

## 前端面试题

---

### web 前端优化

- 1、JS/CSS： JS 会阻塞页面渲染，放在页面最下方； CSS 在页面顶部加载。避免 CSS 表达式 合并重用相同的代码块，对文件进行合并压缩，减少 HTTP 请求数
  - 2、图片：合并压缩图片，减少 HTTP 请求数
  - 3、页面优化： 结构语义化，优化 body 里面的标签 优化 `title`， `description`， `keywords` 等
  - 4、服务端： 开启 `gzip` / `bzip2` 压缩；使用 HTTP 的 `keep-alive` 减少连接数；开启 `Etags`，实体标签，进行页面缓存；开启 `APC`， `opcode` 缓存； `memcache` / `redis` 数据缓存；代码优化；
- 

### ajax 跨域

- 代理
  - `dataType: "jsonp"` 或是一些其他的可以跨域的标签 (`img` `iframe` 等)
  - `header("Access-Control-Allow-Origin:");`
  - jQuery `getJSON()` (`jquery` `jsonp`)
- 

### 事件捕获/事件冒泡

- 捕获由父级到子级
- 冒泡由子级到父级

`element.addEventListener(event, function, useCapture);`

`useCapture`

`true` - 事件句柄在捕获阶段执行

`false` - `false` - 默认。事件句柄在冒泡阶段执行

- 事件按顺序绑定
- 

### 事件委托/事件绑定

- 委托在处理速度和内存占用上都优于事件绑定
- 但委托会在事件冒泡中造成性能损失

委托对未来元素有效，事件绑定只对当前已有元素有效，对脚本后来生成的元素无效

`on > delegate > live > bind`

- `bind` 直接绑定，只对现有元素有效
  - `live` 通过冒泡匹配到对应的元素，对未来有效
  - `delegate` 相对于 `live` 更精确
  - `on` 是以上几种的综合体
- 

### 立即执行函数

`(function() {}())`

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

只用于函数表达式 例：

`var kof = function() {}()`

可以模仿出一个私有作用域，不会遗留全局变量，不会污染全局空间，用于 JS 模块化编程，又被称作“匿名包裹器”、“命名空间”

---

### 阻止冒泡

- `return false`
  - `event.target == event.currentTarget`
  - `event.stopPropagation`
  - `event.preventDefault`
-

## 为什么使用闭包

进行信息的隐藏和封装，模拟一些 OO 的特性，避免变量污染

## 变量提升 (Hoisting)

会将变量声明过程提升到顶部

```
var a = 6;
setTimeout(function () {
  alert(a);
  var a = 666;
}, 1000);
a = 66;
// undefined
```

## 延迟加载

`setTimeout()` / `clearTimeout()`

lazyload.js 通过 `ajax` 实现延迟加载

`setTimeout()` / `clearTimeout()`

加入到队列末执行

```
var kof = 6;
setTimeout(function () {
  alert(kof);
}, 1000);
a = 66;
// 66
```

用于清除 jQuery 操作

`setInterval()` / `clearInterval()`

用于计时器操作

## 模块化编程思想

模块化编程，低内聚，高耦合

保证各模块之间的独立性，使各模块之间的依赖关系变的更加明确

- 原始写法:

```
function module () {

}
```

污染全局变量，变量名可能冲突，各模块之间依赖关系不明确

- 面向对象:

```
var module = new Object({

});
```

会暴露模块成员变量，能够从外部修改

- 立即执行函数:

```
var module = function () {
  var _sum = 0;
  var init = 100;

  var count = function () {
    return _sum + 1;
  };
};
```

```
    return {
      init: init;
      count: count;
    };
  }();
```

标准模块写法，只能访问 return 的值

- 继承：

```
var module = function (mod) {
  mod.k1 = function () {

  };
  return mod;
}(module);
```

- 防止空对象报错

```
var module = function (mod) {
  mod.k1 = function () {

  };
  return mod;
}(module || {});
```

- 输入全局变量

```
var module = function ($) {

}(jQuery);
```

显示输入其他模块

---

## CommonJS 规范

同步加载模块，如果模块太大会阻塞渲染

后期 `seaJS` / `CMD`

---

## AMD 规范

异步模块定义

`require.js` / `curl.js`

```
<script src="js/require.js" defer async="true"></script>
```

`require([module], callback);`

`defer` : HTML4 中的异步加载 `async` : HTML5 中的异步加载，不阻塞渲染

---

## JavaScript 数据类型

### 数据类型

- `boolean` —— 布尔值；
- `string` —— 字符串；
- `number` —— 数值；
- `null` —— 空；
- `object` —— 对象；
- `undefined` —— 未定义；

