的公共属性。

组件应该只关注数据的呈现于交互，业务逻辑不应出现在组件中

所以，业务相关逻辑应该在服务中编写

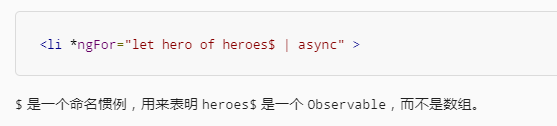
在组件的构造函数的参数中，声明需要依赖的服务，angular 会自动将相关服务创建单例，并注入到组件中。

组件构造函数应该只做属性初始化工作，调用服务的接口去触发业务逻辑应该放置于 ngOnInit 方法内。

服务要能够被其他组件注入，需要在其内部通过 @Injectable注册声明。

路由：ng generate module app-routing –flat –module=app

在路由中维护路由表

跳转交由 a 标签的 router-link 属性

deleteRequest = new [EventEmitter](https://angular.cn/api/core/EventEmitter)<Hero>();

自定义事件

双向绑定：

<app-sizer [(size)]="fontSizePx"></app-sizer>

<app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>

[()] 是语法糖，[(x)] 等效于 [x] 和 (xChange) 两个绑定

但由于原生的HTML元素不遵循 x 和 xChange 事件的模式，所以针对 input 或者 select 这些标签而已，可以使用 [(ngModel)]

你不能把 [([ngModel](https://www.angular.cn/api/forms/NgModel))] 用到非表单类的原生元素或第三方自定义组件上，除非写一个合适的**值访问器**，这种技巧超出了本章的范围。

有些场景也可以将[(ngModel)] 拆开写，如：

<input [[ngModel](https://www.angular.cn/api/forms/NgModel)]="currentHero.name" (ngModelChange)="setUppercaseName($event)">

如果需要对相关值做一些转换工作时。

<div \*[ngFor](https://angular.cn/api/common/NgForOf)="let hero of heroes; trackBy: trackByHeroes">

<input #phone placeholder="phone number"> <!-- lots of other elements --> <!-- phone refers to the input element; pass its `value` to an event handler --> <button (click)="callPhone(phone.value)">Call</button>

大多数情况下，Angular 会把模板引用变量的值设置为声明它的那个元素。 在上一个例子中，phone 引用的是表示**电话号码**的 <input> 框

[a](https://angular.cn/api/router/RouterLinkWithHref)?.b?.c?.d

安全导航符，防止因为访问 null 的属性而造成程序崩溃，等效于判空处理

<input #box (keyup)="0"> <p>{{box.value}}</p>

从一个模板引用变量中获得用户输入

1. <button (click)="timer.start()">Start</button>
2. <button (click)="timer.stop()">Stop</button>
3. <div class="seconds">{{timer.seconds}}</div>
4. <app-countdown-timer #timer></app-countdown-timer>
5. 父组件不能通过数据绑定使用子组件的 start 和 stop 方法，也不能访问子组件的 seconds 属性。
6. 把本地变量(#timer)放到(<countdown-timer>)标签中，用来代表子组件。这样父组件的模板就得到了子组件的引用，于是可以在父组件的模板中访问子组件的所有属性和方法。

以上只适用于父组件在模板中访问子组件，如果父组件需要在类中用代码访问子组件，那么需要使用@ViewChild()

首先，你要使用 [ViewChild](https://angular.cn/api/core/ViewChild) 装饰器导入这个引用，并挂上 [AfterViewInit](https://angular.cn/api/core/AfterViewInit) 生命周期钩子。

接着，通过 @[ViewChild](https://angular.cn/api/core/ViewChild) 属性装饰器，将子组件 CountdownTimerComponent 注入到私有属性 timerComponent 里面。

还有一种，通过注入同一个服务来实现父子组件的通信

插值表达式

<img src="{{heroImageUrl}}" [style](https://www.angular.cn/api/animations/style)="height:30px">

也可以用于标签属性中，或文本块中，上述等价于：

[src]=”heroImageUrl”

