

目录

Web 前端开发之 Html+CSS 精英课堂【渡一教育】	1
Html 基础标签	1
Html 高级标签	2
css 引入、css 基础选择器、选择器权重	4
css 复杂选择器、权重计算问题、css 基础属性、真题讲解	5
css 企业开发经验、盒子模型、层模型、真题讲解	8
居中五环、两栏布局、两个经典 bug、BFC、浮动	13
伪元素、包裹浮动元素、浮动应用、导航栏实例	17
文字溢出处理、背景图片、企业开发经验	20
要点补充说明、项目实战、真题讲解	21
 Web 前端开发之 JavaScript 课堂【渡一教育】	25
JavaScript 浏览器发展史	25
JavaScript 入门、引入，变量、值类型、计算运算符	25
比较运算符、逻辑运算符，条件语句、循环语句	30
条件语句补充，终止循环	40
初识引用值、typeof、类型转换	42
函数、初识作用域	46
递归、预编译、真题讲解	54
作用域、作用域链精解	73

闭包精解、立即执行函数、真题讲解	76
对象、包装类、真题讲解	85
原型、原型链、toString 和 toFixed 知识点补充、call、apply	96
继承模式、命名空间、对象枚举	110
this、callee 和 caller、笔试真题讲解	123
克隆、三目运算符	131
数组	135
类数组、封装 type 方法、数组去重	143
ES 知识点补充、复习题讲解	147
try-catch、ES5 严格模式	149
DOM 操作初探	154
DOM 选择器、节点类型和属性	162
DOM 继承树及基本操作	169
Date 对象、定时器	185
获取窗口属性、获取 DOM 尺寸、脚本化 CSS	193
事件	204
json、异步加载、时间线	228
正则表达式、真题讲解	237

Web 前端开发之 Html+CSS 精英课堂【渡一教育】

Html 基础标签

1. HTML 定义: hyperText markup language 超文本标记语言
2. 新建 HTML 文档: 文件名.html (点后缀是为了让系统更好的识别文件, 让系统知道拿什么软件来打开)
3. html 是由一对对的标签组成的语言, 每一对标签都有它的特殊功能, 例:
<html></html>, 第一个叫头标签, 第二个叫尾标签, 尾标签要加上结束符。
4. <html lang="en"></html>属于根标签, 一个 HTML 文件里只能有一对 HTML 标签, 其他所有的标签都必须写在 HTML 里边。
lang="en": 告诉搜索引擎爬虫, 我们的网站是关于什么内容的。en 是英文的, zh 是中文的, de 是德文的。
注意: 属性名和属性值都写在头标签后面 (空一格), 写法: 属性名="属性值"
5. HTML 里边分为两个标签:<head></head>和<body></body>
 <head></head>标签: 里边装思想, 编辑给浏览器的, 人看不到,
 <body></body>标签: 里边装表皮, 身体。展现给用户看的, 皮肤类的。
拓展: 在 head 标签里写以下代码, 符合搜索引擎爬虫的喜好, 可以在搜索页面把你的页面往前靠:
 <meta content="服装" name="keywords"> (关键字)
 <meta content="这是一套 xxxx" name="description"> (描述)
6. <meta charset="UTF-8"/> 单标签, 防止网页出现乱码, 网页中可以识别任何国家的语言。
7. <title> </title>标签: 里边放网页的标题 (页脚标)。
8. <p></p>标签: 段落标签, 能让标签里的内容成段展示。
9. <h1></h1>到<h6></h6>标签: 六级标题标签, 从 h1 到 h6 字号依次减小, 作用: 独占一行, 加粗字体, 更改字体大小。
10. 标签: 字体加粗
11. 标签: 斜体
如果我想让字体即加粗又斜体, 就把两个标签嵌套起来: 内容, 标签是可以嵌套的。
12. 标签: 文字中划线
13. <address></address>标签: 地址标签, 斜体, 独占一行, 基本上没有用。
14. <div></div>和标签: 容器, 分块, 让页面更加结构化, 可以绑定化

操作。div 可以独占一行，span 不可以。

Html 高级标签

1. 空格（回车）在编辑器里被称为文本分隔符。不管在编辑器里写多少个空格，在网页里只显示一个。



注意：比如你在一个 div 里边写了一堆字母，但是你会发现他到 div 边界处不换行，中文的就可以，那是因为系统认为你写的字母是一个单词，所以就不会换行，你必须加上空格才可以，空格就是英文单词分隔符。

2. ` `：（属于 HTML 编码，&开头，;结尾）空格文本。

3. `<lt;/gt;`：左/右尖角号

4. `
`单标签：回车文本

5. `<hr>`单标签：水平线

6. ``

``

``

``

``

有序列表标签，默认在网页中会以 1. 2. 3. 往下排，但在 `ol` 后边加上 `type=` “1/a/A/i/I”，他就会以指定的排序方式往下排（共五种排序方式，我写的/在笔记中代表的是或者的意思，结束符不算哈），在 `ol` 后边加上 `reversed=` “reversed”，则以倒序排列，在 `ol` 后边加上 `start=` “2”，则以第二个往下排，引号里可以只写数字，写数字几，则从第几个开始往下排。`ol` 和 `li` 基本上不用。

7. ``

``

``

``

``

无序列表标签，默认前边为小黑点，即属性为 `type=` “disc”，小方块为 `type=` “square”，小圆圈为 `type=` “circle”

ul 和 li 致力于做一些功能，有一些功能特别符合 ul 和 li 的父子结构，比如说有一个功能，这个功能由许多的功能子项来组成，每一个功能子项的功能和样式基本上都是相同的，只不过他们的内容有一些差异，很多的功能子项组成了一个功能，这样的东西我们就用 ul 和 li 来展示，因为 ul 和 li 更符合他天生的结构。

8. 单标签：图片标签

src 里边放图片路径，分为网上的 URL(超链接)、本地绝对路径(需要把地址写全)、本地相对路径(必须 html 文件和图片文件在同一目录下 ../返回上一层)

alt 图片占位符，当图片地址发生错误时展示文字信息(只有出错的时候会展示)

title 图片提示符，鼠标移到图片上时，出现提示内容

9. a 标签的作用：

(1) 超链接：

href 里边放网址，target= "_blank" 的作用是跳转后新打开一个页面，a 标签里可以放任何东西，比如你在标签里加一个 img 标签，你点图片的任何一个区域就都可以跳转了。

(2) 锚点，例如：

```
<div id= "demo1" ></div>
```

```
<div id= "demo2" ></div>
```

```
<a href= "#demo1" >find demo1</a>
```

```
<a href= "#demo2" >find demo2</a>
```

当两个 div 在一个页面中时，a 标签可以起到一个目录的作用，点击 a 标签，跳转到对应的位置。

(3) 打电话，例如：给成哥打电话

(4) 发邮件：给成哥发邮件

(5) 协议限定符(黑客)：例如：点我试试啊！来啊，你这么写的话他就会强制运行里边的 js 代码。

10. form 表单：可以发送数据，把前端数据发送给后端工程师

(1) <form method="get/post" action="http://www.baidu.com/123.php"></form>

method= "get/post" 为发送数据的方式，action 后面跟对象(发送给谁)，单纯只有 form 表单自己并不能干什么，必须配合以下组件一起使用：

(2) <input type= "text" name= "username" value= "请输入 xx">：输入框

(3) <input type= "password" name= "password" value= "请输入 xx">：密码框，value= "" 为占位符。

(4) <input type= "submit" value= "" >：登录框，value 后面跟框里的内容，默认为提交，要想数据提交成功，就必须有数据名和数据值，例如上边的两个表单，数

据名就是 name 里的 (username/password)，数据值就是你输入进去的，否则提交失败。

(5) `<input type="radio" name="aaa" value="bbb">`: 单选框，要想实现单选，一个选择题里的 name 值必须相同，这里的 name 里为数据名，value 里的为数据值。

(6) `<input type="checkbox" name="aaa" value="bbb">`: 复选框，和单选框写法相同，规则也是一样的。

注意：要想实现默认被选中，则需要在 input 后边加上 `checked="checked"` (单选复选均可)

(7) 下拉列表：

```
<select name="" >
  <option></option>
  <option></option>
</select>
```

这个里边的数据名是 name 里边的，数据值是 option 标签里边的，但是你如果在 option 标签里加上 `value=""`，则数据值就是 value 里边的。

11. `<!--备注信息-->`：作用：注释，备忘录，调错。快捷键 `ctrl+?`

12. 考点 干货：主流浏览器及其内核（背就完了）：

IE :trident Firefox :Gecko Google chrome:webkit/blink

Safari:webkit opera:presto

CSS 引入、CSS 基础选择器、选择器权重

1. css 定义：cascading style sheet 层叠样式表，css 就是装修 HTML 的。

2. css 的三种引入方式

(1) 行间样式：直接在对应的 HTML 标签后面加上属性 `style="css 样式"`

(2) 页面级 css：在 head 标签里写上 style 标签

```
<style type="text/css">
  css 样式
</style>
```

其中，`type="text/css"` 是告诉浏览器这里边放的是 css 样式（写不写这个属性都可以，但是不能写错）

(3) 外部 css 文件：新建一个 css 文档（文件名.css）然后在 head 标签里写上 `<link rel="stylesheet" type="text/css" href="">`，前两个属性是告诉浏览器这里边链接的是 css 样式，href 里边放地址，然后把 css 样式写在 css 文件里。

异步加载：代码写好上传后，我们如果要访问这个页面，浏览器就要下载他的源码，下载的是副本。浏览器是把 html 和 css 一起下载并执行的，计算机里把两件事情同时

做叫做异步加载，计算机中的同步异步和我们生活中的正好是相反的。

3. css 选择器

(1) id 选择器：给 HTML 元素后边加上属性 `id=“id 值”`，对应应在 css 里边写上 `#id 值 {css 代码}`，一个元素只能有一个 id 值，一个 id 值只能对应一个元素。

(2) class 选择器：给 HTML 元素后边加上属性 `class=“class 值”`，对应应在 css 里边写上 `.class 值 {css 代码}`，和 id 不同的是：一个元素可以有多个 class 值（写法：`class=“class 值 1 class 值 2”`），一个 class 值也可以对应多个元素。

(3) 标签选择器：在 css 里边写上 `标签名 {css 代码}`，则 HTML 里边所有的此类标签都应用此 css 样式属性（不管这个标签在哪，也不管外边被套了多少层，都能被选中）。

(4) 通配符选择器：在 css 里边写上 `*{css 代码}`，则 HTML 里边所有的标签都应用此 css 样式属性。

(5) 属性选择器：在 css 里边写上 `[id] {css 代码}` / `[class] {css 代码}`，则所有的 id/class 元素都会应用此 css 样式，如果写上 `[id=“id 名”] {css 代码}` / `[class=“class 名”] {css 代码}`，则只有被命名成这种 id/class 名的元素可以应用此样式。

4. css 权重，优先级大小排序：括号里写具体的权重值

`!important (Infinity) > 行间样式 (1000) > id (100) > class/属性/伪类 (10) > 标签/伪元素 (1) > 通配符 (0)`

(1) `!important` 写在 css 代码后，Infinity 意思是正无穷。

(2) 这里的值是 256 进制！

(3) 同级标签谁在后以谁为准。

CSS 复杂选择器、权重计算问题、CSS 基础属性、真题讲解

1. `/*内容*/`：css 注释格式

2. 父子选择器：父元素的父元素 父元素 子元素 {css 样式}，当你要选择被许多层父元素包裹着的子元素时，可以这么写（用标签、id、class 等选择器都可以），假如 `div span {css 样式}`，不管这个 span 外边套了多少层，这个时候选中了这个 div 下的所有 span，前边的 div 只是一个修饰条件，真正选择的是 div 下的 span。

3. 直接子元素选择器：父元素 > 子元素 {css 样式}，这个时候只能选择父元素下一级的子元素，不能选择父元素底下的所有子元素。

4. 浏览器遍历父子选择器的顺序是自右向左查找的，因为这么查找的话更快，效率更高。

5. 并列选择器：例如 `标签名.class 值 {css 样式}`，为了精确选择标签，可以限制多个条件，缩小范围进行选择。（中间不加空格，如果标签选择器和其他选择器并列的话就把标签选择器写在最前面。）

6. 权重的计算：同行选择器放在一起系统会把权重值相加，谁大听谁的，如果权重值相等，后边的就会覆盖前边的（必须是 css 的属性相同才会覆盖，否则不会覆盖）。只要写在同一行，不管中间加不加空格，都直接把权重值相加就行了（在 css 里，正无穷+1>正无穷）

7. 分组选择器：分组 1，分组 2，分组 3{css 样式}，这就代表三个分组共用一个 css 样式（分组里用标签、id、class 等选择器都可以，开发的时候，最好逗号后边加个回车）

8. css 代码块写在 {} 里边，写法：属性名：属性值，两个属性中间用分号隔开

9. font-size: (填数字) px;：字体大小（指的是字体的高），单位为 px，默认浏览器字体大小为 16px。

注意：下文中但凡有“（填数字）”就说明这块需要填一个具体的数字进去，但是不能加括号，记笔记的时候加了，写代码的时候别加，直接写数字。

10. font-weight: bold;：字体加粗，它的属性值有 lighter、normal（默认值）、bold、bolder（依次加粗，有时候你会发现 bolder 不好使，不是浏览器的问题，也不是代码写错了，是因为那个字体包里没有更粗的样式），或数字 100-900（数值越大字越粗，数值只能写整百数）

11. font-style: italic;：斜体，normal 归正

12. font-family: xxx;：字体，中文字体名必须加引号。

13. color: xxx;：字体颜色

颜色设置通用方法：

（1）土鳖式：如 red、green、blue（只能测试用，开发不能用）

（2）颜色代码：如 #ff0000，拆分：每两个代码代表一个颜色的饱和度值（00-ff，十六进制的数值），六个代码就分别代表了 RGB 三原色的饱和度数值，当每两位重复时，可以简写，如 #ff4400 可以写成 #f40，但是 #ff4432 就不能简写，必须保证每两位都是重复的才能简写。

（3）颜色函数：如：rgb(255, 255, 255) 这个里边的值是 0-255。

（4）透明色：transparent

14. width: (填数字) px/%;：元素的宽，单位是 px 或者 %

15. height: (填数字) px;：元素的高

16. 例如：border: 1px solid black;：元素的边框，复合属性，拆分：

（1）border-width: 1px;：边框的粗细为 1px

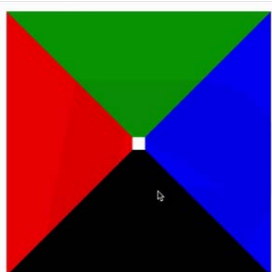
（2）border-style: solid;：边框为实体，还有两个值 dotted（点状虚线），dashed（条状虚线）

（3）border-color: black;：边框颜色为黑色

border 是四条边的框，所以还可以写成：

`border-left/right/top/bottom: 1px solid black;` 设置上下左右四条边框的属性，还可以拆成 `border-left-color: red;` 等。

其实 border 四条边是平分的，当你把边框放大之后可以看到平分线：



17. 真题讲解

练习：（百度笔试题）写一个直角三角形：

html 代码：

```
<div></div>
```

css 代码：

```
div {
  width: 0;
  height: 0;
  border: 200px solid #000;
  border-top-color: transparent;
  border-right-color: transparent;
}
```

效果：



解析：既然他们有平分线，咱就把长宽都设置成 0，他就成了四个三角形，然后把上边和右边的边框设置成透明色即可。

18. `border-radius: (填数字) %/px;`：圆角的设置

画圆方法：先画一个正方形，然后把圆角设置成 50% 即可。

19. `opacity: 0.5;`：透明度，这里的值只能是 0-1 的小数。

20. `background-color: xxx;` 背景颜色，所有的颜色值都可以用上边说的四种通用方法。

CSS 企业开发经验、盒子模型、层模型、真题讲解

1. `text-align: left;`：文字对齐方式，默认 `left`（左对齐），还有 `right`（右对齐），`center`（居中对齐）

2. `line-height: (填数字) px;`：单行文本所占高度，就是单行文字高度+间距高度，如果单行文本所占高度 = 文字高度，则没有行间距，要想单行文本垂直居中，则需要让单行文本所占高度 = 外部盒子高度(必须是单行才可以)。

3. `text-indent: 2em;`：首行缩进 2 个 `em`（空两格），`1em=1font-size`（`em` 为相对单位）。有时候人家让你设置 1.2 倍行高，就 `line-height: 1.2em` 即可。

4. `text-decoration: line-through;`：文字样式：中划线；还有一些值，`none`（无），`underline`（下划线），`overline`（上划线）

5. `cursor: pointer;`：当鼠标移上去后鼠标指针变为小手，还有一些值，`help`（问号），`e-resize/w-resize`（拖曳符）、`copy`、`move` 等

6. 伪类选择器之 `hover` 写法：选择器：`hover{css 代码}`，这个是当鼠标移到标签上之后发生的变化而设定的 `css` 样式，当鼠标移走之后他又撤出这个 `hover` 样式。

7. 元素的分类：

（1）常见的行级元素（内联元素、`inline`）：`span`、`strong`、`em`、`a`、`del` 等，特点：内容决定元素所占位置，不独占一行，不可以通过 `css` 改宽高。宽=内部元素的宽，高=内部元素的高。

（2）常见的块级元素（`block`）：`div`、`p`、`ul`、`li`、`ol`、`form`、`address` 等，特点：独占一行；可以通过 `css` 改宽高。在没有设置宽高时，宽=100%，高=内部元素的高。

（3）行级块元素（`inline-block`）：`img` 等，特点：内容决定大小，不独占一行，可以改宽高。在没有设置宽高时，宽=内部元素的宽，高=内部元素的高。

备注：图片可以只设置一个宽，高就会等比缩放，也可以只设置一个高，宽就会等比缩放。

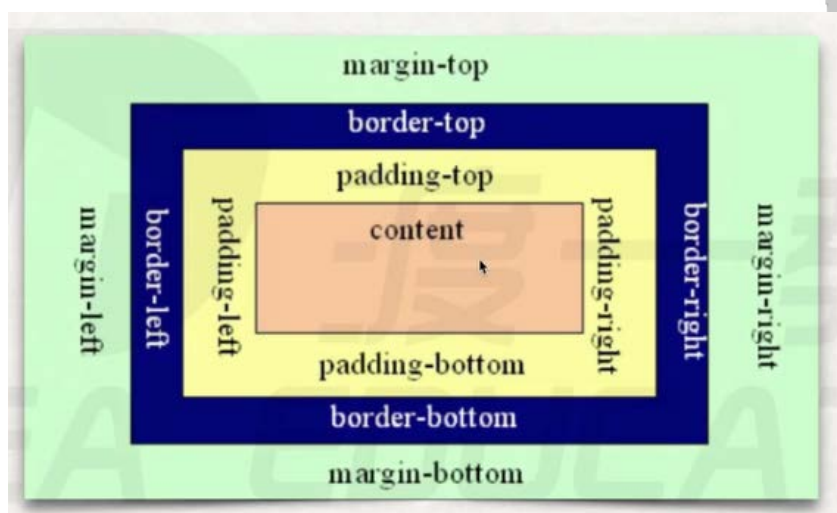
注意：

（1）不管是行级元素、块级元素还是行级块元素，它的 `css` 样式默认就有个 `display: inline/block/inline-block`，他的意思是展示为 `xx` 级别的元素，所以即使被定义了，也可以通过 `display: inline/block/inline-block` 转换为其他级别的元素



（2）凡是带有 `inline` 的元素（文本类属性元素），都有文字特性，比如写四张图片，发现中间有一条缝，因为他会把回车当成文本分隔符，解决这个 `bug` 的正确方法就是写代码的时候两个 `img` 中间不要加空格也不要按回车，紧挨着即可。

8. 团队开发高效方法: 先定义功能, 后引入功能(先写 css, 后在 HTML 里边引入加 class, 在公司开发的时候, 前辈们会把功能写成一个 css 库, 别人用的时候直接引入过来, 然后加上 class 名就可以了, 这么做的话更高效)
9. 初始化标签 (自定义标签): 很多标签最开始的时候并不是我们想要的那种样式, 可以通过标签选择器给他设定一些默认的风格, 如: 去掉无序列表前的小黑点, 可以这么写: `ul{list-style:none;}`, 去掉 a 标签下划线, 颜色归正: `a{text-decoration:none; color:#424242;}`, 还有每个网页在刚开始的时候都有一个内边距和外边距 (下边有讲到), 我们并不希望他天生就有边距, 就给他用通配符初始化一下: `*{padding:0; margin:0;}`, 这个时候用通配符是最好的, 因为他的范围广, 并且他的权重值是最低的, 后边你要改的话也不影响啥。其实通配符也就只在初始化的时候用。
10. 盒子模型: `margin+border+padding+ (content=width+height)`



- (1) 外边距: margin (容器与容器之间的距离)
 - (2) 盒子壁: border
 - (3) 内边距: padding (容器与内部内容之间的距离, 内边距是不能放内容的, 并且会把盒子撑开)
 - (4) 盒子内容: width+height
11. padding: (填数字) px; 在这里 padding 一个值代表了上下左右四个值, padding: (填数字) px (填数字) px (填数字) px (填数字) px; 四个值代表的顺序依次是上、右、下、左的 padding 值, 三个值是上、左右、下, 两个值是上下、左右。也可以分开设置: padding-left/right/top/bottom: (填数字) px, margin 和 padding 是一样的, border-width 也可以设置四个值。

12. 真题讲解

练习 1: 外边写一个盒子, 里边是内容区, 让内容区水平垂直居中:

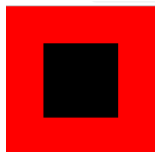
html 代码:

```
<div class="wrapper">
  <div class="content"></div>
</div>
```

css 代码:

```
.wrapper {
  width: 100px;
  height: 100px;
  background-color: red;
  padding: 50px;
}
.content {
  width: 100px;
  height: 100px;
  background-color: #000;
}
```

效果:



解析: 直接让外层和里层大小一样, 然后给外层加上一个 padding 值即可 (保证上下和左右值相等), 他外部的盒子就被撑开了, 里边的就居中了。

练习 2: 求出下列盒子的可视化宽高:

```
div {
  width: 100px;
  height: 100px;
  background-color: red;
  border: 10px solid black;
  padding: 10px 20px 30px;
  margin: 10px 20px;
}
```

解析: 宽 160px, 高 160px, 可视化宽高不包括 margin 值, 宽等于 width 的 100+border 的左右各 10+padding 的左右各 20, 高等于 height 的 100+border 的上下各 10+padding 的上 10 和下 30。

练习 3: (百度笔试题) 求出下列盒子的可视化宽高:

```
#my-defined {
  width: 100px;
  height: 100px;
  padding: 0 100px;
  margin: 10px 20px 30px 40px;
  border: 1px solid orange;
  background-color: orange;
}
```

解析：宽 302px，高 102px，padding 的上下为 0，左右各 100，那个 2 是 border 两边的值。

练习 4：写一个远视图：

html 代码：

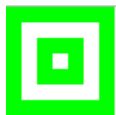
```
<div class="wrapper">
  <div class="box">
    <div class="content">
      <div class="content1"></div>
    </div>
  </div>
</div>
```

css 代码：

```
*{
  margin: 0;
  padding: 0;
}
.content1 {
  width: 10px;
  height: 10px;
  background-color: #fff;
}
.content {
  width: 10px;
  height: 10px;
  padding: 10px;
  background-color: #0f0;
```

```
}  
.box {  
  width: 30px;  
  height: 30px;  
  padding: 10px;  
  background-color: #fff;  
}  
.wrapper {  
  width: 50px;  
  height: 50px;  
  padding: 10px;  
  background-color: #0f0;  
}
```

效果:



解析：这个只要每一层的父元素等于子元素的可视化宽高，然后父元素的 padding 设置成 10 像素即可。

12. body 在默认情况下会有 8px 的 margin 值（考点）

13. 层模型（定位）

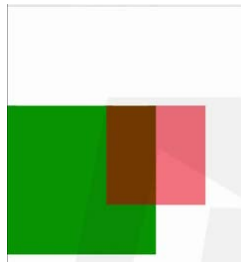
（1）绝对定位：position: absolute; top: (填数字) px; left: (填数字) px; 当你希望某一元素出现在特定位置的时候，可以使用绝对定位，然后配合着 top、left、right、bottom 一起用（距离某一方向多少像素，但是 left 和 right 不能同时出现，并且 left 是左边线距离浏览器多少像素，right 是右边线距离浏览器多少像素，top 是上边线距离浏览器多少像素，一般不用 bottom），但是，他会脱离原来的位置进行定位（会向上一层，底下的位置会空出来，其他的元素顶上去），它是相对于最近的有定位的父级进行定位，如果没有，那么相对于文档进行定位。



如图：脱离原来的层级，向上一层，其他元素顶上去，两个元素在不同层面，所以有重叠部分

（2）相对定位：position: relative; top: (填数字) px; left: (填数字) px; 当

你希望某一元素出现在特定位置的时候，可以使用相对定位，然后配合着 top、left、right、bottom 一起用，但是，**他会保留原来的位置进行定位（原来的位置还占着，但是没东西），它是相对于自己原来的位置进行定位。**



如图，他还是两层，但是原来的位置他还站着呢，其他的元素不会往上窜。

(3) 定位原则：用 relative 作为参照物（保留原来位置进行定位，如果不设置 top 和 left，对后续元素都没有影响），用 absolute 定位（可以任意调换自己的参照物，定位更灵活），这种原则被称为子绝父相。

(4) 固定定位：position: fixed; top: (填数字) px; left: (填数字) px; 当你希望某一元素不随着滚动条的拖曳而改变的时候，可以使用固定定位，然后配合着 top、left、right、bottom 一起用，**无论滚动条怎么拖曳，他都定在那里。**

注意：要想元素居中，先可以给他定位（绝对或者固定），然后把 left 和 top 值都设定为 50%，然后把 margin-top 和 margin-left 的值设定为负的元素的一半身位即可，比如：

```
div{
  position: absolute;
  left: 50%;
  top: 50%;
  width: 100px;
  height: 100px;
  background-color:red;
  margin-left: -50px;
  margin-top: -50px;
}
```

(5) z-index: 1;; 提高层级，值越大越往上，**只对 position 管用。**

居中五环、两栏布局、两个经典 bug、BFC、浮动

1. 居中五环：

要求：写一个五环，并且让他在浏览器水平垂直居中。

html 代码:

```
<div class="plat">
  <div class="circle1"></div>
  <div class="circle2"></div>
  <div class="circle3"></div>
  <div class="circle4"></div>
  <div class="circle5"></div>
</div>
```

css 代码:

```
*{
  margin: 0;
  padding: 0;
}
.plat {
  position: absolute;
  left: 50%;
  top: 50%;
  margin-left: -190px;
  margin-top: -93px;
  height: 186px;
  width: 380px;
}
.circle1,
.circle2,
.circle3,
.circle4,
.circle5 {
  width: 100px;
  height: 100px;
  border: 10px solid black;
  border-radius: 50%;
  position: absolute;
}
.circle1 {
```

```

border-color: red;
left: 0;
top: 0;
}
.circle2 {
border-color: green;
left: 130px;
top: 0;
z-index: 3;
}
.circle3 {
border-color: yellow;
left: 260px;
top: 0;
}
.circle4 {
border-color: blue;
left: 65px;
top: 70px;
}
.circle5 {
border-color: purple;
left: 195px;
top: 70px;
}

```



效果：

解析：以前我们讲过画圆的方法，所以这个比较简单，直接用定位写就可以了，这里的父级元素本来应该是 relative，但是由于父级也要居中，所以也要定位，所以用的是 absolute。

2. 两栏布局

要求：实现两栏布局，一个宽度固定，一个自适应。

html 代码：

```

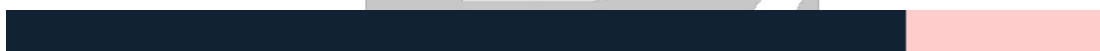
<div class="right"></div>
<div class="left"></div>

```

css 代码:

```
*{
    margin: 0;
    padding: 0;
}
.right {
    width: 100px;
    height: 100px;
    background-color: #fcc;
    position: absolute;
    right: 0;
}
.left {
    height: 100px;
    background-color: #123;
    margin-right: -100px;
}
```

效果:



解析: 这个比较简单, 先写一个需要定位的 div right, 把他的宽高固定, 再写一个 div left, 不设置宽就是自适应, 然后给第一个 right 设置绝对定位, left 就上去了, 然后给 left 设置 margin-right, 因为两个有重合, 设置之后让 right 下面没有东西, 才是真正的两栏布局。注意: 在 html 里, 两个顺序不能颠倒, 假如颠倒的话, right 在第二行, 定位后 left 就窜不下去。

3.margin 塌陷 bug: 垂直方向的 margin 父子元素是结合在一起的, 取最大的一个值, 就是父级元素包裹了一个子元素, 你给父级元素加了一个 margin-top 值, 给子元素也加了一个 margin-top 值, 但是你会发现, 当子元素的值小于父元素的值时, 子元素的值不起作用, 当子元素的值大于父元素的值时, 子元素会和父元素一起动。弥补办法: 给父元素加一个顶棚: border-left: 1px solid red; 这是一个土方法, 开发的时候不能用, 真正解决这个问题的办法就是给父级元素触发 BFC (BFC 后期会讲), 方法:

- (1) position: absolute;
- (2) display: inline-block;
- (3) float: left/right;
- (4) overflow: hidden; (溢出部分隐藏)

以上四种方法都能触发 BFC, 但也会引发新的问题, 在开发的时候, 因情况而定, 哪个

合适用哪个。

4. margin 合并 bug: 两个兄弟级别的元素垂直方向上的 margin 会合并，就是上边的你设置一个 margin-bottom 值，下边的设定一个 margin-top 值，他们只会让一个值起作用（谁大听谁的，一样大的话只让一个值起作用），弥补方法（不能用）：建一个父级元素 div，包裹兄弟级别的一个元素，或者建两个父级 div，把他俩都包裹起来，然后给父级元素触发 BFC。但是在前端开发的时候，由于 HTML 是框架，不能随便就改变，否则会对 css 和 JavaScript 有影响，所以我们允许这个 bug 存在，选择不解决，但是我们可以通过计算来弥补（增大一个元素的 margin 值）

5. 浮动: `float: left/right`; 这个属性可以让元素排队，原本的块级元素是独占一行的，但是加上浮动之后他可以在一行排队，左浮是从左往右排，右浮是从右往左排，并且可以加 margin 值等等，排队的边界就是父级的边界，到了边界后就会另起一行继续排。

浮动元素产生了浮动流，所有产生了浮动流的元素，块级元素看不到他们，产生了 bfc 的元素和文本类属性的元素（inline）以及文本都能看到浮动元素，如下图。



你会发现，浮动元素的父级的宽在没有设置的情况下显示不了，因为父级是块级元素，块级元素看不到浮动元素，所以包不住子集。清除浮动流的方法：在父级里边的最后写一个块级元素，但是即使没有设置浮动，也会受浮动流的影响。所以给这个受浮动流影响的元素设置 `clear: both`，设置了之后就会清除浮动流，把浮动元素的父级给托开，父级的宽就会显示出来了，他自己也就下来了。注意设置清除浮动的元素必须是块级元素，这个办法可以实现，但是不能用哈，因为 HTML 是框架，不能随便更改。真正解决的办法在下边。

伪元素、包裹浮动元素、浮动应用、导航栏实例

1. 伪元素: 伪元素可以当元素来操作，也可以设置 css 样式，当做正常元素使用，是行级元素，但是他没有 html 结构。

每一个 HTML 元素里天生都会自带两个伪元素（前后各一个），我们可以通过 css 来选中它们从而设置样式: `标签名:: before/after{css 样式}, content: “内容”`; 可以给前后的伪元素加内容，他的内容天生就存在于这个标签里，即使里边没有内容，也必须加上 content，写成 `content: “”`;，content 就是修改伪元素的内容的（content 也只能用于伪元素）。

2. 真正解决包裹浮动元素的方法:

(1) 把父级元素后边的伪元素调用出来 `content="";`，然后转为块级元素，再清除浮动。比如：

```
div::after {  
    content: "";  
    display: block;  
    clear: both;  
}
```

(2) 给父级元素触发 bfc

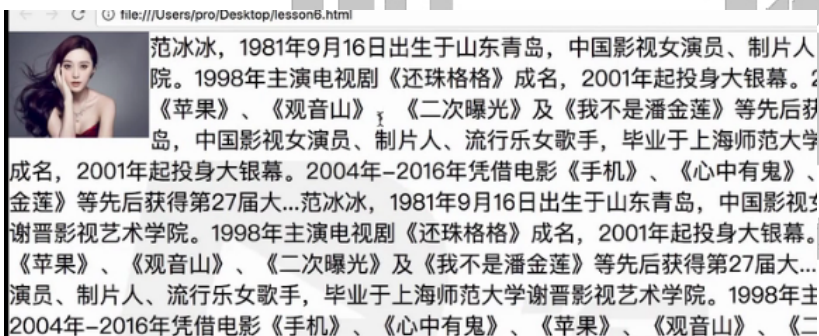
注意：凡是设置了 `float: left/right` 或者 `position: absolute` 的元素内部会自动把他转化为行级块元素 (`inline-block`)。

3. 浮动应用

其实最开始提出浮动的时候是为了实现报纸布局的，就是文字环绕图片，正常如果不给图片加浮动的话是这样的：



你会发现文字和图片的底是对齐的，这时候你给图片加上浮动：



此时就实现了文字环绕图片的效果，这是最初浮动的用法，后来人们研究了一系列复杂的用法。

4. 导航栏实例：

我们来写一个淘宝的一小部分导航栏：

html 代码：

```
<ul class="nav">  
    <li class="list-item"><a href="#">天猫</a></li>
```



```
<li class="list-item"><a href="#">聚划算</a></li>
<li class="list-item"><a href="#">天猫超市</a></li>
</ul>
```

css 代码:

```
*{
    margin: 0;
    padding: 0;
    color: #424242;
    font-family: arial;
}
a {
    text-decoration: none;
}
.nav {
    list-style: none;
}
.nav::after {
    content: "";
    display: block;
    clear: both;
}
.nav .list-item {
    float: left;
    margin: 0 10px;
    height: 30px;
    line-height: 30px;
}
.nav .list-item a {
    color: #f40;
    font-style: bold;
    height: 30px;
    display: inline-block;
    border-radius: 15px;
    padding: 0 5px;
```

```
}  
.nav .list-item a:hover {  
    color: #fff;  
    background-color: #f40;  
}
```

效果：官网上有

解析：这都是利用以前的知识点写出来的，当鼠标移上去之后，你会发现他是圆角的矩形，而且两边有距离，我们直接加 padding 就可以了，每次设置浮动后，最好给父级伪元素清除浮动，不影响后续内容，尽量把功能写全。

文字溢出处理、背景图片、企业开发经验

1. 文字溢出容器处理

(1) 单行文本：三件套：white-space: nowrap; (禁止换行), overflow: hidden; (溢出部分隐藏), text-overflow: ellipsis; (文字打点) (当文字的宽度小于容器的宽度时，没有变化，当文字的宽度大于容器的宽度时就打点显示)

(2) 多行文本只做截断，不做打点 (打点是手写上去的)，先让行高和容器的高成比显示 (比如容器 40px, line-height 是 20px, 你也就只能放两行，所以得把数字算好)，然后溢出部分隐藏处理。

2. 背景图片

(1) background-image: url (路径);: 引入背景图片

(2) background-size: (填数字) px/% (填数字) px/%;: 背景图片的大小 (宽、高)

(3) background-repeat: no-repeat: 图片不重复 (禁止平铺)，默认为 repeat，就是当图片铺不满容器的时候，他会重复的铺满容器，no-repeat 就是禁止平铺，铺不满就铺不满。还有两个值 repeat-x (x 轴平铺)、repeat-y (y 轴平铺)。一般我们用的是 no-repeat。

(4) background-position: (填数字) px (填数字) px; 图片的位置 xy, (在 css 里, x 轴右为正, y 轴下为正), 两个值也可以用 left/right/top/bottom/center 或者百分数来使用, 例如, left top 即为左上方, 50% 50% 就是正中间, 即 center center。

3. 企业开发经验

(1) 图片替代文字

当网速不好的时候，图片加载不出来 (系统会把 css 和 js 屏蔽掉)，就得用文字代替，当网速好的时候显示图片，隐藏文字，这就需要在 HTML 里边加上文字信息，然后 (只能用于背景图片)

方法一：text-indent: (填数字) px; (文字首行缩进，值要大于容器的宽)，然后 white-space: nowrap; (禁止换行)，再 overflow: hidden; (文字溢出部分隐藏) 即可。

方法二：背景上是可以展示背景图片和背景颜色的，但是不能展示内容，那么就给容器 height: 0px; (先把容器的高清零)，然后再用 padding-top: (填数字) px; padding 就会把图片撑开，这时图片就显示出来了，文字就顶出去了，再 overflow: hidden; (文字溢出部分隐藏) 即可。

(2) 规定：行级元素只能套行级元素，块级元素可以套任何元素，但是，p 标签里不能套 div，否则 p 会被砍断，a 标签里不能套 a 标签。

要点补充说明、项目实战、真题讲解

1. 一个大 div 嵌套一个小的 div，让小的 div 水平居中，并且小的 div 不随着浏览器的窗口的大小的改变而改变：先给大的 div 设置定高，不要设置宽，再给小的 div 设置高，高和大的 div 高度相等，然后设置定宽，最后在小的 div 设置 margin: 0 auto; 即可 (auto 意思是自适应)。比如：

html 代码：

```
<div class="wrapper">
  <div class="box"></div>
</div>
```

css 代码：

```
*{
  margin: 0;
  padding: 0;
}

.wrapper {
  height: 100px;
  background-color: #123;
}

.box {
  width: 1200px;
  height: 100px;
  background-color: #00f;
  margin: 0 auto;
}
```

效果:

当窗口缩小的时候,只有黑的会变短,蓝的永远不变而且处于居中状态。这就是利用 `margin: 0 auto` 来实现的,但是两个必须都得是块级元素。

2. 行级元素和行级块元素都是文本类元素 (带有 `inline` 的元素), 文本类元素会受到文本分隔符的影响 (只要有空格或者回车都会空一格, 就是之前讲的, 图片中间有空隙, 和这个是一样的)

3. 文本类元素和文本类元素会底对齐, 但是一旦文本类元素里边包含文字了, 外边的文字就会和里边的文字底对齐, 可以用 `vertical-align: (填数字) px` 调整对齐, 他还有一个默认的值 `middle` 是对齐。



4. 行级元素的 `padding-top`、`padding-bottom`、`margin-top`、`margin-bottom` 属性设置是无效的。

5. 项目实战

这个项目我没有素材, 就直接截图哈, 将就看一下:

`<div>姬教授贴吧</div>`

```

1  *{
2    margin:0;
3    padding:0;
4  }
5
6
7
8  div{
9    padding:10px 10px;
10   width:200px;
11   line-height: 12px;
12   height:12px;
13   font-size:12px;
14   background:linear-gradient(to
15   color:rgba(255,255,255,0.8);
16 }
17

```

```

div::before{
  content:"";
  display: inline-block;
  width:12px;
  height:11px;
  background-image:url(data:imag
  background-size:100% 100%;
  margin-right:5px;
  /*vertical-align: -1px;*/
}
div::after{
  content:"";
  display: inline-block;
  background-size:100% 100%;
  width:6.5px;
  height:11.5px;
  float:right;
  background-image:url(data:imag
}

```



解析: 效果就是上边展示的, 前后两个图标使用伪元素来实现的, 然后居中的话先让内容高度盒子高度, 然后给盒子上下加上 `padding` 值就居中了。

6. 真题讲解

练习 1: (阿里笔试题) 内核为 webkit 的浏览器包括:

A: IE B: Firefox C: Chrome D: Safari E: opera7 (较老版本)

解析: 选 CD, opera7 的较老版本是最经典的版本, 新版的 opera 用的也是 webkit。

练习 2: (阿里笔试题) 下列哪些属于 HTML 单标签?

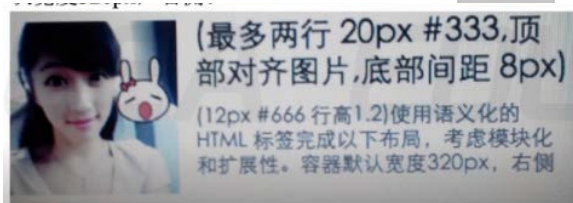
A:
 B: <hr> C: <command> D: <meta>

解析: 选择 ABD, C 是 h5 的标签, 后期会讲的。

练习 3: (阿里笔试题) 使用语义化的 HTML 标签及 css 完成以下布局:

{最多两行 20px #333, 顶部对齐图片, 底部间距 8px}

{12px #666 行高 1.2} 使用语义化的标签完成以下布局, 考虑模块化和扩展性。容器默认宽度 320px, 右侧。



容器默认宽度 320px, 图片 100*100

hover 时容器宽度变成 400px

html 代码:

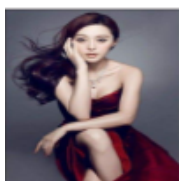
```
<div class="wrapper">
  
  <p class="content1">{最多两行 20px #333, 顶部对齐图片, 底部间距 8px}</p>
  <p class="content2">{12px #666 行高 1.2} 使用语义化的标签完成以下布局, 考虑模块化和扩展性。容器默认宽度 320px, 右侧。</p>
</div>
```

css 代码:

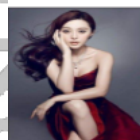
```
*{
  margin: 0;
  padding: 0;
}
.wrapper {
  width: 320px;
}
.wrapper:hover {
  width: 400px;
```

```
}  
.wrapper img {  
    width: 100px;  
    height: 100px;  
    float: left;  
}  
.content1 {  
    font-size: 20px;  
    line-height: 20px;  
    height: 40px;  
    overflow: hidden;  
    color: #333;  
    margin-bottom: 8px;  
}  
.content2 {  
    font-size: 12px;  
    color: #666;  
    line-height: 1.2em;  
}
```

效果：



{最多两行20px #333，
顶部对齐图片，底部间距
(12px #666 行高1.2)使用语义化的标签
完成以下布局，考虑模块化和扩展性。容
器默认宽度320px，右侧。



{最多两行20px #333，顶部对齐
图片，底部间距8px}
(12px #666 行高1.2)使用语义化的标签完成以下布局，
考虑模块化和扩展性。容器默认宽度320px，右侧。

解析：第二张图片是 hover 之后的，题其实并不难，但是当时很多人没有读懂题意，题上的上边两行既是要求又是内容，直接按照要求写下来就完事儿了。

Web 前端开发之 JavaScript 课堂【渡一教育】

JavaScript 浏览器发展史

1. web 发展史: Mosaic 是互联网历史上第一个获普遍使用和能够显示图片的网页浏览器, 于 1993 年问世。1994 年 4 月, 马克安德森和 Silicon Graphics (简称为 SGI, 中译为“视算科技”或“硅图”) 公司的创始人吉姆 克拉克 (Jim Clark) 在美国加州设立了 “Mosaic Communication Corporation”。Mosaic 公司成立后, 由于伊利诺伊大学拥有 Mosaic 的商标权, 且伊利诺伊大学已将技术转让给 Spy Glass 公司, 开发团队必须彻底重新撰写浏览器程式码, 且浏览器名称更改为 Netscape Navigator, 公司名字于 1994 年 11 月改名为 Netscape Communication Corporation, 此后沿用至今, 中译为“网景”。微软的 Internet Explorer 及 Mozilla Firefox 等, 其早期版本皆以 Mosaic 为基础而开发。微软随后买下 Spy Glass 公司的技术开发出 Internet Explorer 浏览器, 而 Mozilla Firefox 则是网景通讯家开放源代码后所衍生出的版本。

2. javascript 历史: Javascript 作为 Netscape Navigator 浏览器的一部分首次出现在 1996 年。它最初的设计目标是改善网页的用户体验。作者是 Brendan Eich。期初 javascript 被命名为 Livescript, 后因和 Sun 公司合作, 因市场宣传需要改名 javascript。后来 Sun 公司被 Oracle 收购, javascript 版权归 Oracle 所有。

3. 浏览器组成:

(1) shell 部分

(2) 内核部分: 分为渲染引擎、js 引擎和其他模块。

4. JavaScript 的逼格:

(1) 属于解释性语言, 解释一行执行一行。(与编译性语言不同, 编译性语言是通篇翻译然后执行)

(2) 单线程

5. js 执行队列: 轮转时间片

6. js 三大部分: ECMAScript、DOM、BOM

JavaScript 入门、引入, 变量、值类型、计算运算符

1. javascript 引入:

(1) 页面级 js: 直接在 HTML 文件里写上 script 标签, 写在哪里都可以。

```
<script type= “text/javascript” >
```

js 代码

```
</script>
```

其中，`type="text/javascript"` 是告诉浏览器这里边放的是 js 代码。

(2)外部 js 文件:新建一个 js 文件(文件名.js),然后在 HTML 里边写上`<script type="text/javascript" src=" " >`, src 里边放文件地址,把 js 代码写在外部 js 文件里。

2. `document.write("内容");`:这句 js 代码的作用是在浏览器中打印出你想要的内容。

3. 变量:

(1) 作用: 存放数据, 我们会把一筐数据放到一些变量里, 以便于后续使用。

(2) 声明变量: `var 变量名;`, 例如: `var a;` 这句话的意思是: 我们向程序申请了一个变量的房间, 这个房间的名字叫做 a, 说通俗一点, 就是我们想向系统要一个名字为 a 的筐, 往这个筐里放东西, 这就叫声明变量。

(3) 变量赋值: 例如 `var a;`, 如果你想给 a 这个筐里边放东西的话, 可以这么写, 例如 `a=100`, 这样就把 100 放进了 a 筐里, 注意这里的 "=" 不能叫等号, 叫赋值符号, 把 100 这个值赋给了 a, 这就叫变量赋值。声明变量和变量赋值两个过程是分开的, 没必要非得连在一起。当然也可以写在一起: `var a=100;`, 这是一个简写, 其实这里边包含了两个过程: 声明变量和变量赋值。如果我们想把 100 打印出来, 则 `document.write(a);` 即可。(注意: `document.write()` 括号里如果是变量名的话, 不需要加引号, 如果是字符串的话, 则需要加引号。)

(4) 单一 var 模式: 如果需要同时申请多个变量, 可以这么写 `var a,b,c;` 中间用逗号隔开 (不能用分号, 分号是 js 每一句结尾用的), 也可以赋值: `var a=1,b=2,c=3;` 打印的时候也要用逗号隔开 `document.write(a,b,c);` 开发的时候, 为了规范书写, 需要在逗号后边加回车, 让多个变量多行展示。

(5) 一个变量里只能存放一个值, 如果给变量赋了一个新的值, 他原来的值就不存在了, 例如 `var a=10; a=20;` 这个时候 a 的值是 20。

(6) 变量命名规则: 变量名必须以英文字母或 _ 或 \$ 开头; 变量名可以包括英文字母或 _ 或 \$ 或数字; 不可以用系统的关键字 (有特殊语法含义的单词)、保留字 (未来系统可能用它作为关键字的单词) 作为变量名。

(7) 变量是可以运算的, 例如: `var a=10; var b=20; var c; c=a+b; document.write(c);` 此时打印出来 30, 在计算机里, 运算的优先级大于赋值的优先级, 也就是先把等号右边的 a+b 算出来后在赋给 c 的。

4. 值类型: 值类型分为原始值和引用值 (js 属于解释性语言, 声明变量只有一个 var, 而且 var 里边可以放入任何值类型, js 里的值类型是由值来决定的, 不是由变量决定的)

(1) 原始值分为 `Number, String, Boolean, undefined, null`

`Number`: 数字类型, 例如 `var a=1;` 或 `var a=-123.321` 都可以。

String: 字符串类型，我们有的时候需要把一些中文、数字或者英文等（字符串）输入到里边，则必须加上引号 `var a="内容"`，同样的，字符串也可以 `document.write(a)` 打印出来。（双引号里边的都是字符串）

Boolean: 这里边的值只有两个，分别是 `var a=true` 和 `var a=false`，打印出来也是 `true` 或者 `false`，它是表示逻辑的词，以后会大有用处。

undefined: 这里边的值只有一个，`var a=undefined`，打印出来也是他，当你声明一个变量并且没有赋值直接打印的话，出来的就是 `undefined`，你也可以给变量赋值为 `undefined`。

null: `null` 表示占位，`var a=null`，打印出来也是 `null`，代表空的意思，空值，假如以后你写了一个方法，又不想要了，又删除不了，这时你可以用空值覆盖。

（2）引用值分为 `array, object, function, date, RegExp` 等，引用值后续会详细讲解，这里拿 `array` 举个例子。

array: 数组，相当于一个筐里有很多小格，可以放入很多东西，例如 `var arr=[1, 2, 3, true, "abc"]`，打印出来也是 `1, 2, 3, true, abc`

（3）例 1: `var a=10;`
`var b=a;`
`a=20;`
`document.write(b);`
 得 10

例 2: `var arr=[1];`
`var arr1=arr;`
`arr.push(2);`
`document.write(arr1);`
 得 1, 2

例 3: `var arr=[1, 2];`
`var arr1=arr;`
`arr=[1, 3];`
`document.write(arr1);`
 得 1, 2

解析：原始值和引用值都是要往计算机里边存的，但是原始值和引用值存的地方不一样，原始值存在 `stack`（栈内存）里，引用值存在 `heap`（堆内存里），栈内存相当于一个有底没有顶的箱子，先进去的东西最后出来，堆内存相当于一个储物格，散列结构，怎么放进去怎么拿出来。通俗的讲，栈内存和堆内存里边有好多房间，举个例子，比如说房间号为 1001, 1002, 1003……

例 1：原始值有一个特点：不可改变，放进去的值永远改变不了，所以，先在栈内存里把 1001 号房间改名为 a 房间，里边放了个 10，然后再在栈内存里把 1002 房间改名为 b 房间，然后，b=a 是把 10 这个数字复制过来的，原来 a 里边的数值还在，此时 a 和 b 里边都有 10，再让 a=20 的话，由于原来 a 里边的 10 改变不了，所以系统会把 1003 房间再次改名为 a 房间，里边放入 20，原来的 a 房间数值 10 还在，只不过把 a 这个名字去掉了，变回了 1001 房间，成为了一个野房间，那么，b 房间里的 10 还是在那里，所以打印 b 得 10。

例 2：由于声明变量和变量赋值是分开的，所以系统会先在栈内存里先把 1001 房间改名为 arr 房间，后来发现 arr 里放的是引用值，所以把数值 1 放进了堆内存里的 1001 房间里，此时，栈内存里的 arr 房间里也有东西，放的是堆内存 1001 房间的地址 heap1001，当 var arr1=arr 时，系统会把栈内存里的 1002 房间改名为 arr1 房间，arr1 房间里也放的是堆内存 1001 房间的地址 heap1001，然后 push 的意思就是给数组里加东西，所以堆内存 1001 房间里加了一个 2 变成了 1, 2，栈内存里 arr 和 arr1 还是指向堆内存 1001 房间，所以打印 arr1 得 1, 2。

例 3：先把栈内存 1001 房间改名为 arr 房间，在堆内存 1001 房间放入 1, 2，把 heap1001 放入 arr 房间，然后把栈内存 1002 房间改名为 arr1 房间，放入 heap1001，此时，给 arr 重新赋值了，则需要在堆内存 1002 房间放入 1, 3，然后把栈内存 arr 里边的地址变为 heap1002，此时 arr 指向 heap1002，arr1 还是指向 heap1001，所以打印 arr1 得 1, 2。

注意：例 2 中 push 是给原数组里边加东西的，所以不用在开房间，例 3 是重新赋值，需要在开一个新的房间。

5. js 语句基本规则

- (1) 语句后边要用英文分号结束。
- (2) js 语法错误会引发后续代码终止，但不会影响其他代码块。
- (3) 书写格式要规范，“+*/=”两边都要有空格。
- (4) js 注释：Ctrl+?，单行注释是//注释内容，多行注释为/*注释内容*/

6. 运算操作符

- (1) +：加号，它的作用是数字运算和字符串连接，例如

`var a=1+2;` 打印 a 得 3，

`var a= "a" + "b";` 打印 a 得到 ab，

任何数字类型加字符串都等于字符串，

`var a=1+ "a";` 打印 a 得到 1a，

`var a= "a" +1+1;` 打印 a 得到 a11，

计算机的运算顺序也和数学一样，

`var a=1+1+ "a";` 打印 a 得到 2a,
`var a= "a" + (1+1);` 打印 a 得到 a2

(2) `-`: 减号, 例如

`var a=3-1;` 打印 a 得到 2

(3) `*`: 乘号, 例如

`var a=3*3,` 打印 a 得到 9

(4) `/`: 除号, 例如

`var a=9/3;` 打印 a 得到 3

`var a=0/0;` 打印 a 得到 NaN (NaN 属于数字类型, 但是又得不出一个具体的数, 则得到 NaN)

`var a=1/0,` 打印 a 得到 infinity (无穷)

`var a=-1/0,` 打印 a 得到-infinity (负无穷)

(5) `%`: 摩尔, 就是求余, 例如

`var a=5%2;` 打印 a 得到 1

`var a=10%4;`打印 a 得到 2

`var a=5%1;` 打印 a 得到 0

`var a=4%6;` 打印 a 得到 4

(6) `=`: 赋值

(7) `()`: 有括号的先算括号里边的。

(8) `++`: 例如 `a=a+1` 就可以简写成 `a++`, `a=a+1` 他是先计算右边的在把值赋到左边去, 就是 `a+1` 再赋给 `a`, 就是自身+1 再赋给自身就叫`++`。

(9) `--`: 例如 `a=a-1` 就可以简写成 `a--`, 就是自身-1 再赋给自身就叫`--`。

注意: `++`和`--`位置还不太固定, 可以放前边也可以放后边, 以`++`为例, 有个 `a++`和`++a`, 他们的区别是先`++`还是后`++`, 比如说:

例 1: `var a=10; document.write(a++); document.write(a);` 打印第一个 a 得 10, 打印第二个 a 得 11.

解析: `++`放在后边 (`a++`) 就是先执行语句后`++`, 也就是第一次先把 a 打印出来然后再`++`, 所以第一次打印 a 得 10, 然后`++`, 第二次打印 a 得 11.