- `function` —— 函数；
  - `array` —— 数组；
-

- `json` —— `json`;
- `NaN` —— 非法数字 (not a number)

## jsonp 原理

- 利用<script>标签没有跨域限制的“漏洞”，来实现与第三方的通信
- 只支持 GET 请求而不支持 POST 等其它类型的 HTTP 请求
- 动态添加一个 `script` 标签，而 `script` 标签的 `src` 属性是没有跨域的限制的。这样说来，这种跨域方式其实与 `ajax XMLHttpRequest` 协议无关了
- jQuery 只支持 `get` 方式的 `jsonp` 实现

```
$.ajax({
  async:false,
  url: http://跨域的 dns/document!searchJSONResult.action,
  type: "GET",
  dataType: 'jsonp',
  jsonp: 'jsoncallback',
  data: qsData,
  timeout: 5000,
  beforeSend: function() {
    //jsonp 方式此方法不被触发. 原因可能是 dataType 如果指定为 jsonp 的话, 就已经不是 ajax 事件了
  },
  success: function (json) { //客户端 jquery 预先定义好的 callback 函数, 成功获取跨域服务器上的 json 数据后, 会动态执行这个 callback 函数
    if(json.actionErrors.length!=0) {
      alert(json.actionErrors);
    }
    genDynamicContent(qsData, type, json);
  },
  complete: function(XMLHttpRequest, textStatus){
    $.unblockUI({ fadeOut: 10 });
  },
  error: function(xhr) {
    //jsonp 方式此方法不被触发. 原因可能是 dataType 如果指定为 jsonp 的话, 就已经不是 ajax 事件了
    //请求出错处理
    alert("请求出错(请检查相关度网络状况.)");
  }
});
```

## JavaScript 同源策略

在 JavaScript 中，有一个很重要的安全性限制，被称为“Same-Origin Policy”（同源策略）。所谓同源是指，域名，协议，端口相同

这一策略对于 JavaScript 代码能够访问的页面内容做了很重要的限制，即 JavaScript 只能访问与包含它的文档在同一域下的内容

## 字符串转化

- `parseInt()` `parseFloat()` `return NaN / number`
- `Boolean(value)` `Number(value)` `String(value)`
- `String.toString()`
- 利用 js 弱类型，使用算术运算，实现字符串到数字的类型转换

## XMLHttpRequest

```
xmlhttp=null;
if (window.XMLHttpRequest) {
    // code for all new browsers
    xmlhttp=new XMLHttpRequest();
} else if (window.ActiveXObject) {
    // code for IE5 and IE6
    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
```

## JavaScript 继承实现

### 原型继承

实例既是父类的实例，又是子类的实例，但无法多重继承

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};
function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

`Hoozit.prototype = new Gizmo();` 是原型继承的核心，把父类 `prototype` 赋给子类 `prototype`

### apply call, 重定向 this, 实现继承

可以实现多重继承，但对象不是父类的实例

```
function Animal(name) {
    this.name = name;
    this.showName = function () {
        alert(this.name);
    }
}
function Cat(name) {
    Animal.call(this, name);
}
var cat = new Cat("Black Cat");
cat.showName();
```

### 实例继承

生成的对象实质仅仅是父类的实例，并非子类的对象；返回父类实例的一个扩充，不能多继承

```
var Base = function () {
    this.level = 1;
    this.name = "base";
    this.toString = function () {
        return "base";
    };
};
```

```
Base.CONSTANT = "constant";  
var Sub = function () {  
    var instance = new Base();  
    instance.name = "sub";  
    return instance;  
};
```

## 自己定义继承函数

```
Function.prototype.inherits = function (parent) {  
    //...  
};
```

## this

指向当前对象，使用 `call` `apply` 可以改变指向

## 动画算法

## 拖拽实现

## 熟悉的框架

## 模块化工具的特点

### 模块化工具

`AMD / CMD` `CommonJS` `sea.js` `webpack`

- 上古时期 Module? 从设计模式说起
- 石器时代 Script Loader 只有封装性可不够，我们还需要加载
- 蒸汽朋克 Module Loader 模块化架构的工业革命
- 号角吹响 `CommonJS` 征服世界的第一步是跳出浏览器
- 双塔奇兵 `AMD/CMD` 浏览器环境模块化方案
- 精灵宝钻 `Browserify/Webpack` 大势所趋，去掉这层包裹！
- 王者归来 `ES6` Module 最后的战役（`ECMAScript 6`）
- **模块**是指可组合成系统的、具有某种确定功能和接口结构的典型的通用独立单元
- 通过模块化工具对代码进行组织编排，使各个模块的功能单一独立，实现高内聚，低耦合，明确各个模块之间的依赖关系，便于资源的统一管理
- 便于多人开发时制定统一的标准规范，以便后期的维护测试