冒号内其实是一个表达式，但不是所有表达式都支持，以下是被禁止的：

赋值表达式： = += -+

new 运算符

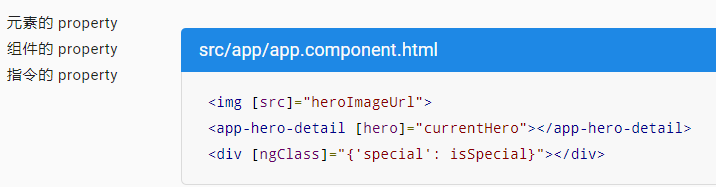
链式表达式

不支持 | & 位运算

局部变量：模板输入变量（let i of x）和模板引用变量（#i）

（click）=”模板语句”

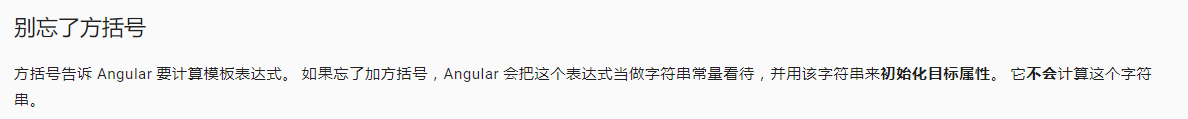
事件绑定跟 [] 区别在于，事件绑定中的是模板语句，模板语句相对于模板表达式，它支持 = 或 ; 运算。

所有的数据绑定的是 dom 的属性，而不是 HTML 元素的属性，两者有区别。比如 a 元素的 src 绑定的是 a 对应的 DOM 对象的 src 属性，而不是 a 标签的 src 属性，所有如果 DOM 对象没有相应的属性会绑定失败。

没有方括号的话，数据就只会传递一次，也应该只传递一次，后续不再变更检测更新。

插值表达式实际上会被转为属性绑定实现。

* 为什么要[给结构型指令的名字加上(\*)前缀？](https://www.angular.cn/guide/structural-directives#asterisk)
* 当没有合适的宿主元素放置指令时，可用 [<ng-container>](https://www.angular.cn/guide/structural-directives#ngcontainer) 对元素进行分组。
* 如何写自己的结构型指令。
* 你只能往一个元素上应用[一个结构型指令](https://www.angular.cn/guide/structural-directives#one-per-element)。



带 **trackBy** 的 **\***[**ngFor**](https://www.angular.cn/api/common/NgForOf)

[ngFor](https://www.angular.cn/api/common/NgForOf) 指令有时候会性能较差，特别是在大型列表中。 对一个条目的一丁点改动、移除或添加，都会导致级联的 DOM 操作。

例如，重新从服务器查询可以刷新包括所有新英雄在内的英雄列表。

他们中的绝大多数（如果不是所有的话）都是以前显示过的英雄。**你**知道这一点，是因为每个英雄的 id 没有变化。 但在 Angular 看来，它只是一个由新的对象引用构成的新列表， 它没有选择，只能清理旧列表、舍弃那些 DOM 元素，并且用新的 DOM 元素来重建一个新列表。

如果给它指定一个 trackBy，Angular 就可以避免这种折腾。 往组件中添加一个方法，它会返回 NgFor**应该**追踪的值。 在这里，这个值就是英雄的 id。

大多数情况下，Angular 会把模板引用变量的值设置为声明它的那个元素。 在上一个例子中，phone 引用的是表示**电话号码**的 <input> 框。 "拨号"按钮的点击事件处理器把这个 **input** 值传给了组件的 callPhone 方法。 不过，指令也可以修改这种行为，让这个值引用到别处，比如它自身。 [NgForm](https://www.angular.cn/api/forms/NgForm) 指令就是这么做的。

模板引用变量的作用范围是**整个模板**。 不要在同一个模板中多次定义同一个变量名，否则它在运行期间的值是无法确定的。

你总是可以在组件自己的模板中绑定到组件的公共属性，而不用管它们是否输入（Input）属性或输出（Output）属性。

这是因为组件类和模板是紧耦合的，它们是同一个东西的两个部分，合起来构成组件。 组件类及其模板之间的交互属于实现细节。

但要绑定到其他组件中的属性，就需要设置输入输出。

另外，还可以在指令元数据的 inputs 或 outputs 数组中标记出这些成员。比如这个例子：

src/app/hero-detail.component.ts

content\_copy@[Component](https://www.angular.cn/api/core/Component)({

inputs: ['hero'],

outputs: ['deleteRequest'],

})

