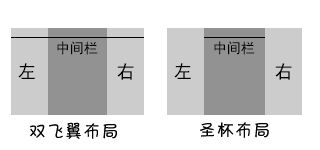
**1.圣杯布局和双飞翼布局的理解和区别，并用代码实现**

圣杯布局和双飞翼布局基本上是一致的，都是两边固定宽度，中间自适应的三栏布局，其中，中间栏放到文档流前面，保证先行渲染。解决方案大体相同，都是三栏全部float:left浮动，区别在于解决中间栏div的内容不被遮挡上，圣杯布局是中间栏在添加相对定位，并配合left和right属性，效果上表现为三栏是单独分开的（如果可以看到空隙的话），而双飞翼布局是在中间栏的div中嵌套一个div，内容写在嵌套的div里，然后对嵌套的div设置margin-left和margin-right，效果上表现为左右两栏在中间栏的上面，中间栏还是100%宽度，只不过中间栏的内容通过margin的值显示在中间。

**对比图：**

****

 圣杯布局：

<body>

<div id='header'>header</div>

<div id='bd'>

<div id='middle'>middle</div>

<div id='left'>left</div>

<div id='right'>right</div>

</div>

<div id='footer'>footer</div>

</body>

css:

<style>

#header{

height:50px;

background:#666;

text-align: center;

}

#bd{

height:200px;

padding:0 200px 0 180px;

}

#footer{

height:50px;

background:#666;

text-align:center;

}

#middle{

float:left;

width:100%;

height:200px;

background:blue;

}

#left{

float:left;

width:180px;

height:200px;

background:yellow;

position:relative;/\*给left推到左边\*/

left:-180px;/\*给left推到左边\*/

margin-left:-100%;/\*让left上去，靠左\*/

}

#right{

float:left;

width:200px;

height:200px;

background:pink;

position:relative;/\*给right推到右边\*/

right:-200px;/\*给right推到右边\*/

margin-left:-200px;/\*让right上去，靠右\*/

}

</style>

双飞翼布局：

<body>

<div id='header'>header</div>

<div id='middle'>

<div id='inside'>middle</div>

</div>

<div id='left'>left</div>

<div id='right'>right</div>

<div id='footer'>footer</div>

</body>

css:

<style>

#header{

height:50px;

text-align:center;

background:#666;

}

#middle{

width:100%;

height:100px;

background:yellow;

float:left;

}

#left{

float:left;

width:180px;

height:100px;

background:blue;

margin-left:-100%;

}

#right{

float:left;

width:200px;

height:100px;

background:pink;

margin-left:-200px;

}

#inside{

margin:0 200px 0 180px;

height:100px;

}

#footer{

clear:both;

height:50px;

text-align:center;

background:#666;

}

</style>

圣杯布局和双飞翼布局解决的问题是一样的，就是两边顶宽，中间自适应的三栏布局，中间栏要在放在文档流前面以优先渲染。

圣杯布局和双飞翼布局解决问题的方案在前一半是相同的，也就是三栏全部float浮动，但左右两栏加上负margin让其跟中间栏div并排，以形成三栏布局。

不同在于解决”中间栏div内容不被遮挡“问题的思路不一样：

圣杯布局，为了中间div内容不被遮挡，将中间div设置了左右padding-left和padding-right后，将左右两个div用相对布局position: relative并分别配合right和left属性，以便左右两栏div移动后不遮挡中间div。

双飞翼布局，为了中间div内容不被遮挡，直接在中间div内部创建子div用于放置内容，在该子div里用margin-left和margin-right为左右两栏div留出位置。

多了1个div，少用大致4个css属性（圣杯布局中间divpadding-left和padding-right这2个属性，加上左右两个div用相对布局position: relative及对应的right和left共4个属性，一共6个；而双飞翼布局子div里用margin-left和margin-right共2个属性，6-2=4）

typeof 对于原始类型来说，除了 null 都可以显示正确的类型

**typeof** 1 // 'number'

**typeof** '1' // 'string'

**typeof** undefined // 'undefined'

**typeof** true // 'boolean'

**typeof** Symbol() // 'symbol'

typeof 对于对象来说，除了函数都会显示 object，所以说 typeof 并不能准确判断变量到底是什么类型

**typeof** [] // 'object'

**typeof** {} // 'object'

**typeof** console.log // 'function'

如果我们想判断一个对象的正确类型，这时候可以考虑使用 instanceof，因为内部机制是通过原型链来判断的

首先我们要知道，在 JS 中类型转换只有三种情况，分别是：

转换为布尔值

转换为数字

转换为字符串

在条件判断时，除了 undefined， null， false， NaN， ''， 0， -0，其他所有值都转为 true，包括所有对象。

加法运算符不同于其他几个运算符，它有以下几个特点：

1)运算中其中一方为字符串，那么就会把另一方也转换为字符串

2)如果一方不是字符串或者数字，那么会将它转换为数字或者字符串

首先，new 的方式优先级最高，接下来是 bind 这些函数，然后是 obj.foo() 这种调用方式，最后是 foo 这种调用方式，同时，箭头函数的 this 一旦被绑定，就不会再被任何方式所改变。

在 JS 中，闭包存在的意义就是让我们可以间接访问函数内部的变量。

for (var i = 1; i <= 5; i++) {

setTimeout(function timer() {

console.log(i)

}, i \* 1000)

}

解决办法：

1）

for(let i=1;i<=5;i++){

setTimeout(function timer(){

console.log(i)

},i\*1000)

}

2）使用 setTimeout 的第三个参数，这个参数会被当成 timer 函数的参数传入。

for(var i=1;i<=5;i++){

setTimeout(function timer(j){

console.log(j)

