A

1.1 安全-前端安全问题

XSS、CSRF、sql 注入、存储 cookie 怎么设置防止 xss 攻击、localstorage 不安全、为什么用 token 做安全

首先一般前端的安全问题有 XSS 跨站请求攻击和 XSRF 跨站请求伪造；XSS 是比如某黑客在新浪博客中写了一篇文章，在文章中插入一段<script>脚本，脚本中有获取 cookie 并发送到自己服务器的攻击代码。这样当他人查看这篇博客浏览器运行页面时，也会运行这段攻击代码，查看者的 cookie 就会发送到攻击者的服务器，隐私被泄露就是一个安全问题。针对跨站请求攻击可以采用替换用户提交的信息中的关键字来预防，比如将<替换为&lt，>替换为&gt，这样在载入页面的时候就不会执行博客中黑客提交的代码了。

然后像上面举例说的之所以 cookie 被窃取会有安全隐患是因为：一般网站在保持用户登录状态的时候的是会把把登录信息如账号、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 中的登录信息来实现的。如果直接把用户名与密码都保持到 Cookie 中，这时候如果 cookie 被窃取就比较危险了。所以一般是把密码加密后保存到 Cookie 中；或者不在 cookie 中保存密码，而是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来保持登录状态；或者设置发送 Cookie 的 http 请求中的 HttpOnly 为 true，来防止 Cookie 值被页面脚本读取。这样的话黑客就不能从 cookie 中提取隐私信息。但是如果在服客户端向服务端发送 cookie 的传输途中被人拦截，那黑客不需要解密 cookie，他只要原样转发 cookie 到服务器一样可以进行伪造身份登录，所以 cookie 的传输也需要安全传输技术，比如 SSL。设置发送 Cookie 的 http 请求中的 Secure 为 true，这时只有在 https 协议下访问的时候，浏览器才会发送该 Cookie，https 协议能够保证 cookie 与 WEB 服务器之间的数据传输过程加密。

对比 cookie 的种种安全性预防措施，localStorage 就很令人担忧了。目前 localStorage 存储没有对 XSS 攻击有任何抵御机制，一旦出现 XSS 漏洞，那么存储在 localStorage 里的数据就极易被获取到；而且除了 Opera 浏览器采用 BASE64 加密外(BASE64 也是可以轻松解密的)，其他浏览器均是采用明文存储数据，所以应该永远不使用明文在浏览器存储中存放数据。 另一方面，在数据存储的时效上，localStorage 并不会像 Cookie 那样可以设置生命周期，只要用户不主动删除，localStorage 存储的数据将会永久存在，隐私的信息一旦被窃取就很麻烦了。 还有就是对于本地存储，为了方便再次加载数据，常常会把数据存储在本地，等再此加载的时候，直接从本地读取数据显示在网页上。如果数据没有经过输入输出严格过滤，那么就很容易被 XSS 攻击。localStorage 还容易遭受跨目录攻击，因为 localStroage 存储方式不会像 Cookie 存储一样可以指定域中的路径，在 localStroage 存储方式中没有域路径的概念。也就是说，如果一个域下的任意路径存在 XSS 漏洞，整个域下存储的数据在知道存储名称的情况下都可以被获取到。 比如用户 xisigr 和 xhack 各自的 blog 链接虽然属于同一个域，但却有不同的路径，一个路径为 xisigr，另一个路径为 xhack。假设 xhack 用户发现自己的路径下存在存储型 XSS 漏洞，那么就可以在自己的 blog 中加入获取数据代码，其中核心代码为 localStorage.getItem(“name”)。xhack 用户不登录 blog,他只要访问 http://h.example.com/xisigr，就能通过之前注入的窃取代码获取到 xisigr 的本地存储数据。

~~localStorage 还容易遭受 DNS 欺骗攻击，DNS 欺骗就是攻击者冒充域名服务器的，然后把查询的 IP 地址设为攻击者的 IP 地址，这样的话，用户上网就只能看到攻击者的主页，而不是用户想要取得的网站的主页了。Google 在没有使用 HTML5 本地存储前，是使用 Google Gears 方式来进行本地存储的，那个时候 Google Gears 就遭到过 DNS 欺骗攻击。Google Gears 支持离线存储，可以把 Gmail,WordPresss 这样网站数据的以 SQLite 数据库的形式存储下来，以后用户就可以对存储的网站数据进行离线读取或删除操作。如果攻击者发动 DNS 欺骗攻击，那么就可以注入本地数据库，获取数据或者留下永久的后门。这样将会造成对用户持久的危害。Google Gears 所遭受的 DNS 欺骗攻击方式在 HTML5 本地存储上也是同样有效的。~~

XSS 跨站请求攻击是黑客恶意嵌入了 script 脚本，还有一种安全问题是黑客将 sql 命令注入表单，然后向服务器发送了一些请求的查询字符串，执行了不在其权限之内的数据库命令。比如有一个人在表单中写了 SELECT * from tablename WHERE XXX="String" or 1=1 DELETE * from * WHERE "*"，由于 1=1 恒成立，所以这个操作就会清空数据库。预防的办法比如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；不要信任用户的输入，要对用户的输入进行校验，对单引号和双"-"进行转换等；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

然后还有一种前端安全问题是跨站请求伪造 XSRF(CSRF)，比如有个购物网站的付费接口是 xxx.com,而且这个接口没有什么安全措施，只要点击就可以付费。那么当用户正在浏览商品的时候，如果突然收到一封邮件，邮件里隐藏着一个 img，它的 src 地址就是这个付费接口，那么当用户打开邮件的时候，图片自动加载，就自动付费了。解决方案呢就是要增加验证流程，比如密码、指纹，或者在客户端页面增加伪随机数，使用短信、语音验证等。

另外关于 XSS 与 CSRF 的区别：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。

关于登录时的验证，如果客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，这样就会增大服务器的压力，所以可以使用 Token，它是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个 Token 便将此 Token 返回给客户端，以后客户端只需带上这个 Token 前来请求数据即可，无需再次带上用户名和密码。这样就可以减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。作为 token 的值有两种，第一种是用设备号/设备 mac 地址作为 Token。客户端在登录的时候获取设备的设备号/mac 地址，并将其作为参数传递到服务端。服务端接收到该参数后，便用一个变量来接收同时将其作为 Token 保存在数据库，并将该 Token 设置到 session 中，客户端每次请求的时候都要统一拦截，并将客户端传递的 token 和服务器端 session 中的 token 进行对比，如果相同则放行，不同则拒绝。此刻客户端和服务器端就统一了一个唯一的标识 Token，而且保证了每一个设备拥有了一个唯一的会话。该方法的缺点是客户端需要带设备号/mac 地址作为参数传递，而且服务器端还需要保存；优点是客户端不需重新登录，只要登录一次以后一直可以使用，至于超时的问题是有服务器这边来处理，如何处理？若服务器的 Token 超时后，服务器只需将客户端传递的 Token 向数据库中查询，同时并赋值给变量 Token，如此，Token 的超时又重新计时。第二种是用 session 值作为 Token，客户端只需携带用户名和密码登陆即可。客户端接收到用户名和密码后并判断，如果正确了就将本地获取 sessionID 作为 Token 返回给客户端，客户端以后只需带上请求数据即可。这种方式使用的好处是方便，不用存储数据，但是缺点就是当 session 过期后，客户端必须重新登录才能进行访问数据。

AJAX 暴露了与服务器交互的细节

1.2 Ajax、Ajax get post、ajax 前后台交互过程、ajax 如何解析 json 字符串、ajax 的参数、failure 方法什么时候执行，如果压根没成功发送到服务器端，还会执行这个方法吗

原生实现：XMLHttpRequest。完全默写出这 11 行代码

```
var xhr = new XMLHttpRequest()
xhr.open("GET", "/api", false)
xhr.onreadystatechange = function () {
    // 这里的函数异步执行，可参考之前 JS 基础中的异步模块
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            alert(xhr.responseText)
        }
    }
}
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 Async=true?

(3)设置响应 HTTP 请求状态变化函数.

(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。

Async=true 表示异步执行 onreadystatechange 函数，不阻塞下面的 send 函数的执行，True 表示脚本会在 send()方法之后继续执行，而不等待来自服务器的响应。

onreadystatechange 事件使代码复杂化了。但是这是在没有得到服务器响应的情况下，防止代码停止的最安全的方法。

通过把该参数设置为 "false"，可以省去额外的 onreadystatechange 代码。如果在请求失败时是否执行其余的代码无关紧要，那么可以使用这个参数。

readyState 状态码：

- 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

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

对比 get 请求，post 请求写法上的不同在于：
多了一个数据的定义，使用 json 格式传输

```
var data =encodeFormData({
        tel : tel,
        pwd : pwd
}) ;
```

多了一个设置请求头的 MIME 类型的命令