输入输出可以有别名

非空断言操作符（!）

类型转换函数 $any （$any( <表达式> )）

ngOnChanges 只是组件的输入属性发生变化时才会被调用，输入属性是指当前组件给父组件使用时，由父组件传入的数据。当前组件内绑定到模板的那些属性发生变化并不会调用它。

输入属性除了使用 ngOnChanges 监听变化之外，还可以通过定义存取器的输入属性 set 来监听。两者区别在于，ngOnChanges 监听多个输入属性变化时比 set 方便。

ng-content 给组件投影使用，类似于 Vue 中的槽 slat，是自定义组件的关键技术。

ng-template 和 ng-container 和 ng-content 三者区别

在 @[Component](https://www.angular.cn/api/core/Component) 的元数据中指定的样式只会对该组件的模板生效。

它们既不会被模板中嵌入的组件继承，也不会被通过内容投影（如 ng-content）嵌进来的组件继承。

在这个例子中，h1 的样式只对 HeroAppComponent 生效，既不会作用于内嵌的 HeroMainComponent ，也不会作用于应用中其它任何地方的 <h1> 标签。

你可以在指令的构造函数中注入 [ElementRef](https://www.angular.cn/api/core/ElementRef)，来引用宿主 DOM 元素，组件也是指令

Angular 把组件的模板看做**从属于**该组件的。 组件和它的模板默认会相互信任。 这也就是意味着，组件自己的模板可以绑定到组件的**任意**属性，无论是否使用了 @[Input](https://www.angular.cn/api/core/Input) 装饰器。

但组件或指令不应该盲目的信任其它组件或指令。 因此组件或指令的属性默认是不能被绑定的。 从 Angular 绑定机制的角度来看，它们是**私有**的，而当添加了 @[Input](https://www.angular.cn/api/core/Input)时，它们变成了**公共**的 只有这样，它们才能被其它组件或属性绑定。

你可以在一个宿主元素上应用多个**属性型**指令，但[只能应用一个](https://www.angular.cn/guide/structural-directives#one-per-element)**结构型**指令。

\*ngIf 的 \* 其实是一个语法糖，它会将宿主元素放在一个 <ng-template> 中，ngIf 是 ng-template 的输入属性。所以，如果是作用于非 ng-template 标签上，需要使用 \*，但当作用于 ng-template 时，可以省略 \* 号。

<ng-template [ngFor](https://www.angular.cn/api/common/NgForOf) let-hero [[ngForOf](https://www.angular.cn/api/common/NgForOf)]="heroes" let-i="index" let-odd="[odd](https://www.angular.cn/api/common/NgForOfContext#odd)" [[ngForTrackBy](https://www.angular.cn/api/common/NgForOf#ngForTrackBy)]="[trackById](https://www.angular.cn/api/core/IterableChangeRecord#trackById)"> <div [class.odd]="[odd](https://www.angular.cn/api/common/NgForOfContext#odd)">({{i}}) {{hero.name}}</div> </ng-template>

微语法：会翻译成 ng-template 上的输入属性：

ngFor 中的 let item of xxx; let i=index;

会被翻译成 let-item let-i 作为 ng-template 输入属性存在。

ng-container 可用来给指令使用，在不想额外增加 html 元素时，来包裹一堆子元素应用相关指令。因为，它并不会被渲染在html 视图上。

ng-template 内的所有内容会被忽视掉，不会被渲染，除非手动去处理，所以经常用来作为参数传给一些三方组件，因为组件内部会去处理 ng-template。

这个例子添加了一些 getter 方法。在响应式表单中，你通常会通过它所属的控件组（FormGroup）的 get 方法来访问表单控件，但有时候为模板定义一些 getter 作为简短形式。 吧

注意，name 和 alterEgo 是兄弟控件。要想在单个的自定义验证器中计算这两个控件，我们就得在它们共同的祖先控件（[FormGroup](https://www.angular.cn/api/forms/FormGroup)）中进行验证。这样，我们就可以查询 [FormGroup](https://www.angular.cn/api/forms/FormGroup) 的子控件，从而让我们能够比较它们的值。

要想给 [FormGroup](https://www.angular.cn/api/forms/FormGroup) 添加验证器，就要在创建时把一个新的验证器传给它的第二个参数。