完整面试题地址：https://interview.poetries.top

作者：程序员poetry

扫码关注作者公众号：「前端进价之旅」 每天分享技术干货

# 1 如何理解HTML语义化

- 用正确的标签做正确的事情!
- `HTML` 语义化就是让页面的内容结构化，便于对浏览器、搜索引擎解析;
- 在没有样式 `CSS` 情况下也以一种文档格式显示，并且是容易阅读的。
- 搜索引擎的爬虫依赖于标记来确定上下文和各个关键字的权重，利于 `SEO` 。
- 使阅读源代码的人对网站更容易将网站分块，便于阅读维护理解

# 2 H5的新特性有哪些

- 画布( `Canvas` ) API
- 地理( `Geolocation` ) API
- 音频、视频API( `audio` , `video` )
- `localStorage` 和 `sessionStorage`
- `webworker` , `websocket`
- 新的一套标签 `header` , `nav` , `footer` , `aside` , `article` , `section`
- `web worker` 是运行在浏览器后台的js程序，他不影响主程序的运行，是另开的一个js线程，可以用这个线程执行复杂的数据操作，然后把操作结果通过 `postMessage` 传递给主线程，这样在进行复杂且耗时的操作时就不会阻塞主线程了
- `HTML5 History` 两个新增的API： `history.pushState` 和 `history.replaceState` ，两个 `API` 都会操作浏览器的历史记录，而不会引起页面的刷新

> `Hash` 就是 `url` 中看到 `#` ,我们需要一个根据监听哈希变化触发的事件( `hashchange` ) 事件。我们用 `window.location` 处理哈希的改变时不会重新渲染页面，而是当作新页面加到历史记录中，这样我们跳转页面就可以在 `hashchange` 事件中注册 `ajax` 从而改变页面内容。 可以为 `hash` 的改变添加监听事件:

```js
window.addEventListener("hashchange", funcRef, false)
```

- `WebSocket` 使用 `ws` 或 `wss` 协议， `Websocket` 是一个持久化的协议，相对于 `HTTP` 这种非持久的协议来说。 `WebSocket API` 最伟大之处在于服务器和客户端可以在给定的时间范围内的任意时刻，相互推送信息。 `WebSocket` 并不限于以 `Ajax` (或 `XHR` )方式通信，因为 `Ajax` 技术需要客户端发起请求，而 `WebSocket` 服务器和客户端可以彼此相互推送信息；`XHR` 受到域的限制，而 `WebSocket` 允许跨域通信

```js
// 创建一个Socket实例
var socket = new WebSocket('ws://localhost:8080');
// 打开Socket
socket.onopen = function(event) {
  // 发送一个初始化消息
  socket.send('I am the client and I\'m listening!');
  // 监听消息
  socket.onmessage = function(event) {
    console.log('Client received a message',event);
  };
  // 监听Socket的关闭
  socket.onclose = function(event) {
    console.log('Client notified socket has closed',event);
  };
  // 关闭Socket....
  //socket.close()
};
```

## # 3 说一下 HTML5 drag api

- `dragstart` :事件主体是被拖放元素，在开始拖放被拖放元素时触发，。
- `darg` :事件主体是被拖放元素，在正在拖放被拖放元素时触发。
- `dragenter` :事件主体是目标元素，在被拖放元素进入某元素时触发。
- `dragover` :事件主体是目标元素，在被拖放在某元素内移动时触发。
- `dragleave` :事件主体是目标元素，在被拖放元素移出目标元素是触发。
- `drop` :事件主体是目标元素，在目标元素完全接受被拖放元素时触发。
- `dragend` :事件主体是被拖放元素，在整个拖放操作结束时触发

# 4 iframe有那些缺点

- `iframe` 会阻塞主页面的 `Onload` 事件；
- 搜索引擎的检索程序无法解读这种页面，不利于 `SEO` ；
- `iframe` 和主页面共享连接池，而浏览器对相同域的连接有限制，所以会影响页面的并行加载。
- 使用 `iframe` 之前需要考虑这两个缺点。如果需要使用 `iframe` ，最好是通过 `javascript`
- 动态给 `iframe` 添加 `src` 属性值，这样可以绕开以上两个问题