```
xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded')
```

表单数据编码格式有一个正式的 MIME 类型：application/x-www-form-urlencoded
当使用 post 方式提交这种顺序表单时，必须设置 Content-Type 请求头为这个值来模仿表单数据的提交。

也可以看出 get 一般用于信息获取而 post 一般用于修改服务器上的资源；它们的区别还有：post 对所发送的信息长度没有限制，而 get 对所发送信息的数量有限制，一般在 2000 个字符；Get 是通过地址栏来传值，而 Post 是通过提交表单来传值。然后在无法使用缓存文件（更新服务器上的文件或数据库）、需要向服务器发送大量数据以、发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠，所以需要使用 POST 请求。

ajax 的作用是进行前后端交互，举个我做过的 esalon 项目中在的新建个人信息的前后端交互过程如下：
1.安装 phpStudy 安装包，就装好了 php+mysql+apache，然后将项目放在安装目录下的 www 文件夹
就可以通过 http://127.0.0.1/esalon/index.html 以开启本地服务的形式打开网页
2.安装 navicat for mysql 可以简单操作数据库
建了一个 esalon 数据库，比如我现在要做的事情是将个人信息页面 me.html 中表单搜集的数据存到数据库中，那数据库中就要建一张 person 表格
3.要完成这个目标，首先是在 me.html 中填好数据，然后用 DOM 操作获取页面的数据并存到一个 json 格式的 data 值中，然后为提交按钮写一个点击事件，点击以后通过 ajax 的 post 方法将 data 传给 php 文件，php 文件中获取收到的数据，和数据库建立链接，然后利用 sql 语句将数据放到数据库中保存起来。
同样，比如有一个页面 square_info.html 是展示所有的 esalon 活动信息，那么实现思路就是每当载入 square_info.html 的时候就通过 ajax get 请求向 php 文件请求数据，php 文件就也是首先连接数据库，然后通过 sql 语句进行数据库查询，将得到的数据存在 json 格式的 data 变量中，和这次请求是否成功的状态值一起传递给 square_info.html，如果状态值是 200，也就是成功，就执行一个回调函数，通过 dom 操作将得到的数据放在新增节点等的代码中，一起渲染页面。

这里面涉及到一个对 json 格式数据的解析，post 提交的时候，get 获取到的数据也是 json 字符串格式，这时候需要使用 eval 或者使用 JSON.parse 方法将 json 字符串解析称为 json 对象，这样一些 dom 操作才能取到正确的值。

```
//json 字符串转换成为 json 对象
data=eval("("+data+")");
```

```
//json 字符串转换成为 json 对象
data=JSON.parse(data);
```

```
var str="";
str+="姓名:"+data.name+"<br>";
str+="年龄:"+data.age+"<br>";
str+="性别:"+data.sex
div1.innerHTML=str;
```

然而在实际应用中，一般都是使 jquery 的$.ajax()更为方便，参数有：

```
$.ajax({
    type: 'GET',  // 这是请求的方式 可以是 GET 方式也可以是 POST 方式,默认是 GET
    url: ' xxx.php ',  // 这是请求的连接地址 一般情况下这个地址是后台给前端的一个连接，直接写就可以
    dataType: 'json', // 这是后台返回的数据类型 一般情况下都是一个 json 数据， 前端遍历一下就 OK
    async: true, // 默认为 true，默认为 true 时，所有请求均为异步请求，如果需要发送同步请求，需设置为 false,
    timeout: 8000, // 设置请求超时时间（毫秒）。此设置将覆盖全局设置
    data: {
        // 要传递的参数
        'xxx' : 'xxx',
        …… 
```

```
    },
    beforeSend: function () {
        // 在发送请求前就开始执行 （一般用来显示 loading 图）
    },
    success: function (data) {
        // 发送请求成功后开始执行，data 是请求成功后，返回的数据
    },
    complete: function () {
        //当请求完成后开始执行，无论成功或是失败都会执行 （一般用来隐藏 loading 图）
    },
    error: function (xhr, textStatus, errorThrown) {
        //  请求失败后就开始执行，请求超时后，在这里执行请求超时后要执行的函数
       //如果压根没成功发送到服务器端，也会执行这个方法，因为没法送成功就是请求超时
    }
}).done(function () {
    // 这个函数是在 ajax 数据加载完之后，对数据进行的判断，在涉及到对 ajax 数据进行操作无效时，在这个函数里面写是可以起到效果的
})
```

最后，ajax 不能进行跨域，它是严格遵守同源策略的，要想跨域 K

1.3 a 链接状态

lvha

a:link 是未访问的链接；a:visited 是已访问的链接；a:hover 是鼠标悬停在链接上；a:active 是被选择的链接

B

1．布局

写一个左侧固定右侧自适应的布局，float+padding

布局-左边栏固定，右边栏自适应，左边 float，右边 margin-left 可以吗

布局-两栏布局 flexbox

布局- flex 实现栅格布局，水平垂直居中

1.1 布局-两栏布局

```
1. <div class="left"></div>
   <div class="right"></div>
   .left{float:left;width:100px;}
   .right{margin-left:100px;}
```

2.圣杯(双飞翼)不要 right(extra)就是左边固定，右边自适应的两栏布局

参考下面，使用 middle 在前，left 在后，都是 float，left 定宽，middle 宽度 100%，left 的 magrin-left:-100%，然后使用整体包裹层 container 使用 padding 空出左边 200px，并相对定位让 left 的 left:-200px;。或者给 main 加上一个 main-inner 层设置 margin 空出左边的 200px。

3. 使用 float+overflow 实现

```
.left{width:100px;float:left;}
.right{overflow:hidden;}
```

overflow:hidden，触发 bfc 模式，浮动无法影响，隔离其他元素，IE6 不支持，左侧 left 设置 margin-left 当作 left 与 right 之间的边距，右侧利用 overflow:hidden 进行形成 bfc 模式

bfc 见下面解释

4. 使用 table 实现

```
.parent{display:table;table-layout:fixed;width:100%;}
.left{width:100px;}
```

.right,.left{display:table-cell;}

automatic	默认。列宽度由单元格内容设定。
fixed	列宽由表格宽度和列宽度设定。

5. 实用 flex 实现，布局-flex 见下方

.parent{display:flex;}

.left{width:100px;}

.right{flex:1;}

利用右侧容器的 flex:1，均分了剩余的宽度，也实现了同样的效果。而 align-items 默认值为 stretch，故二者高度相等

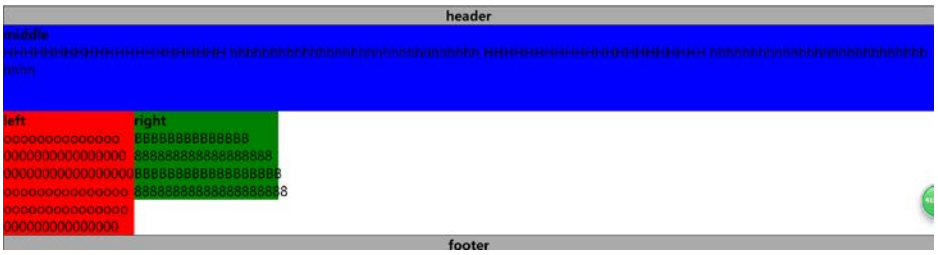
1.2 布局-三栏布局、圣杯、双飞翼

圣杯和双飞翼都是为了实现，左边 200px，右边 220px 宽度固定，中间自适应，container 包含左中右三个部分，它的高度保持一致。

圣杯：left middle right

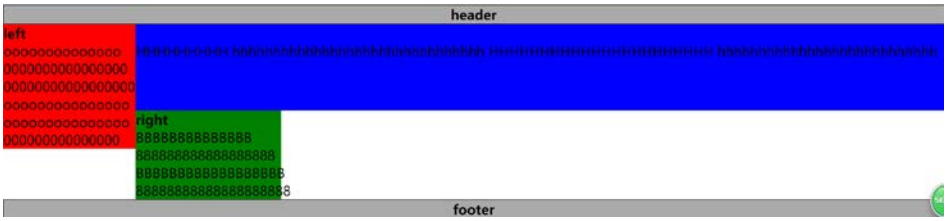
html 代码中 middle 部分首先要放在 container 的最前部分。然后是 left,right

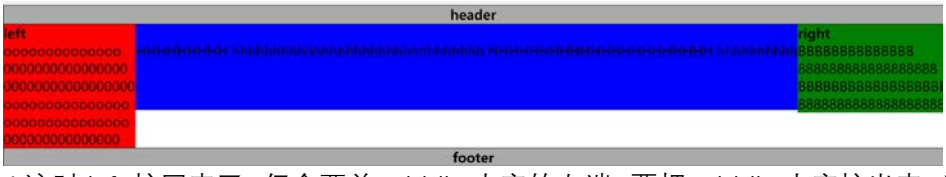
1. 将三者都 float:left，再加上一个 position:relative (因为相对定位后面会用到)，left 和 right 都定宽了。因为 middle 在最前，所以它在图中放在第一位。



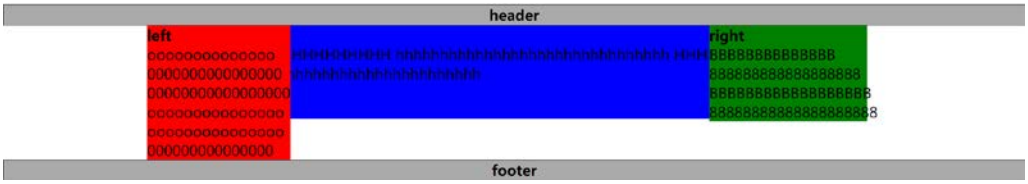
2.middle 部分 width:100%占满，此时样式上没有变化，因为本来没有 100%的时候也会默认

3.此时 middle 占满了，所以要把 left 拉到最左边，使用 margin-left:-100%，这个 100%是父元素的整个宽度，本来 left 是被挤下来到第二行的，现在左移 100%了，就会让 left 处于父元素最前面的位置，覆盖在 middle 上。同样，right 想要上去第一行，也需要 margin-left:负值，因为它本身就是三个元素中最后一个位置，所以只要左移自己的宽度-220px 就行了，同样是覆盖在 middle 上的。





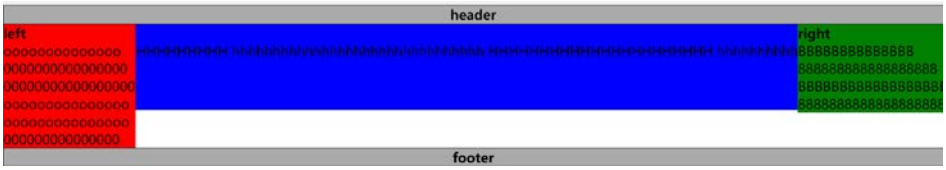
4.这时 left 拉回来了，但会覆盖 middle 内容的左端，要把 middle 内容拉出来，所以在外围 container 加上 padding:0 220px 0 200px(上右下左)，外围设置了 padding，那么 content 就会缩小，left、middle、right 都会随着 content 缩小。



5.middle 内容拉回来了，但 left 也跟着过来了，所以要还原，就对 left 使用相对定位 left:-200px 同理，right 也要相对定位还原 right:-220px。relative 生成相对定位的元素，相对于其在普通流中的位置进行定位。就是 left 本来在

上图这个位置，然后相对自己左移 200px，同样 right 本来是在上图这个位置，然后相对自己右移 220px。
6.到这里大概就自适应好了。如果想 container 高度保持一致可以给 left middle right 都加上 min-height:130px

双飞翼：sub main extra
双飞翼布局比圣杯布局多创建了一个 div，但不用相对布局了
1.html 代码中，main 要放最前边，sub　extra
2.将 main　sub　extra　都 float:left
3.将 main 占满　width:100%
4.此时 main 占满了，所以要把 sub 拉到最左边，使用 margin-left:-100%　同理　extra 使用 margin-left:-220px
（这时可以直接继续上边圣杯布局的步骤，也可以有所改动）如下所示：

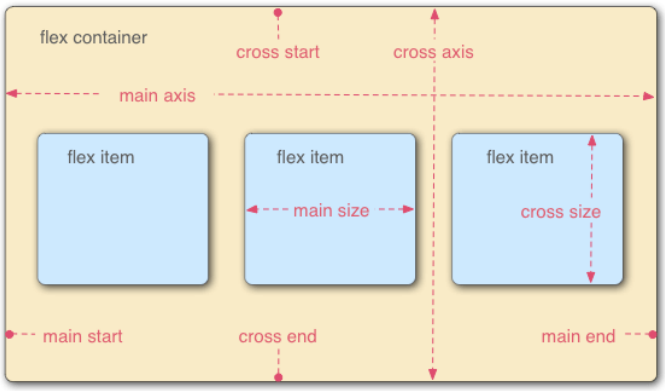


5.main 内容被覆盖了吧，除了使用外围的 padding，还可以考虑使用 margin。
给 main 增加一个内层 div-- main-inner，然后 margin:0 220px 0 200px，这样就可以不用使用相对定位了。
6.main 正确展示

1.3 布局- flex 弹性布局、栅格布局
Flex 是 Flexible Box 的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。
任何一个容器都可以指定为 Flex 布局、行内元素也可以使用 Flex 布局。
Webkit 内核的浏览器，必须加上-webkit 前缀。