},i\*1000,i)

}

3）使用闭包的方式

for(var i=1;i<=5;i+=){

(function(j){

setTimeout(function timer(){

console.log(j)

},j\*1000)

})(i)

}

首先使用了立即执行函数将 i 传入函数内部，这个时候值就被固定在了参数 j 上面不会改变，当下次执行 timer 这个闭包的时候，就可以使用外部函数的变量 j，从而达到目的。

原型的 constructor 属性指向构造函数，构造函数又通过 prototype 属性指回原型，但是并不是所有函数都具有这个属性，Function.prototype.bind() 就没有这个属性。

**[html] 第1天 页面导入样式时，使用link和@import有什么区别？**

1）从属关系的区别：link属于XHTML标签，而@import是CSS提供的语法规则，link除了加载CSS，还可以定义RSS，定义rel连接属性等，@import就只能加载CSS。

2）加载顺序的区别：页面加载时，link会同时被加载，而@import引用的CSS会等页面被加载完后再加载。

3）兼容性的区别：@import只有IE5以上才能被识别，而link是XHTML标签，无兼容问题。

4）DOM可控性区别：通过js操作DOM,可以插入link标签来改变样式；由于DOM方法是基于文档的，无法使用@import方式插入样式

**写一个方法去掉字符串中的空格**

var str = ' abc d e f g ';

str = str.replace(/\s/g,'');

str = str.split(' ').join('');

console.log(str)

**HTML 全局属性：可用于任何 HTML 元素。**

|  |  |
| --- | --- |
| **属性** | **描述** |
| [accesskey](http://www.w3school.com.cn/tags/att_standard_accesskey.asp) | 规定激活元素的快捷键。 |
| [class](http://www.w3school.com.cn/tags/att_standard_class.asp) | 规定元素的一个或多个类名（引用样式表中的类）。 |
| [contenteditable](http://www.w3school.com.cn/tags/att_global_contenteditable.asp) | 规定元素内容是否可编辑。 |
| [contextmenu](http://www.w3school.com.cn/tags/att_global_contextmenu.asp) | 规定元素的上下文菜单。上下文菜单在用户点击元素时显示。 |
| [data-\*](http://www.w3school.com.cn/tags/att_global_data.asp) | 用于存储页面或应用程序的私有定制数据。 |
| [dir](http://www.w3school.com.cn/tags/att_standard_dir.asp) | 规定元素中内容的文本方向。 |
| [draggable](http://www.w3school.com.cn/tags/att_global_draggable.asp) | 规定元素是否可拖动。 |
| [dropzone](http://www.w3school.com.cn/tags/att_global_dropzone.asp) | 规定在拖动被拖动数据时是否进行复制、移动或链接。 |
| [hidden](http://www.w3school.com.cn/tags/att_global_hidden.asp) | 规定元素仍未或不再相关。 |
| [id](http://www.w3school.com.cn/tags/att_standard_id.asp) | 规定元素的唯一 id。 |
| [lang](http://www.w3school.com.cn/tags/att_standard_lang.asp) | 规定元素内容的语言。 |
| [spellcheck](http://www.w3school.com.cn/tags/att_global_spellcheck.asp) | 规定是否对元素进行拼写和语法检查。 |
| [style](http://www.w3school.com.cn/tags/att_standard_style.asp) | 规定元素的行内 CSS 样式。 |
| [tabindex](http://www.w3school.com.cn/tags/att_standard_tabindex.asp) | 规定元素的 tab 键次序。 |
| [title](http://www.w3school.com.cn/tags/att_standard_title.asp) | 规定有关元素的额外信息。 |
| [translate](http://www.w3school.com.cn/tags/att_global_translate.asp) | 规定是否应该翻译元素内容。 |

**JavaScript中label语句的使用**

break和continue语句都可以与label联合使用，从而达到代码中特定的位置，这种联合使用的情况多发生在循环嵌套的情况下。

var num = 0;

outPoint://这里为label，标签名为outPoint

for (var i = 0; i < 10; i++) {

for (var j = 0; j < 10; j++) {

if (i == 5 && j == 5) {

break outPoint;

}else{

console.log(i,j,88);

}

num++;

}

}

console.log(num); //55

内部循环中break语句带了一个参数，要返回到的标签。添加这个标签的结果将导致break语句不仅会退出内部for循环，而且也会退出外部的for循环

var num = 0;

outermost:

for (var i = 0; i < 10; i++) {

for (var j = 0; j < 10; j++) {

if (i == 5 && j == 5) {

continue outermost;

}else{

console.log(i,j,88);

}

num++;

}

}

console.log(num); //95

在这种情况下，continue语句会强制继续执行循环（退出内部循环，执行外部循环）。当变量i和j都等于5时，num的值正好是95.