## JavaScript 压缩合并

- `gulp` `grunt`
- 批处理的方式 `.bat`  
`copy a.css+b.css final.css /b`

## CSS 模块化

配合 `less / sass / stylus` 实现 `css` 的封装继承多态

## 定位

不要总是用 `float`  
`float` 后注意 `clear`  
`position: relative;` 相对于前一个父级元素，用 `margin` 调整  
`display: inline` / `inline-block`

## 标签比较

`<b>` 与 `<strong>`, `<i>` 与 `<em>`

一个是物理标签, 一个是逻辑标签, 前者强调的是物理行为, 后者强调了当前环境的语义, 更符合标签语义化, 更符合 w3c 标准

`<h1>` 与 `<title>`

SEO 时, `title` 权重高于 `h1`

`<alt>` 与 `<title>`

`alt` 主要是为图像未加载时做的说明 (只用于 `img` `area` `input`)

`title` 用于文字或链接加注释

## 双飞翼布局

双飞翼布局:

- 1、三列布局, 中间宽度自适应, 两边定宽;
- 2、中间栏要在浏览器中优先展示渲染。

```
<style>
    .column { float: left; height: 200px; }
    .header { height: 50px; width: 100%; display: inline-block; background: #000; }
    .footer { height: 100px; width: 100%; display: inline-block; background: #000; }
    #container { width: 100%; }
    #center_div { width: 100%; }
    #center_div #mainWrap { margin-left: 200px; margin-right: 220px; background: #abcdef; height: 100%; }
    #left_div { width: 200px; margin-left: -100%; background: #D1EB2F; }
    #right_div { width: 220px; margin-left: -220px; background: #ccc; }</style><div class="header">header</div><div id="container">
    <div id="center_div" class="column">
        <div id="mainWrap">main</div>
    </div>
    <div id="left_div" class="column">left</div>
    <div id="right_div" class="column">right</div></div><div class="footer">footer</div>
```

## 圣杯布局

圣杯布局:

- 1、三列布局, 中间宽度自适应, 两边定宽;
- 2、中间栏要在浏览器中优先展示渲染。
- 3、与双飞翼布局相比, 使用 `relative` 定位, 但少了一个 `div` 块

```
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1"><style type="text/css">
    .main {
        float: left;
        width: 100%;
        background: #39c;
        height: 300px;
    }
    .sub {
        float: left;
        width: 200px;
        margin-left: -100%;
        background: #f60;
        height: 300px;
```

```
position: relative;
left: -200px;
}
.extra {
float: left;
width: 220px;
margin-left: -230px;
background: #666;
height: 300px;
position: relative;
right: -230px;
}
#bd {
padding: 0 220px 0 200px;
}</style><div id="page">
<div id="hd">header</div>
<div id="bd">
<div class="main">main</div>
<div class="sub">left</div>
<div class="extra">right</div>
</div>
<div id="ft">footer</div> </div>
```

---

## IE 某些兼容性问题

---

### CSS 合并方法

- `grunt` `gulp`
- 批处理的方式 `.bat`  
`copy a.css+b.css final.css /b`

---

### 盒子模型

元素内容 (element content)、内边距 (padding)、边框 (border) 和 外边距 (margin)

---

### CSS 动画原理

---

#### 关于 less / sass / stylus

- 开始的时候, 因为配合 less, 能够很好的进行模块化开发, 并且代码量很少很少, 似乎是弥补了 css 原本的缺陷, 他有变量, 能像函数一样用, 有嵌套, 能够更直观的进行开发并使工作量大大减少, 有一段时间简直爱上了他。不过慢慢的, 开始意识到问题了
  - 因为他给我了一种错觉, 让我过分依赖于 less 了, 他的确很强大, 能够用变量全局控制一些值, 不过在某些情况下效果也不是很好, 虽然你的工作量小了, 但是编译后的 css 代码量并没有减少, 在某些情况下甚至会出现更大的代码冗余, 就好比说, 有些样式可以直接通过一个 class 进行重用的, 但是盲目的使用 less 反而会使代码冗余量更大
  - 归根到底, less css 还是要编译你写的 .less 文件, 最终生成的还是标准的 css 代码。换句话说, 就是你再怎么定义变量, 再怎么计算, 最终它生成的还是一个固定的数值, 帮我们减少的只是我们书写的时间
- 