```
.box{
   display: -webkit-flex; /* Safari */
   display: flex;
}
```

注意，设为 Flex 布局以后，子元素的 float、clear 和 vertical-align 属性将失效。
二、基本概念
采用 Flex 布局的元素，称为 Flex 容器（flex container），简称"容器"。它的所有子元素自动成为容器成员，称为 Flex 项目（flex item），简称"项目"。



容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis）。主轴的开始位置（与边框的交叉点）叫做 main start，结束位置叫做 main end；交叉轴的开始位置叫做 cross start，结束位置叫做 cross end。
项目默认沿主轴排列。单个项目占据的主轴空间叫做 main size，占据的交叉轴空间叫做 cross size。
三、容器的属性
以下 6 个属性设置在容器上。
flex-direction
flex-wrap
flex-flow
justify-content

align-items
align-content
3.1 flex-direction 属性
flex-direction 属性决定主轴的方向（即项目的排列方向）。
```
.box {
  flex-direction: row | row-reverse | column | column-reverse;
}
```
它可能有 4 个值。
row（默认值）：主轴为水平方向，起点在左端。
row-reverse：主轴为水平方向，起点在右端。
column：主轴为垂直方向，起点在上沿。
column-reverse：主轴为垂直方向，起点在下沿。
3.2 flex-wrap 属性
默认情况下，项目都排在一条线（又称"轴线"）上。flex-wrap 属性定义，如果一条轴线排不下，如何换行。
```
.box{
  flex-wrap: nowrap | wrap | wrap-reverse;
}
```
它可能取三个值。
（1）nowrap（默认）：不换行。
（2）wrap：换行，第一行在上方。
（3）wrap-reverse：换行，第一行在下方。
3.3 flex-flow
flex-flow 属性是 flex-direction 属性和 flex-wrap 属性的简写形式，默认值为 row nowrap。
```
.box {
  flex-flow: <flex-direction> || <flex-wrap>;
}
```
3.4 justify-content 属性
justify-content 属性定义了项目在主轴上的对齐方式。
```
.box {
  justify-content: flex-start | flex-end | center | space-between | space-around;
}
```
它可能取 5 个值，具体对齐方式与轴的方向有关。下面假设主轴为从左到右。
flex-start（默认值）：左对齐
flex-end：右对齐
center： 居中
space-between：两端对齐，项目之间的间隔都相等。
space-around：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。
3.5 align-items 属性
align-items 属性定义项目在交叉轴上如何对齐。
```
.box {
  align-items: flex-start | flex-end | center | baseline | stretch;
}
```
它可能取 5 个值。具体的对齐方式与交叉轴的方向有关，下面假设交叉轴从上到下。
flex-start：交叉轴的起点对齐。
flex-end：交叉轴的终点对齐。
center：交叉轴的中点对齐。
baseline: 项目的第一行文字的基线对齐。
stretch（默认值）：如果项目未设置高度或设为 auto，将占满整个容器的高度。
3.6 align-content 属性

align-content 属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

```
.box {
  align-content: flex-start | flex-end | center | space-between | space-around | stretch;
}
```

该属性可能取 6 个值。

flex-start：与交叉轴的起点对齐。

flex-end：与交叉轴的终点对齐。

center：与交叉轴的中点对齐。

space-between：与交叉轴两端对齐，轴线之间的间隔平均分布。

space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。

stretch（默认值）：轴线占满整个交叉轴。

四、项目的属性

以下 6 个属性设置在项目上。

order

flex-grow

flex-shrink

flex-basis

flex

align-self

4.1 order 属性

order 属性定义项目的排列顺序。数值越小，排列越靠前，默认为 0。

```
.item {
  order: <integer>;
}
```

4.2 flex-grow 属性

flex-grow 属性定义项目的放大比例，默认为 0，即如果存在剩余空间，也不放大。

```
.item {
  flex-grow: <number>; /* default 0 */
}
```

如果所有项目的 flex-grow 属性都为 1，则它们将等分剩余空间（如果有的话）。如果一个项目的 flex-grow 属性为 2，其他项目都为 1，则前者占据的剩余空间将比其他项多一倍。

4.3 flex-shrink 属性

flex-shrink 属性定义了项目的缩小比例，默认为 1，即如果空间不足，该项目将缩小。

```
.item {
  flex-shrink: <number>; /* default 1 */
}
```

如果所有项目的 flex-shrink 属性都为 1，当空间不足时，都将等比例缩小。如果一个项目的 flex-shrink 属性为 0，其他项目都为 1，则空间不足时，前者不缩小。

负值对该属性无效。

4.4 flex-basis 属性

flex-basis 属性定义了在分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为 auto，即项目的本来大小。

```
.item {
  flex-basis: <length> | auto; /* default auto */
}
```

它可以设为跟 width 或 height 属性一样的值（比如 350px），则项目将占据固定空间。

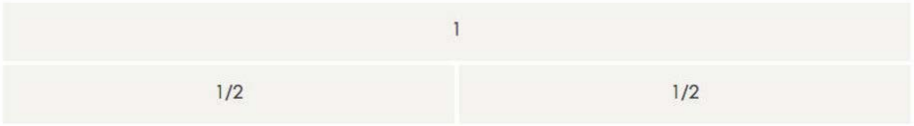
4.5 flex 属性

flex 属性是 flex-grow, flex-shrink 和 flex-basis 的简写，默认值为 0 1 auto。后两个属性可选。

```
.item {
```

flex: none | [<'flex-grow'> <'flex-shrink'>? || <'flex-basis'>]
}
该属性有两个快捷值：auto (1 1 auto) 和 none (0 0 auto)。
建议优先使用这个属性，而不是单独写三个分离的属性，因为浏览器会推算相关值。
4.6 align-self 属性
align-self 属性允许单个项目有与其他项目不一样的对齐方式，可覆盖 align-items 属性。默认值为 auto，表示继承父元素的 align-items 属性，如果没有父元素，则等同于 stretch。
.item {
align-self: auto | flex-start | flex-end | center | baseline | stretch;
}
该属性可能取 6 个值，除了 auto，其他都与 align-items 属性完全一致。

flex 栅格布局




```
<div class="grid">
    <div class="grid-cell">
        <div class="demo">1</div>
    </div>
</div>
<div class="grid">
    <div class="grid-cell">
        <div class="demo">1/2</div>
    </div>
    <div class="grid-cell">
        <div class="demo">1/2</div>
    </div>
</div>
```