**HTML <label> 标签**

所有主流浏览器都支持 <label> 标签。

**定义和用法**

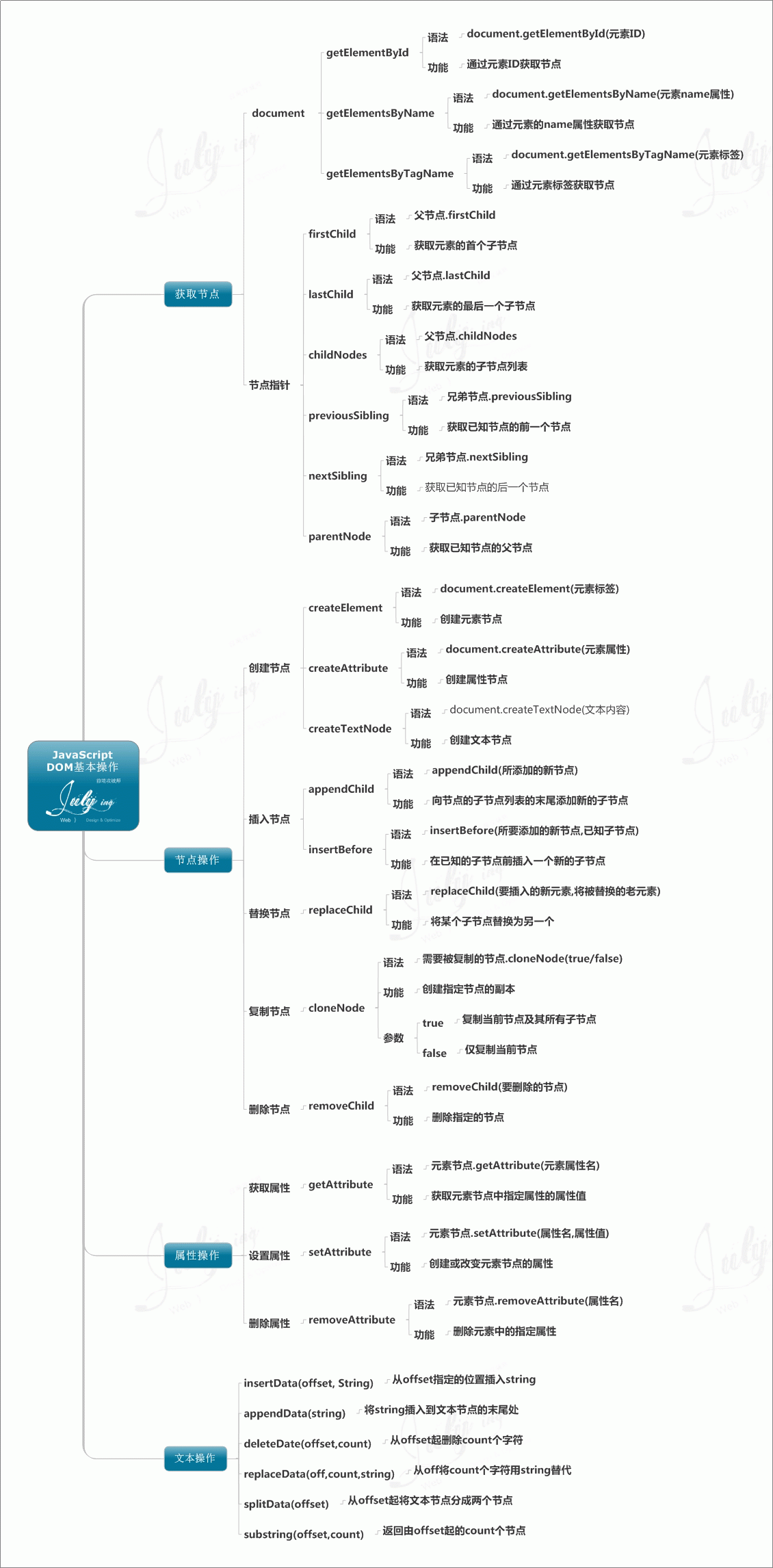
<label> 标签为 input 元素定义标注（标记）。

label 元素不会向用户呈现任何特殊效果。不过，它为鼠标用户改进了可用性。如果您在 label 元素内点击文本，就会触发此控件。就是说，当用户选择该标签时，浏览器就会自动将焦点转到和标签相关的表单控件上。

<label> 标签的 for 属性应当与相关元素的 id 属性相同。

|  |  |  |
| --- | --- | --- |
| **属性** | **值** | **描述** |
| [for](http://www.w3school.com.cn/tags/att_label_for.asp) | *id* | 规定 label 绑定到哪个表单元素。 |
| [form](http://www.w3school.com.cn/tags/att_label_form.asp) | *formid* | 规定 label 字段所属的一个或多个表单。 |

**JavaScript DOM操作：**

  
DOM 是 W3C（万维网联盟）的标准。

DOM 定义了访问 HTML 和 XML 文档的标准：

“W3C 文档对象模型 （DOM） 是中立于平台和语言的接口，它允许程序和脚本动态地访问和更新文档的内容、结构和样式。”

W3C DOM 标准被分为 3 个不同的部分：

核心 DOM - 针对任何结构化文档的标准模型

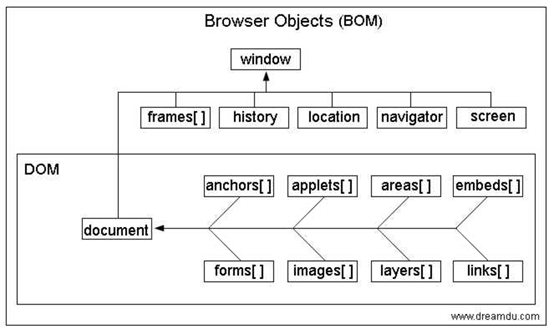
XML DOM - 针对 XML 文档的标准模型

HTML DOM - 针对 HTML 文档的标准模型

浏览器对象模型（BOM）以 window 对象为依托，表示浏览器窗口以及页面可见区域。同时， window对象还是 ECMAScript 中的 Global 对象，

