

0. 引入 js 方法

对 使用 script 标签，将 javascript 代码写到<script></script>之间

对 添加外部 javascript 文件

对 使用行内 javascript

错 使用@import 引入 javascript 文件

在 HTML body 部分中的 JavaScripts 会在页面加载的时候被执行。

在 HTML head 部分中的 JavaScripts 会在被调用的时候才执行

css 在<head></head>中引用 先把样式添加进去让浏览器渲染页面

解释语言的特性有什么？

正确答案: A B 你的答案: C D (错误)

非独立

效率低

IE 与 FF

innerText IE 支持，FIREFOX 不支持

document.createElement FIREFOX 支持，IE 不支持

用 setAttribute 设置事件 FIREFOX 不支持，IE 支持

IE 并不是不支持 setAttribute 这个函数,而是不支持用 setAttribute 设置某些属性，例如对象属性、集合属性、事件属性，也就是说用 setAttribute 设置 style 和 onclick 这些属性在 IE 中是行不通的。为达到兼容各种浏览器的效果，可以用点符号法来设置 Element 的对象属性、集合属性和事件属性。

```
<form name="a">
<select name="a" size="1" id="obj">
<option value="a">1</option>
<option value="b">2</option>
<option value="c">3</option>
</select>
</form>
```

当前默认选中的是第一个，

console.log(obj.options[obj.selectedIndex].text) 输出的是 1,

console.log(obj.options[obj.selectedIndex].value) 输出的是 a

变量命名规则：

第一个字符必须是一个字母、下划线（_）或一个美元符号（\$）；其他字符可以是字母、下划线、美元符号或数字。

下列代码，页面打开后能够弹出 alert(1)的是？

正确答案: A B C

```
<iframe src="javascript: alert(1)"></iframe>
```

```
<img src="" onerror="alert(1)"/>
```

```
IE 下<s style="top:expression(alert(1))"></s>
```

```
<div onclick="alert(1)"></div>
```

A 加载页面的时候触发；

B onerror 事件 当图片不存在时,将触发；

C 在 ie 7 下会连续弹出，IE5 及其以后版本支持在 CSS 中使用 expression，用来把 CSS 属性和 Javascript 表达式关联起来，这里的 CSS 属性可以是元素固有的属性，也可以是自定义属性。就是说 CSS 属性后面可以是一段 Javascript

表达式，CSS 属性的值等于 Javascript 表达式计算的结果。在表达式中可以直接引用元素自身的属性和方法，也可以使用其他浏览器对象。这个表达式就好像是在这个元素的一个成员函数中一样。参考资料 <http://www.blueidea.com/tech/site/2006/3705.asp>

D 不可以,因为 div 里没有内容，盒子的宽度为 0 所以点击不了的；还有就是点击才会弹框。

保留字

(字母排序)

abstract

boolean break byte

case catch char class const continue

debugger default delete do double

else enum export extends

false final finally float for function

goto

if implements import in instanceof int interface

long

native new null

package private protected public

return

short static super switch synchronized

this throw throws transient true try typeof

var void volatile

while with

JS 对象、属性和方法

Array	Date	eval	function	hasOwnProperty
Infinity	isFinite	isNaN	isPrototypeOf	length
Math	NaN	name	Number	Object
prototype	String	toString	undefined	valueOf

Switch 中没有 break 则继续向下执行；++x 取加后的值，x++取加前的值

Yahoo! User Interface 库 (YUI) 包含一个 bucketload。

和 YUI 一样，ExtJS 包含大量开箱即用的组件，其中有很多功能强大的网格控件，支持内联编辑、分页、筛选、分组、汇总、缓冲和数据绑定。

MooTools 和 Prototype、jQuery 不包含开箱即用的 UI 控件和小部件。

javascript 中的数字在计算机内存存储为多少 Byte？8byte

1. Js 变量类型

JS 有 8 种变量类型 → null 和 undefined 和 0 和 1
→ 深拷贝
→ 不同变量类型的区别 → typeof 返回值 → 数据类型判断

1.1 值类型和引用类型

值类型(基本数据类型)有 5 种：布尔、字符串、数值(NaN(自己都不等于自己)和 Infinity 是两个全局变量，不是数据类型)、null、undefined、

"JavaScript 内部，所有数字都是以 64 位浮点数形式储存，即使整数也是如此。"

引用类型：数组、对象、函数

js七种数据类型：Sting Object null
undefined Array Boolean Number
js五种基本类型：String Boolean
Number null undefined
typeof六种返回格式：'string'
'number' 'object' 'function'
'boolean' 'undefined'

```
console.log(Number(""));           //0
console.log(Number(null));          //0
console.log(Number(undefined));     //NaN
// Number() 函数就是将传入值转换为数值形式返回
console.log(parseInt(""));          //NaN
console.log(parseInt(null));         //NaN
console.log(parseInt(undefined));   //NaN

console.log(null == 0);              //false
console.log(undefined == 0);         //false
```

下面几个都会转化为 0：

- Number()
- Number(0)
- Number("")
- Number('0')
- Number(false)
- Number(null)
- Number([])
- Number([0])

而 Number({}) //NaN

另 Boolean([]) //true

Boolea(0)返回布尔值 false，var = Boolea(0)，则 y 的值为 false

定义和用法

isNaN() 函数用于检查其参数是否是非数字值。

语法

isNaN(x)	
参数	描述
x	必需。要检测的值。

返回值

如果 x 是特殊的非数字值 NaN（或者能被转换为这样的值），返回的值就是 true。如果 x 是其他值,则返回 false。

位移运算

假设有如下代码，那么 a(10) 的返回结果是？（ ）

```
function a(a)
{
    a^=(1<<4)-1;
    return a;
```

```

}
1<<4    左移相当于  $1 \times 2^4 = 16$ , 1 左移四位,  $1 \rightarrow 10000$  (二进制)
 $a^{\wedge}16 - 1 = 15$ 
 $a = a^{\wedge}15 = 10^{\wedge}15$ 
^ 异或运算:
10 的二进制 00001010
15 的二进制 00001111
===== > 00000101    转成十进制: 5
(按位异或运算, 同为 1 或同为 0 取 0, 不同取 1)

```

Js 的除法: $5/2=2.5$, 而不是像其他语言的整除; 然后会有精度的问题, 需要两个 n 位小数的浮点操作数都乘以 10 的 n 次方以后再运算, 再缩小同样的倍数。

解决多人开发函数重名问题:

根据不同的开发人员实现的功能, 在函数名加前缀

每个开发人员都把自己的函数封装到类中, 然后调用的时候即使函数名相同, 但是因为是要类. 函数名来调用, 所以也减少了重复的可能性

字符常量是用单引号括起来的单个普通字符或转义字符

js 中的 Blob 对象

在计算机中, BLOB 常常是数据库中用来存储 二进制文件 的字段类型。

这里说的是一种 JavaScript 的对象类型。

MYSQL 中的 BLOB 类型就只是个二进制数据容器。而 HTML5 中的 Blob 对象除了存放二进制数据外还可以设置这个数据的 MIME 类型, 这相当于对文件的储存。

Blob 构造函数, 接受两个参数:

第一个为一个数据序列, 可以是任意格式的值, 例如, 任意数量的字符串, Blobs 以及 ArrayBuffers。

第二个参数, 是一个包含了两个属性的对象, 其两个属性分别是:

- (1) type -- MIME 的类型。
- (2) endings -- 决定 append() 的数据格式,

```
<script>
```

```
var blob = new Blob([ "Hello World!" ],{type: "text/plain" });
```

```
</script>
```

1.2 null 和 undefined 的区别?

null 转为数值时为 0; undefined 转为数值时为 NaN。

undefined 表示“缺少值”, 就是此处应该有一个值, 但是还没有定义。典型用法是:

- (1) 变量被声明了, 但没有赋值(初始化)时, 就等于 undefined。
- (2) 调用函数时, 应该提供的参数没有提供, 该参数等于 undefined。
- (3) 对象没有赋值的属性, 该属性的值为 undefined。
- (4) 函数没有返回值时, 默认返回 undefined。

null 表示“没有对象”, 即该处不应该有值。典型用法是:

- (1) 作为函数的参数, 表示该函数的参数不是对象。
- (2) 作为对象原型链的终点。
- (3) 表示函数企图返回一个不存在的对象

1.3 null==undefined 为真; null===undefined 为假

```
// 问题: 何时使用 === 何时使用 ==

if (obj.a == null) {
    // 这里相当于 obj.a === null || obj.a === undefined, 简写形式
    // 这是 jquery 源码中推荐的写法
}
```

要使用严格相等的时候使用===, 只使用相等的时候使用==。

===:

类型不同则不会进行类型转换, 直接不恒等;

null 和 undefined 不恒等;

比较的数之中有 NaN 则不恒等;

0 和-0 恒等

字符串的长度和内容与编码方式(Unicode、ASCII、UTF-8)都相同则恒等;

同一个对象的不同引用恒等;

==: $B < 0 < S < N$; +时 $100 + '10' = '10010'$, $N < S$

类型不同可以进行转换然后比较内容

true 优先级最低, 首先不管怎样变为数字 1;

字符串、对象相比数字优先级低, 也要变成数字: 字符串是直接字面变, 对象调用 valueOf 和 toString 变;

对象优先级也低于字符串, 相比较的时候变成字符串

1.4 Date 和 Math

1. 获取 2017-06-10 格式的日期

```
1 function formatDate(dt) {
2     if (!dt) {
3         dt = new Date()
4     }
5     var year = dt.getFullYear()
6     var month = dt.getMonth() + 1
7     var date = dt.getDate()
8     if (month < 10) {
9         // 强制类型转换
10        month = '0' + month
11    }
12    if (date < 10) {
13        // 强制类型转换
14        date = '0' + date
15    }
16    // 强制类型转换
17    return year + '-' + month + '-' + date
18 }
19 var dt = new Date()
20 var formatDate = formatDate(dt)
21 console.log(formatDate)
```

'2015-05-04'是无法被各个浏览器中, 使用 new Date(str)来正确生成日期对象的。正确的用法是'2015/05/05'.

2. 获取随机数, 要求长度是一致的字符串格式

```
1 var random = Math.random()
2 var random = random + '0000000000' // 后面加上 10 个零
3 var random = random.slice(0, 10)
4 console.log(random)
```

3. 写一个能遍历对象和数组的通用 forEach 函数

```

1 function forEach(obj, fn) {
2     var key
3     if (obj instanceof Array) {
4         // 准确判断是不是数组
5         obj.forEach(function (item, index) {
6             fn(index, item)
7         })
8     } else {
9         // 不是数组就是对象
10        for (key in obj) {
11            fn(key, obj[key])
12        }
13    }
14 }

```

```

15 var arr = [1,2,3]
16 // 注意，这里参数的顺序换了，为了和对象的遍历格式一致
17 forEach(arr, function (index, item) {
18     console.log(index, item)
19 })
20
21 var obj = {x: 100, y: 200}
22 forEach(obj, function (key, value) {
23     console.log(key, value)
24 })

```

左图 obj 调用的 forEach 是原生数组 API；

知识点：

1. 日期

```

Date.now() // 获取当前时间毫秒数
var dt = new Date()
dt.getTime() // 获取毫秒数
dt.getFullYear() // 年
dt.getMonth() // 月 (0 - 11)
dt.getDate() // 日 (0 - 31)
dt.getHours() // 小时 (0 - 23)
dt.getMinutes() // 分钟 (0 - 59)
dt.getSeconds() // 秒 (0 - 59)

```

2. Math

Math.random() 主要用于清除缓存

比如从服务器中获取什么资源，如果链接一直不变，则得到的不能及时更新，会一直从浏览器缓存中取数，如果在链接后面加上 random() 则会保证一直取得的都是新的版本。

语句 var arr=[a,b,c,d]; 执行后，数组 arr 中每项都是一个整数，下面得到其中最大整数语句正确的是哪几项？

- A. Math.max(arr)
- B. Math.max(arr[0], arr[1], arr[2], arr[3])
- C. Math.max.call(Math, arr[0], arr[1], arr[2], arr[3])
- D. Math.max.apply(Math, arr)

BCD

A 选项错误，因为函数 Math.max(x) 的参数是 Number 类型，可以是小数，整数，正数，负数或者是 0。如果不是上面所述类型就会返回 NaN。

Math.ceil() 执行向上舍入，即它总是将数值向上舍入为最接近的整数；

Math.floor() 执行向下舍入，即它总是将数值向下舍入为最接近的整数；

Math.round() 执行标准舍入，即它总是将数值四舍五入为最接近的整数（这也是我们在数学课上学到的舍入规则）。

3. 数组 API

- forEach 遍历所有元素
- every 判断所有元素是否都符合条件
- some 判断是否有至少一个元素符合条件
- sort 排序
- map 对元素重新组装，生成新数组
- filter 过滤符合条件的元素

在原数组上修改的数组 API: join、reverse、sort、splice、push、pop、unshift、shift、toString、forEach、返回新的修改的数组 API: concat、slice、map、filter、every、some、reduce、reduceRight
splice(index, howmany, item1, ..., itemX), howmany 要删除项目的数量，为 0 则不会删除；item 是要添加的项目，要添加的时候 howmany 要为 0；返回包含被删除项目的新数组；index 是删除或者添加的位置。

arguments 能获得函数对象传入的参数组，类似与一个数组，能够通过 length 获取参数个数，能通过下标获取该位置的参数，但是它不能使用 forEach 等方法。

将 arguments 转化为数组后，截取第一个元素之后的所有元素

```
var args = Array.prototype.slice.call(arguments, 1); //slice 函数的参数是 1, 操作它的对象是 arguments
或
var arr = [];
for (var i = 1; i < arguments.length; i++)
    arr.push(arguments[i]);
```

```
var arr = [];
arr[0] = 0;
arr[1] = 1;
arr.foo = 'c';
console.log(arr.length);
```

这里要理解所谓‘数组’其实是 array 类型对象的一个特殊作用，就是：我们可以对它进行一种模式的数据存储，但除此之外，它依然是一个对象

var arr = ['1', '2'] //这本质上是一系列操作：得到一个数组对象；调用了它的数组方法存入了一些数据，arr.length 根据存入数据的数目被修改

arr.length, 对 arr 对象的 length 属性进行一个访问

arr.foo = 'hello' 对 arr 对象创建一个属性，所以.foo 跟.length 地位是并列的：就是 arr 的一个属性，同时 arr 的数组方法跟这些属性是毫不相关的

所以打印 2，而不是 3。

4. 对象 API

1.4 实现一个函数 clone，可以对 JavaScript 中的 5 种主要的数据类型（包括 Number、String、Object、Array、Boolean）还有 Date 和 RegExp 进行值复制。null 和 undefined 都不能调用函数，在调用之前进行检测就好。


```
Object.prototype.clone = function() {
  if (this.constructor === Number || this.constructor === String || this.constructor ===
  Boolean) {
    return this;
  } else if (this.constructor === Date) {
    return new Date(this.valueOf());
  } else if (this.constructor === RegExp) {
    var pattern = this.valueOf();
    var flags = '';
    flags += pattern.global ? 'g' : '';
    flags += pattern.ignoreCase ? 'i' : '';
    flags += pattern.multiline ? 'm' : '';
    return new RegExp(pattern.source, flags);
  } else {
    var o = this.constructor === Array ? [] : {};
    for (var e in this) {
      o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];
    }
    return o;
  }
}
```

注:

g: global, 全部匹配的项都搜索出来

i: ignore, 匹配时不在意大小写

m: muti, 多行匹配, 待检测的字符串包含多行的话, 每行的开头和结尾都可以使用^和\$看是否匹配, 而不是像之前的只有整个字符串的开头和结尾才可以使用^和\$看是否匹配。

1.5 值类型和引用类型的区别: var a = b 对于值类型 a 和 b 是两个变量, 对于引用类型, 对象还是只有一个, 指针有两个; 引用类型的一个好处是, 当对象很大的时候, 复制对象不会占用过多内存。

Typeof 返回 6 种:

```
typeof undefined // undefined
typeof 'abc' // string
typeof 123 // number
typeof true // boolean
typeof {} // object
typeof [] // object
typeof null // object
typeof console.log // function
```

所以 typeof 只能分辨值类型, 对于引用类型 (对象、数组、函数) 和特殊的值类型 null 是无法区分的, typeof(num) !== 'number'

五种基本类型能识别 4 种, null 是特例; 然后识别 object, function 是特例可被识别

补充: typeof Symbol() // "symbol"

1.6 要想准确判断一个数组:

Object.prototype.toString.call(o) === "[Object Array]"; 通过继承自 Object.prototype 的未经过重写的 toString() 方法来获取对象的类属性。instanceof (对象是不是构造函数的事例) 不是一个可靠的数组检测方法, 只能用于简单的情形。因为浏览器中可能有多个窗口或者窗体存在, 每个窗口都有自己的 js 环境和全局对象, 每个全局对象有自己的一组构造函数, 因此一个窗体中的对象将不可能是另外窗体中的构造函数的实例。

```
o instanceof Array;
this.constructor == Array;
```

关于判断一个字符串

下面哪些语句可以在 JS 里判断一个对象 oStringObject 是否为 String。

正确答案: A 你的答案: A B (错误)

oStringObject instanceof String

typeof oStringObject == 'string'

oStringObject is String

解析:

JS 里面 String 的初始化有两种方式: 直接赋值和 String 对象的实例化


```
var str = "abc";
```

```
var str = new String("abc");
```

通常来说判断一个对象的类型使用 `typeof`，但是在 `new String` 的情况下的结果会是 `object`

此时需要通过 `instanceof` 来判断(题目也说了是对象 `oStringObject`)

2 变量计算:

2.1 值类型的强制类型转换发生在以下四种情况

① 字符串拼接: `100 + '10'`，数字转为字符串; `var a="40", var b=7`, 执行 `a%b` 时 `"40"—>40`，结果是 5

② `==`:

类型不同可以进行转换然后比较内容

`true` 优先级最低，变为数字 1;

字符串、对象相比数字优先级低，也要变成数字: 字符串是直接字面变，对象调用 `valueOf` 和 `toString` 变;

对象优先级也低于字符串，相比较的时候变成字符串; 同类型相比较时的情况同 `===`，两个引用值指向同一个对象、数组或者函数的时候它们才相等。

③ `if` 语句: 转成布尔类型

在 `if` 中当做 `false` 的判断值: `0`、`NaN`、`false`、`null`、`undefined`、`' '`

④. 逻辑运算符

```
console.log(10 && 0) // 0
console.log('' || 'abc') // 'abc'
console.log(!window.abc) // true

// 判断一个变量会被当做 true 还是 false
var a = 100
console.log(!a)
```

`&&`: 左边为假就简单返回左边的值，不会对右操作数进行计算; 左为真就返回右操作数计算结果。左右并不是非要以 `true`、`false` 的形式计算

`||`: 左边为真就简单返回左边的值，不会对右操作数进行计算; 左为假就返回右操作数计算结果。

1 `console.log(1+ "2"+"2");`

做加法时要注意双引号，当使用双引号时，JavaScript 认为是字符串，字符串相加等于字符串合并。

因此，这里相当于字符串的合并，即为 122.