```
.grid { display: flex; }
.grid-cell { flex:1 }
```

布局-自适应的布局
布局-响应式布局

2.BFC
BFC 是“块级格式化上下文”的意思，创建了 BFC 的元素就是一个独立的盒子，不过只有 Block-level box 可以参与创建 BFC，它规定了内部的 Block-level Box 如何布局，并且与这个独立盒子里的布局不受外部影响，当然它也不会影响到外面的元素。
生成 BFC 的情况有:
根元素
float 的值不为 none
overflow 的值不为 visible
display 的值为 inline-block, table-cell, table-caption
position 的值为 absolute 或 fixed
BFC 的约束规则:
内部的 Box 会在垂直方向上一个接一个的放置 ;两个元素都是 float:left;时, 还是同一行显示, 因为这是两个 bfc。
属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠。一个 bfc 的两个 div 的 margin 都设置成 10px 的话，其实它们俩中间还是只有 10px，而不是 20px。
每个元素的左外边距与包含块的左边界相接触（从左向右），即使浮动元素也是如此。(这说明 BFC 中子元素不

会超出他的包含块，而 position 为 absolute 的元素可以超出他的包含块边界）

BFC 的区域不会与 float 的元素区域重叠---所以可以制作两栏布局：比如左宽度固定，右栏自适应 则右栏设置 overflow:hidden 生成 BFC

计算 BFC 的高度时，浮动子元素也参与计算----所以可以清除子元素浮动父元素高度塌陷问题(父元素: overflow:hidden IE: zoom:1(hasLayout))

BFC 就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面元素，反之亦然

3.闭包(作用和缺陷)

3.1 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

闭包的两种使用情形：函数作为返回值(应用在柯里化 K)、函数作为参数传递

```
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);
```

还是打印 100，执行作用域还是定义时的作用域。

使用闭包实现变量和方法的私有化，的例子比如：

```
// 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。

3.2 闭包例题

1.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

2.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。所以需要使用立即执行的函数:()()
    }
    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。

3.

```
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；闭包中的 var foo={n:2};也是新创建的一个 n 变量，对原来的 n 对象没有影响。

4.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);
    }
}
```

5. 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

4. 变量类型与操作符

4.1 变量类型操作符-变量类型转换都是怎么转换的

4.1.1 变量类型操作符-字符串怎么转为数字

1. parseInt()和 parseFloat()

```
parseInt("010"); // 10
parseInt("010", 8); // 8
parseInt("010", 10); // 10
parseInt("")//NaN
parseInt()//NaN
parseInt(null)//NaN
parseInt(undefined)//NaN
```

2.Number()

```
Number(false)// 0
Number(true) //1
Number(undefined)// NaN
Number(null) //0
Number( "5.5 ") //5.5
Number( "56 ") //56
Number( "5.6.7 ") //NaN
Number(new Object())// NaN   Number({})//NaN
Number(100)// 100
Number(“”)//0
Number()//0
Number(0)//0
Number([])//0
Number([0])//0

另 Boolean([])//true
```

3.运算符操作- +

```
var str= '012.345 ';
var x = str-0;
var y = +str;
```

4. zhang=eval("1+1")//2

4.1.2 是将整个值从一种类型转换为另一种数据类型（称作基本数据类型转换），
基本数据类型转换的三种方法：
.转换为字符型：String()；例：String(678)的结果为"678" ；Number 的 toString 方法、Boolean 的 toString 方法、Date 的 toString 方法
.转换为数值型：Number()；例：Number("678")的结果为 678 、Number(date)// 返回 1404568027739
.转换为布尔型：Boolean()；例：Boolean("aaa")的结果为 true

Boolean(null),Boolean(false),Boolean(0),Boolean(''),Boolean()	false

4.2 变量类型操作符-null 和 undefine 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

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

4.3 变量类型操作符-等于和全等区别

```
// 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和编码方式(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

4.4 变量类型操作符-typeof 返回值

Typeof 一般只能分辨值类型，值类型也就是基本数据类型有 5 种：布尔、字符串、数值、null、undefined。引用类型有对象、数组、函数等。五种基本类型能识别 4 种，null 是特例，识别为 object；然后 function 也是特例可被识别为 function。

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
```

typeof(num) !== ‘number’

补充：typeof Symbol() //"symbol"

所以对于 null、非函数的对象是没办法用 typeof 判断变量类型的，判断对象的方式是：比如数组，有以下三种

Object.prototype.toString.call(o) === ″[Object Array]″

o instanceof Array;

this.constructor == Array;

第一种方法是使用 object 原型的 toString 方法，不是使用有可能被重定义的对象的 toString 方法。

5. box-sizing(控制盒模型的组成模式)

首先介绍盒模型 H

使用:

```
box-sizing: content-box | border-box; // for opera
-moz-box-sizing: content-box | border-box;
-webkit-box-sizing: content-box | border-box;
```

说明: 无论是什么盒模型，只要设置 box-sizing，就会按照 box-sizin 规定的来；当然标准盒模型的布局所占宽度 Width 默认就是 content-box 类型；IE 的布局所占宽度 Width 默认就是 border-box 模型。

1. content-box:

使用此值时，盒模型的组成模式是，布局所占宽度 Width = content + padding + border;

2. border-box:

使用此值时，盒模型的组成模式是，布局所占宽度 Width = content(即使设置了 padding 和 border，元素的宽度

也不会变).

C

1.存储

1.1 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 localstorge、cookies 等本地存储方式

浏览器端可以保存一些数据，需要的时候直接从本地获取。本地存储一般有 cookie、localStorage、sessionStorage 三种。服务器端也可以保存所有用户的数据，需要的时候浏览器要向服务器请求数据，服务器存储有 session 机制。

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），会在浏览器和服务器间来回传递。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器间不必要地来回传递。

它们的区别在于：

1.本地和服务器传递

Cookie 会在浏览器和服务器间来回传递

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

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。

3 有期时间：

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

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。

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

用 document.cookie=…可以获取值和修改值，调用方法比较简陋，所以一般需要封装一下再使用

6Web Storage 带来的好处：

使用 local storage 和 session storage 主要通过在 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器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.安全

使用 cookie 来存储用户名和密码等要比 localStorage 安全的多。A 安全

最后 session 数据放在服务器上。一般可以将登录等重要信息存放为 SESSION，它有有效期，对比 cookie 来说很安全。但是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，如果对服务器 session 的访问很多，当然效率性能会下降。服务器端一般用来保存用户的持久数据。Session 的应用见安全 A

还有其他本地存储方法：

IndexedDB：Web Storage 使用简单字符串键值对在本地存储数据，方便灵活，但是对于大量结构化数据存储力不从心，IndexedDB 是为了能够在客户端存储大量的结构化数据，并且使用索引高效检索的 API。
userData ：IE 6、7 专用 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。

2. 存储-cookie 与 http 头相关的属性
浏览器发送 cookie 时会发送哪几个部分?

```
HTTP/1.1 200 OK
Content-type: text/html
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]
跨域请求带上 cookie：
原生：xhr.withCredentials = true;
cors： header("Access-Control-Allow-Credentials: true");
header("Access-Control-Allow-Origin: http://www.xxx.com");

1.2 存储-缓存
web app 能不能做缓存

2.重绘回流
由于:

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。

在加载 CSS 文件时，为了避免出现首屏白屏，因此将 css 放在 head 中比较好。

ps：可以讲一遍解析 url 地址到渲染完成的过程，过程中提前以上两点。
所以：

js 会在页面已经渲染完成后加载，在如果 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。reflow 是不可避免的，只能尽量减小；repaint 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，比如要设置一个元素的隐藏和显示：
display：none 的元素不占据空间，涉及到了 DOM 结构，故产生 reflow(回流)与 repaint(重绘)
visibility：hidden 的元素不可见但存在，保留空间，不影响结构，故只产生 repaint(重绘)，故采用这种方法更佳。

补充：
1.导入样式：在.css 文件中使用@import url("...")来引入另一个 css 样式表
2.外部样式：在 html 页面中的 head 中使用 link 标签引入，如<link rel="stylesheet" type="text/css" href="a.css" />
3.内部样式(嵌入式)：在 HTML 页面中的 style 标签中使用样式 ，<style type="text/css">...</style>,放在 head 中
4.内联样式：与 html 标签的内部使用 style 属性设置的样式，直接与当前 html 标签相关联，如

html 中使用 script 标签

将 javascript 代码写到<script></script>之间，写在 body 末尾或者 head 中(window.onload=function(){…}表示页面加载完毕再执行 js 代码，否则可能导致无法获取对象的情况)

html 中添加外部 javascript 文件，<script type="text/javascript" src="a.js"></script>
html 中使用行内 javascript，<input type="button" value="点击有惊喜" onclick="javascript:alert('哈哈哈哈')">
js 中引入另一个 js 文件：
document.write("<script language=javascript src='/js/import.js'></script>");
不同于模块化，模块化是在 html 中引入一个 js，然后这个 js 自动引入它依赖的 js

除此以外还有很多操作会引发回流：

1.调整窗口大小
2.改变字体
3.增加或者移除样式表
4.内容变化，比如用户在 input 框中输入文字
5.激活 CSS 伪类，比如 :hover (IE 中为兄弟结点伪类的激活)
6.操作 class 属性
7.脚本操作 DOM
8.计算 offsetWidth 和 offsetHeight 属性
9.设置 style 属性的值

应对方案：

1.如果想设定/改变元素的样式，尽可能在 DOM 树的最末端通过添加去掉 class 类的方式。
2.避免设置多项内联样式
3.应用元素的动画，使用 position 属性的 fixed 值或 absolute 值，这样不会影响其他元素
4.权衡平滑和速度
5.避免使用 table 布局
6.避免使用 CSS 的 JavaScript 表达式 (仅 IE 浏览器)
repaint/reflow(浏览器加载文件)

D
E
F

1.浮动-清除浮动
浮动可以理解为让某个 div 元素脱离标准流，漂浮在标准流之上，和标准流不是一个层次。
Adiv 浮动、Bdiv 浮动，则 B 紧跟 A 后；
本来 Adiv 浮动、Bdiv 浮动，则 B 紧跟 A 后；然后 B 设置 clear:both 之后，就会换行
Adiv 不浮动(标准流)、Bdiv 浮动，则 B 换行；
Adiv 浮动、Bspan 不浮动(行内标准流)，则 B 紧跟 A 后。
浮动引起的问题：
1）父元素的高度无法被撑开，影响与父元素同级的元素
2）与浮动元素同级的非浮动元素（内联元素）会跟随其后

清除浮动是为了消除浮动元素高度塌陷的问题。当一个内层元素是浮动的时候，如果没有关闭浮动，且没有给其父元素设置高度，那么外层 div 也就不会再包含这个浮动的内层元素，因为此时浮动元素已经脱离了文档流。所以就会导致：
（1）：背景不能显示 （2）：边框不能撑开 （3）：margin 设置值不能正确显示
清除 css 浮动：
方法一：添加新的元素 、应用 clear：both；
HTML：