### HTML5 新特性

- 新的文档类型: `<!DOCTYPE html>`
- 脚本和链接无需指定 `type` 属性



- 语义 Header 和 Footer

```
<header>
....
</header>
<footer>
....
</footer>
```

- Email Inputs

如果我们给 Input 的 `type` 设置为 `email`，浏览器就会验证这个输入是否是 email 类型，当然不能只依赖前端的校验，后端也得有相应的校验

- 内容可编辑，只需要加一个 `contenteditable` 属性
- 重新定义的 `<small>`

`<small>` 已经被重新定义了，现在被用来表示小的排版，如网站底部的版权声明

- Placeholders 这个 input 属性的意义就是不必通过 js 来做 `placeholder` 的效果

```
<input type="search" name="search" placeholder="please entry the keywords" />
```

- IE 和 HTML5 默认的，HTML5 新元素被以 `inline` 的方式渲染，但 IE 会忽略这些样式
- Local Storage 使用 Local Storage 可以永久存储大的数据片段在客户端（除非主动删除），目前大部分浏览器已经支持，在使用之前可以检测一下 `window.localStorage` 是否存在
- `required` 属性 `required` 属性定义了一个 input 是否是必须填写的，你可以像下面这样声明 `<input type="text" name="someInput" required>`  
或者  
`<input type="text" name="someInput" required="required">`
- `autofocus` 属性 正如它的词义，就是聚焦到输入框里面 `<input type="text" name="someInput" placeholder="focus" required autofocus>`
- `audio` 支持
- `video` 支持
- 预加载视频 `preload` 属性就像它的字面意思那么简单，你需要决定是否需要在页面加载的时候去预加载视频 `<video preload>`
- 显示视频控制 `<video preload controls>`
- 正则表达式 由于 `pattern` 属性，我们可以在你的 markup 里面直接使用正则表达式了

```
<input type="text" name="username" id="username" placeholder="" pattern="[A-Za-z]{4,10}" autofocus required>
```

- 检测属性支持 除了 Modernizr 之外我们还可以通过 javascript 简单地检测一些属性是否支持，如：

```
<script>
if (!'pattern' in document.createElement('input')) {
  // do client/server side validation
}
</script>
```

- Mark 元素 把 `<mark>` 元素看做是高亮的作用
- Output 元素 `<output>` 元素用来显示计算结果，也有一个 `for` 属性

```
<input type="range" id="a" value="50">100<input type="number" id="b" value="50">=<output name="x" for="a b"></output>
```

- 扩充了 `input` 的 `type` 属性
- 重新定义 `<small>` 在 HTML4 或 XHTML 中，`<small>` 元素已经存在。然而，却没有如何正确使用这一元素的完整说明。在 HTML5 中，`<small>` 被用来定义小字。试想下你网站底部的版权状态，根据对此元素新的 HTML5 定义，`<small>` 可以正确地诠释这些信息。
- 什么时候用 `<div>` HTML5 已经引入了这么多元素，那么 `div` 我们还要用吗？`div` 你可以在没有更好的元素的时候去用。

## HTTP 状态码

- 1XX —— 消息报文
  - 2XX —— 成功
  - 3XX —— 重定向
  - 4XX —— 请求错误
  - 5XX / 6XX —— 服务器错误
- 

## Cache-control

缓存控制

---

## 页面加载过程

### 浏览器渲染

加载：

- 加载过程中遇到外部 css 文件，浏览器另外发出一个请求，来获取 css 文件。遇到图片资源，浏览器也会另外发出一个请求，来获取图片资源。这是异步请求，并不会影响 html 文档进行加载
- 内部 `<style></style>` 这种样式定义，也可能会阻塞渲染
- 但是当文档加载过程中遇到 js 文件，html 文档会挂起渲染（加载解析渲染同步）的线程，不仅要等待文档中 js 文件加载完毕，还要等待解析执行完毕，才可以恢复 html 文档的渲染线程。 原因：JS 有可能会修改 DOM

解析：

- html 文档解析生成解析树即 dom 树，是由 dom 元素及属性节点组成，树的根是 document 对象，（DOM 文档对象模型）
- css 解析将 css 文件解析为样式表对象。该对象包含 css 规则，该规则包含选择器和声明对象
- js 解析因为文件在加载的同时也进行解析，加载时解析

渲染：

- 为每一个元素查找到匹配的样式规则，需要遍历整个规则表（解析后生成的最终 CSS 样式表）
- 

## javascript-event-loop 事件循环

由于 javascript 引擎线程为单线程，所以代码都是先压到队列，采用先进先出的方式运行，事件处理函数，timer 函数也会压在队列中，不断的从队头取出事件，这就叫：`javascript-event-loop`

---

## 前后端协同开发

- 确定功能，根据功能制定前后端接口
  - 前后端分别进行开发，并且分别进行测试
  - 前端先使用模拟数据
  - 前后端完成后，进行连接调试
- 

## session

---

## 基础算法