2 `console.log(1+ +"2"+"2");`

第一个 `+"2"` 中的加号是一元加操作符，`+"2"` 会变成数值 2，因此 `1+ +"2"` 相当于 `1+2=3`.

然后和后面的字符串 `"2"` 相合并，变成了字符串 `"32"`.

3 `console.log("A" - "B" + "2");`

`"A" - "B"` 的运算中，需要先把 `"A"` 和 `"B"` 用 `Number` 函数转换为数值，其结果为 `NaN`，在剪发操作中，如果有一个是 `NaN`，则结果是 `NaN`，因此 `"A" - "B"` 结果为 `NaN`。

然后和 `"2"` 进行字符串合并，变成了 `NaN2`.

4 `console.log("A" - "B" + 2);`

根据上题所述，`"A" - "B"` 结果为 `NaN`，然后和数值 2 进行加法操作，在加法操作中，如果有一个操作数是 `NaN`，则结果为 `NaN`。

《JavaScript 权威指南》的部分相关知识点

`"=="` 运算符 (两个操作数的类型不相同)

如果一个值是 `null`，另一个值是 `undefined`，则它们相等

如果一个值是数字，另一个值是字符串，先将字符串转换为数字，然后使用转换后的值进行比较。

如果其中一个值是 `true`，则将其转换为 1 再进行比较。如果其中的一个值是 `false`，则将其转换为 0 再进行比较。

如果一个值是对象，另一个值是数字或字符串，则将对象转换为原始值，再进行比较。

对象到数字的转换

如果对象具有 `valueOf()` 方法，后者返回一个原始值，则 JavaScript 将这个原始值转换为数字（如果需要的话）并返回一个数字。

否则，如果对象具有 `toString()` 方法，后者返回一个原始值，则 JavaScript 将其转换并返回。（对象的 `toString()` 方法返回一个字符串直接量（作者所说的原始值），JavaScript 将这个字符串转换为数字类型，并返回这个数字）。

否则，JavaScript 抛出一个类型错误异常。

空数组转换为数字 0

数组继承了默认的 `valueOf()` 方法，这个方法返回一个对象而不是一个原始值，因此，数组到数字的转换则调用 `toString()` 方法。空数组转换为空字符串，空字符串转换为数字 0。

'+new Array(017)' 输出？

首先，前面+是一元运算符，相当于我们说的正负，无运算效果，但是可以将字符串等转为 `number` 类型。

此题中 017 其实是八进制，故而是 `Array(15)`。

这里相当于对于一个未赋值但是长度为 15 的数组进行 `number` 类型转化，其结果为 `NaN`

2.2 计算规则

2.2.1 `var a=b=3` 相当于 `var a = 3; b = 3;` `b` 是全局的

```
1 var a,b;
2 (function(){
3     alert(a);
4     alert(b);
5     var a=b=3;
6     alert(a);
7     alert(b);
8 })();
9 alert(a);
10 alert(b);
```

undefined, undefined, 3,3, undefined, 3

2.3 运算符优先级

假设 `val` 已经声明, 可定义为任何值。则下面 js 代码有可能输出的结果为:

`console.log('Value is ' + (val !== '0') ? 'define' : 'undefine');`

因为+运算符优先级大于?:运算符, 所以代码执行顺序是 `('Value is ' + (val !== '0')) ? 'define' : 'undefine'`, 而?前面的表达式运算结果为字符串'Value is true' 或者 'Value is false', 它被转换为布尔值是 `true`, 所以打印出来的结果是字符串'define'。(val !== '0'), 如果 val 的值为 0、' 0'、false、[0], 那么计算值为 false, 其他 val 值的计算值都为 true。

3. js 对象的几种创建方式

① 工厂模式

② 构造函数模式

描述 new 一个对象的过程, 比如 `var b = new Array();`

构造函数内首先会新定义一个 `this` 空对象, 然后根据传参给 `this` 对象添加上属性和方法, 最后返回 `this` 对象赋值给 `b`

- 创建一个新对象
- `this` 指向这个新对象
- 执行代码, 即对 `this` 赋值
- 返回 `this`

补充:

- `var a = {}` 其实是 `var a = new Object()` 的语法糖
- `var a = []` 其实是 `var a = new Array()` 的语法糖
- `function Foo(){...}` 其实是 `var Foo = new Function(...)`
- 使用 `instanceof` 判断一个函数是否是一个变量的构造函数

判断一个变量是否为“数组”：
变量 `instanceof Array`

从可读性和性能上讲推荐使用语法糖

```
var A = {n:4399};
var B = function(){this.n=9999};
var C = function(){var n = 8888};
B.prototype = A;
C.prototype = A;
var b = new B();
var c = new C();
A.n++;
console.log(b.n);
console.log(c.n);
```

`var c = new C();`

上面这个语句的实际运行过程是这样的。

```
var c = function() {
    var o = new Object();
    //第一个参数改变函数的作用域，即相当于在函数内部设置 this = o
    C.apply(o, arguments);
    return o;
}
```

这样，由于 `C()` 函数中

`var n = 8888;`

这样只是在函数中创建了一个私有变量，并没有为对象执行任何操作，因此 `C` 的实例中不存在名字为“n”的属性。所以，`c.n` 会访问原型中的属性名为“n”的值。

所以：9999 4400

③原型模式

1. 五个原型规则

所有的引用类型(数组、对象、函数)，都具有对象特性，即可自由扩展属性(`null` 是原始类型，不是引用类型，虽然 `typeof (null)` 是 `object`)

所有的引用类型(数组、对象、函数)，都有一个 `__proto__` 属性(隐式原型)，属性值是一个普通的对象

所有的函数都有一个 `prototype` 属性，属性值也是一个普通的对象，对比 `__proto__`，它是显示原型

所有的引用类型(数组、对象、函数)，`__proto__` 属性指向它的构造函数的 `prototype` 属性

```
console.log(obj.__proto__ === Object.prototype)
```

当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__`（即它的构造函数的 `prototype`）中寻找。

用过哪些设计模式？

工厂模式：

主要好处就是可以消除对象间的耦合，通过使用工程方法而不是 `new` 关键字。将所有实例化的代码集中在一个位置防止代码重复。工厂模式解决了重复实例化的问题，但还有一个问题，那就是识别问题，因为根本无法搞清楚他们到底是哪个对象的实例。

```
function createObject(name, age, profession) { //集中实例化的函数
    var obj = new Object();
    obj.name = name; obj.age = age;
```

```

    obj.profession = profession;
    obj.move = function () {
        return this.name + ' at ' + this.age + ' engaged in ' + this.profession;
    };
    return obj;
}
var test1 = createObject(' trigkit4',22,' programmer'); //第一个实例
var test2 = createObject(' mike',25,' engineer');//第二个实例

```

构造函数模式

使用构造函数的方法，即解决了重复实例化的问题，又解决了对象识别的问题，该模式与工厂模式的不同之处在于：

1. 构造函数方法没有显示的创建对象（new Object()）；
2. 直接将属性和方法赋值给 this 对象；
3. 没有 return 语句。

2. 解释下原型继承的原理。

当查找一个对象的属性时，JavaScript 会向上遍历原型链，直到找到给定名称的属性为止。（大多数 JavaScript 的实现用 __proto__ 属性来表示一个对象的原型链。）

以下代码展示了 JS 引擎如何查找属性：

```

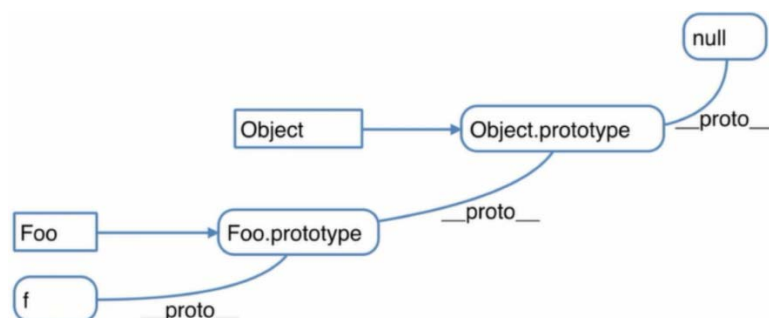
function getProperty(obj, prop) {
    if (obj.hasOwnProperty(prop))
        return obj[prop]
    else if (obj.__proto__ !== null)
        return getProperty(obj.__proto__, prop)
    else
        return undefined
}

```

注：__proto__ 是一个不应在你代码中出现的非正规的用法，这里仅仅用它来解释 JavaScript 原型继承的工作原理

3. 原型链例子：

- f instanceof Foo 的判断逻辑是：
- f 的 __proto__ 一层一层往上，能否对应到 Foo.prototype
- 再试着判断 f instanceof Object



也正是因为这个特性，要注只循环对象自身的属性的时候要使用 hasOwnProperty()

```

1 var item
2 for (item in f) {
3     // 高级浏览器已经在 for in 中屏蔽了来自原型的属性
4     // 但是这里建议大家还是加上这个判断，保证程序的健壮性
5     if (f.hasOwnProperty(item)) {
6         console.log(item)
7     }
8 }

```

④混合构造函数和原型模式

⑤动态原型模式

⑥寄生构造函数模式

⑦稳妥构造函数模式

4. javascript 继承的 6 种方法，js 高程 P162

①原型链继承

原型链继承的缺点

一是字面量重写原型会中断关系，使用引用类型的原型，并且子类型还无法给父类型传递参数。

例子：

```

1 // 动物
2 function Animal() {
3     this.eat = function () {
4         console.log('animal eat')
5     }
6 }
7 // 狗
8 function Dog() {
9     this.bark = function () {
10        console.log('dog bark')
11    }
12 }
13 Dog.prototype = new Animal()
14 // 哈士奇
15 var hashiqi = new Dog()

```

hashiqi 的隐式原型等于 Dog 的显式原型，Dog 的显式原型是 Animal，这样就是继承了 Animal

②借用构造函数继承

借用构造函数虽然解决了刚才两种问题，但没有原型，则复用无从谈起。所以我们需要原型链+借用构造函数的模式，这种模式称为组合继承

③组合继承(原型+借用构造)

组合式继承是比较常用的一种继承方法，其背后的思路是 使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又保证每个实例都有它自己的属性。

④原型式继承

⑤寄生式继承

⑥寄生组合式继承

详情：[JavaScript 继承方式详解](#)

5.js 内置函数和对象有哪些

内置函数 OAB NFS DER

Object、Array、Boolean、Number、Function、String、Date、Error、RegExp

全局函数：isNaN() parseInt() eval() setTimeout() decodeURI() 参数:string 功能描述:对 encodeURI() 函数编码过的 URI 进行解码。

内置对象

JSON 和 Math 一样都是 js 内置对象，但 JSON 也是一种数据格式

```
JSON.stringify({a:10, b:20})
JSON.parse('{"a":10,"b":20}')
```

json 的变量名和变量值都有双引号

数值(NaN(自己都不等于自己)和 Infinity 是两个全局变量，不是数据类型)

Js 内置对象：

Arguments	函数参数集合
Array	数组
Boolean	布尔对象
Date	日期时间
Error	异常对象
Function	函数构造器
Math	数学对象
Number	数值对象
Object	基础对象
RegExp	正则表达式对象
String	字符串对象

另宿主对象，如 window,Dom 等

全局属性：NaN、undefined、Infinity

XML 和 JSON 的区别？相同点：都是数据格式

- (1). 数据体积方面。 JSON 相对于 XML 来讲，数据的体积小，传递的速度更快些。
- (2). 数据交互方面。 JSON 与 JavaScript 的交互更加方便，更容易解析处理，更好的数据交互。
- (3). 数据描述方面。 JSON 对数据的描述性比 XML(标签结构)较差。
- (4). 传输速度方面。 JSON 的速度要远远快于 XML。

XML 与 HTML 的主要差异：

- (1)、XML 不是 HTML 的替代；
- (2)、XML 和 HTML 为不同的目的而设计；
- (3)、XML 被设计为传输和存储数据，其焦点是数据的内容；
- (4)、HTML 被设计用来显示数据，其焦点是数据的外观；
- (5)、HTML 旨在显示信息，而 XML 旨在传输信息。

下面哪个不是 RegExp 对象的方法？（）

正确答案: B D 你的答案: D (错误)

- test
- match
- exec
- compile

JavaScript RegExp 对象有 3 个方法：test()、exec() 和 compile()。

- (1) test() 方法用来检测一个字符串是否匹配某个正则表达式，如果匹配成功，返回 true，否则返回 false；
- (2) exec() 方法用来检索字符串中与正则表达式匹配的值。exec() 方法返回一个数组，其中存放匹配的结果。如果未找到匹配的值，则返回 null；

(3) compile() 方法可以在脚本执行过程中编译正则表达式，也可以改变已有表达式。
match 是支持正则表达式的 String 对象的方法

6. 作用域和闭包

执行上下文(变量提升)→this→call 和 apply→作用域链→闭包(场景、作用、缺点)→垃圾回收

6.1. 执行上下文

- 范围：一段<script>或者一个函数
- 全局：变量定义、函数声明 一段<script>
- 函数：变量定义、函数声明、this、arguments 函数

函数声明和函数表达式的区别&变量提升(变量定义和函数声明)

声明之前可用函数，表达式之前函数名的值为 undefined，执行报错。

```
(function() {  
  var x=foo();  
  var foo=function foo() {  
    return "foobar"  
  };  
  return x;  
})();
```

运行到 var x=foo(); 这句的时候会执行 foo，此时还没有 foo 的具体定义内容所以会报错，因为对于函数表达式变量提升只会提升函数名字的定义，此时还不能执行函数；但是函数声明的变量提升是将整个函数定义体全部提升，此时可以执行函数。

```
var f = function g() {  
  return 23;  
};
```

typeof g();

解释：

如果是 typeof f，结果是 function

如果是 typeof f()，结果是 number

如果是 typeof g,结果是 undefined.

如果是 typeof g(),结果是 ReferenceError, g is not defined

在 JS 里，声明函数只有 2 种方法：

第 1 种：function foo(){...} （函数声明）

第 2 种：var foo = function(){...} （等号后面必须是匿名函数，这句实质是函数表达式）

除此之外，类似于 var foo = function bar(){...} 这样的东西统一按 2 方法处理，即在函数外部无法通过 bar 访问到函数，因为这已经变成了一个表达式。

但为什么不是 "undefined"？

这里如果求 typeof g，会返回 undefined，但求的是 g()，所以会先去调用函数 g，这里就会直接抛出异常，所以是 Error。

6.2. this 要在执行时才能确认值，定义时无法确认


```

var a = {
  name: 'A',
  fn: function () {
    console.log(this.name)
  }
}
a.fn() // this === a
a.fn.call({name: 'B'}) // this === {name: 'B'}
var fn1 = a.fn
fn1() // this === window

```

只要函数作为对象的方法被调用，无论是对象自身的方法，还是函数的原型的方法，this 都是指向调用函数的对象的。

函数的四种执行方式&this 的值会不一样:作为构造函数执行、作为对象属性执行、作为普通函数执行、call apply bind

```

var obj = {a:1, b: function () {alert(this.a)}};
var fun = obj.b;
fun();

```

运行结果是 ()

```

var b = function(){
  alert(this.a);
},
obj = {
  a:1,
  b:b // 把函数独立出来
};

```

```

var fun = obj.b; // 存储的是内存中的地址
fun();

```

答案: 弹出 undefined

虽然 fun 是 obj.b 的一个引用，但是实际上，它引用的是 b 函数本身，因此此时的 fun() 其实 是一个不带任何修饰的函数调用，所以 this 指向 window。

```

var obj = {};
obj.log = console.log;
obj.log.call(console, this);
obj.log.call(console, this) = console.log(this)。obj.log 由于 call 函数改变了主体位 console，所以后来是 console.log。this 这里当然是 windows 啦。所以输出 window

```

6.3. call() 和 apply() 的区别和作用?