例 2: `var a=10; document.write (++a); document.write(a);` 打印第一个 a 得 11, 打印第二个 a 得 11.

解析: `++`放在前边 (`++a`) 就是先`++`再执行语句, 就是先`++`然后把 a 打印出来, 所以第一次打印得到 11, 第二次打印也得到 11.

例 3: `var a=10; var b= ++a -1 + a++; document.write(b+ " " +a);` 得 21 12.

解析: 赋值的顺序是自右向左, 计算的顺序是自左向右, 此例中, 打印 b 和 a (“ ” 是

空格)，起初 $a=10$ ，然后计算 b ， $++a$ ，先 $++$ 后执行语句，此时 a 变成 11， $11-1$ 得 10， $10+a++$ ，先执行语句后 $++$ ，所以 $b=10+11=21$ ，然后 a 是 11 再 $++$ ，打印得到 12。

(10) $+=$ ：例如： $a=a+10$ 就可以简写成 $a+=10$ ，这个符号只能放中间，其实 $a++$ 就是 $a+=1$ 的简写。

(11) $-=$ ：例如： $a=a-10$ 可以简写成 $a-=10$

(12) $*=$ ：例如： $a=a*10$ 可以简写成 $a*=10$

(13) $/=$ ：例如： $a=a/10$ 可以简写成 $a/=10$

(14) $\%=$ ：例如： $a=a\%10$ 可以简写成 $a\%=10$

注意：在 js 运算符里， $()$ 的优先级最强， $=$ 的优先级最弱。

比较运算符、逻辑运算符，条件语句、循环语句

1. 比较运算符

(1) $>$ ：大于号

(2) $<$ ：小于号

(3) $==$ ：等于号，我们之前所说的 $=$ 是赋值符号，在 js 里， $==$ 才是真正的等于号。

(4) $>=$ ：大于等于

(5) $<=$ ：小于等于

(6) $!=$ ：不等于

比如 $\text{var } a=1>2$ ，它打印出来是 `false`，打印出来是 Boolean(布尔值)值，若 $\text{var } a=1<2$ ，则打印出来是 `true`，如果是两个字符串进行比较，他会用自身的阿斯克码进行比较，在 js 里，NaN 不等于任何东西，包括自己。

2. 逻辑运算符

(1) $\&\&$ ：与

`undefined`、`null`、`NaN`、`0`、“”、`false` 这些值转换为布尔值是 `false` (假)，其他的值转化为布尔值都是 `true` (真)。

如果 $\&\&$ 只有两个值的话，他会先看第一个表达式的值转化为布尔值是否为真，如果为真，他就会把第二个表达式的值返回，如果第一个表达式的值为假，他就会直接把第一个表达式的值返回，不看后面的，比如：

`var a=1&&2; document.write (a);` 打印 a 得 2.

`var a=1-1&&2; document.write (a);` 打印 a 得 0.

如果 $\&\&$ 有三个或多个值，也是一样的，他会一个一个判断，如果这些值都为真，则返回最后一个表达式的值，如果判断到哪个表达式的值是假的，他就会把这个表达式的值直接返回，后边的就不看了(遇到假就停)(这里返回得是表达式的值，不是真和假，真和假是用来判断的)，所以 $\&\&$ 被叫做短路语句，比如 $2>1\&\&\text{document.write}(\text{“内容”})$ ，

当前边为真的时候，才会执行后边的语句，如果前边是假的，则不执行后边的语句。

(2) ||: 或

||和&&正好相反，如果||有多个值，他会一个一个判断，如果这些值都为假，则返回最后一个值，如果判断到哪个表达式的值是真的，他就会直接返回这个表达式的值，后边就不看了（遇到真就停）。比如

```
var a=1||2; document.write (a); 打印 a 得 1.
```

```
var a=1-1||2; document.write (a); 打印 a 得 2.
```

(3)!: 非

!的作用就是把一个值转换为布尔值再取反，如 var a=!123;打印 a 得 false,!! 的作用是直接把一个值转化为布尔值，如 var a=!! “”; 打印 a 得到 false, var a=!!123, 打印 a 得 true。

3. 条件语句

(1) if 语句和 if else if

它的形式为:

```
if (条件) {
    执行语句
}
```

他的原理是，当小括号里的条件转化为布尔值为真的时候，才会执行大括号里边的语句:

```
例 1: if (1>0) {
    document.write (“哥很帅”);
}
```

```
例 2: if (1>2) {
    document.write (“哥很帅”);
}
```

```
例 3: if (1>0&&8>9) {
    document.write (“哥很帅”);
}
```

```
例 4: if (1>0||8>9) {
    document.write (“哥很帅”);
}
```

解析：在例 1 中，1>0 为真，所以会执行后边的语句，例 2 中 1>2 为假，所以不会执行后边的语句，例 3 有与运算符，之前讲的，一个一个判断，1>0 为真，继续往下看，8>9 是假的所以整个括号里边返回假，所以不会执行，例 4 是或运算符，1>0 是真的，直接

返回，括号里返回真，执行后边的语句。所以在 if 语句里，&&可以理解成并且，全真才为真，才会执行后边的语句，||理解成或者，有一个真就为真就会执行语句，全假才为假。

练习 1: 写一个程序，弹出一个可以输入数值的对话框，里边输入考试的分数，然后当分数在 90-100 之间的话，打印 alibaba，当分数在 80-90 之间时，打印 tencent，当分数在 70-80 之间时，打印 baidu，当分数在 60-70 分之间时，打印 mogujie，当分数在 60 分以下时，打印 Oh my god!!!you gotta be kidding me!!!开始写源码：

```
var score = parseInt(window.prompt("input"));
if(score > 90 && score <= 100){
    document.write("alibaba");
}else if(score > 80 && score <= 90){
    document.write("tencent");
}else if(score > 70 && score <= 80){
    document.write("baidu");
}else if(score > 60 && score <= 70){
    document.write("mogujie");
}else if(score < 60){
    document.write("Oh my god!!!you gotta be kidding me!!!");
}else{
    document.write("error");
}
```

解析：var score = parseInt(window.prompt("input"));这句的作用是弹出一个可以输入的对话框，对话框的提示信息是“ ”里边的 input，然后把输入的数值转换成整形装到变量 score 里边去，分数在 90-100 之间，不能写成 90<score<=100, 这样计算机识别不了，它是满足了两个条件，大于 90 并且小于等于 100，即 score>90&&score<=100，然后当你写了很多 if 的时候并且条件是互斥的（满足第一个条件就不能满足第二个条件）就可以使用 else，else 的作用就是当有很多 if 的时候，当满足了前边的条件执行完语句后后边的就直接不看了，不用一个一个去判断，但是 else 必须在条件互斥的情况下才可以用，如果两个 if 语句有交叉点就不能用，比如当在 90-100 之间打印一个值，在 80-95 之间打印一个值，两个条件有交叉点就不能用 else，然后最后一个 else 就是除了以上情况以外的情况打印 error（错误分数）。

(2) if 和&&转换

其实 if (1>2) {document.write ("a"); }就等于 1>2&&document.write ("a");

4. 循环语句：系统为了方便我们重复性的做一件事，给我们提供了一个机制，叫循环

机制，他会循环的写东西，还可以控制循环圈数，当你把语句放在里边时，他会自动循环。

(1) for 循环：

例 5：以打印 10 个 a 为例，它的代码为：

```
for(var i=0; i<10; i++){
    document.write( "a" );
}
```

其中，for 为关键字，小括号里放了三个语句，用两个分号隔开，大括号里的语句叫做循环体，这样这个 a 就被循环往复打印了十遍，它的原理在于它的执行顺序：

第一步：先执行小括号里边的第一个语句 var i=0;（这个语句只执行一遍，所有的 for 循环小括号里的三条语句中的第一条只执行一遍，以后就不用了）

第二步：把小括号里的第二条语句塞到 if 里作为条件来判断，如果条件成立则执行大括号里的循环体，在此例中，if (i<10) { document.write("a");}，因为 0<10 是成立的，所以 a 被打印了一遍。

第三步：执行小括号里的第三条语句，i++，此时 i 变成了 1。

第四步：再把小括号里的第二条语句塞到 if 里作为条件来判断，如果条件成立则执行大括号里的循环体，在此例中，因为 1<10 是成立的，所以 a 又被打印了一遍。

第五步：继续执行小括号里的第三条语句，i++，此时 i 变成了 2。

以此类推……………

到最后，当 i 变成了 9，判断 9<10，打印第十个 a 之后，继续 i++，i 变成了 10，10<10 不成立，则立即结束循环。

总结：先执行 for 循环小括号里的第一条语句，然后把第二条语句塞到 if 里作为条件判断，如果条件成立，则执行大括号里的循环体，然后执行第三条语句，执行完之后再第二条语句塞到 if 里作为条件判断，如果条件成立，则执行大括号里的循环体，继续执行小括号里的第三条语句……直到括号里的第二条语句塞到 if 里作为条件判断时条件不成立，则 for 循环结束。（注意：for 循环括号里的第一条语句只用一次，第二条语句判断条件成立执行完循环体之后第三条语句才执行，第三条语句永远在最后执行。）

拓展：其实 for 循环的小括号里也没有非要写三个语句，如果哪个语句是空的，就不会执行，直接跳过，所以例 5 也可以这么写：

```
var i=0;
for(; i<10;){
    document.write( "a" );
    i++;
}
```

```
}
```

把 `var i=0` 提出去，让他先执行，然后，小括号里的第一个语句是空的，直接跳过，然后判断第二句，执行循环体（先打印，再 `i++`），然后小括号第三句空的，跳过，继续判断第二句，执行循环体，循环往复，直到 `i` 加到 10, `10<10` 是假的，循环停止，`i` 从 0 到 9，`a` 被打印了十次。

例 6：接着例 5，打印 10 个 `a`，但是要求 `for` 循环小括号里只有一个 `i`，`i=1`，代码为：

```
var i=1;
var count=0;
for (; i;) {
    document.write ("a");
    count++;
    if (count==10) {
        i=0;
    }
}
```

解析：这里边定义一个计数器 `count`，然后把 `count` 放到 `for` 循环里让他自己++，再写一个 `if` 语句，当 `count` 加到 10 之后，让 `i=0`；加了 10 次后，`i` 变成 0，小括号里的条件就不成立了，`for` 循环结束，`a` 被打印了 10 遍。也可以不用 `count`，用 `i` 作为计数器，当 `i` 等于 11 的时候，让 `i` 归零：

```
var i=1;
for (; i;) {
    document.write ("a");
    i++;
    if (count==11) {
        i=0;
    }
}
```

其实 `for` 循环还能衍生很多东西出来，因为它每次执行的是循环体，所以我们可以抓住一些变化：

例 7：打印 0-9 九个数。

```
for(var i=0;i<10;i++){
    document.write(i);
}
```

解析：`i` 可以控制循环圈数，它也可以代表一个数字在变化，所以直接让 `i<10` 并且自

加，在打印 *i* 即可。

例 8：求出 0-9 的和。

```
var count=0;
for(var i=0;i<10;i++){
    count+=i;
}
document.write(count);
```

解析：和例 7 一样控制 *i*，再定义一个筐 *count*，然后把每一次循环的 *i* 扔到筐里加起来，注意的是：必须等到循环结束后才能打印 *count*。

例 9：打印出 0 到 100 能被 3 整除或者能被 5 整除或者能被 7 整除的所有数。

```
for(var i = 0;i <= 100;i++){
    if(i % 3 == 0 || i % 5 == 0 || i % 7 == 0){
        document.write(i + ' ');
    }
}
```

解析：先把 0-100 的数用 *i* 拿 *for* 列出来，然后 *for* 里边套个 *if*，*if* 的条件为当 *i* 与 3, 5, 7 摩尔余 0 的时候（整除），再打印 *i* 即可。

例 10：打印 0-100 的所有数，要求小括号里边只能写一条语句且不许用比较运算符，循环体里只能写一句话。

```
var i = 101;
for(;i--;){
    document.write(i + ' ');
}
```

解析：换一个思维，括号里写 *i--* 再打印 *i*，当 *i* 减到 0 的时候循环就结束了，倒序打印，哈哈!!!

(2) while 循环

其实 *for* 循环里小括号前后不写东西的话只有中间写东西就是 *while* 循环，*for*(;*i*<10;) {循环体} 就等于 *while* (*i*<10) {循环体}。

例 11：敲 7 游戏：用 *while* 循环写出 0-100 可以被 7 整除或者个位有 7 的所有数。

```
var i = 0;
while(i <= 100){
    if(i % 7 == 0 || i % 10 == 7){
        document.write(i + ' ');
    }
}
```

```
    i++;  
}
```

解析：和 for 循环相似，不再赘述，个位有 7 的数只要摩尔 10 余 7 即可。

(3) do while 循环：有这个东西，一般不用

```
do{  
    执行语句  
}while (条件)
```

不管 while 里的条件能不能满足都会先执行一遍语句，它是先执行语句再判断条件，感觉特别别扭，一般我们不用它，也没人用。

练习 2：计算 2 的 n 次幂，n 可输入，n 为自然数。

```
var n = parseInt(window.prompt("请输入 n"));  
var mul = 1;  
for(var i = 0;i < n;i++){  
    mul *= 2;  
}  
document.write(mul);
```

解析：2 的 n 次幂就是 n 个 2 相乘，用 mul 作为他的结果，先让 mul=1，n 个 2 相乘即 mul 被乘了 n 次 2，用编程语言就可以理解为它是一个循环，然后每次循环都给 mul 自身乘 2，乘了 n 次 2，则循环了 n 圈，然后用 n 去控制循环圈数即可。

练习 3：计算 n 的阶乘，n 可输入。

```
var n = parseInt(window.prompt("请输入 n"));  
var mul = 1;  
for(var i = 1;i <= n;i++){  
    mul *= i;  
}  
document.write(mul);
```

解析：阶乘就是，比如说 5 的阶乘就是 $5*4*3*2*1$ ，所以也可以理解成 for 循环，为了让 n 和循环圈数相等，所以 var mul=1，然后去乘这 5 个数，乘 5 圈，用 i 表示这五个数，先让 i 等于 1，然后去乘，再自加……加到 5 去乘，循环结束。

练习 4：输入 a，b，c 三个数，打印出最大的数。

```
var a = parseInt(window.prompt("请输入 a"));  
var b = parseInt(window.prompt("请输入 b"));  
var c = parseInt(window.prompt("请输入 c"));  
if(a > b) {
```



```

    if(a > c){
        document.write(a);
    }else{
        document.write(c);
    }
}
}
else{
    if(b > c){
        document.write(b);
    }else{
        document.write(c);
    }
}
}

```

解析：先让 a 和 b 比，如果 a 大于 b，就让 a 和 c 比，a 大于 c 的话就打印 a，否则打印 c，如果 a 不大于 b，就让 b 和 c 比，b 大于 c 的话就打印 b，否则打印 c。

练习 5：编写一个程序，输入一个三位数的正整数，输出时反向输出，例如输入 456，输出 654。

```

var num = parseInt(window.prompt("请输入一个三位数（整数）"));
if(num < 100 || num > 999) {
    document.write("输入不正确！");
}else{
    var a = num % 10;
    num = num - a;
    var b = num % 100;
    num = num - b;
    b /= 10;
    var c = num / 100;
    var num1 = a * 100 + b * 10 + c;
    document.write(num1);
}

```

解析：此例中用 cba 来比喻三位数，输出 abc，先用这个数摩尔 10，把 a 取出来，然后用 cba 减去 a 得 cb0，再用 cb0 摩尔 100 得到 b0，用 cb0 减去 b0 得 c00，用 b0 除以 10 取出 b，用 c00 除以 100 取出 c，三个数字取出来之后，组成 abc，a 乘 100 加 b 乘 10 加 c 即可。

练习 6：著名的斐波那契数列：1 1 2 3 5 8，输出第 n 项。

```
var n = parseInt(window.prompt("请输入 n"));
var first = 1,
    second = 1,
    third;
if(n > 2){
    for(var i = 0;i < n - 2;i++){
        third = first + second;
        first = second;
        second = third;
    }
    document.write(third);
}else{
    document.write(1);
}
```

解析：这个斐波那契数列只要你找到规律了就好办，这个数列已知前两个数，后边的数都是未知的，但是未知数等于前两个数字的和，知道了这个规律，就可以 var 三个变量，第一个数和第二个数（first 和 second）都赋值 1，第三个数（third）以及后边的数我们用循环来求，你还会发现一个规律，求第三个数得计算一次，求第四个数得计算两次，求第五个数得计算三次……那么，求第 n 个数就要计算 n-2 次，所以，我们写个 for 循环，用 i 控制循环圈数，循环圈数 = n-2，在循环体里，首先第三个数等于第一个数加第二个数，然后，我们移一下位，让第一个数等于第二个数，第二个数等于第三个数，以便于下一轮再次计算，这样循环的话，等到循环结束，打印 third 即可。还有就是当 n 输入 1 和 2 时，用 if 语句让他打印 1 就好了。

练习 7：打印出 100 以内的所有质数。

方法一：

```
var count = 0;
for(var i = 1;i <= 100;i++){
    for(var j = 1;j <= i;j++){
        if(i % j == 0){
            count++;
        }
    }
    if(count == 2){
        document.write(i + ' ');
    }
}
```

```

    }
    count = 0;
}

```

解析：质数就是只能被 1 或者自身整除的数，其实这么说不标准，就是他自己只能被从 1 到他自身的所有数整除两次，所以 1 不是质数。要打印 100 以内的质数，先把 1-100 的所有数字拿 i 用一个 for 循环表示出来，外边的 for 循环表示每一个 i 的变化，我们再把循环里边套一个 for 循环，里边定义一个 j，让 j 小于等于 i 且自加，我们定义一个计数器 count，那么，当 i 被 j 整除了以后，count 就加一，里边的这个 for 循环是控制 j 和 count 的，返回外边的循环，当里边的循环结束的时候也就是 j 等于 i 的时候，假如 count 等于 2 的话，就打印 i，打印完之后，由于一个 count 控制一个 i，所以当 i 变化的时候，得让 count 归零。

方法二：

```

var count = 0;
for(var i = 2;i <= 100;i++){
    for(var j = 1;j <= Math.sqrt(i);j++){
        if(i % j == 0){
            count++;
        }
    }
    if(count == 1){
        document.write(i + ' ');
    }
    count = 0;
}

```

解析：这是一个思维的飞跃，比如写几个得数为 100 的乘式：10*10, 5*20, 4*25, 2*50, 1*100，这几个乘式你会发现第一个因数变小，第二个因数必然变大，如果 100 能被 2 整除，必然能被 50 整除，能被 4 整除就能被 25 整除，它是以 100 的开方 10 来看的，所以第二个循环没必要向方法一那样循环那么多圈，直接控制 j 小于 i 的开方 Math.sqrt(i)，然后当 count 加到 1 就打印 i 即可。

5. document.write 和 console.log 的区别

在真正开发的时候，document.write 是往页面里边输出文档流的，它干的不是输出的事儿，以前用它是为了显示方便，真正输出用的是 console.log，它的作用是把东西输出至控制台。

条件语句补充，终止循环

1. 条件语句 `switch case` 语句，他的形式为：

```
switch(1) {  
    case 1:  
        console.log('a')  
    case 2:  
        console.log('b');  
    case 3:  
        console.log('c');  
}
```

我们 `if` 语句小括号里放的是条件判断，而 `switch` 里边小括号放的是条件，`case` 才是真正的判断，假如说括号里的条件和 `case` 后边的值相比对，一样的话就会执行 `case` 后边的语句，小括号里的值和 `case` 后边的值可以是数字、字符串、`true` 等等都可以，也可以在外边 `var` 一个变量并赋值，把变量名放在小括号里边。但是例如上边的，假如说小括号里边放的是 3 的话，输出 `c` 没有问题，但是如果小括号里边放的是 1 的话他就会把 `abc` 都输出，如果小括号里是 2 的话输出 `bc`，也就是说只要它判断到哪里，假如说值是一样的，他就会把后边的所有语句全部执行，解决的办法就是在 `case` 后边的语句结束之后加上一个 `break;`，`break` 就是终止语句，终止循环的意思。假如说判断到哪里读到了 `break` 他就会跳出语句，不会继续往下执行。

例 1：编写一个程序，当输入周一到周五的时候，输出 `working`，输入周六或周日的时候输出 `relaxing`。

```
var date = window.prompt('input');  
switch(date) {  
    case "周一":  
    case "周二":  
    case "周三":  
    case "周四":  
    case "周五":  
        console.log('working') ;  
        break;  
    case "周六":  
    case "周日":  
        console.log('relaxing') ;
```

```

    break;
}

```

解析：var date = window.prompt('input');是把输入的值直接赋给变量（和 parseInt(window.prompt("input"));不同的是不用转换成整形直接赋值），这里的写法是简化后的，假如输入周一也会一直往下看执行那个语句，功能一样的。

2. break

break 的真正作用是终止循环，我们可以认为 switch case 也是一种循环，break 还可以用于 while 循环和 for 循环，比如：

```

var i = 0;
while(1){
    i++;
    console.log(i);
    if(i > 100){
        break;
    }
}

```

在此例中，while 里的条件是 1，所以这个循环永远不可能结束，是死循环，但是我们让它当 i 大于 100 的时候就终止循环，那么他就会输出 1-101. 在有一些循环里你不太好控制它到底循环多少圈，而你又希望他到特定的条件停止，这个时候 break 就是一个最好的选择。但是 break 必须写到循环里边，如果写到循环外边他会报错的。同样，break 也可以用于 for 循环。比如：

```

var i;
var sum = 0;
for(i = 1;i <= 100;i++){
    sum += i;
    console.log(i);
    if(sum > 100){
        break;
    }
}

```

我们让 1-100 这些数加起来，但是我们想知道当加到多少的时候就超过 100 了？我们就可以这么写。

3. continue

continue 的作用是终止本次循环进行下一轮循环，就是当满足特定条件之后他就不会

执行循环体，进行下一轮循环。

例 2：反向敲 7：输出 0-100 除了能被 7 整除和个位有 7 之外的所有数。

```
for(var i = 0;i <= 100;i++){
    if(i % 7 == 0 || i % 10 == 7){
        continue;
    }
    console.log(i);
}
```

解析：当 i 能被 7 整除或者摩尔 10 余 7 的时候让他终止本次循环，进行下一轮循环即可。

初识引用值、typeof、类型转换

1. 数组

(1) 数组里边可以放很多东西, 他是一个存放物品的一个集合, 他的形式是一个中括号里边放入很多值 (数字、字符串、undefined 等等都可以), 例如: `var arr = [1, 2, 3, undefined, "abc"]`

(2) 数组的读和写入:

由于数组里边存放的不是一个数据, 是很多数据, 所以我们再计算的时候是要把每一个数拿出来计算的, 而不是计算数组, 拿 `var arr = [1, 2, 3, 4, 5]` 举例:

读: 由于计算机计数是从 0 开始的, 所以我们要取第一位的值就是 `arr[0]`, 数组的第二位是 `arr[1]`, 比如说输出数组第 1 位就是 `console.log(arr[0]);`, 输出第一位得 1。

写入: 当然我们也可以给数组的每一位重新赋值, 比如 `arr[0] = 2`, 此时把上边的数组第一位变成了 2, 数组就变成了 `[2, 2, 3, 4, 5]`

数组的长度: `arr.length` 表示数组的长度, 由于上边的数组有五位, 所以我们 `console.log(arr.length)` 得 5。

(3) 遍历数组, 拿 `var arr = [1, 2, 3, 4, 5]` 举例:

我们可以用 for 循环把数组的每一位单独输出

```
for(var i = 0;i < arr.length;i++){
    console.log(arr[i]);
}
```

因为数组的第一位是 0, 所以定义 i 等于 0, i 小于 `arr.length`, 就是最后一圈 `i=arr.length - 1`, 因为是从 0 开始计数, 所以第五位就是 `arr[4]`, 此时就把数组的每一位都输出了。

如果我们想把数组的每一位都改成 1, 可以这么写


```
for(var i = 0;i < arr.length;i++){
    arr[i] = 1;
}
```

如果我们想把数组的每一位都加 1，可以这么写

```
for(var i = 0;i < arr.length;i++){
    arr[i] += 1;
}
```

2. 对象：对象也是存储数据的一个仓库，只不过比数组更直观一点，数组的每一位都没有给他起名字，只不过叫第一位第二位，对象就不一样了，对象给他加了一个属性名，他的形式是，例如：

```
var deng = {
    lastname: "deng",
    age: 40,
    sex: undefined,
    wife: "xiaoliu",
    father: "dengdaye",
    son: "xiaodeng",
    handsome: false,
}
```

属性名后边加上冒号在加上属性值，最后用逗号（不是分号，写分号会报错的）。

对象的取值方法：变量名.属性名，例如上边的，我们 `console.log(deng.lastname)` 就会输出 deng，我们也可以重新赋值，例如 `deng.lastname = "old deng"`

3. 编程可以分为面向过程和面向对象两种，JavaScript 属于半面向过程半面向对象的语言。

4. typeof 操作符

typeof 操作符的作用是区分数据类型，比如说你 `var num = 123;console.log(typeof(num));`（也可以写成 `console.log(typeof num)`）输出就得 `number`，假如你 `var num = "123";console.log(typeof(num));`输出就得到 `string`。typeof 有六种数据类型：`number`（数字）、`string`（字符串）、`boolean`（true 和 false）、`undefined`（undefined）、`object`（数组和对象和 null）、`function`（函数）。

注意（考点）：我们平时没有定义变量直接输出，如 `console.log(a)` 就会报错，但是我们即使没有定义变量，输出 `console.log(typeof(a))` 会得到 `undefined`，我们 `console.log(typeof(typeof(a)))` 会得到 `string`，因为输出的这个 `undefined` 是字符串类型的。

5. 显式类型转换

(1) **Number**: Number 可以把所有的值类型转化为数字类型，比如说你 `var num = Number("123")`，他就可以把字符串类型的 123 转换为数字类型的 123，你 `console.log(typeof(num) + ":" + num)`；就得到 `number: 123`。“123”转化为数字类型得到 123，“-123”转换后得-123，true 转换后得 1，false 转换后得到 0，null 转换后得到 0，undefined 转换后得到 NaN，“a”转换后得到 NaN，“123a”转换后得到 NaN，所有的值经历了 Number 的转换都能变成数字类型，只不过有些转换不成具体的数字的用 NaN 表示。

(2) **parseInt**: 他有两个作用

作用一：他的作用就是把数字转化成整形，比如你 `var num = parseInt(123.3)`，那么输出 num 就是 123（不能四舍五入，只是把后边的小数去掉），他还可以把带有数字的字符串转化为数字类型的整数，比如你 `var num = parseInt("123.9")`，那么输出 num 就是 123，在比如你 `var num = parseInt("123abc")`，他也会输出 123，他会从数字位往后看，看到非数字位后就把他砍断了，然后转换为数字类型的整数，他也就只能把带有数字的字符串转化为数字类型的整数（字符串里的数字必须在前），其他的例如 true、undefined 等等转换后都是 NaN。

作用二：比如你 `var demo = 10; var num = parseInt(demo, 16)`；parseInt 括号里有两个参数，16 他叫做 radix（基底），他就会认为这个 10 是 16 进制的，以 16 进制为基底，再把它转换成 10 进制的数，所以你输出 num 得 16。就是他会把目标进制做为基底再转换为十进制的数。radix 的取值范围是 2-36。

(3) **parseFloat**

parseFloat 和 parseInt 极其相似，他的作用就是把一个值转换为浮点数，他还可以把带有数字的字符串转化为浮点数，比如说 `var num = parseFloat(123.2abc)`；输出 num 得到 123.2。其他的例如 true、undefined 等等转换后都是 NaN。

(3) **String**

String 的作用就是把值转化为字符串（任何值都可以转），比如你 `var num = String(123); console.log(typeof(num) + ":" + num)` 得到 `string: 123`，比如你 `var num = String(true); console.log(typeof(num) + ":" + num)` 得到 `string: true`。

(4) **Boolean**

Boolean 的作用就是把一个值转换为布尔值，比如 `var num = Boolean("")`，输出 num 得到 false，比如你 `var num = Boolean("123")`，输出 num 得到 true。

(5) **toString**: 他有两个作用

作用一：他也可以把值转换为字符串，但是写法和 String 不一样，`var num = (123).`

`toString()`; `console.log (typeof (num) + “:” + num)` 得到 `string: 123`, 但是 `undefined` 和 `null` 不能用 `toString`, 会报错的。(`toString` 写法有点特殊, 如果你想给数字 `toString`, 你不能直接写成 `数字.toString()`, 会报错的, 你必须给数字加个括号, 写成 `(数字).toString()` 才可以, 其他的不用加)

作用二: 比如你 `var demo = 10; var num = demo.toString(8);` `toString` 括号里放的是 `radix` (基底), 但是他和上面讲的不一样, 它是以 10 进制为基底转化为目标进制的数, 也就是以 10 进制为基底转换为 8 进制的数, 所以输出 `num` 得到 12。

(6) `toFixed`

`toFixed` 的作用就是保留小数点后有效数字位的, 比如 `console.log(123.45.toFixed(1));` 括号里是 1, 保留一位小数, 输出 123.5 (四舍五入)

练习: 一个二进制的数 10101010, 转换为十六进制的数。

```
var demo = 10101010;
var num = parseInt(demo, 2);
console.log(num.toString(16));
```

解析: 先把 10101010 转换为 10 进制的数, 在转换为 16 进制的数, `parseInt` 是把目标进制转换为十进制, `toString` 是把十进制的数转换为目标进制。

附: 进制

满 x 个数后从个位变成十位就叫做进制, 我们通常用的十进制就是满十进一 (满十后让个位变成 0, 十位变成 1), 比如说 $9 + 1 = 10$, 但是比如说十一进制就是满十一进一, 即 $9 + 1 = a$, $a + 1 = 10$, 但是这个 10 (一零) 这个数在我们十进制里就是十一了。我们通常用到的有十进制, 十六进制, 二进制。

十六进制: 1 2 3 4 5 6 7 8 9 a b c d e f 10, 那么 f 就是十进制的 15, 比如说 1f 就等于十进制的 $16 + 15 = 31$,

二进制: 10 代表 2, 11 代表 3, 100 就是 2 的 2 次方, 1000 就是 2 的 3 次方。

6. 隐式类型转换

(1) `isNaN`

`isNaN` 的作用是判断一个值是不是 NaN, 比如 `console.log(isNaN(NaN))`, 输出得 `true`, `console.log (isNaN (123))`, 输出得 `false`, `console.log (isNaN (“123”))`, 输出得 `false`, `console.log(isNaN(“a”))`; 输出得 `true`, `console.log(isNaN(undefined))` 输出得 `true`。

其实经历了 `isNaN` 的值系统会先让他调用 `Number` 来判断他是不是 NaN, 如果是, 就返回 `true`, 不是就返回 `false`。

(2) `++ --, + -` (正/负)

所有的值经过++和--的运算后都会先把他用 Number 转换为数字类型再计算，比如“1”++就等于数字类型的 2，“a”++就等于 NaN，+/-也一样，他调用的也是 Number，“-1”就等于数字类型的-1，“+a”就等于 NaN。

(3) +

+如果加号两边有一边是字符串，他就会调用 String，把两个值连接起来，如果没有字符串，他就会调用 Number

(4) - * / %

减、乘、除、摩尔他都会先调用 Number 转换为数字类型再计算

(5) && || !

!把他转换为布尔值再取反，所以他调用的是 Boolean，其实&&和||调用的也是 Boolean，它是把值转换为布尔值来判断真假的

(6) > < >= <=

如果两个字符串比较，就会用自己的阿斯克码比较，如果字符串和数字比较，就会把字符串转换为数字。

(7) == !=

==和!=也会用隐式类型转换，输出 3==“3”就得到 true。

注意：有几个特殊的，比如 1>2<1 得 true，他会一个一个算，1>2 是 false，转换后是 0，0<1 最后得 true。

还有 undefined 和 null 既不大于 0 又不小于 0 也不等于 0，所以 undefined==null（比较特殊，没有规则，系统定义的，系统规定他俩不能和数字比较，所以他和数字比较的时候也不会发生类型转换，就是不能比），NaN 不等于任何东西，包括自己。

7. 不发生类型转换

=== !==

===是绝对等于，!==是绝对不等于，这次两边必须一样才可以，1===1 得 true，1===“1”得 false，1!==“1”得 true，但是 NaN===NaN 是 false

函数、初识作用域

1. 函数的诞生——声明函数的基本作用及形式：

(1) 基本作用：比如说有以下代码：

```
if (1 > 0) {
    document.write('a')
    document.write('b')
    document.write('c')
}
```

```

if (2 > 0) {
    document.write('a')
    document.write('b')
    document.write('c')
}
if (3 > 0) {
    document.write('a')
    document.write('b')
    document.write('c')
}
.....

```

比如说以上代码，我们满足了这个条件之后要执行很多功能，而且这个条件满足了很多次，但是我们这么写的话代码的耦合（重复）程度非常高，这种代码我们把它叫做低效代码，我们编程讲究的是高内聚弱耦合，就是把相同功能的代码抽取出来放到一个黑匣子里边，我们每次在用的时候直接调用这个黑匣子就好了，不用每次都自己写，这样就方便了很多。这个可以封装成黑匣子的东西我们就把他叫做函数，就是一个函数体里可以放很多条语句，当我们需要用它的时候直接调用就可以了。

（2）函数的形式及调用函数

例如上边的代码，我们可以简化为：

```

function test() {
    document.write('a')
    document.write('b')
    document.write('c')
}
if (1 > 0) {
    test();
}
if (2 > 0) {
    test();
}
if (3 > 0) {
    test();
}

```

函数的基本形式：先是一个关键字 `function`，然后后边跟一个函数名（函数名的命名

规则和变量名差不多，不能以数字开头，可以用下划线等等，但是，不管是函数名还是变量名，如果有多个单词拼接，必须遵循小驼峰命名规则，即第一个单词首字母小写，以后的单词首字母都大写，如 `theFirstName`)，然后一个小括号，再一个大括号，大括号里放执行语句，这样，就把他封装好了，当我们要调用的时候，直接写一个函数名加一个小括号即可调用。例如上边的，函数里有三条语句，怎么让他执行呢？直接调用 `test()` 就可以了。而且你写上一个 `test()` 他就会执行一遍，你在写上一个 `test()` 他还会再执行一遍。

你把语句写在函数里和函数外效果是一样的，再比如：

```
function test() {  
    var a = 123;  
    var b = 234;  
    var c = a + b;  
    document.write(c);  
}
```

当你在底下 `test()` 的时候就会打印出 357，其实函数和变量差不多，都可以理解成一个筐，只不过变量里装的是数据，函数里装的是语句，而且当你调用他的时候他才会执行，如果你底下不写 `test()` 的话，他就不会执行里边的语句，他就是个装语句的筐，不会打印出 357，只有当你调用他的时候，他才会执行，打印出 357，而且调用两次的话就执行两次，打印两个 357。

既然函数可以把一些语句圈到里边，那么但凡能圈到它里边的东西我们就可以抽象出一类功能。就是说我们正常写的话会先把语句抽出来放在函数里，现在我们反着来，跟 `css` 一样，先定义功能，在使用功能，比如说我们要定义一个功能，打印 `abc`，就可以：

```
function test() {  
    document.write('abc');  
}
```

然后当你要用这个功能的时候直接调用函数就可以了。

注意：比如说

```
function a() {  
    var b = 1;  
}  
document.write(a);
```

你不调用这个函数，你想打印这个函数，他就会输出 `function a() {var b = 1; }`

2. 函数表达式

(1) 命名函数表达式

命名函数表达式的形式是这样的，例如：

```
var test = function abc() {
    document.write('a');
}
```

他会先定义一个变量，变量里边放入函数，这就是函数表达式，但是命名函数表达式的变量名会替代函数名的作用，意思就是调用函数你必须 `test()` 才可以，假如你 `abc()` 就会报错，这个 `abc` 就不起作用了，但是你输出 `test.name` 会得到 `abc`。

(2) 匿名函数表达式

既然命名函数表达式的函数名不起作用了，后来我们就省掉了，例如：

```
var test = function () {
    document.write('a');
}
```

这也是后来比较常用的函数表达式，但是假如你输出 `test.name` 会得到 `test`，一个声明函数的函数名 `.name` 也是函数名。

3. 参数——形参和实参

(1) 参数的定义及作用：我们知道函数的组成形式必须有关键字 `function`、函数名、小括号和大括号，这是函数必须有的东西，其实函数的组成形式中还有一个东西叫做参数，这个参数可有可无，但是正是因为这个参数才让函数变的神奇了，真正的高级编程中如果没有参数，那么函数也就没啥用了。但是参数比较简单，比如说：

```
function sum(a, b) {
    var c = a + b;
    document.write(c);
}
```

`sum(1, 2);`

例如上边的，我们小括号里边放的就是参数，函数名后边的小括号里放了两个值 `a, b`，这么一放就相当于给函数体里 `var a` 和 `var b`（但是写在括号里不能带 `var`，否则会报错），但是并没有给他赋值，我们就把他叫做形式参数（形参），我们在调用函数的时候小括号里放了 `1` 和 `2`，我们把它叫做实际参数（实参），这就相当于给 `a` 赋值 `1`，给 `b` 赋值 `2`，传参后 `a=1, b=2`，所以调用函数的时候打印 `c` 得 `3`。

作用：我们再来看上边这个函数，这个函数的功能就是实现两个数相加，传参数进去之后两个数就能相加了，就是我们现在不看调用函数那个语句，只看这个函数体，我们就相当于把两个数相加的规则给抽象出来了，这个 `a` 和 `b` 就跟变量一样，因为你每次执行的时候传的参数都可以不一样，比如你 `sum(1, 2)` 打印 `c` 就得 `3`，`sum(3, 4)`

打印 c 就得 7，这个就和数学里边的函数差不多了，数学里的函数为了抽象规则，xy 可以随时代换值，咱们这里边的变量就是参数，所以说有了参数之后我们的函数就真正变成抽象规则了，而不是原来只是为了聚合代码。这个参数结合里边的代码可以组成无数种用法，因为我们每次传进去的参数都可以不同，所以我们针对不同的参数可以进行不同的处理，所以有了参数后函数才变得强大了。

注意：实参里边传任何值类型都可以。

例 1：写一个函数体，加上两个形参，如果第一个数大于 10，就打印第一个数减去第二个的差，如果第一个数小于 10，就打印两个数的和，如果第一个数等于 10 就打印 10。

```
function test(a,b){
    if(a > 10){
        document.write(a - b);
    }else if(a < 10){
        document.write(a + b);
    }else{
        document.write(10);
    }
}
```

(2) JavaScript 不定参

JavaScript 天生不定参，不是说在一个函数里有几个形参就必须有几个实参，不是这样的，在 JS 里，形参比实参多，可以，实参比形参多，也行。

在同一个函数里，形参比实参多，比如说形参有 a,b,c，实参只有 1, 2，那么打印 a 得 1，打印 b 得 2，打印 c 就得 undefined。

在同一个函数里，实参比形参多，比如形参只有 a，实参有 1, 2，那么 a 就等于 1，那个 2 传不进去就算了。

但是无论实参怎么往里传，无论形参有没有把实参表示出来，这个实参都是有地方放的。在每一个函数里边都有一个隐式的东西叫做 **arguments**，arguments 是实参列表，他就是一个类数组，我们把它理解成一个数组，比如一个函数的实参是 1, 2, 3，那么 arguments 里边就是 [1, 2, 3]，就是说无论形参有没有接受完实参，arguments 都会把实参当做一个数组存起来，比如说你在函数里的形参是 a，实参是 1, 2, 3，你表面上看 2 和 3 没地方传，但是他会存放在 arguments 数组里的，你在函数里 console.log(arguments) 就会得到 [1, 2, 3]，既然是个数组，就有长度，你 arguments.length 就等于 3，我们也可以在函数体里把 arguments 的每一位用 for 循环单独输出。其实形参的长度也可以表示出来，用函数名.length 表示。

例 2：写一个程序，在一个函数里，如果形参比实参多，输出形参多，如果实参比形参多，输出实参多，如果一样多，输出相等。

```
function sum() {
    if(sum.length > arguments.length) {
        console.log('形参多');
    }else if(sum.length < arguments.length) {
        console.log('实参多');
    }else{
        console.log('相等');
    }
}
sum();
```

备注：我这里没写参数，自己试。

例 3：写一个求和功能的函数，无论实参里传多少值都能把他求出来。

```
function sum() {
    var result = 0;
    for(var i = 0;i < arguments.length;i++){
        result += arguments[i];
    }
    console.log(result);
}
sum();
```

解析：由于实参数量不固定，我们就可以用遍历数组把 arguments 的每一位都相加即可。

(3) 形参与实参的映射规则

比如说一个函数里形参是 a, b, 实参是 1, 2, 那么 arguments 的数组里就是 [1, 2], a 里边存的也是 1, 好, 我们现在在函数体里写个 a = 2; 这时给 a 重新赋值了, 你 console.log(arguments[0]) 也会得到 2, 继续写, 我们 arguments[0]=3, 这时你 console.log(a) 也得 3. 你们看, 这里边都是原始值吧? 怎么一个变一个跟着变呢? 形参和 arguments 确实是一个变一个跟着变的, 确实有这样的绑定规则, 但是, 他俩不是同一个变量, 系统内部有映射规则, 就是这俩我变你也得变, 但是他们是两个变量。

但是, 假如说形参比实参多, 比如说形参是 a, b, 实参只有 1, 就算你在函数体里写了 b=2, 你 console.log(arguments[1]) 会得到 undefined, 实参列表天生有几位就有

几位，只有形参和实参一一对应时才会映射。

4. return

(1) return 的第一个作用就是终止函数，比如说你在一个函数体里写

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

```
return
```

```
console.log ('b');
```

他就会只输出 a，不输出 b。因为你中间终止函数了，他就不执行下边的了。

(2) return 的第二点作用就是返回值，比如说

```
function sum() {  
    return 123;  
}
```

```
sum();
```

这时，当你执行函数的时候他就会把 123 返回，但是返回的值没地方放，我们就需要一个变量来存他，比如说 `var num = sum();` 此时输出 num 就会得到 123，这里的 return 即返回值又终止函数，你写在 return 下边的语句依然不管用。一般函数处理完一个操作之后并不是为了打印，都是返回给我们以便后续利用，我们一般用变量来接收他的返回值。

5. 初识作用域：

作用域后边会细讲，在这里只简单说一下，比如说：

```
var a = 1;
```

```
function test() {  
    document.write(a);  
    var b = 2;  
    function sum() {  
        document.write(a);  
        document.write(b);  
        var c = 3;  
    }  
    document.write(c);  
}
```

```
document.write(b);
```

这是一个会报错的代码，写在外边的 `var a = 1` 是全局变量，定义在函数里的变量叫局部变量。因为函数相当于一个房间，函数里边的东西能看到外面的东西，也可以访问、修改等等，但是外边的看不到里边的东西，所以你写在最外边的打印 b 会报错，

test 函数体里可以打印 a，但是打印 c 就会报错，最里边的 sum 函数则可以把 ab 都打印出来。

练习 1：写一个函数，功能是告知你所选定的小动物的叫声。

```
function scream(animal){
    switch(animal){
        case "dog":
            document.write('wang!')
            return;
        case "cat":
            document.write('miao!')
            return;
        case "fish":
            document.write('o~o~o')
            return;
    }
}
```

解析：以前我们跳出 switch 用 break，现在我们可以直接用 return 跳出函数也可以。

练习 2. 定义一组函数，逆转并输出汉字形式。

```
function transfer(target){
    switch(target){
        case '1':
            return '一';
        case '2':
            return '二';
        case '3':
            return '三';
        case '4':
            return '四';
        case '5':
            return '五';
        case '6':
            return '六';
        case '7':
            return '七';
    }
}
```

```
        case '8':
            return '八';
        case '9':
            return '九';
        case '0':
            return '零';
    }
}
function reverse() {
    var num = window.prompt('input');
    var str = "";
    for(var i = num.length - 1; i >= 0; i--) {
        str += transfer(num[i]);
    }
    document.write(str);
}
reverse();
```

解析：我们需要解决两个问题，逆转、输出汉字形式。我们在这里先用 transfer 函数体把数字转化为汉字的功能用 switch 语句写出来并用 return 结束语句并返回汉字，在这个函数里，我们传一个形参 target，待会会给他传实参，我们现在来看第二个 reverse 函数体，先让他弹出对话框输入数字，由于输入的数字是字符串形式（字符串也可以理解成一个数组，例如 var a=“123”，他的每一位也可以拿出来，你输出 a[0] 就得 1，你 a.length 就是 3），所以我们就可以在 for 循环里把他的每一位都拿出来并且让他逆转，先定义一个 str 空串，然后在 for 循环里让 i 等于 num.length-1，先取最后一位，然后让 i 大于等于 0 并自减，这样就把 num 从后往前的把每一位都拿了出来，然后 str+=num[i], str 里就把 num 里的数字逆转了，在这一步，我们把 num[i] 作为 transfer 函数的实参传给 target，那么，str += transfer(num[i]); 就把数字即逆转了又接收了 transfer 里 return 的返回值，把数字转换为汉字了，最后输出 str 调用 reverse 即可。

递归、预编译、真题讲解

1. 递归

其实递归就是一种找规律和找出口的方法，他特别符合人的思维过程。

练习 1: 用函数体写出 n 的阶乘。

```
function mul(n){
    if(n == 1 || n == 0){
        return 1;
    }
    return n * mul(n - 1);
}
mul();
```

备注：这里边没有传实参，实验的时候必须有实参，否则会报错。

解析：我们先来找规律，比如说实参传一个 5 进去，我们就来求 5 的阶乘，首先，我们写的这个函数他的功能就是求 n 的阶乘的，我们发现 5 的阶乘就是 5 乘以 4 的阶乘，4 的阶乘就等于 4 乘以 3 的阶乘，那么， n 的阶乘就是 n 乘以 $n-1$ 的阶乘，我们这个函数就是求阶乘的，所以有 `return n * mul(n - 1);`；好，我们来验证一下， $mul(5) = 5 * mul(4)$ ，由于前边是 `return`，所以这么算肯定还没完，`return` 最后必须返回一个具体的值才可以，所以系统会继续算 $mul(4) = 4 * mul(3)$ ，然后 $mul(3) = 3 * mul(2)$ ， $mul(2) = 2 * mul(1)$ ……他会一直这么循环的算下去，所以递归必须找到一个出口，不然就是死循环，由于我们知道 1 的阶乘等于 1，所以我们用 `if` 语句表示当 i 等于 1 或者 0 的时候返回 1，因为 0 的阶乘是 1，那么，这就可以了，反着推， $mul(1) = 1$ ， $mul(2) = 2 * 1 = 2$ ， $mul(3) = 3 * 2 = 6$ ， $mul(4) = 4 * 6 = 24$ ，那 $mul(5) = 5 * 24 = 120$ 。

注意：递归必须最后用 `return`，`return` 必须返回一个具体的值，所以他会一直循环的计算。

练习 2: 写一个函数，实现斐波那契数列

```
function fb(n){
    if(n == 1 || n == 2){
        return 1;
    }
    return fb(n - 1) + fb(n - 2);
}
fb();
```

备注：这里边没有传实参，实验的时候必须有实参，否则会报错。

解析：斐波那契数列就是第三位等于头两位的和，那第 n 位 $fb(n)$ 就等于 $fb(n - 1) + fb(n - 2)$ ；，好，假如说我们求第 5 位，实参传 5， $fb(5) = fb(4) + fb(3)$ ， $fb(4) = fb(3) + fb(2)$ ， $fb(3) = fb(2) + fb(1)$ ……好，现在我们找出口，因为我们知道前两位是 1，所以当 $n=1$ 或者 $n=2$ 时返回 1，即 $fb(1) = 1$ ， $fb(2) = 1$ ，那么

```
fb(3) = 1+1=2, fb(4) = 2+1=3, fb(5) = 3+2=5.
```

2. 预编译前奏

JavaScript 执行三部曲：我们知道，JS 是解释性语言，解释一行执行一行，但是在解释执行之前，他会进行两个过程，语法分析和预编译，语法分析就是他先会把整篇 js 代码扫描一遍，看看有没有语法错误，例如少写一个括号，标点错误等等，如果有语法错误则一行都不执行，接下来进行的就预编译。预编译结束之后才是解释一行执行一行。在学习预编译之前我们要了解一些知识：

(1) 比如说：

```
test();
function test() {
    console.log('a');
}
```

我们知道，如果把 test() 写在下边的话他会执行，但是我们现在把 test() 写在上边，他其实也会执行，输出 a，我们 js 不是解释一行执行一行吗？为什么写在上边还可以执行呢？其实就是预编译的作用，再比如：

```
console.log(a);
var a = 123;
```

我们如果把输出放在下边，会输出 123，但是反过来放在上边他会输出 undefined，我们假如没有下边的 var a = 123 就是变量未经声明就使用肯定会报错，但是有了之后为啥不报错？而且输出的是 undefined？这也是预编译的作用。鉴于以上两种情况，我们总结出两句话：

第一句：函数声明整体提升

解释：就是不管你函数声明写在哪里，他都会在逻辑上把他提到最前边，所以上边的 test() 写在上边或者下边效果是一样的。

第二句：变量 声明提升

解释：比如说上边的 var a = 123 他其实是两个过程，变量声明加赋值，var a; a = 123; 声明提升就是他会把变量声明 var a 提到逻辑的最前边，所以输出 undefined。但是这两句话太肤浅了，比如说：

```
console.log(a);
function a(a) {
    var a = 234;
    var a = function () {

    }
}
```

```

    a();
}
var a = 123;

```

这个时候 a 得啥？函数是 a，变量是 a，形参也是 a，而且不会报错，那么输出得啥？这就是那两句话解决不了的，那两句话只是把预编译过程中的两个现象抽象出来当方法用来用，其实那不是知识点，等我们学完预编译，这个问题轻松解决。在学习预编译之前，我们还要知道以下两个知识点：

(2) **imply global 暗示全局变量：即任何变量，如果变量未经声明就赋值，此变量就为全局变量所有。**

解释：假如我们直接 `console.log(a)` 这叫变量未经声明就使用肯定会报错，但是我们 `a = 10`，这叫变量未经声明就赋值，系统不会报错，而且你输出 a 得 10. 就好像这个变量被声明了一样，但是这个和声明的变量还是不一样。我们再来看，变量未经声明就赋值，此变量为全局变量，全局对象就是 window，这个 window 属于对象，对象上可以加一些属性，比如说 `window.a=10`，我们 `console.log(window.a)` 就得到 10. 好，现在我们单纯访问 a，即 `console.log(a)` 也得 10. 就是说未经声明就赋值的变量为全局所有也属于 window 所有，其实全局对象就是 window，即 `a=10` 就是 `window.a=10`。

(3) **一切声明的全局变量，全是 window 属性。**

解释：全局上的任何变量即使你声明了也归 window 所有，比如说你在全局 `var b=234`，那么你输出 `window.b` 也是 234，即全局 `var b=234` 就是 `window.b=234`。其实 window 就是全局的域，比如说我们在全局 `var a=123`，当我们要输出 a 的时候去哪里拿呢？就去 window 里拿，其实你 `var` 一个 a 等于 123，就相当于在 window 里边挂了一个 a 等于 123，所以你输出 `window.a` 也是 123，你下一次访问 a 的时候他就会去 window 里边找看有没有这个 a。其实你在全局 `console.log(a)` 访问的 a 就是 `window.a`。

比如说：

```

function test() {
    var a = b = 123;
}

```

```
test();
```

变量赋值是从后往前的，所以这句的过程是先让 `b=123` 然后 `var` 一个 a 等于 b 的值，所以我们这里的 b 属于未经声明就赋值的变量，所以我们输出 `window.b` 得 123，输出 `window.a` 得 `undefined`，因为只有声明的全局变量才属于 window，这里的 a 是局部变量。

3. 函数预编译

(1) 预编译四部曲：

第一步：创建 A0 对象。

第二步：找形参和变量声明，将变量和形参名作为 A0 的属性名，值为 undefined。

第三步：将实参值与形参统一。

第四步：在函数体里找函数声明，值赋予函数体。

下面用几个例子来讲解预编译四部曲。

例 1:

```
function fn(a) {  
    console.log(a); ①  
    var a = 123; ②  
    console.log(a); ③  
    function a() {}; ④  
    console.log(a); ⑤  
    var b = function () {}; ⑥  
    console.log(b); ⑦  
    function d() {}; ⑧  
}  
fn(1);
```

备注：为了方便起见，解析时提到的第几行就是上边的圈几。

解析：这里边又有函数声明，又有变量，而且都叫 a，都提升的话，到底谁再谁前边？谁覆盖谁？这里边有一个覆盖的问题，所以这一块就是预编译发生的过程，预编译发生在函数执行的前一刻，所以等函数执行的时候，预编译已经发生完了。预编译就把这些矛盾调和了，那么，现在就这个例子，我们来讲预编译四部曲。

第一步：创建 A0 对象，A0 对象就是 Activation Object，它是活跃对象，我们可以理解成作用域，但是他翻译过来叫执行期上下文。创建完之后就是 A0{ }（通常第一步都是创建 A0 对象，不重要，重要的是下边的，以后解析时忽略第一步）

第二步：他会去找函数里的形参和变量声明，将变量和形参名作为 A0 的属性名，值为 undefined。在此例中，形参是 a，然后第 2 行有变量声明是 a，第 6 行有个变量声明是 b（虽然后边是函数，但是前面属于变量声明），这里边有两个 a，只记一个，所以：

```
A0{  
  a: undefined,  
  b: undefined  
}
```

第三步：将实参值与形参统一。这里的形参是 a，实参是 1，上一步的 a 是 undefined，所以在这一步把 a 的值改为 1：

```
A0{
a: 1,
b: undefined
}
```

第四步：在函数体里找函数声明，值赋予函数体。第 4 行有一个函数声明 a，第 8 行有个函数声明是 d（第 6 行是函数表达式，不是函数声明），找到了之后，依然把函数名作为 A0 对象的属性名挂起来，值为函数体，所以这时 a 的值改为函数体，再挂个 d，值为函数体。

```
A0{
a: function a() {},
b: undefined,
d: function d() {}
}
```

这样，A0 对象就创建完了，预编译就完了，现在来解释执行：

第 1 行：输出 a，去哪里找 a，就去 A0 对象里找 a（不是在下边的语句找 a，你写的语句是给电脑看的，你想拿东西就必须在电脑定义的冰箱里拿，A0 就是那个冰箱。真正的存储机构是 A0，因为预编译就是把 A0 创建好方便你去用），所以第 1 行输出得 function a() {}

第 2 行：var a = 123，其实这一句不完全执行，因为在预编译的第二步将变量和形参名作为 A0 的属性名，这里就是变量声明提升的过程，变量声明已经被提升上去了，你再声明就不用看了，因为已经看过了，所以第二句只剩下 a=123 没执行，执行后 a 的值改为 123：

```
A0{
a: 123,
b: undefined,
d: function d() {}
}
```

第 3 行：输出 a，去 A0 对象里找，输出得 123.