因而所有全局变量和函数都是它的属性，且所有原生的构造函数及其他函数也都存在于它的命名空间下。

BOM 和 DOM 的关系图解



DOM 是为了操作文档出现的 API，document 是其的一个对象；

BOM 是为了操作浏览器出现的 API，window 是其的一个对象。

**正常模式和严格模式**

ECMAScript 5 的严格模式是可选择的一个限制JavaScript的变体一种方式 。

严格模式不仅仅是一个子集：它故意地具有与正常代码不同的语义。

不支持严格模式的浏览器将会执行与支持严格模式的浏览器不同行为的严格模式代码。

进入"严格模式"的标志，是下面这行语句：**"use strict";**

**一、引用方式**

**严格模式标志**

|  |  |
| --- | --- |
| 1  2  3 | "use strict"; |

**针对整个脚本文件**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <script>      "use strict";  [console.log(](https://kdb.homecredit.cn/pages/console.log()"这是严格模式。");  </script>  <script>  [console.log(](https://kdb.homecredit.cn/pages/console.log()"这是正常模式。");  </script> |

**针对单个函数的严格模式**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function strict(){  　　  "use strict";  　　　　return "这是严格模式。";  　　}  　　function notStrict() {  　　　　return "这是正常模式。";  　　} |

**针对模块的严格模式**

|  |  |
| --- | --- |
| 1  2  3  4  5 | require(['moduleA', 'moduleB', 'moduleC'], function (moduleA, moduleB, moduleC){        "use strict";  　　// some code here  }); |

**声明匿名函数**

|  |  |
| --- | --- |
| 1  2  3  4 | "use strict";  function (){  [console.info(123)](https://kdb.homecredit.cn/pages/console.info(123))  } |

**二、语法和行为改变**

**2.1 全局变量显式声明**

**变量声明**

|  |  |
| --- | --- |
| 1  2  3  4 | "use strict";  v = 1; // 报错，v未声明  for(i = 0; i < 2; i++) { // 报错，i未声明  } |

在正常模式中，如果一个变量没有声明就赋值，默认是全局变量。严格模式禁止这种用法，全局变量必须显式声明。

**2.2 静态绑定**

Javascript 语言的一个特点，就是允许"动态绑定"，即某些属性和方法到底属于哪一个对象，不是在编译时确定的，而是在运行时（runtime）确定的。

**禁止使用with语句**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | "use strict";  function Company(name) {      this.name = name;  }  var HomeCredit = new Company("HOME CREDIT");  with(HomeCredit){     // 语法错误      var welcome= "Hello  " + name + " !";  [console.info(welcome);](https://kdb.homecredit.cn/pages/console.info(welcome);%C2%A0)  } |

严格模式对动态绑定做了一些限制。某些情况下，只允许静态绑定。也就是说，属性和方法到底归属哪个对象，在编译阶段就确定。

这样做有利于编译效率的提高，也使得代码更容易阅读，更少出现意外。

**创建eval作用域**

|  |  |
| --- | --- |
| 1  2  3  4 | "use strict";  var x = 2;  [console.info(eval(](https://kdb.homecredit.cn/pages/console.info(eval()"var x = 5; x")); // 5  [console.info(x);](https://kdb.homecredit.cn/pages/console.info(x);) // 2 |

正常模式下，JavaScript语言有两种变量作用域（scope）：全局作用域和函数作用域。严格模式创设了第三种作用域：eval作用域。

正常模式下，eval语句的作用域，取决于它处于全局作用域，还是处于函数作用域。严格模式下，eval语句本身就是一个作用域，不再能够生成全局变量了，它所生成的变量只能用于eval内部。

**2.3 增强的安全措施**

禁止 this 关键字指向全局对象

**示例一**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function f(){  　　　　return !this;  　　}  　　// 返回false，因为"this"指向全局对象，"!this"就是false  　　function f(){  　　　　"use strict";  　　　　return !this;  　　}  　　// 返回true，因为严格模式下，this的值为undefined，所以"!this"为true。 |

**示例二**

|  |  |
| --- | --- |
| 1  2  3  4  5 | function f(){  　　　　"use strict";  　　　　this.a = 1;  　　};  　　f();// 报错，this未定义 |

**禁止在函数内部遍历调用栈**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function f1(){  　　　　"use strict";  [f1.caller;](https://kdb.homecredit.cn/pages/f1.caller;) // 报错  [f1.arguments;](https://kdb.homecredit.cn/pages/f1.arguments;) // 报错  　　}  　　f1(); |

**为什么使用严格模式:**

消除JavaScript语法的一些不合理、不严谨之处，减少一些怪异行为;

消除代码运行的一些不安全之处，保证代码运行的安全；

提高编译器效率，增加运行速度；

为未来新版本的JavaScript做好铺垫。

**语言类型**

JS 属于动态弱类型语言。（Java是静态强类型）。

弱类型相对于强类型来说类型检查更不严格，比如说允许变量类型的隐式转换，允许强制类型转换等等。

强类型语言一般不允许这么做。

**二.引用类型转值类型:**

（1）toString() 方法。

（2）valueOf() 方法。

通常情况下我们认为，将一个对象转换为字符串要调用 toString() 方法，转换为数字要调用 valueOf() 方法，但是真正应用的时候并没有这么简单

总结如下:

（1）有些对象只是简单继承了 toString() 或者 valueOf() 方法，比如第一个例子。

（2）有些对象则不但是继承了两个方法，而且还进行了重写。

所以有些对象的方法能够达成转换成字符串或者数字的目标，有些则不能。

调用 toString() 或者 valueOf() 将对象转换成字符串或者数字的规则如下:

调用 toString() 时，如果对象具有这个方法，则调用此方法；如果此方法返回一个值类型数据，那么就返回这个值类型数据，然后再根据所处的上下文环境进行相关数据类型转换。

如果没有 toString()，或者此方法返回值并不是一个值类型数据，那么就会调用valueOf()（如果此方法存在的话），如果valueOf()返回一个值类型数据，那么再根据所处的上下文环境进行相关的数据类型转换。

进一步说明:

（1）上面介绍了通常默认情况下 valueOf() 和 toString() 方法的作用（将对象转换为数字或者字符串），但是需要注意的是，这并不是硬性规定，也就是说并不是 valueOf() 方法必须

