语言介绍

JavaScript 官方名称是 `ECMAScript` 是一种属于网络的脚本语言,已经被广泛用于 Web 应用开发,常用来为网页添加各式各样的动态功能,为用户提供更流畅美观的浏览效果。
1995 年 2 月 Netscape 的`布兰登.艾奇`开发了针对网景公司的 `Netscape Navigator`浏览器的脚本语言 LiveScript。之后 Netscape 与 Sun 公司联盟后 LiveScript 更名为 JavaScript。
微软在 javascript 发布后为了抢占市场推出了 JScript。为了让脚本语言规范不在混乱，根据 javascript 1.1 版本推出了 ECMA-262 的脚本语言标准。
ECMA 是欧洲计算机制造商协会由 Sum、微软、NetScape 公司的程序员组成。

> 文档中会经常使用 `JS` 简写来代替 `JavaScript`

#适用场景

- 浏览器网页端开发
- 做为服务器后台语言使用Node.js(opens new window)
- 移动端手机 APP 开发，如 Facebook 的 React Native (opens new window)、

uniapp、
PhoneGap、
IONIC

- 跨平台的桌面应用程序，如使用 electronjs(opens new window)

> 所以 JS 是一专多能的语言，非常适合学习使用。

#发展历史

- 1994 年 Netscape（网景）公司发布了

`Navigator` 浏览器 1.0 版本，市场占用率超过 90%

- 1995 年发布了

`JavaScript` 语言

- 1996 年 JS 在

`Navigator`浏览器中使用

- 1996 年微软发布

`JScript`在 IE3.0 中使用

- 1996 年 11 月网景公司将 JS 提交给 ECMA(国际标准化组织)成为国际标准，用于对抗微软。

由 ECMA 的第 39 号技术专家委员会（Technical Committee 39，简称 TC39）负责制订 ECMAScript 标准，成员包括 Microsoft、Mozilla、Google 等大公司。