第 4 行：在预编译时看过了，不用看了

第 5 行：输出 a 得 123

第 6 行：var b 看过了，执行 b = function() {}

```
A0{
a: 123,
b: function() {},
```

```
d: function d() {}  
}
```

第 7 行：输出 b 得 function() {}

第 8 行：看过了

最后输出的结果：function a() {} 123 123 function() {}

例 2：

```
function test(a,b){  
    console.log(a); ①  
    c = 0; ②  
    var c; ③  
    a = 3; ④  
    b = 2; ⑤  
    console.log(b); ⑥  
    function b() {}; ⑦  
    function d() {}; ⑧  
    console.log(b); ⑨  
}  
test(1);
```

解析：执行预编译第二步，A0 对象里的形参为 a、b，变量为 c，值为 undefined：
A0{

```
a: undefined,  
b: undefined,  
c: undefined  
}
```

第三步：实参传 1，a 为 1（把对应的实参传到对应的形参里，没对应就不用管了）

A0{

```
a: 1,  
b: undefined,  
c: undefined  
}
```

第四步：函数声明有 b，d：

A0{

```
a: 1,  
b: function b() {},
```



```
c: undefined,
d: function d() {}
}
```

预编译完毕，现在解释执行：

第 1 行：输出 a 得 1

第 2 行：c 改为 0

```
A0{
a: 1,
b: function b() {},
c: 0,
d: function d() {}
}
```

第 3 行看过了跳过

第 4 行第 5 行：a 改为 3，b 改为 2

```
A0{
a: 3,
b: 2,
c: 0,
d: function d() {}
}
```

第 6 行：输出 b 得 2

第 7 行第 8 行跳过

第 9 行：输出 b 得 2

最后输出结果：1 2 2

例 3：

```
function test(a,b){
  console.log(a); ①
  console.log(b); ②
  var b = 234; ③
  console.log(b); ④
  a = 123; ⑤
  console.log(a); ⑥
  function a() {}; ⑦
  var a; ⑧
```

```
b = 234; ⑨
var b = function() {}; ⑩
console.log(a); ⑪
console.log(b); ⑫
}
test(1);
```

解析：执行预编译第二步，形参有 a、b，变量也有 a、b
A0{

a: undefined,
b: undefined
}

第三步：传参 a 为 1

A0{
a: 1,
b: undefined
}

第四步：函数声明有 a

A0{
a: function a() {},
b: undefined
}

预编译完毕，解释执行

第 1 行：输出 a 得 function a() {}

第 2 行：输出 b 得 undefined

第 3 行：执行 b=234

A0{
a: function a() {},
b: 234
}

第 4 行：输出 b 得 234

第 5 行：执行 a=123

A0{
a: 123,
b: 234

```
}
```

第 6 行：输出 a=123

第 7 行第 8 行跳过，第 9 行 b 没变

第 10 行，执行 b=function() {}

```
A0{
```

```
a: 123,
```

```
b: function() {}
```

```
}
```

第 11 行：输出 a 为 123

第 12 行：输出 b 为 function() {}

最后输出结果：function a() {} undefined 234 123 123 function() {}

4. 全局预编译

全局的预编译和函数里的预编译一样，只不过少了几个步骤：

第一步：创建 G0 对象。（G0 对象就是 Global Object，是全局的执行期上下文，换了个名，其实和 A0 是一样的）

第二步：找变量声明，将变量名作为 G0 的属性名，值为 undefined。

第三步：找函数声明，值赋予函数体。

例 4：

```
var a = 123;
```

```
function a() {};
```

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

解析：执行第二步：

```
G0{
```

```
a: undefined
```

```
}
```

执行第三步：

```
G0{
```

```
a: function a() {}
```

```
}
```

执行 a = 123:

```
G0{
```

```
a: 123
```

```
}
```

最后输出 a 得 123.

其实我们上边讲的 window 就是 G0，G0 就是一个筐，为了我们拿东西方便，这个筐还有一个名就是 window。我们定义完的东西要放到 G0 里边储存，也就是放到 window 里边储存，所以 window 自来就有了这些变量的引用。你在全局访问 a 就拿的是 G0 里的 a 也是 window 里的 a。我们上边讲的未经声明就赋值的变量归 window 所有也就是归 G0 所有。

练习 1:

```
console.log(test); ①
function test(test) { ②
    console.log(test); ③
    var test = 234; ④
    console.log(test); ⑤
    function test() {} ⑥
}
test(1); ⑦
var test = 123; ⑧
```

解析：我们知道，函数的预编译发生在函数执行的前一刻，全局的发生在全局执行的前一刻，所以 G0 比 A0 创建的早，我们先来全局预编译：

全局预编译：

第二步：G0 里有变量 test

```
G0{
test: undefined
}
```

第三步：函数声明有 test，值赋予函数体

```
G0{
test : function test(test){ console.log(test); var test = 234;
console.log(test); function test() {};}
}
```

全局执行：

第 1 行：输出 test 得 function test(test){ console.log(test); var test = 234; console.log(test); function test() {};}

第 2 行声明函数，整个函数体不看了

第 7 行：执行函数，就在执行函数的前一刻，A0 对象产生了

第 8 行：执行 test=123.

```
G0{
```

```
test: 123
```

```
}
```

函数预编译：当 test 执行的前一刻，A0 就产生了。

第二步：形参变量都是 test

```
A0{
```

```
test: undefined
```

```
}
```

第三步：传参，test= 1

```
A0{
```

```
test: 1
```

```
}
```

第四步：函数声明有 test

```
A0{
```

```
test: function test() {}
```

```
}
```

函数执行：

第 3 行：输出 test 得 function test() {}（当 A0 里有这个属性值时就拿 A0 的属性值，没有的话他就会往上找，看看 G0 里有木有）

第 4 行：执行 test = 234

```
A0{
```

```
test: 234
```

```
}
```

第 5 行输出 test 得 234，第 6 行跳过

最后输出结果：function test(test){console.log(test); var test = 234; console.log(test); function test(){};} function test(){} 234

练习 2：

```
global = 100; ①
```

```
function fn(){ ②
```

```
    console.log(global); ③
```

```
    global = 200; ④
```

```
    console.log(global); ⑤
```

```
    var global = 300; ⑥
```

```
}
```

```
fn(); ⑦
```

`var global;` ⑧

解析：全局预编译：

第二步：变量有 `global`

```
G0{  
global: undefined  
}
```

第三步：函数有 `fn`

```
G0{  
global: undefined,  
fn: function fn(){console.log(global);global = 200;console.log(global);var  
global = 300;}  
}
```

全局执行第 1 行：`global=100;`

```
G0{  
global: 100,  
fn: function fn(){console.log(global);global = 200;console.log(global);var  
global = 300;}  
}
```

下边的跳过，函数预编译：

第二步：形参没有，变量有 `global`

```
A0{  
global: undefined  
}
```

第三步第四步跳过，没有实参和函数声明

第 3 行：输出 `global` 得 `undefined`

第 4 行：执行 `global=200`

```
A0{  
global: 200  
}
```

第 5 行：输出 `global` 得 200

第 6 行：执行 `global=300`

```
A0{  
global: 300  
}
```


最后结果: undefined 200

练习 3:

```
function test() { ①
  console.log(b); ②
  if(a) { ③
    var b = 100; ④
  }
  console.log(b); ⑤
  c = 234; ⑥
  console.log(c); ⑦
}
var a; ⑧
test(); ⑨
a = 10; ⑩
console.log(c); ⑪
```

解析: 全局预编译第二步: 变量有 a

```
G0{
a: undefined
}
```

第三步: 函数有 test

```
G0{
a: undefined,
test : function test() {console.log(b);if(a){var b = 100;}c =
234;console.log(c);}
}
```

第 8 行之前跳过了, 但是, 记住, 在第 9 行执行的前一刻有了 A0, 这个时候 a 是 undefined。

我们先来看函数预编译, 第二步: 变量有 b (预编译的时候不看 if, 不管能不能执行, 你 var 一个 b 就得挂起来)

```
A0{
b: undefined
}
```

第三步没实参, 第四步没函数, 跳过

第 2 行: 输出 b 得 undefined

第3行：a 是 undefined，假的，所以 if 里边根本执行不了。

第5行：输出 b 得 undefined

第6行：c 属于未经声明就赋值的变量，归 GO 所有：

```
GO{
a: undefined,
test : function test() {console.log(b);if(a) {var b = 100;} c =
234;console.log(c);},
c: 234
}
```

第7行：输出 c 得 234

第10行：执行 a=10

```
GO{
a: 10,
test : function test() {console.log(b);if(a) {var b = 100;} c =
234;console.log(c);},
c: 234
}
```

第11行：输出 c 得 234。

最后结果：undefined undefined 234 234

5. 真题讲解

练习 4：（百度试题）

```
(1) function bar() {
    return foo;
    foo = 10;
    function foo() {

    }
    var foo = 11;
}
console.log(bar());
(2) console.log(bar());
function bar() {
    foo = 10;
    function foo() {
```

```

    }
    var foo = 11;
    return foo;
}

```

解析：第（1）题最上边 `return foo`，在外边输出函数执行，就相当于 `console.log(foo)`，我们做了这么多练习你会发现写在最上边的输出如果底下有声明函数名跟他一样的，一定输出函数，因为他是第四步优先级最高，所以输出得 `function foo() {}`，第（2）题，不管你怎么预编译，只要最后被重新赋值了，就把前边的值覆盖了，所以输出得 11。

练习 5：（经典题，长度过瘾）

```

a = 100; ①
function demo(e) { ②
    function e() {}; ③
    arguments[0] = 2; ④
    document.write(e); ⑤
    if(a) { ⑥
        var b = 123; ⑦
        function c() {} ⑧
    }
    var c; ⑨
    a = 10; ⑩
    var a; ⑪
    document.write(b); ⑫
    f = 123; ⑬
    document.write(c); ⑭
    document.write(a); ⑮
}
var a; ⑯
demo(1); ⑰
document.write(a); ⑱
document.write(f); ⑲

```

备注：两年前 `if` 语句里边套函数是可以的，但是现在不可以了，我们这里就当他可以套。

解析：全局预编译第二步：变量有 a：

```
G0{  
a: undefined  
}
```

第三步：函数有 demo：

```
G0{  
a: undefined,  
demo: function demo(e) {……（函数体太多不写了）}  
}
```

全局执行：

第 1 行：执行 a=100,：

```
G0{  
a: 100,  
demo: function demo(e) {……（函数体太多不写了）}  
}
```

第 16 行之前直接跳过，执行第 17 行之前，有了函数的预编译：

第二步：形参有 e，变量有 a，b，c：

```
A0{  
e: undefined,  
a: undefined,  
b: undefined,  
c: undefined  
}
```

第三步：实参传 1：

```
A0{  
e: 1,  
a: undefined,  
b: undefined,  
c: undefined  
}
```

第四步：声明函数有 e，c（其实后来 if 语句里边不能放函数，这里 c 是 undefined，我们姑且认为是 function c() {}）：

```
A0{  
e: function e() {},
```

```

a: undefined,
b: undefined,
c: undefined (function c() {} )
}

```

执行第 3 行跳过，第 4 行 `arguments[0] = 2`，也就是实参的第一位变成 2，那么 e 就是 2。

```

A0{
e: 2,
a: undefined,
b: undefined,
c: undefined (function c() {} )
}

```

第 5 行：打印 e 为 2。

第 6 行：a 是 undefined，假的，if 语句跳过不执行。

第 9 行跳过，第 10 行执行 `a=10`：

```

A0{
e: 2,
a: 10,
b: undefined,
c: undefined (function c() {} )
}

```

第 11 行跳过。

第 12 行打印 b 为 undefined。

第 13 行 f 未经声明就赋值属于 G0：

```

G0{
a: 100,
demo: function demo(e) {…… (函数体太多不写了) },
f: 123
}

```

第 14 行打印 c 为 `undefined (function c() {})`。

第 15 行打印 a 为 10（取 A0 里的 a）。

第 18 行打印 a 为 100（取 G0 里的 a）。

第 19 行打印 f 得 123。

最后结果：2 undefined undefined (function c() {}) 10 100 123

注意：以后 if 里不能写函数哈！

练习 6：（成哥原创精品）

```
var str = false + 1; ①
document.write(str); ②
var demo = false == 1; ③
document.write(demo); ④
if(typeof(a) && -true + (+undefined) + "") { ⑤
    document.write('基础扎实'); ⑥
}
if(11 + "11" * 2 == 33) { ⑦
    document.write('基础扎实'); ⑧
}
!!" " + !!"" - !!false || document.write('你觉得能打印，你就是猪!'); ⑨
```

解析：这道题主要考的是隐式类型转换，

第 1 句：false + 1，加号两边没有字符串，所以调用 Number，false 转换为数字为 0，0 + 1 = 1。

第 2 句：打印 str 得 1。

第 3 句：因为“=”的优先级最弱，所以先算 false == 1，false 转换为数字是 0，0 不等于 1，所以右边算出来得 false，然后再赋给 demo。

第 4 句：打印 demo 得 false。

第 5 句：我们说过，typeof(a)，a 没定义，但是放在 typeof 里边不会报错，输出 undefined，但是这个 undefined 是字符串类型的，所以转换为布尔值为 true，右边调用 Number 计算，-true 就是 -1，+undefined 得 NaN，-1 + NaN 得 NaN，NaN 加空串得字符串类型的 NaN，所以转换为布尔值为 true，中间是 && 运算符，两边都为真则返回真，所以第 6 句可以执行。

第 7 句：有加有乘先算乘法，乘法就调用 Number，所以“11” * 2 得 22，22 + 11 == 33，第 8 句可以执行。

第 9 句：在这里一定要区分开带空格的字符串和空串的区别，!! 就是转换为布尔值，所以第一个带空格的字符串转换后为 true，空串转换后为 false，所以就是 true + false - false 等于 1 + 0 - 0 等于 1，转换后为 true，|| 遇到真就停，所以后边的执行不了。

练习 7：请写出 window.foo 的值。（百度外卖试题）

```
(window.foo || (window.foo = 'bar'))
```

解析：这个外边的括号基本没用，如果没有里边的括号就会报错，但是加上之后先算

括号里边的，先算右边的，把 bar 赋给 foo，再来左边的，所以 foo 的值为 bar。

作用域、作用域链精解

1. 我们之前讲过，一个函数就相当于一个屋子，他和外边有阻隔，函数里边的东西能看到外边的东西，但是外边的看不到里边的东西，我们可以把函数所生成的这种空间就叫做作用域，但是这么叫不精准，作用域的确是随着一个函数的产生而产生的，现在就来探索一下作用域。

(1) 执行期上下文（就是我们之前讲的 AO）：当函数执行的前一刻，会创建一个称为执行期上下文的内部对象，一个执行期上下文定义了一个函数执行时的环境，函数每次执行时对应的执行期上下文都是独一无二的，所以多次调用一个函数会导致创建多个执行期上下文，当函数执行完毕，他创建的执行期上下文被销毁。

解释：我们每次执行函数的前一刻系统都会创建 AO 对象，这个 AO 就定义了变量提升函数提升等，比如说定义了一个函数 test，当你调用他的时候他会创建一个 AO，当你再次调用的时候他还会再次创建另一个 AO，AO 只是一个临时的存储对象，当函数执行完毕时，AO 就会被销毁。

(2) `[[scope]]`：每个 JavaScript 函数都是一个对象，对象中有些属性我们可以访问，但有些不可以，这些属性仅供 JavaScript 引擎存取，`[[scope]]`就是其中一个，`[[scope]]`指的就是我们所说的作用域，其中存储了运行期上下文的集合。

(3) 作用域链：`[[scope]]`中所存储的执行期上下文对象的集合，这个集合呈链式链接，我们把这种链式链接叫做作用域链。在哪个函数里边查找变量，就去哪个函数的作用域链顶端依次向下查找。

例 1：

```
function a() {
  function b() {
    var b = 234;
  }
  var a = 123;
  b();
}
var glob = 100;
a();
```

解析：当 a 函数被定义的时候，函数就有自己的属性，当然就有 `[[scope]]`，我们就叫他 a. `[[scope]]`，当 a 刚被定义的时候，这个作用域链只有一个东西（如果我们把作用域理解成一个数组，那么就是第 0 位），就是 G0：

a 函数被定义: a. `[[scope]]`----->0: G0{a: function; glob: 100}

接下来, 当 a 函数执行的前一刻, 就产生了 A0, 系统就会把 A0 放到作用域链的最顶端, 把刚才的 G0 往下放一格 (把 A0 放到第 0 位, G0 放到第 1 位):

a 函数执行前: a. `[[scope]]`----->0: a 的 A0{a: 123; b: function}

1: G0{a: function; glob: 100}

这时, 如果你要在 a 函数里访问一个变量, 他就会在 a 的 scope 里边找, 从作用域链的最顶端依次往下寻找 (先找第 0 位, 再找第 1 位)

然后, 当 a 执行的时候有了 b 函数的定义, 由于 b 也是函数, 所以他也有自己的作用域, 那么, 当 b 被定义的时候, 由于他在 a 函数里边, 所以他作用域链最开始的时候有 a 的 A0, 还有 G0, 等于说 b 最开始被定义的时候, 他拿的是 a 函数的劳动成果:

b 函数被定义: b. `[[scope]]`----->0: a 的 A0{a: 123; b: function}

1: G0{a: function; glob: 100}

然后, 当 b 函数执行前, 他也创建了自己的 A0, 他会把自己的 A0 放到作用域链最顶端, 其他的往下移:

b 函数执行前: b. `[[scope]]`----->0: b 的 A0{b: 234}

1: a 的 A0{a: 123; b: function}

2: G0{a: function; glob: 100}

这时, 如果你在 b 函数里边访问一个变量, 他就会在 b 的作用域链里边从上往下 (从第 0 位到第 2 位) 依次寻找。

注意: 例如上边这个例子, 假如说我们在 b 函数里写一句 `a = 0`, 然后在 a 函数里等 b 函数调用之后我们输出 a:

```
function a() {
  function b() {
    var b = 234;
    a = 0;
  }
  var a = 123;
  b();
  console.log(a);
}
var glob = 100;
a();
```

此时输出 a 得 0, 这就说明 a 产生的这个 A0 和 b 访问的这个 A0 它是一个东西, b 其实拿的就是 a 的一个引用。

然后，b 函数执行完之后，他就会销毁自己的 A0，回归到被定义的状态，作用域链里边只剩下 a 的 A0 和 G0，然后等待下一次被执行（下一次被执行时他又会创建一个新的 A0 放到作用域链最顶端），a 函数执行完也要销毁自己的 A0 回归到被定义状态，这时，由于 a 的 A0 里有 b 函数，那么销毁后连 b 函数都没了，然后等到 a 下一次被执行时，他又会产生一个新的 A0，新的 A0 里有一个新的 b 函数定义，新的 b 函数会拿 a 的劳动成果，然后执行 b 函数时，他又会产生一个新的 A0 放到作用域顶端方便查找变量，周而复始。。。

例 2:

```
function a() {
  function b() {
    function c() {

    }
    c();
  }
  b();
}
a();
```

解析:

a 函数被定义: a. `[[scope]]`----->0: G0

a 函数执行前: a. `[[scope]]`----->0: a 的 A0
1: G0

b 函数被定义: b. `[[scope]]`----->0: a 的 A0
1: G0

b 函数执行前: b. `[[scope]]`----->0: b 的 A0
1: a 的 A0
2: G0

c 函数被定义: c. `[[scope]]`----->0: b 的 A0
1: a 的 A0
2: G0

c 函数执行前: c. `[[scope]]`----->0: c 的 A0
1: b 的 A0
2: a 的 A0
3: G0

然后当你要在哪个函数里边访问变量，就去哪个作用域链从上往下查找即可。这个就很形象的解释了里边的可以访问外边的，外边的访问不了里边的。

注意：假设 a 不调用执行，那就不会产生函数 b，如果不调用，里边的代码系统是不会看的。

闭包精解、立即执行函数、真题讲解

1. 闭包

(1) 先来看个例子

例 1:

```
function a() {  
    function b() {  
        var bbb = 234;  
        console.log(aaa);  
    }  
    var aaa = 123;  
    return b;  
}  
  
var glob = 100;  
var demo = a();  
demo();
```

解析：我们来看这个例子，简便的分析一下，在全局有一个函数 a，变量 glob、demo，然后 a 执行，执行前 a 的作用域链里放的是 a 的 A0 和 G0，然后在 a 函数里有一个函数 b 和变量 aaa，b 函数最开始拿的是 a 的劳动成果（a 的 A0 和 G0），但是最重要的是，b 函数没有被执行，在 a 函数结束的时候把 b 函数返回了出去，在全局 var 一个 demo 等于 a 执行，就是说现在把 a 函数的返回值赋给了 demo，也就是说 demo 里边现在装的是 b 函数。好，a 函数现在执行完了，他要销毁自己的 A0，我们看着好像就连同 b 函数一块儿被销毁了，然而并没有，全局的 demo 现在就是里边的函数 b，而且手里还攥着 a 的劳动成果，就是说虽然 a 把自己的 A0 销毁了，然而 b 手里还有 a 的 A0 和 G0 呢，而且还被保存到了外部的 demo 里，所以你在外边执行 demo，就等于执行函数 b，执行时要访问 aaa，先创建自己的 A0 放到作用域链最顶端，然后发现自己的 A0 里没有 aaa，然后往下找，找到 a 的 A0 有变量 aaa，然后输出得 123，我们视觉上看着好像 b 函数被保存到了外部，再去访问函数里的 aaa 好像是不行的，然而事实上是可以的，这个过程就叫做闭包。

注意：这里把 b 扔给 demo 扔的是引用值，就等于把 b 的地址扔给 demo 了。demo 和 b

就相当于同一个人的不同名字，所以 demo 执行就等于 b 执行。

闭包的产生：但凡内部函数被保存到了外部，它一定生成闭包。例如上边的，内部函数会保存外部函数的劳动成果，然后还没执行呢，就通过 return 的方式被保存到了外部，一保存到外部，那么 a 函数想删除自己的 AO 都删除不了了。因为 b 始终攥着呢。

(2) 闭包的应用：

例 2：

```
function a() {
    var num = 100;
    function b() {
        num++;
        console.log(num);
    }
    return b;
}
var demo = a();
demo();
demo();
```

解析：a 执行时产生了自己的 AO 和 GO，然后 b 在定义的时候拿到 a 的劳动成果，a 函数最后把 b 返回给了 demo，然后 a 销毁自己的 AO，然而 demo 现在就是 b 函数，他还保存着 a 的 AO 和 GO，所以 demo 第一次执行的时候 b 先创建自己的 AO，然后自己的 AO 里边没有 num，就去 a 的 AO 里拿，然后++，输出得 101，销毁自己的 AO，然后第二次执行时在创建自己的 AO，没有 num，然后去 a 的 AO 里拿，此时 num 是 101，再++，输出得 102，这里每次执行创建和销毁的是 b 的 AO，然而里边并没有东西，他始终拿的是 a 的 AO 里的 num，a 的 AO 一直存在着。

(3) 闭包的现象

当内部函数被保存到外部时，就会生成闭包，闭包会导致原有作用域链不释放，造成内存泄露。

解释：例如上边的例 1，本来 a 执行完之后就会销毁自己的 AO，但是 b 被扔出来之后他说你销毁了也是白扯，我还拿着呢！所以他就会导致作用域链该释放的时候不释放，你有意做的闭包还好，假如闭包是无意做的，那就很占内存空间了。所以就会造成内存泄露，什么叫内存泄露？可以这么理解，你占得内存多了，剩余的内存就少了，我们关注剩余的内存，他是不是越来越少？就好像泄露了一样。所以这得反向的理解。

(4) 闭包的作用

作用一：实现公有变量，比如函数累加器：

例 3:

```
function add() {  
    var count = 0;  
    function demo() {  
        count ++;  
        console.log(count);  
    }  
    return demo;  
}  
  
var counter = add();  
counter();  
counter();  
counter();  
.....
```

解析：我们用闭包就可以做一个函数累加器，在 add 里边先定义一个变量 count，然后定义一个函数 demo，demo 里实现 count++，最后把 demo 扔给全局的 counter，自此之后 counter 就会把 count 无限用了，你每调用一次 counter 他就会在原有的基础上加一次，这样做就更加模块化，你每执行一次方法他就会累加一次，结构化更好。

作用二：可以做缓存（存储结构）

例 4:

```
function test() {  
    var num = 100;  
    function a() {  
        num ++;  
        console.log(num);  
    }  
    function b() {  
        num --;  
        console.log(num);  
    }  
    return [a,b];  
}  
  
var myArr = test();  
myArr[0]();
```



```
myArr[1]();
```

解析：首先定义一个 test 函数，test 里边 var 一个 num 等于 100，还放着两个函数 a 和 b，a 函数里边让 num++，b 函数里边让 num--，然后在 test 最后通过数组的方式把 a 和 b 都扔给外部的 myArr，这也可以形成闭包，你在下边执行 myArr 的第 0 位就等于 a 函数执行，执行 myArr 的第 1 位就等于执行 b 函数，先执行 a 函数，a 拿的是 test 的 A0，num++后输出得 101，再执行 b 函数，b 和 a 同级，拿的也是 test 的 A0，由于刚才 num 被 a++后得 101，所以再一输出得 100。

例 5:

```
function eater() {
    var food = "";
    var obj = {
        eat:function () {
            console.log("I am eating " + food);
            food = "";
        },
        push:function (myFood){
            food = myFood;
        }
    }
    return obj;
}
```

```
var eater1 = eater();
eater1.push("banana");
eater1.eat();
```

解析：首先定义一个 eater 函数，里边 var 一个 food 等于空串，然后定义一个 obj，他是一个对象，其实对象里边也可以有函数，只不过定义方法不一样，obj 里有一个 eat，eat 里放的是函数，函数里先输出 I am eating 加上 food 里的内容，在让 food 清空，然后还有一个 push 函数，函数里把形参 myFood 的值赋给 food，最后在 eater 里边把 obj 返回，扔给全局的 eater1，这就相当于把里边的两个函数一起返回了，这也可以形成闭包，因为两个函数操作的都是同一个 food，先执行 push 并传参 banana，那么 myFood 就是 banana，最后在赋给 food，在执行 eat，此时 food 等于字符串的 banana，所以输出得 I am eating banana，这里边的 food 就是一个隐式的存储结构，这就是缓存的一个应用。

作用三：可以实现封装，属性私有化（高级应用，后边圣杯模式那里讲）

作用四：模块化开发，防止污染全局变量（高级应用，后边命名空间那里讲）

2. 立即执行函数

比如说我现在在全局 `function a() {}`, `function b() {}`, 这两个函数假如说写在全局，如果 JavaScript 不执行完，这两个函数永远都是等待被执行，永远释放不了，假如定义一个函数你只想用一次，后边就不用他了，他还一直写在全局不释放，这样的话就很占空间了，鉴于此，JS 给我们提供了一类函数，叫**立即执行函数**，**这种函数执行完马上就释放了，是针对初始化功能的函数，不占用内存空间。**

(1) 形式：

```
(function () {
```

```
})();
```

先写一对小括号，然后里边写上 `function`，都不用起名，然后小括号大括号，最后来个小括号，他也是 js 里唯一一个执行完立即销毁的函数。他和普通函数除了执行完就被销毁之外没有任何区别。他也可以有参数，比如说：

```
(function (a,b) {  
    console.log(a + b);  
})(1,2))
```

输出就得 3，他同样也有返回值，

```
var num = (function (a,b) {  
    var d = a + b;  
    return d;  
})(1,2))
```

我们在外边 `var` 一个 `num` 接收返回值，那么，`num` 就是 3。

(2) 写法：

第一种： `(function () {} ())` W3C 组织建议这一种写法

第二种： `(function () {} ())()`

(3) 拓展：

只有表达式才能被执行符号执行。

比如说：

```
function test() {  
    console.log(1);  
}()
```

这么写系统会报错，因为这个是函数声明不是表达式，你必须在底下单独写 `test()`；下边写的 `test` 才叫做表达式，像 `123`、`234`、`1 + 2` 这都叫表达式，再比如：

```
var test = function () {
    console.log(1);
}()
```

这么写就可以执行，因为它本身就是函数表达式。能被执行符号执行的表达式他的函数名就会被自动放弃，也就是说能被执行符号执行的表达式他就成了立即执行函数，就像上边的，你在后边加了括号就是立即执行，他就成了立即执行函数。再比如：

```
+ function test() {
    console.log(1);
}()
```

本来他是函数声明，但是你在前边加上正号他就理论上转换为数字了，那他就是一个表达式，你再后边直接加()就会执行，变成立即执行函数，然后忽略函数名，你执行完后输出 test 就会报错，当然前边加上-或者!都可以，&&和||也可以，不过两边都要有东西，但是*和/不行，这里的+-代表正负号。再比如：

```
(function test() {
    console.log(1);
})();
```

我们如果在外边加一个小括号，他也就成了一种表达式，因为我们计算的时候，有时候有小括号，那么，他就会自动放弃函数名，后来人们一看，有没有函数名都无所谓，所以后来就去掉了，成了现在的立即执行函数（执行符号()放在括号里边或者外边都可以，W3C 组织建议放在里边）。

练习 1：（阿里巴巴笔试题）

```
function test(a,b,c,d){
    console.log(a + b + c + d);
}(1,2,3,4)
```

解析：这是阿里巴巴一道非常恶心的题，这么写理论会报错，但是最后括号里有东西，还有逗号，所以系统把括号里的东西理解成表达式，把他们分开：

```
function test(a,b,c,d){
    console.log(a + b + c + d);
}
(1,2,3,4)
```

括号里的东西写在下边也没啥意义，所以这个题既不报错也不执行。

3. 闭包的解决办法 先来看下边这个例子：

```
function test() {
    var arr = [];
```

```
for(var i = 0;i < 10;i ++){
    arr[i] = function (){
        document.write(i + " ")
    };
}
return arr;
}
var myArr = test();
for(var j = 0;j < 10;j ++){
    myArr[j]();
}
```

解析：定义一个 test 函数，函数里先 var 一个 arr 等于一个空数组，然后我们写一个 for 循环，循环里用遍历数组的方式让数组的每一位都等于一个函数，并让他转十圈，等于说循环结束后数组里放了十个这样的函数，最后我们把数组返回给外部的 myArr，现在 myArr 就是 arr 数组，然后我们在外部写一个 for 循环，同样用遍历数组的方式让 myArr 的每一位都执行，等于说让里边的十个函数都执行。结果打印十个 10，那么，问题来了，为什么会打印出 10？而且全是 10？原因是这样的：

这个 for 循环转了十圈就执行了十次，执行十次就产生了十个彼此独立的函数，并且把这十个函数放到了数组里，最后还把这个数组返回了，那么这十个函数和 test 都形成了闭包，也就是十对一的闭包，那么这十个函数要访问 test 里边的变量的时候访问的都是同一套，test 里边有变量 arr 和 i，虽然 i 写在循环里，但是他也属于 test 的变量，所以这十个函数访问的是同一个 i，那么，i 到底是几？我们在外部访问的就是说等到 test 执行完才访问的 i，那么，在 for 循环里当 i 加到 9 的时候， $9 < 10$ ，还能执行一遍，最后 i 再++一下变成了 10， $10 < 10$ 是假的，循环结束，但是此时 i 已经加到 10 了，只有 $i=10$ ，循环才能结束，所以打印出十个 10。

那么，问题来了，`arr[i] = function () {document.write(i + " ")};`这一句的时候前边的 i 变现后边的不变吗？对的，因为这是一条赋值语句，数组的当前位我们是要马上索取出来的，但是后边是一个函数体啊，他又不是现在就执行，你不执行凭啥要里边这个 i 现在就变现呢？如果不执行，系统都不知道里边装的是啥，他就是一个函数体，等到最后执行的时候他才会去看 i 得几，但是这个时候 i 已经变成 10 了。

闭包解决办法：就像上边这个例子就属于一个不好的闭包，因为我们本来想打印出 0123456789，结果打印了十个 10，那怎么解决这个问题呢？我就想打印 0123456789，而且还要保存到外部，其实我们可以这么做：

```
function test() {
```

```

var arr = [];
for(var i = 0;i < 10;i ++){
    (function (j){
        arr[j] = function (){
            document.write(j + " ")
        }
    })(i)
}
return arr;
}
var myArr = test();
for(var j = 0;j < 10;j ++){
    myArr[j]();
}

```

解析：我们在 for 循环里加一个立即执行函数，形参传一个 j 进去，实参传 i，把里边的 i 换成 j 就完美解决了。是这样的，现在 for 循环里写了一个立即执行函数，并且转了十圈，等于说现在这里边有十个这样的立即执行函数。好，我们来看第一个立即执行函数，这时候 i 必须得变现，因为他是实参啊，所以 i 现在得 0，那么形参 j 就是 0，里边的 arr[j] 这里的 j 也得变现得 0，后边的是函数体，j 就不变现了，第二个立即执行函数，i 得 1，形参 j 就是 1，arr[j] 的 j 就是 1，后边的不变现……现在数组被保存到了外部，在外部执行了：现在形成了十个立即执行函数，每个立即执行函数里包裹了一个被保存到外边的小函数，也就是说现在是十个函数对十个函数，是一一对应的关系，然后第一个函数找 j 的时候找的是第一个立即执行函数里边的 j，第二个函数找 j 的时候找的是第二个立即执行函数里边的 j……因为虽然立即执行函数执行完被销毁了，但是他里边套的这个函数是可以拿到他的 AO 的，所以有十个 j 和他对应了，这个 j 是被每一圈变了现了，每一个立即执行函数里边的 j 是不会再变了，所以在外部我们找 j 的时候找的就是原本对应的那个 j 了。所以输出得 0 1 2 3 4 5 6 7 8 9。

4. 真题讲解

练习 2：（阿里巴巴 UC 移动事业群社招笔试题）

```

<ul>
  <li>a</li>
  <li>a</li>
  <li>a</li>

```

```
<li>a</li>
</ul>
```

使用 `addEventListener`，给每个 `li` 元素绑定一个 `click` 事件，输出他们的顺序。

```
function test() {
    var liCollection = document.getElementsByTagName('li')
    for (var i = 0; i < liCollection.length; i++) {
        (function (j) {
            liCollection[j].addEventListener("click", function () {
                console.log(j);
            }, false)//这个语法是事件，后期会讲
        })(i)
    }
}
test();
```

解析：`var liCollection = document.getElementsByTagName('li')` 这一句的作用就是把所有的 `li` 都选中，后期会讲到，然后现在 `liCollection` 就相当于一个数组，里边放了四个 `li`，我们拿 `for` 循环用遍历数组的方式给每个 `li` 绑定一个事件，`click` 就是鼠标点击的时候我们定义一个函数功能是输出当前位的索引，虽然说 `liCollection` 在函数里边，但是我们通过 `click` 把这个函数绑定到了外部的 `li` 身上，等于说还是被保存到了外部，会产生闭包，所以我们用立即执行函数来解决这个问题，如果没有立即执行函数，挨个点击会输出四个 4，和上边的例子是一样的，加上立即执行函数挨个点下来就会输出 0 1 2 3。在以后开发的时候，当输出的东西本来想按顺序输出的，结果输出了同样的数，那多半就是闭包的问题，凡是遇到类似这种情况，都要用立即执行函数解决。

练习 3：（2017 微店校招笔试题）【腾讯旗下】

```
var f = (
    function f() {
        return "1";
    },
    function g() {
        return 2;
    }
)();
console.log(typeof f);
```


解析：这里要讲的一点是逗号操作符，其实有时候逗号也可以当操作符用，但是两边必须加上括号，他会返回后边表达式的值，比如说 `var a = (1, 2)`，那么 `a` 就得 2，所以这道题 `f` 等于 `g` 函数的返回值，输出得 `number`。

练习 4：（2017 微店校招笔试题）【腾讯旗下】

```
var x = 1;
if (function f() { }) {
    x += typeof f;
}
console.log(x);
```

解析：这道题是非常考验基础知识的题，首先 `if` 括号里放的是条件，这个函数声明不属于 `false` 里的那六个值，所以这个语句是可以执行的，那既然放到括号里当做判断条件来用了，他就成了表达式了，就不是函数声明了，再看 `typeof f`，我们以前讲过，变量未经声明就使用肯定报错，但是加上 `typeof` 就不会报错，而且输出字符串类型的 `undefined`，所以最后输出得 `1undefined`。

对象、包装类、真题讲解

1. 对象

我们之前讲过对象，对象就是一种基本的变量类型，他和这个数组、`function` 等等，这些都属于引用值。对象里有一些属性和方法，他们之间可以互相更改。

（1）我们用所学的知识，描述一下心目中的对象，我们来描述一下可爱的邓哥
例 1：

```
var mrDeng = {
  name: "MrDeng",
  age: 40,
  sex: "male",
  health: 100,
  smoke: function () {
    console.log('I am smoking ! cool!!!');
    this.health--;
  },
  drink: function () {
    console.log('I am drink');
    this.health++;
  }
}
```

}

例如上边的，对象里的格式是先写属性名，加一个冒号，后边写上属性值（当然属性名后边也可以跟上一个 function 方法），中间用逗号隔开，这时候我们调用 `mrDeng.age` 就得到 40，我们也可以给属性重新赋值，例如 `mrDeng.age = 50`；这个我们之前都讲过。好，现在我们定义两个方法，当老邓抽烟的时候，输出 `I am smoking ! cool!!!`，并让他的健康值 `healthn` 减一（这里函数里边不能直接写 `healthn--`，必须指定是谁的 `healthn`，所以这里可以写成 `mrDeng.healthn--`，但是他可以写成第一人称，叫 `this`，`this` 有很多用法，后期会讲到，这里用 `this` 来表示这个对象 `mrDeng`），当老邓喝酒的时候，输出 `I am drink`，并让他的健康值加一。现在我们输出 `mrDeng.smoke` 得话就会输出对应的函数体，但是我们 `mrDeng.smoke()`，这就调用了后边的函数，输出对应的句子后我们再访问 `mrDeng.healthn` 就会得到 99，这时我们继续调用 `mrDeng.drink()`，输出对应的句子后，再访问 `mrDeng.healthn` 就会得到 100。

（2）属性的增删改查：我们继续用上边的例子来讲

增：如果需要给一个对象增加一个属性，就直接对象. `xxx=xxx` 即可，例如：`mrDeng.wife='xiaoaliu'`；这样这个属性就加到对象里边了。

删：如果需要删除对象的一个属性，我们需要用到一个关键字叫 `delete`，`delete` 后边跟上对象的属性即可，例如：`delete mrDeng.age`；你再去访问 `mrDeng.age` 就会得到 `undefined`（注意：变量未经声明就使用肯定报错，但是在对象里不会报错，而且输出 `undefined`，比如你访问 `mrDeng.abc` 就得到 `undefined`）

改：就是重新赋值，上边讲到了。

查：就是查看、访问，上边也讲到了。

知道了增删改查，我们再用老邓来举一个例子：

例 2：

```
var deng = {
  name: "laodeng",
  sex: "male",
  prepareWife: "xiaowang",
  gf: "xiaoaliu",
  wife: "",
  getMarried: function () {
    this.wife = this.gf;
  },
  divorce: function () {
    delete this.wife;
  }
}
```

```

    this.gf = this.prepareWife;
  },
  changePrepareWife: function (someone) {
    this.prepareWife = someone;
  }
}

```

解析(皮): 这就是老邓的幸福生活, 其中 prepareWife 代表女朋友, gf 代表结婚对象, wife 代表妻子, getMarried 代表结婚, divorce 代表离婚, changePrepareWife 代表找女朋友。好, 最开始老邓的结婚对象 gf 是 xiaoliu, 外边还养了一个女朋友 prepareWife 叫 xiaowang, 由于还没有结婚, 所以妻子 wife 是空的。现在老邓要结婚, 即 deng.getMarried(), 函数里把结婚对象 gf 的值赋给妻子 wife, 所以结婚后我们访问 deng.wife 就得到 xiaoliu, 过了一段时间, 老邓要离婚, 即 deng.divorce(), 函数里先删除妻子 wife 属性, 然后把女朋友 prepareWife 的值赋给结婚对象 gf, 所以此时老邓的结婚对象就是 xiaowang, 我们访问 deng.wife 就是 undefined, 过了几天邓哥又结婚了, 即 deng.getMarried(), 我们访问 deng.wife 就是 xiaowang, 这时老邓和小王结婚了, 女朋友的位置就空出来了, 他决定在找一个, 即 deng.changePrepareWife("xiaozhang"), 找女朋友的函数里传了一个形参 someone, 我们让 someone 等于女朋友 prepareWife, 调用的时候实参传 xiaozhang, 现在他的女朋友就是 xiaozhang, 然后邓哥又离婚, 即 deng.divorce(), 此时 deng.wife 就是 undefined, 结婚对象 gf 就是 xiaozhang, 然后他又结婚 deng.getMarried(), 此时访问 deng.wife 就是 xiaozhang。

(3) 对象的创建方法

第一种: plainObject 对象字面量/对象直接量

这种方法就是我们一直讲的 var obj = {...}

第二种: 构造函数创建方法

1) 系统自带的构造函数 Object

这个 Object 就是一个系统自带的函数, 这个函数可以执行, Object() 就可以执行这个函数, 但是这样执行没啥用, 我们想利用这个函数产生对象的话, 必须在系统自带的构造函数执行前边加个 new, 即 new Object(), 这样在 new 后边加上一个构造函数方法的执行他就可以产生一个实实在在的对象, 在这个方法执行的时候当然有个返回值, 所以我们在外部 var 一个变量来接收他, 如 var obj = new Object(), 其实这种方法和对象字面量的创建方法是一样的, 后期如果我们要加东西的话, 直接在后面写, 如 obj.name = "abc" 即可 (等号和冒号区分开, 冒号是写在大括号里的, 大括号外边用等号)。构造函数在每次调用的时候都会产生一个对象, 产生的多个对象都是一样的,

而且彼此是独立的。

2) 自定义

例如：

```
function Person() {  
  
}
```

```
var person = new Person();
```

这样的话我们就创建了一个构造函数，构造函数其实和普通函数没有区别，但是我想让构造函数产生对象的话，必须借用 `new` 操作符才能产生一个空对象，和上边的是一样的。

构造函数命名规则：由于和普通函数没有任何区别，所以我们在命名的时候把他们区分开，遵循大驼峰式命名规则，就是所有的单词首字母都要大写。

例 3：

```
function Car(color) {  
    this.color = color;  
    this.name = 'BMW';  
    this.height = '1400';  
    this.lang = '4900';  
    this.weight = 1000;  
    this.health = 100;  
    this.run = function () {  
        this.health--;  
    }  
}
```

解析：假如说我们在下边写上 `var car = new Car();` 的话，我们在外部访问变量 `car` 就会得到一个对象，对象里有设定好的 `name`、`height`、`lang` 等属性，这种方式就是用 `this` 的方式来构造的，这里的 `this` 也可以理解成第一人称。我们在底下继续写上 `var car1 = new Car();` 这就是通过同一个构造函数产生的两个不同的对象，我们在浏览器访问 `car` 和 `car1` 会得到同样的结果，属性相同，但是 `car` 和 `car1` 是两个人，属性不可以互通，比如说我们在下边写上 `car.name = 'Masarati'; car1.name = 'Merz';` 我们访问 `car` 就会得到 `Car{color: undefined, name: "Masarati", height: "1400", lang: "4900", weight: 1000, ...}`，访问 `car1` 就会得到 `Car{color: undefined, name: "Merz", height: "1400", lang: "4900", weight: 1000, ...}`，所以说同一个构造函数生产的不同对象是两个人，彼此独立的，互不影响，再比如我们调用 `car.run();`，

再访问 `car.health` 就是 99，但是我们访问 `car1.health` 就是 100，两个对象虽然才开始属性相同，但是我们在后来通过不断调用自身的方法来改变自己的属性后，两个对象会变得很不一样。我们函数里传了一个形参 `color`，在函数里第一行的时候我们让 `color` 属性等于形参 `color`，在外部调用的时候传参：`var car = new Car('red');``var car1 = new Car('green');`；此时我们访问 `car.color` 就是 `red`，访问 `car1.color` 就是 `green`。

例 4:

```
function Student(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
    this.grade = 2018;
}
```

```
var student = new Student('zhangsan', 18, 'male');
```

解析：此时我们访问 `student` 就会得到 `Student{name: "zhangsan", age: 18, sex: "male", grade: 2018}`，有三个是我们自己选配的，有一个是固定好了的，有选配有固定好了的就可以产生对象，并且对象和对象之间是独立的，调用多次构造函数可以产生多个独一无二的对象。

(4) 构造函数的内部原理

为什么构造函数里加上一个 `this` 他就能生产出来对象？`this` 为啥放到构造函数内部依然好使？其实他内部有一个三段式（都是隐式的），但是在执行的时候前边必须加上 `new` 才可以。下面介绍隐式的三段式：

1) 在函数的最前边隐式的加上 `this = {}`；（空对象）

2) 执行 `this.xxx = xxx`;

3) 在最后隐式的返回 `this` （`return this`）

我们来写个例子，隐式的注释掉哈：

```
function Person(name, height) {
    // var this = {};
    this.name = name;
    this.height = height;
    this.say = function () {
        console.log(this.say);
    }
    // return this;
```

```
}  
console.log(new Person('xiaowang', 180).name);
```

我们在外部通常 var 一个变量来接收他，但是我们现在直接把他打印出来也可以，输出得 xiaowang. 那我们现在已经知道原理了，我们现在来模拟一下构造函数：

```
function Person(name,height) {  
    var that = {};  
    that.name = name;  
    that.height = height;  
    return that;  
}
```

```
var person = Person('xiaowang',180);  
var person1 = Person('xiao Zhang',175);
```

既然我们模拟出来了，都是显式的定义和返回，就不用 new 了，但是我们一般不这么用，因为这里边还有很多东西是我们模拟不了的。

现在我们回归到上边的隐式的：

```
function Person(name,height) {  
    // var this = {};  
    this.name = name;  
    this.height = height;  
    this.say = function () {  
        console.log(this.say);  
    }  
    return {};  
    // return this;  
}
```

```
var person = new Person('xiaowang',180);  
var person1 = new Person('xiao Zhang',175);
```

假如我们显示的返回一个空对象，这就成了捣乱了，再访问 person 就得到空对象，因为它是显式的。但是：

```
function Person(name,height) {  
    // var this = {};  
    this.name = name;  
    this.height = height;
```



```

    this.say = function () {
        console.log(this.say);
    }
    return 123;
    // return this;
}

```

```

var person = new Person('xiaowang', 180);
var person1 = new Person('xiaozhang', 175);

```

假如我们显式返回 123 的话，在访问 person 的话就会得到 `Person{name: "xiaowang", height: 180, say: f}`，是因为系统规定了有 new 的话就不能返回原始值，必须返回引用值，返回原始值的话系统会自动忽略。

2. 包装类

我们知道，`var num = 123`，这个 num 属于原始值的 123，原始值是不能有属性和方法的，属性和方法是对象（这里的对象指的是对象、数组和 function）独有的东西，原始值只是一个值，他作为一个独立的个体存在。

然而数字不全是原始值，只有原始值的数字才是原始值，在我们 js 里边，数字分两种数字，字符串分两种字符串，比如：

```
var num = new Number(123);
```

前边都加上 new 了，他就是构造函数所产生的对象，他就属于对象类型的 123，你输出 num 就会得到 `Number{123}`，既然是对象了，就可以加属性了，比如：`num.abc = 'a'`；，你在访问 num 就得到 `Number{123, abc: "a"}`，他成了对象了，但是依然可以参与运算，比如 `num * 2` 就得 246，但是参加完运算之后这个 246 又变成原始值的数字了。但是你不让他参与运算，给他加个属性或者方法，他又能当对象那么来用。字符串和布尔类型完全一致。比如：

```

var str = new String('abcd');
str.a = 'bcd';
str.sayValue = function () {
    return this.a;
}

```

我们 `console.log(str.sayValue());` 就会得到 bcd。你想让对象类型的数字（或者字符串）有什么值就直接在后边的括号里边写即可。还有布尔类型：

```
var bol = new Boolean('true');
```

我们访问 bol 就得到 `Boolean{true}`，但是 undefined 和 null 他俩不可以有属性，你

访问 `undefined.abc = 123` 或者 `null.abc = 123` 就会报错. 知道了这些, 我们就来讲包装类, 比如: `var str = "abcd"` 这是一个原始值, 但是我们访问 `str.length` 就会得到 4, 这是为啥? 人家明确规定原始值是没有属性和方法的, 那么这个 `length` 是哪来的? 我们在下边继续写上 `str.abc = "a"`, 这理论上是不可以的, 因为原始值不能有属性, 但是你会发现不报错, 而且你访问 `str.abc` 得到 `undefined`, 这又是为啥? 再比如:

```
var num = 4;
num.len = 3;
console.log(num.len);
```

要记住原始值是不能有属性和方法的, 不能有属性还硬能加属性, 到底因为啥? 其实这里边经历了一个隐式的过程叫做包装类, 在第二行的时候, 由于 `num` 不能加属性, 他就会隐式的 `new.Number(4).len = 3`; 他会新建一个数字对象, 让数字对象的 `len` 等于 3, 来弥补你操作的不足, 完事后他会立马 `delete` 删除, 因为系统认为新建的 `Number` 没有用, 没保存到任何地方就直接删掉了, 然后你在下边访问的时候系统继续隐式的 `new.Number(4).len`, 但是他和上边的 `Number` 是两个对象, 刚才那个 `len` 有值但是被销毁了, 你现在访问的 `len` 他没有值啊, 所以就是 `undefined`。

考点: 有时候他基于截断数组, 就是比如说你 `var arr = [1, 2, 3]; arr.length = 2`; 这时候访问 `arr` 就是 `[1, 2]`, 这就是截断数组, 他基于截断数组会这么考:

```
var str = "abcd";
str.length = 2;
console.log(str);
```

这时应该输出 `abcd`, 因为在第二步的时候他会隐式的 `new String('abcd').length = 2`, 然后 `delete` 扔了, 你在第三行访问 `str` 的时候访问的是第一行的 `abcd`, 第二行新建的和第一行没有关系啊, 而且新建完就被销毁了。所以第三行访问的是第一行的 `str`。如果在下面写上 `console.log(str.length)`; 就会得到 4, 因为 `length` 是系统自带的属性, 但是是对象的 `length` 属性, 你访问的时候系统就隐式的 `new String('abcd').length`, 这个对象原本就有 `length` 属性是 4, 所以我们会产生一种错觉, 其实你访问 `str.length` 就是对象的 `length` 属性给你抛回来的值。所以表面上看原始值可以操作属性, 实际上是操作不了的。

3. 真题讲解

练习 1: (成哥原创)

```
var str = "abc";
str += 1;
var test = typeof (str);
```

```
if (test.length == 6) {
    test.sign = 'typeof 的返回值可能为 string';
}
```

```
console.log(test.sign);
```

解析：执行第二句后 `str = "abc1"`，然后 `var` 一个 `test` 等于 `typeof (str)`，那么 `test` 现在就等于字符串类型的“string”，然后 `if` 里边的条件，因为他是字符串，所以通过包装类的过程后，`test.length` 就等于 6，所以条件成立，再看里边的，注意，这是一个坑，`test` 是字符串，属于原始值，所以你加上属性，系统就会调用包装类，隐式的 `new String("string").sign = 'typeof 的返回值可能为 string'`，然后立马 `delete` 销毁掉，所以你在外部访问 `test.sign`，系统就继续 `new String("string").sign`，然后没有值，输出 `undefined`。所以里边即使赋值了，也没有什么卵用。

练习 2：（单选）分析下面的 JavaScript 代码段：

```
function employee(name, code) {
    this.name = "wangli";
    this.code = "A001";
}

var newemp = new employee("zhangming", "A002");
document.write("雇员姓名: " + newemp.name + "<br/>");
document.write("雇员代号: " + newemp.code + "<br/>");
```

输出的结果是（ ）。

- A: 雇员姓名: wangli 雇员代号: A001
- B: 雇员姓名: zhangming 雇员代号: A002
- C: 雇员姓名: null 雇员代号: null
- D: 代码有错误，无输出结果

解析：这个明显选 A 啊，虽然说参数传进去了，但是人家函数里根本没有用到参数啊，在函数里已经写死了。

练习 3：

```
function Person(name, age, sex) {
    var a = 0;
    this.name = name;
    this.age = age;
    this.sex = sex;
    function sss() {
```

```

        a++;
        document.write(a);
    }
    this.say = sss;
}
var oPerson = new Person();
oPerson.say();
oPerson.say();
var oPerson1 = new Person();
oPerson1.say();

```

解析：这道题的综合力很强，在外部用 new 了，他就等于隐式的把 this 这个对象保存到了外部的 oPerson 身上，而 this 里的 say 属性等于 sss，sss 是一个函数，等于说 sss 也被保存到了外部，他就必然形成闭包，然后第一次调用 oPerson.say 的时候他就可以拿到 Person 的劳动成果访问 a 并++，所以打印 1，第二次再访问再++，打印出 2，然后 oPerson1 等于 new Person 执行，他又返回了一个全新的对象，和里边的 sss 形成了一个全新的闭包，拿到了一个新的执行期上下文，所以 oPerson1.say 执行拿到了一个全新的 a++ 后得 1，最后结果 1 2 1。

练习 4：（2013 年百度笔试题）执行完下边的代码后，访问 x、y、z 分别是什么？

```

var x = 1, y = z = 0;
function add(n) {
    return n = n + 1;
}
y = add(x);
function add(n) {
    return n = n + 3;
}
z = add(x);

```

解析：访问 x、y、z 得 1、4、4，并不是 1、2、4，因为他要走预编译环节，两个函数名字相同，都会被提升，但是后边的会覆盖掉前边的。

练习 5：请问以下表达式的结果是什么？

```

parseInt(3, 8);
parseInt(3, 2);
parseInt(3, 0);