作用: 动态改变某个类的某个方法的运行环境 (执行上下文)。在全局作用域执行函数 fn, fn.call(o) 就是改变 fn 中的 this 为 o, 而不是 window

apply() 函数有两个参数: 第一个参数是上下文, 第二个参数是参数组成的数组。如果上下文是 null, 则使用全局对象代替。例如:

```
function.apply(o1, [1, 2, 3])
```

call() 的第一个参数是上下文, 后续是实例传入的参数序列, 例如:

```
function.call(o2, 1, 2, 3);
```

6.4. 作用域链、作用域: 无块级作用域; 只有函数作用域和全局作用域

```

var a = 100
function fn() {
  var b = 200

  // 当前作用域没有定义的变量，即“自由变量”
  console.log(a)

  console.log(b)
}
fn()

```

在函数中找不到 a，那就去它的父级作用域中找，这就是作用域链

函数的父级作用域是函数在定义的时候的作用域，不是函数在执行的时候的父级作用域

说说你对作用域链的理解

作用域链的作用是保证执行环境里有权访问的变量和函数是有序的，作用域链的变量只能向上访问，变量访问到 window 对象即被终止，作用域链向下访问变量是不被允许的。

with 改变作用域：

```

[javascript]
01. var scope = "global";
02. function f1() {
03.   console.log(scope);
04. }
05. f1() // output: global
06. function f2() {
07.   scope = "f2"
08.   with(scope) {
09.     f1();
10.   }
11. }
12. f2(); // output: f2

```

with 会改变其范围内的作用域链。

这里 with 把在其范围内的 f1() 函数的执行作用域链的父级作用域更改为了自己，所以在执行 f1() 函数时，找到父级作用域链中 scope 定义为“f2”，因此加入 with 后，第二次可以输出“f2”。

eval 改变作用域：eval 的执行上下文就是局部作用域，当间接使用 eval(var geval=eval;geval();)时执行上下文是全局作用域；

try catch 改变作用域：伪块作用域(throw 的内容可以通过 e 传到 catch 伪块作用域中；删除 e 错误是因为 e 是一个栈变量或逻辑栈变量；只有 e 的上下文是伪块作用域，x 就是普通的函数局部变量)

```

(function(){
  e="default";
  try{
    throw "test";
  }catch(e){
    var e,x=123;
    console.log(e); //test
    console.log(delete e); //false
    e=456;
    console.log(e); //456
  };
  console.log(x); //123
  console.log(e); //default
  console.log(window.e); //undefined
})();

```

6.5. 闭包

```
function F1() {
  var a = 100

  // 返回一个函数 (函数作为返回值)
  return function () {
    console.log(a)
  }
}
// f1 得到一个函数
var f1 = F1()
var a = 200
f1()
```

f1() 虽然是在全局作用域中执行，但是它的定义时的父级作用域是 F1()，所以 f1() 执行的时候打印 100。
闭包的使用场景：函数作为返回值(上图)；函数作为参数传递(下图)

```
//5. 闭包---函数作为参数传递
function F1(){
  var a = 100;
  return function(){
    console.log(a);
  }
}
var f1 = F1();
function F2(fn) {
  var a = 200;
  fn();
}
F2(f1);
```

实际开发中闭包的应用

```
// 闭包实际应用中主要用于封装变量，收敛权限
function isFirstLoad() {
  var _list = []
  return function (id) {
    if (_list.indexOf(id) >= 0) {
      return false
    } else {
      _list.push(id)
      return true
    }
  }
}

// 使用
var firstLoad = isFirstLoad()
firstLoad(10) // true
firstLoad(10) // false
firstLoad(20) // true
```

否则如果要实现 firstLoad() 一定要创建一个数组，这样就会让数组暴露在外面，别人会修改；所以使用闭包。

说说你对闭包的理解

使用闭包主要是为了设计私有的方法和变量(例如局部变量可累加)。闭包的优点是可以避免全局变量的污染，缺点是闭包会常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。在 js 中，函数即闭包，只有函数才会产生作用域的概念

闭包有三个特性：

1. 函数嵌套函数
2. 函数内部可以引用外部的参数和变量
3. 参数和变量不会被垃圾回收机制回收

6.5.2 立即执行函数

假设 output 是一个函数，输出一行文本。下面的语句输出结果是什么？

```
output(typeof (function() {output(“Hello World!”)})());
```

1. 先立即执行匿名函数，输出 Hello World!
2. 函数执行后无返回值，则输出未定义

即：Hello World! undefined

例题：实现函数 makeClosures，调用之后满足如下条件：

1、返回一个函数数组 result，长度与 arr 相同

2、运行 result 中第 i 个函数，即 result[i]()，结果与 fn(arr[i]) 相同

错误写法：

```
function makeClosures(arr, fn) {  
    var result = [];  
    for(var i=0;i<arr.length;i++){  
        var item = function() {  
            console.log(i);  
            return fn(arr[i]);  
        }  
        result.push(item);  
    }  
}
```

//这样的确是往 result 中放了三项，但是每项的内容都是 function() {console.log(i);return fn(arr[i]);}

//但是并不会执行，直到下方 console.log 的时候才执行，闭包中可以使用定义时的父级局部变量，i 和 arr 都

//是有定义的，但是这时 i 经过循环成为 3，arr[3]自然是 undefined。所以需要立即执行的函数: () ()

```
    }  
    return result;  
}  
  
var s = makeClosures([1, 2, 3], function (x) { return x * x;});  
console.log(s[1]() );
```

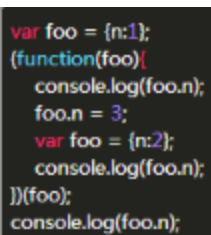
//正确写法 1:

```
function makeClosures(arr, fn) {  
    var result = [];  
    for(var i=0;i<arr.length;i++){  
        (function(i) { //立即函数可以执行给函数赋值的任务。  
            result[i] = function() { //给每个函数赋值函数  
                return fn(arr[i]); //赋值函数的内容  
            }  
        })(i);  
    }  
}
```

//正确写法 2

```
    arr.forEach(function(item) {  
        result.push(function() {  
            return fn(item);  
        })  
    })
```

为什么 forEach 没有使用立即执行也可以呢？那就说明 forEach 是自动执行的。



```
var foo = {n:1};  
(function(foo){  
    console.log(foo.n);  
    foo.n = 3;  
    var foo = {n:2};  
    console.log(foo.n);  
})(foo);  
console.log(foo.n);
```

1 2 3

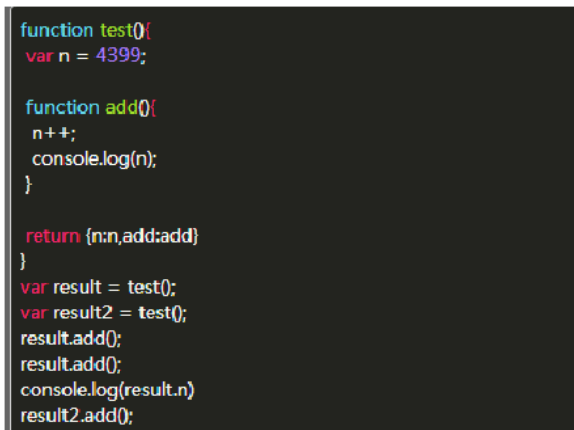
然后即使立即执行函数没有传入 foo 对象作为引用形参，在函数中也能访问到父级的参数 n，修改也有效

例子：已知函数 fn 执行需要 3 个参数。请实现函数 partial，调用之后满足如下条件：

- 1、返回一个函数 result，该函数接受一个参数
- 2、执行 result(str3)，返回的结果与 fn(str1, str2, str3) 一致

```
function partial(fn, str1, str2) {  
    return function(str3) {  
        return fn(str1, str2, str3);  
    }  
}
```

例题：



```
function test(){  
    var n = 4399;  
  
    function add(){  
        n++;  
        console.log(n);  
    }  
  
    return {n:n,add:add}  
}  
var result = test();  
var result2 = test();  
result.add();  
result.add();  
console.log(result.n)  
result2.add();
```

test 函数返回的 {n:n, add:add} 中，n 是新创建的变量，add 是原函数的引用。

执行两次 test 函数后会生成 **两个** 不会互相干预的对象。

所以结果是 4400 4401 4399 4400

6.6. 垃圾回收机制

(1) 在 javascript 中，如果一个对象不再被引用，那么这个对象就会被回收；

(2) 如果两个对象互相引用，而不再被第 3 者所引用，那么这两个互相引用的对象也会被回收。

Javascript 垃圾回收方法：

标记清除 (mark and sweep)

这是 JavaScript 最常见的垃圾回收方式，当变量进入执行环境的时候，比如函数中声明一个变量，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”。

垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了

引用计数 (reference counting)：

在低版本 IE 中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加 1，如果该变量的值变成了另外一个，则这个值得引用次数减 1，当这个值的引用次数变为 0 的时候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为 0 的值占用的空间。

在 IE 中虽然 JavaScript 对象通过标记清除的方式进行垃圾回收，但 BOM 与 DOM 对象却是通过引用计数回收垃圾的，也就是说只要涉及 BOM 及 DOM 就会出现循环引用问题。

哪些操作会造成内存泄漏？

内存泄漏指任何对象在您不再拥有或需要它之后仍然存在。

1. 垃圾回收器定期扫描对象，并计算引用了每个对象的其他对象的数量。如果一个对象的引用数量为 0（没有其他对象引用过该对象），或对该对象的惟一引用是循环的，那么该对象的内存即可回收。

2. setTimeout 中调用了调用 setTimeout 的函数会引发内存泄漏。

```
var call = function(x) {
  console.log(x++);
  setTimeout(function() {
    call(x);
  }, 1)
}
call(0);
```

3. 闭包、控制台日志、循环（在两个对象彼此引用且彼此保留时，就会产生一个循环）

7. 异步、同步、单线程

1. 同步和异步

```
console.log(100)
setTimeout(function () {
  console.log(200)
}, 1000)
console.log(300)
```

异步不阻塞代码执行，100、300、200，如上，setTimeout 是异步

同步阻塞代码执行，100、200、300，如下，alert 是同步

```
console.log(100)
alert(200) // 1秒钟之后点击确认
console.log(300)
```

在标准的 JavaScript 中，Ajax 异步执行调用基于下面哪一个机制才能实现？

Event 和 callback

Deferral 和 promise

不是多线程操作、多 CPU 核

2. 前端使用异步的场景

它们都需要等待，所以为了避免阻塞就采用异步

- 定时任务：setTimeout，setInterval
- 网络请求：ajax 请求，动态 加载
- 事件绑定

Ajax

```
console.log('start')
$.get('./data1.json', function (data1) {
  console.log(data1)
})
console.log('end')
```

Img

```
console.log('start')
var img = document.createElement('img')
img.onload = function () {
  console.log('loaded')
}
img.src = '/xxx.png'
console.log('end')
```

事件绑定

```
console.log('start')
document.getElementById('btn1').addEventListener('click', function () {
  alert('clicked')
})
console.log('end')
```

3. 异步和单线程

js 之所以是异步的，因为它是单线程的

```
console.log(100)
setTimeout(function () {
  console.log(200)
})
console.log(300)
```

setTimeout 这种异步的代码会先拿出来放在一边，等待主要的代码执行完之后再执行放在一边的异步代码。

- 执行第一行，打印 100
- 执行setTimeout后，传入setTimeout的函数会被暂存起来，不会立即执行（单线程的特点，不能同时干两件事）
- 执行最后一行，打印 300
- 待所有程序执行完，处于空闲状态时，会立马看有没有暂存起来的要执行。
- 发现暂存起来的setTimeout中的函数无需等待时间，就立即来过来执行

如果 setTimeout 设置了时间如 1000ms，则暂存的时候记下 1s 后解封，要去执行它的时候，它必须是处于解封状态的。

网络请求异步情况下，当请求返回时才会解封

事件绑定情况下，当事件点击的时候才会解封

4. 异步实现

每一个任务有一个回调函数，前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务执行完毕就执行，所以程序的执行顺序与任务的排序顺序是不一致的。

回调时，被回调的函数会被放在 event loop 里，等待线程里的任务执行完后才执行 event loop 里的代码。

因此，上述代码会先把线程里的 console.log('first')执行完后，再执行 event loop 里的 console.log('second')。

首先全部输出 first，然后全部输出 second

```
function foo()
{
  console.log('first');
  setTimeout(function() {
    console.log('second');
  }, 5);
}

for (var i
= 0; i < 439999999; i++) {
  foo();
}
```

异步的方法：

a 回调函数

b 事件监听：采用事件驱动模式，任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

c 观察者模式

d promise 对象：每一个异步任务返回一个 promise 对象，该对象有一个 then 方法，允许指定回调函数，回调函数，这是异步编程最基本的方法。

事件监听，另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

发布/订阅，上一节的"事件"，完全可以理解成"信号"。

Promises 对象，Promises 对象是 CommonJS 工作组提出的一种规范，目的是为异步编程提供统一接口。

<http://www.ruanyifeng.com/blog/2012/12/asynchronous%E7%BB%9C%E7%BB%9Cjavascript.html>

5. 异步加载和延迟加载

1. 异步加载的方案： 动态插入 script 标签 2. 通过 ajax 去获取 js 代码，然后通过 eval 执行 3. script 标签上添加 defer 或者 async 属性 4. 创建并插入 iframe，让它异步执行 js 5. 延迟加载：有些 js 代码并不是页面初始化的时候就立刻需要的，而稍后的某些情况才需要的。

实现延迟加载的方式有：

1. 标签放在底部

将<script>节点放置在</body>之前，这样 js 脚本会在页面显示出来之后再加载。

2. Defer 和 Async 属性

使用 script 标签的 defer 和 async 属性

Defer 属性

defer 是在 html4.0 中定义的(以前只有 IE 支持，现在除了欧朋基本都支持)

html5 规范要求脚本按照出现的顺序执行，对应的 js 文件在页面解析到<script>标签时开始下载，但不会执行，直到 DOM 加载完成，即 onload 事件触发前才会执行

当一个带有 defer 属性的 JavaScript 文件下载时，它不会阻塞浏览器的其他进程，因此这类文件可以与其他资源文件一起并行下载。

```
<script type="text/javascript" defer="defer" src="example.js">
```

```
<script type="text/javascript" defer="defer" src="example.js">
```

缺点：并不是所有浏览器支持该属性

async 属性

async 是 html5 新增的属性，IE10 和其他浏览器都支持该属性

同 defer 一样，不会阻塞其余资源的加载，也不会影响页面的加载，但 js 一旦加载好了就会执行，所以很有可能不是按照原本的顺序来执行

3. 动态脚本元素

DOM 动态创建<script>

```
var script = document.createElement('script');
```

```
script.type = "text/javascript";
```

```
script.src = "script.js";
```

```
document.getElementsByTagName('head')[0].appendChild(script);
```

可以监听脚本是否加载成功

```
script.onload = function() {
```

```
    alert('script loaded!');
```

```
};
```

/*IE 中的监听方式不一样*/

```
script.onreadystatechange = function() {
```

```
    if(script.readyState == 'load' || script.readyState == 'complete') {
```

```
        script.onreadystatechange = null;
```

```
        alert('script loaded');
```

```
    }
```

```
};
```

4. 使用 XMLHttpRequest (XHR)

```
var xhr = new XMLHttpRequest();
```

```
xhr.open('get', 'script.js', true);
```

```
xhr.onreadystatechange = function() {
```

```
    if(xhr.readyState == 4) {
```

```
        if(xhr.status >= 200 && xhr.status < 300 || xhr.status == 304) {
```

```

        var script = document.createElement('script');
        script.type = "text/javascript";
        script.text = xhr.responseText;
        document.body.appendChild(script);
    }
}
};

```

xhr.send(null);

优点:

可以下载不立即执行的 javascript 代码;

同样的代码在所有现代浏览器中都可以正常运行。

缺点:

javascript 文件必须和页面放置在同一个域中, 不能从 CDN 下载, 所以大型网页通常都不采取 XHR 脚本注入技术。

总结:

减少 JavaScript 对性能的影响有以下几种方法:

将所有的<script>标签放到页面底部, 也就是</body>闭合标签之前, 这能确保在脚本执行前页面已经完成了渲染。

尽可能地合并脚本。页面中的<script>标签越少, 加载也就越快, 响应也越迅速。无论是外链脚本还是内嵌脚本都是如此。

采用无阻塞下载 JavaScript 脚本的方法:

使用<script>标签的 defer 属性 (仅适用于 IE 和 Firefox 3.5 以上版本);

使用动态创建的<script>元素来下载并执行代码;

使用 XHR 对象下载 JavaScript 代码并注入页面中。

8.DOM

从 js 基础到浏览器执行→DOM 操作有哪些→

8.1. js 基础知识只是我们知道怎么去发布命令, js-web-api 是浏览器知道怎么去执行命令

- JS基础知识: ECMA 262 标准

- JS-Web-API: W3C 标准

比如:

- 页面弹框是 window.alert(123), 浏览器需要做:

- 定义一个 window 全局变量, 对象类型

- 给它定义一个 alert 属性, 属性值是一个函数

再比如:

- 获取元素 document.getElementById(id), 浏览器需要:

- 定义一个 document 全局变量, 对象类型

- 给它定义一个 getElementById 的属性, 属性值是一个函数

基础知识和 js-web-api 的关系

- 但是W3C标准没有规定任何JS基础相关的东西

- 不管什么变量类型、原型、作用域和异步

- 只管定义用于浏览器中JS操作页面的API和全局变量

总结

- 常说的JS（浏览器执行的JS）包含两部分：
- JS基础知识（ECMA262标准）
- JS-Web-API（W3C标准）

8.2. 全面考虑，js 内置的全局函数和对象有哪些？（OAB NFS DER）

- 之前讲过的 Object Array Boolean String Math JSON 等
- 刚刚提到的 window document

•接下来还有继续讲到的所有未定义的全局变量，如 navigator.userAgent
setTimeout 是 window 的一个方法，如果把 window 当做全局对象来看待的话，它就是全局函数。严格来讲，它不是。全局函数与内置对象的属性或方法不是一个概念。全局函数它不属于任何一个内置对象。JavaScript 中包含以下 7 个全局函数 escape()、eval()、isFinite()、isNaN()、parseFloat()、parseInt()、unescape()。
Javascript 全局函数

函数	描述
decodeURI()	解码某个编码的 URI。
decodeURIComponent()	解码一个编码的 URI 组件。
encodeURIComponent()	把字符串编码为 URI。
encodeURIComponent()	把字符串编码为 URI 组件。
escape()	对字符串进行编码。
eval()	计算 JavaScript 字符串，并把它作为脚本代码来执行。
getClass()	返回一个 JavaScript 的 JavaClass。
isFinite()	检查某个值是否为有穷大的数。
isNaN()	检查某个值是否是数字。
Number()	把对象的值转换为数字。
parseFloat()	解析一个字符串并返回一个浮点数。
parseInt()	解析一个字符串并返回一个整数。
String()	把对象的值转换为字符串。
unescape()	对由 escape() 编码的字符串进行解码。

8.3. DOM 的本质——数据结构：树

将 HTML 结构化为树形结构，让浏览器、js 可识别

浏览器把拿到的 html 代码结构化为一个浏览器能识别并且 js 可操作的一个模型而已。

8.3. 1DOM 节点操作：获取 DOM 节点以及节点的 property 和 Attribute

8.3.1.1 获取 DOM 节点

```
var div1 = document.getElementById('div1') // 元素
var divList = document.getElementsByTagName('div') // 集合
console.log(divList.length)
console.log(divList[0])

var containerList = document.getElementsByClassName('.container') // 集合
var pList = document.querySelectorAll('p') // 集合
```

等号左边的变量本质上都是 js 对象，这就是 js 可操作

8.3.1.2

property

```
1 var pList = document.querySelectorAll('p')
2 var p = pList[0]
3 console.log(p.style.width) // 获取样式
4 p.style.width = '100px' // 修改样式
5 console.log(p.className) // 获取 class
6 p.className = 'p1' // 修改 class
7
8 // 获取 nodeName 和 nodeType
9 console.log(p.nodeName)
10 console.log(p.nodeType)
```

浏览器就是这么规定的可以使用这些属性，property 是 js 对象的标准属性

attr 是文档的标签的属性

```
1 var pList = document.querySelectorAll('p')
2 var p = pList[0]
3 p.getAttribute('data-name')
4 p.setAttribute('data-name', 'imooc')
5 p.getAttribute('style')
6 p.setAttribute('style', 'font-size:30px;')
```

```
<li class="course-item">
  <a href="http://www.imooc.com/learn/746">
    
```