- 1997 年 ECMA 发布 ECMA-262 标准，推出浏览器标准语言

`ECMAScript 1.0`

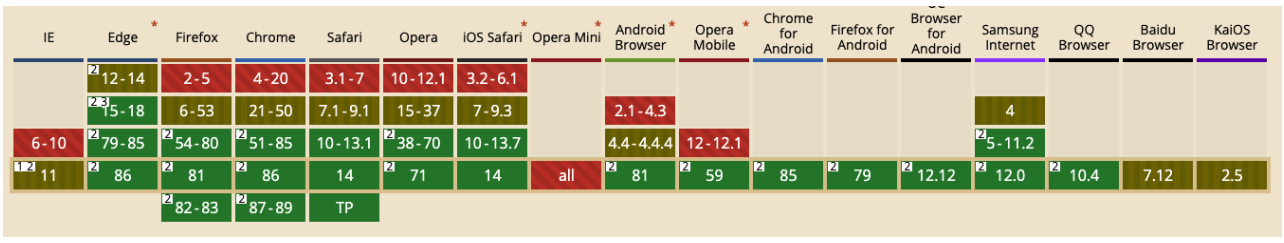
ECMAScript 是标准而 Javascript 是实现

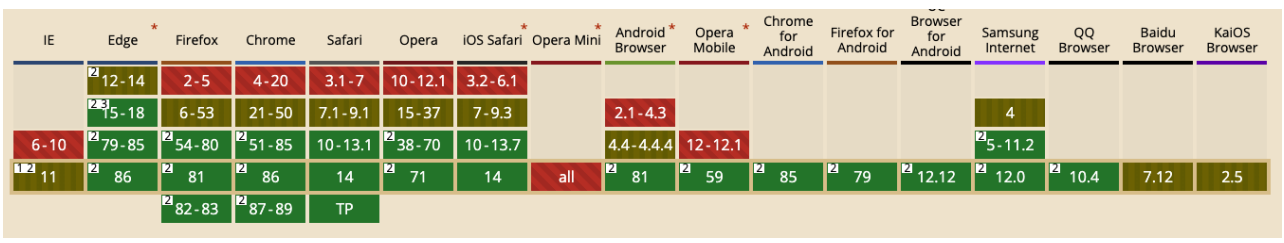
- ...
- 2009 年 ECMAScript 5.0 发布
- 2011 年 ECMAScript5.1 发布，成为 ISO 国际标准，从而推动所有浏览器都支持
- ...
- 2015 年 ECMAScript6 发布，更名为 ECMAScript 2015。
- 2016 年 ECMAScript7 发布，ECMAScript2016
- 2017 年 ECMAScript8 发布，ECMAScript2017
- 2018 年 ECMAScript9 发布，ECMAScript2018
- 2019 年 ECMAScript10，ECMAScript2019
- 2020 年 ECMAScript11，ECMAScript2020
- ....

从 2015 年开始 `tc39`委员会决定每年发布新的 ECMAScript 版本

#兼容性

下面是 ES6 兼容性状况，你也可以在 https://caniuse.com/ (opens new window)网站查看。





#学习准备

浏览器

本套课程会讲到新的 JS 知识，所以请使用`chrome` 或 `firefox` 浏览器来进行学习。

编辑器

推荐使用 `vscode` 编辑器开始学习，你可能需要安装一些插件才可以高效使用，观看后盾人发布的 vscode 视频教程将会给你带来帮助。

有些练习可以直接在浏览器的控制台完成，根据自己的需要选择窗口的排列方向。

控制台是调试代码的好地方，打开方式如下

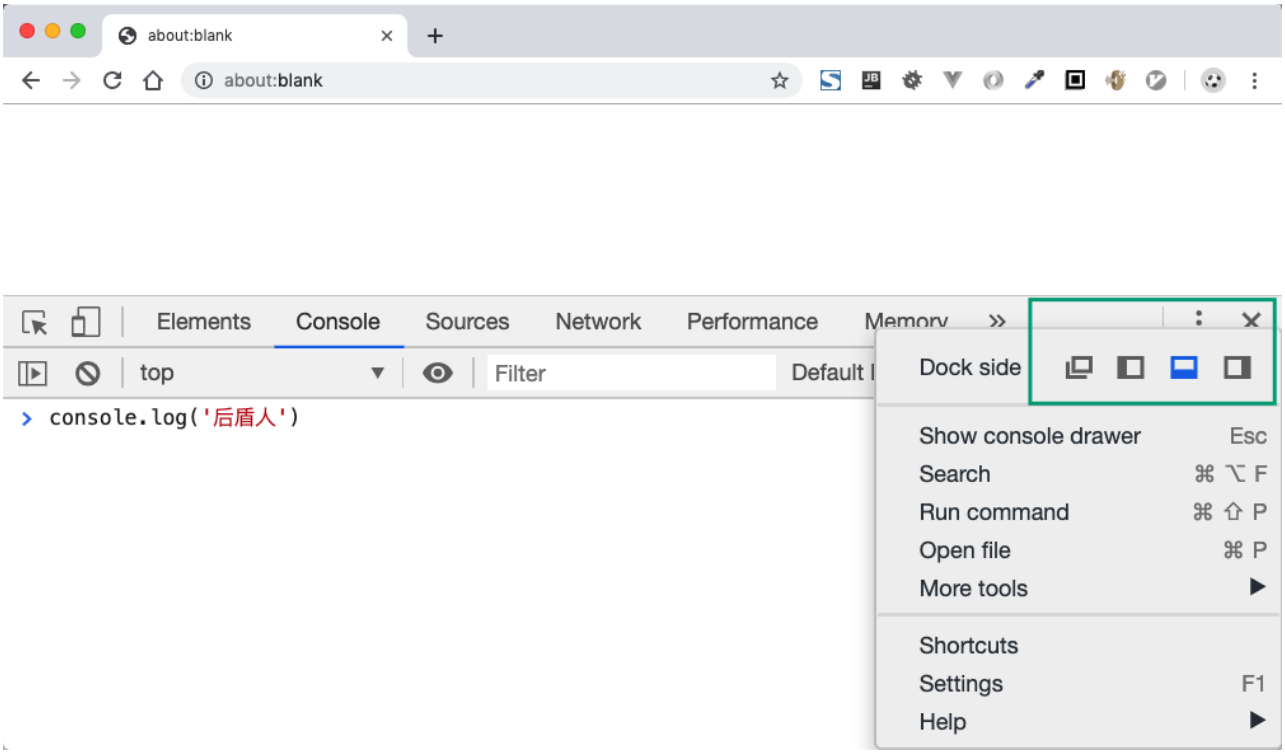
- Mac:

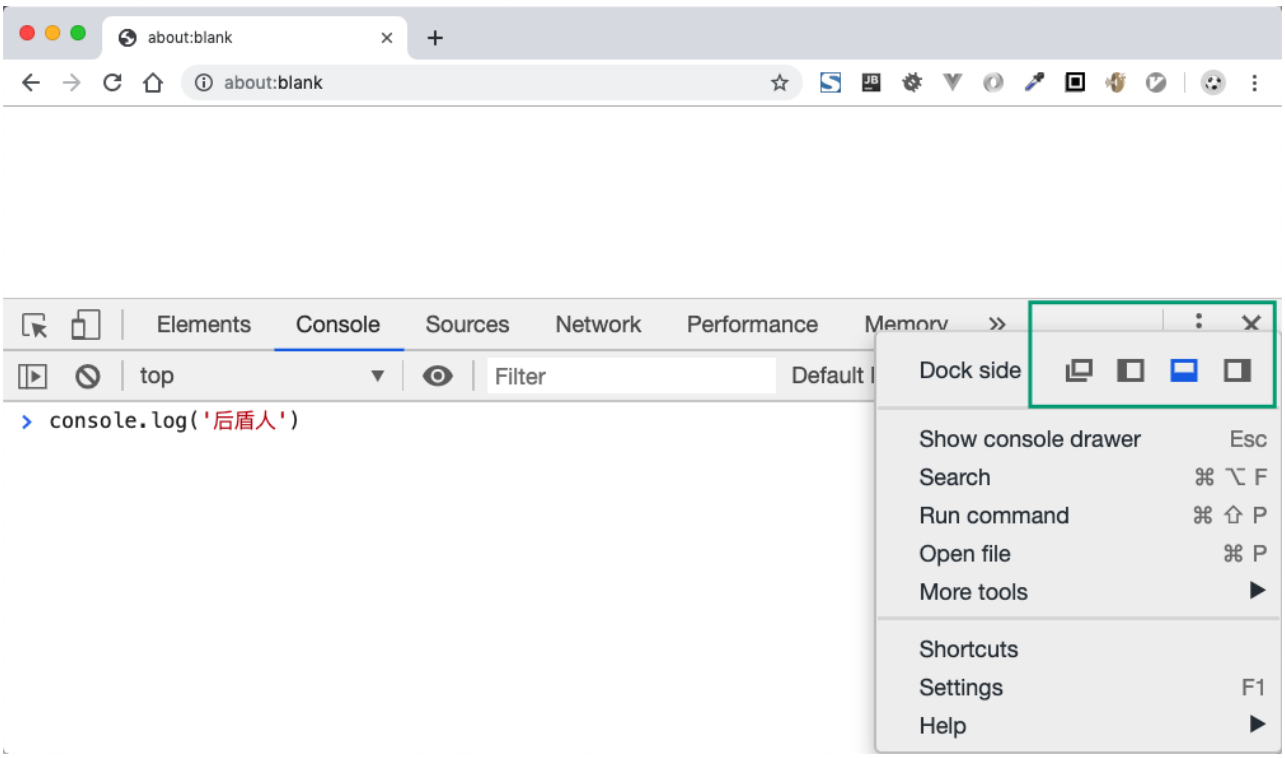
Option+Command+J

- Win/Linux:

f12 或

Ctrl+Shif+I

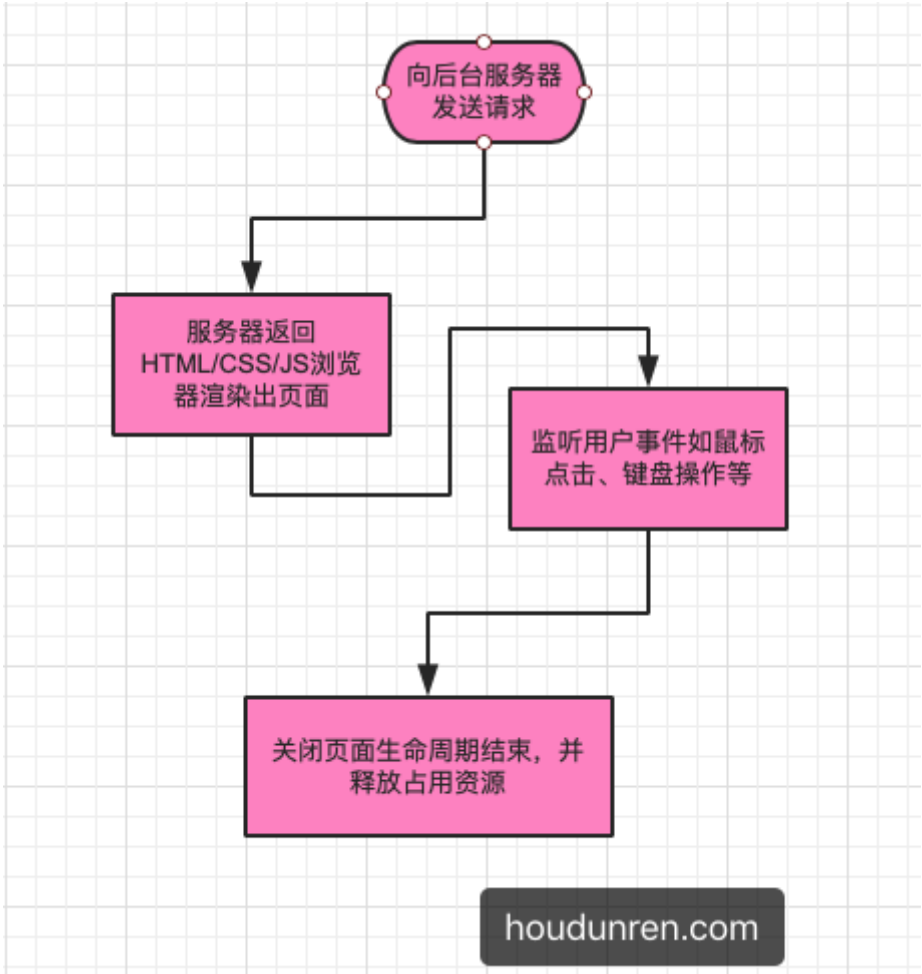


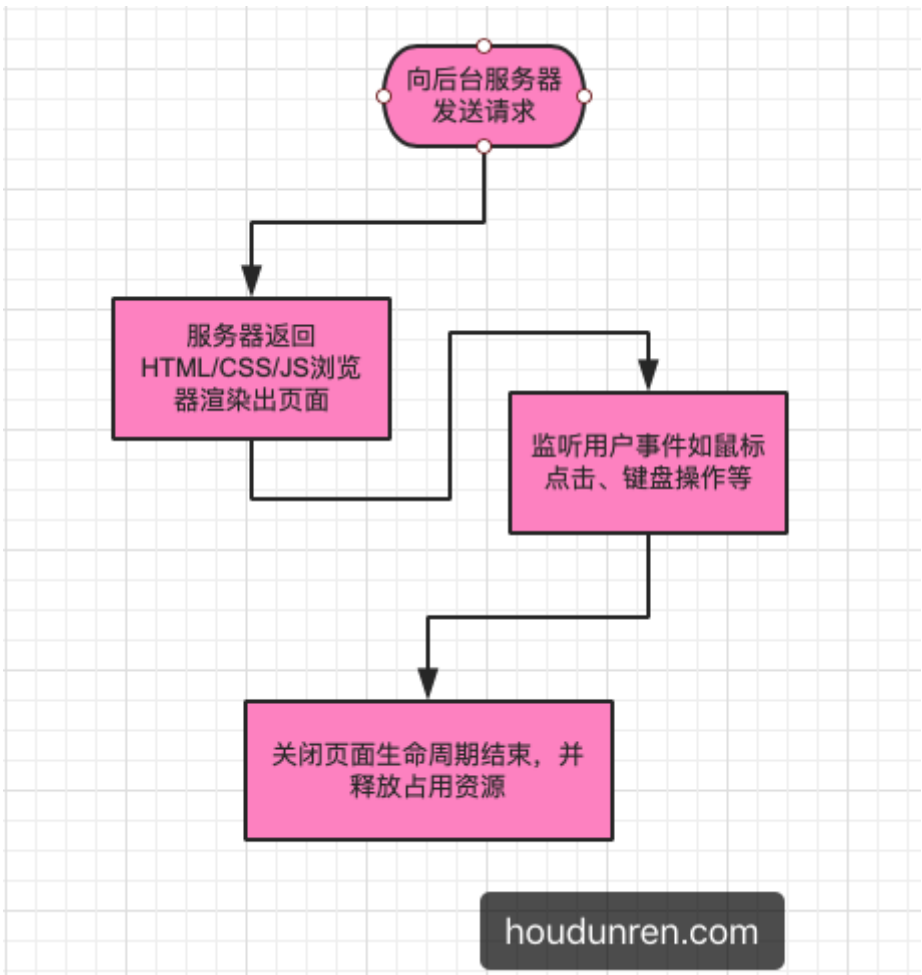


#运行流程

所有内容需要在特定的环境中运行，就像 PSD 需要在类似 PS 的软件处理一样。浏览器内置了处理的 JS 的解析器，但不同浏览器的性能不同，所以 JS 一般都在浏览器中执行，当然也有可以在服务器后台执行的 JS 解析器。

JS 请求处理步骤如下：


向后台服务器
发送请求
服务器返回
HTML/CSS/JS浏览
器渲染出页面
监听用户事件如鼠标
点击、键盘操作等
关闭页面生命周期结束，并
释放占用资源
houdunren.com


向后台服务器
发送请求
服务器返回
HTML/CSS/JS浏览
器渲染出页面
监听用户事件如鼠标
点击、键盘操作等
关闭页面生命周期结束，并
释放占用资源
houdunren.com

#脚本定义

内嵌脚本

像 style 标签一样，可以在 html 文档中使用`script`标签嵌入 javascript 代码。

```
<script>
        alert('houdunren.com');
</script>

```

外部文件

通过设置 `src` 属性引入外部 js 文件。

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

```

引入外部文件在标签体内的脚本不会执行，下面的 alert 弹窗不会执行。

```
<script src="houdunren.js">
        alert('houdunren.com');
</script>

```

#避免延迟

如果 js 放在 `<head>` 标签中要等到 js 加载并解析后才会显示`<body>`标签中的内容。

延迟体验

下面是延迟加载的示例

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>后盾人</title>
    <script src="houdunren.js">
    </script>
</head>

<body>
```

```
    <h1>hdcms.com</h1>
</body>

```

houdunren.js 内容如下

```
alert('后盾人');

```

h1 会在 houdunren.js 文件加载并解析后才会显示。

推荐做法

为了解决上面的问题，可以将 js 放在 标签前如下所示

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>后盾人</title>
</head>

<body>
    <h1>hdcms.com</h1>
    <script src="houdunren.js">
    </script>
</body>

```

#代码注释

和大部分语言使用的注释方式相仿，有单行和多行注释。

单行注释