```

解析：parseInt 的作用就是以目标进制为基底，再转换为 10 进制的数，所以第一个以

八进制为基底把 3 转换为十进制的数为 3，第二个二进制没有 3 啊，所以输出 NaN，第三个没有 0 进制啊，所以自动忽略输出 3（有的浏览器输出 MaN）。

练习 6：以下哪些是 JavaScript 语言可能返回的结果？

A: string B: array C: object D: null

解析：这个题选 AC，array（数组）属于 object 里边的，null 也属于 object 里边的。

练习 7：看看下面 alert 的结果是什么？

```
function b(x,y,a) {
    arguments[2] = 10;
    alert(a);
}
```

b(1,2,3);

如果函数体改成下面，结果又会是什么？

```
a = 10;
alert(arguments[2]);
```

解析：都是 10，这道题考的就是 arguments[2] 和 a 是互相映射的，一个改另一个跟着改。

练习 8：写一个方法，求一个字符串的字节长度。（提示：字符串有一个方法 charCodeAt()；一个英文占一个字节，一个中文占两个字节）

定义和用法：

charCodeAt() 方法可以返回指定位置的字符的 Unicode 编码，这个返回值是 0-65535 之间的整数。（当返回值 ≤ 255 时，为英文，当返回值 > 255 时为中文）

语法：

StringObject.charCodeAt(index)

eg:

```
<script type="javascript/text">
    var str = "Hello word!"
    document.write(str.charCodeAt(1)); // 输出 101
</script>
```

方法一：

```
function bytesLength(str) {
    var count = 0;
    for (var i = 0; i < str.length; i++) {
        if (str.charCodeAt(i) > 255) {
            count += 2;
        }
    }
}
```

```

        } else {
            count++;
        }
    }
    return count;
}

```

方法二:

```

function bytesLength(str) {
    var count = str.length;
    for (var i = 0; i < str.length; i++) {
        if (str.charCodeAt(i) > 255) {
            count++;
        }
    }
    return count;
}

```

解析：其实题里边都解释的很清楚了，其实 asc 编码就是 Unicode 的一部分，你调用 `str.charCodeAt(i)` 时，他会调用包装类，和 `string.length` 是一样的。我们先看第一种方法，第一种方法就是常规方法，用 `for` 循环把字符串的每一位先扫一遍，然后当字符为中文的时候（Unicode 编码大于 255），让 `count+=2`，当字符为英文的时候让 `count++`，第二种方法就是由于中文比英文多一个字节，那么就让 `count` 先等于字符串的长度，然后当字符串为中文的时候，再让 `count++` 即可。

原型、原型链、toString 和 toFixed 知识点补充、call、apply

1. 原型

(1) 定义：原型是 `function` 对象的一个属性，它定义了构造函数制造出的对象的公共祖先，通过该构造函数产生的对象，可以继承该原型的属性和方法，原型也是对象。

例如：

```

Person.prototype.name = "hehe";
function Person() {

```

```

}
var person = new Person();

```

例如上边的，这个 `person` 就是构造函数 `Person` 生产出来的对象，由于 `Person` 是函数，

函数也属于一种对象，对象就有属性和方法，那么我现在就在上边给 Person 加上一个 **prototype** 属性，这个 prototype 就是原型，它是系统自带的属性，在这个 Person 函数刚出生的时候，这个 Person.prototype 就被定义好了，**Person.prototype** 才开始可以把它理解成一个空对象，他就是这个 Person 构造函数构造出对象的祖先，好，现在我们给祖先加一个属性，即 **Person.prototype.name = "hehe"**；那么 person 是 Person.prototype 祖先的晚辈，他就能继承祖先的属性，假如你单纯访问 person 就是一个空对象，但是你访问 **person.name** 就得到 "hehe"，他拿的就是祖先的属性，这就叫做继承。什么是公共祖先，假如说我们在下边继续写上 **var person1 = new Person()**；，这个 person 和 person1 都是 Person 所生产出来的对象，是兄弟俩，所以他们的祖先都是同一个人 **Person.prototype**，他俩都能拿到 **Person.prototype** 的属性，你访问 **person1.name** 也能得到 "hehe"，所以原型是他构造出对象的共有祖先，他描述的就是继承的关系。

再比如

```
Person.prototype.LastName = "deng";
Person.prototype.say = function () {
    console.log('hehe');
}
function Person() {
    this.LastName = "ji";
}
```

```
var person = new Person();
var person1 = new Person();
```

此时我们调用 person（或 person1）.say 都可以调用这个函数，输出 hehe，这属于借用，假如说我们在函数里边写上 **this.LastName = "ji"**；，那么访问 **person.LastName** 就得到 ji，取近的，自己身上有就拿自己的，没有的话就去祖先那里找。

（2）作用：可以提取公有属性

例 1：

```
function Car(color, owner) {
    this.owner = owner;
    this.color = color;
    this.carName = "BMW"
    this.height = 1400;
    this.lang = 4900;
}
```

```
var car = new Car('red', 'prof. ji');
```

这么写理论是没有问题，里边选配的可以这么写，但是流程化生产的每一次执行都要执行这三条语句，这就是代码的耦合，我们学完原型就可以用继承的方式把他们提取出来：

```
Car.prototype.carName = "BMW";
```

```
Car.prototype.height = 1400;
```

```
Car.prototype.lang = 4900;
```

```
function Car(color, owner) {
```

```
    this.owner = owner;
```

```
    this.color = color;
```

```
}
```

```
var car = new Car('red', 'prof. ji');
```

现在我们来看这个代码，上边的原型就只执行一次，我们表面上访问 car 得到 Car {owner: "prof. ji", color: "red"}，但是我们访问 car.carName 也可以得到 "BMW"，就是我们可以把一些公有的部分提取到原型里边。其实还可以简单的写，即然原型是空对象，那么就可以这么来写：

```
Car.prototype = {
```

```
    carName: "BMW",
```

```
    height: 1400,
```

```
    lang: 4900
```

```
}
```

```
function Car(color, owner) {
```

```
    this.owner = owner;
```

```
    this.color = color;
```

```
}
```

```
var car = new Car('red', 'prof. ji');
```

(3) 原型的增删改查 例如：

```
Person.prototype.lastName = "Deng";
```

```
function Person(name) {
```

```
    this.name = name;
```

```
}
```

```
var person = new Person('xuming');
```

查：直接访问 person.lastName 即可。

改：假如说我们直接 person.lastName = "James"，这样是不可以的，他会给 Person

自己身上加一个 `lastName` 等于 James，就是通过对对象想操作原型的東西是不可能的，你访问的话自己没有会去原型那里找的，真要改的话就必须 `Person.prototype.lastName = "James"`;

增删：一样的，如果真要操作原型属性，就必须用 `Person.prototype`。

(4) constructor

```
function Car() {
}
var car = new Car();
```

例如上边的，我们调用 `car.constructor` 就得到 `function Car() {}`，`constructor` 的作用就是指向该对象的构造函数，其实他是 `Car.prototype` 里的一个隐式属性：

```
Car.prototype = {
  constructor: Car(); (隐式的)
}
```

但是隐式的属性也可以更改：

```
Car.prototype = {
  constructor: Person();
}
function Person() {
}
function Car() {
}
```

```
var car = new Car();
```

这就成了捣乱了，现在我们 `car.constructor` 就得到 `function Person() {}`。

(5) __proto__

这个 `__proto__` 到底是干嘛的？例如：

```
Person.prototype.name = "abc";
function Person() {
```

```
}
```

```
var person = new Person;
```

此时我们访问 `person.__proto__` 就得到 `{name: "abc", constructor: f}`，而我们只给原型加 `name` 了，所以 `__proto__` 里边放的就是原型，那么这个 `__proto__` 到底是哪里来的？其实是这样的：在我们用 `new` 来构造这个函数的时候，他会隐式的发生三段式，第一步就是创建一个 `this` 空对象，其实并不是空对象，他一上来就有一个属性叫

```
__proto__:  
this{  
  __proto__:Person.prototype;  
}
```

这个__proto__里放的就是 Person.prototype，他的作用就是当你要访问这个对象里的属性的时候，如果对象没有这个属性，他就会通过__proto__指向的这个索引，去找 Person.prototype 身上有没有这个属性，这个__proto__就起到了一个连接的作用，把原型和自己连接在一起。

这个__proto__前后都有_，这是一种命名规则，是告诉人们这个属性不要随便动，尽量不要随意更改，但是还是可以更改的：

```
Person.prototype.name = "abc";  
function Person() {  
  
}  
var obj = {  
  name: "sunny"  
}  
var person = new Person;
```

最开始我们访问 person.__proto__得到{name: "abc", constructor: f}，但是我们现在 person.__proto__ = obj，再访问 person.name 就得到 "sunny"，因为他的原型被修改了，就是通过 Person 构造函数构造出来的对象的原型未必非得是 Person.prototype，它是可以被经过修改的。因为 person 能找到他的原型的根本原因就是__proto__指向，现在我们更改了他的指向，person.__proto__ = obj，你再访问 person.name 他就会顺着__proto__的指向去找，找到 obj，输出 sunny，那么，person 自此之后就 and Person.prototype 没有任何关系了。

例 2：下列代码执行完之后访问 person.name 是啥？

```
Person.prototype.name = "sunny";  
function Person() {  
  
}
```

```
var person = new Person;  
Person.prototype.name = "cherry";
```

解析：肯定是 cherry 啊，person 自己身上没有 name 属性，他就会通过__proto__找到 Person.prototype，你在下边吧他的 name 值改成 cherry 了，再访问就是 cherry。

例 3：下列代码执行完之后访问 person.name 是啥？

```
Person.prototype.name = "sunny";
function Person() {
```

```
}
var person = new Person;
Person.prototype = {
  name:"cherry"
}
```

解析：此时应该是 sunny，因为我们之前讲过，比如说：

```
var obj = {name:'a'};
var obj1 = obj;
obj = {name:'b'};
```

一开始的时候 obj1 和 obj 都指向同一个房间，后来 obj 指向了一个新的房间，但是 obj1 还是指向原来那个房间啊，所以访问 obj1.name 就是 a，这道题也是一样的：

```
Person.prototype = {name:"sunny"};
```

```
__proto__ = Person.prototype;
```

```
Person.prototype = {name:"cherry"};
```

最开始的时候 __proto__ 和 Person.prototype 指向的是同一个房间，后来 Person.prototype 指向了一个新的房间，但是访问 person.name 他找的是 __proto__，__proto__ 还是指向原来那个房间啊，所以得 sunny，例 2 改的是对象里的属性，而例 3 直接把对象都改了，所以这两个不一样。

例 4：下列代码执行完之后访问 person.name 是啥？

```
Person.prototype.name = "sunny";
function Person() {
```

```
}
Person.prototype = {
  name:"cherry"
}
```

```
var person = new Person;
```

解析：应该是 cherry，我们来看一下执行顺序：先预编译，把 function 提上去，然后执行的时候后边的 Person.prototype 会覆盖掉前边的，然后你在 new，这个时候 this 对象才产生，__proto__ 指向后边的 Person.prototype，输出得 cherry，他和例 3 的

区别就是：例 3 是 new 完之后才改的 Person.prototype，也就是说在 this 对象产生后，__proto__ 已经指向原来的 Person.prototype 了，然后 Person.prototype 才改了对对象，但是例 4 是 Person.prototype 已经改好了才产生的 this 对象，这时候__proto__才指向 Person.prototype。

2. 原型链

(1) 比如说：

```
function Person() {
```

```
}
```

```
var person = new Person();
```

此时我们访问 Person.prototype 发现里边还有一个__proto__属性，那就说明，原型还有原型，那我们就来写一写原型连成链的东西：

```
Grand.prototype.lastName = "Deng";
```

```
function Grand() {
```

```
}
```

```
var grand = new Grand();
```

```
Father.prototype = grand;
```

```
function Father() {
```

```
    this.name = "xuming";
```

```
}
```

```
var father = new Father();
```

```
Son.prototype = father;
```

```
function Son() {
```

```
    this.hobbit = 'smoke';
```

```
}
```

```
var son = new Son();
```

最开始的时候 Grand 有一个属性 prototype，Grand 是一个构造函数，他构造出了对象 grand，然后 grand 是这个 Father 构造函数的原型，Father 又构造出对象 father，这个 father 又是 Son 构造函数的原型，这个 Son 又构造出对象 son，现在我们给 Father 里加上一个 this.name = "xuming";给 Son 里边加上 this.hobbit = 'smoke';，现在我们访问 son.hobbit，自己身上就有，所以得“smoke”，我们接着访问 son.name，自己身上没有就找__proto__，然后指向原型 father，father 里有，所以输出“xuming”，现在我们访问一个高级的，访问 son.lastName，自己身上没有，找到__proto__，然后

指向原型 father, father 自己也没有, 就继续找, 找到自己的__proto__, 指向原型 grand, grand 自己身上依然没有, 就找自己的__proto__, 指向原型 Grand.prototype, 这里找到了 lastName, 输出得 Deng。这个把原型上边再加一个原型再加一个原型……形成了一个链, 访问顺序是依照链的顺序, 我们就把他叫做原型链, 原型链的连接点就是__proto__, 他的访问顺序和作用域链差不多, 可近的来, 近的有就访问近的, 没有的话再往远处找, 找到终端依然没有就输出 undefined。其实我们表面上看 Grand 就是终端了, 然而并不是, 因为你会发现当你访问 Grand.prototype 的时候里边还有__proto__, 这个__proto__指向 Object, 其实 Object.prototype 才是所有对象的最终原型, 是原型链的终端, 你在访问 Object.prototype.__proto__就得到 null。比如你访问 son.toString 就得到 `f toString() { [native code] }`, 他拿的就是 Object.prototype 里的东西。

(2) 原型链上的增删改查 (接着上个例子讲)

原型链上的增删改查和原型上的增删改查差不多是一致的:

查看: 可近的来, 近处没有就往远处找, 找到终端依然没有就输出 undefined。

删除: 通过他的子孙是没办法删除的, 但是通过自己可以删, 例如: `delete Father.prototype.xxx`

修改: 除非本人修改, 后代是无法修改的。

增加: 除非自己增加, 后代无法增加。

但是有个特例: 现在我给上边的 Father 函数里加点儿东西:

```
function Father() {
  this.name = "xuming";
  this.fortune = {
    card1: "visa"
  }
}
```

现在我们访问 `son.fortune` 就得到 `{card1: "visa"}`, 我们修改 `son.fortune = 200`; 这就等于给 son 加了一个 fortune 属性, 你访问 son 得到 `Son{hobbit: "smoke", fortune: 200}`, 但是你访问 `father.fortune` 还是 `{card1: "visa"}`, 但是现在我不这么改, 我不像上边那样去改, 我直接 `son.fortune.card2 = "master"`, 现在我访问 son 得到 `Son{hobbit: "smoke"}`, 好像 son 自己并没有变, 我们访问 `father.fortune` 得到 `{card1: "visa", card2: "master"}`, 但是这种修改是引用值自己的修改, 因为你 `son.fortune` 就相当于 fortune 被你取出来了, 取出来之后.name 就相当于 fortune.name 给自己加东西, 这不算一种赋值的修改, 这算一种调用的修改, 这种层面上的修改是可以的, 但是直接给他加值覆盖进去是不行的, 这种修改也仅限于引用值, 原始值是不可以的。

再比如：

```
function Father() {  
    this.name = "xuming";  
    this.fortune = {  
        card1:"visa"  
    };  
    this.num = 100;  
}
```

我们访问 son.num 得到 100，现在 son.num++，我们访问 father.num 还是 100，但是我们访问 son 就是 Son{hobbit: "smoke", num: 101}，++就是自身加 1 赋给自身，son.num = son.num + 1，他先会把 son.num 取出来，再+1 赋给自身，赋值后就变成自己的了。

例 5：

```
Person.prototype = {  
    name: "a",  
    sayName: function () {  
        console.log(this.name);  
    }  
}  
  
function Person() {  
  
}
```

```
var person = new Person();
```

现在我们调用 person.sayName(); 就会输出 a，那我们现在给 Person 函数里加上 this.name = "b"，在调用 person.sayName();，虽然 sayName 属性是原型里的，但是他里边打印的是 this.name，有一条规定就是谁调用的函数 this 就指向谁，因为 person 调用的，所以 this 指向 person，输出 b。第一次调用的时候是因为 person 里没有 name 所以输出 a，你如果调用 Person.prototype.sayName() 就会输出 a。

例 6：

```
Person.prototype = {  
    height: 100,  
}  
  
function Person() {  
    this.eat = function () {  
        this.height ++;  
    }  
}
```

```

    }
}
var person = new Person();

```

我们现在调用 `person.eat()`，此时访问 `Person.prototype` 是 `{height: 100}`，访问 `person` 就得到 `Person {eat: f, height: 101}`，这个和上边的是一样的，取出来加 1 赋给自身。

(3) Object.create (原型)

```

var obj = {}
var obj1 = new Object();

```

我们来看，第一种创建对象的方法叫做对象字面量，他这么创建出来也是有原型的，原型就是 `Object.prototype`，在你创建的时候系统就会隐式的加上 `new Object`，那他和第二种创建的就是一样的了，因为这个 `obj1` 的原型也是 `Object.prototype`，所以第二种也没有啥用，而且加东西不好加，所以第二种很少见，在以后开发的时候不能用第二种方法，必须用对象字面量的创建方式。

```

Person.prototype = {};
function Person () {
}

```

这个 `Person.prototype` 默认是一个空对象，也是对象字面量的方法创建的，那么他的原型毋庸置疑就是 `Object.prototype`。

其实还有一种对象的创建方法：

```

var obj = {name:"a"};
var obj1 = Object.create(obj);

```

这样通过 `Object.create` 也能创建出对象，括号里放的就是原型，你现在访问 `obj1.name` 就是 `a`。

考点：阿里巴巴以前有一道判断题说所有的对象最终都继承自 `Object.prototype`，这么说是不对的，我们前边确实讲的原型的终端都是 `Object.prototype`，但是有例外，学了 `Object.create()` 就不一样了，这个括号里还必须放东西，不放东西会报错。但是可以放两种值：`object`（对象）和 `null`。好，那既然可以放 `null`，我们就试试：

```

var obj = Object.create(null);

```

现在我们访问 `obj` 得到 `{}`，这是一个真正的空对象，里边啥都没有，所以他没有原型，但他可以当对象来用，比如说加个属性等。我们人为的 `obj.__proto__ = {name: "sunny"}`，你再访问 `obj.name` 得到 `undefined`，所以这个 `__proto__` 是系统自带的，人们可以修改，但是你自己加的系统是不会认得。

结论：绝大多数对象最终都继承自 `Object.prototype`（并不是所有）。

3. 知识点补充

(1) toString 知识点补充

我们以前讲过，说 `undefined` 和 `null` 不能调用 `toString`，会报错的，现在我们就知道原因了：数字能调用 `toString` 是因为数字可以调用包装类，包装成对象然后一层一层往上访问一直到终端 `Object.prototype`，所以数字是肯定有 `toString` 的，但是 `undefined` 和 `null` 是没有包装类的，他就是一个原始值，他也没有原型，所以他俩不可能有 `toString`。现在我们就来研究一下 `toString`

我们访问 `true.toString()` 就得到 “true”，我们访问 `123.toString()` 就会报错，因为他会认为这个数字是浮点型的，是小数点，小数点后必须跟数字啊，跟字母肯定不行，所以会报错，我们 `var num = 123;` 然后访问 `num.toString()` 得到 “123”，但是我们 `var obj = {};` 然后访问 `obj.toString()` 得到 “[object Object]”，这是为啥？不是应该返回字符串的大括号吗？

其实上边的 `obj.toString()` 他毋庸置疑拿的就是 `Object.prototype` 里的 `toString`，但是你 `var num = 123;` 然后 `num.toString()`，他就会隐式的经过包装类 `new Number(num).toString()`，这里的原型就是 `Number.prototype`，`Number.prototype` 的原型才是 `Object.prototype`，但是在 `Number.prototype` 里就有这个 `toString` 方法，即 `Number.prototype.toString = function() {…}`，这就形成了一个原型链，那么假如说 `Number` 上有这个 `toString` 方法的话我就不用你 `Object` 上的了，然而 `Number` 上正好有这个方法，所以 `Number` 调用的 `toString` 是自己重写过的，原型上有这个方法，我又写了一个同一个名字不同功能的方法叫做重写。比如说：

```
Person.prototype = {
  toString:function () {
    return "hehe";
  }
}
function Person() {
}
var person = new Person();
```

我们假如 `Person.prototype` 里边不写东西的话，调用 `person.toString()` 他就会拿到 `Object.prototype` 里的 `toString`，输出 “[object Object]”，但是我们在近处写一个 `toString`，这时候你再访问 `person.toString` 就得到 “hehe”，因为我们在近处截断了。这种和原型链上名字一样的方法但实现不同功能的就叫方法的重写，重写就是覆盖。

其实系统内部把 Number、Array、Boolean、String 这些里的原型 prototype 都把 toString 重写了，所以 `var num = 123`，访问 `num.toString()` 就是 '123'。因为 `Object.prototype` 里的 `toString` 没有啥用，所以他的后代子孙都重写了这个方法。注意：我们之前讲的 `document.write` 不能算是真正意义上的打印，他会调用 `toString` 方法，然后给你返回，虽然你 `var num = 123; document.write (num)` 可以打印 123，但是比如你 `var obj = {}`，在 `document.write (obj)` 就是 **【object Object】**，`var obj=Object.create (null)`，在 `document.write (obj)` 就会报错，因为你没有原型就没有 `toString` 方法，所以 `document.write` 不是真正的输出方法，我们一般用 `console.log`。

(2) toFixed 相关知识点补充

这里讲一个小 bug 哈，比如说我们在控制台访问 `0.14*100` 就得到 `14.000000000000002`，这是一个无法解决的 bug，是因为 js 精度不准，数据会发生一小点的偏差，所以要尽量避免小数操作，即使有小数操作，也要借用两个工具：

向上取整：例如 `Math.ceil (123.234)` 得到 124，向上取整 0.1 他也会认为是 1

向下取整：例如 `Math.floor (123.999)` 得到 123，不管后边有多少位小数，都把他们的割掉。

产生随机数：`Math.random()`，调用一次产生一个随机数，0-1 中间的随机小数，两边不会到头。比如说：

```
for(var i = 0;i < 10;i++) {
    var num = Math.random().toFixed(2) * 100;
    console.log(num);
}
```

我们想随机打印十个 0 到 100 的整数，先用 `Math.random()` 把 0 到 1 的随机数取出来，再保留两位小数，那么就是 0.01-0.99 的随机数，然后在乘以 100，var 一个 num 等于他，然后输出 num 并让循环转十圈，有时候会输出 54 6 84 64 60 54 37 38 44 56.000000000000001，看最后一个数，这就是 js 精度不准造成的 bug，解决办法：

```
for(var i = 0;i < 10;i++) {
    var num = Math.floor(Math.random() * 100);
    console.log(num);
}
```

我们不保留有效数字位了，直接乘以 100 然后取整即可。

考点：js 可正常计算的范围：小数点前 16 位，后 16 位，有科学计数法的除外，比如 `0.000000000000000001 + 0.000000000000000001` 就得 `2e-18` (2 乘以 10 的 -18 次方)。

但是你 `0.100000000000000001 + 0.000000000000000001` 他就输出 0.1。

4. call 和 apply

call 和 apply 都是更改 this 指向的

(1) call

比如说：

```
function test() {  
}
```

```
test();
```

任何一个方法都可以.call，其实方法的执行就是方法.call 执行，例如上边的，test() 就相当于 this.call()，call 还有一个更高深的用法：

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
}
```

```
var obj = {};
```

```
Person.call(obj);
```

call 后边的括号里是可以传东西的，比如传一个 obj 进去，他就会引导着 Person 发生很大的变化。我们在以前讲到如果在 Person 底下 new 的话，this 就代表了构造函数所产生的对象，我们假如说没有 new 的话，this 默认是指向 window 的，但是我们在 call 里传 obj 了，那么这个 this 就不是默认的了，就变成 obj 了，就是 call 里边传谁进去，this 就是谁。等于说里边现在执行的时候就是 obj.name = name, obj.age = age，即使是这样，依然可以传参：

```
Person.call(obj, 'cheng', 300);
```

现在访问 obj 就得到 {name: "cheng", age: 300}，因为 call 括号里的第一位会改变 this 指向，第一位以后都会当成正常的实参传到形参里边去，所以这里边第二个实参对应的是函数执行的第一位形参，第三个实参对应的是函数执行的第二位形参。所以 call 的根本原因是改变 this 指向，借用 Person 的执行封装了 obj，借用你的方法来实现我的功能。

应用：我们在开发的时候讲究的是快准狠，首先，快，同事之间开发，你写的方法可以实现我的功能，那我就不用写了。

例 7：

```
function Person(name, age, sex) {  
    this.name = name;  
    this.age = age;  
    this.sex = sex;
```



```

}
function Student(name, age, sex, tel, grade) {
    Person.call(this, name, age, sex);
    this.tel = tel;
    this.grade = grade;
}
var student = new Student('sunny', 123, 'male', 139, 2017);

```

解析：这是一个非常好的例子，这个 call 的作用只有一个，就是改变 this 指向，那么就倒出来借用别人的函数来实现自己的功能，你 Person.call() 他就一定会执行 Person，然而里边传了一个 this，我们在外边 new 了，那么这个 this 就代表 student，我们再来看 Student 这个函数，预编译第三步的时候实参和形参就相互统一了，而预编译发生在函数执行的前一刻，等于说执行第一句 Person.call(this, name, age, sex); 的时候 Student 里的形参已经有值了，所以第一句 call 括号里拿的是 Student 传好的形参，是具体的值，然后在把他作为实参传到 Person 里，然后利用 Person 的方法实现了自己的一个封装，其实 call 写在那里就相当于把 Person 里边的三条语句拿到 Student 里了。封装后 student 里就有 name、age 和 sex 了，再执行自己的两行，加上 tel 和 grade。完事后我们访问 student 就得到 Student {name: "sunny", age: 123, sex: "male", tel: 139, grade: 2017}。但是这里你自己的需求必须完全涵盖人家的需求，比如说你不想用 sex 那是不行的，你用 call 了，那么这三条语句一定会走下来。

例 8:

```

function Wheel(wheelSize, style) {
    this.wheelSize = wheelSize;
    this.style = style;
}
function Sit(c, sitColor) {
    this.c = c;
    this.sitColor = sitColor;
}
function Model(height, width, len) {
    this.height = height;
    this.width = width;
    this.len = len;
}
function Car(wheelSize, style, c, sitColor, height, width, len) {

```

```

    Wheel.call(this, wheelSize, style);
    Sit.call(this, c, sitColor);
    Model.call(this, height, width, len);
}
var car = new Car(100, "花里胡哨的", "真皮", "red", 1800, 1900, 4900);

```

解析：这个和上边的是一样的，此时我们访问 car 就是 Car{wheelSize: 100, style: "花里胡哨的", c: "真皮", sitColor: "red", height: 1800, ...}，但是构造函数 Car 里边一行都没有写，都是用的别人的零件，在开发的时候也是一样的，有的时候把零件都给你写好了，你直接用就可以了。

(2) apply

call 和 apply 基本上是一样的，只是写法不一样，call 需要把实参一个个传进去，但是 apply 只能传一个值 arguments，即实参列表：

```

function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}
function Student(name, age, sex, tel, grade) {
    Person.apply(this, [name, age, sex]);
    this.tel = tel;
    this.grade = grade;
}
var student = new Student('sunny', 123, 'male', 139, 2017);

```

其实非常简单，arguments 是数组，那么就直接加一个[]就行了，但是记住 this 不能加在里边哈。

继承模式、命名空间、对象枚举

1. 继承发展史

(1) 传统方式——原型链

```

Grand.prototype.lastName = "ji";
function Grand() {

}
var grand = new Grand();

```

```

Father.prototype = grand;
function Father() {
    this.name = "hehe";
}
var father = new Father();
Son.prototype = father;
function Son() {

}
var son = new Son();

```

弊端: 过多的继承了一些没有用的属性, 比如说我现在就想让 son 继承顶端的 lastName, 而因为它是原型链, 他就把 father 上的, father 原型上的, grand 上的, grand 原型上的东西全部继承了, 那么就有个不好的地方就是我想继承的你继承过来了, 不想继承的你也给继承过来了, 这样是很影响效率的, 所以第一种方式很快就被毙掉了。

(2) 借用构造函数

```

function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}
function Student(name, age, sex, tel, grade) {
    Person.call(this, name, age, sex);
    this.tel = tel;
    this.grade = grade;
}
var student = new Student('sunny', 123, 'male', 139, 2017);

```

之前有构造函数实现了部分功能, 那么之后我就不用把所有功能都写在我自己身上了, 直接用你的, 这也可以理解为一种继承。

弊端: 你只能继承他自己, 并不能继承他的原型, 因为原型还是自己的 Student.prototype, 而且每次构造函数都要多走一个函数, 这么写虽然看着省了点代码量, 但是在执行的时候复杂了, 因为他多调用了方法的执行, 很浪费效率。(虽然在继承层面上来说这样是不好的, 但是实际开发的时候当别人的方法涵盖了你的方法的话, 还是提倡这么用)

(3) 共享原型

```
Father.prototype.lastName = "Deng";  
function Father() {  
  
}
```

```
function Son() {  
  
}  
Son.prototype = Father.prototype;  
var son = new Son();  
var father = new Father();
```

以前我们想让 Son 继承 Father 的属性就让 Father 构造函数构造出一个对象，再让 Son.prototype 等于这个对象，就是原型链的方法，它是有弊端的，现在我直接让 Son 的原型等于 Father 的原型，即 Son.prototype = Father.prototype;，Father.prototype 属于引用值，引用值就可以把地址直接扔给 Son.prototype，相当于一个原型给了两个函数，就是说无论是 Son 的构造函数还是 Father 的构造函数他们的原型都是 Father.prototype，这就叫公有原型。那么以后 Son 构造出来的对象就可以继承 Father.prototype 上的属性了，比如说你访问 son.lastName 就是“Deng”，你访问 father.lastName 也是“Deng”，这两个函数有同一个原型，那么这两个函数生产出来的对象都继承自同一个原型。我们慢慢要学会一种技能，叫抽象出一种功能封装成一个函数，我们来封装一下，我们不是想让一个对象继承某一个东西，最终是想让构造函数继承某一个东西，那么这个构造函数生产出来的对象就都继承自那个东西了，所以最好是打根上来：

```
function inherit(Target, Origin) {  
    Target.prototype = Origin.prototype;  
}
```

这样就封装好了，我们来试一下：

```
Father.prototype.lastName = "Deng";  
function Father() {  
}  
function Son() {  
}  
function inherit(Target, Origin) {  
    Target.prototype = Origin.prototype;  
}
```

```
inherit(Son, Father);
```

```
var son = new Son;
```

此时我们访问 son.lastName 就是 'deng'，我现在调换一下位置：

```
Father.prototype.lastName = "Deng";
```

```
function Father() {  
}
```

```
function Son() {  
}
```

```
function inherit(Target, Origin) {  
    Target.prototype = Origin.prototype;  
}
```

```
var son = new Son;
```

```
inherit(Son, Father);
```

这样肯定不行，访问 son.lastName 就是 undefined，因为我们之前讲过你都 new 了，他都指向原来的空间了，你才在下边改肯定不行，所以必须先继承后 new。

弊端：如果说 Son 想给自己的原型身上加点东西方便它自己产生的对象来用的话，比如说 Son.prototype.sex = 'male'：

```
Father.prototype.lastName = "Deng";
```

```
function Father() {  
}
```

```
function Son() {  
}
```

```
function inherit(Target, Origin) {  
    Target.prototype = Origin.prototype;  
}
```

```
inherit(Son, Father);
```

```
Son.prototype.sex = 'male';
```

```
var son = new Son;
```

```
var father = new Father;
```

此时我们访问 son.sex 就是 "male"，但是你发现访问 father.sex 也是 "male"，是因为 Son.prototype = Father.prototype，他们都是引用值，指向了同一个房间，你给房间里边加东西肯定另一个跟着变啊，这就不好了，Son 是继承了 Father 的原型，但是这么写假如说 Son 想给自己的原型加东西的话会影响 Father 的原型的，鉴于此，我们最后研究了一种丰满的方法——圣杯模式。

(4) 圣杯模式

我们现在要解决一个问题，就是假如说给 Son.prototype 加东西的话，Father.prototype 不受影响，而且 Son 还要继承 Father.prototype，其实还是用公有原型，只不过改变了一下：

```
function F() {};  
F.prototype = Father.prototype;  
Son.prototype = new F();
```

我们先定义一个构造函数 F，这个 F 是个中间层，我们让 F.prototype = Father.prototype；那么这个 F.prototype 就是 Father.prototype，然后 new F() 当做 Son 的 prototype，这就形成了一个原型链，好处就是现在 Son.prototype 是 new F()；new F() 是一个干干净净的对象，你现在想给 Son.prototype 加东西就根本影响不了 Father.prototype，而且 Son 还能通过原型链继承 Father.prototype，两全其美，这就是最终的圣杯模式。我们来封装一下吧：

```
function inherit(Target, Origin) {  
    function F() {};  
    F.prototype = Origin.prototype;  
    Target.prototype = new F();  
}
```

我们来试一下：

```
function inherit(Target, Origin) {  
    function F() {};  
    F.prototype = Origin.prototype;  
    Target.prototype = new F();  
}
```

```
Father.prototype.lastName = "Deng";  
function Father() {  
}
```

```
function Son() {  
}
```

```
inherit(Son, Father)  
var son = new Son();  
var father = new Father();
```

现在我们访问 son.lastName 是 “Deng”，访问 father.lastName 也是 “Deng”，现在 Son.prototype.sex = “male”，此时访问 son.sex 就是 “male”，但是访问 father.sex

就是 undefined。但是这么写还有一个小问题就是：我们知道原型里都有一个属性叫 constructor，这个 constructor 的作用就是指向构造函数，但是现在我们访问 son.constructor 得到 `f Father() {}`，是因为 son 的原型指向 `new F(); new F()` 里没有这个属性，就去 `Father.prototype` 里找到了这个属性，这就造成了指向紊乱，我们手动给他归一下位即可：

```
function inherit(Target, Origin) {
    function F() {};
    F.prototype = Origin.prototype;
    Target.prototype = new F();
    Target.prototype.constructor = Target;
    Target.prototype.uber = Origin.prototype;
}
```

这就是一个最完美的圣杯模式，其中最后一句定义了一个 uber，假如说你真正想知道他继承自谁就可以查看 uber，是一个信息的储存。现在我调换一下位置，还好使吗？

```
function inherit(Target, Origin) {
    function F() {};
    Target.prototype = new F();
    F.prototype = Origin.prototype;
    Target.prototype.constructor = Target;
    Target.prototype.uber = Origin.prototype;
}
```

肯定不好使，讲过好多遍了，new 完了人家都指向原来那个房间了，你在改的话肯定不好使，所以记住一定改完之后才能 new，千千万万不能倒过来。

然而这个只是一个通俗的写法，还有一个高大上的写法。（YUI 库写法）：

```
var inherit = function () {
    var f = function () {};
    return function (Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
        Target.prototype.constructor = Target;
        Target.prototype.uber = Origin.prototype;
    }
}();
```

这个是不是很难理解？没事儿，借着这个，我们来讲闭包的第三个作用。

附：闭包的作用三：可以实现封装，属性私有化。比如说：

```
function Deng(name,wife) {  
    var prepareWife = "xiaozhang";  
    this.name = name;  
    this.wife = wife;  
    this.divorce = function () {  
        this.wife = prepareWife;  
    }  
    this.changePrepareWife = function (target) {  
        prepareWife = target;  
    }  
    this.sayPrepareWife = function () {  
        console.log(prepareWife);  
    }  
}  
var deng = new Deng("deng","xiaoliu");
```

解析：咱们继续讲邓哥的幸福生活！现在邓哥要离婚，即在外调用 `deng.divorce()`，现在我们访问 `deng.wife` 就是“xiaozhang”，那 `prepareWife` 是函数里的变量，外部的对象 `deng` 怎么可以访问呢？因为 `deng.divorce` 是在外部执行的，所以会形成闭包，拿到里边的执行期上下文，就是说里边写的三个函数和外部的 `Deng` 会形成三对一的闭包，所以可以拿到 `Deng` 的 `A0` 并随意存取。那么这个变量 `prepareWife` 和外部的对象 `deng` 的关系就变得及其微妙，比如说有一天你问邓哥是不是背着嫂子外边有人啊？即访问 `deng.prepareWife`，老邓说没有，我是清白的！即得到 `undefined`。然而他真实的自己却能操作 `prepareWife`。所以说这个变量他可以操作，却不是他自己的，但是这个闭包会永远跟着他。就感觉这个变量是他私有的一样，只有自己能看到，别人看都看不到，这就叫私有化变量。就像你问邓哥有没有二嫂？老邓永远不会承认的，但是真正有没有只有他自己知道，除非你调用 `deng.sayPrepareWife()` 可以看到，你想通过对象点的方法根本看不到，因为他不是对象上的东西，它是对象和原有空间形成闭包后闭包里的东西，这就是闭包的第三点应用，叫私有化变量。

现在我们继续讲 YUI 库里写的圣杯模式：

最后我们会把 `return` 的东西（就是 `return` 后边的函数体）传给变量 `inherit`，那么，里边的功能我们都知道了，唯一的问题就是 `F` 哪去了？不是真正没有了，是形成闭包了，成了 `return` 后边函数里的私有化变量了，而这么写是非常好的写法，因为本来这

个 F 就是起到过度的作用，没有太大用处，我们就放到闭包里作为私有化变量，看起来更好，语义化也好一点。

2. 命名空间

作用：管理变量，防止污染全局，适用于模块化开发。

解释：一个网站或者是一个项目，他是需要很多很多人一起来完成的，有许多个人写 HTML，许多人写 css，许多人写 js，最后把他们拼接在一起，而且在你入职后，是要经过专门的培训的，其中就包括什么样的方法用什么样的名字，它是规定好了的，所以你后期在写的时候就会非常规范，那么，就存在一种问题，大家都用这个名字，虽然你自己写的 js 是你自己的文件包，但是最终都是要通过 script 标签把他们引入到同一个 HTML 文件里的，那么就会存在一种命名冲突：都用这个变量名或者函数名，重了的话会覆盖的，咋办？鉴于此，我们研究了两种方法：

方法一：命名空间（老旧方法，已经被抛弃了），例如：

```
var org = {  
  department1:{  
    jicheng:{  
      name:"abc",  
      age:123  
    },  
    xuming:{  
  
    },  
  },  
  department2:{  
    zhangsan:{  
  
    },  
    lisi:{  
  
    }  
  }  
}  
  
var jicheng = org.department1.jicheng;  
jicheng.name
```

解析：这个命名空间也是一个对象，对象里也可以套对象，比如说对象里有个部门 1，

部门 2，部门 1 里 jicheng 定义了一个自己的空间，空间里有一些变量啥的，然后再外部 var 一个 jicheng 等于他自己的空间，再用变量的时候直接 jicheng.name 即可。这样就解决了一个命名冲突的问题，因为 a.name 和 b.name 是不一样的两个人，看起来也更加规整化了，但是现在程序越来越复杂了，这种方法就被抛弃了。

方法二：闭包（现在用的方法，也是闭包的第四个作用）

附：闭包的作用四：模块化开发，防止污染全局变量。比如说：

```
var init = (function () {  
    var name = "abc";  
    function callName() {  
        console.log(name);  
    }  
    return function () {  
        callName();  
    }  
})();
```

解析：最后 init 等于 return 后边的函数体，那么就会形成闭包，闭包的作用就是变量私有化，我就可以把我要定义的一系列变量放到这个闭包里，防止污染全局变量，然后里边比如说定义了一系列复杂的方法，最后返回出去的函数里留一个接口，就是在函数里执行定义好的一系列函数来作为一个终端，因为你要实现一个功能就必须调用函数，然后函数再调用函数再使用变量……这里边定义好一系列功能，然后留一个接口返回给 init，然后 init 调用的时候再启用里边的功能，这么做的好处就是不污染全局变量，而且功能也都能实现，比如说你在外部 var name = "bcd"，你执行 init()，他还是输出 abc，这就是模块化开发。

init：这个 init 是初始化的意思，比如说后边你写了一万行 js 代码，你想让人读懂的话很难，因为他不知道入口在哪，所以你定义一个 init，人们最开始的时候就找你的 init，执行后顺着 init 再依次的去下来。

比如说你在上边的例子中继续写个方法：

```
var Deng = (function () {  
    var name = 123;  
    function callName() {  
        console.log(name);  
    }  
    return function () {  
        callName();  
    }  
})();
```

```

    }
  }()
}

```

你执行 `init()` 就输出 `abc`，执行 `Deng()` 就输出 `123`，他们并不互相影响，这样的话我的变量可以随便起名，你的变量也可以随便取名，就是把全局的变量放在局部，就是为了他们互相不污染，那么比如说我们在以后定义一个方法可以复用的话，我们就把他放到闭包里保存起来，下次直接复制过来用就行了。

3. 模仿 jQuery 实现链式调用模式，例如：

```

var deng = {
  smoke: function () {
    console.log("smoking....xuan cool!!!")
    return this;
  },
  drink: function () {
    console.log("drinking....ye cool!!!")
    return this;
  },
  perm: function () {
    console.log("preming....cool!!!")
    return this;
  }
}

```

```

deng.smoke().drink().perm().smoke().drink();

```

解析：这是邓哥的三大爱好，抽烟喝酒烫头!!! 但是你要连续调用的话必须加上 `return this`，如果不加的话 `deng.smoke().drink()` 写到这里就会报错，他会先把 `smoking....xuan cool!!!` 输出然后报错，因为他执行完他就会关注返回值，你没写的话就是 `undefined`，那么就是 `undefined.drink()`，`undefined` 没有 `drink` 属性啊，所以报错，假如你在每个函数后边加上 `return this`，`this` 就代表 `deng`，所以第一个函数执行完结果是 `deng`，然后 `deng.drink()` 继续执行。所以要实现连续调用的话，必须加上 `return this` 然后就可以 `deng.smoke().drink().perm().smoke().drink()`；像这样连续调用连续执行。

4. 属性的表示方法

```

var obj = {
  name: "abc"
}

```

假如说我们想访问里边的 name，就 obj.name，其实还有一种方法 obj[“name”]，但是方括号里必须是字符串形式的，其实你写 obj.name 系统就会自动转换成 obj[“name”]。再比如：

```
var deng = {  
  wife1: {name: "xiaoliu"},  
  wife2: {name: "xiaozhang"},  
  wife3: {name: "xiaomeng"},  
  wife4: {name: "xiaowang"},  
  sayWife: function (num) {  
    return this["wife" + num];  
  }  
}
```

解析：我们想实现一个功能就是当我执行下边的函数的时候传 1 进去返回 wife1，传 2 返回 wife2，正常我们要用到 switch case 语句，但是现在可以不用了，我就希望把 wife 和 num 拼接起来然后返回，只有字符串可以拼接，那么我们直接返回 this["wife" + num] 即可，现在我们访问 deng.sayWife(1) 就得到 {name: "xiaoliu"}，访问 deng.sayWife(2) 就得到 {name: "xiaozhang"}。

5. 对象枚举

其实枚举就是遍历的意思，我们以前讲过遍历数组，遍历就是挨个知道信息的过程。例：

```
var obj = {  
  name: '13',  
  age: 123,  
  sex: 'male',  
  height: 100,  
  weight: 75  
}
```

我想知道这个对象里边的每个值是什么，可能这个值是后端传给你的，你拆不开，因为你不能以编辑器的角度去看他，程序在自己执行的时候看不到，所以是无法办到了，假如你用 for 循环，那么到底该循环几圈？它连 length 都没有，所以说 for 循环也不行。想真正的遍历对象，就必须借助一个新的语法：for in 循环。

(1) for in 循环

```
for(var prop in obj) {  
  console.log(prop + " " + typeof(prop))  
}
```



```
}
```

解析：这个输出结果就是 name string age string sex string height string weight string，这个循环把对象的每一个属性名都提取出来了，而且有多少个属性名他就会循环多少圈，所以说 for in 循环的功能只有一个，就是遍历对象用的，它是通过属性的个数来控制循环圈数的，它在循环每一圈的时候，都会把对象的属性名放到 prop 里边，var prop in obj 这一句话就是每一圈 prop 都代表一个变量，每一圈这个属性名都不一样，这个 obj 就是我们想要遍历的那个对象，prop 可以随便换名字但是 in 是固定的，你想遍历谁你就 in 谁。那么有了属性名我们就可以知道每一圈每个属性对应的属性值了：

```
for(var prop in obj){
    console.log(obj[prop]);
}
```

此时就得到 13 123 male 100 75

注意：这个循环里不能写成 console.log(obj.prop); 他会输出 5 个 undefined 的，因为你 obj.prop 系统就自动的 obj["prop"]，那么他就会把这个 prop 当成一个属性的，输出 undefined，你写成 obj[prop]（里边不能加引号）他就认为这个 prop 是个变量，而且我们在上边也验证了这个 prop 里边的值刚好是字符串，所以能说通，以后一定要记住，访问的话可以点啥啥啥，写在 for in 里用来枚举的千万要写成方括号的形式，而且里边不能加引号！

(2) hasOwnProperty

我现在把上边的对象改成：

```
var obj = {
    name:'13',
    age:123,
    sex:'male',
    height:100,
    weight:75,
    __proto__: {
        lastName:'deng'
    }
}
```

我们现在再来用上边的 for in 循环遍历一下里边的值就得到 13 123 male 100 75 deng，这就存在一个问题，deng 不是我自己身上的属性，他是我的原型的属性，你怎么也给遍历了？这就造成了一种误差，现在我不想把原型上的东西拿出来，就可

以这么做，一个对象天生就有一种方法叫 `hasOwnProperty`，比如上边的，我们 `obj.hasOwnProperty(prop)`，里边需要传参数，把属性名的字符串形式传进去，这个方法就是判断 `prop` 是不是你自己的属性的，会返回给你一个布尔值，是自己的属性的话就返回 `true`，否则返回 `false`，那么，我们就可以把它塞到 `if` 语句里，如果这个 `prop` 是自己的属性的话就输出，如果不是的话就是原型上的，不输出：

```
for(var prop in obj){
    if(obj.hasOwnProperty(prop)){
        console.log(obj[prop]);
    }
}
```

此时就得到 13 123 male 100 75，那个原型上的 `deng` 就没有了。这个 `obj.hasOwnProperty(prop)` 起到一个过滤的作用，因为你遍历的时候并不希望拿到原型上的东西，所以说遍历对象的时候他们是成套来用的。

注意：单说这个 `for in` 循环，他会把你原型和原型链上的东西都打印出来，但是必须是手动设置的，系统自带的不会打印出来的。

(3) in

`in` 是一个操作符，他的作用也是判断这个属性属不属于这个对象的，其实这么说不标准，比如（2）中的例子，“`height`” `in obj` 得到 `true`（属性是字符串形式的，必须加上引号，不加会报错的），但是你会发现“`lastName`” `in obj` 也是 `true`，他认为原型链上的东西也属于你的，所以确切的说 `in` 只能判断这个属性到底可不可以被这个对象访问到，并不能判断这个属性属不属于这个对象的，判断属性属不属于对象只能用 `hasOwnProperty`。（`in` 基本不用，但是考试有时候会考到）

(4) instanceof

他也是一种操作符，现在定义一个通式：

A instanceof B

官方给出的解释是：判断 A 对象是不是 B 构造函数构造出来的

其实官方解释是不够的，可以这么理解：看 A 对象的原型链上有没有 B 的原型。

```
function Person() {
```

```
}
```

```
var person = new Person();
```

现在 `person instanceof Person` 得到 `true`，`person instanceof Object` 也是 `true`，`[] instanceof Array` 得到 `true`，`var obj = {}`；`obj instanceof Person` 就是 `false`。所以说他的真正作用是看 A 对象的原型链上有没有 B 的原型，可能官方认为原型链上

的也算他的构造器吧。

思考：现在 var arr，这个 arr 可能是数组，也可能是对象，不知道是啥，那么现在需要你想办法来判断他是啥。

方法一：arr.constructor，如果返回 function Array() {} 就是数组，返回 function Object() {} 就是对象。

方法二：arr instanceof Array，如果返回 true 就是数组，返回 false 就是对象。

方法三：调用 Object.prototype.toString.call(arr)，数组返回 "[object Array]"，对象返回 "[object Object]"

方法三解析，首先 Object.prototype.toString = function () {}，他是一个方法，正常这个方法一定是被别人调用的，那么函数里就涉及到 this 的一个知识点，谁调用的方法 this 就是谁，因为 toString 是处理调用者的，把特定的字符串返回，然而他在定义的时候并不知道调用者是谁，他在括号里只能用 this，所以函数里的功能就是识别 this 并返回相应的结果，我们在外部调用 Object.prototype.toString.call(arr)，把 arr 传进去，那么 arr 就是里边的 this，他就会识别 arr 返回相应的结果。

this、callee 和 caller、笔试真题讲解

1. 真题讲解

练习 1：(阿里巴巴试题) 下列选项中，console.log 的结果是 [1, 2, 3, 4, 5] 的选项是 ()

A:

```
function foo(x) {
  console.log(arguments);
  return x;
}
```

foo(1, 2, 3, 4, 5);

B:

```
function foo(x) {
  console.log(arguments);
  return x;
}(1, 2, 3, 4, 5);
```

C:

```
(function foo(x) {
  console.log(arguments);
  return x;
})(1, 2, 3, 4, 5);
```

```

D:
function foo() {
    bar.apply(null, arguments);
}
function bar() {
    console.log(arguments);
}
foo(1, 2, 3, 4, 5);

```

解析：这个题选 ACD，先看 A，人家问的是 console.log，那就和 return 没有关系了，在函数里输出 arguments，即实参列表，还是数组形式的，所以 A 可以，B 我们之前讲过，不报错不执行，他会把小括号移到下边，C 是立即执行函数，只不过把执行写在了括号外边，可以执行。D 实际上做了一个参数的传导，虽然 foo 里边没有写形参，但是实参也是可以传到 arguments 这个实参列表里边去的，在 foo 里边调用 bar，把 arguments 里的 12345 又当做参数传到 bar 里边，所以 bar 的 arguments 里也是 12345，可以输出，其实在 foo 里边写的语句就是 bar(arguments)，因为 apply 是更改 this 指向的，没有 this，所以第一位是 null，和没写一样。

练习 2：（用友试题）以下哪些表达式的结果为 true（ ）

A: undefined==null B: undefined===null
C: isNaN("100") D: parseInt("1a")==1

解析：选 A、D，知识点以前都讲过。这里讲个题外的，{} == {} 得 false，记住，引用值比的是地址，它是两个房间，当然不相等，但是比如 var obj = {}，var obj1 = obj，那么 obj1 == obj 会得到 true，obj1 === obj 也是 true，因为他都指向同一个房间。比的是地址。

2. this

（1）函数预编译过程，this 指向 window，比如说

```

function test(c) {
    var a = 123;
    function b() {};
}
test(1);

```

在函数预编译的时候，其实 A0 里真正的内容是这样的：

```

A0{
  a: undefined,
  b: function b() {},

```

```

c: 1,
arguments: [1],
this: window
}

```

这里边比以前讲的多两个东西，一个就是实参列表 arguments，这个肯定是有，如果没有的话你怎么能访问到呢？在一个就是 this，这个 this 假如说我们没有 new 的话，它是指向 window 的，如果在底下 new test ()，那就会隐式的在函数里 var this = Object.create(test.prototype)（这是标准写法），那这个 this 就会把前边的 this 覆盖掉，但是没 new 的话就是指向 window。试验一下：

```

function test() {
    console.log(this);
}

```

```
test();
```

此时就输出 Window{postMessage: f, blur: f, focus: f, close: f, frames: Window, ...}，你在全局访问 window 也得到 Window{postMessage: f, blur: f, focus: f, close: f, frames: Window, ...}。

(2) 全局作用域里 this 也指向 window

你在全局访问 this 也得到 Window{postMessage: f, blur: f, focus: f, close: f, frames: Window, ...}。

(3) call/apply 可以改变函数运行时的 this 指向。

(4) obj.func ();function 函数里的 this 指向 obj。

```

var obj = {
    a:function () {
        console.log(this.name);
    },
    name:'abc'
}

```

```
obj.a();
```

此时就输出 abc，就是谁调用的 this 就是谁，obj 调用的 this 就是 obj，如果是函数自己执行 this 就是 window。

3. 真题讲解

练习 3: (成哥原创精品，阿里巴巴压轴题)

```

var name = "222";
var a = {

```

```

    name: "111",
    say: function () {
        console.log(this.name);
    }
}
var fun = a.say;
fun();
a.say();
var b = {
    name: "333",
    say: function (fun) {
        fun();
    }
}
b.say(a.say);
b.say = a.say;
b.say();

```

解析：首先 `var fun = a.say;` 他会把 `a` 里边 `say` 的函数体赋给 `fun`，然后 `fun` 在全局执行，`this` 指向 `window`，输出 222，然后 `a.say()`，`a` 执行的 `this` 就是 `a`，输出 111，下边的 `b.say(a.say);` 他会把 `a.say` 这个函数体当做实参传给形参 `fun`，现在 `fun` 就是那个函数体，虽然他是在 `b` 里边执行的，但是他是自己执行，并没有 `this.fun()`，假如说里边有 `this`，那么 `this` 确实应该是 `b`，但是 `fun` 人家是自己执行的，并没有被谁调用，所以 `this` 还是指向 `window`，输出 222，然后 `b.say = a.say;`，把 `a.say` 这个函数体再赋给 `b.say`，然后 `b.say` 执行，`this` 就是 `b`，输出 333，所以最后结果是 222 111 222 333。

4. callee、caller

(1) arguments.callee

这个 `arguments.callee` 指的就是函数的引用，也就是它自己，比如说：

```

function test() {
    console.log(arguments.callee);
}
test();

```

此时就得到 `f test() {console.log(arguments.callee);}`，假如你在函数里 `console.log(arguments.callee == test);` 那就得到 `true`。

用处：比如说我们要初始化一个非常复杂的数据，求 20 的阶乘，那么就用递归来求，写一个立即执行函数，然后在外部 var 一个 num 来接受它：

```
var num = (function (n) {
    if(n == 1){
        return 1;
    }
    return n * arguments.callee(n - 1);
})(20);
```

此时访问 num 就得到 2432902008176640000，这里用递归的话，他必须调用自身才可以递归，但是立即执行函数连名字都没有，就用 arguments.callee 来表示它自己。

例：

```
function test() {
    console.log(arguments.callee);
    function demo() {
        console.log(arguments.callee);
    }
    demo();
}
test();
```

此时就得到 `f test() {console.log(arguments.callee); function demo() {console.log(arguments.callee); }demo();}`

`f demo() {console.log(arguments.callee); }`，就是 arguments.callee 在哪个函数里就代表哪个函数。

(2) function.caller

这个 caller 是函数自己的属性，比如说：

```
function test() {
    demo();
}
function demo() {
    console.log(demo.caller);
}
test();
```

这个 caller 是代表了函数的被调用者，demo 是 test 调用的，那么就输出 `f test() {demo();}`。

5. 真题讲解

练习 4: (用友试题) 请阅读以下代码, 写出以下程序的执行结果。

```
var foo = "123";  
function print() {  
    var foo = "456";  
    this.foo = "789";  
    console.log(foo);  
}  
print();
```

解析: 输出 456, 因为 this 是全局 window, 打印的是自己 A0 里边的。

变式 1:

```
var foo = 123;  
function print() {  
    this.foo = 234;  
    console.log(foo);  
}  
print();
```

解析: 这个输出 234, 因为自己 A0 里没有 foo, 就去全局里边找, 然而 this 就指向全局, 他改的就是全局的 foo。

变式 2:

```
var foo = 123;  
function print() {  
    this.foo = 234;  
    console.log(foo);  
}  
new print();
```

解析: 输出 123, 你在底下 new 了, this 就不指向 window 了, 而你访问的是全局的 foo。

练习 5: 运行 test() 和 new test() 的结果分别是什么?