DOM 节点的 attr 和 property 有何区别？

修改和获取

- property 只是一个 JS 对象的属性的修改
- Attribute 是对 html 标签属性的修改

8.3.2 DOM 结构操作

- 新增节点
- 获取父元素
- 获取子元素
- 删除节点

新增

```
1 var div1 = document.getElementById('div1')
2 // 添加新节点
3 var p1 = document.createElement('p')
4 p1.innerHTML = 'this is p1'
5 div1.appendChild(p1) // 添加新创建的元素
6 // 移动已有节点
7 var p2 = document.getElementById('p2')
8 div1.appendChild(p2)
```

获取父元素和子元素

```
1 var div1 = document.getElementById('div1')
2 var parent = div1.parentElement
3
4 var child = div1.childNodes
5 div1.removeChild(child[0])
```

DOM 操作——怎样添加、移除、移动、复制、创建和查找节点。

1) 创建新节点

createDocumentFragment() //创建一个 DOM 片段

createElement() //创建一个具体的元素

createTextNode() //创建一个文本节点

2) 添加、移除、替换、插入

appendChild() removeChild() replaceChild() insertBefore() //并没有 insertAfter()

3) 查找

getElementsByName() //通过标签名称

getElementsByTagName() //通过元素的 Name 属性的值(IE 容错能力较强, 会得到一个数组, 其中包括 id 等于 name 值的)

getElementById() //通过元素 Id, 唯一性

8.4. BOM 操作

检测浏览器、历史记录、url 分析 → src 和 href

8.4.1 如何检测浏览器、系统的类型

userAgent 是一个字符串, 检测里面有没有 chrome 就是检测是不是 chrome 浏览器

navigator & screen

```
1 // navigator
2 var ua = navigator.userAgent
3 var isChrome = ua.indexOf('Chrome')
4 console.log(isChrome)
5
6 // screen
7 console.log(screen.width)
8 console.log(screen.height)
```

8.4.2 拆解 url 的各部分

location.

location & history

```
1 // location
2 console.log(location.href)
3 console.log(location.protocol) // 'http:' 'https:'
4 console.log(location.pathname) // '/learn/199'
5 console.log(location.search)
6 console.log(location.hash)
7
8 // history
9 history.back()
10 history.forward()
```

coding.m.imooc.com/classindex.html?cid=99&a=b#mid=100

```
> location.protocol
< "http:"
> location.host
< "coding.m.imooc.com"
> location.pathname
< "/classindex.html"
> location.search
< "?cid=99&a=b"
> location.hash
< "#mid=100"
```

```
location.href = 'http://imooc.com'
```

8.4.3 href 和 src 的区别

8.5 打开一个窗口的 js 代码: `window.open("http://www.w3school.com.cn", "window2")`

`open()` 方法可以查找一个已经存在或者新建的浏览器窗口。

语法:

```
window.open([URL], [窗口名称], [参数字符串])
```

参数说明:

URL: 可选参数, 在窗口中要显示网页的网址或路径。如果省略这个参数, 或者它的值是空字符串, 那么窗口就不显示任何文档。

窗口名称: 可选参数, 被打开窗口的名称。

1. 该名称由字母、数字和下划线字符组成。
2. `"_top"`、`"_blank"`、`"_self"` 具有特殊意义的名称。
 - `_blank`: 在新窗口显示目标网页
 - `_self`: 在当前窗口显示目标网页
 - `_top`: 框架网页中在上部窗口中显示目标网页
3. 相同 name 的窗口只能创建一个, 要想创建多个窗口则 name 不能相同。
4. name 不能包含有空格。

参数字符串: 可选参数, 设置窗口参数, 各参数用逗号隔开。

8.6 window 对象的理解

就是全局对象, 声明的对象都会在里面。

```
if (!(a in window)) {  
    var a = 1;  
}
```

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

由于声明提前, 所以 window 中有 a, `a in window` 为真, 所以判断条件为假, 所以 a 的赋值不成立, 所以打印 undefined。

Ps:

`!"a"`: `"a"` 为 true, 整体为 false

! 的优先级高于 window, 如果判断条件是 `!a in window`, 那因为 a 是 undefined, 所以 !a 是 true, 所以整体就是 true in window, 为假, 还是不执行 a 的赋值语句, 还是打印 undefined。

与浏览列表有关的对象: history (历史) location

windows 是浏览器的一个实例, 也代表全局

location 包含有关当前 url 的信息

navigator 包含有关浏览器的信息

screen 包含浏览器窗口外部的显示器的信息, 如像素的宽度和高度

history 包含用户访问过的 url

history:

length 返回浏览器历史列表中的 URL 数量

back() 加载 history 列表中的前一个 URL

forward() 加载 history 列表中的下一个 URL

go() 加载 history 列表中的某个具体页面。

9. 事件

标准事件模型的事件机制：事件捕获→事件处理→事件冒泡

先事件捕获从 windows → document 往下级直到特定的事件节点，然后进行事件处理，再事件冒泡，从特定节点往上级

把鼠标移到按钮并点击时，会产生一串什么样的事件

hover → focus → active

悬停 → 聚焦 → 响应

事件是 submit，onsubmit 只是属性名，或是触发的函数

要在 10 秒后调用 checkState，下列哪个是正确的

正确答案: B 你的答案: D (错误)

window.setTimeout(checkState, 10);

window.setTimeout(checkState, 10000);

window.setTimeout(checkState(), 10);

window.setTimeout(checkState(), 10000);

checkState 加了圆括弧相当于函数表达式，会立即执行，执行的结果作为返回值传递给 setTimeout。

1. 通用事件绑定

普通浏览器：addEventListener

IE 低版本：attachEvent

```
1 var btn = document.getElementById('btn1')
2 btn.addEventListener('click', function (event) {
3     console.log('clicked')
4 })
5
6 function bindEvent(elem, type, fn) {
7     elem.addEventListener(type, fn)
8 }
9 var a = document.getElementById('link1')
10 bindEvent(a, 'click', function(e) {
11     e.preventDefault() // 阻止默认行为
12     alert('clicked')
13 })
```

a 标签有默认行为就是点击以后打开链接，所以需要阻止默认行为

- IE 低版本使用 attachEvent 绑定事件，和 W3C 标准不一样
- IE 低版本使用量以非常少，很多网站都早已不支持
- 建议对 IE 低版本的兼容性：了解即可，无需深究

绑定事件时默认第三个参数是 false 事件冒泡，由内向外；显示设置为 true 就是事件捕获，由外向内；

2. 事件→事件代理

2.1 用事件冒泡实现点击弹出不同信息


```

1 <body>
2   <div id="div1">
3     <p id="p1">激活</p>
4     <p id="p2">取消</p>
5     <p id="p3">取消</p>
6     <p id="p4">取消</p>
7   </div>
8   <div id="div2">
9     <p id="p5">取消</p>
10    <p id="p6">取消</p>
11  </div>
12 </body>

```

```

1 var p1 = document.getElementById('p1')
2 var body = document.body
3 bindEvent(p1, 'click', function (e) {
4   e.stopPropagation()
5   alert('激活')
6 })
7 bindEvent(body, 'click', function (e) {
8   alert('取消')
9 })

```

无论点击哪里，都会弹出取消，但是 p1 上阻止了事件冒泡，所以只会弹出激活，不会冒泡上去

Ps: stopPropagation() 阻止冒泡，preventDefault() 阻止事件的默认操作。

DOM 中的事件对象：(符合 W3C 标准)

preventDefault() 取消事件默认行为

stopImmediatePropagation() 取消事件冒泡同时阻止当前节点上的事件处理程序被调用。

stopPropagation() 取消事件冒泡对当前节点无影响。

IE 中的事件对象：

cancelBubble() 取消事件冒泡

returnValue() 取消事件默认行为

IE 与火狐的事件机制有什么区别？ 如何阻止冒泡？

1. 我们在网页中的某个操作（有的操作对应多个事件）。例如：当我们点击一个按钮就会产生一个事件。是可以被 JavaScript 侦测到的行为。
2. 事件处理机制：IE 是事件冒泡；firefox 同时支持两种事件模型，也就是：捕获型事件和冒泡型事件。
3. ev.stopPropagation(); 注意旧 ie 的方法 ev.cancelBubble = true;

2.2 事件代理

2.2.1 什么是事件代理

事件代理 (Event Delegation)，又称之为事件委托。是 JavaScript 中常用绑定事件的常用技巧。顾名思义，“事件代理”即是把原本需要绑定的事件委托给父元素，让父元素担当事件监听的职务。事件代理的原理是 DOM 元素的事件冒泡。使用事件代理的好处是可以提高性能。

2.2.2 理实现点击 a 标签时弹出特有内容

```

1 <div id="div1">
2   <a href="#">a1</a>
3   <a href="#">a2</a>
4   <a href="#">a3</a>
5   <a href="#">a4</a>
6   <!-- 会随时新增更多 a 标签 -->
7 </div>

```

```

1 var div1 = document.getElementById('div1')
2 div1.addEventListener('click', function (e) {
3   var target = e.target
4   if (target.nodeName === 'A') {
5     alert(target.innerHTML)
6   }
7 })

```

由于不确定 a 标签的个数，所以只能在 div 上绑定事件，然后通过点击事件传参 e 的 target 属性可以获取具体是在哪一个 a 标签上进行的点击事件。

2.3 完善通用绑定事件的函数

```

1 function bindEvent(elem, type, selector, fn) {
2     if (fn == null) {
3         fn = selector
4         selector = null
5     }
6     elem.addEventListener(type, function (e) {
7         var target
8         if (selector) {
9             target = e.target
10            if (target.matches(selector)) {
11                fn.call(target, e)
12            }
13        } else {
14            fn(e)
15        }
16    })
17 }

```

```

1 // 使用代理
2 var div1 = document.getElementById('div1')
3 bindEvent(div1, 'click', 'a', function (e) {
4     console.log(this.innerHTML)
5 })
6
7 // 不适用代理
8 var a = document.getElementById('a1')
9 bindEvent(div1, 'click', function (e) {
10     console.log(a.innerHTML)
11 })

```

第三个参数是使用代理的时候使用，表示用 elem 元素去代理 selector 元素的事件(比如右例中的 div1 代理 a)，如果不想使用代理就只用传入三个参数，然后第三个参数的值和第四个空参数的值交换这样 fn 就有正确值。target.matches(selector)就是看当前点击的目标是不是我们为之代理的目标，如果是就执行 fn 并改变 this 指向，使用 call 函数传入对象和参数，这样右例中的 this 才能正确绑定在 target 对象上，而不是函数调用式时 this 只能指向 window。

代理的好处：代码简洁、减少浏览器内存占用。

不用写那么多的事件绑定，事件绑定多了自然也就占用的内存多了。

在一个表单中，如果想要给输入框添加一个输入验证，可以用下面的哪个事件实现？

正确答案: A D 你的答案: B (错误)

hover(over, out)

keypress (fn)

change()

change(fn)

页面有一个按钮 button id 为 button1，通过原生的 js 如何禁用？

```
document.getElementById("button1").setAttribute("disabled", "true");
```

```
document.getElementById("button1").disabled=true;
```

document.getElementById("button1").Readonly=true;不行，因为 Readonly 只针对 input(text/password)和 textarea 有效，而 disabled 对于所有的表单元素有效，包括 select,radio,checkbox,button 等。

fn 用于校验

A 模仿悬停事件，即当鼠标移动到一个匹配的元素上面时，会触发指定的第一个函数。当鼠标移出这个元素时，会触发指定的第二个函数。

B 在每一个匹配元素的 keypress 事件中绑定一个处理函数

C 调用执行绑定到 change 事件的所有函数，包括浏览器的默认行为。

D 给所有的文本框增加输入验证

属性	当以下情况发生时，出现此事件	FF	N	IE
onabort	图像加载被中断	1	3	4
onblur	元素失去焦点	1	2	3
onchange	用户改变域的内容	1	2	3
onclick	鼠标点击某个对象	1	2	3
ondblclick	鼠标双击某个对象	1	4	4
onerror	当加载文档或图像时发生某个错误	1	3	4
onfocus	元素获得焦点	1	2	3
onkeydown	某个键盘的键被按下	1	4	3
onkeypress	某个键盘的键被按下或按住	1	4	3
onkeyup	某个键盘的键被松开	1	4	3
onload	某个页面或图像被完成加载	1	2	3
onmousedown	某个鼠标按键被按下	1	4	4
onmousemove	鼠标被移动	1	6	3
onmouseout	鼠标从某元素移开	1	4	4
onmouseover	鼠标被移到某元素之上	1	2	3
onmouseup	某个鼠标按键被松开	1	4	4
onreset	重置按钮被点击	1	3	4
onresize	窗口或框架被调整尺寸	1	4	4
onselect	文本被选定	1	2	3
onsubmit	提交按钮被点击	1	2	3
onunload	用户退出页面	1	2	3

10. ajax

1. 原生实现：XMLHttpRequest。完全默写出这 11 行代码

```

1  var xhr = new XMLHttpRequest()
2  xhr.open("GET", "/api", false)
3  xhr.onreadystatechange = function () {
4      // 这里的函数异步执行，可参考之前 JS 基础中的异步模块
5      if (xhr.readyState == 4) {
6          if (xhr.status == 200) {
7              alert(xhr.responseText)
8          }
9      }
10 }
11 xhr.send(null)

```

创建 ajax 过程

- (1) 创建 XMLHttpRequest 对象，也就是创建一个异步调用对象。
- (2) 创建一个新的 HTTP 请求，并指定该 HTTP 请求的方法、URL 及验证信息。
- (3) 设置响应 HTTP 请求状态变化的函数。
- (4) 发送 HTTP 请求。
- (5) 获取异步调用返回的数据。
- (6) 使用 JavaScript 和 DOM 实现局部刷新。

函数 onreadystatechange 是异步执行的，所以是 send 先执行，然后随时监听 readyState 的状态，如果改变就触发 onreadystatechange 函数

- IE 低版本使用 ActiveXObject，和 W3C 标准不一样
- IE 低版本使用量以非常少，很多网站都早已不支持
- 建议对 IE 低版本的兼容性：了解即可，无需深究
- 如果遇到对 IE 低版本要求苛刻的面试，果断放弃

2. readyState 状态码说明

- 0 - (未初始化) 还没有调用send()方法
- 1 - (载入) 已调用send()方法, 正在发送请求
- 2 - (载入完成) send()方法执行完成, 已经接收到全部响应内容
- 3 - (交互) 正在解析响应内容
- 4 - (完成) 响应内容解析完成, 可以在客户端调用了

3. status 状态码

- 2xx - 表示成功处理请求。如 200
- 3xx - 需要重定向, 浏览器直接跳转
- 4xx - 客户端请求错误, 如 404
- 5xx - 服务器端错误

4. GET 和 POST 的区别, 何时使用 POST?

GET: 一般用于信息获取, 使用 URL 传递参数, 对所发送信息的数量也有限制, 一般在 2000 个字符

POST: 一般用于修改服务器上的资源, 对所发送的信息没有限制。

GET 方式需要使用 Request.QueryString 来取得变量的值, 而 POST 方式通过 Request.Form 来获取变量的值, 也就是说 Get 是通过地址栏来传值, 而 Post 是通过提交表单来传值。

然而, 在以下情况中, 请使用 POST 请求: 无法使用缓存文件 (更新服务器上的文件或数据库); 向服务器发送大量数据 (POST 没有数据量限制); 发送包含未知字符的用户输入时, POST 比 GET 更稳定也更可靠

3. AJAX 优点和缺点?

3.1 优点

- 1、最大的一点是页面动态不刷新 (局部刷新), 用户的体验非常好。就是能在不更新整个页面的前提下维护数据。这使得 Web 应用程序更为迅捷地回应用户动作, 并避免了在网络上发送那些没有改变过的信息
- 2、使用异步方式与服务器通信, 具有更加迅速的响应能力, 提升了用户体验。
- 3、可以把以前一些服务器负担的工作转嫁到客户端, 利用客户端闲置的能力来处理, 减轻服务器和带宽的负担, 节约空间和带宽租用成本。并且减轻服务器的负担, ajax 的原则是 “按需取数据”, 可以最大程度的减少冗余请求, 和响应对服务器造成的负担, 即优化了浏览器和服务器之间的传输, 减少不必要的数据往返, 减少了带宽占用。
- 4、基于标准化的并被广泛支持的技术, 不需要下载插件或者小程序。

3.2 缺点

- 1、ajax 不支持浏览器 back 按钮。
- 2、安全问题 AJAX 暴露了与服务器交互的细节。
- 3、对搜索引擎的支持比较弱。
- 4、破坏了程序的异常机制。
- 5、不容易调试。

3.3 Ajax 在 IE 下使用的缓存问题

在 IE 浏览器下, 如果请求的方法是 GET, 并且请求的 URL 不变, 那么这个请求的结果就会被缓存。解决这个问题的办法可以通过实时改变请求的 URL, 只要 URL 改变, 就不会被缓存, 可以通过在 URL 末尾添加上随机的时间戳参数 ('t' = + new Date().getTime()) 或者: open('GET', 'demo.php?rand='+Math.random(), true);

3.4 Ajax 请求的页面历史记录状态问题

3.4.1 可以通过锚点来记录状态, location.hash。让浏览器记录 Ajax 请求时页面状态的变化。

hashchange 事件是在浏览器 URL 中 hash 发生变化后触发的事件 (事件触发后会在浏览器历史记录中添加一条记录), URL 中 # 后的内容就是 hash, 它的变化所触发的 hashchange 事件与 ajax 搭配最多。按我的理解, 因为

hash 变化并不会向服务器发生请求，所以如果没有 hashchange 事件，当我们点击浏览器前进和后退按钮时，服务器无法作出反应（因为服务器无法收到请求），有了这个事件，就可以使用 js 触发 ajax 的新请求让服务器作出响应。hashchange 事件本身只是监测 hash 的变化，我认为目前其主要意义就是与 ajax 搭配使用从而使得在 ajax 下历史记录前进后退按钮依然有效。可以使用以下简单的代码体会下：

```
<body onhashchange="alert('Got a hashchange.');">
    <a href="#foo">Click me foo</a>
    <a href="#bar">Click me bar</a>
</body>
```

3.4.2 还可以通过 HTML5 的 history.pushState，来实现浏览器地址栏的无刷新改变

<http://www.zhangxinxu.com/wordpress/2013/06/html5-history-api-pushstate-replacestate-ajax/>