```
<div class="outer">
    <div class="div1">1</div>
```

```
    <div class="div2">2</div>
    <div class="div3">3</div>
    <div class="clear"></div>
</div>
```

CSS：

```
.clear{clear:both; height: 0; line-height: 0; font-size: 0}
```

方法二：父级 div 定义 overflow: auto（注意：是父级 div 也就是这里的 div.outer）

HTML：

```
<div class="outer over-flow"> //这里添加了一个 class
    <div class="div1">1</div>
    <div class="div2">2</div>
    <div class="div3">3</div>
</div>
```

CSS：

```
.over-flow{
    overflow: auto; zoom: 1; //zoom: 1; 是在处理兼容性问题
}
```

原理：使用 overflow 属性来清除浮动有一点需要注意，overflow 属性共有三个属性值：hidden,auto,visible。我们可以使用 hiddent 和 auto 值来清除浮动，但切记不能使用 visible 值，如果使用这个值将无法达到清除浮动效果，其他两个值都可以，计算 BFC 的高度时，浮动子元素也参与计算----所以可以清除子元素浮动父元素高度塌陷问题。

方法三:after 方法：（注意：作用于浮动元素的父亲）

它利用:after 和:before 来在元素内部插入两个元素块，本质上和增加一个 div 块并设置其样式为 clear:both 方法相同，这里利用伪类 clear:after 在元素内部增加一个类似于 div.clear 的效果。

```
 .outer {zoom:1;}       /*==for IE6/7 Maxthon2==*/
 .outer:after {clear:both;content:'.';display:block;width: 0;height: 0;visibility:hidden;}/*==for FF/chrome/opera/IE8==*/
```

注意 outer 后面没有空格。visibility:hidden;的作用是允许浏览器渲染 content 中的点，但是不显示出来

{clear: both; content: '';display: block;}内部这样设置也可以。

方法四：

divA、divB 是兄弟，A 的浮动让 B 清除了(B 设置样式 clear:both;)，B 自己又不浮动，这样父元素就可以被撑开

ps 题目

```
<div style="width:400px;height:200px;">
  <span style="float:left;width:auto;height:100%;">
          <i style="position:absolute;float:left;width:100px;height:50px;">hello</i>
  </span>
</div>
```

问题：span 标签的 width 和 height 分别为多少？

首先 span 不是块级元素，是不支持宽高的，但是 style 中有了个 float :left ;就使得 span 变成了块级元素支持宽高，height:100%;即为，200，宽度由内容撑开。

但是内容中的 i 是绝对定位，脱离了文档流，所以不占父级空间，所以 span 的 width=0，height 是 200px.

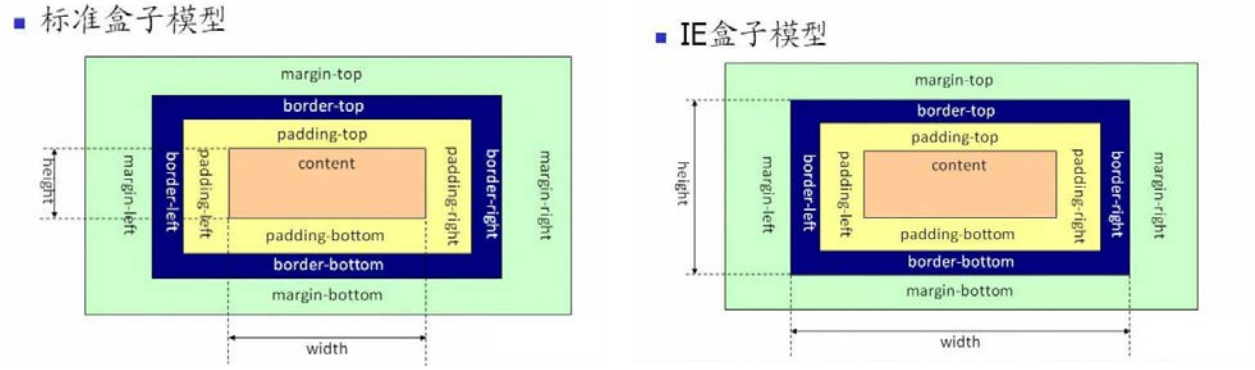
G

1.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
ifconfig 命令查看本机 IP 地址

H
W3C 盒子模型——属性高（height）和属性宽（width）这两个值不包含 填充（padding）和边框（border）
IE 盒子模型——属性高（height）和属性宽（width）这两个值包含 填充（padding）和边框（border）

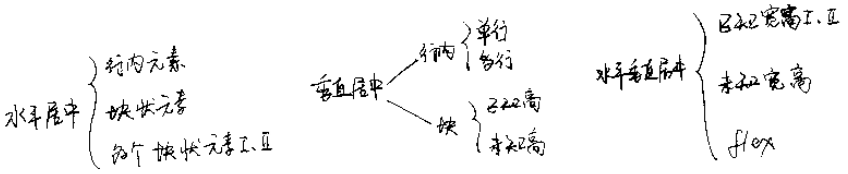


标准 w3c 盒子模型的范围包括 margin、border、padding、content，并且 content 部分不包含其他部分
ie 盒子模型的范围也包括 margin、border、padding、content，但它的 content 部分包含了 border 和 pading。

例：一个盒子的 margin 为 20px，border 为 1px，padding 为 10px，content 的宽为 200px、高为 50px;
假如用标准 w3c 盒子模型解释，
那么这个盒子需要占据的位置为：宽 20*2+1*2+10*2+200=262px、高 20*2+1*2*10*2+50=112px,
盒子的实际大小(border、padding、content)为：宽 1*2+10*2+200=222px、高 1*2+10*2+50=72px；
假如用 ie 盒子模型，
那么这个盒子需要占据的位置为：宽 20*2+200=240px、高 20*2+50=70px,
盒子的实际大小(border、padding、content)为：宽 200px、高 50px。

解决兼容型为题最简洁和值得推荐的方式是：将页面设为“标准模式”(在网页的顶部加上 doctype 声明)，添加对应的 dtd 标识；或者使用 hack 或者在外面套上一层 wrapper

I
J
1.居中问题



对于居中问题需要分行内元素、块状元素、行内块状元素来讨论，首先它们的区别是：
block, inline 和 inline-block 的区别
1 起新行
block 元素会独占一行，多个 block 元素会各自新起一行. 默认情况下, block 元素宽度自动填满其父元素宽度
inline 元素不会独占一行，多个相邻的行内元素会排列在同一行里，直到一行排列不下，才会新换一行，其宽度随元素的内容而变化
2 设置宽高
block 元素可以设置 width, height 属性. 块级元素即使设置了宽度，仍然独占一行
inline 元素设置 width, height 无效，但是可以设置 line-height
3 内外边距
block 元素可以设置 margin 和 padding 属性
inline 元素的 margin 和 padding 属性,水平方向的 padding-left, padding-right, margin-left, margin-right 都会产生边距效果. 但是竖直方向的 margin/padding-top/bottom 不会产生边距效果
4 包含
block 可以包含 inline 和 block 元素,而 inline 元只能包含 inline 元素
而 display: inline-block，则是将对象呈现为 inline 对象，但是对象的内容作为 block 对象呈现. 之后的内联对象会被排列到一行内. 比如我们可以给一个 link(a 元素)inline-block 的属性，使其既有 block 的高宽特性又有 inline 的同行特性
ps 几种元素举例
0.常见的空元素：br hr img input link meta　base area command embed keygen param source track wbr....
1.常见的块级元素(自动换行，可设置高宽)有：
div, h1-h6, p, pre, ul, li, ol, form, table, label, dl, dt, dd, hr
2.常见的行内元素（无法自动换行，无法设置宽高）有：
a,img,span,i,em,sub,sup,input, strong select,br,q,var,cite,code
3.常见的行块级元素(拥有内在尺寸，可设置高宽，不会自动换行)有：行内块元素也是行内元素
选择 button,select,输入 input,textarea, img 等

~~垂直居中对齐的标签定义是:<vertical-align:center>~~
~~text-left 用于左对齐~~
~~text-center 水平居中对齐~~
~~text-uppercase 可以将字母全部大写~~

水平居中：行内元素解决方案
只需要把行内元素包裹在一个属性 display 为 block 的父层元素中，并且把父层元素添加如下属性即可，因为行内元素就像是 text 一样。

```
.parent {
    text-align:center;
}
```

水平居中：块状元素解决方案，因为块状元素可以设置 margin

```
.item {
    /* 这里可以设置顶端外边距 */
    margin: 10px auto;
}
```

水平居中：多个块状元素解决方案，因为 inline-block 可以并排，inline 使之有像 text 一样的特性
将元素的 display 属性设置为 inline-block，并且把父元素的 text-align 属性设置为 center 即可:

```
.parent {
    text-align:center;
}
```

水平居中：多个块状元素解决方案（使用 flexbox 布局实现）
使用 flexbox 布局，只需要把待处理的块状元素的父元素添加属性 display:flex 及 justify-content:center 即可, justify-content:center 项目在主轴上的对齐方式，容器中子元素会按照规则对齐:

```
.parent {
    display:flex;
    justify-content:center;
}
```

垂直居中：单行的行内元素解决方案

```
.parent {
    background: #222;
    height: 200px;
}
/* 以下代码中，将 a 元素的 height 和 line-height 设置的和父元素一样高度即可实现垂直居中 */
a {
    height: 200px;
    line-height:200px;
    color: #FFF;
}
```

垂直居中：多行的行内元素解决方案
组合使用 display:table-cell 和 vertical-align:middle 属性来定义需要居中的元素的父容器元素生成效果，如下：

```
.parent {
    background: #222;
    width: 300px;
    height: 300px;
    /* 以下属性垂直居中 */
    display: table-cell;
    vertical-align:middle;
}
```

垂直居中：已知高度的块状元素解决方案

```
.item{
    top: 50%;
    margin-top: -50px;  /* margin-top 值为自身高度的一半 */
    position: absolute;//这个是为了让 top 值设置成功
    padding:0;
}
```

垂直居中：未知高度的块状元素解决方案

```
.item{
    top: 50%;
```

```
    position: absolute;
    transform: translateY(-50%);  /* 使用 css3 的 transform 来实现 */
}
```

水平垂直居中：已知高度和宽度的元素解决方案 1
这是一种不常见的居中方法，可自适应，比方案 2 更智能，如下：

```
.item{
    position: absolute;
    margin:auto;
    left:0;
    top:0;
    right:0;
    bottom:0;
}
```

水平垂直居中：已知高度和宽度的元素解决方案 2

```
.item{
    position: absolute;
    top: 50%;
    left: 50%;
    margin-top: -75px;  /* 设置 margin-left / margin-top 为自身高度的一半 */
    margin-left: -75px;
}
```

水平垂直居中：未知高度和宽度元素解决方案

```
.item{
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);  /* 使用 css3 的 transform 来实现 */
}
```

水平垂直居中：使用 flex 布局实现

```
.parent{
    display: flex;
    justify-content:center;
    align-items: center; /*该属性定义项目在交叉轴上如何对齐*/
    /* 注意这里需要设置高度来查看垂直居中效果 */
    background: #AAA;
    height: 300px;
}
```

对于不定宽的浮动元素我们也有一个常用的技巧解决它的水平居中问题。如下：
HTML 代码：

```
<div class="box">
    <p>我是浮动的</p>
    <p>我也是居中的</p>
</div>
```

CSS 代码：

```
.box{
    float:left;
    position:relative;
    left:50%;
}
p{
    float:left;
    position:relative;
    right:50%;
}
```

这样就解决了浮动元素水平居中了；

父元素和子元素同时左浮动，然后父元素相对左移动 50%，再然后子元素相对右移动 50%，或者子元素相对左移动-50%也就可以了。

K

1 跨域

首先之所以有跨域的需求是因为浏览器有同源策略，只要协议(http、https)、域名(www.baidu.com)、端口号有一个不同就不是同源的，就需要跨域。之所以要有同源限制，是因为比如一个黑客程序，他利用 iframe 引入不同源的银行登录页面，把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。但是在很多时候都需要在页面中使用 js 获取其他网站的数据，比如 ajax 请求不在同一个域的资源，如果没有跨域解决办法，ajax 会因为同源策略没办法获取到数据。

可跨域的三个标签<img>、<link>和<script>。img 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，但是 img 只能发送 get 请求，浏览器无法获取响应数据，所以只适用于浏览器与服务器之间的单向通信。但是 img 是一个古老的标签，它的兼容性非常好。<link>标签我们经常用来引入外部样式表，<script>用来引入外部 js 脚本，他们都可以使用 CDN，这样可以提高页面资源的加载速度，都是可以跨域的。但是肯定所有的跨域请求都必须经过信息提供方的允许。

利用<script>实现的跨域方法是 jsonp。json+padding（内填充）就是把 JSON 填充到一个盒子里。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返回的指定的函数，并且会把我们需要的 json 数据作为参数传入。例如我们需要跨域访问慕课网的一个接口，慕课网给了我们一个地址 http://imooc.com/api.js，当我们使用 script 引入这个 js 文件的时候，慕课网给我们返回的是一个函数的执行代码，这个函数名称就是前后端约定好的，比如是 callback，参数是我们需要的数据 data，那么返回内容的格式比如说就是 callback({x:100,y:200})，然后这个函数我们之前在网页中就已经定义了，那么就会顺利执行。所以就可以跨过同源限制得到所需要的数据。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。

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

跨域的第二种方法是 CORS (Cross-origin resource sharing)跨域资源共享，它的实现非常简洁，但是需要服务器端设置 http header，这也会是将来解决跨域问题的一个趋势。原理是将后台作为代理，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，当然前后台是同源的。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 进行跨域的访问。这种方法就支持很多操作了。

```
// 注意：不同后端语言的写法可能不一样

// 第二个参数填写允许跨域的域名称，不建议直接写 "*"
response.setHeader("Access-Control-Allow-Origin", "http://a.com, http://b.com");
response.setHeader("Access-Control-Allow-Headers", "X-Requested-With");
response.setHeader("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS");

// 接收跨域的cookie
response.setHeader("Access-Control-Allow-Credentials", "true");
```

还有其他跨域方法
比如在主域相同子域不同的情况下，想让两个子域实现跨域，可以设置 document.domain：只适用于；
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。比如两个子域名 aaa.xxx.com 和 bbb.xxx.com 的基础域名一样，aaa 里的一个网页（a.html）引入了 bbb 里的一个网页（b.html），然后想在 a.html 里操作 b.html 里面的内容的。就可以通过 Javascript，将两个页面的 domain 改成一样的，需要在 a.html 里与 b.html 里都加入：document.domain = "xxx.com";这样这两个页面就可以互相操作了。

还可以使用 window.name+iframe 进行跨域。
window 对象 name 属性有一个特征：在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个 window.name 的，每个页面对 window.name 都有读写的权限。
比如有两个页面 a.com/app.html 是应用页面。b.com/data.html 是数据页面。
在应用页面（a.com/app.html）中创建一个 iframe，把其 src 指向数据页面（b.com/data.html）。数据页面会把数据附加到这个 iframe 的 window.name 上(这也是信息提供方允许的证明)，比如是 window.name = 'I was there!'； 这里数据格式可以是 json 或者字符串。 然后在应用页面（a.com/app.html）中监听 iframe 的 onload 事件，一旦载入成功就可以通过 iframe.contentWindow.name 使用 data 数据了。获取数据以后使用 removeChild 销毁这个 iframe 来释放内存和保证了安全（不被其他域 frame js 访问）。iframe 的 src 属性指向外域，跨域数据即由 iframe 的 window.name 从外域传递到本地域。这个就巧妙地绕过了浏览器的跨域访问限制，但同时它又是安全操作。

还可以使用 postMessage(data,origin)方法接受两个参数实现跨域

2. 柯里化

```
3. var fn = function (a, b, c) {return a + b + c}; curryIt(fn)(1)(2)(3);
```

```
function currylt(fn) {
    return function a(a){
        return function b(b){
            return function c(c){
                return fn(a,b,c);
            }
        }
    }
}
```

L

M
1. 模块化

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 });
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：

```
// util.js
function getFormatDate(date, type) {
    // type === 1 返回 2017-06-15
    // type === 2 返回 2017年6月15日 格
    // ……
}

// a-util.js
function aGetFormatDate(date) {
    // 要求返回 2017年6月15日 格式
    return getFormatDate(date, 2)
}

// a.js
var dt = new Date()
console.log(aGetFormatDate(dt))
```

使用：

```
<script src="util.js"></script>
<script src="a-util.js"></script>
<script src="a.js"></script>

<!-- 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 -->
<!-- 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

```
// util.js
export {
    getFormatDate: function (date, type) {
        // type === 1 返回 2017-06-15
        // type === 2 返回 2017年6月15日 格式
    }
}

// a-util.js
var getFormatDate = require('util.js')
export {
    aGetFormatDate: function (date) {
        // 要求返回 2017年6月15日 格式
        return getFormatDate(date, 2)
    }
}
```

```
// a.js
var aGetFormatDate = require('a-util.js')
var dt = new Date()
console.log(aGetFormatDate(dt))

// 直接`<script src="a.js"></script>`，其他的根据依赖关系自动引用
// 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

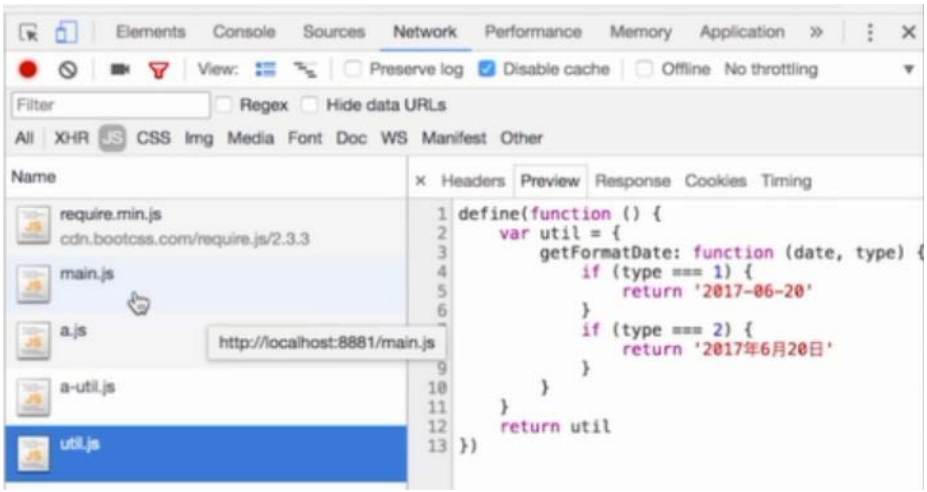
```
// util.js
define(function () {
    return {
        getFormatDate: function (date, type) {
            if (type === 1) {
                return '2017-06-15'
            }
            if (type === 2) {
                return '2017年6月15日'
            }
        }
    }
})