```
var a = 5;  
function test() {  
    a = 0;  
    alert(a);  
    alert(this.a);  
    var a;
```

```
    alert(a);
}
```

解析：运行 test() 的时候 alert 的结果是 0 5 0，因为第二个 alert 里的 this 指向全局 window。运行 new test() 的时候输出 0 undefined 0，因为你 new 了，this 就代表一个对象了，对象里没有 a，所以就是 undefined。

练习 6：（用友试题）请阅读以下代码，写出以下程序的执行结果。

```
function print() {
    console.log(foo);
    var foo = 2;
    console.log(foo);
    console.log(hello);
}
```

```
print();
```

解析：这道题考的是预编译，结果是 undefined 2 报错

练习 7：（用友试题）请阅读以下代码，写出以下程序的执行结果。

```
function print() {
    var test;
    test();
    function test() {
        console.log(1);
    }
}
```

```
print();
```

解析：还是考预编译，函数会提升且覆盖掉变量声明，最后打印 1.

练习 8：（用友试题）请阅读以下代码，写出以下程序的执行结果。

```
function print() {
    var x = 1;
    if(x == "1")console.log('One!')
    if(x === "1")console.log('Two!')
}
```

```
print();
```

解析：考的是隐式类型转换，打印 One！（判断结果只有一句话可以不写大括号）

练习 9：（用友试题）请阅读以下代码，写出以下程序的执行结果。

```
function print() {
```

```
var marty = {
  name: "marty",
  printName: function () {
    console.log(this.name);
  }
}
var test1 = {
  name: "test1"
}
var test2 = {
  name: "test2"
}
var test3 = {
  name: "test3"
}
test3.printName = marty.printName;
marty.printName.call(test1);
marty.printName.apply(test2);
marty.printName();
test3.printName();
}
print();
```

解析：test3.printName = marty.printName;就是把 marty 的 printName 函数体赋给 test3，然后 marty.printName.call(test1);，call 后边传 test1，this 就是 test1，输出 test1，marty.printName.apply(test2);，apply 和 call 是一样的，this 就是 test2，输出 test2，marty.printName();，marty 调用的 this 就是 marty，输出 marty，test3.printName();，this 就是 test3，输出 test3，所以最后结果为：test1 test2 marty test3.

练习 10：（用友试题）请阅读以下代码，写出以下程序的执行结果。

```
var bar = {
  a: "002"
}
function print() {
  bar.a = "a";
```

```

Object.prototype.b = "b";
return function inner() {
    console.log(bar.a);
    console.log(bar.b);
}
}

```

```
print()();
```

解析：输出 a b，函数里第一句改的是外边 bar 里的 a，然后第二次打印 bar.b 就去原型链上找。最后俩 () 就是第一个执行后返回的函数再执行。

克隆、三目运算符

1. 浅层克隆

比如说现在有：

```

var obj = {
    name: "abc",
    age: 123,
    sex: "female"
}

```

```
var obj1 = {};
```

我现在有一个对象 obj，obj 里有一些属性和属性值，还有一个 obj1，他是一个空对象，现在我想把 obj 里的属性克隆给 obj1，写一个克隆的方法：

```

function clone(origin, target) {
    var target = target || {};
    for(var prop in origin) {
        target[prop] = origin[prop];
    }
    return target;
}

```

```
clone(obj, obj1);
```

这里边传两个形参进去，origin 代表原始对象，target 代表目标对象，然后用 for in 循环把原始对象的每个属性赋给目标对象即可，我们在上边容个错，就是有时候假如说人家没有传第二个参数进来的话，只提供一个原始对象，我们就自己生产一个空对象并在最后返回即可，现在访问 obj1 就得到 {name: "abc", age: 123, sex: "female"} 这么写的话原始值没毛病，比如说现在把 obj.name = "bcd"，访问 obj1.name 还是 abc，

原始值没问题，但是引用值就有问题了，现在我把 obj 改成：

```
var obj = {  
  name: "abc",  
  age: 123,  
  sex: "female",  
  card: ["visa", "unionpay"]  
}
```

然后经过克隆后，我们访问 obj1 就得到 {name: "abc", age: 123, sex: "female", card: Array(2)}，现在我 obj1.card.push("master")，访问 obj1.card 就得到 ["visa", "unionpay", "master"]，但是我们访问 obj.card 也得到 ["visa", "unionpay", "master"]，因为是引用值，他俩都指向同一个房间，我改你也改。

2. 深层克隆

```
var obj = {  
  name: "abc",  
  age: 123,  
  card: ["visa", "master"],  
  wife: {  
    name: "bcd",  
    son: {  
      name: "aaa"  
    }  
  }  
}
```

```
var obj1 = {};
```

现在我们来写一个深度克隆，就是我克隆过来的东西还是和你一模一样，但是就是不是一个人，你改的话我不改，不互相影响，这就叫深度克隆。

分析：

我们先要看这个值是不是原始值，是原始值的话按原来的方法来，是引用值的话就看看他是数组还是对象，是数组的话就建立一个空数组，对象的话就建立一个空对象，然后一层一层剥开看，因为引用值不能直接拷贝，拷过来的是地址。这里就用到了递归，因为引用值里边可能还有引用值，得挨个看，挨个分析，而且看的过程中和第一次看对象的过程是一样的，找到了规律，就用递归。

步骤：

第一步：先用 for in 循环遍历对象，看看每一个属性是原始值还是引用值（用 typeof()）

判断，如果是引用值就返回 object，否则就是原始值），如果是原始值就直接拷贝。

第二步：如果是引用值就看看是数组还是对象（建议使用 toString，另外两个 instanceof 和 constructor 会有个跨父子域的问题，后期会讲到），如果是数组就建立空数组，如果是对象就建立空对象。

第三步：把原来的引用值和要被拷贝的引用值当做一对新的克隆对象，然后再一次进行新的循环判断的步骤。（递归）

综上，得出以下代码：

```
function deepClone(origin, target) {
    var target = target || {},
        toString = Object.prototype.toString,
        arrStr = "[object Array]";
    for (var prop in origin) {
        if (origin.hasOwnProperty(prop)) {
            if (origin[prop] !== "null" && typeof (origin[prop]) == "object") {
                if (toString.call(origin[prop]) == arrStr) {
                    target[prop] = [];
                } else {
                    target[prop] = {};
                }
                deepClone(origin[prop], target[prop]);
            } else {
                target[prop] = origin[prop];
            }
        }
    }
    return target;
}
```

解析：现在我们调用 deepClone(obj, obj1)，然后访问 obj1 得 {name: "abc", age: 123, card: Array(2), wife: {...}}，访问 obj.card 得到 ["visa", "master"]，访问 obj1.card 也得 ["visa", "master"]，现在我 obj.card.push("abc")，访问 obj.card 得到 ["visa", "master", "abc"]，但是访问 obj1.card 还是 ["visa", "master"]。这就是按照上边所说的三个步骤写出来的深度克隆，这里说三点：我们一般在写一个方法的时候尽量把它写全，所以加上 hasOwnProperty 来过滤一下，因为我们并不希望拿到原型上的东西；在判断他是不是引用值的时候，我们用到了 typeof，但是返回 object 的话他有三

个值，数组、对象和 null，我们就加个条件把 null 排除掉，就是当它绝对不等于 null 并且 typeof 返回 object 的时候，那么他不是数组就是对象；递归的时候他会一直循环的判断，那什么时候才是出口？else 之后当它全部是原始值的时候就是出口。

3. 三目运算符

三目运算符的结构是这样的：

条件判断 ? 是 : 否

问号前边是一个条件判断，判断是的话走冒号前边的，判断否的话走冒号后边的，并且会返回值。

例如：`var num = 1 > 0 ? 2 + 2 : 1 + 1;`他会先看三目运算符，判断 `1 > 0`，是的话返回 `2+2`，后边的就不看了，否的话返回 `1+1`，所以访问 num 得 4。

`var num = 1 < 0 ? 2 + 2 : 1 + 1;`，此时访问 num 就是 2。

其实这就是一个简化版的 if else，而且他比 if 语句还高级，因为他还可以返回值，相当于 if else 再加一个 return，再比如：

`var num = 1 > 0 ? ("10" > "9" ? 1 : 0) : 2;`这个先算括号里的，字符串和字符串比较，比较的是自己的阿斯克码，字符串的 10 是不大于 9 的，所以括号里返回 0，在看括号外的，1 大于 0，所以 num 得 0。

我们用三目运算符把深度克隆简化后得：

```
function deepClone(origin, target) {
    var target = target || {},
        toStr = Object.prototype.toString,
        arrStr = "[object Array]";
    for (var prop in origin) {
        if (origin.hasOwnProperty(prop)) {
            if (origin[prop] !== "null" && typeof (origin[prop]) == "object") {
                target[prop] = toStr.call(origin[prop]) == arrStr ? [] : {};
                deepClone(origin[prop], target[prop]);
            } else {
                target[prop] = origin[prop];
            }
        }
    }
    return target;
}
```

数组

我们之前讲过数组，他和对象一样，也属于引用值。

1. 数组定义方式

第一种：数组字面量 `var arr = [];`

比如说 `var arr = [1, 2, 3];` 你中间也可以只写逗号，不加内容，`var arr = [1, 2, , , 3, 4];` 这叫做稀疏数组，中间三个值没填内容，会返回 3 个 `undefined`，你再访问 `arr.length` 就是 7。

第二种：构造方法定义数组 `var arr = new Array();`，这是系统提供的构造方式，所以说数组能用的一切方法全部来源于 `Array.prototype`。其实你写一个字面量的形式就相当于 `new Array()`。

比如说 `var arr = new Array(1, 2, 3, 4, 5)`，这个括号里也可以传参，现在访问 `arr` 就是 `[1, 2, 3, 4, 5]`；这个和字面量基本没啥区别，只有一丁点小的区别，比如说 `var arr = [10]`，那么你访问 `arr` 就得到 `[10]`，但是 `var arr1 = new Array(10)`，假如说你括号里只写一个参数的话，它不会把实参当成数组的第一位，他会当成长度为 10 的稀疏数组，你再访问 `arr1` 就得到 10 个 `undefined`。以后你想创建一个固定长度的空数组就只能这么创建。那么就会产生一种矛盾：

`var arr = [10.2]`，这个数组的第一位就是 10.2。

`var arr1 = new Array(10.2)`，这么写就会报错，告诉你长度是非法的，因为他不会把它当数组的第一位，他会当成数组的长度，长度不能是小数，就会报错，所以记住，这么写的话第一位不能写小数。

不管是数组还是对象，都推荐使用字面量的创建方法，除非你有特殊情况，比如说批量生产，加个原型等等。

2. 数组的读和写

JavaScript 里的数组规定极其松散，基本上没有什么报错的地方，比如说 `var arr = [];` 你访问 `arr[10]` 就是 `undefined`，`arr[10] = "abc"`，那么你访问 `arr` 就得到 10 个 `undefined` 和 `abc`，你访问 `arr.length` 就是 11。

原因：数组是基于对象的，他就是一种特殊的对象，比如说 `var obj = {};` 你访问 `obj[10]` 也是 `undefined`。（对象的属性名可以用数字）

3. 数组常用的方法

因为数组是引用值，所以他的方法是自身就有的。

第一种 改变原数组的方法：

(1) `push`:

`push` 是给数组的最后一位加数据的，添加任何类型的数据都可以。

比如说 `var arr = [], arr.push(1)`; 访问 `arr` 就是 `[1]`, 继续 `arr.push(2)`; 访问 `arr` 就得到 `[1, 2]`。

再比如 `var arr1 = []; arr1.push(1, 2, 3)`; 访问 `arr1` 就是 `[1, 2, 3]`。

我们来写一下 `push` 方法:

```
Array.prototype.push = function () {  
    for(var i = 0; i < arguments.length; i++){  
        this[this.length] = arguments[i];  
    }  
    return this.length;  
}
```

解析: 因为我们不知道谁调用他, 所以我们只能用 `this`, 函数里也没办法传参数, 因为你不知道实参到底有几位, 但是我们可以用 `arguments`, 用 `for` 循环把实参列表的每一位都赋给 `this` 的最后一位即可, `this` 的最后一位怎么表示? 比如说 `this` 现在有两位数, 那么我们就给第三位加东西, 第三位就是 `this[2]`, 也就是 `this[this.length]`。然后循环结束后返回 `this.length` 即可。现在 `var arr = [1, 2]`, 我们 `arr.push(3, 4)`; 再访问 `arr` 就得 `[1, 2, 3, 4]`。

(2) pop

`pop` 就是从数组的最后一位剪切, 比如说 `var arr = [1, 2, 3], arr.pop()`, 再访问 `arr` 就得 `[1, 2]`, 剪切就是说, 比如我继续 `var num = arr.pop()`; 此时我们访问 `num` 就是 `2`, 访问 `arr` 就是 `[1]`, 他把 `2` 剪切出去了 (其实剪切后返回 `2`, 访问 `arr` 得 `[1]`, 他会把剪切的值返回)。

注意: `pop` 传参数不起作用, 比如说 `var arr = [1, 2, 3], arr.pop(2)`, 再访问 `arr` 还是 `[1, 2]`。

(3) shift

`shift` 是从数组前边剪切, 他和 `pop` 方向相反, 比如说 `var arr = [1, 2, 3]; arr.shift()`; 他会返回 `1`, 再访问 `arr` 就是 `[2, 3]`。

(4) unshift

`unshift` 是给数组前边加东西的, 他和 `push` 的方向相反, 比如说 `var arr = [1, 2, 3]; arr.unshift(0)`; 再访问 `arr` 就是 `[0, 1, 2, 3]`。

思考: `unshift` 方法怎么写出来? 你不能 `this[0]`, 这样会覆盖的, 你只能在方法里新建一个数组, 然后把两个数组拼接起来。

(5) reverse

`reverse` 就是逆转, 比如 `var arr = [1, 2, 3]; arr.reverse()`; 此时访问 `arr` 就是 `[3, 2, 1]`, 继续 `arr.reverse()`, 访问 `arr` 就是 `[1, 2, 3]`。

(6) splice

splice 是切片的意思，比如说现在 `var arr = [1, 1, 2, 2, 3, 3]`，这个 splice 括号里可以传很多参数，第一位参数是从第几位开始，第二位参数是截取多少的长度，第三位参数及以后是在切口处添加新的数据。比如说现在 `arr.splice(1, 2)`，第三位可以不传，先来看两位的，它是从第一位开始（包括第一位），这里第一个 1 是第 0 位，第二个 1 是第 1 位，从这里开始截取两位，所以此时访问 arr 就是 `[1, 2, 3, 3]`。其实截取就是剪切，你也可以 var 一个变量来接收他截取的数（其实返回 `[1, 2]`，访问 arr 得 `[1, 2, 3, 3]`）。再比如 `arr.splice(0, 3)`，就是从第 0 位开始切三位（返回 `[1, 1, 2]`），访问 arr 就是 `[2, 3, 3]`。再 `arr.splice(1, 1, 0, 0, 0)`，从第 1 位开始切 1 位（返回 `[1]`）然后加上后边的参数，此时访问 arr 就是 `[1, 0, 0, 0, 2, 2, 3, 3]`。

比如说现在 `var arr = [1, 2, 3, 5]`，我们要把 4 填在 3 和 5 中间，即 `arr.splice(3, 0, 4)`，从第三位开始切 0 个，然后加上 4，此时访问 arr 就是 `[1, 2, 3, 4, 5]`。那么。unshift 的方法就好办了，它是从第 0 位开始截 0 个，然后往里加东西。所以说 splice 非常强大，他会把截取的东西返回回来，还改变原数组。

再比如：`var arr = [1, 2, 3, 4]`，`arr.splice(-1, 1)`；数组是有负数位的，-1 位就是倒数第一位，所以返回 `[4]`，访问 arr 得 `[1, 2, 3]`。

思考：系统内部是怎么把负数转化为正数的？其实原理是这样的，我们用 pos 来代表数字位：`pos += pos > 0 ? 0 : this.length`；如果 pos 大于零，就加零等于不加，如果小于零就加 length，例如上边的，`-1 + 4 = 3`；第三位正好是倒数第一位。怎么防止 pos 越界？当 `0 <= pos < length` 的时候 pos 才有意义，否则就是越界。

(7) sort

sort 就是给数组排序的。比如说现在 `var arr = [1, 3, 4, 0, -1, 9]`，然后 `arr.sort()`；再访问 arr 就是 `[-1, 0, 1, 3, 4, 9]`，这是升序，我想降序的话就 `arr.sort().reverse()`；此时访问 arr 就是 `[9, 4, 3, 1, 0, -1]`。但是这是按照字符的阿斯克码排的，`var = [1, 3, 5, 4, 10]`，`arr.sort()`，此时访问 arr 得到 `[1, 10, 3, 4, 5]`。所以说这并不是我们理想中的排序，他全当成字符了，按阿斯克码排的。所以假如我们想按数字排序的话，sort 是满足不了我们的，但是他给我们留了一个接口，那个传参数的括号里可以写一个方法，写了这个方法，我们就可以按我们的意思给数字排序了：

```
arr.sort(function (a, b) {
```

```
})
```

这个括号里是一个匿名函数，匿名的，不写名字也可以，他就是一个引用，而且里边必须传两个形参进去，这个函数用不着你自己调用，你把函数当做参数传进去之后系统会在合适的时候帮你调用的，我们只需要把规则写在函数里即可。

规则：

1) 必须写俩形参；

2) 看返回值：当返回值为负数时，那么前面的数放在前面；为正数时，后面的数在前；为0时，不动。

解释：var arr = [1, 3, 5, 4, 10]，他会调用很多次这里边的函数。怎么调用？第一次调用的时候他会把数组的第0位和第1位传进来，把1和3传进来，通过我们的一系列规则来进行比较，当返回值为正数的时候，他会让后面的数在前，也就是1和3调换位置，当返回负数的时候，前面的数就在前，就不变，那么具体什么时候返回值为正？什么时候为负？这就是咱们控制了，系统第一次传1 3，然后1 5, 1 4, 1 10, 3 5, 3 4, 3 10, 5 4, 5 10, 4 10依次传参（这是按位置传，不按数字传），这符合冒泡排序的算法，现在我定义规则：

```
arr.sort(function (a,b) {  
    if(a > b) {  
        return 1;  
    }else {  
        return -1;  
    }  
});
```

这个1和-1的作用只是为了代表正负。试一下：

第一次：1 > 3 为 false，返回-1，不动。

第二次：1 > 5 为 false，返回-1，不动。

第三次：1 > 4 为 false，返回-1，不动。

第四次：1 > 10 为 false，返回-1，不动。

第五次：3 > 5 为 false，返回-1，不动。

第六次：3 > 4 为 false，返回-1，不动。

第七次：3 > 10 为 false，返回-1，不动。

第八次：5 > 4 为 true，返回1，5和4调换位置得[1, 3, 4, 5, 10]。

第九次：4 > 10 为 false，返回-1，不动。

第十次：5 > 10 为 false，返回-1，不动。

函数调用了十次，最后结果[1, 3, 4, 5, 10]，这是按照系统给咱们留的接口弄出来的升序。

我们再来试一个，var arr = [2, 10, 20, 13, 4, 18, 9];这次简单点写哈：

第一圈：2走一圈比完之后都返回-1，不动。

第二圈：10和4比返回1，调换位置，改为[2, 4, 20, 13, 10, 18, 9];4和18、9比都不动。

第三圈：20 和 13 比调换位置，改为[2, 4, 13, 20, 10, 18, 9]，13 和 10 比，继续换，改[2, 4, 10, 20, 13, 18, 9]，10 和 18 比，不动，10 和 9 比继续换:[2, 4, 9, 20, 13, 18, 10]。
 第四圈：20 和 13 换位置：[2, 4, 9, 13, 20, 18, 10]，13 和 18 比，不动，13 和 10 换位置：[2, 4, 9, 10, 20, 18, 13]。

第五圈：20 和 18 换：[2, 4, 9, 10, 18, 20, 13];18 和 13 换：[2, 4, 9, 10, 13, 20, 18]。

第六圈：20 和 18 换：[2, 4, 9, 10, 13, 18, 20]。

最后结果：升序，调用 sort 接口执行上边定义的规则后访问 arr 得 [2, 4, 9, 10, 13, 18, 20]。

好，现在我们回归 var arr = [2, 10, 20, 13, 4, 18, 9];，改变规则：

```
arr.sort(function (a,b) {
    if(a < b) {
        return 1;
    }else {
        return -1;
    }
});
```

现在我们当 a 小于 b 的时候返回正数，改变位置：2 先和 10 换，然后 10 和 20 换……就像刚才推导的过程一样，你会发现，前边第一圈结束的时候最小的在前边，现在第一圈结束最大的在前边，前边是升序，现在结束后就是降序。所以我们得出一个结论：就我们写的这个方法是可以控制升序和降序的，这里边当 $a > b$ 为升序，当 $a < b$ 为降序。

拓展：我们来看升序这个规则：

```
arr.sort(function (a,b) {
    if(a > b) {
        return 1;
    }else {
        return -1;
    }
});
```

当 a 大于 b 的时候， $a - b$ 肯定大于 0，所以我们返回 $a - b$ 就可以表示正数，如果 a 小于 b，那么 $a - b$ 肯定小于 0，同样返回 $a - b$ 就可以表示负数，那么，不管咋样，返回的都是 $a - b$ ，那我们直接返回 $a - b$ 不就得了吗？所以升序的简化后得：

```
arr.sort(function (a,b) {
    return a - b;
```

```
});
```

同理，降序的也可以简化：

```
arr.sort(function (a,b) {  
    return b - a;  
});
```

练习 1：给一个有序的数组乱序。

```
arr.sort(function () {  
    return Math.random() - 0.5;  
})
```

解析：我们之前讲的都是有理可寻的，那当我们没理可寻的时候，让他随机的返回正数或者负数，就是乱排了，我们以前说过有个 `Math.random()`，他会随机的产生 0-1 的开区间数，就是随机小数，两边不会到头，我们把产生的随机数减去 0.5 后返回，那么每次调用的返回值就有可能是正数，也有可能是负数了，就乱序了。

练习 2：

```
var cheng = {  
    name: "cheng",  
    age: 18,  
    sex: "male",  
    face: "handsome"  
}
```

```
var deng = {  
    name: "deng",  
    age: 40,  
    sex: undefined,  
    face: "amazing"  
}
```

```
var zhang = {  
    name: "zhang",  
    age: 20,  
    sex: "male"  
}
```

```
var arr = [cheng, deng, zhang];
```

把这三个对象按照年龄排序，升序。

```
arr.sort(function (a,b) {
```

```
    return a.age - b.age;
  })
```

解析：既然 a 和 b 代表那几个对象，那我们就把 age 属性取出来排序即可。只要你知道这个 a, b 代表什么，那么 sort 用起来就会非常灵活。

练习 3:

```
var arr = ["abc", "bcd", "cccc", "dddd", "asdfkhiuqwe", "asdoifqwoeiur", "asdf"];
把这个数组按照字符串的长度排序。
```

```
arr.sort(function (a,b) {
    return a.length - b.length;
})
```

解析：此时访问 arr 就是["abc", "bcd", "cccc", "dddd", "asdf", "asdfkhiuqwe", "asdoifqwoeiur"]

练习 4:

```
var arr = ["a 邓", "ba 邓", "cc 邓 cc", "老邓", "残邓", "asdoifqwoeiur", "asdf"];
把这个数组按照字节长度排序。
```

```
function retBytes(str) {
    var num = str.length;
    for(var i = 1; i < str.length; i++){
        if(str.charCodeAt(i) > 255) {
            num ++;
        }
    }
    return num;
}
```

```
arr.sort(function (a,b) {
    return retBytes(a) - retBytes(b);
})
```

解析：我们以前讲过求字节长度，先定义一个方法，把字节长度返回，然后再用 sort 调用定义好的方法排序即可，此时访问 arr 就是["a 邓", "老邓", "残邓", "ba 邓", "asdf", "cc 邓 cc", "asdoifqwoeiur"]

第二种 不改变原数组的方法

(1) concat

concat 的作用是连接两个数组

比如说 var arr = [1,2,3,4,5,6];var arr1 = [7,8,9];现在我们 arr.concat(arr1)

就会返回[1, 2, 3, 4, 5, 6, 7, 8, 9]。我们之前讲的都是改变原数组的，这种是不改变原数组，他会把新的返回，原来的 arr 和 arr1 不变。

(2) toString

数组的 toString 就是把它变成字符串展示出来，比如说 `var arr = [1, 2]; arr.toString()` 就返回 “1, 2”。

(3) slice

slice 和 splice 有点相似，他可以不传参数，也可以传一个参数，也可以传俩参数。比如说 `var arr = [1, 2, 3, 4, 5, 6]`

传两个参数：就是从第几位开始截，截取到第几位，`arr.slice(1, 2)` 就是从第一位开始截取到第二位，返回[2]，原数组依然不变，所以截取的话你必须用变量来接收，例如 `var newArr = arr.slice(1, 3)`，此时访问 newArr 就是[2, 3]。

传一个参数：就是从第几位开始截取，一直截取到最后，`var newArr = arr.slice(1)`，此时访问 newArr 就是[2, 3, 4, 5, 6]。当然这里边你也可以传负数位哈。

不传参数：整个截取，`var newArr = arr.slice()`，此时访问 newArr 就是[1, 2, 3, 4, 5, 6]。

(4) join 和 split

join: `var arr = [1, 2, 3], arr.join(“-”)`; 这个 join 里边必须传字符串类型的参数，传进去后他会把数组的每一位连接起来并返回一个字符串，此时就返回 “1-2-3”，`arr.join(“!”)` 就返回 “1!2!3”。

split: split 是字符串的一个方法，他和数组的 join 方法是互逆的，`var str = “1-2-3”, str.split(“-”)` 这个就是按照 “-” 拆分成数组，此时就返回[“1”, “2”, “3”]。假如 `str.split(“2”)`，按照 2 拆就返回[“1-”, “-3”]。

练习 5:

```
var str = "alibaba";
```

```
var str1 = "baidu";
```

```
var str2 = "tencent";
```

```
var str3 = "toutiao";
```

```
var str4 = "wangyi";
```

把上边的连成一个字符串。

```
var arr = [str, str1, str2, str3, str4];
```

```
console.log(arr.join(""));
```

解析：以前我们用加号也可以，但是操作的是栈内存，翻来覆去的不好，我们把他存到数组里，就是散列结构，连接起来比较好一点，括号里穿个空串就可以了（必须传参数，不传中间就是逗号），此时就打印 alibababaidutencenttoutiaowangyi。

类数组、封装 type 方法、数组去重

1. 类数组

类数组就是长得像数组，但他就不是数组，数组有的方法他全没有，我们以前接触过一个类数组：arguments。比如：

```
function test() {  
    console.log(arguments);  
}
```

```
test(1, 2, 3, 4, 5, 6);
```

此时就输出[1, 2, 3, 4, 5, 6]，但是 arguments.push(7);就会报错，说明他不是数组，但是他还长得像数组，我们就把他叫做类数组。我们来探究一下类数组是怎么构成的：

```
var obj = {  
    "0": "a",  
    "1": "b",  
    "2": "c",  
    "length": 3,  
    "push": Array.prototype.push  
}
```

我们以前说过，对象的属性可以是任何一种形式，而且属性名可以加双引号，我们 push 里放的是 Array 的方法，这样一个类数组就构建好了，现在 obj.push("d")，再访问 obj 就是{0: "a", 1: "b", 2: "c", 3: "d", length: 4, push: f}，你会发现多了一个 3: "d"，而且 length 变了。这都是一个对象不能具备的东西，为啥加了一个 push 就又加东西又变东西啊？所以说这就是一个类数组，类数组必须有以下几个组成部分：

- (1) 属性要为索引（数字）属性，
- (2) 必须有 length 属性
- (3) 最好加上 push

其实这还是一个对象，只不过能当数组来用，但是这个访问后也不像数组，那么我再加一个方法：

```
var obj = {  
    "0": "a",  
    "1": "b",  
    "2": "c",  
    "length": 3,
```

```
"push": Array.prototype.push,  
"splice": Array.prototype.splice  
}
```

这回访问 obj 就是["a", "b", "c", push: f, splice: f], 这成了一种定律了, 记住只要你给对象加上 splice 方法后, 他就和数组长得一样了, 它是对象, 但是可以当数组来用, 这就叫类数组。

我们再来研究一下 push, 咱就假设 push 里只能传一个参数, 那么:

```
Array.prototype.push = function (target) {  
    this[this.length] = target;  
    this.length++;  
}
```

这个 push 里边有这样一系列操作手法, 那么, 如果对象调用他的话, this 就是对象, 而我们正好给他加了 length 属性, 有 length 就能经过一系列处理了, 那刚才 obj.push("d"), 第 length 位就是第 3 位, 得到"3": d, 然后 length++得 4.

以前人们并不重视类数组, 直到有一年, 阿里巴巴出了一道非常恶心的题后, 人们开始关注类数组了。

练习: (阿里巴巴试题)

```
var obj = {  
    "2": "a",  
    "3": "b",  
    "length": 2,  
    "push": Array.prototype.push  
}
```

obj.push("c");

obj.push("d");

问: obj 现在长什么样子?

答:

```
var obj = {  
    "2": "c",  
    "3": "d",  
    "length": 4,  
    "push": Array.prototype.push  
}
```

解析: 咱们上边都把 push 写好了, 直接套就行了, 第一次调用 length 是 2, 那么就是

"2": "c", 然后 length++得 3, 第二次调用就是"3": "d", 然后 length++得 4. 他会把刚才的覆盖掉的, 所以类数组的关键点是 length, push 里读的是 length, 所以必须把 length 搞懂。

变式:

```
var obj = {
  "1": "a",
  "2": "c",
  "3": "d",
  "length": 3,
  "push": Array.prototype.push
}
```

obj.push("b");

问: obj 现在长什么样子?

答:

```
var obj = {
  "1": "a",
  "2": "c",
  "3": "b",
  "length": 4,
  "push": Array.prototype.push
}
```

类数组的好处: 类数组可以当数组用, 也可以当对象来用, 他具备数组和对象的两种特性的话, 那么他存储数据就会更强大一些。比如

```
var obj = {
  "0": "a",
  "1": "b",
  "2": "c",
  name: "abc",
  age: 123,
  length: 3,
  "push": Array.prototype.push,
  splice: Array.prototype.splice
}
```

现在你访问 obj 就是["a", "b", "c", "b", name: "abc", age: 123, push: f, splice:

f], 访问 obj.name 就是"abc", 访问 obj.age 就是 123, 访问 obj.length 就是 4, 所以他既能当数组用, 也能当对象用, 你也可以用 for(prop in obj) {console.log(obj[prop])} 把他们的属性全部遍历出来。

类数组不好的一点: 并不是数组的所有方法他都能用, 除非你自己动手添上去。

2. 封装 type 方法

要求: 我们之前接触过 typeof, 但是这个方法不精准, 现在我们自己封装一个 type 方法, 要求彻底把他们区分开: 数组返回 array, 对象返回 object, null 就返回 null, 包装类的话返回 number/string/boolean-object 等等。

```
function type(target) {  
    var ret = typeof (target);  
    var template = {  
        "[object Array]": "array",  
        "[object Object]": "object",  
        "[object Number]": "number - object",  
        "[object Boolean]": "boolean - object",  
        "[object String]": "string - object"  
    }  
    if (target === null) {  
        return "null";  
    } else if (ret == "object") {  
        var str = Object.prototype.toString.call(target);  
        return template[str];  
    } else {  
        return ret;  
    }  
}
```

解析: 思路是这样的, 我们先要区分开是原始值还是引用值, 如果是原始值就直接返回 typeof 结果, 如果是引用值的话就再用 Object.prototype.toString.call 继续区分是数组还是对象还是包装类, 我们 var 一个对象把返回的结果存好, 再 var 一个 str 等于 toString 的返回结果, 然后我们 return 的时候直接返回 template 对应的属性值即可, 这么写的话方便一些。有两点特殊的: null 是原始值, 但是 typeof 的时候却把它归到 object 里了, 我们单独把他摘出来即可; 还有 function 是引用值, 但是 typeof 后还是他, 我们就把 function 和原始值写在一起, 这样代码量能少点儿。

3. 数组去重

要求：比如说 `var arr = [1, 1, 1, 2, 2, 5, 5, 5, 5]`，去重后返回 `[1, 2, 5]`，在原型链上编程。

```
Array.prototype.unique = function () {
    var temp = {},
        arr = [],
        len = this.length;
    for (var i = 0; i < len; i++) {
        if(!temp[this[i]]){
            temp[this[i]] = "abc";
            arr.push(this[i]);
        }
    }
    return arr;
}
```

解析：思路是这样的，我们要用到对象，一个对象的属性值可以重复，但是他的属性名是独一无二的，我们先新建一个对象，然后把目标数组的每一位都拿出来当做对象的属性名，去重后，`var` 一个新的数组 `arr`，然后把对象里的属性名再 `push` 给这个 `arr` 并返回 `arr` 即可。

进一步推导：我们拿题上的数组举个例子，`arr.unique()`，`arr` 调用的 `this` 就是 `arr`，我们用 `for` 循环把数组的每一位都遍历之后，然后把每一位的值都当做 `temp` 的属性名，第一圈取第 0 位的时候，`this[0]` 就是 1，然后 `if` 里最开始 `temp[1]` 是 `undefined`，为 `false`，我们在前边加上非!就是 `true` 了，走进去后，给对象 `temp[1]` 赋值，原来是 `undefined`，现在是“abc”，然后把 `1` push 到 `arr` 里。然后循环第二圈，`this[1]` 还是 1，此时 `if` 里的条件 `temp[1]` 已经是“abc”了，为 `true`，加上非就是 `false`，所以根本走不进去，第三圈 `this[2]` 还是 1，`if` 依然执行不了，第四圈 `this[3]` 是 2，所以 `temp[2]` 是 `undefined`，加上非走进去继续赋值 `abc`，然后给 `arr` `push` 一个 2 进去，依次类推最后返回 `arr` 就是 `[1, 2, 5]`。

注意：`if` 里赋值的时候赋一个布尔值为 `true` 的任何值都可以，但是不能付 `false` 值，赋 `false` 值 `if` 里判断是 `false` 加非是 `true`，就都能走进来了，那就达不到去重的效果。

ES 知识点补充、复习题讲解

1. 知识点补充

(1) 属性的可配置性和不可配置性

在全局 `window` 里，一旦经历了 `var` 的操作所得出的属性，这种属性叫做不可配置的属性

性，不可配置的属性，delete 不掉。

解释：比如 `var num = 123`，你 `delete num` 就是 `false`，`delete window.num` 也是 `false`，因为你在前边 `var` 了，他就是不可配置的属性。再比如 `var obj = {}`，`obj.num = 123`，`delete num` 就是 `true`，还有 `window.num = 123`，`delete window.num` 也是 `true`。记住有 `var` 的话就成了不可配置性属性，delete 不掉，没有 `var` 的话就是可配置的。

(2) 引用值的类型转换

引用值的类型转换比较复杂，以后基本用不上，我们不做深入的研究，这里只写几个特殊的：`[] + "" = ""`；`[] + 1 = "1"`；`[] - 1 = -1`；`Number([]) = 0`；`{ } + 1 = 1`；`Number({ }) = NaN`；哎呀，这里边的东西太恶了，我们也没必要去深入研究，你对象加一干嘛啊？减一干嘛啊？一天闲的，一点儿用都没有哈。你只需要知道 `[] == []` 是 `false`，因为引用值比的是地址，他俩指向两个不同的房间，所以是 `false`，`[] === []` 也是 `false`。其他的就不用考虑了。

2. 复习题讲解

练习 1：（微店笔试题）

```
(function (x) {  
    delete x;  
    return x;  
})(1)
```

解析：这个题特别简单，形参是 `x` 就相当于 `var x`，既然 `var` 了就删除不了，所以返回 1。

练习 2：（微店笔试题）

```
function test() {  
    console.log(typeof arguments);  
}  
test();
```

解析：object。

练习 3：（微店笔试题）

```
var h = function a() {  
    return 23;  
}  
console.log(typeof a());
```

解析：这道题会报错，这是一个函数表达式，所以那个 `a` 就跟没写一样，那就执行不了 `a`，所以会报错。

try-catch、ES5 严格模式

1. try-catch

try 的意思是尝试，catch 是捕捉，这个 **try catch 就是防止我们报错的**。比如说我现在写了好多行代码，这段代码里边有可能报错，也有可能不报错，我把握不了，而我又不想让报错的这一行影响后续代码的执行，这个时候就用 try catch，比如说：

```
try{  
    console.log("a");  
    console.log(b);  
    console.log("c");  
}catch(e){  
  
}
```

```
console.log("d");
```

我们把代码写在 try 里边，现在第二行 b 报错了，但是他不会抛出错误，try 外边的代码依然可以执行，但是在 try 里边发生的错误，不会执行错误后的 try 里边的代码，例如上边的，b 那行出错后，c 那行也不会执行，但是 try 外边的 d 可以执行，此时就输出 a d。假如说 try 里边没有发生错误，他就会正常执行的，就是说如果你不确定这段代码是不是有错误，你就把它放到 try 里边，他不会影响外边代码的执行。那么，catch 是干嘛的？

```
try{  
    console.log("a");  
    console.log(b);  
    console.log("c");  
}catch(e){  
    console.log(e.name + ":" + e.message)  
}  
console.log("d");
```

如果 try 里边的代码没有错误，那么 catch 就不会执行，这个 catch 的作用就是捕捉错误信息的，一般情况下系统会把报错信息抛给控制台，显示的时候是红色的，但是有了 catch 后，他就会捕捉到错误信息，不会抛给控制台，捕捉之后你想打印或者弹出啥的都可以，就交给你让你自己处理了，这个错误信息是一个对象 Error，他里边就只有两个属性 name 和 message，在 catch 括号里传一个参数 e（参数名不固定），然后我们例如上边的，打印错误信息，他执行的时候先走 try，再走 catch，再走外边的，

此时就输出 `a ReferenceError:b is not defined d`，如果 `try` 里有多行都是错的，他就只看第一行错误后，后边的就不看了。这个 `try catch` 的好处就是不会影响后续代码的执行。

2. `Error.name` 六种值对应的错误信息：

(1) **EvalError**: `eval()` 的使用与定义不一致

解释：这个 `eval` 是不允许我们用的，如果用了的话他就会显示这个错误信息。

(2) **RangeError**: 数值越界

解释：这个很少见，以后遇到了会讲的。

(3) **ReferenceError**: 非法或不能识别的引用数值

解释：当一个变量未经声明就使用或者当一个函数未经声明就调用就会显示这个。

(4) **SyntaxError**: 发生语法解析错误

解释：代码在执行前系统先通篇扫描一遍，如果有低级语法错误就显示这个，例如写了一个中文，少写一个括号等等。

(5) **TypeError**: 类型操作错误

解释：比如说用数组方法操作对象了就会显示这个，后边遇到了再说。

(6) **URIError**: URI 处理函数使用不当

解释：引用地址发生错误，后面遇到再说。

3. es5 严格模式

es 版本的发展从 `es1.0` 一直到现在的是 `es7.0`，但是大多数浏览器都是基于 `es3.0` 的语法和 `es5.0` 的新增方法使用的，也就是说当 `es3.0` 和 `es5.0` 产生冲突的部分就是用 `es3.0` 的方法。今天我们来讲一个 `es5.0` 的严格模式，在这个模式下，二者产生冲突的部分就用 `es5.0` 的方法，否则使用 `es3.0` 的方法。

(1) `es5.0` 严格模式的启动

在逻辑的最顶端写上 `"use strict"`；（逻辑的最顶端就是前边可以有空格回车，但是不能有别的代码，语法规则，必须写在最顶端），你可以写在全局的最顶端，也可以写在局部的最顶端，写在全局就代表全局上的所有代码都启用 `es5.0` 的严格模式，写在局部就只让当前函数体里启用严格模式，比如：

```
function demo() {  
    console.log(arguments.callee);  
}  
demo();  
function test() {  
    "use strict";  
    console.log(arguments.callee);  
}
```



```

}
test();

```

arguments.callee 在 es5 里是不允许用的，此时第一个函数会执行，第二个函数就会报错 (TypeError)，因为第二个启用了 es5 的严格模式。

(2) 不再兼容 es3 的一些不规则语法，使用全新的 es5 规范。

(3) 两种用法：全局严格模式（写在全局最顶端）；局部函数内的严格模式（局部最顶端，推荐）

(4) 思考：为啥启用 es5 的严格模式要用字符串的启用方式，而不是函数调用？

答：不管是 es3 还是 es5 都必须让浏览器内核实现了 es 对应版本的标准前提下才能好使，那假如说老版本的浏览器没更新到 es5，你吭当一开始来个函数调用，他识别不了，就会立马报错，那么后边的代码都没办法执行了，你写成字符串，他就跟表达式一样，摆在那里也不影响啥，而且新版本的浏览器支持 es5 的话他就会识别并开启严格模式，起到了向后兼容的作用，即老版本保证不出错，新版本保证能识别。

4. es5 严格模式的语法

(1) 不允许用 with

with 之前我们没有讲，今天来讲一下：

```

var obj = {
  name: "obj",
  age: 234
}

var name = "window";
function test() {
  var age = 123;
  var name = "scope";
  with(obj) {
    console.log(name);
    console.log(age);
  }
}

test();

```

这个 with 的小括号里如果不放东西的话他就正常执行，如果小括号里放一个对象进去的话那就了不得了：他会把这个对象当做 with 所圈定的代码体的作用域链的最顶端，他会改变作用域链，会把 obj 当成自己的 A0 连到作用域链的最顶端，所以此时输出 obj 234。他会把 obj 当成自己最直接的 A0，反正 A0 也是一个对象，如果 obj 里没有他才会

去找 test 的 AO 再找 GO。

作用：简化代码，我们拿命名空间举个例子：

```
var org = {  
  dp1: {  
    jc: {  
      name: "abc",  
      age: 123  
    },  
    deng: {  
      name: "xiaodeng",  
      age: 234  
    }  
  },  
  dp2: {  
  
  }  
}
```

我们以前说命名空间用起来不太方便，其实他是这么用的，你想拿东西就直接：

```
with(org.dp1.jc){  
  console.log(name);  
}
```

这个 with 里放的就是我的代码，那么我的代码能访问的最直接的执行期上下文就是 org.dp1.jc，那么我在里边访问的 name 就是 abc，所以我在 with 里访问的话就直接写变量名就可以了，达到了代码简化的目的。

再比如 document 上有很多方法，我们再用的时候每次都得 document 点啥，太麻烦了，那我现在直接放到 with 里边：

```
with(document){  
  write("a");  
}
```

此时就输出 a，你就直接写 write 就行了，因为你写了之后他就会去对应的执行期上下文里边找，找到 document，而 document 正好有这个方法，就把 write 拿出来了。

但是，有一点不好就是 with 太强大了，你改啥不好你去改作用域链啊，作用域链是经过很复杂的过程才生成的，比如说作用域链有十层，你又新加了一层，就等于改了十一层，因为那十层得往下移啊，改的太深了，那就会达到效率的丧失，所以 es5 为了

提高效率就不准用 with 了。

```
"use strict";
with(document) {
    write("a");
}
```

此时就会报错，告诉你 with 不能使了。

(2) 不允许用 arguments.callee 和 function.caller，在启用 es5 的严格模式下，用这两个也报错，上边试过了。

(3) 变量赋值前必须声明

以前我们 es3 变量未经声明就使用叫做暗示全局变量，但是 es5 严格模式不行了：

```
"use strict";
a = 3;
```

此时就会报错，告诉你 a is not defined。

(4) 局部的 this 必须被赋值，赋什么就是什么。比如说：

```
"use strict";
function test() {
    console.log(this);
}
test();
```

我们说在 es3 里边，这个局部的 this 指向 window，但是在 es5.0 里边你必须给他赋值，不赋值就输出 undefined，你可以用 new 赋值，new test() 就输出 test {}，也可以 test.call({}) 就输出 {}。

注意：在 es3.0 里，假如说 test.call(123) 他会输出对象类型的 123：即 Number{123}，因为 es3 里规定 call 里边不能传原始值，你传了的话他就会调用包装类包装成对象，但是 es5 里就输出 123，赋什么就是什么。

(5) 拒绝重复属性和参数。

```
"use strict";
function test(name,name) {
    console.log(name);
}
test(123);
```

在 es3 里边这么写不报错，虽然没有人这么写，但是在 es5 里这么写就会报错，重复的属性也不行：

```
"use strict";
```

```
var obj = {
  name: "123",
  name: "234"
}
```

这么写也不行，但是有一点小的区别就是这么写不报错，这可能是浏览器在实现语法和规定语法的时候一个小误差了，但是后边的会覆盖掉前边的，也许现在不报错，可能以后就报错了，因为人家明令规定了不能有重复参数和属性。

(6) 不能用 eval

这个 eval es3 里都不能用，他也非常强大，他可以把字符串当做代码执行，比如：

```
var a = 123;
eval("console.log(a)");
```

此时就输出 123，这个 eval 是魔鬼，它还可以改变作用域，他还有自己的作用域，这里边很复杂，就不说了，反正不能用哈。

备注：以上内容才是笔面试的重点。

DOM 操作初探

1. **DOM: Document Object Model**，即文档对象模型。DOM 里边定义了一系列方法，是用来操作 html 和 xml 功能的一类对象的集合，也有人称 DOM 是 html 和 xml 的标准编程接口。

解释：DOM 提供的这一系列方法就是用来操作 html 和 xml 的，但是他操作不了 css。xml 是 html 的早期版本，除了可以自定义标签之外，基本和 html 没有区别，后来就被 js 里的 json 取代了，所以现在 xml 已经不用了。

注意：我们所说的改变不了 css 指的是改变不了 css 样式表，但是我们可以改变 HTML 的行间样式，也就是说我们可以通过间接地改变行间样式来改变他，比如说：

例 1：

```
<div></div>
<script type="text/javascript">
  var div = document.getElementsByTagName('div')[0];
  div.style.width = "100px";
  div.style.height = "100px";
  div.style.backgroundColor = "red";
</script>
```

效果：



你现在访问 div 就得到<div style="width: 100px; height: 100px; background-color: red;"></div>, 所以 DOM 虽然不能改变 css 样式表, 但是我们可以间接地通过改变 HTML 的行间样式来改变 css。

这节我们不讲知识点, 咱就通过几个例子来了解一下什么是 DOM, 之后咱们再慢慢了解他的语法。

解析: 例如上边的, 我们说 DOM 是操作 html 的编程接口, 但是在操作之前, 我们是需要先把他选中的, 选中标签有好几种方法, 其中有一种就叫 getElementByTagName, 就是 get 元素 ByTagName, TagName 就是标签名, 就可以通过标签名的方式把你这个标签给选出来, 里边添上 "div", 就是把所有的 div 都选出来, 选出来后封装成一个类数组, 所有的 div 会按照他的索引位进行排序, 那么, 我要选第一个 div, 就在后边加上 [0] 即可。现在这个 div 就被选中到 JavaScript 里可以被我们操作了, 你在控制台访问 div 就得到<div></div>, 选出来之后他就是 DOM 对象, 现在我们就可以给对象加属性了, 加个 .style 就代表样式列表, 然后再 .width 属性改成 100px, 那么他的行间样式就改了, 然后再加个 .height 等于 100px, 注意 js 里边不能写 -, 写 - 会出错, 所以 background-color 必须写成小驼峰式 backgroundColor。

例 2: 有了 DOM 之后, 我们就可以做一些动画效果, 现在把例 1 改一下:

```
<div></div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
    div.style.width = "100px";
    div.style.height = "100px";
    div.style.backgroundColor = "red";
    div.onclick = function () {
        this.style.backgroundColor = "green";
        this.style.width = "200px";
        this.style.height = "50px";
        this.style.borderRadius = "50%";
    }
</script>
```



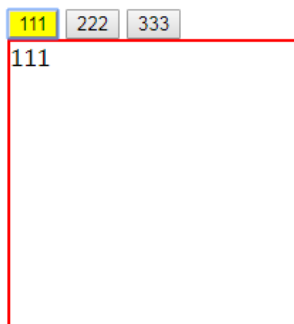
解析: onclick 就是当鼠标点击之后发生的变化, 我们给 div 的 onclick 时间绑定一个函数, 这个函数不需要你去调用, 当你鼠标点击之后他就会自动调用, 函数里设置了一些样式, div 调用 onclick, this 就是 div, 点击之后的效果如上图所示。

例 3：还是例 1，现在要求鼠标点击一次变绿，再点变红，再点变绿，再点变红……

```
<div></div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
    div.style.width = "100px";
    div.style.height = "100px";
    div.style.backgroundColor = "red";
    var count = 0;
    div.onclick = function () {
        count++;
        if(count % 2 == 1) {
            this.style.backgroundColor = "green";
        }else {
            this.style.backgroundColor = "red";
        }
    }
</script>
```

解析：onclick 是永久绑定的，那么就直接 var 一个计数器 count，当点击的时候让 count++，当 count 摩尔 2 余 1 时，让他变绿，否则就余 0，让他变红，所以说 DOM 还是用我们之前的思想，只不过在这个基础上加了些语法，来让我们操作 html。

例 4：我们来写一个选项卡，先看一下效果：



要求：点击哪个按钮的时候按钮就变黄，然后底下展示对应的内容。

css 代码：

```
<style>
    .content{
        display: none;
        width: 200px;
```



```

        height: 200px;
        border: 2px solid red;
    }
    .active {
        background-color: yellow;
    }
</style>

```

Html 和 js 代码:

```

<div class="wrapper">
    <button class="active">111</button>
    <button>222</button>
    <button>333</button>
    <div class="content" style="display:block">111</div>
    <div class="content">邓哥..222</div>
    <div class="content">333</div>
</div>
<script type="text/javascript">
    var btn = document.getElementsByTagName("button");
    var div = document.getElementsByClassName("content");
    for (var i = 0; i < btn.length; i++) {
        (function (n) {
            btn[n].onclick = function () {
                for (var j = 0; j < btn.length; j++) {
                    btn[j].className = "";
                    div[j].style.display = "none";
                }
                this.className = "active";
                div[n].style.display = "block";
            }
        })(i)
    }
</script>

```

解析: 先写三个 button 标签 (button 标签就是一个按钮形状, 其他没有什么) 和三个 div 标签, 然后对应的给 div 加上 css 样式并隐藏, 给第一个 div 通过行间样式把他显

示出来，定义一个.active 的样式，把背景颜色设为黄色，并给第一个 button 标签加上 class 名 active。现在开始写 js，先 var 一个 btn，用标签名的选择方式把三个 button 选中，没有指定第几位这个 btn 里就装了三个 button，同样，var 一个 div，用 class 名把三个 div 都选中，然后用 for 循环把 btn 的每一位都遍历一遍，给每一位都绑定一个 onclick 事件，函数里再用 for 循环把 btn 的每一位的 class 名清空，清空后那个 active 样式就作用不上了，再把 div 的每一位的样式改成让他隐藏，循环外边把此时点击的这个 button 元素加上 class 名 active，再把 div 对应的索引位给他显示出来，注意这里不能写 this，因为是 btn 调用的，所以我最先写的是 div[i]，但是这么写的话里边的 i 现在变不了现，因为他在函数体里，等循环结束后 i 已经变成 3 了，就和我们之前讲的闭包一样的原理，所以我在外边加上一个立即执行函数，形参传 n，实参传 i，里边也用 n，问题就解决了。

例 5：写一个方块加速运动，并让他在特定位置停止运动。

```
<script type="text/javascript">
    var div = document.createElement("div");
    document.body.appendChild(div);
    div.style.width = "100px";
    div.style.height = "100px";
    div.style.backgroundColor = "red";
    div.style.position = "absolute";
    div.style.left = "0";
    div.style.top = "0";
    var speed = 1;
    var timer = setInterval(function () {
        speed += speed/20;
        div.style.left = parseInt(div.style.left) + speed + "px";
        div.style.top = parseInt(div.style.top) + speed + "px";
        if(parseInt(div.style.left) > 200 && parseInt(div.style.top) >
        200) {
            clearInterval(timer);
        }
    }, 10)
</script>
```

解析：动态效果无法展示，就是一直斜着往右下移动然后停止，我这里边没有用 HTML 标签，全用 js 写的，第一行是用 js 创建一个 div，然后第二行是把他加到 body 标签

里,这时,body 里就有 div 了,然后给他加一些行间样式并让他定位,然后 setInterval 是定时器的意思,方法后的 10 代表每 10 毫秒调用一次方法,然后函数里增加他的 left 和 top 值,用 parseInt 把之前的值取出来,因为之前是字符串值,取出来变成数字的加上 speed,最后用 if 语句当他的 top 值和 left 值大于 200 时就停止运动。

例 6: 用键盘控制小方块运动, 并且实现点击加速后移动的更快。