# 5 如何实现浏览器内多个标签页之间的通信

1. 使用 `WebSocket` 可以实现多个标签页之间的通信
2. 调用 `localStorage`

- 在一个标签页里面使用 `localStorage.setItem(key,value)` 添加（修改、删除）内容；
- 在另一个标签页里面监听 `storage` 事件。
- 即可得到 `localstorge` 存储的值，实现不同标签页之间的通信

标签页1

```html
<input id="name">
<input type="button" id="btn" value="提交">
<script type="text/javascript">
    $(function(){
        $("#btn").click(function(){
            var name=$("#name").val();
            localStorage.setItem("name", name);
        });
    });
</script>
```

标签页2：

```html
<script type="text/javascript">
    $(function(){
        window.addEventListener("storage", function(event){
            console.log(event.key + "=" + event.newValue);
        });
    });
</script>
```

3. 调用 `cookie+setInterval()`

> 将要传递的信息存储在 `cookie` 中，每隔一定时间读取 `cookie` 信息，即可随时获取要传递的信息。

页面1：

```html
<input id="name">
<input type="button" id="btn" value="提交">
<script type="text/javascript">
    $(function(){
        $("#btn").click(function(){
            var name=$("#name").val();
            document.cookie="name="+name;
        });
    });
</script>
```

页面2：

```html
<script type="text/javascript">
    $(function(){
        function getCookie(key) {
            return JSON.parse("{\"" + document.cookie.replace(/;\s+/gi
m,"\",\"").replace(/=/gim, "\":\"") + "\"}")[key];
        }
        setInterval(function(){
            console.log("name=" + getCookie("name"));
        }, 10000);
```

```
    });
</script>
```

## # 6 简述一下src与href的区别

- `src` 用于替换当前元素，`href` 用于在当前文档和引用资源之间确立联系。
- `src` 是 `source` 的缩写，指向外部资源的位置，指向的内容将会嵌入到文档中当前标签所在位置；在请求 `src` 资源时会将其指向的资源下载并应用到文档内，例如 `js` 脚本，`img` 图片和 `frame` 等元素

```
<script src =”js.js”></script>
```

> 当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等元素也如此，类似于将所指向资源嵌入当前标签内。这也是为什么将 `js` 脚本放在底部而不是头部

- `href` 是 `Hypertext Reference` 的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加

```
<link href=”common.css” rel=”stylesheet”/>
```

> 那么浏览器会识别该文档为css文件，就会并行下载资源并且不会停止对当前文档的处理。这也是为什么建议使用 `link` 方式来加载css，而不是使用 `@import` 方式。

## # 7 知道的网页制作会用到的图片格式有哪些

> `png-8` , `png-24` , `jpeg` , `gif` , `svg`

但是上面的那些都不是面试官想要的最后答案。面试官希望听到是 `Webp` 。（是否有关注新技术，新鲜事物）

> 科普一下Webp：WebP格式，谷歌（google）开发的一种旨在加快图片加载速度的图片格式。图片压缩体积大约只有JPEG的2/3，并能节省大量的服务器带宽资源和数据空间。Facebook Ebay等知名网站已经开始测试并使用WebP格式。

在质量相同的情况下，`WebP`格式图像的体积要比JPEG格式图像小`40%`

## # 8 script标签中defer和async的区别

- `script`：会阻碍`HTML`解析，只有下载好并执行完脚本才会继续解析`HTML`。
- `defer`:浏览器指示脚本在文档被解析后执行，`script`被异步加载后并不会立刻执行，而是等待文档被解析完毕后执行。
- `async`:同样是异步加载脚本，区别是脚本加载完毕后立即执行，这导致`async`属性下的脚本是乱序的，对于`script`有先后依赖关系的情况，并不适用