// a-util.js
define(['./util.js'], function (util) {
    return {
        aGetFormatDate: function (date) {
            return util.getFormatDate(date, 2)
        }
    }
})
```

```
// a.js
define(['./a-util.js'], function (aUtil) {
    return {
        printDate: function (date) {
            console.log(aUtil.aGetFormatDate(date))
        }
    }
})

// main.js
require(['./a.js'], function (a) {
    var date = new Date()
    a.printDate(date)
})
```

main.js 是一个入口：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <p>AMD test</p>

    <script src="/require.min.js" data-main="./main.js"></script>
</body>
</html>
```

require.js 在加载的时候会检察 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

```
// define(['./util.js'], function (util) {
//     var aUtil = {
//         aGetFormatDate: function (date) {
//             return util.getFormatDate(date, 2)
//         }
//     }
//     return aUtil
// })

define(function () {
    return {
        aGetFormatDate: function () {
            return 'abc'
        }
    }
})
```

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：

```
// util.js
module.exports = {
    getFormatDate: function (date, type) {
        if (type === 1) {
            return '2017-06-15'
        }
        if (type === 2) {
            return '2017年6月15日'
        }
    }
}

// a-util.js
var util = require('util.js')
module.exports = {
    aGetFormatDate: function (date) {
        return util.getFormatDate(date, 2)
    }
}
```

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

N

1.内存泄漏和垃圾回收

1.1 内存泄漏指任何对象在不再拥有或需要它之后仍然存在。

http://www.jb51.net/article/81117.htm

1.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);
```

2. 闭包
在一个闭包中引用的任何局部变量都会被该闭包保留，只要该闭包存在就永远保留
3. 控制台日志
由 console.log 和 console.dir 方法记录的对象，会一直保留在内存中
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）
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。

1.2 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。
在高版本 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。

2. new 关键字中间发生了哪些事情
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空对象，然后将 this 指向这个对象，然后根据传参给 this 对象添加上属性和方法，最后返回 this 对象并赋值给 b。
另外即使使用 var a = {}这样的赋值方法，实际上也是使用了 new 方法，这是一种语法糖格式，所以还是能够使用 instanceof 函数来判断这个对象的构造函数是否是某个函数；另外从可读性和性能上讲推荐使用语法糖
说到 instanceof，还有三种方法是用来判断不同类型变量的类型的：
typeof o == 'number'　　//number、string、undefined、boolean、function
Object.prototype.toString.call(o) === "[Object Array]"//所有类型适用
o instanceof Array;//对象等
this.constructor == Array;//所有

O
P
Q
1.前端
如何学习前端：书(红宝书、js 权威指南、css 禅意花园、html5 权威指南)，w3c 和官方文档，博客(王旭鑫、csdn、github)，~~微信公众号~~，知乎，问大神，慕课网、妙味课堂

R
S
1.深拷贝浅拷贝
当执行 var a = b 时,首先需要考虑的是 b 变量是什么类型，js 中的变量类型有两种，值类型比如数值、字符串、布

尔值、null、undefined，对它们的复制就是简单的浅拷贝，会新建一个变量 a，a 中存放的值和 b 相同，a、b 的改变互不影响。但是如果 b 是一个对象，那么 a 变量只是指向 b 变量也指向的那个内存空间引用，这时无论 a 还是 b 的操作都会改变该内存中存放的只，也只是浅拷贝。要想真正实现一个对象的复制也就是深拷贝，需要用到以下代码：

```
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。

2.事件

2.1 事件机制

--首先解释事件流的概念

DOM(文档对象模型)结构是一个树型结构，当一个 HTML 元素产生一个事件时，该事件会在元素结点与根结点之间的路径传播，路径所经过的结点都会收到该事件，这个传播过程可称为 DOM 事件流。

--然后事件冒泡和时间捕获

事件流的流向分为冒泡事件和捕获事件。冒泡事件规定，事件由叶子结点沿祖先结点一直向上传递直到根结点；从浏览器界面视图 HTML 元素排列层次上理解就是事件由特定的事件节点一直传递到根节点。事件捕获的方向和它相反。

--事件模型

然后标准事件模型的事件机制包含了捕获和冒泡两个过程：事件捕获->事件处理->事件冒泡

而 IE 的事件模型机制只有事件冒泡过程。

--怎么定义事件

可以直接在 DOM 中绑定事件，但是这样就不能对同一种事件绑定多个事件处理函数。

还可以使用事件绑定函数，普通浏览器用 addEventListener；IE 低版本用 attachEvent，使用 addEventListener 可以指定事件流方向，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；

```
var btn = document.getElementById('btn1')
btn.addEventListener('click', function (event) {
    console.log('clicked')
})

function bindEvent(elem, type, fn) {
    elem.addEventListener(type, fn)
}
var a = document.getElementById('link1')
bindEvent(a, 'click', function(e) {
    e.preventDefault() // 阻止默认行为
    alert('clicked')
})
```

ps：阻止默认行为，比如点击 a 标签不要打开链接，而是弹出一句话，要使用 preventDefault()函数；想要阻止事件冒泡可以使用 stopPropagation

2. 用事件冒泡实现事件代理，来降低内存。
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。
2. 1 用事件冒泡实现点击弹出不同信息

```html
<body>
    <div id="div1">
        <p id="p1">激活</p>
        <p id="p2">取消</p>
        <p id="p3">取消</p>
        <p id="p4">取消</p>
    </div>
    <div id="div2">
        <p id="p5">取消</p>
        <p id="p6">取消</p>
    </div>
</body>
```

```js
var p1 = document.getElementById('p1')
var body = document.body
bindEvent(p1, 'click', function (e) {
    e.stopPropatation()
    alert('激活')
})
bindEvent(body, 'click', function (e) {
    alert('取消')
})
```

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
2.适配
2.1 适配-媒体查询
media type(媒体类型)是 css 2 中的属性，通过 media type 可以对不同的设备指定特定的样式，从而实现更丰富的界面。media query(媒体查询)是对 media type 的一种增强。通常会用到的 media type 会是 all(表示适配所有) 和 screen(表示彩色电脑屏幕)，然后是 print(表示打印时的样式)。
媒体类型的使用方法是：

```
@media screen{
    selector{rules}
}
```

语法：由 media type+一到多个 CSS 属性判断组成，而多个 CSS 属性判断可以用关键字 and 连接：

```
@media screen and (min-width:1024px) and (max-width:1280px){
    body{font-size:medium;}
}
```

css 属性判断还有 device-aspect-ratio 宽高比、device-height、device-width、resolution 分辨率、orientation 横屏或者竖屏。
not 和 only
not：排除某种制定的媒体类型

```
@media not print and (color){
}
```

only：指定某种特定的媒体类型，可以用来排除不支持媒体查询的浏览器：

```
@media only screen and (color){
}
```

检测 iPhone safari：

```
<link media="only screen and (max-device-width: 480px)" href="style.css">
```

3.设计模式
使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。
3.1 单例模式
单体模式在我们平时的应用中用的比较多的，相当于把我们的代码封装在一个起来，只是暴露一个入口，从而避免全部变量的污染。实例化一次

```
/*Basic Singleton*/
var Singleton = {

    attribute:true,

    method1:function(){},

     method2:function(){}
};
```

3.2 单例模式
单例就是保证一个类只有一个实例，实现的方法一般是先判断实例存在与否，如果存在直接返回，如果不存在就创建了再返回，使用闭包生成。 单例模式是一种常用的模式，有一些对象我们往往只需要一个，比如全局缓存、浏览器的 window 对象。
3.3 工厂模式
用函数封装以特定接口创建对象。其实现方法非常简单，也就是在函数内创建一个对象，给对象赋予属性及方法再将对象返回即可。

```
function createBlog(name, url) {
    var o = new Object();
    o.name = name;
    o.url = url;
    o.sayUrl= function() {
        alert(this.url);
    }
    return o;
}
var blog1 = createBlog('wuyuchang', 'http://www.jb51.net/');
```

3.4 构造函数模式

```
function Blog(name, url) {
    this.name = name;
    this.url = url;
    this.alertUrl = function() {
        alert(this.url);
    }
}

var blog = new Blog('wuyuchang', 'http://www.jb51.net/');
console.log(blog instanceof Blog);   // true， 判断 blog 是否是 Blog 的实例，即解决了工厂模式中不能
```

没有显示的创建对象
直接将属性和方法赋值给了 this 对象
没有 return 语句
使用 new 创建对象
3.5 原型模式
们创建的每个函数都有 prototype（原型）属性，这个属性是一个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。使用原型对象的好处就是可以让所有对象实例共享它所包含的属性及

方法。

```
function Blog() {
}

Blog.prototype.name = 'wuyuchang';
Blog.prototype.url = 'http://tools.jb51.net/';
Blog.prototype.friend = ['fr1', 'fr2', 'fr3', 'fr4'];
Blog.prototype.alertInfo = function() {
    alert(this.name + this.url + this.friend );
}

// 以下为测试代码
var blog = new Blog(),
    blog2 = new Blog();
blog.alertInfo();    // wuyuchanghttp://tools.jb51.net/fr1,fr2,fr3,fr4
blog2.alertInfo();    // wuyuchanghttp://tools.jb51.net/fr1,fr2,fr3,fr4

blog.name = 'wyc1';
blog.url = 'http://***.com';
blog.friend.pop();
blog2.name = 'wyc2';
blog2.url = 'http://+++.com';
blog.alertInfo();    // wyc1http://***.comfr1,fr2,fr3
blog2.alertInfo();    // wyc2http://+++.comfr1,fr2,fr3
```