要返回数字或者 toString() 方法必须要转换为字符串，比如简单继承的这两个方法就无法进行实现转换为数字和字符串的功能，再比如，我们可以自己重写这两个方法，返回值也没有必要是数字或者字符串。

（2）还有需要特别注意的一点就是，很多朋友认为，转换为字符串首先要调用 toString() 方法， 其实这是错误的认识，我们应该这么理解，调用 toString() 方法可以转换为字符串，但不一定转换字符串就是首先调用 toString() 方法。

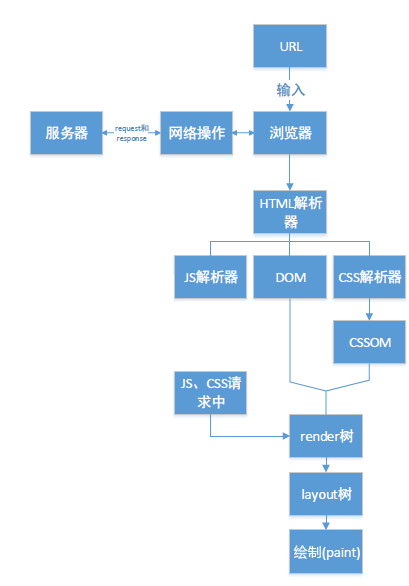
**总结如下:**

大多数对象隐式转换为值类型都是首先尝试调用 valueOf() 方法。但是 Date对象 是个例外，此对象的 valueOf() 和 toString() 方法都经过精心重写，默认是调用 toString() 方法，比如使用+运算符，如果在其他算数运算环境中，则会转而调用 valueOf() 方法。

JS中所有数值不管是整数还是浮点数都是以浮点数的形式保存的。

**机制**

**网页生成过程**



**大致可以分成五步：**

1)HTML 代码转化成 DOM。

2)CSS 代码转化成 CSSOM（CSS Object Model）

3)结合 DOM和 CSSOM，生成一棵渲染树（包含每个节点的视觉信息）。

4)生成布局（layout），即将所有渲染树的所有节点进行平面合成。

5)将布局绘制（paint）在屏幕上。

**重排和重绘**

"布局"（layout）和"绘制"（paint）这两步，合称为"渲染"（render）。

重新渲染，就需要重新生成布局和重新绘制。前者叫做"重排"（reflow），后者叫做"重绘"（repaint）。

需要注意的是，"重绘"不一定需要"重排"，比如改变某个网页元素的颜色，就只会触发"重绘"，不会触发"重排"，因为布局没有改变。但是，"重排"必然导致"重绘"，比如改变一个网页元素的位置，就会同时触发"重排"和"重绘"，因为布局改变了。

一般的规则是：

样式表越简单，重排和重绘就越快。

重排和重绘的DOM元素层级越高，成本就越高。

table元素的重排和重绘成本，要高于div元素。

**执行机制**

**单线程**

JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。那么，为什么JavaScript不能有多个线程呢？这样能提高效率啊。

JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。

比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？

所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

**任务队列**

单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。

如果排队是因为计算量大，CPU忙不过来，倒也算了，但是很多时候CPU是闲着的，因为IO设备（输入输出设备）很慢（比如Ajax操作从网络读取数据），不得不等着结果出来，再往下执行。

JavaScript语言的设计者意识到，这时主线程完全可以不管IO设备，挂起处于等待中的任务，先运行排在后面的任务。等到IO设备返回了结果，再回过头，把挂起的任务继续执行下去。

于是，所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；

异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

具体来说，异步执行的运行机制如下。（同步执行也是如此，因为它可以被视为没有异步任务的异步执行。）

（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

定义函数的方式：1）函数声明

2）函数表达式

3）new Function()

4)箭头函数

函数声明的调用方式:1)fn()

2)fn.call(fn,’aaa’)

3)fn.apply(fn,[‘aaa’])

4)new fn()

5)setTimeout(fn(),0)

6)fn`aaa`

7)(function fn(){})()

##### 标签模板

模板字符串的功能，不仅是上面那些，它还可以紧跟在一个函数后面，该函数将被调用来处理这个模板字符串，这种称为“标签模板”功能(Tagged template)。

标签模板函数第一个参数是字符串模板的常量数组，后面的每一个参数为表达式的计算结果，函数名称可以任意指定。

标签模板其它是一种特殊的函数调用形式，“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。

函数表达式：

const fn = function(text){

  console.log(text)

}('函数表达式')

声明函数的第三种方法new Function – 不推荐使用

new Function (arg1, arg2, ... , function\_body)

Function的最后一个参数是函数体，之前的参数是函数的参数。  
Function的参数必须是字符串。

function add(a, b) {

retrun a + b;

}

//等价于

var add = new Function ('a', 'b', 'return a + b');

()中只能放一个表达式

function add(){

var a=1;

a+=1;

alert(a);

}()

//函数声明后面加()不会立即执行，报错

var add = function(){

var a = 1;

a+=1;

alert(a);

}()

//函数表达式后面加()会立即执行

函数声明转为函数表达式的方式：

1）

(function fn(){})()

2）

(function fn(){}())

自执行函数的作用：1)不必为函数命名，避免全局变量的使用

函数变成表达式的方式：

0 + function(text){

console.log(text);

}('aaa');

0,function(text){

console.log(text);

}('bbb')

true && function(text){

console.log(text)

}('ccc')

false || function(text){

console.log(text)

}('ddd')

~function(text){

console.log(text)

}('eee')

+function(text){

console.log(text)

}('fff')

-function(text){

console.log(text)

}('ggg')

typeof function(text){

console.log(text)