3.5 flash 和 ajax

Flash 适合处理多媒体、矢量图形、访问机器；对 CSS、处理文本上不足，不容易被搜索。

Ajax 对 CSS、文本支持很好，支持搜索；多媒体、矢量图形、机器访问不足。

共同点：与服务器的无刷新传递消息、用户离线和在线状态、操作 DOM

1. Ajax 的优势：1. 可搜索性(但还是弱) 2. 开放性 3. 费用 4. 易用性 5. 易于开发。

2. Flash 的优势：1. 多媒体处理 2. 兼容性 3. 矢量图形 4. 客户端资源调度

3. Ajax 的劣势：1. 它可能破坏浏览器的后退功能 2. 使用动态页面更新使得用户难于将某个特定的状态保存到收藏夹中，不过这些都有相关方法解决。

4. Flash 的劣势：1. 二进制格式 2. 格式私有 3. flash 文件经常会很大，用户第一次使用的时候需要忍耐较长的等待时间 4. 性能问题

Flash:

flash 有自己的一套安全策略，服务器可以通过 crossdomain.xml 文件来声明能被哪些域的 SWF 文件访问，SWF 也可以通过 API 来确定自身能被哪些域的 SWF 加载。当跨域访问资源时，例如从域 baidu.com 请求域 google.com 上的数据，我们可以借助 flash 来发送 HTTP 请求。首先，修改域 google.com 上的 crossdomain.xml (一般存放在根目录，如果没有需要手动创建)，把 baidu.com 加入到白名单。其次，通过 Flash URLLoader 发送 HTTP 请求，最后，通过 Flash API 把响应结果传递给 JavaScript。Flash URLLoader 是一种很普遍的跨域解决方案，不过需要支持 iOS 的话，这个方案就不可行了。

4. 什么是 ajax

AJAX 是 “Asynchronous JavaScript and XML” 的缩写。它是指一种主要使用脚本操纵 HTTP 的 Web 应用架构，它的主要特点是使用脚本操纵 HTTP 和 Web 服务器进行数据交换，不会导致页面重载。

Ajax 包含下列技术：

基于 web 标准 (standards-based presentation) XHTML+CSS 的表示；

使用 DOM (Document Object Model) 进行动态显示及交互；

使用 XML 和 XSLT 进行数据交换及相关操作；

使用 XMLHttpRequest 进行异步数据查询、检索；

使用 JavaScript 将所有的东西绑定在一起。

5. 请介绍一下 XMLHttpRequest 对象？

Ajax 的核心是 JavaScript 对象 XMLHttpRequest。该对象在 Internet Explorer 5 中首次引入，它是一种支持异步请求的技术。简而言之，XmlHttpRequest 使您可以使用 JavaScript 向服务器提出请求并处理响应，而不阻塞用户。通过 XMLHttpRequest 对象，Web 开发人员可以在页面加载以后进行页面的局部更新。IE 中通过 new ActiveXObject() 得到，Firefox 中通过 new XMLHttpRequest() 得到。

6. AJAX 应用和传统 Web 应用有什么不同？

在传统的 Javascript 编程中，如果想得到服务器端数据库或文件上的信息，或者发送客户端信息到服务器，需要建立一个 HTML form 然后 GET 或者 POST 数据到服务器端。用户需要点击” Submit” 按钮来发送或者接受数据信息，然后等待服务器响应请求，页面重新加载。

因为服务器每次都会返回一个新的页面， 所以传统的 web 应用有可能很慢而且用户交互不友好。

使用 AJAX 技术， 就可以使 Javascript 通过 XMLHttpRequest 对象直接与服务器进行交互。

通过 HTTP Request， 一个 web 页面可以发送一个请求到 web 服务器并且接受 web 服务器返回的信息(不用重新加载页面)，展示给用户的还是通一个页面，用户感觉页面刷新，也看不到到 Javascript 后台进行的发送请求和接受响应。

7. AJAX 请求总共有多少种 CALLBACK

Ajax 请求总共有八种 Callback

onSuccess

onFailure

onUninitialized

onLoading

onLoaded

onInteractive

onComplete

onException

\$.post(url)是 ajax 请求；

ajax 的事件是：

ajaxComplete(callback)

ajaxError(callback)

ajaxSend(callback)

ajaxStart(callback)

ajaxStop(callback)

ajaxSuccess(callback)

jquery ajax 中都支持哪些返回类型:xml、html、jsonp、json

11. 跨域

什么是跨域

•浏览器有同源策略，不允许 ajax 访问其他域接口

•<http://www.yourname.com/page1.html>

•<http://m.imooc.com/course/ajaxcoursecom?cid=459>

•跨域条件：协议、域名、端口，有一个不同就算跨域

为什么要有同源限制？

我们举例说明：比如一个黑客程序，他利用 iframe 把真正的银行登录页面嵌到他的页面上，当你使用真实的用户名，密码登录时，他的页面就可以通过 Javascript 读取到你的表单中 input 中的内容，这样用户名，密码就轻松到手了。

11.1 可跨域的三个标签

•用于打点统计，统计网站可能是其他域

•<link> <script>可以使用CDN，CDN的也是其他域

•<script>可以用于JSONP，马上讲解

:因为网页可以从任何网页中加载图片，而不用担心跨域。请求数据通过字符串形式发送，而响应可以是任何内容。这种方法，1) 只能发送 get 请求。2) 浏览器无法获取响应数据。3) 只适用于浏览器与服务器之间的单向通信。img 还有一个优势就是兼容性非常好，它是一个古老的标签

跨域注意事项

- 所有的跨域请求都必须经过信息提供方允许
- 如果未经允许即可获取，那是浏览器同源策略出现漏洞

11.2 jsonp

原理是：动态插入 script 标签，通过 script 标签引入一个 js 文件，这个 js 文件载入成功后会执行我们在 url 参数中指定的函数，并且会把我们需要的 json 数据作为参数传入。

JSONP实现原理

- 加载 <http://coding.mimooc.com/classindex.html>
- 不一定服务器端真正有一个 classindex.html 文件
- 服务器可以根据请求，动态生成一个文件，返回
- 同理于 `<script src="http://coding.mimooc.com/api.js">`
- 例如你的网站要跨域访问慕课网的一个接口
- 慕课给你一个地址 <http://coding.mimooc.com/api.js>
- 返回内容格式如 `callback({x:100, y:200})` (可动态生成)

```
1 <script>
2 window.callback = function (data) {
3     // 这是我们跨域得到信息
4     console.log(data)
5 }
6 </script>
7 <script src="http://coding.mimooc.com/api.js"></script>
8 <!--以上将返回 callback({x:100, y:200}) -->
```

有两部分：1) 回调函数(js 中 callback)：响应到来时在页面中使用；2) 数据(data)：传入回调函数中的 JSON 数据

首先在自己的网页中定义一个函数 callback，参数是 data

慕课网给了我们一个地址

使用 script 标签，返回的值中的命令是执行这个函数

重点是 script 可以跨过同源限制

优点是兼容性好，简单易用，支持浏览器与服务器双向通信。缺点是只支持 GET 请求，不支持 post 请求。

JSONP: json+padding (内填充)，顾名思义，就是把 JSON 填充到一个盒子里

代码示例：

11.3 CORS：服务器端设置 http header

- 另外一个解决跨域的简洁方法，需要服务器端来做
- 但是作为交互方，我们必须知道这个方法
- 是将来解决跨域问题的一个趋势

```
1 // 注意：不同后端语言的写法可能不一样
2
3 // 第二个参数填写允许跨域的域名称，不建议直接写 "*"
4 response.setHeader("Access-Control-Allow-Origin", "http://a.com, http://b.com");
5 response.setHeader("Access-Control-Allow-Headers", "X-Requested-With");
6 response.setHeader("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS");
7
8 // 接收跨域的cookie
9 response.setHeader("Access-Control-Allow-Credentials", "true");
```


后台代理方法：将后台作为代理，每次对其它域的请求转交给本域的后台，本域的后台通过模拟 http 请求去访问其它域，再将返回的结果返回给前台

服务器端对于 CORS 的支持，主要就是通过设置 Access-Control-Allow-Origin 来进行的。如果浏览器检测到相应的设置，就可以允许 Ajax 进行跨域访问。

11.4 其他跨域方法

11.4.1 设置 document.domain：只适用于主域相同子域不同；

将子域和主域的 document.domain 设为同一个主域。前提条件：这两个域名必须属于同一个基础域名！而且所用的协议，端口都要一致，否则无法利用 document.domain 进行跨域

11.4.2 使用 window.name：+iframe。

window 对象有个 name 属性，该属性有个特征：即在一个窗口(window)的生命周期内，窗口载入的所有页面都是共享一个 window.name 的，每个页面对 window.name 都有读写的权限，window.name 是持久存在一个窗口载入过的所有页面中的

应用页面创建 iframe，src 指向数据页面；数据页面把数据附加到 window.name 上；应用界面监听 iframe 的 onload 事件，在此事件中设置这个 iframe 的 src 指向本地域的代理文件；获取数据后销毁 iframe
window.name 的方法既不复杂，也能兼容到几乎所有浏览器，这真是极好的一种跨域方法。

11.4.3 使用 html5 新方法：window.postMessage(message, targetOrigin)

12. 本地存储和服务器存储

如何实现浏览器内多个标签页之间的通信：调用 localStorage、cookies 等本地存储方式

1. cookie(是一个字符串)

- 本身用于客户端和服务端通信
- 但是它有本地存储的功能，于是就被“借用”
- 使用 document.cookie = ... 获取和修改即可

例子：

The screenshot shows a web browser window displaying a React course page titled "React 高级实战 - 打造大众点评 WebApp". The page includes a navigation bar, a main content area with a "课程介绍" (Course Introduction) section, and a sidebar with "课程目录" (Course Directory). The browser's Application tab is open, showing a list of cookies. The cookies are organized into sections: Local Storage, Session Storage, IndexedDB, Web SQL, and Cookies. The Cookies section is expanded, showing a list of cookies with columns for Name, Value, Domain, Path, Expires, Size, HTTP, Security, and Sample. The cookies are as follows:

Name	Value	Dom...	Path	Expi...	Size	HTTP	Sec...	Sam...
BAIDUID	E85F65A6052F1F7...	.bai...	/	201...	44			
BDORZ	B49085EBF6F3CD...	.bai...	/	201...	37			
BDRCVF	R[eW]1Vr5u3D]	.bai...	/	Ses...	33			
BDUSS	EIoN3VWSjM4dk0...	.bai...	/	202...	197		✓	
BIDUPSID	1C4B7E00A1845B...	.bai...	/	204...	40			
CNZZDATA	1261728817	.codi...	/	201...	57			
HMACCOUNT	794BA3DCEF68B4...	.hm...	/	203...	25			
HMMVT	7a3960b6f067eb0...	.hm...	/	Ses...	48			
H_PS_PSSID	23202_1468_2109...	.bai...	/	Ses...	32			
Hm_lpvt_c1c5f01e0f4d75d5cbb16f2...	1497693549	.cod...	/	Ses...	50			
Hm_lpvt_c92536284537e1806a07ef3e...	1497705971	.mi...	/	Ses...	50			
Hm_lpvt_f0cfcccd7b1393990c78efdee...	1497693540	.jmo...	/	Ses...	50			
Hm_lpvt_c1c5f01e0f4d75d5cbb16f2e...	1497666349,1497...	.cod...	/	201...	82			
Hm_lpvt_c92536284537e1806a07ef3e6...	1497279401,1497...	.mi...	/	201...	71			
Hm_lpvt_f0cfcccd7b1393990c78efdee...	1497490608,1497...	.jmo...	/	201...	82			
IMCNDNS	0	.jmo...	/	201...	7			
PHPSESSID	qc6lqr07s0omoqp...	.img...	/	Ses...	35			
PSINO	1	.bai...	/	Ses...	6			
PSTM	1494250816	.bai...	/	208...	14			
RK	Pc1+G+6q9]	.qq...	/	202...	12			
UM_distinctid	15b8148d0342d4...	.jmo...	/	201...	72			
_cfduid	d24da4d0f93a28a...	.bai...	/	201...	51		✓	
aimx	DFZBEObYSkCAT...	.jrm...	/	202...	39			
apsid	Y2ZWQ4MjRINTdl...	.jmo...	/	201...	275			
atpsida	10c661127927e8a9...	.cnz...	/	Ses...	44			
atpsida	76a50ee4d41d5c5...	.jrm...	/	Ses...	44			
omida	1401053355_2017...	.cnz...	/	201...	30			

cookie 用于存储的缺点

- 存储量太小，只有 4KB
- 所有 http 请求都带着，会影响获取资源的效率
- API 简单，需要封装才能用 `document.cookie =`

1.2 跨域带上 cookie:

原生 ajax 请求方式：

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://xxx.com/demo/b/index.php", true);
xhr.withCredentials = true; //支持跨域发送 cookies
xhr.send();
```

jquery 的 ajax 的 post 方法请求：

```
$.ajax({
    type: "POST",
    url: "http://xxx.com/api/test",
    dataType: 'jsonp',
    xhrFields: {
        withCredentials: true
    },
    crossDomain: true,
    success: function() {
    },
    error: function() {
    }
})
```

服务器端设置：

```
header("Access-Control-Allow-Credentials: true");
header("Access-Control-Allow-Origin: http://www.xxx.com");
```

1.3 Cookie 是可以覆盖的，如果重复写入同名的 Cookie，那么将会覆盖之前的 Cookie

如果要删除某个 Cookie，只需要新建一个同名的 Cookie，并将 `maxAge` 设置为 0，并添加到 `response` 中覆盖原来的 Cookie。注意是 0 而不是负数。负数代表其他的意义。

如何删除一个 cookie

1. 将时间设为当前时间往前一点。

```
var date = new Date(); date.setDate(date.getDate() - 1); //真正的删除
setDate() 方法用于设置一个月的某一天。
```

2. expires 的设置

```
document.cookie = 'user=' + encodeURIComponent('name') + ';expires = ' + new Date(0)
```

1.4 如何保持登陆状态?如何进行图片表单验证?

答: 把登录信息如账号、密码等保存在 Cookie 中，并控制 Cookie 的有效期，下次访问时再验证 Cookie 中的登录信息即可。

保存登录信息有多种方案。最直接的是把用户名与密码都保持到 Cookie 中，下次访问时检查 Cookie 中的用户名与密码，与[数据库](#)比较。这是一种比较危险的选择，一般不把密码等重要信息保存到 Cookie 中。

还有一种方案是把密码加密后保存到 Cookie 中，下次访问时解密并与数据库比较。这种方案略微安全一些。如果不希望保存密码，还可以把登录的时间戳保存到 Cookie 与数据库中，到时只验证用户名与登录时间戳就可以了。这几种方案验证账号时都要查询数据库。

图片表单验证方法：服务器端生成验证码后一方面通过图片将验证码返回给客户端，同时在服务器端保存文本的验证码，由服务器端验证输入内容是否正确；

而把正确的验证码文本放在了客户端，这是违背了验证码的初衷的。爬虫或者是恶意程序依旧可以通过各种手段获取你嵌入在 html 文本或者保存在 cookie 中的正确验证码文本，模拟表单提交来达到攻击的目的。

1.5 cookie 优缺点

优点：极高的扩展性和可用性

1. 通过良好的编程，控制保存在 cookie 中的 session 对象的大小。
2. 通过加密和安全传输技术（SSL），减少 cookie 被破解的可能性。
3. 只在 cookie 中存放不敏感数据，即使被盗也不会有重大损失。
4. 控制 cookie 的生命期，使之不会永远有效。偷盗者很可能拿到一个过期的 cookie。

缺点：

1. `Cookie` 数量和长度的限制。每个 domain 最多只能有 20 条 cookie，每个 cookie 长度不能超过 4KB，否则会被截掉。
2. 安全性问题。如果 cookie 被人拦截了，那人就可以取得所有的 session 信息。即使加密也与事无补，因为拦截者并不需要知道 cookie 的意义，他只要原样转发 cookie 就可以达到目的了。
3. 有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务器端保存一个计数器。如果我们把这个计数器保存在客户端，那么它起不到任何作用。

1.6 提高 cookie 的安全性的几种方式

一、对保存到 cookie 里面的敏感信息必须加密

二、设置 HttpOnly 为 true

- 1、该属性值的作用就是防止 Cookie 值被页面脚本读取。
- 2、但是设置 HttpOnly 属性，HttpOnly 属性只是增加了攻击者的难度，Cookie 盗窃的威胁并没有彻底消除，因为 cookie 还是有可能传递的过程中被监听捕获后信息泄漏。

三、设置 Secure 为 true

- 1、给 Cookie 设置该属性时，只有在 https 协议下访问的时候，浏览器才会发送该 Cookie。
- 2、把 cookie 设置为 secure，只保证 cookie 与 WEB 服务器之间的数据传输过程加密，而保存在本地的 cookie 文件并不加密。如果想让本地 cookie 也加密，得自己加密数据。

四、给 Cookie 设置有效期

- 1、如果不设置有效期，万一用户获取到用户的 Cookie 后，就可以一直使用用户身份登录。
- 2、在设置 Cookie 认证的时候，需要加入两个时间，一个是“即使一直在活动，也要失效”的时间，一个是“长时间不活动的失效时间”，并在 Web 应用中，首先判断两个时间是否已超时，再执行其他操作。

1.7 Cookie 的应用：

访问者的信息一般都可以处理成 kv 键值对的形式，故可以保存在 Cookie 中，A 正确。

通过设置 Cookie 的 path 等属性，可以在特定域名或 URI 下共享 Cookie 信息，B 正确。

通过在 Cookie 中保存用户 uid、服务器会话 sid 等方法，可以记录用户登录状态，C 正确。

Cookie 是保存在用户浏览器上的小文本文件，不是数据库，也没有提供操作数据库的 API，故 D 错误。

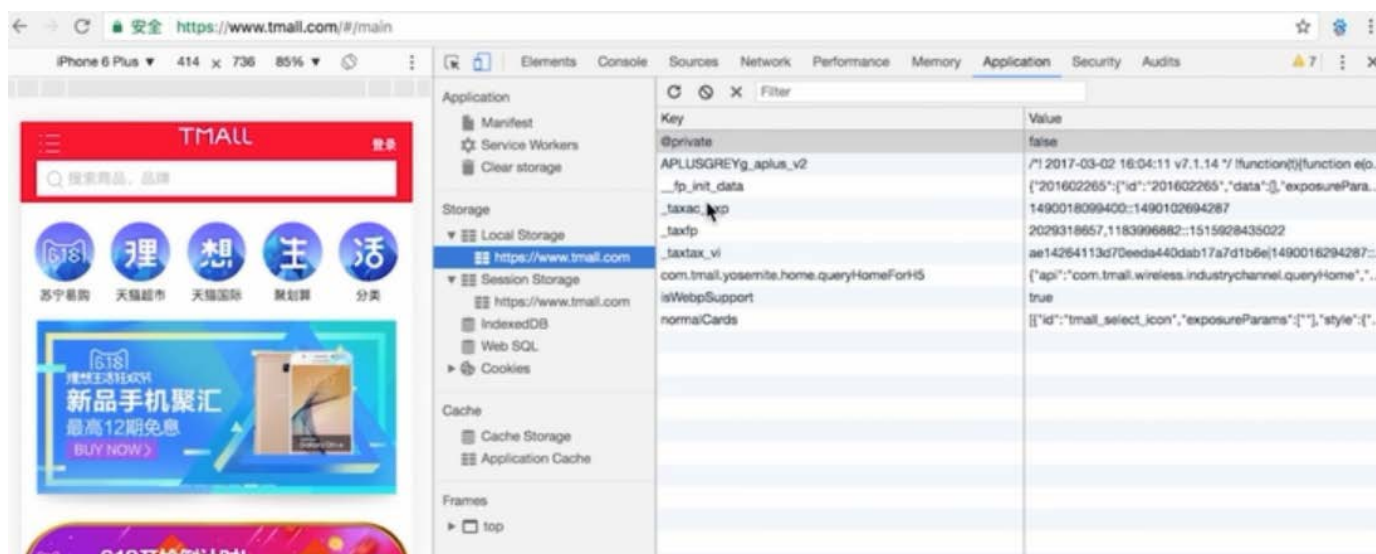
2. localStorage 和 sessionStorage

•HTML5 专门为存储而设计，最大容量 5M

•API 简单易用：

•localStorage.setItem(key, value); localStorage.getItem(key);

不用在请求中附带，所以可以做到很大
chrome 模拟



注意

•iOS safari 隐藏模式下

•localStorage.getItem 会报错

•建议统一使用 try-catch 封装

sessionStorage 是在同源的窗口（或 tab）中，始终存在的数据。也就是说只要这个浏览器窗口没有关闭，即使刷新页面或进入同源另一页面，数据仍然存在。关闭窗口后，sessionStorage 即被销毁。同时“独立”打开的不同窗口，即使是同一页面，sessionStorage 对象也是不同的。