```
<script>
  // 这是单行注释
</script>

```

多行注释

```
<script>
  /*
  这是多行注释体验
  请关注后盾人每晚直
  */
</script>

```

#自动分号

使用分号表示一段指令的结束，当没有输入分号时如果有换行符 JS 会自动添加分号，减少错误的发生。

- 但推荐每个指令都以分号结束
- 在使用构建工具时，不使用分号结束可能会造成异常

```
let stat = true;
if (stat) {
  document.write('hdcms.com');
}

```

#变量声明

#命名规则

JS 中的变量是弱类型可以保存所有类型的数据，即变量没有类型而值有类型。变量名以字母、$、_ 开始，后跟字母、数字、_。

下面都是合法的命名

```
let name = 'houdunren';
let $='hdcms'

```

JS 语言关键字不能用来做变量名，比如 `true、if、while、class` 等。

```
let class = 'houdunren';
```

```
2
```

#变量声明

可以使用多种方式定义变量比如 var、let 等（后面作用域章节会再讨论变量）。

```
let name = 'houdunren';

```

以上代码是声明和赋值的结合

```
let name ;
name = 'houdunren'

```

> 变量的其他细节使用会在函数、对象等章节中体验

使用`,`可以同时声明多个变量

```
let n = 2,f = 3;
console.log(f);

```

下面演示了变量可以更换不同类型的数据

```
let hd = 'houdunren';
console.log(typeof hd);

hd = 18;
console.log(typeof hd);

```

#弱类型

在 JS 中变量类型由所引用的值决定

```
var web = "hdcms";
console.log(typeof web); //string
web = 99;
```

```
console.log(typeof web); //number
web = {};
console.log(typeof web); //object

```

#变量提升

解析器会先解析代码，然后把声明的变量的声明提升到最前，这就叫做变量提升。
下面代码在解析过程中发现`while`不能做为变量名，没有到执行环节就出错了，这是一个很好的解析过程的体验。

```
var web = 'houdunren';
console.log(web);
let while = 'hdcms'; //Uncaught SyntaxError: Unexpected token 'while'

```

使用 `var` 声明代码会被提升到前面

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

//以上代码解析器执行过程如下
var a;
console.log(a); //1
a = 1;
console.log(a); //1

```

下面是 `if(false)` 中定义的 var 也会发生变量提升，注释掉`if` 结果会不同

```
var web = "houdunren";
function hd() {
  if (false) {
    var web = "后盾人";
  }
  console.log(web);
}
hd();
```

```
9
```

使用 `var` 定义的代码，声明会被提升到前面，赋值还在原位置

```
console.log(hd);
var hd = '后盾人';

//以上代码解析器执行过程如下
var hd;
console.log(hd); //后盾人
hd = '后盾人';

```

#TDZ

TDZ 又称暂时性死区，指变量在作用域内已经存在，但必须在`let/const`声明后才可以使用。
TDZ 可以让程序保持先声明后使用的习惯，让程序更稳定。

- 变量要先声明后使用
- 建议使用 let/const 而少使用 var

使用`let/const` 声明的变量在声明前存在临时性死区（TDZ）使用会发生错误

```
console.log(x); // Cannot access 'x' before initialization
let x = 1;

```

在`run`函数作用域中产生 TDZ，不允许变量在未声明前使用。

```
hd = "houdunren";
function run() {
  console.log(hd);
  let hd = "hdcms";
}
run();

```

下面代码 b 没有声明赋值不允许直接使用

```
function hd(a = b, b = 3) {}
```

```
hd(); //Cannot access 'b' before initialization

```

因为 a 已经赋值，所以 b 可以使用 a 变量，下面代码访问正常

```
function hd(a = 2, b = a) {}
hd();

```

#块作用域

#共同点

`var/let/const`共同点是全局作用域中定义的变量，可以在函数中使用

```
var hd = 'hdcms';
function show() {
 return hd;
}
console.log(show());