> 蓝色线代表网络读取，红色线代表执行时间，这俩都是针对脚本的;绿色线代表 HTML 解析

## # 9 说一下 web worker

> 在 HTML 页面中，如果在执行脚本时，页面的状态是不可响应式的，直到脚本执行完成后， 页面才变成可响应。`web worker`是运行在后台的`js`，独立于其他脚本，不会影响页面你 的性能。并且通过`postMessage`将结果回传到主线程。这样在进行复杂操作的时候，就 不会阻塞主线程了

**如何创建 web worker:**

- 检测浏览器对于`web worker`的支持性
- 创建`web worker`文件(js，回传函数等)
- 创建`web worker`对象

## # 10 用一个div模拟textarea的实现

> 给 `div` 添加 `contenteditable=true` 即可

## # 11 介绍下资源预加载 prefetch/preload

> 都是告知浏览器提前加载文件(图片、视频、 `js` 、 `css` 等)，但执行上是有区别的。

- `prefetch` ：其利用浏览器空闲时间来下载或预取用户在不久的将来可能访问的文档。 `<link href="/js/xx.js" rel="prefetch">`
- `preload` ：可以指明哪些资源是在页面加载完成后即刻需要的，浏览器在主渲染机制介入前就进行预加载，这一机制使得资源可以更早的得到加载并可用，且更不易阻塞页面的初步渲染，进而提升性能。 `<link href="/js/xxx.js" rel="preload" as="script">` 需要 `as` 指定资源类型**目前可用的属性类型有如下**：

```
audio: 音频文件。
document: 一个将要被嵌入到<frame>或<iframe>内部的HTML文档。
embed: 一个将要被嵌入到<embed>元素内部的资源。
fetch: 那些将要通过fetch和XHR请求来获取的资源，比如一个ArrayBuffer或JSON文件。
font: 字体文件。
image: 图片文件。
object: 一个将会被嵌入到<embed>元素内的文件。
script: JavaScript文件。
style: 样式表。
track: WebVTT文件。
worker: 一个JavaScript的web worker或shared worker。
video: 视频文件。
```

## # 12 介绍下 viewport

```html
<meta name="viewport" content="width=500, initial-scale=1">
```

- `width` ：页面宽度，可以取值具体的数字，也可以是 `device-width` ，表示跟设备宽度相等。

- `height`：页面高度，可以取值具体的数字，也可以是 `device-height`，表示跟设备高度相等。
- `initial-scale`：初始缩放比例。
- `minimum-scale`：最小缩放比例。
- `maximum-scale`：最大缩放比例。
- `user-scalable`：是否允许用户缩放。

## # 13 如何解决a标点击后hover事件失效的问题?

改变 `a` 标签 `css` 属性的排列顺序

只需要记住 `LoVe HAte` 原则就可以了(爱恨原则)：

```
link → visited → hover → active
```

比如下面错误的代码顺序：

```css
a:hover{
  color: green;
  text-decoration: none;
}
a:visited{ /* visited在hover后面，这样的话hover事件就失效了 */
  color: red;
  text-decoration: none;
}
```

正确的做法是将两个事件的位置调整一下。

注意⚠各个阶段的含义：

- `a:link`：未访问时的样式，一般省略成 `a`
- `a:visited`：已经访问后的样式
- `a:hover`：鼠标移上去时的样式
- `a:active`：鼠标按下时的样式

# 14 点击一个input依次触发的事件

```js
const text = document.getElementById('text');
text.onclick = function (e) {
  console.log('onclick')
}
text.onfocus = function (e) {
  console.log('onfocus')
}
text.onmousedown = function (e) {
  console.log('onmousedown')
}
text.onmouseenter = function (e) {
  console.log('onmouseenter')
}
```

答案：

```js
'onmouseenter'
'onmousedown'
'onfocus'
'onclick'
```

# 15 有写过原生的自定义事件吗

**创建自定义事件**

原生自定义事件有三种写法：

1. 使用 `Event`

```js
let myEvent = new Event('event_name');
```

2. 使用 `customEvent` （可以传参数）

```js
let myEvent = new CustomEvent('event_name', {
    detail: {
        // 将需要传递的参数放到这里
        // 可以在监听的回调函数中获取到：event.detail
    }
})
```