```
sessionStorage.setItem("键名","键值");
```

```
localStorage.getItem("键名");
```

写入字段有三种方式：

```
localStorage["a"]=1;
```

```
localStorage.b=1;
```

```
localStorage.setItem("c",3);
```

读取字段也有三种方式：

```
var a= localStorage.a;
```

```
var b= localStorage["b"];
```

```
var c= localStorage.getItem("c");
```

3. cookie 和 session 的区别(本地存储和服务端存储):

cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗；考虑到安全应当使用 session。

session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能；考虑到减轻服务器性能方面，应当使用 cookie。

单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

所以建议是：

将登陆信息等重要信息存放为 SESSION

其他信息如果需要保留，可以放在 COOKIE 中

4.浏览器本地存储与服务器端存储之间的区别

其实数据既可以在浏览器本地存储，也可以在服务器端存储。

浏览器端可以保存一些数据，需要的时候直接从本地获取，sessionStorage、localStorage 和 cookie 都由浏览器存储在本地数据。

服务器端也可以保存所有用户的所有数据，但需要的时候浏览器要向服务器请求数据。

1.服务器端可以保存用户的持久数据，如数据库和云存储将用户的大量数据保存在服务器端。

2.服务器端也可以保存用户的临时会话数据。

服务器端的 session 机制，如 jsp 的 session 对象，数据保存在服务器上。实现上，服务器和浏览器之间仅需传递 session id 即可，服务器根据 session id 找到对应用户的 session 对象。会话数据仅在一段时间内有效，这个时间就是 server 端设置的 session 有效期。

服务器端保存所有的用户的数据，所以服务器端的开销较大，而浏览器端保存则把不同用户需要的数据分布保存在用户各自的浏览器中。

浏览器端一般只用来存储小数据，而服务器可以存储大数据或小数据。

服务器存储数据安全一些，浏览器只适合存储一般数据。

5.浏览器发送 cookie 时会发送哪几个部分？

1	HTTP/1.1 200 OK
2	Content-type: text/html
3	Set-Cookie: name=value; expires=失效时间; domain=域名;secure

Cookie 由变量名和值组成，其属性中既有标准的 Cookie 变量，也有用户自己创建的变量,属性中变量是用"变量=值"形式来保存

Set - Cookie: NAME=VALUE ;

Expires=DATE[有效终止日期]；

Path=PATH[Path 属性定义了 Web 服务器上哪些路径下的页面可获取服务器设置的 Cookie]；

Domain=DOMAIN_NAME；

SECURE[在 Cookie 中标记该变量，表明只有当浏览器和 Web Server 之间的通信协议为加密认证协议时，浏览器才向服务器提交相应的 Cookie。当前这种协议只有一种，即为 HTTPS]

6.cookies, sessionStorage 和 localStorage 的区别？

6.0 定义

Cookie 是 cookie 是网站为了标示用户身份而储存在用户本地终端（Client Side）上的数据（通常经过加密）。cookie 数据始终在同源的 http 请求中携带（即使不需要），会在浏览器和服务端间来回传递。也可以在跨域请求带上 Cookie。

sessionStorage 和 localStorage 是 HTML5 Web Storage API 提供的，这两种方式都允许开发者使用 js 设置的键值对进行操作，在重新加载不同的页面的时候读出它们。这一点与 cookie 类似。可以方便的在 web 请求之间保存数据。有了本地数据，就可以避免数据在浏览器和服务端间不必要地来回传递。

6.1 本地和服务端传递

Cookie 会在浏览器和服务端间来回传递

sessionStorage 和 localStorage 不会自动把数据发给服务器，仅在本地保存。

6.2 存储大小：

cookie 数据大小不能超过 4k。

sessionStorage 和 localStorage 虽然也有存储大小的限制，但比 cookie 大得多，可以达到 5M 或更大。

6.3 有效期时间：

localStorage 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据；

sessionStorage 数据在当前浏览器窗口关闭后自动删除。

cookie 设置的 cookie 过期时间之前一直有效，即使窗口或浏览器关闭

6.4 作用域不同:

sessionStorage 不在不同的浏览器窗口中共享，即使是同一个页面；

localStorage 在所有同源窗口中都是共享的；

cookie 也是在所有同源窗口中都是共享的。

6.5 可操作性

Web Storage(localStorage、sessionStorage) 支持事件通知机制，可以将数据更新的通知发送给监听者。

Web Storage 的 api 接口使用更方便。

6.5 联系

sessionStorage、localStorage、cookie 都是在浏览器端存储的数据，其中 sessionStorage 的概念很特别，引入了一个“浏览器窗口”的概念。

6.6 Web Storage 带来的好处：

使用 local storage 和 session storage 主要通过 js 中操作这两个对象来实现，分别为 window.localStorage 和 window.sessionStorage。这两个对象均是 Storage 类的两个实例，自然也具有 Storage 类的属性和方法。

减少网络流量：一旦数据保存在本地后，就可以避免再向服务器请求数据，因此减少不必要的数据请求，减少数据在浏览器和服务端间不必要地来回传递。

快速显示数据：性能好，从本地读数据比通过网络从服务器获得数据快得多，本地数据可以即时获得。再加上网页本身也可以有缓存，因此整个页面和数据都在本地的话，可以立即显示。

临时存储：很多时候数据只需要在用户浏览一组页面期间使用，关闭窗口后数据就可以丢弃了，这种情况使用 sessionStorage 非常方便。

7. 浏览器在一次 HTTP 请求中，需要传输一个 4097 字节的文本数据给服务端，可以采用那些方式？

IndexedDB、Cookie、url、Session、**post**、local Storage

IndexedDB 是 HTML5 的本地存储，把一些数据存储在浏览器（客户端）中，当与网络断开时，可以从浏览器中读取数据，用来做一些离线应用。

Cookie 通过在客户端（浏览器）记录信息确定用户身份，最大为 4 kb。

url 参数用的是 get 方法，从服务器上获取数据，大小不能大于 2 kb。

Session 是服务器端使用的一种记录客户端状态的机制。

post 是向服务器传送数据，数据量较大。

local Storage 也是 HTML5 的本地存储，将数据保存在客户端中（一般是永久的）。

IE 支持 userData 本地存储数据，但是基本很少使用到，除非有很强的浏览器兼容需求。

13. 缓存

7. 强缓存(本地缓存)和协商缓存(服务器)的命中和管理

答:

1) 浏览器在加载资源时，先根据这个资源的一些 http header 判断它是否命中强缓存，强缓存如果命中，浏览器直接从自己的缓存中读取资源，不会发请求到服务器。比如某个 css 文件，如果浏览器在加载它所在的网页时，这个 css 文件的缓存配置命中了强缓存，浏览器就直接从缓存中加载这个 css，连请求都不会发送到网页所在服务器；

2) 当强缓存没有命中的时候，浏览器一定会发送一个请求到服务器，通过服务器端依据资源的另外一些 http header 验证这个资源是否命中协商缓存，如果协商缓存命中，服务器会将这个请求返回，但是不会返回这个资源的数据，而是告诉客户端可以直接从缓存中加载(下载)这个资源，于是浏览器就又会从自己的缓存中去加载这个资源；

3) 强缓存与协商缓存的共同点是：如果命中，都是从客户端缓存中加载资源，而不是从服务器加载资源数据；区别是：强缓存不发请求到服务器，协商缓存会发请求到服务器。

4) 当协商缓存也没有命中的时候，浏览器直接从服务器加载资源数据。

关于强缓存:

当浏览器对某个资源的请求命中了强缓存时，返回的 http 状态为 200，在 chrome 的开发者工具的 network 里面 size 会显示为 from cache

强缓存是利用 Expires 或者 Cache-Control 这两个 http response header 实现的，它们都用来表示资源在客户端缓存的有效期。

Expires: 是 http1.0 提出的一个表示资源过期时间的 header，它描述的是一个绝对时间，由服务器返回，用 GMT 格式的字符串表示，如：Expires:Thu, 31 Dec 2037 23:55:55 GMT，它的缓存原理是：

浏览器第一次跟服务器请求一个资源，服务器在返回这个资源的同时，在 response 的 header 加上 Expires 的 header 浏览器在接收到这个资源后，会把这个资源连同所有 response header 一起缓存下来（所以缓存命中的请求返回的 header 并不是来自服务器，而是来自之前缓存的 header）；浏览器再请求这个资源时，先从缓存中寻找，找到这个资源后，拿出它的 Expires 跟当前的请求时间比较，如果请求时间在 Expires 指定的时间之前，就能命中缓存，否则就不行。如果缓存没有命中，浏览器直接从服务器加载资源时，Expires Header 在重新加载的时候会被更新。

Cache-Control: Expires 是较老的强缓存管理 header，由于它是服务器返回的一个绝对时间，在服务器时间与客户端时间相差较大时，缓存管理容易出现問題，比如随意修改下客户端时间，就能影响缓存命中的结果。

所以在 http1.1 的时候，提出了一个新的 header，就是 Cache-Control，这是一个相对时间，在配置缓存的时候，以秒为单位，用数值表示，如：Cache-Control:max-age=315360000，它的缓存原理是：

浏览器第一次跟服务器请求一个资源，服务器在返回这个资源的同时，在 response 的 header 加上 Cache-Control 的 header

浏览器在接收到这个资源后，会把这个资源连同所有 response header 一起缓存下来；浏览器再请求这个资源时，先从缓存中寻找，找到这个资源后，根据它第一次的请求时间和 Cache-Control 设定的有效期，计算出一个资源过期时间，再拿这个过期时间跟当前的请求时间比较，如果请求时间在过期时间之前，就能命中缓存，否则就不行。如果缓存没有命中，浏览器直接从服务器加载资源时，Cache-Control Header 在重新加载的时候会被更新。

Cache-Control 描述的是一个相对时间，在进行缓存命中的时候，都是利用客户端时间进行判断，所以相比较 Expires，Cache-Control 的缓存管理更有效，安全一些。

这两个 header 可以只启用一个，也可以同时启用，当 response header 中，Expires 和 Cache-Control 同时存在时，Cache-Control 优先级高于 Expires

关于协商缓存

当浏览器对某个资源的请求没有命中强缓存，就会发一个请求到服务器，验证协商缓存是否命中，如果协商缓存命中，请求响应返回的 http 状态为 304 并且会显示一个 Not Modified 的字符串

协商缓存是利用的是【Last-Modified, If-Modified-Since】和【ETag, If-None-Match】这两对 Header 来管理的。

【Last-Modified, If-Modified-Since】的控制缓存的原理是：

浏览器第一次跟服务器请求一个资源，服务器在返回这个资源的同时，在 response 的 header 加上 Last-Modified 的 header，这个 header 表示这个资源在服务器上的最后修改时间

浏览器再次跟服务器请求这个资源时，在 request 的 header 上加上 If-Modified-Since 的 header，这个 header 的值就是上一次请求时返回的 Last-Modified 的值：

服务器再次收到资源请求时，根据浏览器传过来 If-Modified-Since 和资源在服务器上的最后修改时间判断资源是否有变化，如果没有变化则返回 304 Not Modified，但是不会返回资源内容；如果有变化，就正常返回资源内容。当服务器返回 304 Not Modified 的响应时，response header 中不会再添加 Last-Modified 的 header，因为既然资源没有变化，那么 Last-Modified 也就不会改变，这是服务器返回 304 时的 response header：

浏览器收到 304 的响应后，就会从缓存中加载资源。

如果协商缓存没有命中，浏览器直接从服务器加载资源时，Last-Modified Header 在重新加载的时候会被更新，下次请求时，If-Modified-Since 会启用上次返回的 Last-Modified 值。

【ETag, If-None-Match】：【Last-Modified, If-Modified-Since】都是根据服务器时间返回的 header，一般来说，在没有调整服务器时间和篡改客户端缓存的情况下，这两个 header 配合起来管理协商缓存是非常可靠的，但是有时候也会服务器上资源其实有变化，但是最后修改时间却没有变化的情况，而这种问题又很不容易被定位出来，而当这种情况出现的时候，就会影响协商缓存的可靠性。所以就有了另外一对 header 来管理协商缓存，这对 header 就是【ETag, If-None-Match】。它们的缓存管理的方式是：

浏览器第一次跟服务器请求一个资源，服务器在返回这个资源的同时，在 response 的 header 加上 ETag 的 header，这个 header 是服务器根据当前请求的资源生成的一个唯一标识，这个唯一标识是一个字符串，只要资源有变化这个串就不同，跟最后修改时间没有关系，所以能很好的补充 Last-Modified 的问题：

浏览器再次跟服务器请求这个资源时，在 request 的 header 上加上 If-None-Match 的 header，这个 header 的值就是上一次请求时返回的 ETag 的值：

服务器再次收到资源请求时，根据浏览器传过来 If-None-Match 和然后再根据资源生成一个新的 ETag，如果这两个值相同就说明资源没有变化，否则就是有变化；如果没有变化则返回 304 Not Modified，但是不会返回资源内容；如果有变化，就正常返回资源内容。与 Last-Modified 不一样的是，当服务器返回 304 Not Modified 的响应时，由于 ETag 重新生成过，response header 中还会把这个 ETag 返回，即使这个 ETag 跟之前的没有变化

浏览器收到 304 的响应后，就会从缓存中加载资源。

浏览器行为对缓存的影响

如果资源已经被浏览器缓存下来，在缓存失效之前，再次请求时，默认会先检查是否命中强缓存，如果强缓存命中则直接读取缓存，如果强缓存没有命中则发请求到服务器检查是否命中协商缓存，如果协商缓存命中，则告诉浏览器还是可以从缓存读取，否则才从服务器返回最新的资源。这是默认的处理方式

以下行为可能改变缓存的默认处理方式

当 ctrl+f5 强制刷新网页时，直接从服务器加载，跳过强缓存和协商缓存；

当 f5 刷新网页时，跳过强缓存，但是会检查协商缓存；

参考：[浏览器缓存知识小结及应用](#) by 流云诸葛

什么是 Etag？

当发送一个服务器请求时，浏览器首先会进行缓存过期判断。浏览器根据缓存过期时间判断缓存文件是否过期。

情景一：若没有过期，则不向服务器发送请求，直接使用缓存中的结果，此时我们在浏览器控制台中可以看到 200 OK(from cache)，此时的情况就是完全使用缓存，浏览器和服务端没有任何交互的。

情景二：若已过期，则向服务器发送请求，此时请求中会带上①中设置的文件修改时间，和 Etag

然后，进行资源更新判断。服务器根据浏览器传过来的文件修改时间，判断自浏览器上一次请求之后，文件是不是没有被修改过；根据 Etag，判断文件内容自上一次请求之后，有没有发生变化

情形一：若两种判断的结论都是文件没有被修改过，则服务器就不给浏览器发 index.html 的内容了，直接告诉它，文件没有被修改过，你用你那边的缓存吧—— 304 Not Modified，此时浏览器就会从本地缓存中获取 index.html 的内容。此时的情况叫协议缓存，浏览器和服务端之间有一次请求交互。

情形二：若修改时间和文件内容判断有任意一个没有通过，则服务器会受理此次请求，之后的操作同①

① 只有 get 请求会被缓存，post 请求不会

Expires 和 Cache-Control

Expires 要求客户端和服务端的时钟严格同步。HTTP1.1 引入 Cache-Control 来克服 Expires 头的限制。如果 max-age 和 Expires 同时出现，则 max-age 有更高的优先级。

Cache-Control: no-cache, private, max-age=0 ETag: abcde Expires: Thu, 15 Apr 2014 20:00:00 GMT

Pragma: private Last-Modified: \$now // RFC1123 format

ETag 应用：

ETag 由服务器端生成，客户端通过 If-Match 或者说 If-None-Match 这个条件判断请求来验证资源是否修改。常见的是使用 If-None-Match。请求一个文件的流程可能如下：

====第一次请求====

1. 客户端发起 HTTP GET 请求一个文件； 2. 服务器处理请求，返回文件内容和一堆 Header，当然包括 Etag (例如 "2e681a-6-5d044840") (假设服务器支持 Etag 生成和已经开启了 Etag)。状态码 200

====第二次请求====

客户端发起 HTTP GET 请求一个文件，注意这个时候客户端同时发送一个 If-None-Match 头，这个头的内容就是第一次请求时服务器返回的 Etag: 2e681a-6-5d044840。服务器判断发送过来的 Etag 和计算出来的 Etag 匹配，因此 If-None-Match 为 False，不返回 200，返回 304，客户端继续使用本地缓存；流程很简单，问题是，如果服务器又设置了 Cache-Control: max-age 和 Expires 呢，怎么办

答案是同时使用，也就是说在完全匹配 If-Modified-Since 和 If-None-Match 即检查完修改时间和 Etag 之后，

服务器才能返回 304. (不要陷入到底使用谁的问题怪圈)

为什么使用 Etag 请求头?

Etag 主要为了解决 Last-Modified 无法解决的一些问题。

14.git 操作

git add 用于修改或者新增文件, 如果只是想要新建一个文件 README.md, 则 git add README.md; 如果是修改了多个文件又不想一个个输入, 就直接 git add .

git checkout xxx.js 如果改错了(只是已经修改了文件并 add 了), 想要恢复这个文件

git checkout master checkout 已有的分支, 就是之前是在 dev 上修改, 现在换到 master 上才能进行 merge

git commit -m "xxx" 将修改提交到本地仓库, 后面是这次修改的注释

git push origin master 将本地仓库提交到服务器(源)上的主分支上, 如果将 master 改为 dev(某一个分支名)则提交到 dev 分支上

git pull origin master 从服务器(源)拉取最新版本的代码

git checkout -b dev 创建 dev 分支, -b 代表 branch, 这个命令不需要 commit+push 的, 分支的意义在于让个人进行修改, 而不会影响到主分支的上限; 区别于 git checkout xxx

git branch 查看分支

git merge dev 将 dev 分支 merge 到主分支, 个人修改完成后就合并到主分支上; 这之前必须把服务器上的最新代码先下载到本地

git clone 在 github 上获得地址后, 可以 clone 到本地

git status 查看现在本地的修改状态, 如果有一个修改就会显示出来

git diff 新增的文件有什么具体的修改

15.模块化

15.1. 不使用模块化

- util.js getFormatDate函数
- a-util.js aGetFormatDate函数 使用getFormatDate
- a.js aGetFormatDate

util.js 中是底层的基础函数库,

a-util.js 中是针对 a 业务的基础函数库

a.js 中是业务代码, 层层往上依赖

举例 js 代码:

```
1 // util.js
2 function getFormatDate(date, type) {
3   // type === 1 返回 2017-06-15
4   // type === 2 返回 2017年6月15日 格式
5   // ....
6 }
7
8 // a-util.js
9 function aGetFormatDate(date) {
10   // 要求返回 2017年6月15日 格式
11   return getFormatDate(date, 2)
12 }
13
14 // a.js
15 var dt = new Date()
16 console.log(aGetFormatDate(dt))
```

使用:

```

1 <script src="util.js"></script>
2 <script src="a-util.js"></script>
3 <script src="a.js"></script>
4
5 <!-- 1. 这些代码中的函数必须是全局变量，才能暴露给使用方。全局变量污染 -->
6 <!-- 2. a.js 知道要引用 a-util.js，但是他知道还需要依赖于 util.js吗? -->

```

缺点：

1. 三个 script 标签的顺序一定不能乱，他们是强依赖关系
2. 如上，因为不同 js 之间的数据传输只能通过全局变量；全局变量污染是指你的代码中定义的全局变量可能把别人刚刚定义的全局变量给覆盖了
3. 如上

15. 2. 使用模块化——一种希望的用法

```

1 // util.js
2 export {
3   getFormatDate: function (date, type) {
4     // type === 1 返回 2017-06-15
5     // type === 2 返回 2017年6月15日 格式
6   }
7 }
8
9 // a-util.js
10 var getFormatDate = require('util.js')
11 export {
12   aGetFormatDate: function (date) {
13     // 要求返回 2017年6月15日 格式
14     return getFormatDate(date, 2)
15   }
16 }

```

```

1 // a.js
2 var aGetFormatDate = require('a-util.js')
3 var dt = new Date()
4 console.log(aGetFormatDate(dt))
5
6 // 直接`<script src="a.js"></script>`，其他的根据依赖关系自动引用
7 // 那两个函数，没必要做成全局变量，不会带来污染和覆盖

```

3. AMD 异步模块定义规范

- require.js requirejs.org/
- 全局 define 函数
- 全局 require 函数
- 依赖JS会自动、异步加载

AMD 的一个很好用的工具是 require.js

引用 require.js 之后会定义全局函数 define 和 require

依赖的 js 会自动异步加载就是像上面的例子里讲的那样，只用引用最表层的 js，后面的依赖关系可以自己加载使用 require.js

```

1 // util.js
2 define(function () {
3     return {
4         getFormatDate: function (date, type) {
5             if (type === 1) {
6                 return '2017-06-15'
7             }
8             if (type === 2) {
9                 return '2017年6月15日'
10            }
11        }
12    }
13 })
14
15 // a-util.js
16 define(['./util.js'], function (util) {
17     return {
18         aGetFormatDate: function (date) {
19             return util.getFormatDate(date, 2)
20         }
21     }
22 })

```

```

1 // a.js
2 define(['./a-util.js'], function (aUtil) {
3     return {
4         printDate: function (date) {
5             console.log(aUtil.aGetFormatDate(date))
6         }
7     }
8 })
9
10 // main.js
11 require(['./a.js'], function (a) {
12     var date = new Date()
13     a.printDate(date)
14 })

```

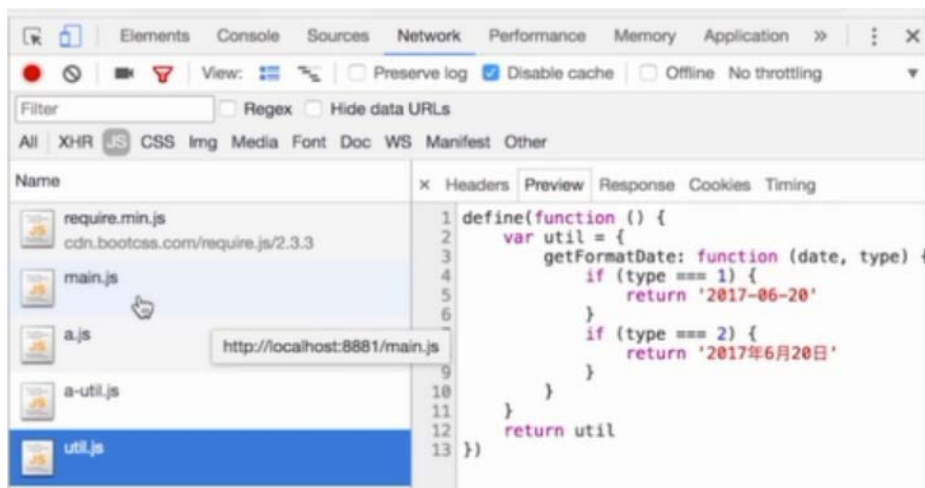
main.js 是一个入口:

```

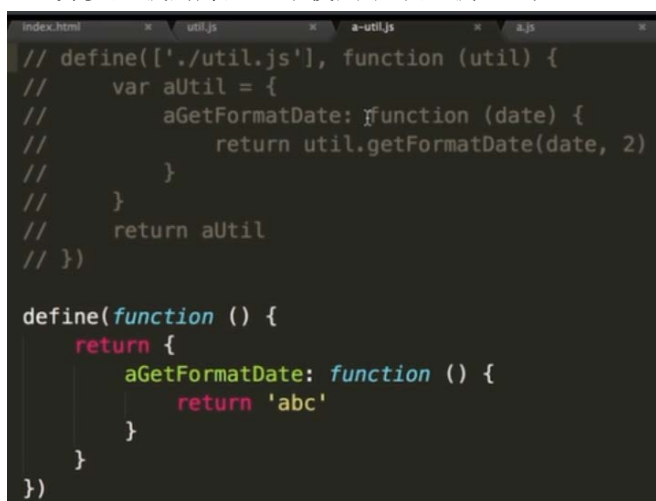
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8     <p>AMD test</p>
9
10    <script src="/require.min.js" data-main="./main.js"></script>
11 </body>
12 </html>

```

require.js 在加载的时候会检查 **data-main** 属性:你可以在 **data-main** 指向的脚本中设置模板加载选项,然后加载第一个应用模块。



在 chrome 中可以看出是自动加载依赖文件
AMD 异步加载的好处：不使用就不加载，比如



就不会加载 util.js 了，这样能提高页面渲染的效率

4. CommonJS

- nodejs 模块化规范，现在被大量用前端，原因：
- 前端开发依赖的插件和库，都可以从 npm 中获取
- 构建工具的高度自动化，使得使用 npm 的成本非常低
- CommonJS 不会异步加载JS，而是同步一次性加载出来

CommonJS 和 AMD 的区别：同步加载

CommonJS 从哪里来：nodejs

为什么在前端中用的这么多：npm 使用简单，npm 是集成在 nodejs 中的，但是和 CommonJS 有啥关系使用：


```

1 // util.js
2 module.exports = {
3   getFormatDate: function (date, type) {
4     if (type === 1) {
5       return '2017-06-15'
6     }
7     if (type === 2) {
8       return '2017年6月15日'
9     }
10  }
11 }
12
13 // a-util.js
14 var util = require('util.js')
15 module.exports = {
16   aGetFormatDate: function (date) {
17     return util.getFormatDate(date, 2)
18   }
19 }

```

符合之前的预期，简洁，符合大众的理解

- 代码演示下一节介绍
- 需要构建工具支持
- 一般和 npm 一起使用

说说你对前端模块化\AMD 和 Commonjs 的理解

CommonJS 是服务器端模块的规范，Node.js 采用了这个规范。CommonJS 规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。AMD 规范则是非同步加载模块，允许指定回调函数。

AMD 推荐的风格通过返回一个对象做为模块对象，CommonJS 的风格通过对 module.exports 或 exports 的属性赋值来达到暴露模块对象的目的。

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。

CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

AMD 是提前执行，CMD 是延迟执行。

CMD 模块方式:define(function(require, exports, module) { // 模块代码 });

AMD 和 CommonJS 的使用场景

- 需要异步加载JS，使用AMD
- 使用了 npm 之后建议使用 CommonJS

关于 javascript 模块化，下列描述错误的是

正确答案: C 你的答案: D (错误)

- A.模块化有利于管理模块间的依赖，更依赖模块的维护
- B.主流的模块化包括 CommonJS,AMD,CMD 等
- C.Seajs 遵循 AMD 规范，RequireJS 遵循 CMD 规范
- D.AMD 推崇依赖前置，CMD 推崇依赖就近

[AMD](#) 是 "Asynchronous Module Definition" 的缩写，意思就是 "异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。

AMD 也采用 require() 语句加载模块，但是不同于 CommonJS。

主要有两个 Javascript 库实现了 AMD 规范：[require.js](#) 和 [curl.js](#)。

参考链接：http://www.ruanyifeng.com/blog/2012/10/asynchronous_module_definition.html

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。

CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

区别：

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。
2. CMD 推崇依赖就近，AMD 推崇依赖前置。

// CMD

```
define(function(require, exports, module) {  
  var a = require('./a')  
  a.doSomething()  
  // 此处略去 100 行  
  var b = require('./b') // 依赖可以就近书写  
  b.doSomething()  
  // ...  
})
```

// AMD 默认推荐的是

```
define(['./a', './b'], function(a, b) { // 依赖必须一开始就写好  
  a.doSomething()  
  // 此处略去 100 行  
  b.doSomething()  
  ...  
})
```

16. 页面加载过程、页面渲染过程

16.1 加载形式：

- 输入 url（或跳转页面）加载 html
- <http://coding.mimooc.com>
- 加载 html 中的静态资源
- `<script src="/static/js/jquery.js"></script>`

16.2 加载 html 或者 html 中的资源(js、css、图片、视频文件等)的过程

1. 用户访问网页，DNS 服务器(域名解析系统)会根据用户提供的域名查找对应的 IP 地址，找到后，系统会向对应 IP 地址的网络服务器发送一个 http 请求

比如：`http://www.baidu.com` 只是一个域名，但是浏览器是不认识它的，所以需要 DNS 解析成 IP 地址才行，这个 IP 地址是百度的服务器的唯一 IP 地址。

最好的体会这个流程的过程就是去买一个域名，买一个服务器，把域名绑定到服务器上，备案

2. 网络服务器解析请求，并发送数据给数据库服务器
3. 数据库服务器将请求的资源返回给网络服务器，网络服务器解析数据，并生成 html 文件，放入 http response 中，返回给服务器。（服务器收到、处理并返回 http 请求）
4. 浏览器得到返回内容

浏览器解析 http response；浏览器解析 http response 后，需要下载 html 文件，以及 html 文件内包含的外部引用文件，及文件内涉及的图片或者多媒体文件

16.3 浏览器渲染页面过程

1. 根据 HTML 结构生成 DOM Tree
2. 如果加载过程中遇到外部 css 文件，浏览器会发出一个请求，来获取 css 文件。这时同步阻塞，原因是可能会有：`var width = $('#id').width()`。这意味着，在 js 代码执行前，浏览器必须保证 css 文件已下载和解析完

成。这也是 css 阻塞后续 js 的根本原因。当 js 文件不需要依赖 css 文件时，可以将 js 文件放在头部 css 的前面。

样式表在下载完成后将和以前下载的所有样式表一起进行解析(根据 CSS 生成 CSSOM, object model, 就是把 css 进行结构化处理)，解析完成后，将对此前所有元素(含以前已经渲染的)重新进行渲染

3. 将 DOM 和 CSSOM 整合形成 RenderTree, 渲染树，就是拥有样式设置的结构树

a 渲染树和 DOM 树有区别，DOM 树完全与 html 标签一一对应，但是渲染树会忽略掉不需要渲染的元素，比如 head, display: none 的元素等

b 一大段文本中的每一行在渲染树中都是一个独立的节点

c 渲染树的每一个节点都存储有对应的 css 属性

4. 根据渲染树开始渲染和展示

5. 遇到<script>时，会执行<script>，并阻塞渲染

为什么 js 可以阻塞渲染，因为 js 可以改变 DOM 结构；所以一般将外部引用的 js 文件放在</body>前

6. 遇到图片资源，浏览器会发出请求获取图片资源。这是异步请求，并不会影响 html 文档进行加载，

流利说法：

1. 用户输入网址（假设是个 html 页面，并且是第一次访问），浏览器向服务器发出请求，服务器返回 html 文件；

2. 浏览器开始载入 html 代码，发现<head>标签内有一个<link>标签引用外部 CSS 文件；

3. 浏览器又发出 CSS 文件的请求，服务器返回这个 CSS 文件；

4. 浏览器继续载入 html 中<body>部分的代码，并且 CSS 文件已经拿到手了，可以开始渲染页面了；

5. 浏览器在代码中发现一个标签引用了一张图片，向服务器发出请求。此时浏览器不会等到图片下载完，而是继续渲染后面的代码；

6. 服务器返回图片文件，由于图片占用了一定面积，影响了后面段落的排布，因此浏览器需要回过头来重新渲染这部分代码；

7. 浏览器发现了一个包含一行 Javascript 代码的<script>标签，赶快运行它；

8. Javascript 脚本执行了这条语句，它命令浏览器隐藏掉代码中的某个<div>（style.display="none"）。杯具啊，突然就少了这么一个元素，浏览器不得不重新渲染这部分代码；

9. 终于等到了</html>的到来，浏览器泪流满面.....

10. 等等，还没完，用户点了一下界面中的“换肤”按钮，Javascript 让浏览器换了一下<link>标签的 CSS 路径；

11. 浏览器召集了在座的各位<div>们，“大伙儿收拾收拾行李，咱得重新来过.....”，浏览器向服务器请求了新的 CSS 文件，重新渲染页面。

repaint/reflow(浏览器加载文件)

文件加载顺序

浏览器加载页面上引用的 CSS、JS 文件、图片时，是按顺序从上到下加载的，每个浏览器可以同时下载文件的个数不同，因此经常有网站将静态文件放在不同的域名下，这样可以加快网站打开的速度。

reflow

在加载 JS 文件时，浏览器会阻止页面的渲染，因此将 js 放在页面底部比较好。

因为如果 JS 文件中有修改 DOM 的地方，浏览器会倒回去把已经渲染过的元素重新渲染一遍，这个回退的过程叫 reflow。

CSS 文件虽然不影响 js 文件的加载，但是却影响 js 文件的执行，即使 js 文件内只有一行代码，也会造成阻塞。因为可能会有 var width = \$('#id').width()这样的语句，这意味着，js 代码执行前，浏览器必须保证 css 文件已下载和解析完成。

办法：当 js 文件不需要依赖 css 文件时，可以将 js 文件放在头部 css 的前面。

常见的能引起 reflow 的行为：

1. 改变窗口大小

2. 改变文字大小

- 3.添加/删除样式表
- 4.脚本操作 DOM
- 5.设置 style 属性
- 等等……

reflow 是不可避免的，只能尽量减小，常用的方法有：

- 1.尽量不用行内样式 style 属性，操作元素样式的时候用添加去掉 class 类的方式
- 2.给元素加动画的时候，可以把该元素的定位设置成 absolute 或者 fixed，这样不会影响其他元素 repaint

另外，有个和 reflow 相似的术语，叫做 repaint（重绘），在元素改变样式的时候触发，这个比 reflow 造成的影响要小，所以能触发 repaint 解决的时候就不要触发 reflow……

16.4 不同资源加载过程示例

示例二

```
1  div {  
2    width: 100%;  
3    height: 100px;  
4    font-size: 50px;  
5  }
```

```
1  <!DOCTYPE html>  
2  <html>  
3  <head>  
4    <meta charset="UTF-8">  
5    <title>Document</title>  
6    <link rel="stylesheet" type="text/css" href="test.css">  
7  </head>  
8  <body>  
9    <div>test</div>  
10 </body>  
11 </html>
```

加载 HTML 代码，然后一步一步分析生成 DOM 树，分析到 head 里面发现有一个 link 要求下载一个 css 文件，加载了以后就开始生成 CSSOM，然后浏览器就知道遇到 div 就要怎么怎么设置样式，然后继续解析到 div 的时候就可以一次将 div 渲染出来

思考：为什么 css 放在 head 标签中

加载完 css 后浏览器直接知道规则，在首次渲染 div 的时候就可以把样式考虑进去一起渲染；否则如果把 link 那句话放到 div 后面，首次渲染 div 时就会先按照默认情况渲染，后来读到 link 之后再重新渲染，这样带来的问题就是：性能差、用户体验差

示例三

```
1  document.getElementById('container').innerHTML = 'update by js'
```

```
1  <!DOCTYPE html>  
2  <html>  
3  <head>  
4    <meta charset="UTF-8">  
5    <title>Document</title>  
6  </head>  
7  <body>  
8    <div id="container">default</div>  
9    <script src="index.js"></script>  
10 <p>test</p>  
11 </body>  
12 </html>
```

首先输入 url 跳转页面，将 html 代码加载出来，逐行解析成 DOM 树，8 行渲染出一个 div，内容是 default，9 行发现有个 script，然后就加载并执行，这时是阻塞页面渲染的，10 行一直没被渲染，一直等着，直到 js 执行完，继续渲染

思考：为什么要把 js 放在 body 最下面

首先不会阻塞正常页面的渲染；script 放在最下面的话，它能拿到所有标签；这是一个性能优化的问题

示例四

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <p>test</p>
9   <p></p>
10  <p>test</p>
11 </body>
12 </html>
```

- 定时任务：setTimeout，setInterval
- 网络请求：ajax 请求，动态 加载
- 事件绑定

遇到异步加载的情况，img 的加载不会阻塞渲染

16.5 window.onload 和 DOMContentLoaded

```
1 window.addEventListener('load', function () {
2   // 页面的全部资源加载完才会执行，包括图片、视频等
3 })
4 document.addEventListener('DOMContentLoaded', function () {
5   // DOM 渲染完即可执行，此时图片、视频还可能没有加载完
6 })
```

这是一些函数的定义，为了更好的性能，要尽量使用第二种情况

17. 性能和性能优化

白屏时间 first paint 和可交互时间 dom ready 的关系是？A

- A 先触发 first paint，后触发 dom ready
- B 先触发 dom ready，后触发 first paint
- C 一起触发
- D 没关系

白屏时间（first Paint Time）——用户从打开页面开始到页面开始有东西呈现为止

首屏时间——用户浏览器首屏内所有内容都呈现出来所花费的时间

用户可操作时间(dom Interactive)——用户可以进行正常的点击、输入等操作，默认可以统计 domready 时间，因为通常会在这时候绑定事件操作

总下载时间——页面所有资源都加载完成并呈现出来所花的时间，即页面 onload 的时间

总加载时间 = 白屏时间(first paint) + 首屏时间 + 核心功能可用时间 + 页面可交互(DOM Ready).