```

函数中声明的变量，只能在函数及其子函数中使用

```
function hd() {
  var web = "后盾人";

  function show() {
    console.log(web);
  }
  show(); //子函数结果: 后盾人
  console.log(web); //函数结果: 后盾人
}
hd();
console.log(web); //全局访问: hd is not defined

```

函数中声明的变量就像声明了私有领地，外部无法访问

```

```

```
var web = "hdcms.com";
function hd() {
  var web = "houdunren.com";
  console.log(web); //houdunren.com
}
hd();
console.log(web); //hdcms.com

```

#var

使用 `var` 声明的变量存在于最近的函数或全局作用域中，没有块级作用域的机制。
没有块作用域很容易污染全局，下面函数中的变量污染了全局环境

```
function run() {
  web = "houdunren";
}
run();
console.log(web); //houdunren

```

没有块作用作用域时 var 也会污染全局

```
for (var i = 0; i < 10; i++) {
  console.log(i);
}
console.log(i);

```

使用`let`有块作用域时则不会

```
let i = 100;
for (let i = 0; i < 6; i++) {
  console.log(i);
}
console.log(i);

```

下例中体验到 `var` 没有块作用域概念， `do/while` 定义的变量可以在块外部访问到

```
var num = 0;

function show() {
  var step = 10;
  do {
    var res = 0;
    console.log(num = step++);
    res = num;
  } while (step < 20);
  console.log(`结果是${res}`);
}
show();

```

`var` 全局声明的变量也存在于 `window`对象中

```
var hd = "houdunren";
console.log(window.hd); //houdunren

```

以往没有块任用时使用立即执行函数模拟块作用域

```
(function() {
  var $ = this.$ = {};
  $.web = "后盾人";
}.bind(window)());
console.log($.web);

```

有了块作用域后实现就变得简单多了

```
{
  let $ = (window.$ = {});
  $.web = "后盾人";
}
console.log($.web);

```

#let

与 `var` 声明的区别是 `let/const` 拥有块作用域，下面代码演示了块外部是无法访问到`let`声明的变量。

- 建议将

let在代码块前声明

- 用逗号分隔定义多个

`let`存在块作用域特性，变量只在块域中有效

```
if (true) {
    let web = 'hdcms',url = 'houdunren.com';
    console.log(web); //hdcms
}
console.log(web); //web is not defined

```

块内部是可以访问到上层作用域的变量

```
if (true) {
  let user = "向军大叔";
  (function() {
    if (true) {
      console.log(`这是块内访问: ${user}`);
    }
  })();
}
console.log(user);

```

每一层都是独立作用域，里层作用域可以声明外层作用域同名变量，但不会改变外层变量

```
function run() {
  hd = "houdunren";
  if (true) {
    let hd = "hdcms";
    console.log(hd); //hdcms
  }
  console.log(hd); //houdunren
}
run();
```

```
10
```

#const

使用 `const` 用来声明常量，这与其他语言差别不大，比如可以用来声明后台接口的 URI 地址。

- 常量名建议全部大写
- 只能声明一次变量
- 声明时必须同时赋值
- 不允许再次全新赋值
- 可以修改引用类型变量的值
- 拥有块、函数、全局作用域

常量不允许全新赋值举例

```
try {
  const URL = "https://www.houdunren.com"
  URL = "https://www.hdcms.com"
} catch (error) {
  throw new Error(error);
}

```

改变常量的引用类型值

```
const INFO = {
  url: "https://www.houdunren.com",
  port: '8080'
};
INFO.port = '443';
console.log(INFO);

```

下面演示了在不同作用域中可以重名定义常量

```
const NAME = '后盾人';
```

```
function show() {
  const NAME = '向军大叔';
  return NAME;
}
console.log(show());
console.log(NAME);
```

#重复定义

使用 var 可能造成不小心定义了同名变量

```
//优惠价
var price = 90;
//商品价格
var price = 100;
console.log(`商品优惠价格是:${price}`);
```