}('hhh')

new function(text){

console.log(text)

}('iii')

**逗号操作符** 对它的每个操作数求值（从左到右），并返回最后一个操作数的值。

;function aa(){}

分号是为了和前面的代码隔开，js可以用换行分隔代码，但是合并压缩多个js文件之后，换行符一般会被删掉，所以连在一起可能会出错，加上分号就保险了。

自执行函数的作用：2）模块化开发

3）惰性函数（可以提高性能）

<https://www.cnblogs.com/xiaohuochai/p/8028192.html>

惰性函数表示函数执行的分支只会在函数第一次调用的时候执行，在第一次调用过程中，该函数会被覆盖为另一个按照合适方式执行的函数，这样任何对原函数的调用就不用再经过执行的分支了。

**函数重写**

在介绍惰性函数之前，首先介绍函数重写技术。由于一个函数可以返回另一个函数，因此可以用新的函数来覆盖旧的函数

**惰性函数**

惰性函数的本质就是函数重写。所谓惰性载入，指函数执行的分支只会发生一次，有两种实现惰性载入的方式

1. 第一种是在函数被调用时，再处理函数。函数在第一次调用时，该函数会被覆盖为另外一个按合适方式执行的函数，这样任何对原函数的调用都不用再经过执行的分支了。代码重写如下

function addEvent(type, element, fun) {

if (element.addEventListener) {

addEvent = function (type, element, fun) {

element.addEventListener(type, fun, false);

}

}

else if(element.attachEvent){

addEvent = function (type, element, fun) {

element.attachEvent('on' + type, fun);

}

}

else{

addEvent = function (type, element, fun) {

element['on' + type] = fun;

}

}

return addEvent(type, element, fun);

}

在这个惰性载入的addEvent()中，if语句的每个分支都会为addEvent变量赋值，有效覆盖了原函数。最后一步便是调用了新赋函数。下一次调用addEvent()时，便会直接调用新赋值的函数，这样就不用再执行if语句了

但是，**这种方法有个缺点，如果函数名称有所改变，修改起来比较麻烦**

1. 第二种是声明函数时就指定适当的函数。把嗅探浏览器的操作提前到代码加载的时候，在代码加载的时候就立刻进行一次判断，以便让addEvent返回一个包裹了正确逻辑的函数

var addEvent = (function () {

if (document.addEventListener) {

return function (type, element, fun) {

element.addEventListener(type, fun, false);

}

}

else if (document.attachEvent) {

return function (type, element, fun) {

element.attachEvent('on' + type, fun);

}

}

else {

return function (type, element, fun) {

element['on' + type] = fun;

}

}

})();

ES6代码，很多时候都会看到三个点(...)的存在,它在ES6语法中，有两种应用形式，分别为函数中的rest参数，以及扩展运算符

ES6 rest参数

rest参数（形式为“...变量名”）可以称为不定参数，rest参数和一个变量名搭配使用，生成一个数组，用于获取函数多余的参数

rest参数作用： 将多余的逗号分隔的参数序列转换为数组参数

注意： rest参数必须是最后一个参数，否则报错

函数的 length 属性，不包括rest参数

如果传入的参数连正常定义的参数都没填满，也不要紧，rest参数会接收一个空数组（注意不是undefined）。

Rest 参数接受函数的多余参数，组成一个数组，放在形参的最后，形式如下：

function func(a, b, ...theArgs){

// ...

}

**(2) Rest参数和arguments对象的区别：**

rest参数只包括那些没有给出名称的参数，arguments包含所有参数

arguments 对象不是真正的数组，而rest 参数是数组实例，可以直接应用sort, map, forEach, pop等方法

arguments 对象拥有一些自己额外的功能

**扩展运算符**

扩展运算符可以理解为rest参数的逆运算，将数组转换为逗号分隔的参数序列

**扩展运算符的应用(常用)**

a、合并数组

扩展运算符提供了数组合并的新写法。//es6

arr1 = [1,2,3]

arr2 = [4,5,6]

arr3 = [7,8,9]

arr4 = [...arr1, ...arr2, ...arr3] //[1,2,3,4,5,6,7,8,9] 用于数组合并

arr1.concat(arr2, arr3); //es5

// [1,2,3,4,5,6,7,8,9]

b、与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于生成数组。

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1

rest // [2, 3, 4, 5]

const [first, ...rest] = [];

first // undefined

rest // []:

const [first, ...rest] = ["foo"];

first // "foo"

rest // []

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

const [...butLast, last] = [1, 2, 3, 4, 5];

// 报错

const [first, ...middle, last] = [1, 2, 3, 4, 5];

// 报错

什么是 Function Expression（函数表达式）？

Function Expression 将函数定义为表达式语句（通常是变量赋值）的一部分。

通过 Function Expression 定义的函数可以是命名的，也可以是匿名的。

Function Expression 不能以“function”开头（下面自调用的例子要用括号将其括起来）。

那 Function Declaration（函数声明）是什么？

Function Declaration 可以定义命名的函数变量，而无需给变量赋值。

Function Declaration 是一种独立的结构，不能嵌套在非功能模块中。可以将它类比为 Variable Declaration（变量声明）。

就像 Variable Declaration 必须以“var”开头一样，Function Declaration 必须以“function”开头。

**变量的声明**

JS不同于JAVA，在变量声明时并不需要声明变量的存储空间。

变量中所存储的数据可以分为两类：基本类型和引用类型。其中数值、布尔值、null和undefined属于基本类型，对象、数组和函数属于引用类型。

基本类型在内存中具有固定的内存大小。例如：数值型在内存中占有八个字节，布尔值只占有一个字节。

对于引用型数据，他们可以具有任意长度，因此他们的内存大小是不定的，

因此变量中存储的实际上是对此数据的引用，通常是内存地址或者指针，通过它们我们可以找到这个数据。

JS中变量声明分显式声明和隐式声明。

在函数中使用var关键字进行显式申明的变量是做为局部变量，而没有用var关键字，使用直接赋值方式声明的是全局变量。

用 var 定义的变量还有一个特点，就是不能被主动 delete.