3.6 混合模式

混合模式（原型模式 + 构造函数模式）

```
function Blog(name, url, friend) {
  this.name = name;
  this.url = url;
  this.friend = friend;
}

Blog.prototype.alertInfo = function() {
  alert(this.name + this.url + this.friend);
}
var blog = new Blog('wuyuchang', 'http://tools.jb51.net/', ['fr1', 'fr2', 'fr3']),
  blog2 = new Blog('wyc', 'http://**.com', ['a', 'b']);

blog.friend.pop();
blog.alertInfo();  // wuyuchanghttp://tools.jb51.net/fr1,fr2
blog2.alertInfo();    // wychttp://**.coma,b
```

T

U

1.url(输入一个 url 以后发生的)+渲染原理

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。
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

5.根据 HTML 结构生成 DOM Tree
6.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，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
7.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
8.根据渲染树开始渲染和展示
9.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
10.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，

V
W
1.网络五层
TCP/IP 五层协议:应用层、传输层、网络层、数据链路层、物理层.http 对应应用层

2.网络状态码
100-199 : 表示成功接收请求，要求客户端继续提交下一次请求才能完成整个处理过程
200-299: 表示成果接收请求并已完成整个处理过程. 常用 200
300-399: 为完成请求，客户需进一步细化需求: 例如: 请求的资源已经移动一个新地址，常用 302(重定向)，307 和 304(拿缓存)
400-499: 客户端的请求有错误，包含语法错误或者不能正确执行. 常用 404(请求的资源在 web 服务器中没有) 403(服务器拒绝访问，权限不够)
500-599: 服务器端出现错误

常用:
200 正常 表示一切正常，返回的是正常请求结果
302/307 临时重定向 指出请求的文档已被临时移动到别处，此文档的新的 url 在 location 响应头中给出
304 未修改 表示客户机缓存的版本是最新的，客户机应该继续使用它
403 禁止 服务器理解客户端请求，但拒绝处理它，通常用于服务器上文件或目录的权限设置所致
404 找不到 服务器上不存在客户机所请求的资源

500　　服务器内部错误　　服务器端的 cgi, asp, jsp 等程序发生错误

3.网络-三次握手四次挥手
三次握手:
首先 Client 端发送连接请求报文，Server 段接受连接后回复 ACK 报文(确认字符)，并为这次连接分配资源。Client 端接收到 ACK 报文后也向 Server 段发生 ACK 报文，并分配资源，这样 TCP 连接就建立了。
第一步: 客户机的 TCP 先向服务器的 TCP 发送一个连接请求报文.
第二步: 服务器端的 TCP 收到连接请求报文后，若同意建立连接，就向客户机发送请求，并为该 TCP 连接分配 TCP 缓存和变量.
第三步: 当客户机收到确认报文后，还要向服务器给出确认，并且也要给该连接分配缓存和变量.
四次挥手
第一步: 客户机打算关闭连接,就向其 TCP 发送一个连接释放报文,并停止再发送数据,主动关闭 TCP 连接,
第二步: 服务器接收连接释放报文后即发出确认,此时，从客户机到服务器这个方向的连接就释放了，TCP 连接处于半关闭状态. 但服务器若发送数据，客户机仍要接收，即从服务器到客户机的连接仍未关闭.
第三步: 若服务器已经没有了要向客户机发送的数据，就通知 TCP 释放连接,
第四步: 客户机收到连接释放报文后，必须发出确认.此时, TCP 连接还没有释放掉，必须经过等待计时器设置的时间后, A 才进入到连接关闭状态.

4.网络-TCP 与 UDP
TCP 是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次“对话”才能建立起来
UDP 是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去。 UDP 适用于一次只传送少量数据、对可靠性要求不高的应用环境。

5.http 和 https
HTTP 协议通常承载于 TCP 协议之上，在 HTTP 和 TCP 之间添加一个安全协议层（SSL 或 TSL）。

6. WebSocket

在实现 websocket 连线过程中，需要通过浏览器发出 websocket 连线请求，然后服务器发出回应，这样浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。优点是互相沟通的 Header 很小；实现了服务器的推送，服务器不再被动的接收到浏览器的 request 之后才返回数据，而是在有新数据时就主动推送给浏览器。

7.WebWorker
是运行在后台的 javascript，不会影响页面的性能。
在 HTML 页面中运行脚本时，会阻塞页面渲染，直到脚本运行结束。如果使用了 webworker，用户可以对页面进行点击，选取等，因为 webworker 运行在服务器，不会影响。

X
1.优化-性能优化
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)

首先可以从加载页面和静态资源、页面渲染这两方面入手进行性能优化

加载资源优化
1. 静态资源的压缩、合并、优化：css、js、图片

合并：CommonJS 打包合并，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；源码压缩：margin-top,margin-left 变成 margin x x x x; 使用雪碧图缩小整体图片的大小、减少 http 访问;
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
5.使用 GIF 图片格式，它提供的颜色较少，可用在一些对颜色要求不高的地方

页面渲染优化：
1.使用服务端渲染 SSR 让后端去完成渲染，服务器将每个要展示的页面都运行完成后，将整个相应流传送给浏览器，所有的运算在服务器端都已经完成，数据直接输出到 HTML。
2. 渲染时减少 cpu 计算量：标明图片的高度和宽度（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。）

3.css 放前面，js 放后面，这里再说一下 url 加载过程

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

```
<img id="img1" src="preview.png" data-realsrc="abc.png"/>
<script type="text/javascript">
    var img1 = document.getElementById('img1')
    img1.src = img1.getAttribute('data-realsrc')
</script>
```

5.缓存 DOM 查询的结果

```
// 未缓存 DOM 查询
var i
for (i = 0; i < document.getElementsByTagName('p').length; i++) {
    // todo
}

// 缓存了 DOM 查询
var pList = document.getElementsByTagName('p')
var i
for (i = 0; i < pList.length; i++) {
    // todo
}
```

3 行每次循环都会执行一次 dom

6.合并 DOM 插入 createDocumentFragment 片段

```
var listNode = document.getElementById('list')

// 要插入 10 个 li 标签
var frag = document.createDocumentFragment();
var x, li;
for(x = 0; x < 10; x++) {
    li = document.createElement("li");
    li.innerHTML = "List item " + x;
    frag.appendChild(li);
}

listNode.appendChild(frag);
```

7. 事件节流

```
var textarea = document.getElementById('text')
var timeoutId
textarea.addEventListener('keyup', function () {
    if (timeoutId) {
        clearTimeout(timeoutId)
    }
    timeoutId = setTimeout(function () {
        // 触发 change 事件
    }, 100)
})
```

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

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

9.减少重绘回流

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

11.使用父元素代理重复子元素的事件

```
<div id="div1">
    <a href="#">a1</a>
    <a href="#">a2</a>
    <a href="#">a3</a>
    <a href="#">a4</a>
    <!--会随时新增更多 a 标签-->
</div>
```

```
var div1 = document.getElementById('div1')
div1.addEventListener('click', function (e) {
    var target = e.target
    if (target.nodeName === 'A') {
        alert(target.innerHTML)
    }
})
```

Y

1.原型链

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。
2.原型链继承例子

```
// 动物
function Animal() {
    this.eat = function () {
        console.log('animal eat')
    }
}
// 狗
function Dog() {
    this.bark = function () {
        console.log('dog bark')
    }
}
Dog.prototype = new Animal()
// 哈士奇
var hashiqi = new Dog()
```

4.异步和同步
4.1.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)
```

4.2.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，setInverval
- 网络请求：ajax 请求，动态 <img> 加载
- 事件绑定

因为 js 是单线程的，每次只能做一件事，但是如果所有事情都用同步机制，用户体验就会很差，用户需要等待一件事完全做完才能做下一件事情，所以需要异步机制来提高用户体验，也是性能。一般需要做的就是异步加载一些资源。比如怎么加载 js 能够让用户体验更好。一般情况下浏览器会立即加载并执行指定的脚本，“立即”指的是在渲染该 script 标签之下的文档元素之前，也就是说不等待后续载入的文档元素，读到就加载并执行。所以需要一些方法来实现异步加载 js 代码：
1.dom 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
有 async，加载和渲染后续文档元素的过程将和 script.js 的加载与执行并行进行（异步）

有 defer，加载后续文档元素的过程将和 script.js 的加载并行进行（异步），但是 script.js 的执行要在所有元素解析完成之后，DOMContentLoaded 事件触发之前完成。async 则是一个乱序执行的主，反正对它来说脚本的加载和执行是紧紧挨着的，所以不管你声明的顺序如何，只要它加载完了就会立刻执行。
所以 async 对于应用脚本的用处不大，因为它完全不考虑依赖（哪怕是最低级的顺序执行），不过它对于那些可以不依赖任何脚本或不被任何脚本依赖的脚本来说却是非常合适的。
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。
比如将<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。

Z
1.作用域和变量提升

- **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**这些都是 js 的作用域

变量提升就是将变量声明提升到作用域的第一行，在普通变量定义之前使用会报 undefined；在函数声明之前调用函数能够成功，因为函数声明会包括函数体整体提升；在函数表达式之前调用会 error，因为是函数表达式只会提升函数名，函数值为 undefined，undefined()不是一个函数。
例题 1：

```
(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。
例题 2：

```
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()，所以会去先去调用函数 g，这里就会直接抛出异常，所以是 Error。