使用`let` 可以避免上面的问题，因为 let 声明后的变量不允许在同一作用域中重新声明

```
let web = 'houdunren.com';
let web = '后盾人'; //Identifier 'web' has already been declared
```

不同作用域可以重新声明

```
let web = 'houdunren.com';
if (true) {
 let web = '后盾人'; //Identifier 'web' has already been declared
}
```

但可以改变值这是与 const 不同点

```
let price = 90;
price = 88;
console.log(`商品价格是:${price}`);
```

```
4
```

`let` 全局声明的变量不存在于 `window`对象中，这与`var`声明不同

```
let hd = "hdcms";
console.log(window.hd); //undefined

```

#Object.freeze

如果冻结变量后，变量也不可以修改了，使用严格模式会报出错误。

```
"use strict"
const INFO = {
  url: '
https://www.houdunren.com
  url: '
  port: '8080'
};
Object.freeze(INFO);
INFO.port = '443'; //Cannot assign to read only property
console.log(INFO);

```

#传值与传址

基本数据类型指数值、字符串等简单数据类型，引用类型指对象数据类型。

> 类型的详细介绍会在后面章节讲解

基本类型复制是值的复制，互相不受影响。下例中将 a 变量的值赋值给 b 变量后，因为基本类型变量是独立的所以 a 的改变不会影响 b 变量的值。

```
let a = 100;
let b = a;
a = 200;
console.log(b);

```

对于引用类型来讲，变量保存的是引用对象的指针。变量间赋值时其实赋值是变量的指针，这样多个变量就引用的是同一个对象。

```
let a = {
  web: "后盾人"
};
let b = a;
a.web = "hdcms";
console.log(b);

```

#undefined

对声明但未赋值的变量返回类型为 `undefined` 表示值未定义。

```
let hd;
console.log(typeof hd);

```

对未声明的变量使用会报错，但判断类型将显示 `undefined`。

```
undefined
▶Uncaught ReferenceError: houdunren is not defined
    at 1.html:52
```

```
console.log(typeof houdunren);
console.log(houdunren);

```

我们发现未赋值与未定义的变量值都为 `undefined` ，建议声明变量设置初始值，这样就可以区分出变量状态了。

函数参数或无返回值是为`undefined`

```
function hd(web) {
  console.log(web); //undefined
  return web;
}
console.log(hd()); //undefined

```

#null

`null` 用于定义一个空对象，即如果变量要用来保存引用类型，可以在初始化时将其设置为 null

```
var hd = null;
console.log(typeof hd);

```

#严格模式

严格模式可以让我们及早发现错误，使代码更安全规范，推荐在代码中一直保持严格模式运行。

> 主流框架都采用严格模式，严格模式也是未来 JS 标准，所以建议代码使用严格模式开发

#基本差异

变量必须使用关键词声明，未声明的变量不允许赋值

```
"use strict";
url = 'houdunren.com'; //url is not defined

```

强制声明防止污染全局

```
"use strict";
function run() {
  web = "houdunren";
}
run();
console.log(web); //houdunren

```

关键词不允许做变量使用

```
"use strict";
var public = 'houdunren.com';

```

变量参数不允许重复定义

```
"use strict";
//不允许参数重名
function hd(name, name) {}

```

单独为函数设置严格模式

```
function strict(){
  "use strict";
  return "严格模式";
}
function notStrict() {
  return "正常模式";
}

```

为了在多文件合并时，防止全局设置严格模式对其他没使用严格模式文件的影响，将脚本放在一个执行函数中。

```
(function () {
  "use strict";
  url = 'houdunren.com';
})();

```

#解构差异

非严格模式可以不使用声明指令，严格模式下必须使用声明。所以建议使用 let 等声明。

```
// "use strict";
({name,url} = {name:'后盾人',url:'houdunren.com'});
console.log(name, url);

```

> 使用严格模式编码总是推荐的