```
<button style="width:100px;height:50px;background:linear-gradient(to
left,#999,#000,#432,#fcc);position:fixed;right:0;top:50%;text-align:center
;line-height:50px;color:#fff;font-size:25px;font-family:arial;"> 加 速
</button>
```

```
<script type="text/javascript">
    var btn = document.getElementsByTagName("button")[0];
    var div = document.createElement("div");
    document.body.appendChild(div);
    div.style.width = "100px";
    div.style.height = "100px";
    div.style.backgroundColor = "red";
    div.style.position = "absolute";
    div.style.left = "0";
    div.style.top = "0";
    var speed = 5;
    btn.onclick = function () {
        speed++;
    }
    document.onkeydown = function (e) {
        switch (e.which) {
            case 38:
                div.style.top = parseInt(div.style.top) - speed + "px";
                break;
            case 40:
                div.style.top = parseInt(div.style.top) + speed + "px";
                break;
            case 37:
                div.style.left = parseInt(div.style.left) - speed + "px";
                break;
```

```

        case 39:
            div.style.left = parseInt(div.style.left) + speed + "px";
            break;
    }
}
</script>

```

加速

解析：这个 button 的按钮有一些样式是用 css3 做的，是不是很炫酷？我们 var 一个 speed 等于 5，然后当你点击加速按钮后，让 speed++ 并参与后边的计算，就实现了加速运动，这个 onkeydown 是键盘输入时的事件，里边传一个形参 e，我们每次按下一个按钮后，这个 e.which 都会显示不同的数，所以再写代码的时候，我们在控制台 console.log(e.which) 捕捉到对应的数字，例子里边的 38 40 37 39 分别是键盘上下左右四个方向键的 which 值，然后当你按下后，调整对应的值即可。

按住加速思路：就是这一次按下与下一次按下的速度间隔时间十分短暂，就认为他加速了。每一次按下的时候都记录一个新的时间片段，都减去上一个执行的时间片段，如果时间片段都小于一定的毫秒数的话，我们让一个计数器去++，当连续小于的时候，就让计数器连续++，如果++到一定数的时候，我们认为是连续按了，再按就加速了，让每一次按的时候都判断一下，如果时间间隔过大的话，就让计数器重新归 0。这个在 js 运动课里面讲。

例 7：画一个棋盘，鼠标经过时可以在上边画图。

css 代码：

```

<style>
    *{
        margin: 0;
        padding: 0;
    }
    li{
        box-sizing: border-box;
        float: left;
        width: 10px;
        height: 10px;
        border: 1px solid black;
    }

```

```
ul{
    list-style:none;
    width: 200px;
    height: 200px;
}
```

</style>

Html 和 js 代码:

```
<ul>
```

```
  <li img-date="0"></li>
```

```
  // (……这里有 400 个这样的 li)
```

```
</ul>
```

```
<script type="text/javascript">
```

```
  var ul = document.getElementsByTagName('ul')[0];
```

```
  ul.onmouseover = function (e) {
```

```
    var event = e || window.event;
```

```
    var target = event.target || event.srcElement;
```

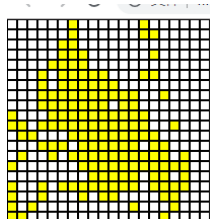
```
    target.style.backgroundColor="rgb(255,255,"+
```

```
    target.getAttribute('img-date')+")";
```

```
    target.setAttribute('img-date',parseInt(target.getAttribute('img-date'))+6);
```

```
  }
```

```
</script>
```



解析：效果就是当你鼠标经过的时候可以在上边画图。

备注：以上所有的实例都是先看一下 DOM 能实现哪些效果，真正的学习语法从下一讲开始。

附录：html 快捷生成标签方法（输入后按 tab）

(1) div*3 直接创建 3 个 div

(2) div.demo 就是直接设置了 class 名字

- (3) `div#only` 直接设置了 id 名字
- (4) `div>p` 就是让 div 下面包含一个子元素 p
- (5) `p[style='background-color:red;width:100px;height:100px;']` 方括号直接生成对应的属性
- (6) `div>(p+span)` div 下边包裹了 span 和 p
- (7) `div{123}` 大括号直接加内容
- (8) `ul>li{$}*10` 这里的 \$ 代表变量 \$ 可以按照 1, 2, 3 排序, 内容、class、id 都能用 \$

DOM 选择器、节点类型和属性

1. DOM 选择器（方法类操作）

(1) `document` 代表整个文档

解释：首先，`document` 是一个对象，这个对象上边有一些属性和方法，单独的 `document` 就代表的是整个文档在 js 里的显示形式，我们现在所说的整个文档最顶级的标签看上去好像是 `html` 标签，但是如果在 `html` 标签外边再套一个标签的话，这个标签就是 `document`，就是说 `document` 才是真正的代表整个文档，`html` 只是他下面的一个根标签。

(2) id 选择器

这个选择器和 `css` 里边讲的极其类似，比如说你在 `html` 里边写上一个 `<div id="only">123</div>`，然后在 js 里 `var div = document.getElementById("only")`，此时就通过 id 名选中了这个 `div` 元素，由于 id 和元素是一一对应的，所以 `Element` 后边不用加 `s`，现在你访问 `div` 就得到 `<div id="only">123</div>`。

拓展：元素 id 在 ie8 以下的浏览器，不区分 id 大小写，而且也返回匹配 `name` 属性的元素，就是在老版本的 ie 里，假如你不写 id，写个 `name` 等于 `only`，也能通过这种方式选出来，但是新版本不可以了。

注意：id 选择器不能太依赖，一般用于顶级框架，因为前后端一链接就会把你的 id 改掉，所以在 `css` 里最好用 `class`。

(3) 标签选择器

比如说你在 `HTML` 里边写一个 `<div>123</div>`，然后在 js 里 `var div = document.getElementsByTagName("div")`，此时就把页面里所有的 `div` 都选中了，封装成了一个类数组（DOM 里几乎所有的成组的都是类数组），现在你访问 `div` 就得到 `HTMLCollection[div]`，即使里边只有一个，也是一组，所以你要选中一个的话，就 `var div = document.getElementsByTagName("div")[0]`，此时再访问 `div` 就是 `<div>123</div>`。

备注：`document.getElementsByTagName("*")`，里边可以写 `*`，和通配符一样，选中了

所有标签。

(4) class 选择器

他和这个标签选择器差不多，选出来也是一组，你先`<div class="demo">123</div>`，然后 `var div = document.getElementsByClassName("demo")[0]`，此时访问 `div` 就得到`<div class="demo">123</div>`。

注意：这个 class 选择器在 ie8 和 ie8 以下的浏览器中没有，但是新版本是可以的，所以在 js 里，class 选择器并没有标签选择器那么常用，标签选择器 `getElementsByTagName` 在任意一个浏览器里都好使。

(5) name 选择器

比如`<input name="abc">`，然后 `var input = document.getElementsByName("abc")[0]`，访问 `input` 就得到`<input name="abc">`。需注意这个 name 属性只对部分元素生效，如表单、表单元素、img 等，name 选择器很少用。

(6) css 选择器

这个 css 选择器能让我们在 js 里选择元素的时候和 css 里一样灵活，比如说：

```
<div>
  <strong></strong>
</div>
<div>
  <span>
    <strong></strong>
  </span>
</div>
```

然后在 js 里写上 `var strong = document.querySelector("div span strong");` 括号里写的东西就和 css 里选择标签的方法是一样的，此时访问 `strong` 就得到 ``，这选的是一个，还有一个 `var strong = document.querySelectorAll("div span strong");`，他选出来是一组，你再访问 `strong` 就得到 `NodeList[strong]`，然而 css 选择器并不用，他有两个问题，第一个问题就是在 ie7 及以下的版本没有，这个对我们并没有什么影响，还有一个致命的问题，就是这个 `querySelector` 和 `querySelectorAll` 选出来的东西不是实时的，什么是实时的？比如：

```
<div></div>
<div></div>
<div></div>
<script type="text/javascript">
```

```

var div = document.getElementsByTagName("div");
var newDiv = document.createElement("div");//意思是一个新的 div
document.body.appendChild(newDiv);//意思是把这个新的 div 扔到页面里
</script>

```

此时访问 `div` 就得到 `HTMLCollection(4) [div, div, div, div]`，可见选中了四个 `div`，我在 `js` 里边是先选的后加的新的，但是也可以被访问到，这就是实时的。但是 `css` 选择器就不行了，我把代码改成：

```

<div></div>
<div></div>
<div></div>
<script type="text/javascript">
    var div = document.querySelectorAll("div");
    var newDiv = document.createElement("div");
    document.body.appendChild(newDiv);
</script>

```

此时再访问 `div` 就得到 `NodeList(3) [div, div, div]`，他选出来的是一个副本，选出来几个就是几个，后边你再怎么改都跟他没关系了，所以他不是实时的，在用法上就及其受局限，除非极特殊情况，你就想选他的副本保存起来才会用这个，否则的话我们不用。

2. 遍历节点树（非方法类操作）

(1) `parentNode`：父节点 比如说：

```

<div>
    <span></span>
    <strong></strong>
    <em></em>
</div>
<script type="text/javascript">
    var strong = document.getElementsByTagName("strong")[0];
</script>

```

现在访问 `strong.parentNode` 就是他的父节点，得到 `<div>...</div>`，继续 `strong.parentNode.parentNode` 就是 `div` 的父亲，即 `<body>...</body>`，继续 `strong.parentNode.parentNode.parentNode` 就是 `body` 的父级，即 `<html>...</html>`，再 `strong.parentNode.parentNode.parentNode.parentNode` 就是 `HTML` 的父级，即 `#document`，`strong.parentNode.parentNode.parentNode.parentNode.parentNode` 就

得到 null，这就说明 document 就到顶层了，他是顶层的父级节点，代表整个文档。

(2) childNodes 子节点们

我们说一个元素只有一个父节点，但是他的子节点就不止一个了，比如说上边的代码，我 `var div = document.getElementsByTagName("div")[0];` 然后 `div.childNodes` 他肯定是一组，然后 `div.childNodes.length` 得到 7，为什么是 7 而不是 3？我们 `childNodes` 遍历的是节点树，并不是只有元素节点算节点，上边有三个元素节点和四个文本节点（回车文本），节点的类型有很多，它包括文本节点、元素节点、属性节点、注释节点等。再比如：

```
<div>
  <!-- this is comment! -->
  <strong></strong>
  <span></span>
</div>
<script type="text/javascript">
  var div = document.getElementsByTagName("div")[0];
</script>
```

现在访问 `div.childNodes` 就得到 `odeList(7)[text, comment, text, strong, text, span, text]`，可见他有七个节点：依次是文本节点、注释节点、文本节点、元素节点、文本节点、元素节点、文本节点，其实注释系统可以看到，只是不运行而已。现在继续改：

```
<div>
  123
  <!-- this is comment! -->
  <strong></strong>
  <span></span>
</div>
```

现在 `div` 的子节点还是 7 个，空格文本和文本和回车文本都写在一起，就是一个文本节点。

(3) firstChild/lastChild 第一个子节点/最后一个子节点

例如上边的，现在 `div.firstChild` 就是文本 123 和回车，`div.lastChild` 就是 `#text`，还是文本节点，只不过是空的，他就那样显示了。

(4) nextSibling/previousSibling 后一个兄弟节点/前一个兄弟节点

还是上边的，`var strong = document.getElementsByTagName("strong")[0];`，然后 `strong.nextSibling` 就得到 `#text`，然后 `strong.nextSibling.nextSibling` 就得到

``，继续 `strong.nextSibling.nextSibling.nextSibling` 就得到 `#text`，`strong.nextSibling.nextSibling.nextSibling.nextSibling` 就是 `null`。再比如 `strong.nextSibling.previousSibling` 就得到 ``。下一个的前一个就是自己。

3. 遍历元素节点树

这几种方法只遍历的是元素节点，其他的都不掺杂了。比如说：

```
<div>
  123
  <!-- this is comment! -->
  <strong></strong>
  <span></span>
</div>
```

(1) `parentElement` 元素父节点

在 js 里 `var div = document.getElementsByTagName("div")[0]`；然后 `div.parentElement` 他的元素父节点就是 `<body>...</body>`，然后 `div.parentElement.parentElement`，就是 `body` 的元素父节点，即 `<html>...</html>`，在 `div.parentElement.parentElement.parentElement` 就是 `null`，因为 `document` 不叫元素节点，他自称为一个节点，所以 `parentNode` 和 `parentElement` 的区别就在于有没有 `document`。

(2) `children` 元素子节点

现在 `div.children` 就得到 `HTMLCollection(2)[strong, span]`，只有两个。

(3) `childElementCount === children.length` 元素子节点的个数

比如说 `div.childElementCount` 就得到 2，你 `div.children.length` 也是 2，所以记第二个就行了，第二个好记一点。

(4) `firstElementChild/lastElementChild` 第一个元素子节点/最后一个元素子节点

现在 `div.firstElementChild` 就是 ``，`div.lastElementChild` 就是 ``。

(5) `nextElementSibling/previousElementSibling` 后一个兄弟元素节点/前一个兄弟元素节点

现在把代码改一下：

```
<div>
  123
  <!-- this is comment! -->
```

```

    <strong></strong>
    <span></span>
    <em></em>
    <i></i>
    <b></b>
</div>
<script type="text/javascript">
    var strong = document.getElementsByTagName("strong")[0];
</script>

```

现在 `strong.nextElementSibling` 就得到 ``，继续 `strong.nextElementSibling.nextElementSibling` 就得到 ``，再 `strong.nextElementSibling.nextElementSibling.previousElementSibling` 就得到 ``

备注：以上遍历节点树的方法所有浏览器都兼容，但是遍历元素节点树的方法除 `children` 以外，ie9 及以下的浏览器都不兼容。

4. 节点的四个属性

(1) nodeName 元素的标签名，只读

例如上边的代码段，现在我把 `div` 选出来 `var div = document.getElementsByTagName("div")[0]`；你访问 `document.nodeName` 就得到 `"#document"`，访问 `div.firstChild.nodeName`，他是一个文本节点，就得到 `"#text"`，继续 `div.childNodes[1].nodeName`，第二个子节点是注释，就得到 `"#comment"`，`div.childNodes[3].nodeName` 是元素节点，就得到 `"STRONG"`。返回的是一个字符串，只读就是只能读取不能写入，比如说我把 `div.childNodes[3].nodeName = "abc"`，再访问的话还是 `"STRONG"`。

(2) nodeValue text 节点或 comment 节点的文本内容，可读写

这个属性只有文本节点和注释节点才有，访问 `div.childNodes[0].nodeValue` 就得到文本的回车和 123，这个可以改，你 `div.childNodes[0].nodeValue = "234"`，再访问的话就是 `"234"`，再比如访问 `div.childNodes[1].nodeValue` 就得到 `"this is comment!"`，如果 `div.childNodes[1].nodeValue = "that is comment!"`，再访问 `div.childNodes[1]` 就是 `<!-- that is comment! -->`。

(3).nodeType 该节点的类型，只读

比如说现在有一个节点，我也不知道里边是什么节点，就有他来分辨，每一个节点都有这个 `nodeType` 属性，你调用他返回的是一个具体的值：元素节点是 1，属性节点是 2，文本节点是 3，注释节点是 8，document 节点是 9，DocumentFragment（文档碎片

节点) 是 11. 现在 `document.nodeType` 就是 9, `div.childNodes[1].nodeType` 第二个是注释节点就返回 8, `div.childNodes[0].nodeType` 文本节点返回 3, `div.childNodes[3].nodeType` 元素节点就是 1.

练习: 还是上边的例子, 现在封装一个方法, 要求返回 `div` 的直接子元素节点, 但是不允许用 `children`.

```
function retElementChild(node) {
    var temp = {
        length: 0,
        push: Array.prototype.push,
        splice: Array.prototype.splice
    },
        child = node.childNodes,
        len = child.length;
    for (var i = 0; i < len; i++) {
        if (child[i].nodeType === 1) {
            temp.push(child[i]);
        }
    }
    return temp;
}
```

解析: 不允许用 `children`, 我们就用 `childNodes`, 把他的所有子节点都遍历一遍, 然后当他的 `nodeType` 为 1 的时候就是元素节点, 就 `push` 给 `temp` 并返回 `temp`, 咱们加个 `splice` 方法返回的类数组就长成数组那样了, 现在 `retElementChild(div)` 就得到 `[strong, span, em, i, b]`。

(4) attributes 元素节点的属性集合

这个属性就是访问元素的属性节点的, 上边的代码现在不看了, 只留一个 `div`:

```
<div id="only" class="demo"></div>
```

然后继续把 `div` 选出来: `var div = document.getElementsByTagName("div")[0]`; 现在 `div.attributes` 就得到 `NamedNodeMap` {0: id, 1: class, id: id, class: class, length: 2}, 访问 `div.attributes[0]` 就得到 `id="only"`, 访问 `div.attributes[1]` 就是 `class="demo"`. `div.attributes[1].nodeType` 就得到 2.

你也可以把他的属性名和属性值都取出来, `div.attributes[0].name` 就得到 `"id"`, `div.attributes[0].value` 就得到 `"only"`. 属性名是只读的, 属性值可读可写.

5. 节点的一个方法 `Node.hasChildNodes()`

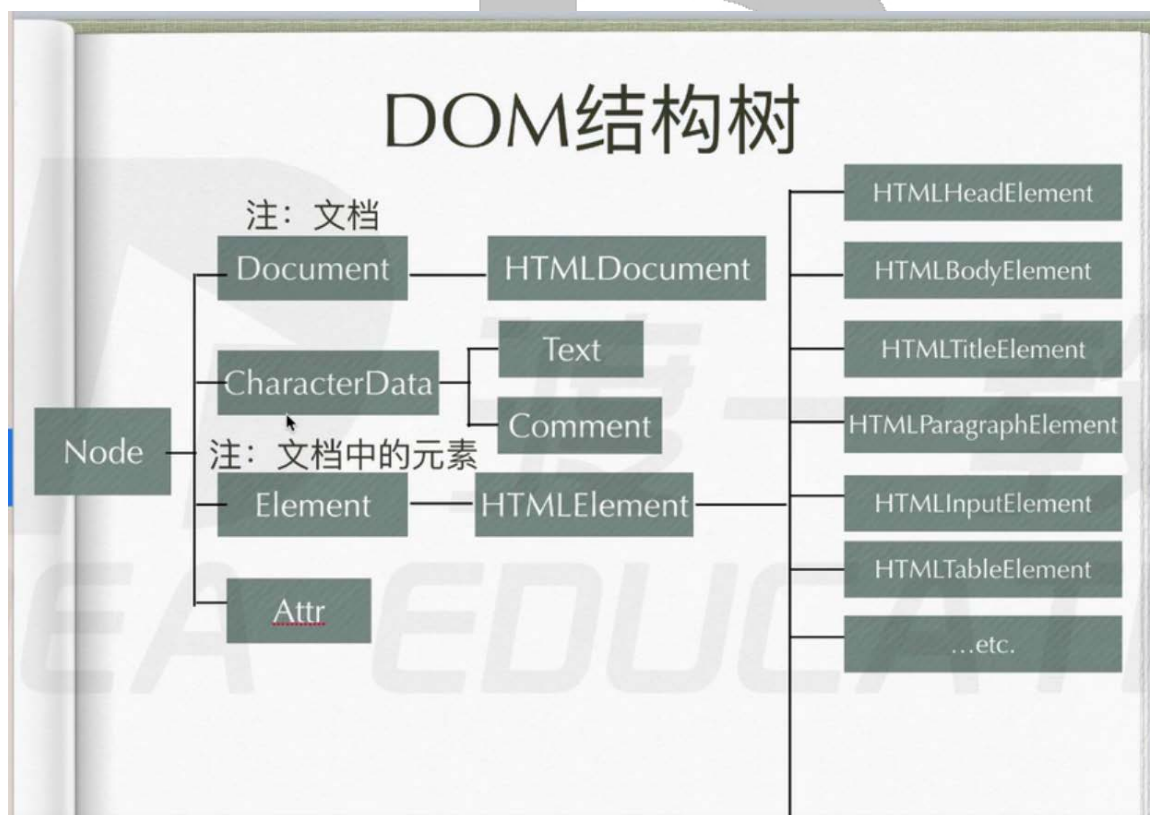

```
<div id="only" class="demo">
```

```
</div>
```

这个方法就是判断这个元素有没有子节点，现在 `div.hasChildNodes()` 就返回 `true`，因为空格回车文本也算文本节点，除非 `<div id="only" class="demo"></div>` 的话，访问 `div.hasChildNodes()` 是 `false`，中间啥都没有（属性节点不算，他是 `div` 自己的，不能算子节点）才能返回 `false`。

DOM 继承树及基本操作

1. DOM 结构树：如图



这就是 DOM 的结构树，咱们一个一个看：

（1）首先我们知道 `document` 代表的是整个文档，访问就得到 `#document`，但是我访问 `Document` 就得到 `f Document() { [native code] }`，是一个函数，而且首字母大写了，说明他是一个构造函数，但是这个构造函数有点特殊，就是你不能自己 `new Document()`，这是系统留给自己 `new` 的，你不能 `new`。然后假如说你给 `Document` 的原型上加一些属性的话，比如 `Document.prototype.abc = "abc"`，那么 `document` 就能继承他的属性，你访问 `document.abc` 就得到 `"abc"`，上边的结构树就表示这一系列继承的关系。

其实，`document` 这个对象的构造函数是 `HTMLDocument`，然后 `HTMLDocument.prototype`

就是一个对象，里边的__proto__属性就是 Document.prototype。也就是说 document 的原型是 HTMLDocument.prototype，然后 HTMLDocument.prototype 的原型是 Document.prototype，他们形成了一个原型链。所以刚才我们在 Document.prototype 上加了一个属性 abc，就能通过原型链让 document 继承到。那比如说我们在 HTMLDocument.prototype.bcd = "123"，然后访问 document.bcd 也得到"123"。就能继承到。再比如 Document.prototype.abc = "1"，然后 HTMLDocument.prototype.abc = "2"，再访问 document.abc 就是"2"，因为继承是可近来的。

其实上图中的 Document 底下有两个分支，一个是 HTMLDocument，一个是 XMLDocument，然后 html 里的 document 的方法都继承自 HTMLDocument，xml 里的 document 方法都继承自 XMLDocument，Document 里则定义了一系列 html 和 xml 都能用的方法，只不过 xml 现在不用了，图上就没有显示出来。然后看图上 Document 上边还有一个 Node，他也是一个构造函数，位于所有的最顶端。

(2) 第二个 CharacterData 下边有 Text 和 Comment，这就说明文本节点能用的属性和方法全部继承自 Text.prototype，注释节点的方法则全部继承自 Comment.prototype，然后一层一层向上继承。

(3) 第三个最长，Element 下边有一个 HTMLElement(那下边就肯定还有一个 XMLElement，只不过没写)，然后在下边有一堆东西，最后还没写完，但是你会发现那一堆东西都是一些标签能用的一些属性和方法，比如说：

```
HTMLHeadElement.prototype.abc = "demo";
```

```
var body = document.getElementsByTagName("body")[0];
```

```
var head = document.getElementsByTagName("head")[0];
```

然后你访问 head.abc 就得到"demo"，但是你访问 body.abc 就是 undefined。说明 HTMLHeadElement.prototype 里边定义的方法只能让 head 标签用，其他的也是一样的，但是 HTMLElement 定义的方法是所有标签都能用的。

(4) 现在 document.__proto__ 得到 HTMLDocument{...}，

document.__proto__.__proto__ 就得到 Document{...}，

document.__proto__.__proto__.__proto__ 得到 Node{...}，

document.__proto__.__proto__.__proto__.__proto__ 得到 EventTarget{...}

document.__proto__.__proto__.__proto__.__proto__.__proto__ 就得到 Object{...}。

这就说明 DOM 对象最终也继承自 Object.prototype，比如你访问 document.toString() 就得到"[object HTMLDocument]"，这个 Object.prototype 是所有对象原型链上的终端。

2. DOM 结构树的应用

(1) getElementById 方法定义在 Document.prototype 上，即 Element 节点上不能

使用。

(2) `getElementsByName` 方法定义在 `HTMLDocument.prototype` 上, 即非 `html` 中的 `document` 不能使用 (`xml` 的 `document` 和 `Element` 不能用)。

(3) `getElementsByTagName` 方法定义在 `Document.prototype` 和 `Element.prototype` 上。

解释: 这个方法在两个地方都定义了, 就都能用, 比如说:

```
<div>
  <span>1</span>
</div>
<span></span>
<script type="text/javascript">
  var div = document.getElementsByTagName('div')[0];
  var span = div.getElementsByTagName("span")[0];
</script>
```

现在我要选 `div` 下边的 `span` 就可以这么写, 因为人家说 `getElementsByTagName` 方法在 `Element.prototype` 里也定义了, 所以 `div` 就可以使用这个方法, 选 `div` 的时候选的是整个文档下边的 `div` 的第一个, 但是选 `span` 的时候就选的是这个 `div` 下的 `span`, 现在访问 `span` 就是 `1`。

(4) `HTMLDocument.prototype` 定义了一些常用的属性, `body`, `head`, 分别指代 `HTML` 文档中的 `<body>` 和 `<head>` 标签。

解释: 就是人家都给你定义好了, 你以后要选 `head` 标签或者 `body` 标签的话就直接 `document.head` 或者 `document.body` 就可以了。

(5) `Document.prototype` 上定义了 `documentElement` 属性, 指代文档的根元素, 在 `HTML` 文档中, 他总是指代 `<html>` 元素。

解释: 现在你选 `html` 标签就直接 `document.documentElement` 就行了。

(6) `getElementsByClassName`、`querySelectorAll`、`querySelector` 这几个方法在 `Document.prototype` 和 `Element.prototype` 类中均有定义。

解释: 和上边一样, 两边都能使用, 比如说 `div.getElementsByClassName()` 也可以用。

练习 1: 遍历元素节点树 (在原型链上编程)

```
<body>
  <div>
    <span>
      <em></em>
      <strong>
```

```

        <em>
            <a href=""></a>
        </em>
    </strong>
</span>
<p></p>
</div>
<span></span>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
    Element.prototype.retChildElements = function () {
        var child = this.childNodes,
            len = child.length;
        for (var i = 0; i < len; i++) {
            if (child[i].nodeType === 1) {
                console.log(child[i]);
            }
        }
    }
</script>
</body>

```

```
> div.retChildElements()
```

```

▼ <span>
  <em></em>
  ▼ <strong>
    ▼ <em>
      <a href=""></a>
    </em>
  </strong>
</span>
<p></p>
< undefined

```

解析：先把他的子节点们全部遍历一遍，然后当他的子节点的 nodeType 为 1 的时候就是元素节点，然后把它打印出来即可。

练习 2：封装函数，返回元素 a 的第 n 层祖先元素节点。

```

<body>
    <div>
        <span>
            <strong>

```

```

        <em>
            <a href=""></a>
        </em>
    </strong>
</span>
</div>
<script type="text/javascript">
    var a = document.getElementsByTagName('a')[0];
    function retParent(elem, n) {
        while (elem && n) {
            elem = elem.parentElement;
            n--;
        }
        return elem;
    }
</script>
</body>

```

解析：这个题其实非常简单，咱们用 while 循环让 elem 每次都等于他的父元素节点，然后用 n 控制循环圈数，每次让 n--，当 n 等于 0 的时候循环结束返回 elem 即可。现在比如说把 a 选出来，然后 retParent(a, 2) 就得到 ...，retParent(a, 3) 就得到 ...，但是有一个问题就是比如说 retParent(a, 10) 就会报错，我们就在 while 里边容个错，条件写成 elem && n，那么当 elem 等于 null 的时候就走不进去，直接返回 elem 即可，你再访问 retParent(a, 10) 就得到 null。

练习 3：编辑函数，封装 myChildren 功能，解决以前部分浏览器的兼容性问题。

```

<body>
    <div>
        <b></b>
        abc
        <!-- this is comment -->
        <strong>
            <span>
                <i></i>
            </span>

```

```

    </strong>
</div>
<script type="text/javascript">
    Element.prototype.myChildren = function () {
        var child = this.childNodes;
        var len = child.length;
        var arr = [];
        for (var i = 0; i < len; i++) {
            if (child[i].nodeType == 1) {
                arr.push(child[i]);
            }
        }
        return arr;
    }
    var div = document.getElementsByTagName("div")[0];
</script>

```

</body>

解析：children 方法其实兼容性是很好的，只不过 ie4 和 ie5 没有，这个题的意思就是自己封装一个 children 方法，但是不能用 children 属性，咱们就用 childNodes 把子节点们全部遍历一遍，然后当他的 nodeType 等于 1 的时候就 push 给 arr 并返回 arr 即可。现在把 div 选出来，然后 div.myChildren() 就得到 [b, strong]。

练习 4：自己封装 hasChildren() 方法，不可用 children 属性。

<body>

```

<div>
    <b></b>
    abc
    <!-- this is comment -->
    <strong>
        <span>
            <i></i>
        </span>
    </strong>
</div>
<script type="text/javascript">

```



```

Element.prototype.hasChildren = function () {
    var child = this.childNodes;
    var len = child.length;
    for (var i = 0; i < len; i++) {
        if (child[i].nodeType == 1) {
            return true;
        }
    }
    return false;
}
var div = document.getElementsByTagName("div")[0];
</script>

```

</body>

解析：这个和上边的思路是一样的，先遍历一遍，然后仿照 hasChildNodes() 方法，如果有的话返回 true，没有的话循环结束后返回 false 即可。现在 div.hasChildren() 就是 true。

练习 5：封装函数，返回元素 e 的第 n 个兄弟元素节点，n 为正，返回后面的兄弟元素节点，n 为负，返回前面的，n 为 0，返回自己。

<body>

<div>

<p></p>

<!-- this is comment -->

<i></i>

<address></address>

</div>

<script type="text/javascript">

```
function retSibling(e, n) {
```

```
    while (e && n) {
```

```
        if (n > 0) {
```

```
            if (e.nextElementSibling) {
```

```
                e = e.nextElementSibling;
```

```
            } else {
```

```

        for (e = e.nextSibling; e && e.nodeType != 1; e =
            e.nextSibling);
    }
    n--;
} else {
    if (e.previousElementSibling) {
        e = e.previousElementSibling;
    } else {
        for (e = e.previousSibling; e && e.nodeType != 1; e
            = e.previousSibling);
    }
    n++;
}
}
return e;
}
var strong = document.getElementsByTagName("strong")[0];
</script>
</body>

```

解析：和练习 2 一样，写一个 while 循环，用 n 控制循环圈数，并且 e 得有意义，然后循环里边当 n 大于 0 的时候每一圈都让 e 等于他的下一个兄弟元素节点，当 n 小于 0 的时候每一圈都让 e 等于他的前一个兄弟元素节点，最后返回 e 即可，如果 n 等于 0，就直接返回 e，但是这么写有一个兼容性问题，因为老版本的 ie 没有 nextElementSibling 和 previousElementSibling，那么，我们就得把程序写全，拿 n 为正举例：当 e.nextElementSibling 有意义时就走里边的，没有意义的话，我们在里边写个 for 循环，先让 e 等于他下一个兄弟节点，然后判断 e 的 nodeType 是不是不等于 1，如果不等于 1，就继续让 e 等于他的下一个兄弟节点，直到等于 1 的时候循环结束（我们以前讲过 for 循环的执行顺序，这里边的大括号里边没有内容的话就可以不写，如果第二句条件成立的话，他会执行第三条语句，再判断第二条语句是否成立，一直循环，直到 e 的 nodeType 等于 1，循环结束）。但是这么写还有一个问题，比如说我们假设用的是老版本的 ie，retSibling(strong, 3) 就会报错，因为第二圈的时候 e 就是 address 标签了，然后第三圈执行 for 第一句后 e 是文本节点，文本节点的 nodeType 不是 1，继续后边的 e 变成了 null，然后判断，null 没有 nodeType 所以报错，故此我们在判断的时候必须加上条件 e，如果 e 等于 null，循环结束。下边的也是一样的，

现在 `retSibling(strong,1)` 就是 `<i></i>`，`retSibling(strong,2)` 就是 `<address></address>`，`retSibling(strong,3)` 是 `null`，`retSibling(strong,4)` 也是 `null`，`retSibling(strong,-1)` 就是 `<p></p>`，`retSibling(strong,-2)` 就是 ``，`retSibling(strong,-3)` 和 `retSibling(strong,-4)` 都是 `null`。

3. DOM 基本操作

上边讲的所有方法都是查看操作，下边讲其他的几个操作。

第一种：增

(1) 创建元素节点（即创建标签）：`document.createElement()`

比如说现在 `var div = document.createElement("div")`，你在访问 `div` 就得到 `<div></div>`，括号里写什么字符串，就能创建出什么标签，但是现在增加的这个标签还在 `div` 这个变量里边，还没有插入到页面里。插入下边讲。

(2) 创建文本节点：`document.createTextNode()`

比如 `var text = document.createTextNode("邓宝宝")`，你在访问 `text` 就是“邓宝宝”，同样也可以插入。

(3) 创建注释节点：`document.createComment()`

比如说 `var comment = document.createComment("this is comment")`，访问 `comment` 就得到 `<!--this is comment-->`。

(4) 创建文档碎片节点：`document.createDocumentFragment()`（后期会讲到）

注意：创建节点的方法你里边必须要添加东西的，不然创建的是空的，没有内容。

第二种：插

(1) `appendChild()`

每个元素都有 `appendChild` 方法，这个方法就跟 `push` 方法一样，就是在最后插入东西的，比如说

`<body>`

`<div></div>`

`<script type="text/javascript">`

`var div = document.getElementsByTagName("div")[0];`

`var text = document.createTextNode("邓宝宝");`

`var span = document.createElement("span");`

`div.appendChild(text);`

`div.appendChild(span);`

`var text1 = document.createTextNode("demo");`

`span.appendChild(text1);`

`</script>`

</body>

现在写了一个 div 并把它选出来了，然后创建了一个文本节点邓宝宝和标签 span，并把他们插入到 div 里，那么这个 span 标签就在邓宝宝后边，再创建一个文本节点 demo 并插入到标签 span 里边，那么现在就是：

```
<body> -- 20
  <div>
    "邓宝宝"
    <span>demo</span>
  </div>
```

再比如说：

<body>

<div></div>

<script type="text/javascript">

var div = document.getElementsByTagName("div")[0];

var text = document.createTextNode("邓宝宝");

var span = document.createElement("span");

div.appendChild(text);

div.appendChild(span);

var text1 = document.createTextNode("demo");

span.appendChild(text1);

span.appendChild(text);

</script>

</body>

我最后把邓宝宝插入到 span 里，再来看效果：

```
<body>
  <div> == $0
    <span>
      "demo"
      "邓宝宝"
    </span>
  </div>
```

刚才 div 里边的邓宝宝就没有了，这就说明 appendChild 进行的是剪切操作，把一个地方的东西剪切到另一个地方。再看一个直接的：

<body>

<div></div>

<script type="text/javascript">

var div = document.getElementsByTagName("div")[0];

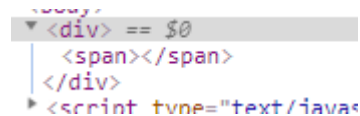
var span = document.getElementsByTagName("span")[0];

div.appendChild(span);

```

</script>
</body>

```



```

<div> == $0
├── <span></span>
└── <script type="text/javascript">

```

原来 div 和 span 是兄弟结构的，现在把 span 插入到 div 里，他们变成父子结构了，这就再一次证明 appendChild 进行的是剪切的操作。

(2) 父级.insertBefore(a, b)

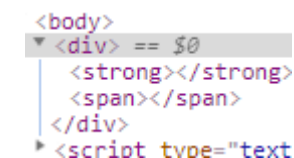
这个方法必须是父级调用，里边传两个参数，他的意思是在 b 之前插入 a，比如：

```

<body>
  <div>
    <span></span>
  </div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    var span = document.getElementsByTagName("span")[0];
    var strong = document.createElement("strong");
    div.insertBefore(strong, span);
  </script>
</body>

```

这样就是在 span 前边插入 strong：



```

<body>
  <div> == $0
  ├── <strong></strong>
  └── <span></span>
</div>
<script type="text/javascript">

```

第三种：删

(1) 父级.removeChild() 父级删除自己的子节点

```

<body>
  <div>
    <span></span>
    <strong></strong>
    <i></i>
  </div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    var span = document.getElementsByTagName("span")[0];

```

```

    var strong = document.getElementsByTagName("strong")[0];
    var i = document.getElementsByTagName("i")[0];
  </script>
</body>

```

现在 `div.removeChild(i)` 后得到：

```

<body>
  <div> == $0
    <span></span>
    <strong></strong>
  </div>

```

其实是把 `i` 标签剪切出来了，比如说上边的代码刷新后 `var ii = div.removeChild(i)`，你在访问 `ii` 就是 `<i></i>`。

(2) 自己.remove()

比如上边代码刷新后 `i.remove();strong.remove();`

```

<body>
  <div> == $0
    <span></span>
  </div>

```

`div` 里边就只剩下 `span` 了，`remove` 是真正的删除，删掉后就啥都没有了。

第四种：替换

父级.replaceChild(new, origin)

这个也是父级调用，里边传入两个参数，第一个是 `new` 就是新的，第二个是目标，比如说上边的代码，我在新建一个 `p`： `var p = document.createElement("p")`，然后 `div.replaceChild(p, i)`，用 `p` 替换掉 `i`：

```

<body>
  <div> == $0
    <span></span>
    <strong></strong>
    <p></p>
  </div>

```

4. Element 节点的一些属性

(1) innerHTML

这个属性可以改变 `html` 里的内容，比如说：

```

<body>
  <div>
    <span>123</span>
    <strong>234</strong>
  </div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];

```



```
</script>
```

```
</body>
```

我们把 div 选出来，然后 div.innerHTML 就得到“123234”，这是访问，我们还可以修改：div.innerHTML = “123”，这样就把原来的内容覆盖掉了：

```
<body> == $0
<div>123</div>
<script type="te
```

再比如我们在 123 后边想加上一个 234，就 div.innerHTML += “234”即可。再比如：div.innerHTML = “123”，这个也是好使的：

```
<body>
▼ <div>
  <span style="background-color:red;color:#fff;font-size:20px">123</span> == $0
  </div>
```

123

这个 innerHTML 取得是 HTML 结构，是可读写的，所以你写进去什么东西他都会识别成 HTML 结构。

(2) innerText

比如说：

```
<body>
```

```
<div>
```

```
<span>123</span>
```

```
<strong>234</strong>
```

```
</div>
```

```
<script type="text/javascript">
```

```
var div = document.getElementsByTagName("div")[0];
```

```
</script>
```

```
</body>
```

现在 div.innerText 就得到“123 234”，这个属性取的是 div 里边的内容，也可以赋值：div.innerText = “123”。

```
<nead>...</nead>
<body> == $0
<div>123</div>
<script type="te
```

但是这么赋值的话里边的东西就全部被覆盖了，所以在用这个方法的时候需要谨慎，如果标签底下有其他子标签，最好在赋值的时候不要用这个。还有这个 innerText 老版本的火狐浏览器不兼容，当时火狐有一个属性 textContent 和这个作用是一样的，

但是火狐这个方法老版本的 ie 不好使（都是老版本的兼容性问题，现在的新版本不存在不兼容的）。

5. Element 节点的一些方法

(1) 元素.setAttribute() 添加属性

```
<body>
  <div></div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
  </script>
</body>
```

现在 div.setAttribute("class", "demo"), 括号里第一个是属性名, 第二个是属性值, 在访问 div 就得到<div class="demo"></div>, 继续 div.setAttribute("id", "only"), 在访问 div 就是<div class="demo" id="only"></div>。

(2) 元素.getAttribute() 查看属性

接着上边的, 现在 div.getAttribute("id")就得到"only", div.getAttribute("class")就得到"demo"。

有了这些操作, 就更灵活了, 比如说我在 css 里定义了一个 class 样式, 然后可以再 js 里动态的添加 class 属性, 让这个样式作用在对应的元素上。

练习 6: 写三个标签 div, span, strong, 然后给这三个标签加上属性, 属性名是 "this-name", 属性值是标签名。

```
<body>
  <div></div>
  <span></span>
  <strong></strong>
  <script type="text/javascript">
    var all = document.getElementsByTagName("*");
    for(var i = 0; i < all.length; i++) {
      all[i].setAttribute("this-name", all[i].nodeName);
    }
  </script>
</body>
```

```
<div this-name="DIV"></div>
<span this-name="SPAN"></span>
<strong this-name="STRONG"></strong>
```

解析: 直接把他们都选中, 然后用遍历的方法给他们每一个调用 setAttribute 方法, 属性值就是对应的 nodeName。

练习 7: 请编写一段 JavaScript 脚本生成下边这段 DOM 脚本结构。要求使用标准的 DOM 方法和属性。

```
<div class="example">
  <p class="slogan">成哥，你最帅</p>
</div>

<script type="text/javascript">
  var div = document.createElement("div");
  var p = document.createElement("p");
  div.setAttribute("class", "example");
  p.setAttribute("class", "slogan");
  document.body.appendChild(div);
  div.appendChild(p);
  p.innerHTML = "成哥，你最帅";
</script>
```

效果:

```
<body>
  <script type="text/javascript">...</script>
  <div class="example">
    <p class="slogan">成哥，你最帅</p>
  </div>
```

解析: 这都是讲过的一些方法, 这里提示一下, `dom.className` 可以读写 `class`。

练习 8: 封装函数 `insertAfter()`, 功能类似 `insertBefore()`。

提示: 可忽略老版本浏览器, 直接在 `Element.prototype` 上编程。

```
<body>
  <div>
    <span></span>
    <i></i>
    <b></b>
    <strong></strong>
  </div>

  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    var b = document.getElementsByTagName("b")[0];
    var strong = document.getElementsByTagName("strong")[0];
    Element.prototype.insertAfter = function (targetNode, afterNode) {
      var beforeNode = afterNode.nextElementSibling;
```

```

        if(beforeNode == null) {
            this.appendChild(targetNode);
        }else {
            this.insertBefore(targetNode, beforeNode);
        }
    }

    var p = document.createElement("p");
</script>

```

</body>

解析：这是一个思想性的问题，咱们可以借助 insertBefore()，先把 afterNode 的最后一个元素节点求出来，在调用 insertBefore 方法，但是如果说 afterNode 后边没有了的话，咱就让父级直接 appendChild 即可。

```

> div.insertAfter(p,strong)
< undefined
> div
< ▼ <div>
  <span></span>
  <i></i>
  <b></b>
  <strong></strong>
  <p></p>
</div>
> div.insertAfter(p,b)
< undefined
> div
< ▼ <div>
  <span></span>
  <i></i>
  <b></b>
  <p></p>
  <strong></strong>
</div>

```

练习 9：将目标节点的内部节点逆序。

<body>

```

<div>
    <span></span>
    <i></i>
    <b></b>
    <strong></strong>
</div>
<script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];

```

```

    Element.prototype.inversElement = function () {
        var len = this.children.length;
        for(var i = len - 1; i >= 0; i--) {
            this.appendChild(this.children[i]);
        }
    }
</script>
</body>

```

```

> div.inversElement()
< undefined
> div
< <div>
  <strong></strong>
  <b></b>
  <i></i>
  <span></span>
</div>

```

解析：我们直接用 `appendChild` 剪切，第一次把倒数第一个节点剪切到最后，第二次把倒数第二个剪切到最后……写个 `for` 循环倒着操作即可。

Date 对象、定时器

1. 日期对象 Date

这个日期对象和我们之前学过的对象没有什么区别，只不过这个对象提供的属性和方法全部是关于日期的，日期对象是系统帮我们设定好了的，我们直接来研究他的一些属性和方法就行了，比如说 `var date = new Date();` 然后你访问 `date` 就得到 `Sun Feb 10 2019 11:56:47 GMT+0800 (中国标准时间)`。

我们去百度，查一下 W3Cschool 里提供的 `date` 属性和方法：

Date 对象属性	
属性	描述
constructor	返回对创建此对象的 Date 函数的引用。
prototype	使您有能力向对象添加属性和方法。
Date 对象方法	
方法	描述
Date()	返回当日的日期和时间。

<u>getDate()</u>	从 Date 对象返回一个月中的某一天 (1 ~ 31)。
<u>getDay()</u>	从 Date 对象返回一周中的某一天 (0 ~ 6)。
<u>getMonth()</u>	从 Date 对象返回月份 (0 ~ 11)。
<u>getFullYear()</u>	从 Date 对象以四位数字返回年份。
<u>getYear()</u>	请使用 <u>getFullYear()</u> 方法代替。
<u>getHours()</u>	返回 Date 对象的小时 (0 ~ 23)。
<u>getMinutes()</u>	返回 Date 对象的分钟 (0 ~ 59)。
<u>getSeconds()</u>	返回 Date 对象的秒数 (0 ~ 59)。
<u>getMilliseconds()</u>	返回 Date 对象的毫秒(0 ~ 999)。
<u>getTime()</u>	返回 1970 年 1 月 1 日至今的毫秒数。
<u>getTimezoneOffset()</u>	返回本地时间与格林威治标准时间 (GMT) 的分钟差。
<u>getUTCDate()</u>	根据世界时从 Date 对象返回月中的一天 (1 ~ 31)。
<u>getUTCDay()</u>	根据世界时从 Date 对象返回周中的一天 (0 ~ 6)。
<u>getUTCMonth()</u>	根据世界时从 Date 对象返回月份 (0 ~ 11)。
<u>getUTCFullYear()</u>	根据世界时从 Date 对象返回四位数的年份。
<u>getUTCHours()</u>	根据世界时返回 Date 对象的小时 (0 ~ 23)。
<u>getUTCMinutes()</u>	根据世界时返回 Date 对象的分钟 (0 ~ 59)。
<u>getUTCSeconds()</u>	根据世界时返回 Date 对象的秒钟 (0 ~ 59)。
<u>getUTCMilliseconds()</u>	根据世界时返回 Date 对象的毫秒(0 ~ 999)。
<u>parse()</u>	返回1970年1月1日午夜到指定日期 (字符串) 的毫秒数。
<u>setDate()</u>	设置 Date 对象中月的某一天 (1 ~ 31)。
<u>setMonth()</u>	设置 Date 对象中月份 (0 ~ 11)。
<u>setFullYear()</u>	设置 Date 对象中的年份 (四位数字)。
<u>setYear()</u>	请使用 <u>setFullYear()</u> 方法代替。
<u>setHours()</u>	设置 Date 对象中的小时 (0 ~ 23)。
<u>setMinutes()</u>	设置 Date 对象中的分钟 (0 ~ 59)。
<u>setSeconds()</u>	设置 Date 对象中的秒钟 (0 ~ 59)。
<u>setMilliseconds()</u>	设置 Date 对象中的毫秒 (0 ~ 999)。
<u>setTime()</u>	以毫秒设置 Date 对象。
<u>setUTCDate()</u>	根据世界时设置 Date 对象中月份的一天 (1 ~ 31)。
<u>setUTCMonth()</u>	根据世界时设置 Date 对象中的月份 (0 ~ 11)。
<u>setUTCFullYear()</u>	根据世界时设置 Date 对象中的年份 (四位数字)。
<u>setUTCHours()</u>	根据世界时设置 Date 对象中的小时 (0 ~ 23)。
<u>setUTCMinutes()</u>	根据世界时设置 Date 对象中的分钟 (0 ~ 59)。
<u>setUTCSeconds()</u>	根据世界时设置 Date 对象中的秒钟 (0 ~ 59)。
<u>setUTCMilliseconds()</u>	根据世界时设置 Date 对象中的毫秒 (0 ~ 999)。
<u>toSource()</u>	返回该对象的源代码。
<u>toString()</u>	把 Date 对象转换为字符串。

<u>toSource()</u>	返回该对象的源代码。
<u>toString()</u>	把 Date 对象转换为字符串。
<u>toTimeString()</u>	把 Date 对象的时间部分转换为字符串。
<u>toDateString()</u>	把 Date 对象的日期部分转换为字符串。
<u>toGMTString()</u>	请使用 <u>toUTCString()</u> 方法代替。
<u>toUTCString()</u>	根据世界时，把 Date 对象转换为字符串。
<u>toLocaleString()</u>	根据本地时间格式，把 Date 对象转换为字符串。
<u>toLocaleTimeString()</u>	根据本地时间格式，把 Date 对象的时间部分转换为字符串。
<u>toLocaleDateString()</u>	根据本地时间格式，把 Date 对象的日期部分转换为字符串。
<u>UTC()</u>	根据世界时返回 1970 年 1 月 1 日 到指定日期的毫秒数。
<u>valueOf()</u>	返回 Date 对象的原始值。

(1) Date 对象的属性：只有两个，**prototype** 和 **constructor**，我们之前都讲过，这里就不说了，只不过要区分开：**prototype** 是函数类对象（构造函数）上的属性，**constructor** 是对象类对象（原型）上的属性。

(2) Date 对象的方法：

Date()：Date 空执行后返回的是字符串类型的日期时间，得“Sun Feb 10 2019 18:10:59 GMT+0800（中国标准时间）”。

以下方法全部是 Date 生产的对象调用的，比如说 `var date = new Date()`；

getDate()：他返回的是这个月中的第几天，拿今天 2 月 10 日为例，`date.getDate()` 就得到 10。

getDay()：他返回的是一周中的第几天，但是他是从 0 记起的，并规定星期天为第一天，则返回 0，星期一返回 1，那么今天 `date.getDay()` 就得到 0。

getMonth()：返回当前月份是今年的第几月，但也是从 0 开始记起的，那么今天二月 `date.getMonth()` 就得到 1。

getYear()：这个方法不准，是早期方法，`date.getYear()` 得到 119，上边让我们用 `getFullYear()` 代替。

getFullYear()：返回四位数的数字年份，`date.getFullYear()` 就得到 2019。

getHours()：返回小时，`date.getHours()` 就得到 18。

getMinutes()：返回分钟，`date.getMinutes()` 就得到 10。

getSeconds()：返回秒数，`date.getSeconds()` 就得到 59。

getMilliseconds()：返回毫秒，范围是 0-999。

注意：以上 get 的所有方法都不是实时的，它记录的是 date 对象出生的时间，比如说我把网页刷新一下，然后 `date.getSeconds()` 得到 2，然后不刷新继续 `date.getSeconds()` 还是 2。所以说他记录的不是你访问的时间，而是 date 出生的时间。

getTime()：返回 1970 年 1 月 1 日至今的毫秒数，`date.getTime()` 就得到 1549795905341。

解释：这个 1970 年 1 月 1 日 0 时 0 分 0 秒被定义为计算机的纪元时间，这个方法就可以准确地计算出在这个 date 出生的时间距离纪元时间多少毫秒，他的作用就是可以当做一个时间戳用，来检验一个程序运行的效率，比如说：

```
var firstTime = new Date().getTime();
for(var i = 0; i < 1000000000; i++){

}
```

```
var lastTime = new Date().getTime();
console.log(lastTime - firstTime);
```

我们让计算机空循环一亿圈，在循环开始之前记录一下时间戳，结束后再记录一下时间戳，然后用后边的时间毫秒数减去前边的时间毫秒数来看一下计算机运行的效率。

再比如你写了一个程序，想看一下运行的效率，也可以用时间戳测一下写的这个程序运行花了多少秒，看能不能继续优化一下。

这个日期对象除了一些 get 的方法以外，还有一些 set 的方法，就是除了读取时间以外，还可以手动设置时间。现在继续 `var date = new Date()`；然后 `date.setDate(15)`，再访问 date 就得到 Fri Feb 15 2019 11:30:29 GMT+0800（中国标准时间），这个 set 的方法就是比如你把日期改了，其他的星期啥的就也跟着一起变了。set 还有一些方法，比如 `setMonth()`、`setFullYear()`、`setHours()` 等等，这里就不试了。

set 的这些方法的作用就是可以当做一个闹钟来用，比如说你设置了一个时间，然后和系统的时间相比对（用 `getTime()` 的方法做差），如果时间相吻合的话，就触发某种事件：

```
var date = new Date();
date.setMinutes(54);
setInterval(function () {
    if(date.getTime() - new Date().getTime() < 1000){
        console.log("老邓还是个宝宝");
    }
}, 1000)
```

这个 `setInterval` 是定时器，就是每隔 1000 毫秒执行一次这个函数，我们设置了一个时间，用时间戳来求一下，如果设置的这个时间减去系统时间小于 1000 毫秒的话，我们就说他们是相吻合的，然后打印里边的内容，因为每隔 1000 毫秒都会执行一次，所以时间到了以后他会一直打印。

还有一个 `setTime()` 的方法，就是通过毫秒数来设置时间，比如说 `date.setTime(12345678999)`，在访问 date 就是 Sun May 24 1970 05:21:18 GMT+0800

(中国标准时间)，这个 `setTime()` 方法除非有人给你传送数据来设置，自己设置的话太扯淡了。

`toString()`: 把 `date` 转换为字符串，`date.toString()` 就得到 "Sat Feb 16 2019 12:12:44 GMT+0800 (中国标准时间)"。

除了以上讲的这些方法，表中的其他方法都没啥用。记住上面这些就行了。

练习 1: 封装函数，打印当前是何年何月何日何时几分几秒。

```
function myDate() {
    var date = new Date();
    console.log("当前是" + date.getFullYear() + "年" + (date.getMonth() + 1)
+ "月" + date.getDate() + "日" + date.getHours() + "时" + date.getMinutes()
+ "分" + date.getSeconds() + "秒");
}
```

解析：这个比较简单，直接 `get` 就行了，但是注意 `getMonth()` 是从 0 开始记起的，所以得给他加 1，现在 `myDate()` 就打印 当前是 2019 年 2 月 16 日 12 时 23 分 20 秒。

2. 定时器

(1) `setInterval()` 比如说：

```
<script type="text/javascript">
    setInterval(function () {
        console.log("a");
    }, 1000)
</script>
```

这个 `setInterval()` 方法里边传进去两个参数，第一个是函数，第二个是数字，后边单位是毫秒，他的意思就是每隔 1000 毫秒就会执行一次这个函数，打印 a，无休无止。你后边传入数字几，他就会每隔几毫秒执行一次函数。

注意：比如说：