我们从最基本的开始，在面向对象的强类型语言中（Java），其作用域都是基于块的（即：{}），块内可以对块外的变量进行操作，但是块外却对块内的变量是无法操作的。

但是JS呢？一门弱类型语言，其并没有实现基于块的作用域，而是**基于function的**。

JavaScript运行三部曲

脚本执行的整个过程中，js 引擎都做了什么呢？

语法分析

预编译

解释执行

在执行代码前，还有两个步骤

语法分析很简单，就是引擎检查你的代码有没有什么低级的语法错误

预编译简单理解就是在内存中开辟一些空间，存放一些变量与函数

解释执行顾名思义便是执行代码了

js的代码在首次被加载完成后进行编译时，会将**所有的function和var提前进行声明，但是并不会对其进行赋值，赋值则都是在该代码块进行执行时才会对其进行赋值，**

如果不对变量进行var的话，它是不会存在于function执行的时创建的新Scope的。

console.log(varscope);

var varscope='test' //undefined

console.log(notVarScope);

notVarScope='test1'; //notVarScope is not defined

var a = 1;

var b = 2;

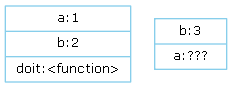
function doit(){

var b = 3;

console.log(a);

}

doit(4);



在js中，var和function在预编译中，作用都是一样的，都是提前声明了变量，~~但是并没有对其进行赋值~~，**变量及表达式没有对其赋值**，**函数会被赋值** ，所以这段代码完整的Scope应该是拥有这三个变量的，只是doit是一个指向堆的引用。而在doit执行时，才会创建新的Scope，由于js语言的特殊性，虽然doit在这个Scope里定义的，但是其执行环境可以通过引用改变到任何地方，但是doit这个函数的定义环境永远都是确定的，即这个Scope内。

作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

**概括几点：**

第一点：js没有块级作用域（你可以自己闭包或其他方法实现），只有函数级作用域，函数外面的变量函数里面可以找到，函数里面的变量外面找不到。

var a = 10;

function aaa(){ //step-4

alert(a);//step-5->执行alert，此时只能找到外面的a=10故弹框10

}

function bbb(){//step-2

var a = 20;

aaa();//step-3

}

//定义了函数没啥用，调用才是真格的所以这里是step-1

bbb();//step-1

第二点：变量的查找是就近原则，去寻找var定义的变量，当就近没有找到的时候就去查找外层。

var a=10;

function aaa(){

alert(a);//undefined，查找a的时候会现在函数内查找，由于预解析的作用，此时的a是undefined,因此永远不会去查找外面的10了

var a = 20;

/\*预解析

var a

alert(a);

var a = 20;\*/

}

aaa();

第三点：当参数跟局部变量重名时，优先级是等同的。

var a=10;

function aaa(a){

console.info(a);

var a = 20;

}

aaa(a); //10

还有：传参时，基本类型传值，引用类型传引用。（但是重新赋值之后就不是这样了喔）

var a=[1, 2, 3];

var b=a;

b=[1, 2, 3, 4];

console.info(a); //[1, 2, 3]

**无论函数是在哪里调用，也无论函数是如何调用的，其确定的词法作用域永远都是在函数被声明的时候确定下来的**

总结：

js 没有块级作用域，除了全局作用域之外，只有函数可以创建作用域，所以，我们在声明变量时，全局代码要在代码前端声明，函数中药在函数体一开始就声明好，除了这两个地方，其他地方都不要出现变量声明，而且建议使用“单var” 模式。

二.JS堆栈研究

1、栈（stack）和堆（heap）

stack为自动分配的内存空间，它由系统自动释放；而heap则是动态分配的内存，大小不定也不会自动释放。

2、基本类型和引用类型

（1）基本类型：存放在栈内存中的简单数据段，数据大小确定，内存空间大小可以分配。

5种基本数据类型有Undefined、Null、Boolean、Number 和 String，它们是直接按值存放的，所以可以直接访问。

（2）引用类型：存放在堆内存中的对象，变量实际保存的是一个指针，这个指针指向另一个位置。每个空间大小不一样，要根据情况开进行特定的分配。

当我们需要访问引用类型（如对象，数组，函数等）的值时，首先从栈中获得该对象的地址指针，然后再从堆内存中取得所需的数据。

3、传值与传址

前面之所以要说明什么是内存中的堆、栈以及变量类型，实际上是为了更好的理解什么是“浅拷贝”和“深拷贝”。

基本类型与引用类型最大的区别实际就是传值与传址的区别。

深度拷贝: 既然属性值类型是数组和或象时只会传址，那么我们就用递归来解决这个问题，把父对象中所有属于对象的属性类型都遍历赋给子对象即可。

判断一个变量是不是对象非常简单。值类型的类型判断用 typeof，引用类型的类型判断用 instanceof。

js 是弱类型语言 , 一切（引用类型）都是对象，**对象是属性的集合**。

**函数和对象的关系**

**对象都是通过函数创建的**

//语法糖 var obj = { a: 10, b: 20 };

var obj = new Object();

obj.a = 10;

obj.b = 20;

//语法糖 var arr = [5, 'x', true];

var arr = new Array();

arr[0] = 5;

arr[1] = 'x';

arr[2] = true;