17.1 原则:

1. 多使用内存、缓存或者其他存储
2. 减少 CPU 计算、减少网络(前端没有对硬盘的操作, 否则还应该有减少 IO 操作)

入手: 加载页面和静态资源、页面渲染

17.2 加载资源优化

1. 静态资源的压缩、合并

压缩就是可以使用 webpack 进行压缩; 合并就是如果要加载三个 js, 如果单个加载, 就是三个请求, 会比较慢, 如果能合并就只要一个请求

2. 静态资源缓存

比如要是用 jq, 只要 jq 版本没有变化, 每次都会从缓存中使用 jq; 要更新就换名字, 重新加载

3. 使用 CDN 让资源加载更快

很多大网站都有自己的 CDN(Content Delivery Network, 即内容分发网络), 可以去网上搜索比如 jq 的 CDN, 它之所以快是因为如果在北京要访问一个地址, CDN 可以将地址转到北京的一个地址

4. 使用 SSR(服务端渲染)后端渲染, 数据直接输出到 HTML

请说出三种减少页面加载时间的方法。(加载时间指感知的时间或者实际加载时间)

1. 优化图片

2. 图像格式的选择 (GIF: 提供的颜色较少, 可用在一些对颜色要求不高的地方)

3. 优化 CSS (压缩合并 css, 如 margin-top, margin-left...) 网址后加斜杠 (如 www.campr.com/ 目录, 会判断这个“目录是什么文件类型, 或者是目录。)

4. 标明高度和宽度 (如果浏览器没有找到这两个参数, 它需要一边下载图片一边计算大小, 如果图片很多, 浏览器需要不断地调整页面。这不但影响速度, 也影响浏览体验。当浏览器知道了高度和宽度参数后, 即使图片暂时无法显示, 页面上也会腾出图片的空位, 然后继续加载后面的内容。从而加载时间快了, 浏览体验也更好了。)

5. 减少 http 请求 (合并文件, 合并图片)。

17.3 页面渲染优化

- CSS 放前面, JS 放后面
- 懒加载 (图片懒加载、下拉加载更多)
- 减少 DOM 查询, 对 DOM 查询做缓存
- 减少 DOM 操作, 多个操作尽量合并在一起执行
- 事件节流
- 尽早执行操作 (如 DOMContentLoaded)

示例:

1. 资源合并

1.1 可以用 CommonJS 打包合并

1.2 暴力合并, a、b、c 可能没有什么关系, 但是也可以放在一起, 一般都通过构建工具合并, 否则人工容易出错

```
1 <script src="a.js"></script>
2 <script src="b.js"></script>
3 <script src="c.js"></script>
```

```
1 <script src="abc.js"></script>
```

2. 缓存

- 通过连接名称控制缓存
- `<script src="abc_1.js"></script>`
- 只有内容改变的时候，链接名称才会改变
- `<script src="abc_2.js"></script>`

3. cdn，比在自己的服务器上加载快得多

```
<link href="https://cdn.bootcss.com/bootstrap/4.0.0-alpha.6/css/bootstrap.css" rel="sty
<script src="https://cdn.bootcss.com/zepto/1.0rc1/zepto.min.js"></script>
```

4. 懒加载，每次都用一个被缓存的很小的图 preview.png 图片先显示着，然后指定 realsrc，再通过 js 去重新获取

```
1 
2 <script type="text/javascript">
3     var img1 = document.getElementById('img1')
4     img1.src = img1.getAttribute('data-realsrc')
5 </script>
```

5. 缓存 DOM 查询

```
1 // 未缓存 DOM 查询
2 var i
3 for (i = 0; i < document.getElementsByTagName('p').length; i++) {
4     // todo
5 }
6
7 // 缓存了 DOM 查询
8 var pList = document.getElementsByTagName('p')
9 var i
10 for (i = 0; i < pList.length; i++) {
11     // todo
12 }
```

3 行每次循环都会执行一次 dom

6. 合并 DOM 插入 createDocumentFragment 片段

```
1 var listNode = document.getElementById('list')
2
3 // 要插入 10 个 li 标签
4 var frag = document.createDocumentFragment();
5 var x, li;
6 for(x = 0; x < 10; x++) {
7     li = document.createElement("li");
8     li.innerHTML = "List item " + x;
9     frag.appendChild(li);
10 }
11
12 listNode.appendChild(frag);
```

7. 事件节流

```

1 var textarea = document.getElementById('text')
2 var timeoutId
3 textarea.addEventListener('keyup', function () {
4     if (timeoutId) {
5         clearTimeout(timeoutId)
6     }
7     timeoutId = setTimeout(function () {
8         // 触发 change 事件
9     }, 100)
10 })

```

比如输入 abcde, 就要触发 5 次 change 事件, keyup 事件可能间隔非常短

所以设置一个 timeoutId, 当 keyup 事件之后停了 100ms 的时候才触发 change 事件, 连续无间隔输入的时候就不用触发 change 事件, keyup 事件可能间隔非常短, 短时间内又有一个 keyup 的时候, 上一个定时器就被清除掉了就不会执行了

8. 尽早操作, 使用 DOMContentLoaded

```

1 window.addEventListener('load', function () {
2     // 页面的全部资源加载完才会执行, 包括图片、视频等
3 })
4 document.addEventListener('DOMContentLoaded', function () {
5     // DOM 渲染完即可执行, 此时图片、视频还可能没有加载完
6 })

```

前端开发的优化问题:

1 减少 http 请求次数: css spirit, data uri

2 JS, CSS 源码压缩

3 前端模板 JS+数据, 减少由于 HTML 标签导致的带宽浪费, 前端用变量保存 AJAX 请求结果, 每次操作本地变量, 不用请求, 减少请求次数

4 用 innerHTML 代替 DOM 操作, 减少 DOM 操作次数, 优化 javascript 性能

5 用 setTimeout 来避免页面失去响应

6 用 hash-table 来优化查找

7 当需要设置的样式很多时设置 className 而不是直接操作 style

8 少用全局变量

9 缓存 DOM 节点查找的结果

10 避免使用 CSS Expression

11 图片预载

12 避免在页面的主体布局中使用 table, table 要等其中的内容完全下载之后才会显示出来, 显示比 div+css 布局慢

13 请求带宽: 控制网页在[网络](#)传输过程中的数据量

启用 GZIP 压缩文件

保持良好的编程习惯, 避免重复的 CSS, JavaScript 代码, 多余的 HTML 标签和属性

14 代码层面: 避免使用 css 表达式, 避免使用高级选择器, 通配选择器。

15 缓存利用: 缓存 Ajax, 使用 CDN, 使用外部 js 和 css 文件以便缓存, 添加 Expires 头, 服务端配置 Etag, 减少 DNS 查找等

16 请求数量: 合并样式和脚本, 使用 css 图片精灵, 初始首屏之外的图片资源按需加载, 静态资源延迟加载。

代码层面的优化

17 避免全局查询

18 避免使用 with(with 会创建自己的作用域, 会增加作用域链长度)

19 多个变量声明合并

20 避免图片和 iFrame 等的空 Src。空 Src 会重新加载当前页面, 影响速度和效率

21 尽量避免写在 HTML 标签中写 Style 属性

补充:

```

[javascript]
01. var scope = "global";
02. function f1() {
03.     console.log(scope);
04. }
05. f1() // output: global
06. function f2() {
07.     scope = "f2"
08.     with(scope) {
09.         f1();
10.     }
11. }
12. f2(); // output: f2

```

with 会改变其范围内的作用域链。

这里 with 把在其范围内的 f1()函数的执行作用域链的父级作用域更改为了自己，所以在执行 f1()函数时，找到父级作用域链中 scope 定义为“f2”，因此加入 with 后，第二次可以输出“f2”。

性能参数

3) getTimes()

在这个方法中计算各个参数之间的值。

在网上参考了很多资料，再结合了一点自己的理解，有些参数的理解可能有误，具体的计算方式可以查看源码“[primus.js](#)”。

1. firstPaint：白屏时间，也就是开始解析 DOM 耗时，用户在没有滚动时候看到的内容渲染完成并且可以交互的时间
2. loadTime：加载总时间，这几乎代表了用户等待页面可用的时间
3. unloadEventTime：Unload 事件耗时
4. loadEventTime：执行 onload 回调函数的时间
5. domReadyTime：用户可操作时间
6. firstScreen：首屏时间，用户在没有滚动时候看到的内容渲染完成并且可以交互的时间，记录载入时间最长的图片
7. parseDomTime：解析 DOM 树结构的时间，期间要加载内嵌资源
8. initDomTreeTime：请求完毕至 DOM 加载耗时
9. readyStart：准备新页面时间耗时
10. redirectTime：重定向的时间
11. appcacheTime：DNS 缓存耗时
12. lookupDomainTime：DNS 查询耗时
13. connectTime：TCP 连接耗时
14. requestTime：内容加载完成的时间
15. requestDocumentTime：请求文档时间，开始请求文档到开始接收文档
16. responseDocumentTime：接收文档时间，开始接收文档到文档接收完成
17. TTFB (Time To First Byte)：读取页面第一个字节的时间

domready 是 DOM 树解析完成的时刻，这个时刻可以开始调用操作 dom 的事件；

firstPaint 白屏时间，是内容渲染完成并且可以交互的时间

按理说 DOM 树加载完以后还要生成渲染树，这时才能出现界面这时才能够不是白屏

18. 安全性

18.1 问题：场景的前端安全性问题有哪些

- XSS 跨站请求攻击
- XSRF 跨站请求伪造

XSS

- 再新浪博客写一篇文章，同时偷偷插入一段<script>
- 攻击代码中，获取cookie，发送自己的服务器
- 发布博客，有人查看博客内容
- 会把查看者的 cookie 发送到攻击者的服务器

预防：

- 前端替换关键字，例如替换 < 为 < > 为 >
- 后端替换

但是在前端替换会导致 script 执行效率更低

XSRF (CSRF)

- 你已登录一个购物网站，正在浏览器商品
- 该网站付费接口是 xxx.com/pay?id=100 但是没有任何验证
- 然后你收到一封邮件，隐藏着
- 你查看邮件的时候，就已经悄悄的付费购买了

解决方案：

- 增加验证流程，如输入指纹、密码、短信验证码

18.2 常见 web 安全及防护原理

18.2.1 sql 注入原理

就是通过把 SQL 命令插入到 Web 表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。

总的来说有以下几点：

1. 永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对单引号和双“-”进行转换等。
2. 永远不要使用动态拼装 SQL，可以使用参数化的 SQL 或者直接使用存储过程进行数据查询存取。
3. 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。
4. 不要把机密信息明文存放，请加密或者 hash 掉密码和敏感的信息。

18.2.2 XSS 原理及防范

Xss(cross-site scripting)攻击指的是攻击者往 Web 页面里插入恶意 html 标签或者 javascript 代码。比如：攻击者在论坛中放一个看似安全的链接，骗取用户点击后，窃取 cookie 中的用户私密信息；或者攻击者在论坛中加一个恶意表单，当用户提交表单的时候，却把信息传送到攻击者的服务器中，而不是用户原本以为的信任站点。

XSS 防范方法

首先代码里对用户输入的地方和变量都需要仔细检查长度和对“<”, “>”, “;”, “'”, “”等字符做过滤；其次任何内容写到页面之前都必须加以 encode，避免不小心把 html tag 弄出来。这一个层面做好，至少可以堵住超过一半的 XSS 攻击。

首先，避免直接在 cookie 中泄露用户隐私，例如 email、密码等等。

其次，通过使 cookie 和系统 ip 绑定来降低 cookie 泄露后的危险。这样攻击者得到的 cookie 没有实际价值，不可能拿来重放。

如果网站不需要再浏览器端对 cookie 进行操作，可以在 Set-Cookie 末尾加上 HttpOnly 来防止 javascript 代码直接获取 cookie。

尽量采用 POST 而非 GET 提交表单

18.3 XSS 与 CSRF 有什么区别吗？

XSS 是获取信息，不需要提前知道其他用户页面的代码和数据包。CSRF 是代替用户完成指定的动作，需要知道其他用户页面的代码和数据包。

要完成一次 CSRF 攻击，受害者必须依次完成两个步骤：登录受信任网站 A，并在本地生成 Cookie；在不登出 A 的情况下，访问危险网站 B。

CSRF 的防御：服务端的 CSRF 方式方法很多样，但总的思想都是一致的，就是在客户端页面增加伪随机数，通过验证码的方法。

19.IE

列举 IE 与其他浏览器不一样的特性？

IE 支持 `currentStyle`，Firefox 使用 `getComputedStyle`

IE 使用 `innerText`，Firefox 使用 `textContent`

滤镜方面：IE: `filter:alpha(opacity= num)`；Firefox: `-moz-opacity:num`

事件方面：IE: `attachEvent`；火狐是 `addEventListener`

鼠标位置：IE 是 `event.clientX`；火狐是 `event.pageX`

IE 使用 `event.srcElement`；Firefox 使用 `event.target`

IE 中消除 list 的原点仅需 `margin:0` 即可达到最终效果；Firefox 需要设置 `margin:0;padding:0` 以及 `list-style:none`

CSS 圆角：ie7 以下不支持圆角

ie 各版本和 chrome 可以并行下载多少个资源

IE6 两个并发，IE7 升级之后的 6 个并发，之后版本也是 6 个 Firefox, chrome 也是 6 个

20.

`parseInt(string, radix)` 当参数 `radix` 的值为 0，或没有设置该参数时，`parseInt()` 会根据 `string` 来判断数字的基数。

举例，如果 `string` 以 "0x" 开头，`parseInt()` 会把 `string` 的其余部分解析为十六进制的整数。如果 `string` 以 0 开头，那么 ECMAScript v3 允许 `parseInt()` 的一个实现把其后的字符解析为八进制或十六进制的数字。如果 `string` 以 1~9 的数字开头，`parseInt()` 将把它解析为十进制的整数。

```
var num = '0x12';
```

```
console.log(parseInt(num,10)) ;//0
```

```
console.log(parseInt(num)) ;//18
```

21.打包工具

Webpack

webpack 可以在开发阶段使用浏览器没实现的语法规则

17. jQuery

17.1 提供了 `val()` 方法，使用它我们可以快速地获取和设置表单的文本框、单选按钮、以及单选按钮的值。

使用 `val()` 不带参数，表示获取元素的值

使用 `val()` 给定参数，则表示把值赋给元素

17.2 `fadeIn()`、`fadeOut()` 淡入淡出

17.3 获取值(不带参数时)、设置值(设置成传入参数值)

`val()`

`text()`

`html()`

18.正则表达式

`^/d+[^/d]+`

特别字符	描述
\$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 \$ 也匹配 '\n' 或 '\r'。要匹配 \$ 字符本身，请使用 \\$。
()	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 .，请使用 \.。
[]	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{ }	标记限定符表达式的开始。要匹配 {，请使用 \{。
	指明两项之间的一个选择。要匹配 ，请使用 \ 。

19.回调陷进

Node.js 回调黑洞全解：Async、Promise 和 Generator 解决回调黑洞

<http://blog.csdn.net/derek518/article/details/50386300>

我们常常把这个问题叫做"回调黑洞"或"回调金字塔"：

```
doAsync1(function () {
  doAsync2(function () {
    doAsync3(function () {
      doAsync4(function () {
        // ...
      })
    })
  })
})
```

回调黑洞是一种主观的叫法，就像嵌套太多的代码，有时候也没什么问题。为了控制调用顺序，异步代码变得非常复杂，这就是黑洞。有个问题非常合适衡量黑洞到底有多深：如果 doAsync2 发生在 doAsync1 之前，你要忍受多少重构的痛苦？目标不单单是减少嵌套层数，而是要编写模块化（可测试）的代码，便于理解和修改。

20.定时功能的函数

setTimeout

setInterval

requestAnimationFrame 方法告诉浏览器您希望执行动画，并请求浏览器调用指定的函数在下一次重绘之前更新动画。该方法使用一个回调函数作为参数，这个回调函数会在浏览器重绘之前调用

21.localeCompare 实现中文首字母排序，然后利用这个实现汉字转拼音

<http://www.php.cn/js-tutorial-362947.html>