```
<script type="text/javascript">
    var time = 1000;
    setInterval(function () {
        console.log("a");
    }, time)
    time = 2000;
</script>
```

现在 `var` 一个 `time` 等于 1000，把 `time` 传进去，这样也是可以的，但是你在下边给 `time` 重新赋值的话，上边定时器里还是 1000，因为 `time` 在定时器里会取一次值，你想在后

边通过改变 time 来改变定时器里的时间是不可以的。

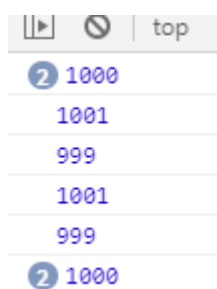
setInterval() 的作用就是每隔一段时间去执行一次函数，那么我们就可以拿他做很多东西，例如用它查数：

```
<script type="text/javascript">
    var i = 0;
    setInterval(function () {
        i++;
        console.log(i);
    }, 1000)
</script>
```

拓展：我们来写一个程序看一下这个定时器到底准不准：

```
<script type="text/javascript">
    var firstTime = new Date();
    setInterval(function () {
        var lastTime = new Date();
        console.log(lastTime - firstTime);
        firstTime = lastTime;
    }, 1000)
</script>
```

在函数外边 var 一个 firstTime，函数里在 var 一个 lastTime，打印出时间差，最后让 firstTime 等于 lastTime 以便于下一圈计算，来看一下效果：



由上图可以得出结论：定时器是非常不准的，至于为什么不准，咱们后边再说。

备注：setInterval() 是全局 window 上的方法，所以你在写的时候不用写 window.setInterval()，直接写 setInterval() 他就会在全局的作用域 G0 里边找。

(2) clearInterval()

这个方法是让定时器停的，其实 setInterval() 方法会返回值，比如说：

```
var timer = setInterval(function() {}, 1000);
var timer1 = setInterval(function() {}, 1000);
```

你现在访问 timer 就是 1，访问 timer1 就是 2，返回的值作为唯一标识来用，你想让定时器停的话就直接把这个定时器的返回值传入 clearInterval() 里即可，所以通常我们写定时器的时候 var 一个变量来接受他的返回值。比如现在写一个定时器打印 i++，然后当 i 大于 10 就让定时器停止：

```
<script type="text/javascript">
  var i = 0;
  var timer = setInterval(function () {
    console.log(i++);
    if (i > 10) {
      clearInterval(timer);
    }
  }, 10)
</script>
```

(3) setTimeout() 比如：

```
<script type="text/javascript">
  setTimeout(function () {
    console.log("a");
  }, 1000)
</script>
```

这个 setTimeout() 才是真正的定时器，他的作用是隔 1000 毫秒之后才会执行这个函数打印 a，并且只执行一次。而 setInterval() 是一直会循环执行。

(4) clearTimeout()

同样的，setTimeout() 也会返回值作为唯一标识，并且返回的值不会和 setInterval() 重复，所以他也可以用 clearTimeout() 来让他停：

```
<script type="text/javascript">
  var timer = setTimeout(function () {
    console.log("a");
  }, 1000)
  clearTimeout(timer);
</script>
```

这样的话就不会打印 a 了。

备注：以上四个方法全是 window 上的，函数里假如有 this 就会指向 window；还有就是定时器还有另一种写法，第一个参数传字符串的话也可以当代码执行，例如 setInterval("console.log('a')", 1000)，这样也可以打印，但是一般不怎么用。

练习 2: 计时器, 到三分钟停止。

```
<style>
  input {
    border: 1px solid rgba(0, 0, 0, 0.8);
    text-align: right;
    font-size: 20px;
    font-weight: bold;
  }
</style>
<body>
  minutes:<input type="text" value="0">
  seconds:<input type="text" value="0">
  <script type="text/javascript">
    var min = document.getElementsByTagName("input")[0];
    var sec = document.getElementsByTagName("input")[1];
    var timer = setInterval(function () {
      sec.value++;
      if (sec.value == 60) {
        sec.value = 0;
        min.value++;
      }
      if (min.value == 3) {
        clearInterval(timer);
      }
    }, 1000)
  </script>
</body>
```

效果:

minutes: seconds:

解析: 这个计数器后边的秒数加到 59 然后归零, 分钟数进 1, 其实就是用 DOM 元素的 value 值做的, 写一个定时器, 然后当分钟数加到 3 的时候, 我们让定时器停止就可以了。

获取窗口属性、获取 DOM 尺寸、脚本化 CSS

1. 查看滚动条的滚动距离

有的时候一个页面会出现滚动条，我们就需要知道滚动条滚动了多少距离，比如说滚动条滚动了 400 像素的距离，他的意思就是在第一屏的基础上又加了 400px 的内容。因为假如你不过第一屏的话也不可能用到滚动条。

标准方法: `window.pageXOffset` `window.pageYOffset`

咱们先来看一下垂直方向的滚动条，比如说你在 body 里写了 100 个 br 标签，然后当你没有滚动的时候访问 `window.pageYOffset` 就得到 0，然后比如说你往下滚动了一段距离，在访问 `window.pageYOffset` 就得到 407，他这个数字的单位肯定是 px，只不过他返回的没有加单位，你把滚动条拉到底部，在访问 `window.pageYOffset` 就是 1532。（上边返回的数字是我在我电脑上操作的，仅供参考）

思考：比如说滚动条往下滚动了 400px，那么从这个浏览器的最顶端到滚动条滚动的这个位置的底端一共有多少像素？

答：应该是 400 像素加上首屏像素，因为滚动条滚动的距离就相当于多挪出来的距离。然后咱们再来看看横向的滚动条的滚动距离，写一个 hr 标签（水平线），把他的宽度设置成 10000px，肯定超过屏幕宽度了，就有滚动条了：`<hr style="width:10000px;">`，然后我滚动一段距离，再访问 `window.pageXOffset` 就得到 1874。（返回的数字是我在我电脑上操作的，仅供参考）

但是以上两种方法 ie8 及 ie8 以下浏览器不兼容，这些浏览器提供了两种方法：

第一种: `document.body.scrollLeft`（和 `window.pageXOffset` 效果一样）

`document.body.scrollTop`（和 `window.pageYOffset` 效果一样）

第二种: `document.documentElement.scrollLeft`

`document.documentElement.scrollTop`

以上两种方法兼容性比较混乱，就是 ie8 及 ie8 以下的浏览器有的浏览器版本第一个方法好使，有的版本第二个方法好使，但是任何一个浏览器版本只要一种方法好使，返回的有值，另一种不好使的方法返回的一定是 0，所以咱们在 ie8 及 ie8 以下的浏览器不管哪个版本直接把两个值相加就行了。

练习 1：封装兼容性方法，求滚动条滚动距离 `getScrollOffset()`

```
function getScrollOffset() {
    if (window.pageXOffset) {
        return {
            x: window.pageXOffset,
            y: window.pageYOffset
        }
    }
}
```

```

    }
  } else {
    return {
      x:document.body.scrollLeft + document.documentElement.scrollLeft,
      y:document.body.scrollTop + document.documentElement.scrollTop
    }
  }
}

```

解析：这个比较简单，如果 window 上的方法能用就用 window 上的，如果用不了就走 else 把两个方法的值相加，最后 return 的是一个对象。方法封装好后可以放进 js 代码库里备注好，下次直接调用即可。

2. 查看可视区窗口尺寸

可视区窗口就是咱们编写的 html 文档能看到的部分，不包括菜单栏、地址栏和控制台。

标准方法：`window.innerWidth` `window.innerHeight`

现在访问 `window.innerWidth` 就得到 1920，访问 `window.innerHeight` 得到 584，（返回的数字是我在我电脑上操作的，仅供参考），这里注意假如说我把页面放大了，那么访问的这个值也会跟着变，就缩小了，因为你放大页面之后看到的只是那一小块的距离。但是注意以上两个方法还是 ie8 及 ie8 以下版本不兼容。这些浏览器版本提供了两种方法，一种是在标准模式下用的，一种是在怪异模式下用的。

备注：什么是怪异模式？比如说在很久以前 ie7 还没有诞生，我写了一个页面，语法全部用的是 ie6 的语法，但是一年之后 ie7 诞生了，人们都开始用新的浏览器，那么我之前写的那个页面的部分语法就不能用了，因为有冲突，重写的话又太浪费时间，后来人们研究了一种新的渲染模式叫怪异模式，比如说现在我启动了怪异模式，在这个模式下即使人们用的是 ie7 浏览器，他也能根据 ie6 的语法把这个页面渲染出来，这个怪异模式也叫混杂模式，这个模式一经启动他识别的就不是现在的语法而是之前的语法，起到了一个向后兼容的作用，兼容之前的语法。那么怎么启用怪异模式？其实我们在讲 html 的时候他第一行应该是 `<!DOCTYPE html>`，这个我们一直没说，其实有这一行就是标准模式，要想启动怪异模式，直接把这一行删掉即可。

第一种：标准模式下，任意浏览器都能兼容

`document.documentElement.clientWidth`

`document.documentElement.clientHeight`

第二种：适用于怪异模式下的浏览器

`document.body.clientWidth`

`document.body.clientHeight`

怎么区分标准模式和怪异模式？document 上有个属性 `compatMode`，在标准模式下访问 `document.compatMode` 得 `"CSS1Compat"`，在怪异模式下访问 `document.compatMode` 得 `"BackCompat"`。

练习 2：封装兼容性方法，返回浏览器视口尺寸 `getViewportOffset()`

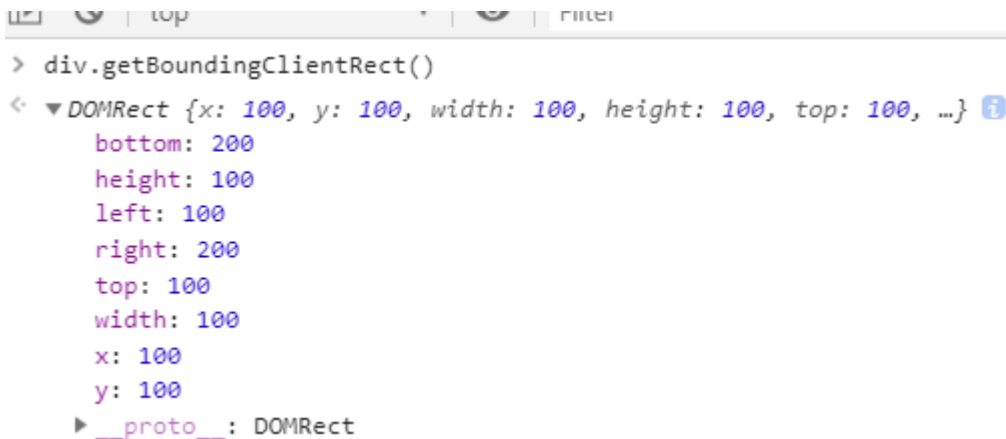
```
function getViewportOffset() {
    if (window.innerWidth) {
        return {
            w: window.innerWidth,
            h: window.innerHeight
        }
    } else {
        if (document.compatMode === "BackCompat") {
            return {
                w: document.body.clientWidth,
                h: document.body.clientHeight
            }
        } else {
            return {
                w: document.documentElement.clientWidth,
                h: document.documentElement.clientHeight
            }
        }
    }
}
```

解析：这个和上边封装的那个方法思想一样，比较简单，封装好之后也放到代码库里存起来便于后续使用。

3. 查看元素几何尺寸

`dom.getBoundingClientRect()`

比如说我现在写了一个 div 并给他加了一些样式：`<div style="width:100px;height:100px;background-color:red;position:absolute;top:100px;left:100px;"></div>`，然后在 js 里把他选出来：`var div = document.getElementsByTagName("div")[0]`；现在访问 `div.getBoundingClientRect()` 就得到：



```

> div.getBoundingClientRect()
< DOMRect {x: 100, y: 100, width: 100, height: 100, top: 100, ...}
  bottom: 200
  height: 100
  left: 100
  right: 200
  top: 100
  width: 100
  x: 100
  y: 100
  __proto__: DOMRect

```

可以看到返回的是一个对象，对象里边有 left、top、right、bottom 等属性，left 和 top 代表该元素左上角的 X 和 Y 坐标，right 和 bottom 代表元素右下角的 X 和 Y 坐标，这个方法兼容性很好，但是老版本的 ie 里并没有 height 和 width，所以我们在老版本的 ie 只能计算来求宽高，而且这个方法返回的结果并不是实时的，比如说我 `var box = div.getBoundingClientRect();div.style.width = "200px"`，再访问 box 的话里边的 width 还是 100px，所以他不是实时的。

4. 查看元素的尺寸: `dom.offsetWidth` `dom.offsetHeight`

比如说我现在写了一个 div 并给他加了一些样式：`<div style="width:100px;height:100px;background-color:red;position:absolute;top:100px;left:100px;"></div>`，然后在 js 里把他选出来：`var div = document.getElementsByTagName("div")[0];`然后 `div.offsetWidth` 就得到 100，`div.offsetHeight` 也是 100. 这两个方法返回的是可视区的宽高，比如说我现在给这个 div 样式里加上 `padding: 100px`，在访问 `div.offsetHeight` 就是 300，所以这个方法求出来的宽高是可视区的宽高，我们再来看一下 `div.getBoundingClientRect()`，访问后得到 `DOMRect {x: 100, y: 100, width: 300, height: 300, top: 100, ...}`，可见他这个方法求出来的也是可视区的宽高。

5. 查看元素的位置: `dom.offsetLeft` `dom.offsetTop`

还是上边的这个 div，现在 `div.offsetLeft` 就得到 100，`div.offsetTop` 也是 100. 再比如：

```

<style>
  .wrapper {
    width: 300px;
    height: 300px;
    border: 2px solid black;
    position: relative;

```

```

        top: 100px;
        left: 100px;
    }
    .box {
        width: 100px;
        height: 100px;
        background-color: red;
        position: absolute;
        top: 100px;
        left: 100px;
    }
</style>
<body>
    <div class="wrapper">
        <div class="box"></div>
    </div>
    <script type="text/javascript">
        var div = document.getElementsByClassName("box")[0];
    </script>
</body>

```

我们把里边的 div 选出来，然后 `div.offsetLeft` 得到 100，可见他求出来是相对于父级的位置，然后我继续改：

```

    .box {
        width: 100px;
        height: 100px;
        background-color: red;
        margin-left: 100px;
        margin-top: 100px;
    }

```

我把里边的 div 的定位去掉，然后又加上 margin，再访问 `div.offsetLeft` 还是 100，所以说这个 `offsetLeft` 忽略自己是否为定位元素，他求的只是自己和有定位的父级之间的距离，也就是说不管这段距离是自己定位出来的还是 margin 生成的都能求出来，这个 `offsetLeft` 求得只是一个自己和父级之间的距离，他并不是语法，只是一个方向。但是他关注的是外边的方块是不是有定位的父级，如果是，他求的就是自己和有定位

的父级之间的距离，如果不是，那么求出来就是自己和文档之间的距离。现在我在改一下：

```
.wrapper {
  width: 300px;
  height: 300px;
  border: 2px solid black;
  margin-top: 100px;
  margin-left: 100px;
}
```

我们把父级的定位去掉，然后又加上 margin 值，那么现在求出来的就是相对于文档的距离了，div.offsetLeft 就得到 210（两个 margin 各 100，border 2 像素，还有由于我们没有初始化，所以 body 有 8 像素的 margin）。div.offsetTop 得到 202（上边的 8 像素和外边的 div 的 margin 重叠了，margin 塌陷）。

dom.offsetParent: 返回最近有定位的父级，若无，返回 body，body.offsetParent 返回 null。

6. 让滚动条滚动

window 上有三个方法：**window.scroll()** **window.scrollTo()** **window.scrollBy()**

三个方法功能类似，用法都是将 x、y 坐标传入，即实现让滚动条滚动到当前位置。

window.scroll() 和 **window.scrollTo()** 两个方法完全一样，兼容性也一样，比如说 **window.scroll(0,100)** 他就能让 y 方向的滚动条滚动到 100 像素这个位置，你继续 **window.scroll(0,100)** 他是不变的，说明他是让滚动条滚动到当前位置。

而 **window.scrollBy()** 是累加滚动当前距离，比如说 **window.scrollBy(0,10)** 他是让滚动条向下滚动 10 像素，继续 **window.scrollBy(0,10)** 就继续向下滚动 10 像素，继续 **window.scrollBy(0,-10)** 就又向上滚动 10 像素。

应用：我们在手机上浏览网页的时候，经常有展开更多，点开阅读完点击收起他又回到了刚才那个位置，其实这个特别简单，在点击展开更多的时候咱们记录一下滚动条的位置，就是前边讲的滚动条的滚动距离，用封装好的方法记录好数据之后点击收起的时候用 **window.scroll()** 再把刚才记录的数据传进去就完事了。

练习 3：咱们来模仿手机阅读器，做一个自动阅读的功能。（源码中的文字我就不往这里粘贴了哈）

```
<div
style="width:100px;height:100px;background-color:orange;color:#fff;font-size:40px;font-weight:bold;text-align:center;line-height:100px;position:fixed;bottom:200px:right:50px;border-radius: 50%;opacity:0.5;">start</div>
```


打开锁即可。

7. 脚本化 css

(1) 读写元素 css 属性 `dom.style`

比如说：

`<body>`

```
<div style="width:100px;height:100px;background-color:red"></div>
```

```
<script type="text/javascript">
```

```
var div = document.getElementsByTagName("div")[0];
```

```
</script>
```

`</body>`

首先，任何一个 dom 元素都会有 style 属性，我们在控制台访问 `div.style` 得到：

```
> div.style
< CSSStyleDeclaration {0: "width", 1: "height", 2: "background-color", alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: "", ...}
```

返回的是一个类数组，他这里边把这个 div 能用的所有属性都罗列出来了，其中 width、height、backgroundColor 咱们设置了是有值的，其他的样式咱们没有设置但他也是存在的，只不过没有值。返回给我们是一个样式表，这个样式表是可读可写的，现在 `div.style.width` 就得到“100px”，也可以写入，比如 `div.style.width = "200px"`，写值的时候必须是字符串形式的才可以。这里改了之后页面中的宽也就改了。这就是间接地改变 css，就是通过改变 html 的 style 属性来改变他的行内样式。还有就是组合单词在 js 里不能用中划线，所以在 css 里有些带有中划线的组合单词在 js 里必须采用小驼峰式大写来读写属性，比如 `div.style.backgroundColor = "green"`。其实你在 html 里没写的行内样式属性也可以改的，因为你 `div.style` 里返回的样式表里边没有值的话我就可以添值，比如说 `div.style.borderRadius = "50%"`，设置后他就有了圆角了。再比如：

```
<style>
```

```
div{
```

```
width: 200px;
```

```
}
```

```
</style>
```

```
<body>
  <div style="height:100px;background-color:red"></div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
  </script>
</body>
```

我们知道 style 标签里写的样式和行间样式都能作用到 div 上，但是你访问 div.style.width 就得到“”，虽然可以作用到 div 上，但是你没有写在行间，js 里就访问不到。就是 style 这个属性无论是读的还是在写的都是在行间里的。

这个 dom.style 没有任何兼容性问题，但是注意碰到 float 这样的保留字属性，前边应该加上 css，如 div.style.cssFloat = “right”。还有就是复合属性（如 border）最好把他拆解开设置，但是现在写在一起也是可以的，最好把他拆解开。

（2）查询计算样式 `window.getComputedStyle(dom, null)`

比如说：

```
<style>
  div{
    width: 200px;
  }
</style>
<body>
  <div style="height:100px;background-color:red"></div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
  </script>
</body>
```

这个方法需要传入两个参数，第一个是 dom 元素，第二个是 null，比如现在访问 window.getComputedStyle(div, null) 得到：



```
> window.getComputedStyle(div, null)
CSSStyleDeclaration {0: "animation-delay", 1: "animation-direction", 2: "animation-duration", 3: "animation-fill-mode", 4: "animation-iteration-count", 5: "animation-name", 6: "animation-play-state", 7: "animation-timing-function", 8: "background-attachment", 9: "background-blend-mode", 10: "background-clip", 11: "background-color", 12: "background-image", 13: "background-origin", 14: "background-position", 15: "background-repeat", 16: "background-size", 17: "border-bottom-color", 18: "border-bottom-left-radius", 19: "border-bottom-right-radius", 20: "border-bottom-style", 21: "border-bottom-width", 22: "border-collapse", 23: "border-image-outset", 24: "border-image-repeat", 25: "border-image-slice", 26: "border-image-source", 27: "border-image-width", 28: "border-left-color", 29: "border-left-style", 30: "border-left-width", 31: "border-right-color", 32: "border-right-style", 33: "border-right-width", 34: "border-top-color", 35: "border-top-left-radius", 36: "border-top-right-radius", 37: "border-top-style", 38: "border-top-width", 39: "bottom", 40: "box-shadow", 41: "box-sizing", 42: "break-after", 43: "break-before", 44: "break-inside", 45: "caption-side", 46: "clear", 47: "clip", 48: "color", 49: "content", 50: "cursor", 51: "direction", 52: "display", 53: "empty-cells", 54: "float", 55: "font-family", 56: "font-kerning", 57: "font-size", 58: "font-stretch", 59: "font-style", 60: "font-variant", 61: "font-variant-ligatures", 62: "font-variant-numeric", 63: "font-variant-underline", 64: "font-weight", 65: "height", 66: "image-rendering", 67: "isolation", 68: "justify-items", 69: "justify-self", 70: "left", 71: "letter-spacing", 72: "line-height", 73: "list-style-type", 74: "list-style-image", 75: "list-style-position", 76: "list-style-type", 77: "margin-bottom", 78: "margin-left", 79: "margin-right", 80: "margin-top", 81: "max-height", 82: "max-width", 83: "min-height", 84: "min-width", 85: "mix-blend-mode", 86: "object-fit", 87: "object-position", 88: "offset-distance", 89: "offset-path", 90: "offset-rotate", 91: "opacity", 92: "orphans", 93: "outline-color", 94: "outline-offset", 95: "outline-style", 96: "outline-width", 97: "overflow-anchor", 98: "overflow-wrap", 99: "overflow", ...}
```

他返回的也是一个样式表，但是和上边的不一样，style 里读的只是行间里的样式，假如说你在行间没有设置的话他就没有值，但是这个方法返回的属性里即使你没有设置

他也是有值的，是默认值，这个方法获取的是当前元素所展示的一切 css 的显示值，就是假如说你通过多个选择器给一个元素设置了一个属性，那么只有权重最高的那个起作用，而这个方法获取的只是那个起作用的也就是显示的那个值和一些默认值（不管写在行间还是外部只要能显示就都能获取到），比如说 `window.getComputedStyle(div,null).width` 就得到“200px”。这个方法是只读的，不可以写入，写入值会报错。而且返回的计算样式都是绝对值，没有相对单位，就是人家给你计算好之后才返回给你的，没有那些相对的表达形式。比如说你给这个 div 设置 10em 的宽，再访问 `window.getComputedStyle(div,null).width` 就得到“160px”，再比如你访问 `window.getComputedStyle(div,null).backgroundColor` 就得到“rgb(255, 0, 0)”，返回的是 rgb 值。最后就是这个方法 ie8 及 ie8 以下不兼容。

备注：你知道 `window.computedStyle()` 这个方法第二个参数是干嘛的吗？为啥要传 null 呢？第二个参数传对了，可以获取伪元素的样式，比如说上边这个 div，我给他伪元素加点样式：

```
div::after{
    content: "";
    width: 50px;
    height: 50px;
    background-color: green;
    display: inline-block;
}
```



现在教你在 js 里怎么把这个 div 的伪元素的样式拿出来（唯一方法），把 div 选出来后，调用方法，第二个值传字符串形式的 after 即可，例如 `window.getComputedStyle(div,"after").width` 就得到“50px”。也是只能读取不能写入。

练习 4：就上边这个 div，现在加了一个绿色方块的伪元素，好，我现在要求当鼠标点击 div 之后伪元素变黄。咋办？我在上边写的代码的基础上继续写代码哈：

```
<style>
    .yellow::after{
        content: "";
        width: 50px;
        height: 50px;
        background-color: yellow;
        display: inline-block;
    }
```

```

</style>
<body>
  <div class="demo" style="height:100px;background-color:red"></div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    div.onclick = function () {
      div.className = "yellow";
    }
  </script>
</body>

```

解析：这是一个思想性的问题，因为伪元素的属性是只读的，根本写入不了，我们就在 css 里先定义一个 yellow class 的样式，然后当鼠标点击的时候我们把 div 的 class 名改成 yellow 即可。咱们以后开发的时候，也可以借用这种思想，比如说一个 div，点击一次后他的长宽背景颜色啥的都要改变，在 js 里 style 确实能实现，但是太浪费效率，咱们在 css 里把点击后的样式用 class 名定义好，然后在 js 里当鼠标点击的时候直接改他的 class 名就可以了，这样既节省了效率，后期也比较好维护。

(3) 查询样式 `dom.currentStyle` (ie 独有的属性)

这个是 ie 独有的属性，他也能返回一个样式表，和 `window.getComputedStyle()` 方法类似，也是只能读取不能写入，他获取的也是最终展示的那个值，但是他返回的计算样式的值不是经过转换的绝对值，写啥就展示啥。

练习 5：封装兼容性方法 `getStyle(elem, prop)`

```

function getStyle(elem, prop) {
  if (window.getComputedStyle) {
    return window.getComputedStyle(elem, null)[prop];
  } else {
    return elem.currentStyle[prop];
  }
}

```

解析：这个比较简单，注意第二个参数你传进来必须是字符串形式的，所以里边写的时候要用中括号的形式。

练习 6：做一个小木块运动。

```

<body>
  <div
    style="width:100px;height:100px;background-color:red;position:absolute;"><

```

```

</div>
<script type="text/javascript">
    function getStyle(elem, prop) {
        if (window.getComputedStyle) {
            return window.getComputedStyle(elem, null)[prop];
        } else {
            return elem.currentStyle[prop];
        }
    }
    var div = document.getElementsByTagName("div")[0];
    var speed = 2;
    var timer = setInterval(function () {
        speed += speed / 7;
        div.style.left = parseInt(getStyle(div, "left")) + speed + "px";
        if (parseInt(getStyle(div, "left")) > 500) {
            clearInterval(timer);
        }
    }, 10)
</script>
</body>

```

解析：现在我们做一个让木块运动就比较简单了，直接用定时器改变 left 值就可以了，但是改变之前你要获取到，我们封装的那个方法把他引进来，然后返回的值是字符串，在做运算的时候用 `parseInt()` 把后边的 px 砍掉然后转换为数字进行运算，你也可以让他在特定位置停，还可以做加速之类的都可以。

DUYI EDUCATION 事件

1. 交互和事件的定义：交互就是你对程序作了一个小动作，程序给你的一个反馈。事件就是交互体验的核心功能。事件就是一个动作，没效果也是事件，比如说一个 div，你点击一下他没有效果，但是也可以被点击，他天生就有被点击的事件。

2. 如何绑定事件处理函数

我们说一个元素天生就有事件，天生就可以被别人操作，我们绑定的只是事件处理函数，就是当你点击之后给你的反馈。

(1) `dom.onxxx = function () {}`

比如说：


```

<body>
  <div style="width:100px;height:100px;background-color:red;"></div>
  <script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    div.onclick = function () {
      this.style.backgroundColor = "green";
    }
  </script>
</body>

```

这样就是通过 on 的方式给这个 div 绑定一个 click 事件，即点击事件，那么当你点击的时候他的背景颜色就变绿了。这个绑定方式兼容性非常好，但是一个元素的同一个事件上只能绑定一个处理函数，比如说：

```

div.onclick = function () {
  console.log("a");
}
div.onclick = function () {
  console.log("b");
}

```

这样是不可以的，因为虽然他是事件，但是他也属于给对象上的属性赋值，你这么写的话后边的就会覆盖掉前边的。然后这种绑定方式基本等同于写在 HTML 行间上，就是比如说：

```

<div style="width:100px;height:100px;background-color:red;"
onclick="console.log('a')"></div>

```

我们可以把它直接写在行间，效果是一样的，行间就不用写 function 了，这种叫做句柄的写法，`dom.onxxx = function () {}` 叫做句柄的绑定方式。

(2) `dom.addEventListener()`

这是最标准的绑定方法，里边需要传三个参数，第一个是事件类型，第二个是处理函数，第三个传 `false`。比如上边那个 div：

```

div.addEventListener("click", function() {
  console.log("a")
}, false)

```

这样也可以实现 click 事件的绑定，然后这个可以为一个事件绑定多个处理程序，比如说：

```

div.addEventListener("click", function() {

```

```
    console.log("a")
}, false)
div.addEventListener("click", function() {
    console.log("a")
}, false)
```

这样就能点一次打印两个 a，但是比如说：

```
div.addEventListener("click", test, false)
div.addEventListener("click", test, false)
function test() {
    console.log('a')
}
```

这样点击一次只能打印一个 a，因为上边的绑定的函数虽然长得一样，但是是两个人，下边显然是一个人，所以说这个方法同一个函数不能重复绑定多次，还有就是 ie9 以下不兼容。

(3) `dom.attachEvent()` (ie 独有)

这个方法是 ie 独有的，里边传入两个参数，第一个是“on” + 事件类型，第二个是处理函数，比如说：

```
div.attachEvent("onclick",function () {
    console.log("a");
})
```

这个方法和 `dom.addEventListener()` 极其类似，也可以为一个事件绑定多个处理程序，而且同一个函数可以重复绑定多次。

备注：去看一下前边闭包那一节的练习 2，这里提示一下，以后但凡事件被放在循环里，你就要考虑一下是否形成闭包，如果闭包对你有影响，就要在外边套立即执行函数。

3. 事件处理程序的运行环境

(1) `dom.onxxx = function () {}` 程序 `this` 指向 dom 元素本身，比如说：

`<body>`

```
<div style="width: 100px;height: 100px;background-color: red;"></div>
<script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    div.onclick = function () {
        console.log(this);
    }
</script>
```

</body>

你点击这个 div 后控制台就会打印 <div style="width: 100px;height: 100px;background-color: red;"></div>。

(2) dom.addEventListener() 程序 this 指向 dom 元素本身，还是上边的 div:

```
div.addEventListener("click", function () {
    console.log(this);
}, false)
```

点击 div 控制台打印<div style="width: 100px;height: 100px;background-color: red;"></div>。

(3) dom.attachEvent() 程序 this 指向 window，这是一个 bug。那我现在就想让 this 指向调用者，可以这么写，比如还是上边的 div:

```
div.attachEvent("onclick", function () {
    handle.call(div);
})
function handle() {
    //事件处理程序
}
```

我把事件处理的程序写在外边，然后事件的第二个参数函数里执行外部的函数并用 call 把 div 传进去，那么外部的函数里就可以用 this 了，他就通过 call 的方式让 this 指向 div 了。

练习 1: 封装兼容性方法 addEvent(), 给一个 dom 对象添加一个事件处理函数。

```
function addEvent(elem, type, handle) {
    if (elem.addEventListener) {
        elem.addEventListener(type, handle, false);
    } else if (elem.attachEvent) {
        elem.attachEvent("on" + type, function () {
            handle.call(elem);
        })
    } else {
        elem["on" + type] = handle;
    }
}
```

解析：我们封装的方法里边传三个参数，第一个是 dom 元素，第二个是事件类型，第三个是处理函数，然后里边就比较简单了，就是第二个 else if 里要把 this 指向处理

好，然后最后如果两个方法都不好使，就用 `dom.onxxx = function () {}` 这种，因为传进去是字符串形式的，所以必须加中括号。封装好之后放到代码库里。

4. 解除事件处理程序

(1) `dom.onxxx = null` 比如说一个 `div` 绑定的这个处理函数只能让他点击第一次的时候起作用，以后就不起作用了，就可以让他执行完后 `onclick` 属性就等于 `null`

`<body>`

```
<div style="width: 100px;height: 100px;background-color: red;"></div>
<script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    div.onclick = function () {
        console.log("a");
        this.onclick = null;
    }
</script>
```

`</body>`

(2) `dom.removeEventListener()`

这个和 `addEventListener` 是对应的，里边传入三个参数，第一个是事件类型，第二个是处理函数，第三个是 `false`，而且要和 `addEventListener` 的 `dom` 元素相对应，事件类型相对应，最重要的是处理函数得是一个人，所以你只能把函数提取出来，然后传参的时候放引用：

`<body>`

```
<div style="width: 100px;height: 100px;background-color: red;"></div>
<script type="text/javascript">
    var div = document.getElementsByTagName("div")[0];
    function test() {
        console.log("a");
    }
    div.addEventListener("click", test, false);
    div.removeEventListener("click", test, false);
</script>
```

`</body>`

(3) `dom.detachEvent()` (ie 独有)

这个和 `attachEvent()` 对应，里边传入两个参数，第一个是“on”+ 事件类型，第二个是处理函数，也是必须一一对应，然后处理函数得是一个人。

5. 事件处理模型——事件冒泡、事件捕获

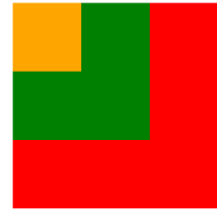
(1) 事件冒泡：结构上（非视觉上）嵌套关系的元素，会存在事件冒泡的功能，即同一事件，自子元素冒泡向父元素。（自底向上），比如说：

```
<style>
    .wrapper {
        width: 300px;
        height: 300px;
        background-color: red;
    }
    .content {
        width: 200px;
        height: 200px;
        background-color: green;
    }
    .box {
        width: 100px;
        height: 100px;
        background-color: orange;
    }
</style>
<body>
    <div class="wrapper">
        <div class="content">
            <div class="box"></div>
        </div>
    </div>
    <script type="text/javascript">
        var wrapper = document.getElementsByClassName("wrapper")[0];
        var content = document.getElementsByClassName("content")[0];
        var box = document.getElementsByClassName("box")[0];
        wrapper.addEventListener("click", function () {
            console.log("wrapper");
        }, false)
        content.addEventListener("click", function () {
```

```

        console.log("content");
    }, false)
    box.addEventListener("click", function () {
        console.log("box");
    }, false)
</script>
</body>

```



我现在写了三个方块，如图，然后在 js 里把他们选出来并绑定事件，就是打印各自的 class 名，然后当我点击红色就打印 wrapper，但是当我点击绿色的时候，他会按顺序打印出 content wrapper，然后当我点击黄色的时候，他会按顺序打印出 box content wrapper。好像他执行的顺序是漏下去的，因为我点黄色的时候绿色的也触发了，然后红色的也触发了。我们把这种现象叫事件冒泡，事件像水泡一样一层一层往上冒。这个往上冒泡不是视觉上的，是结构上的，就是如果说你事件点击到了子元素上，他就会一层一层的向父元素上传递这个事件，所以他从代码的角度来说是自底向上的。现在我把样式改一下，他就变成了这个样子：

```

.content {
    margin-left: 300px;
    width: 200px;
    height: 200px;
    background-color: green;
}
.box {
    margin-left: 200px;
    width: 100px;
    height: 100px;
    background-color: orange;
}

```



现在他们视觉上已经不嵌套了，但是结构上依然嵌套着，我现在点黄的他依然打印 box content wrapper，所以事件冒泡是存在于代码结构上的，和视觉上没关系。

（2）事件捕获：结构上（非视觉上）嵌套关系的元素，会存在事件捕获的功能，即同一事件，自父元素捕获至子元素（事件源元素）。（自顶向下）（只有谷歌实现了，IE 没有捕获事件）

首先，一个对象的一个事件类型上边绑定的一个处理函数只能遵循一种事件模型，要么冒泡，要么捕获，不可能冒泡和捕获同时存在。那么，怎么触发事件捕获呢？在

`addEventListener()` 这个方法的最后一个参数是 `false`，你把它改为 `true` 就触发了：

```
<script type="text/javascript">
    var wrapper = document.getElementsByClassName("wrapper")[0];
    var content = document.getElementsByClassName("content")[0];
    var box = document.getElementsByClassName("box")[0];
    wrapper.addEventListener("click", function () {
        console.log("wrapper");
    }, true)
    content.addEventListener("click", function () {
        console.log("content");
    }, true)
    box.addEventListener("click", function () {
        console.log("box");
    }, true)
</script>
```

捕获和冒泡正好相反，比如我点红的就打印 `wrapper`，点绿的就按顺序打印 `wrapper` `content`，点黄的就打印 `wrapper` `content` `box`，捕获就是先把最外层的先抓住，然后在往子元素一层一层捕获。但是当我点黄色的时候，先是红色捕获并执行，再是绿色捕获执行，最后黄色不是捕获，人家就正常执行。

(3) 现在我给一个对象的一个事件类型上边绑定两个处理函数，一个是捕获，一个是冒泡，那么，谁先执行呢？

第一种情况：

```
<script type="text/javascript">
    var wrapper = document.getElementsByClassName("wrapper")[0];
    var content = document.getElementsByClassName("content")[0];
    var box = document.getElementsByClassName("box")[0];
    wrapper.addEventListener("click", function () {
        console.log("wrapper");
    }, true)
    content.addEventListener("click", function () {
        console.log("content");
    }, true)
    box.addEventListener("click", function () {
        console.log("box");
    }, true)
```

```
    }, true)
    wrapper.addEventListener("click", function () {
        console.log("wrapperBubble");
    }, false)
    content.addEventListener("click", function () {
        console.log("contentBubble");
    }, false)
    box.addEventListener("click", function () {
        console.log("boxBubble");
    }, false)
</script>
```

现在我点黄的依次打印 wrapper content box boxBubble contentBubble wrapperBubble，好像是先捕获后冒泡，再看第二种情况，我把位置换一下。

第二种情况：

```
<script type="text/javascript">
    var wrapper = document.getElementsByClassName("wrapper")[0];
    var content = document.getElementsByClassName("content")[0];
    var box = document.getElementsByClassName("box")[0];
    wrapper.addEventListener("click", function () {
        console.log("wrapperBubble");
    }, false)
    content.addEventListener("click", function () {
        console.log("contentBubble");
    }, false)
    box.addEventListener("click", function () {
        console.log("boxBubble");
    }, false)
    wrapper.addEventListener("click", function () {
        console.log("wrapper");
    }, true)
    content.addEventListener("click", function () {
        console.log("content");
    }, true)
    box.addEventListener("click", function () {
```

```

        console.log("box");
    }, true)
</script>

```

我再点黄的依次打印 wrapper content boxBubble box contentBubble wrapperBubble, 我们看第三位和第四位, 咋回事? 我们刚才说了, 黄色就正常执行, 顺序的确是先捕获后冒泡, 他的顺序是红的先捕获, 然后绿的捕获, 然后是黄的正常执行, 黄的再执行, 再绿色冒泡, 再红的冒泡。黄色的是正常执行, 那么就是谁先绑定谁就先执行。

总结: 触发顺序是先捕获后冒泡。

(4) focus、blur、change、submit、reset、select 等事件不冒泡。

6. 取消冒泡和阻止默认事件

(1) 取消冒泡

有的时候我们并不希望他有冒泡功能, 比如说:

```

<body>
    <div class="wrapper"></div>
    <script type="text/javascript">
        document.onclick = function () {
            console.log("你闲的啊");
        }
    </script>
</body>

```

我们说 document 也能绑定事件, 我们让用户点文档的时候打印“你闲的啊”, 但是现在当用户点击 div 的时候也能打印“你闲的啊”, 这就说明事件天生就有, 即使你没给 div 绑定事件也能冒泡, 再比如:

```

<body>
    <div class="wrapper"></div>
    <script type="text/javascript">
        var div = document.getElementsByClassName("wrapper")[0];
        document.onclick = function () {
            console.log("你闲的啊");
        }
        div.onclick = function () {
            this.style.background = "green";
        }
    </script>
</body>

```

```
</script>
```

```
</body>
```

我们想让点 div 的时候背景颜色变绿（backgroundColor 可以简写成 background，也能识别的），点击后是变绿了，但是通过冒泡的方式把“你闲的啊”也打印了，我不想要这种效果，我想点空白区的时候在打印那句话，点 div 别打印，这么说的话冒泡就不好了，我们就要取消冒泡。在讲取消冒泡之前，我们先来了解一点东西：在每一个事件处理函数里我们可以传一个形参进去（只写一个），实参系统会帮我们传的，传进去的是一个事件对象，上边有很多属性，每一个属性都记载了这个事件发生时的关键性数据，比如说事件类型、事件时刻、鼠标坐标点等，比如说：

```
document.onclick = function (e) {  
    console.log(e);  
    console.log("你闲的啊");  
}
```

打印出来是这样的：index.html:36 MouseEvent {isTrusted: true, screenX: 720, screenY: 391, clientX: 720, clientY: 320, ...}，这个事件对象上有一个方法可以取消冒泡事件。

第一种：W3C 标准写法 `e.stopPropagation()`，比如说：

```
<body>
```

```
<div class="wrapper"></div>
```

```
<script type="text/javascript">
```

```
var div = document.getElementsByClassName("wrapper")[0];  
document.onclick = function () {  
    console.log("你闲的啊");  
}  
div.onclick = function (e) {  
    e.stopPropagation();  
    this.style.background = "green";  
}
```

```
</script>
```

```
</body>
```

你这么写的话就取消了冒泡事件，你点 div 他的颜色变绿，而且不会打印，你点空白地方他就会打印那句话，但是 ie9 以下不兼容。

第二种：ie 独有 `e.cancelBubble = true` 但是现在这个方法谷歌给实现了。

```
<body>
```

```

<div class="wrapper"></div>
<script type="text/javascript">
    var div = document.getElementsByClassName("wrapper")[0];
    document.onclick = function () {
        console.log("你闲的啊");
    }
    div.onclick = function (e) {
        e.cancelBubble = true;
        this.style.background = "green";
    }
</script>
</body>

```

这样也可以取消冒泡。

练习 2: 封装兼容性方法取消冒泡事件。

```

function stopBubble(event) {
    if (event.stopPropagation) {
        event.stopPropagation();
    } else {
        event.cancelBubble = true;
    }
}

```

解析: 调用的时候, 比如上边的, 给 div 取消冒泡, 直接 stopBubble(e) 把 e 当实参传进去即可。

(2) 阻止默认事件

默认事件有很多, 比如右键出菜单, a 标签跳转, 表单提交等。阻止默认事件有三种方法:

第一种: **return false;** 以对象属性的方式注册的事件 (即句柄绑定的事件) 才生效, 比如我们让右键不出菜单:

```

document.oncontextmenu = function () {
    console.log("a");
    return false;
}

```

这个事件就是右键出菜单事件, 那么我这么写的话右键就只打印 a, 不出菜单。

第二种: **e.preventDefault()**, W3C 标准, ie9 以下不兼容

```
document.oncontextmenu = function (e) {  
    console.log("a");  
    e.preventDefault();  
}
```

这样也可以右键只打印 a，不出菜单。

第三种：`e.returnValue = false` 兼容 IE，都能用。

```
document.oncontextmenu = function (e) {  
    console.log("a");  
    e.returnValue = false;  
}
```

这样也可以右键只打印 a，不出菜单。

练习 3：封装兼容性方法阻止默认事件。

```
function cancelHandler(event) {  
    if(event.preventDefault) {  
        event.preventDefault();  
    }else{  
        event.returnValue = false;  
    }  
}
```

解析：还是调用的时候把事件对象 e 当实参传入即可，`return false` 是特殊的，封装不进去。

应用：a 标签即使 href 里放一个空锚点站位，你点击一下他也会跳转到页面最顶端，而我们经常拿他当按钮用，就不想让他跳转，就：

```
<a href="#">www.baidu.com</a>  
<script type="text/javascript">  
    var a = document.getElementsByTagName("a")[0];  
    a.onclick = function () {  
        return false;  
    }  
</script>
```

阻止默认事件后就不跳转了，他就失去了 a 标签的默认功能。还可以把 js 代码写在 a 标签里边（就是我们最开始讲得 a 标签的协议限定符）：`www.baidu.com`，这样也能阻止默认事件。默认事件还有很多，后期遇到了再说。

7. 事件对象和事件委托

```
<body>
```

```
  <div style="width:100px;height:100px;background-color:red"></div>
```

```
  <script type="text/javascript">
```

```
    var div = document.getElementsByTagName("div")[0];
```

```
    div.onclick = function (e) {
```

```
      console.log(e);
```

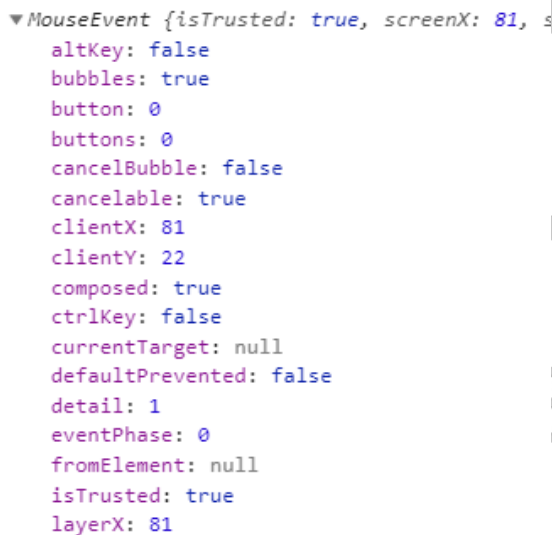
```
    }
```

```
  </script>
```

```
</body>
```

我们说系统内部会把一些关键性数据打包成一个对象传到参数 e 里边去，但是是在非 ie 浏览器下，如果在 ie 下 e 就会失效，但是他会在 window.event 上记录，所以我们得做一个兼容：

```
div.onclick = function (e) {
  var event = e || window.event;
  console.log(event);
}
```



```
▼ MouseEvent {isTrusted: true, screenX: 81, s
  altKey: false
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 81
  clientY: 22
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 1
  eventPhase: 0
  fromElement: null
  isTrusted: true
  layerX: 81
```

这个事件对象里有很多属性，如 cancelBubble 默认是 false，clientX 和 clientY (pageX 和 pageY) 是鼠标点击触发事件时鼠标点的位置，returnValue 默认为 true，其他的后边再讲，再比如：

```
<body>
```

```
  <div
```

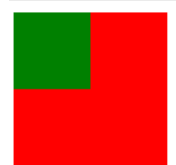
```
    class="wrapper"
```

```
  style="width:100px;height:100px;background-color:red">
```

```

<div class="box"
style="width:50px;height:50px;background-color:green;"></div>
</div>
<script type="text/javascript">
    var wrapper = document.getElementsByClassName("wrapper")[0];
    var box = document.getElementsByClassName("box")[0];
    wrapper.onclick = function (e) {
        var event = e || window.event;
        console.log(event);
    }
</script>
</body>

```



现在我点红的他会执行打印，点绿的通过冒泡也会执行打印，但是我点红色是点击到他自己身上来执行的，点绿色时，其实触发事件的那个点在绿色身上，是绿色传给你的，我们把触发的这个地方叫做事件源，事件对象上有两个属性 `target` 和 `srcElement`，他们都是记录事件源对象的，我点红色的时候他俩的值都是 `div.wrapper`，我点绿色的时候他俩的值都是 `div.box`。

`event.target` 火狐只有这个

`event.srcElement` ie 只有这个

但是这俩谷歌都有，我们来做一个兼容，还是上边的代码：

```

wrapper.onclick = function (e) {
    var event = e || window.event;
    var target = event.target || event.srcElement;
    console.log(target);
}

```

现在点红的打印 `<div class="wrapper" style="width:100px;height:100px;background-color:red">...</div>`，点绿的 `<div class="box" style="width:50px;height:50px;background-color:green;"></div>`

应用：事件委托，比如说：

```

<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>

```

```
<li>5</li>
<li>6</li>
<li>7</li>
<li>8</li>
<li>9</li>
<li>10</li>
</ul>
```

ul 下有十个 li，现在要求点哪个 li 输出哪个 li 的内容。

方法一：常规写法

```
<script type="text/javascript">
  var li = document.getElementsByTagName("li");
  var len = li.length;
  for(var i = 0; i < len; i++) {
    li[i].onclick = function () {
      console.log(this.innerText)
    }
  }
</script>
```

弊端：现在换个要求，题类似，但是一个 ul 里边有三千亿个 li，三千亿圈得转到明天，所以这种方法不行，而且我要在底下动态的加几个 li 并让他具有同样的功能就得重新循环，所以这种方法不好。

方法二：事件委托

```
<script type="text/javascript">
  var ul = document.getElementsByTagName("ul")[0];
  ul.onclick = function (e) {
    var event = e || window.event;
    var target = event.target || event.srcElement;
    console.log(target.innerText);
  }
</script>
```

解析：我们把 ul 选出来，ul 是父级，那么如果冒泡的话就是 li 冒给 ul，再想想，ul 的大小是 li 决定的，li 覆盖了 ul 的所有区域，那么我点第一个 li 的话 ul 就一定会执行，打印的是事件源对象的 innerText，事件源对象不还是第一个 li 吗？所以说这个方法很好，第一不用循环效率高，第二后续你动态加的话还是 ul 的子集就能冒泡给

ul，所以动态加的问题也解决了。

事件委托就是利用事件冒泡和事件源对象进行处理（本来是儿子干的活交给爹干）。

优点：性能好，不需要循环所有元素一个一个绑定事件；灵活，当有新的子元素时不需要重新绑定事件。

练习 4：预习后边的内容，写一个拖拽功能。

```
<body>
  <div
style="width:100px;height:100px;background-color:red;position:absolute;left:0;top:0;"></div>
  <script type="text/javascript">
    function getStyle(elem, prop) {
      if (window.getComputedStyle) {
        return window.getComputedStyle(elem, null)[prop];
      } else {
        return elem.currentStyle[prop];
      }
    }
    var div = document.getElementsByTagName("div")[0];
    var disX,
        disY;
    div.onmousedown = function (e) {
      var event = e || window.event;
      disX = event.pageX - parseInt(getStyle(this, "left"));
      disY = event.pageY - parseInt(getStyle(this, "top"));
      document.onmousemove = function (e) {
        var event = e || window.event;
        div.style.left = event.pageX - disX + "px";
        div.style.top = event.pageY - disY + "px";
      }
      document.onmouseup = function () {
        document.onmousemove = null;
      }
    }
  </script>
```

</body>

解析：思路是这样的，当 div 触发 onmousedown 的时候，即鼠标点下去的时候我们在触发 onmousemove 事件，即鼠标移动事件，然后当鼠标松开时，即 onmouseup 时，我们在清除移动事件即可。onmousemove 里，我们让 div 的 top 和 left 等于鼠标此时的位置，但是这么写有个问题，就是因为你把 left 值和 top 值设置成了鼠标点的坐标，那么当你点上去的时候方块左上角那个点就会跑到鼠标此时的位置，所以说我们还得把鼠标此时的点离左上角那个点的距离求出来，我们用 disX 和 disY 来表示他俩之间的水平距离和垂直距离，我把之前封装好的 getStyle 方法引入，然后当鼠标点下去的时候，我们来求一下距离，距离就等于鼠标此时的位置减去 div 实时的 left 值或 top 值，距离求好后我们在 onmousemove 里让他的 left 值和 top 值减去对应的距离就好了。还有一个小问题就是一开始我写的是 div.onmousemove，这时我点击拖拽他跟着走然后我迅速移开方块就跟不上了，因为我们把事件绑定到 div 身上了，只有 div 挪动才好使，而鼠标挪动是靠系统监控的，每一秒比如鼠标移动 100 下，但是 div 每一秒对事件的监听达不到 100 次，你鼠标移动太快他反应不过来，监听不到，挪出去了你都反应不过来，那就说明 div 面积太小，其实事件绑到谁身上无所谓，因为里边改的是 div 的 left 和 top，那我就把 onmousemove 绑到 document 上，document 是文档，你挪的再快你也挪不出去。

练习 5：用标准方法封装拖拽功能。

```
function drag(elem) {
    var disX,
        disY;
    elem.addEventListener("mousedown", function (e) {
        var event = e || window.event;
        disX = event.clientX - parseInt(getStyle(elem, 'left'));
        disY = event.clientY - parseInt(getStyle(elem, 'top'));
        document.addEventListener("mousemove", mouseMove, false);
        document.addEventListener("mouseup", mouseUp, false);
        stopBubble(event);
        cancelHandler(event);
    }, false)
    function mouseMove(e) {
        var event = e || event;
        elem.style.left = event.clientX - disX + "px";
        elem.style.top = event.clientY - disY + "px";
    }
}
```

```
}  
function mouseUp(e) {  
    var event = e || event  
    document.removeEventListener("mousemove", mouseMove, false);  
    document.removeEventListener("mouseup", mouseUp, false);  
}  
}
```

解析：这就是通过上边的思想封装的最标准的拖拽函数，说一下这里必须把之前封装到库里的方法都引入，不然不好使，这个你也可以放到库里，很实用的。

备注：有时候面试官会问你什么是事件捕获？除了我们之前讲的事件类型的捕获其实还有一种捕获，就是解决我们上边拖拽事件的鼠标帧频比事件监听帧频快的问题，刚才我们通过 document 解决的，他们不这么解决，还有一种方法也叫事件捕获，即 div.setCapture(); 执行完这个方法后 div 会捕获这个页面发生的所有事件，硬捕获到自己身上来，所以你在别的地方点依然算在 div 身上的，对应的有个方法 div.releaseCapture() 在适当的时候可以释放，但是这个只在 ie 浏览器上好使，也比较老旧，我们一般不用，面试的问你了你就这么说。

8. 鼠标事件（事件没有小驼峰式大写，直接拼一起即可）

click、mousedown、mousemove、mouseup、contextmenu、mouseover、mouseout、mouseenter、mouseleave

(1) click 就是鼠标点击事件，mousedown 就是鼠标按下去的事件，mouseup 就是鼠标松开事件，其实 click 就等于 mousedown 加上 mouseup。

```
document.onclick = function () {  
    console.log("click");  
}  
document.onmousedown = function () {  
    console.log("mouseDown");  
}  
document.onmouseup = function () {  
    console.log("mouseUp");  
}
```

他会按顺序打印 mouseDown mouseUp click，可见触发顺序是 mousedown 先执行，在 mouseup，再 click。

(2) contextmenu 就是右键出菜单事件，他能用到的就是右键取消菜单。

(3) mousemove 就是鼠标移动事件。

(4) `mouseover` (和 `mouseenter` 一样) 是鼠标移入事件, `mouseout` (和 `mouseleave` 一样) 是鼠标移出事件。

```
<body>
```

```
  <div style="width:100px;height:100px;background-color:red; "></div>
```

```
  <script>
```

```
    var div = document.getElementsByTagName("div")[0];
```

```
    div.onmouseover = function () {
```

```
      this.style.background = "yellow";
```

```
    }
```

```
    div.onmouseout = function () {
```

```
      this.style.background = "green";
```

```
    }
```

```
  </script>
```

```
</body>
```

这时当你鼠标移入时他的背景颜色就变黄了, 当你鼠标移出时背景颜色就变绿了。他和这个 css 里的 `hover` 效果是一样的。

(5) 用 `button` 来区分鼠标的按键

能区分鼠标左右键的只有两个事件, `mouseup` 和 `mousedown`, 其他的事件都不可以。

```
document.onmousedown = function (e) {
```

```
  console.log(e);
```

```
}
```

这个事件对象里有一个属性是 `button` 属性, 如果点击左键的话 `button` 属性是 0, 点右键是 2, 点滚轮是 1.