//语法糖 function functionName(arg1, arg2, ..., argN){}

var functionName = new Function(arg1, arg2, ..., argN,function\_body);

// example

var test= new Function('a','b','c','alert(a);alert(b);alert(c)');

**对象是函数创建的，而函数却又是一种对象**

先别迷糊，想知道他们到底什么关系我们得先去了解一下 prototype 原型。

function Fn() { }

Fn.prototype.name = 'Home Credit';

Fn.prototype.getYear = function () {

return 2004;

};

var fn = new Fn();

console.log(fn.name);

console.log(fn.getYear());

即，Fn是一个函数，fn对象是从Fn函数new出来的，这样fn对象就可以调用Fn.prototype中的属性。

因为每个对象都有一个隐藏的属性 “ \_\_proto\_\_ ”，这个属性引用了创建这个对象的函数的prototype。即：fn.\_\_proto\_\_ === Fn.prototype

这里的 " \_\_proto\_\_ " 他的官方名字叫 “non-standard \_\_proto\_\_ pseudo-property”、“非标准的\_\_proto\_\_魔法属性”。业内通常称之为 “隐式原型”、“非标准选择器”

new

特别注意，如果要对prototype进行修改，尽可能的不要使用对象字面量方式进行修改

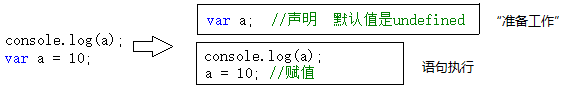
建议使用 A.prototype.aProp={}；

非常不建议 A.prototype = { aProp : XXX }

**执行上下文**

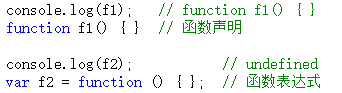
什么是“执行上下文”（也叫“执行上下文环境”）？

在一段js代码拿过来真正一句一句运行之前，浏览器已经做了一些“准备工作”，其中就包括对变量的声明，而不是赋值。变量赋值是在赋值语句执行的时候进行的。可用下图模拟：



与第一种情况不同的是：第一种情况只是对变量进行声明（并没有赋值），而此种情况直接给this赋值。

在第三种情况中，需要注意代码注释中的两个名词——“函数表达式”和“函数声明”。虽然两者都很常用，但是这两者在“准备工作”时，却是两种待遇



我们总结一下，在“准备工作”中完成了哪些工作：

变量、函数表达式——变量声明，默认赋值为undefined；

this——赋值；

函数声明——赋值；

这三种数据的准备情况我们称之为“执行上下文”或者“执行上下文环境”。

函数每被调用一次，都会产生一个新的执行上下文环境。因为不同的调用可能就会有不同的参数。

另外一点不同在于，函数在定义的时候（不是调用的时候），就已经确定了函数体内部自由变量的作用域。

JavaScript 在执行一个代码段之前，都会进行这些“准备工作”来生成执行上下文。这个“代码段”其实分为三种情况（全局代码、函数体、eval 代码）。

eval不常用，也不推荐大家用。

总结一下上下文环境的数据内容。

全局代码的上下文环境数据内容为：

|  |  |
| --- | --- |
| 普通变量（包括函数表达式），  如： var a = 10; | 声明（默认赋值为undefined） |
| 函数声明，  如： function fn() { } | 赋值 |
| this | 赋值 |

如果代码段是函数体，那么在此基础上需要附加

|  |  |
| --- | --- |
| 参数 | 赋值 |
| arguments | 赋值 |
| 自由变量的取值作用域 | 赋值 |

Context ： 函数在被调用时的上下文环境，被调用函数中的 this 指向该上下文环境。

Scope ： 函数定义时的作用域，和变量的可见性相关。

在函数中 this 到底取何值，是在函数真正被调用执行的时候确定的，函数定义的时候确定不了。因为 this 的取值是执行上下文环境的一部分，每次调用函数，都会产生一个新的执行上下文环境。

当函数调用完成时，这个上下文环境以及其中的数据都会被消除，再重新回到全局上下文环境。处于活动状态的执行上下文环境只有一个。

有一种情况，而且是很常用的一种情况，无法做到这样干净利落的说销毁就销毁。这种情况就是伟大的——闭包。

闭包有两大特点——函数作为返回值，函数作为参数传递。

访问一个对象的属性时，先在基本属性中查找，如果没有，再沿着 \_\_proto\_\_ 这条链向上找，这就是原型链。

对象的原型链是沿着 \_\_proto\_\_ 这条线走的，因此在查找 f1.hasOwnProperty 属性时，就会顺着原型链一直查找到 Object.prototype。

由于所有的对象的原型链都会找到Object.prototype，因此所有的对象都会有Object.prototype的方法。这就是所谓的“继承”。

1.ES6 是什么

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，因为当前版本的 ES6 是在 2015 年发布的，所以又称 ECMAScript 2015。

也就是说，ES6 就是 ES2015 。

虽然目前并不是所有浏览器都能兼容ES6全部特性，但越来越多的程序员在实际项目当中已经开始使用ES6了。所以就算你现在不打算使用ES6，但为了看懂别人的你也该懂点ES6的语法了。

ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。第一种场景就是你现在看到的内层变量覆盖外层变量。而 let 则实际上为JavaScript 新增了块级作用域。用它所声明的变量，只在**let命令所在的代码块内有效**。

另外一个var带来的不合理场景就是用来计数的循环变量泄露为全局变量

ES5 规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。

ES6 引入了块级作用域，明确允许在块级作用域之中声明函数。ES6 规定，块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。

函数声明是函数的声明和实现都被提升了。

函数表达式和变量表达式只是其声明被提升了。

ES6 明确规定，如果区块中存在let和const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。