```
document.onmousedown = function (e) {
```

```
  if(e.button == 0) {
```

```
    console.log("left");
```

```
  }else if(e.button == 2) {
```

```
    console.log("right");
```

```
  }
```

```
}
```

`mouseup` 也是一样的, 用 `button` 来区分, 记住 `click` 是万万不能的, `click` 右键根本不起作用。

DOM3 标准规定 `click` 事件只能监听左键, 只能通过 `mousedown` 和 `mouseup` 来判断鼠标键。

(6) 如何解决 mousedown 和 click 的冲突

比如说还是上边的拖拽，但是我把 div 换成了 a 标签，现在要求点击 a 正常跳转，但是点击后拖拽就正常拖拽，就拖拽的时候不执行点击事件，点击的时候不执行拖拽，怎么实现？我们可以利用时间戳，因为你点击肯定时间短，拖拽是需要过程的，上边只是举个例子，我们写一个类似的，目的就是区分他俩就可以了。

```
var firstTime = 0,
    lastTime = 0,
    key = false;
document.onmousedown = function () {
    firstTime = new Date().getTime();
}
document.onmouseup = function () {
    lastTime = new Date().getTime();
    if(lastTime - firstTime < 300){
        key = true;
    }
}
document.onclick = function () {
    if(key) {
        console.log("click");
        key = false;
    }
}
```

解析：因为 mousedown 和 mouseup 加起来无论多久执行完 click 都会执行，那么我们用时间戳记录一下，如果 mousedown 和 mouseup 相差时间小于 0.3 秒的话，我们就让 click 执行，否则 click 不执行，定义一个开关一开始等于 false，然后 mousedown 和 mouseup 相差时间小于 0.3 秒就让 key 等于 true，把 key 作为 click 里执行语句的判断条件，如果是 true 他才会走进来，if 最后再让 key 等于 false 即可。现在点击一下 click 才会打印出来，鼠标按住不放在松开则不会打印。

9. 键盘事件

keydown、keyup、keypress

```
document.onkeypress = function () {
    console.log("keypress");
}
```

```
document.onkeydown = function () {  
    console.log("keydown");  
}  
document.onkeyup = function () {  
    console.log("keyup");  
}
```

现在我在键盘上随便按个键，比如按个 A，然后就打印 `keydown` `keypress` `keyup`，然后我按住 A 不放，他会一直循环打印 `keydown` `keypress`，等到我松开的时候才会打印 `keyup`。他们的执行顺序是 `keydown`，再 `keypress`，再 `keyup`，而且 `keydown` 和 `keypress` 不抬起就一直能触发，其实 `keydown` 和 `keypress` 差不多，只是有一点小的区别：`keydown` 可以响应任意键盘按键，`keypress` 只能响应字符类按键。

`Keypress` 返回 ASC 码，可以转换成相应字符。

```
document.onkeypress = function (e) {  
    console.log(e);  
}  
document.onkeydown = function (e) {  
    console.log(e);  
}
```

我按一下 A 键，咱们看一下事件对象的 `charCode` (ASC 码) 属性，因为 `keydown` 先触发，它里边的 `charCode` 是 0，在看 `keypress` 里 `charCode` 是 97。

我按下键，只打印一个事件对象，按下键也只打印一个事件对象，里边的 `charCode` 都是 0，`type` 都是 “`keydown`”，其实上下左右键、`Ctrl`、`shift` 等操作类的键 `keypress` 都响应不了。其实 `keydown` 能够监测到键盘上的所有按键 (`fn` 除外)，而 `keypress` 只能监测到字符类按键，就是 `asc` 码里有的按键。

`keydown` 监测键盘类事件不准，我按 A `keydown` 里有个 `which` 值是 65，我按 B `keydown` 里 `which` 值是 66，但是我按 `shift + A`，A 里边 `which` 还是 65，不能区分大小写。

而 `keypress` 里按个 A `charCode` 就是 97，按个 `shift + A`，A 里边 `charCode` 就是 65，能区分大小写。所以 `keypress` 监测字符类按键会很准。

总结：如果是字符类按键并且想区分大小写的话用 `keypress`，如果是操作类按键就用 `keydown`，例如他的上下左右对应的 `which` 值分别是 38 40 37 39 (唯一的)。这个 `which` 值其实是 108 个键，给键位排的号，一个值对应一个键，但是对应不了 `shift` 加啥。我们如果只要求哪个键是哪个键的话，那 `keydown` 就能解决一切问题，但是得现测 `which` 值。如果是 `keypress` 就可以按照 `asc` 码把他拿出来。

备注：

```
document.onkeypress = function (e) {  
    console.log(String.fromCharCode(e.charCode));  
}
```

String 上有个方法 `fromCharCode()` 可以把阿斯克码转换成对应的字母,且区分大小写,那么我这么写按啥字母就打印啥字母(区分大小写)。

10. 文本操作事件

`input`、`focus`、`blur`、`change`

(1) `input` 和 `change`

```
<body>  
    <input type="text">  
    <script>  
        var input = document.getElementsByTagName("input")[0];  
        input.oninput = function () {  
            console.log(this.value);  
        }  
    </script>  
</body>
```

我在框里输入 a 就打印 a,继续输入 b 就打印 ab,再删掉 b 就打印 a,无论是删还是新增,但凡里边内容有变化都会触发 `input` 事件。

```
input.onchange = function () {  
    console.log(this.value);  
}
```

现在我鼠标聚焦输入 abc,再失去焦点才会打印 abc,我在聚焦输入 123 然后又删掉,失去焦点他就不会打印,change 比的是鼠标聚焦和失去焦点前后是否有变化,如果有变化就触发事件,没变化就不触发。

(2) `focus` 是鼠标聚焦事件, `blur` 是失去焦点事件。

```
<input  
type="text"  
value="请输入内容"  
style="color:#999"  
onfocus="if(this.value=='请输入内容')  
{this.value='';this.style.color='#424242'}"  
onblur="if(this.value=='')  
{this.value='请输入内容';this.style.color='#999'}">
```

（为了看着方便，我把空格改成了回车，正常不能这么写）

我们用句柄的写法绑定 onfocus 和 onblur 事件，当鼠标聚焦时，如果里边的内容是默认的就让他变成空，鼠标失去焦点时如果里边是空的就让他变回默认的，再把对应的颜色改一下就是一个比较完整的输入框功能了，但是有一个 bug，假如我们就输入了“请输入内容”，失去焦点再聚焦字就没了，哈哈，很多网站首页（如新浪、58 同城等）的输入框都有这个 bug。

11. 窗体操作类（window 上的事件）

scroll、load

（1）scroll 是当滚动条一滚动他就触发事件

```
window.onscroll = function () {
    console.log(window.pageXOffset + " " + window.pageYOffset)
}
```

这么写的话当你滚动条滚动时，他就会打印滚动条滚动的距离。

思考：css 里的 fixed 固定定位在 ie6 浏览器不好使，我们可以用 js 模拟一下，就是 div 先设置 absolute 定位，然后用 window.onscroll 事件，当滚动条滚动时，让 div 的 left 值和 top 值等于他最开始的值加上滚动条的滚动距离即可。

（2）load

我们说 script 标签能阻断页面，所以我们必须把 script 标签写在下面才能把上边的标签读出来，如果我们把 script 标签写在上边的话，我们在 js 里选标签就选不出来，因为页面还没有渲染到那儿呢就卡死了，就执行 js 了，但是有些“高手”会这么写：

```
<body>
    <script>
        window.onload = function () {
            var div = document.getElementsByTagName("div")[0];
        }
    </script>
    <div></div>
</body>
```

这么写的话不但能选出来，还能在 js 里操作 div 呢，听我给你慢慢讲原因哈，我们在请求网址的时候，会先把代码下载下来然后一行一行去加载，他的顺序是这样的（下节还会细讲）：html 和 css 会并行一起解析的，html 在解析的时候会形成一个 domTree，css 在解析的时候会形成 cssTree，就是树形图，dom 树和 css 树拼在一起会形成渲染树 renderTree，然后通过渲染树绘制页面，那什么时候把节点挂到树上呢，比如一个 img 标签，他识别是图片标签的话就立马挂到树上，然后再开启一个线程异步的下载图

片，我们把 script 标签写在下边的好处就是等标签解析完 js 就可以操作他们了，没必要等他下载完，而 window.onload 是等到渲染树生成之后，整个文档全部解析完后，所有文字图片全部下载完之后，一切就绪之后，window.onload 才执行，window 触发 onload 的时候就代表所有自动的过程都完事了，所以他的效率最低了，我们一般不用。但是他可以提醒你什么时候是整个页面都完事了，等待用户交互体验了。我们可以把一些广告事件放到 load 里，注意主程序千万别放在里边，太 low 了。

json、异步加载、时间线

1. json

JSON 是一种传输数据的格式（以对象为样板，本质上就是对象，但用途有区别，对象就是本地用的，json 是用来传输的）。

解释：我们说前后端得互相配合，得互相传送数据，那么传的这个数据就得有固定的格式，因为得保证互相都能认识，在以前他们用的是 xml 格式，和 html 唯一不同的是：xml 可以自定义标签，如：

```
<student>
  <name>deng</name>
  <age>40</age>
</student>
```

可以自定义标签之后，他就和对象差不多了，但是现在我们不用他了。现在我们传的就是对象：

```
{
  name:"deng",
  age:123
}
```

本来他就是一个对象，但是应用到数据传输里我们给他起名叫 json，为了和普通对象区分开，人们规定 json 格式必须这么写：

```
{
  "name":"deng",
  "age":123
}
```

我们知道对象的属性名可以加双引号也可以不加，后来人们规定：如果这个对象是 json 格式的属性名就必须加双引号，现在人们传输数据的格式全用的是 json 格式的，其实传的也不是对象，而是文本格式的字符串，前后端传的都是字符串，只不过是 json 格式的字符串，然后前后端都有特殊的语法把字符串再转换为 json 对象格式的，前端有

两个语法：

JSON.parse() 可以把字符串转换为 json 对象；

JSON.stringify() 可以把 json 对象转换为字符串。比如说：

```
var obj = {
  name:"deng",
  age:123
}
```

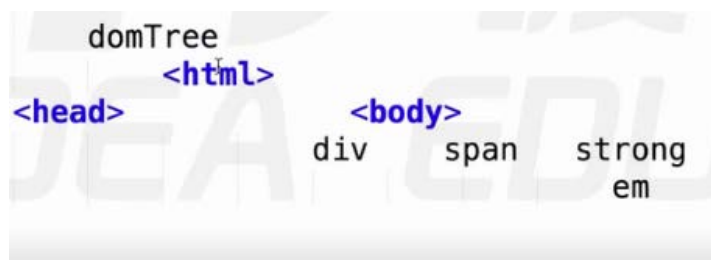
现在 JSON.stringify(obj) 就得到 "{name: 'deng', 'age': 123}"，这样就可以把 json 对象转换为字符串传给后端，你可以发现他自动的给属性名加引号了，然后我 var str = JSON.stringify(obj)；JSON.parse(str) 就得到 {name: "deng", age: 123}，这种方法就可以把后端传过来的数据转换为对象供我们使用。

2. 我们先来深入了解一下浏览器解析的过程，我写一个完整的页面哈：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    div{
      width: 100px;
      height: 100px;
      background-color: red;
    }
  </style>
</head>
<body>
  <div></div>
  <span></span>
  <strong>
    <em></em>
  </strong>
</body>
```

</html>

一个页面在展示给我们的时候证明他已经绘制完了，绘制页面就是浏览器里有一个渲染引擎，他会一行一行（以像素为高度）的绘制页面，那例如上边的，css 里的 div 和 html 里的 div 是怎么联系到一起的呢？其实在内核的内部是不认识代码的，不会让他俩直接产生联系，他的加载顺序是这样的，浏览器里的内核会对页面进行一步一步检索，首先识别 HTML 代码，然后形成一个 dom 树，即 domTree，他会一行一行识别 html 代码然后把他挂到一棵树上，这个 dom 树是这个样子：



首先树的顶端是 html 标签，然后 html 下边左侧是 head 标签，右侧是 body 标签，然后 head 底下可能还有其他标签，body 底下有 div、span、strong，strong 下边有 em，这是一个树形结构，浏览器会根据节点的排列方式绘制一个 dom 树，而且绘制的原则符合深度优先原则，深度优先就是先把 html 挂到树上，然后是 head 标签，然后他不会立马看 body，他会先看 head 底下有啥标签，就是深度的把一条枝干全部走完之后再广度优先，就是他先把 head 底下的所有节点都挂到树上后才会看 body。然后比如说有个 img 标签，他一读到 img 标签就会立马把他挂到树上，并不会等到图片下载完之后才挂到树上，他一读到就挂起来，然后异步的去下载图片（就是同时的）。所以说整个页面先解析完后加载完，解析完一定在下载完之前。

dom 树形成后，等着 css 树形成，cssTree 就是根据我们写的 css 形成的一个和 domTree 一样的树形结构，他和 dom 树里的节点都是一一对应的，也是深度优先，css 树形成后会和 dom 树拼在一起形成一个新的渲染树 renderTree，他的样子和 dom 树一样，只不过在节点后边加上他的样式，渲染树形成后渲染引擎才会根据渲染树的每一条规则绘制页面。

后期我们通过 js 给 dom 节点删除、增加，还有他的宽高变化位置变化、display 等都会让渲染树重排（reflow）并重新绘制整个页面，这个效率上是最低的。

但是比如你改个背景色，字体颜色只是部分改变，不影响后边的节点，他就会重绘（repaint），这个效率也会降低，但是比重排效率高。

3. 异步加载 js

（1）思考：为啥 js 能阻断 HTML 和 css 的加载线

js 是同步加载的，就是读到 js 的时候就阻断页面了，等我 js 加载完并且执行完的时候下边的内容才可以执行，那为什么 js 的下载过程和执行过程不能和 html 和 css 并

行去做呢？因为 js 会修改 html 和 css，你还没绘制完呢我就修改了，这是不行的。就跟 js 为啥是单线程一样，假如是多线程的话，一个线程增加节点，一个线程删除这个节点，那到底听谁的呢？就没法执行了。

(2) 同步加载 js 的缺点：加载工具方法没必要阻塞文档，过多 js 加载会影响页面效率，一旦网速不好，那么整个网站将等待 js 加载而不进行后续渲染等工作。

有的工具包需要按需加载，用到时加载，不用不加载。

解释：有的时候这个 js 里根本没有操作主页面，就是对数据的一些处理，或者里边全部是工具类的方法，不调用就不会执行，我们就希望他异步加载，就跟 html 和 css 并行加载，不阻断页面。

(3) js 异步加载的三种方案

第一种 defer 异步加载：但要等到 dom 文档全部解析完才会被执行，只有 ie 能用，也可以把代码写在 script 标签里。

```
<script type="text/javascript" src="toos.js" defer="defer"></script>
```

这样在后边加上 defer="defer" 就变成异步加载了，你也可以把 src 删掉，把代码写到标签里。这种标签执行的时刻发生在整个页面解析完毕时，就是 dom 树生成的时候。

第二种 async 异步加载，加载完立马就执行，async 只能加载外部 js 脚本，不能把 js 写在 script 标签里。(W3C 标准方法，ie9 以下不能用)

```
<script type="text/javascript" src="toos.js" async="async"></script>
```

这两种方法都能实现异步加载，执行时不会阻断页面。那怎么实现兼容呢？你不能又写 defer="defer" 又写 async="async" 那样会崩的，也不能写两个标签引入一个 js 文件，那样里边的数据会乱的，因为两种方法的执行时刻不一样，真正兼容性写法在下边。

第三种 兼容性方法：创建 script，插入到 dom 中。这样也能实现异步加载，并且是按需加载。

```
<script type="text/javascript">
  var script = document.createElement("script");
  script.type = "text/javascript";
  script.src = "index.js";
  document.head.appendChild(script);
</script>
```

这里边写到第三行的时候他就开始下载外部的 js 文件了，但是如果你不写第四行的话他永远不会执行。你必须把他插入到页面里的时候他才会解析并执行，否则只是下载完，啥都不干。比如说现在我在 index.js 里写上：

```
alert("a");
```

他就会正常执行，这样就实现了异步加载的过程。
但是有一个问题，比如说 index.js 里是一个方法：

```
function test() {  
    alert("a");  
}
```

然后主页面里我这么写：

```
<script type="text/javascript">  
    var script = document.createElement("script");  
    script.type = "text/javascript";  
    script.src = "index.js";  
    document.head.appendChild(script);  
    test();  
</script>
```

这么写会报错的，告诉你 test is not defined。因为这里边第三行就下载，下载就需要发一个请求，然后回归资源，这是需要一个过程的，在这个过程中他就把后两行执行完了，所以报错是因为当执行 test 的时候还没有下载完，因为下载是异步下载的，并不是阻塞在那里等你下载完才执行，不是那样的。那怎么解决啊？我什么时候才可以执行呢？

```
<script type="text/javascript">  
    var script = document.createElement("script");  
    script.type = "text/javascript";  
    script.src = "index.js";  
    script.onload = function () {  
        test();  
    }  
    document.head.appendChild(script);  
</script>
```

我们 load 这个事件并不是只有 window 上有，script 等很多节点都有，我们 script 触发 load 事件的时候就证明他已经下载完了，这个时候我们再去执行 test 即可。但是 ie 浏览器上就 script 上没有 load 事件，他自己提供了一套方法：ie 里 script 上有个状态码 readyState，其实他就是一个属性，里边存值了，一开始是“loading”，加载完之后里边的值会变成“complete”或“loaded”，然后他还提供一个事件，onreadystatechange，就是当里边的状态码发生改变的时候触发这个事件：

```
<script type="text/javascript">
```

```

var script = document.createElement("script");
script.type = "text/javascript";
script.src = "index.js";
script.onreadystatechange = function () {
    if(script.readyState == "complete" || script.readyState ==
        "loaded") {
        test();
    }
}
document.head.appendChild(script);
</script>

```

这个是 ie 独有的方法，我们得有一个事件监听，如果状态码发生改变的话得有一个监听者来触发这个事件。

练习 1: 封装方法 loadScript，解决异步加载的兼容性问题。

```

function loadScript(url, callback) {
    var script = document.createElement("script");
    script.type = "text/javascript";
    if (script.readyState) {
        script.onreadystatechange = function () {
            if (script.readyState == "complete" || script.readyState ==
                "loaded") {
                callback();//备注
            }
        }
    } else {
        script.onload = function () {
            callback();//备注
        }
    }
    script.src = url;
    document.head.appendChild(script);
}

```

解析：里边传入俩参数，第一个是外部文件脚本，第二个是调用函数。但是这么写有个小的问题就是假如我网速特别快，执行主程序后立马 url 就下载完了，readyState

瞬间就变成“complete”状态，那绑定的事件就监听不了了，所以我们得把 `script.src = url`; 这行移到绑定事件之后，先把事件绑上然后你再加载文件。那怎么调用呢？外部文件不变哈，比如说 `loadScript("index.js", test)`; 这么写会报错，告诉你 `test is not defined`，为啥啊？因为他的执行顺序是先解析一下 `function`，不会去看里边是啥，然后就执行 `loadScript("index.js", test)`; 去导致 `function` 的执行，那在这个时候就塞进去一个 `test` 肯定会报错啊，文件还没加载过来呢，传 `test` 发生在文件加载过来之前，在此时 `test` 是未定义的，解决办法：

第一种：

```
loadScript("index.js", function () {  
    test();  
});
```

调用的时候放一个匿名函数，他是一个引用，在解析的时候他也不知道里边装的是啥，执行的时候才会看，里边让 `test` 执行就可以了，按需加载。

第二种：调用的时候这样传：`loadScript("index.js", "test()");`，然后 `function` 里把备注那两行都改成 `eval(callback)`；因为 `eval` 会把字符串当代码执行，但是不建议这么用。

第三种：我们把外部的 `js` 文件改成对象形式的：

```
var tools = {  
    test: function () {  
        alert("a");  
    }  
}
```

调用的时候 `loadScript("index.js", "test")`；，然后把 `function` 里备注的两行都改为 `tools[callback]()`；这就调用了 `tools` 里的 `test`，他是一个方法，然后方法执行。

4. 浏览器加载时间线

这个时间线记载了页面出生那一刻开始，浏览器按照顺序做的事儿，总共分为十步：（建议背过，为后期打下基础）

（1）创建 `Document` 对象，开始解析 `web` 页面。解析 `HTML` 元素和他们的文本内容后添加 `Element` 对象和 `Text` 节点到文档中。这个阶段 `document.readyState = 'loading'`。

（2）遇到 `link` 外部 `css`，创建线程，进行异步加载，并继续解析文档。

（3）遇到 `script` 外部 `js`，并且没有设置 `async`、`defer`，浏览器同步加载，并阻塞，等待 `js` 加载完成并执行该脚本，然后继续解析文档。

（4）遇到 `script` 外部 `js`，并且设置有 `async`、`defer`，浏览器创建线程异步加载，

并继续解析文档。对于 `async` 属性的脚本，脚本加载完成后立即执行。（异步禁止使用 `document.write()`，因为当你整个文档都加载完的时候，再调用 `document.write()`，会把之前所有的文档流都清空，用它里面的文档代替）

比如说：

```
<body>
  <div style="width:100px;height:100px;background-color:red;"></div>
  <script type="text/javascript">
    document.write("a");
  </script>
</body>
```

正常是这种效果：



但是假如等到页面全部加载完时：

```
<body>
  <div style="width:100px;height:100px;background-color:red;"></div>
  <script type="text/javascript">
    window.onload = function () {
      document.write("a");
    }
  </script>
</body>
```

他就会把之前所有文档都清空，用它里边的文档代替，异步加载会达到同样的效果，所以我们一般别用。

(5) 遇到 `img` 等（带有 `src`），先正常解析 `dom` 结构，然后浏览器异步加载 `src`，并继续解析文档。

(6) 当文档解析完成（`domTree` 建立完毕，不是加载完毕），`document.readyState = 'interactive'`。

(7) 文档解析完成后，所有设置有 `defer` 的脚本会按照顺序执行。（注意与 `async` 的不同，但同样禁止使用 `document.write()`）；

(8) 文档解析完成后，`document` 对象触发 `DOMContentLoaded` 事件，这也标志着程序执行从同步脚本执行阶段，转化为事件驱动阶段。

- (9) 当所有 `async` 的脚本加载完成并执行后、`img` 等加载完成后（页面所有的都执行加载完之后），`document.readyState = 'complete'`，`window` 对象触发 `load` 事件。
- (10) 从此，以异步响应方式处理用户输入、网络事件等。

现在比如说：

```
<script type="text/javascript">
    console.log(document.readyState);
</script>
```

应该打印的是 `loading`，因为 `interactive` 得是 `dom` 树形成之后才会变成这个，而 `script` 标签也属于 `dom` 节点啊，读到这句话的时候 `dom` 树的 `script` 才解析到一半。再比如：

```
<script type="text/javascript">
    window.onload = function () {
        console.log(document.readyState);
    }
</script>
```

此时应该打印 `complete`，因为此时所有页面都加载完了，那我就想看打印 `interactive` 咋办？

```
<script type="text/javascript">
    document.onreadystatechange = function () {
        console.log(document.readyState);
    }
</script>
```

我们上边说了有一个监听事件 `onreadystatechange`，一旦状态发生改变就触发这个事件，此时就把 `interactive` 和 `complete` 都打印出来了。

还有一个 `DOMContentLoaded` 事件，我们来试一下，这个只有在 `addEventListener` 这种绑定方式上才好使：

```
<script type="text/javascript">
    console.log(document.readyState);
    document.onreadystatechange = function () {
        console.log(document.readyState);
    }
    document.addEventListener("DOMContentLoaded", function () {
        console.log("a");
    }, false)
</script>
```

此时就依次打印 loading interactive a complete, 因为 DOMContentLoaded 事件是文档解析完成就触发的。

思考：为什么我们把 script 写在文档的最下边是最合适的？因为 script 标签是要操作 dom 的，你写在下面就证明上边的 dom 已经处理完事了，有一个小问题就是写在下边 dom 树还没有创建完，只剩他自己了，但是 script 标签是不会处理自己的，所以写在下边是比较科学的。其实最好的方法就是等文档全部解析完在执行 js：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <script type="text/javascript">
    document.addEventListener("DOMContentLoaded", function () {
      var div = document.getElementsByTagName("div")[0];
      console.log(div);
    }, false)
  </script>
</head>
<body>
  <div style="width:100px;height:100px;background-color:red;"></div>
</body>
</html>
```

这样你把 script 写在上边就可以了，千万别用 window.onload，太慢了，他要等文档全部加载完才执行，假如有 1kb 没下载下来他都不会执行的，而这个 DOMContentLoaded 等文档解析完就执行了，效率是最高的，不过还是建议把 script 写在下面哈！

正则表达式、真题讲解

1. 课前补充

(1) 转义字符 “\”（反斜杠）

现在比如说 `var str = "abcdef"`；然后我想给字符串里加一个双引号怎么办？你不能直接加，直接加系统会识别成语法的，你就加一个转义字符，然后把引号写在后面即可：

`var str = "abcd\"ef"`；现在你访问 str 就得到"abcd"ef"，就是假如说你想给字符

串里加一个特殊符号，而且这个符号还有特殊语法你就给前边加一个转义字符，转义字符的作用就是把它后一位转化为字符串，转义字符自己不会显示的，会和自己的后一位结合在一起。那我想在字符串里打一个转义符号就 `var str = "abcd\\ef"`；你再访问 `str` 就得到 `"abcd\ef"`。

转义字符后边还可以跟字母，他配合特殊的字母就会有特殊的语法意义：

`\n`：换行功能，这个 `n` 就不代表字符串 `n` 了，例如 `var str = "abcd\nef"`；你再访问 `str` 就得到两行的字符串：

```
"abcd
ef"
```

但是 `document.write()` 这个方法是换不了行的，你当文档流输入进去，文档里是识别不了回车的，回车都变成文字分隔符了，必须 `console.log(str)` 才可以。

`\r`：行结束符，代表行结束，你看不出来效果，`var str = "abcd\r\nef"`；访问 `str` 得：

```
"abcd
ef"
```

`\t`：table 功能，制表符缩进，`var str = "abcd\t ef"`；访问 `str` 得到 `"abcd ef"`，继续 `var str = "abcd\t\t ef"`；，在访问就得到 `"abcd ef"`，一般一个 table 代表四个空格。

\后边还可以跟很多字母，待会再说。

(2) 多行字符串

比如说我要写一个 html 结构，在编程的时候让字符串换行：

```
var test = "
<div></div>
<span></span>
"
```

这么写会报错的，而且是低级的语法错误，系统规定字符串是不能多行显示的，但是加上转义字符\就可以了：

```
var test = "\
<div></div>\
<span></span>\
"
```

转义字符会把文本形式的回车转义掉，让他不再是回车，现在就可以实现了，访问 `test` 就得到 `"<div></div>"`，还有一种方法就是通过字符串拼接的方式也可以实现：

```
var test =
```

```
"<div></div>" +
"<span></span>"
```

访问 test 还是 "<div></div>"。但是这么写比较 low。

2. 正则表达式 RegExp 的作用：匹配特殊字符或有特殊搭配原则的字符的最佳选择。

解释：比如说一个字符串里有很多的特殊字符串片段，我们想知道这个字符串里有多少个这样的字符串片段，用原生方法就很难做到了；再比如说一个邮箱是 "cheng.ji@alibaba-inc.com"，我们要判断这个邮箱是否符合格式，就要对字符串进行判断，用原生的方法也很难办；再比如让用户输入一个 xxyy 式的字符串，即头两位必须相同，后两位必须相同，按原来的方法也非常难判断……以上所有功能最方便的就是利用正则表达式，正则表达式一行就能给他处理了，他就浓缩了很多规则，我们今天就来看一下它的规则。

3. 正则表达式的两种创建方式

正则表达式是规则对象，也是一种对象，只不过对象里定义了一些规则。

(1) 直接量：比如说 `var reg = /abc/`；他的格式就是两个斜杠（问号下边的这个），里边写上 abc，这就代表了正则表达式的 abc，就是我匹配字符串的规则是 abc。比如：

```
var reg = /abc/;
var str = "abcd";
```

然后我在控制台 `reg.test(str)` 就得到 true，正则表达式上有一个方法叫 test 测试把 str 放进去，它的功能就是我要测验一下你这个字符串里含不含有我规定的片段，我规定的是 abc 字符片段，这个字符必须是 abc，三个字符必须得挨着，而且顺序不能乱，还都得是小写。

```
var reg = /abce/;
var str = "abcde";
```

然后我在控制台 `reg.test(str)` 就得到 false，因为字符串里没有我规定的字符片段，你必须挨着，不挨着不行。

然后字面量的方式第二个斜杠后面还可以跟上属性，后边可以加个 i，还可以加 g，还可以加 m，这是三个属性，如果后边加上 i 这个正则表达式就忽略大小写了：

```
var reg = /abcd/i;
var str = "ABCDE";
```

我在控制台调用 `reg.test(str)` 就得到 true，但是你不加 i 就是 false。还有两个属性 m 和 g 这三个属性可以单独写，也可以写在一起且顺序不固定，随便写。

(2) 构造函数创建方法：如

```
var reg = new RegExp("abc", "i");
var str = "ABCDE";
```

里边第一个参数传入规则，得是字符串形式的，第二个参数传入属性，没有就不写，当然也得是字符串形式的，现在 `reg.test(str)` 就是 `true`。

当然你也可以这么写：

```
var reg = /abc/m;  
var reg1 = new RegExp(reg);
```

你可以把创建好的正则表达式直接当参数传进去，现在你访问 `reg` 就得到 `/abc/m`，访问 `reg1` 也是 `/abc/m`，但是他俩是两个人，比如你 `reg.abc = 123`，你在访问 `reg1.abc` 就是 `undefined`。但是比如说：

```
var reg = /abc/m;  
var reg1 = RegExp(reg);
```

你前边不加 `new` 的话也可以，他俩就指向同一个引用了，比如你 `reg.abc = 123`，你在访问 `reg1.abc` 就是 `123`。

4. 修饰符（就是 `i`、`m`、`g` 三个属性）

(1) `i`：上边讲过了，就是匹配时忽视大小写。

(2) `g`：执行全局匹配。比如说：

```
var reg = /ab/;  
var str = "abababab";
```

现在 `str.match(reg)` 就得到 `["ab"]`，这个字符串方法就是把符合正则表达式规则的字符串片段选出来放到数组里，但是这时只有一个，我现在加个 `g`：

```
var reg = /ab/g;  
var str = "abababab";
```

`str.match(reg)` 就得到 `["ab", "ab", "ab", "ab"]`，他把找到的所有方法都给你返回，匹配完一个后继续往后看。

(3) `m`：执行多行匹配，我们先来看一例子：

```
var reg = /a/g;  
var str = "abcdea";
```

现在 `str.match(reg)` 就得到 `["a", "a"]`，有 `g` 匹配了两个 `a`，再比如：

```
var reg = /^a/g;  
var str = "abcdea";
```

这个前边加上 `^` 就代表以 `a` 开头的这个 `a`，匹配的这个 `a` 必须是字符串的开头，那么 `str.match(reg)` 就得到 `["a"]` 一个 `a`，虽然 `g` 是全局匹配，但是后边的 `a` 不符合要求。

```
var reg = /^a/g;  
var str = "abcde\na";
```

现在我在 `a` 前边加一个换行，就变成多行字符串了，继续 `str.match(reg)` 得到 `["a"]`

还是一个，因为此时你没有多行匹配的功能，直到你在正则表达式后边加上一个 `m`，他就能多行匹配了：

```
var reg = /^a/gm;
var str = "abcde\na";
```

现在 `str.match(reg)` 就得到 `["a", "a"]`。这就是 `m` 的基本意义。

小结：`reg.test()` 只能判断字符串有没有符合要求的片段，返回结果只有 `true` 和 `false`，而 `str.match()` 可以把符合要求的片段放到数组里给你返回，更直观一些。

5. 方括号（表达式）

比如说 `var str = "12309u9873zpoixcuypiouqwer"`；我瞎写了一个字符串，现在我要匹配这个字符串有没有三个数字相连接的情况。那么正则表达式就得这么写：

```
var reg = /[1234567890][1234567890][1234567890]/g;
```

这一个方括号代表一位，三个方括号就代表三位，括号里写的是范围，就是在方括号里任取一个值（只能取一个），三个范围组成一个正则表达式，那现在能匹配几位？说一下，比如 123 匹配完就不会回来再看了，他就从 0 往后看，所以 `str.match(reg)` 得到 `["123", "987"]`，他不会反过来调过去看的。

```
var reg = /[ab][cd][d]/g;
var str = "abcd";
```

这个第一位只能是 a 或 b，第二位只能是 c 或 d，第三位只能是 d，所以 `str.match(reg)` 得到 `["bcd"]`。这个方括号里代表一个区间，还可以这么写：

```
var reg = /[0-9A-Za-z][cd][d]/g;
```

这第一个的区间就是 0 到 9，A 到 Z，a 到 z，还可以简写：

```
var reg = /[0-9A-z][cd][d]/g;
```

因为他会按阿斯克码去排，所以 A-z 把 52 个字母都包括了（大小写各 26 个），所以第一个取值范围是 0-9 加上 52 个字母。

```
var reg = /[0-9A-Za-z][cd][d]/g;
var str = "ablcd";
```

`str.match(reg)` 得到 `["lcd"]`。

```
var reg = /^[^a][^b]/g;
var str = "ablcd";
```

`^` 放到表达式里意义就不一样了，代表非的意思，就是第一位不是 a，第二位不是 b，那么 `str.match(reg)` 就得出 `["b1", "cd"]`。表达式还有一种写法：

```
var reg = /(abc|bcd)[0-9]/g;
var str = "abc2";
```

小括号也有优先计算的作用，`|` 代表或，正则里是一个竖线，这就代表 abc 加个数字或

bcd 加个数字，那么 `str.match(reg)` 就得到 `["abc2"]`。

6. 元字符

`\w`: `\w` 也代表一位，他就完全等于 `[0-9A-z_]`，0 到 9，大 A 到小 z，再加下划线。他就是表达式的翻译版本。

`\W`: 他代表非 `\w`，`[\^w]`，取它的补集。

```
var reg = /\wcd2/g;
```

```
var str = "bcd2";
```

`str.match(reg)` 得到 `["bcd2"]`。

```
var reg = /\Wcd2/g;
```

```
var str = "bcd2";
```

现在第一位就是除了 `\w` 以外，那就匹配不出来了，`str.match(reg)` 得到 `null`。

```
var reg = /\Wcd2/g;
```

```
var str = "b*cd2";
```

现在 `str.match(reg)` 就得到 `["*cd2"]`。

`\d`: 代表 `[0-9]`。

`\D`: 代表 `[\^d]`。

```
var reg = /\d\d\d/g;
```

```
var str = "123";
```

`str.match(reg)` 得到 `["123"]`，还可以这么写：

```
var reg = /[w\d]/g;
```

```
var str = "s";
```

表达式里也可以写元字符，而且可以写多个，他的意思就是匹配一位，范围的话只要满足里边任意一个元字符的规则即可。`str.match(reg)` 得到 `["s"]`。

`\n`: 换行符

`\f`: 换页符

`\r`: 回车符

`\t`: 制表符

`\v`: 垂直制表符

```
var reg = /\tc/g;
```

```
var str = "abc cdefgh";
```

这个就是匹配前边有制表符的 `c`，`str.match(reg)` 得到 `null`，这个他不认识，你必须在字符串里加上 `\t` 才可以。

```
var reg = /\tc/g;
```

```
var str = "abc\tcdefgh";
```

`str.match(reg)`就得到[" c"], 其他几个也是一样的。

`\s`: 空白字符, 能展示空白的一些字符, 他的范围是`[\t\n\r\v\f]`, 最后加一个空格, 你在正则表达式里打一个空格他就真的是匹配一个空格。

`\S`: 代表`[\s]`

`\b`: 单词边界

`\B`: 非单词边界

```
var reg = /\bc/g;
```

```
var str = "abc cde fgh";
```

现在我这个字符串里有三个单词, 那么他就有 6 个单词边界, 我现在匹配 c, 但是要求这个 c 的前边得是单词边界, `str.match(reg)`就得到["c"]。

```
var reg = /\bcde\b/g;
```

```
var str = "abc cde fgh";
```

这就要求 cde 前后都得有单词边界, `str.match(reg)`就得到["cde"]。

```
var reg = /\bcde\b/g;
```

```
var str = "abc cdefgh";
```

现在我把空删除了, `str.match(reg)`就得到 null。

```
var reg = /\bcde\B/g;
```

```
var str = "abc cdefgh";
```

cde 要求前边单词边界, 后边非单词边界, `str.match(reg)`就得到["cde"]。

`\uxxxx`: 查找以十六进制 xxxx 规定的 Unicode 字符 (十六进制从 0000 到 ffff, 包含了所有字符, 汉字也在里边)

```
var reg = /\u8001\u9093\u8eab\u4f53\u597d/g;
```

```
var str = "老邓身体好";
```

现在 `str.match(reg)`就得到["老邓身体好"], 这是我在网上用编码器转的, Unicode 也可以写区间:

```
var reg = /[\u0000-\uffff]/g;
```

```
var str = "老邓身体好";
```

从 0000 到 ffff 代表了所有字符, `str.match(reg)`就得到 ["老", "邓", "身", "体", "好"], 其实代表所有字符还可以这么写, 比如:

```
var reg = /[\d\D]/g;
```

一个小 d 一个大 D 就代表一切, 即 `\d` 和除了 `\d` 的所有。当然 `[\s\S]` 等也可以代表一切。

∴ 查找除了换行和行结束符的单个字符, 即 `[\r\n]`, 除了 `\r\n` 以外的所有。

```
var reg = /. /g;
```

```
var str = "老邓 身体好";
```

`str.match(reg)`就得到 ["老", "邓", " ", "身", "体", "好"]

7. 量词

n+: 这个 n 是变量, 可以代表任何东西, +代表这个变量可以重复出现一次到无数次。

n*: *代表可以重复出现 0 次到无数次。

```
var reg = /\w+/g;
```

```
var str = "abc";
```

这就是 \w 可以出现一次到多次, 那么 `str.match(reg)`就得到 ["abc"], 他就说明了 \w 出现了三次。

```
var reg = /\w*/g;
```

```
var str = "abc";
```

这个 \w 可以出现 0 次到多次, `str.match(reg)`得到 ["abc", ""], 最后有个空串是因为 g 是全局匹配, 然后他把 abc 匹配出来之后, 光标就在 c 后边, 这个 c 后边还有一段逻辑上的距离, 因为是*, *如果变成 0 的话, \w0 就是没有, 正则里就是一个空, 所以他匹配了一个空出来。一开始匹配*是 3, 第二次匹配*就是 0 了。

```
var reg = /\d*/g;
```

```
var str = "abc";
```

`str.match(reg)`就得到 ["", "", "", ""], 一开始光标在 a 前边匹配一个空, 然后看 a, 匹配不了, 再到 b 前边再匹配一个空……那么有多少个光标定位点就匹配多少个空, 因为最后一个空是逻辑距离匹配出来的。

那上一个例子中 \w 为啥不匹配四个空加 abc 呢? 他不会那么识别的, 一旦 \w 能识别出值的话, 因为 abc 都能识别, 他会一连串先把 abc 能识别的识别了, 到最后*才成 0, 能匹配多就不匹配少。

```
var reg = /\w+/g;
```

```
var str = "aaaaaaaaaaaaa";
```

`str.match(reg)`就得到 ["aaaaaaaaaaaaa"], 能多就不少, 这叫贪婪匹配原则。

n?: 这个重复出现的范围是 0-1 次。

```
var reg = /\w?/g;
```

```
var str = "aaa";
```

`str.match(reg)`就得到 ["a", "a", "a", ""], 最后*变成 0, 匹配一个空。

n{x}: 就是 n 要重复出现 x 次, 范围定死了。

```
var reg = /\w{3}/g;
```

```
var str = "aaaaaa";
```

\w 要重复出现 3 次, 三个三个匹配, `str.match(reg)`就得到 ["aaa", "aaa"]。

n{x,y}: n 可以重复出现 x 到 y 次。

```
var reg = /\w{3,5}/g;
var str = "aaaaaaaaaaaaa";
```

\w 可以重复出现 3 到 5 次，str.match(reg) 得["aaaaa", "aaaaa", "aaaa"], 先匹配 5 个，最后匹配 4 个，能多就不少。

n{x,}: 可以重复出现 n 到无数次。

量词是 n 去乘以这个量词，例如上边的，是 \w 乘以量词，几个 \w。

8. 开头符和结尾符

```
var reg = /^abc/g;
var str = "abcde";
```

这个把 ^ 写到方括号外边就代表匹配以 abc 开头的 abc，str.match(reg) 就得到["abc"]。

```
var reg = /de$/g;
var str = "abcde";
```

后边加上 \$ 就是匹配以 de 结尾的 de，str.match(reg) 得到["de"]。

```
var reg = /^abc$/g;
var str = "abcabc";
```

str.match(reg) 得到 null，这个的意思是以 abc 开头并且以这个 abc 结尾，所以这么写正则表达式就把字符串限定死了。

```
var reg = /^abc$/g;
var str = "abc";
```

str.match(reg) 得到["abc"]。

备注：开头和结尾会受到 m 的影响。

练习 1: (阿里巴巴笔试题) 写一个正则表达式，检验一个字符串首尾是否含有数字。

```
var reg = /^\\d|\\d$/g;
```

解析：这个比较简单，因为他没说都，所以首尾有一个地方含有数字就行了，中间用个或，两边开头结尾都匹配 \\d 即可。

变式：检验一个字符串首尾是否都含有数字。

```
var reg = /^\\d[\\s\\S]*\\d$/g;
```

解析：中间得用一个区间出现 0 到很多次拉伸一下，区间里只要能代表任意字符就行。

9. 正则表达式上的属性

- (1) **global**: 正则表达式上是否具有标志 g，例如上边的，reg.global 就是 true。
- (2) **ignoreCase**: 正则表达式上是否具有标志 i。
- (3) **multiline**: 正则表达式上是否具有标志 m。
- (4) **source**: 返回正则表达式的内容。
- (5) **lastIndex**: 这个属性很重要，待会再讲。

10. 正则表达式上的方法

- (1) `test()`: 检测字符串中指定的值, 返回 `true` 或 `false`。
- (2) `exec()`: 这也是一种匹配方法, 比如说:

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
```

我们来看结果:

```

▶ [\"ab\", index: 0, input: \"abababab\", groups: undefined]
▶ [\"ab\", index: 2, input: \"abababab\", groups: undefined]
▶ [\"ab\", index: 4, input: \"abababab\", groups: undefined]
▶ [\"ab\", index: 6, input: \"abababab\", groups: undefined]
null
▶ [\"ab\", index: 0, input: \"abababab\", groups: undefined]
> |

```

他每次匹配智能匹配一个“ab”, 这里边 `index` 第一次是 0, 代表了游标的位置, 说明第一次匹配的游标是在第 0 位, 第二次匹配的“ab”游标在第 2 位, 一直调用游标一直往后走, 直到走到底 6 位匹配最后一个之后, 你再调用方法的话就是 `null`, 继续调用的话游标回到最开始的位置匹配第一个。游标在一圈一圈的转。而游标的位置有一个属性, 就是上面提到的 `reg.lastIndex`。

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
```



```

console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));

```

```

0
▶ ["ab", index: 0, input: "abababab", groups: undefined]
2
▶ ["ab", index: 2, input: "abababab", groups: undefined]
4
▶ ["ab", index: 4, input: "abababab", groups: undefined]
6
▶ ["ab", index: 6, input: "abababab", groups: undefined]
8
null
0
▶ ["ab", index: 0, input: "abababab", groups: undefined]
>

```

我们可以发现每次 `reg.exec()` 执行匹配之后都会自动的把游标往后挪，然后当匹配为 `null` 的时候把游标又挪到最开始的地方。那么：

```

var reg = /ab/g;
var str = "abababab";
console.log(reg.lastIndex);
console.log(reg.exec(str));
reg.lastIndex = 0;
console.log(reg.exec(str));

```

```

0
▶ ["ab", index: 0, input: "abababab", groups: undefined]
▶ ["ab", index: 0, input: "abababab", groups: undefined]

```

我可以通过 `reg.lastIndex` 给游标手动的挪到第 0 位，那么他下一次匹配的还是最开始的“ab”，所以他从哪里开始匹配完全受 `lastIndex` 控制，并且我们可以手动控制 `lastIndex`。

那如果我们不加 `g` 呢？

```

var reg = /ab/;
var str = "abababab";
console.log(reg.lastIndex);

```

```
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
```

```
0
```

```
▶ ["ab", index: 0, input: "abababab", groups: undefined]
```

```
0
```

```
▶ ["ab", index: 0, input: "abababab", groups: undefined]
```

可以看到游标根本就不动，他永远从第 1 个开始匹配，匹配不了后边的。

在讲下边的之前，我们先拓展一点小知识：

比如说我现在想匹配一个结构为 xxxx 的字符串片段，就是 4 位一样即可，怎么写？

```
var reg = /(\w)\1\1\1/g
```

这里边的小括号还有一个意思叫子表达式，当你把他括起来之后，他会记录里边匹配的内容，然后我们就可以通过 \1 把他反向引用出来，\1 就是反向引用第一个子表达式匹配的内容，就是你 \w 匹配的东西，我要复制一个一模一样的出来，三个 \1 就代表后三位的内容必须和第一位完全雷同。

```
var reg = /(\w)\1\1\1/g;
```

```
var str = "aaaabbbb";
```

str.match(reg) 就得到 ["aaaa", "bbbb"]。

那我要匹配 aabb 的形式，就：

```
var reg = /(\w)\1(\w)\2/g;
```

```
var str = "aabb";
```

\1 引用第一个子表达式的内容，\2 就引用第二个子表达式的内容，str.match(reg) 就得到 ["aabb"]。

```
var reg = /(\w)\1(\w)\2/g;
```

```
var str = "aabb";
```

```
console.log(reg.exec(str));
```

```
▶ (3) ["aabb", "a", "b", index: 0, input: "aabb", groups: undefined]
```

你会发现 exec() 这个方法把子表达式匹配的内容都打印出来了。

11. 支持正则表达式的 String 对象的方法

(1) match(): 找到一个或多个正则表达式的匹配，这个我们很熟悉了，给你看一个现象哈：

```
var reg = /(\w)\1(\w)\2/;
```

```
var str = "aabb";
```

```
console.log(str.match(reg));
```

```
top Filter
▶ (3) ["aabb", "a", "b", index: 0, input: "aabb", groups: undefined]
```

你不加 `g` 只匹配一个，而且把子表达式的引用内容都打印出来了。

```
var reg = /(\w)\1(\w)\2/g;
```

```
var str = "aabb";
```

```
console.log(str.match(reg));
```

```
▶ ["aabb"]
```

加上 `g` 那些累赘信息全没有了。

(2) `search()`：检索与正则表达式相匹配的值

这个 `search()` 如果匹配不到就返回-1，但凡返回的不是-1 就都代表可以匹配到。

```
var reg = /(\w)\1(\w)\2/g;
```

```
var str = "edaabbbbee";
```

```
console.log(str.search(reg));
```

```
console.log(str.search(reg));
```

此时打印的都是 2，他返回的是匹配到的这个位置，他追求的只是能不能匹配到，他不管你匹配多少个，所以加不加 `g` 无所谓，但是匹配不到就返回-1。

(3) `split()`：把字符串分割为字符串数组

```
var reg = /(\w)\1/g;
```

```
var str = "bntgjzeasrykzeaskl,zntgjzeasrykkzeaskl,";
```

```
console.log(str.split(reg));
```

我们以两个相同字母为界来拆，此时就打印 `["bntgjze", "a", "srykzeaskl,zntgjzeasry", "k", "zeaskl,"]`，不好的一点是他会把子表达式的匹配内容一起返回，这就恶心了。

```
var reg = /f/g;
```

```
var str = "avfdw";
```

```
console.log(str.split(reg));
```

此时就打印 `["av", "dw"]`。

(4) `replace()`：替换与正则表达式匹配的字串。

我们先来看一个普通的：

```
var str = "aa";
```

```
console.log(str.replace("a", "b"));
```

里边传如两个参数，意思是把 `a` 替换成 `b`，此时就打印 `ba`，因为此时他没有访问全局的能力，现在我加上正则表达式：

```
var reg = /a/;
```

```
var str = "aa";
```

```
console.log(str.replace(reg, "b"));
```

此时依然打印 ba，因为正则表达式没加 g。

```
var reg = /a/g;
```

```
var str = "aa";
```

```
console.log(str.replace(reg, "b"));
```

此时就打印 bb。

现在要求把 aabb 转换为 bbaa，怎么办？

方法一：

```
var reg = /(\w)\1(\w)\2/g;
```

```
var str = "aabb";
```

```
console.log(str.replace(reg, "$2$2$1$1"));
```

我们先把 aabb 的形式用正则表达式写出来，然后把正则表达式传入第一个参数，匹配到了之后，第二个参数也得是字符串，这里边也有一个方法可以反向引用正则表达式里子表达式里匹配的内容，\$1 引用的是第一个子表达式里匹配的内容，\$2 引用的是第二个，然后把它拼在一起就可以了。现在就打印 bbaa。

方法二：

```
var reg = /(\w)\1(\w)\2/g;
```

```
var str = "aabb";
```

```
console.log(str.replace(reg, function ($, $1, $2) {
```

```
    return $2 + $2 + $1 + $1;
```

```
}));
```

里边第二个参数直接传 function，这个 function 系统会帮我们调用的，最后返回一个字符串即可。系统会传几个参数进去，我们得用形参接收一下。第一个参数是正则表达式匹配结果，第二个参数是第一个子表达式匹配的内容，第三个参数是第二个子表达式匹配的内容，所以我们最后把他们拼在一起即可。此时就打印 bbaa。

备注：var str = "aabb"; 然后 str.toUpperCase() 得到 "AABB"，把字符串转换为大写，str.toUpperCase().toLowerCase() 得到 "aabb" 在转为小写。toUpperCase() 变大写，toLowerCase() 变小写。

练习 2：把 the-first-name 变成小驼峰式大写。

```
var reg = /-(\w)/g;
```

```
var str = "the-first-name";
```

```
console.log(str.replace(reg, function ($, $1) {
```

```
    return $1.toUpperCase();
```

```
}));
```

解析：我们只需要把-f 变成 F，-n 变成 N 即可，那么先定义正则表达式把-f 和-n 选出来，我们把 f 和 n 那一位装到子表达式里，然后用 replace 替换，现在都选出来了，然后替换成子表达式的大写即可，现在就打印 theFirstName。

注意：reg 找了几次 function 就执行几次。

12. 正向预查（正向断言）

```
var reg = /a(?=b)/g;
```

```
var str = "abaaaaaa";
```

比如说我现在要找 a，但是有条件，要找的是后边紧跟着 b 的那个 a，就可以这么写，这里边 b 只是起到修饰的作用，不参与选择，此时 str.match(reg) 就得到 ["a"]。

```
var reg = /a(?!b)/g;
```

```
var str = "abaaaaaa";
```

这就是找的 a 必须是后边没有紧跟着 b 的 a，此时 str.match(reg) 就得到 ["a", "a", "a", "a", "a", "a"]

13: 非贪婪匹配

贪婪匹配是能多就不少，非贪婪匹配就是能少就不多，只需要在量词后边加上?即可成为非贪婪匹配。

```
var reg = /\w+?/g;
```

```
var str = "aaa";
```

str.match(reg) 即得到 ["a", "a", "a"]。

```
var reg = /\w??/g;
```

```
var str = "aaa";
```

这里第一个? 是量词，第二个? 是转为非贪婪匹配，str.match(reg) 就得到 ["", "", "", ""]，能匹配 0 个就不匹配 1 个。

14. 补充

(1) 你要匹配空格就直接在正则表达式里敲空格即可。

```
var reg = / /g;
```

```
var str = "a aa";
```

str.match(reg) 就得到[" ""]。

(2) 如果你用 replace() 想把里边的内容替换成\$的话必须在前边再加一个\$，因为单个的是有语法含义的。

(3) 正则表达式里的| 如果不加括号就代表前边一个或，后边一个或。

(4) 正则表达式里如果你想匹配\、?、+、*、()等有语法含义的，前边必须加转义符号\。

练习 3：字符串去重

```
var reg = /(\w)\1*/g;  
var str = "aaaaaabbbbbbbbbbcccccccc";  
console.log(str.replace(reg, "$1"));
```

解析：先把他选出来，正则里*是\1 的量词，所以他选出来了一片同样的字符，然后替换成第一个子表达式里匹配的内容即可。加上 g 就都替换了，打印 abc。

练习 4：（2014 百度试题，成哥有史以来见过最难的题）有一个数字 1000000000000，现在要求科学计数法，从后往前每隔三位打个点。

```
var str = "1000000000000";  
var reg = /(?=(\B)(\d{3})+)$/g;  
console.log(str.replace(reg, "."));
```

解析：首先得是从后往前，我们的思路是先找空，怎样的空，从后往前每隔三位的那个空，所以得用正向预查修饰这个空，正向预查的括号前边啥都不写就代表空，然后括号里\$代表以啥啥啥结尾，也就是从后往前查的意思，\d{3}代表三位数，+代表出现了一次到多次，所以这就把空修饰好了，得再加一个修饰\B，必须是非单词边界，现在把这个空替换成.即可，打印得 100.000.000.000。

渡一教育
DUYI EDUCATION