3. 使用 `document.createEvent('CustomEvent')和initCustomEvent()`

```js
let myEvent = document.createEvent('CustomEvent');// 注意这里是为'CustomEvent'
myEvent.initEvent(
    // 1. event_name: 事件名称
    // 2. canBubble: 是否冒泡
    // 3. cancelable: 是否可以取消默认行为
)
```

- `createEvent`：创建一个事件
- `initEvent`：初始化一个事件

可以看到，`initEvent` 可以指定3个参数。

（有些文章中会说还有第四个参数 `detail`，但是我查看了 `W3C` 上并没有这个参数，而且实践了一下也没有效果）

**事件的监听**

自定义事件的监听其实和普通事件的一样，使用 `addEventListener` 来监听：

```js
button.addEventListener('event_name', function (e) {})
```

**事件的触发**

触发自定义事件使用 `dispatchEvent(myEvent)` 。

注意⚠️，这里的参数是要自定义事件的对象(也就是 `myEvent` )，而不是自定义事件的名称( `'myEvent'` )

**案例**

来看个案例吧：

```js
// 1.
// let myEvent = new Event('myEvent');
// 2.
// let myEvent = new CustomEvent('myEvent', {
//   detail: {
//     name: 'lindaidai'
//   }
// })
// 3.
let myEvent = document.createEvent('CustomEvent');
myEvent.initEvent('myEvent', true, true)

let btn = document.getElementsByTagName('button')[0]
btn.addEventListener('myEvent', function (e) {
  console.log(e)
  console.log(e.detail)
})
setTimeout(() => {
  btn.dispatchEvent(myEvent)
}, 2000)
```

# # 16 addEventListener和attachEvent的区别?

- 前者是标准浏览器中的用法，后者 `IE8` 以下
- `addEventListener` 可有冒泡，可有捕获； `attachEvent` 只有冒泡，没有捕获。
- 前者事件名不带 `on` ，后者带 `on`
- 前者回调函数中的 `this` 指向当前元素，后者指向 `window`

# 17 addEventListener函数的第三个参数

第三个参数涉及到冒泡和捕获，是 `true` 时为捕获，是 `false` 则为冒泡。

> 或者是一个对象 `{passive: true}` ，针对的是 `Safari` 浏览器，禁止/开启使用滚动的时候要用到。

# 18 DOM事件流是什么?

事件发生时会在元素节点之间按照**特定的顺序**传播，这个传播过程就叫做DOM事件流。

> DOM事件流分为三个阶段：

1. **捕获阶段**：事件从 `window` 发出，自上而下向目标节点传播的阶段
2. **目标阶段**：真正的目标阶段正在处理事件的阶段
3. **冒泡阶段**：事件从目标节点自下而上向 `window` 传播的阶段

(注意⚠：`JS` 代码只能执行捕获或者冒泡其中一个阶段，要么是捕获要么是冒泡)

# 19 冒泡和捕获的具体过程

> 冒泡指的是：当给某个目标元素绑定了事件之后，这个事件会依次在它的父级元素中被触发(当然前提是这个父级元素也有这个同名称的事件，比如子元素和父元素都绑定了 `click` 事件就触发父元素的 `click` )。

捕获则是从上层向下层传递，与冒泡相反。

（非常好记，你就想想水底有一个泡泡从下面往上传的，所以是冒泡）

来看看这个例子：

```html
<!-- 会依次执行 button li ul -->
<ul onclick="alert('ul')">
  <li onclick="alert('li')">
    <button onclick="alert('button')">点击</button>
```

```
    </li>
  </ul>
  <script>
    window.addEventListener('click', function (e) {
      alert('window')
    },false)
    document.addEventListener('click', function (e) {
      alert('document')
    },true)
  </script>
```

- 冒泡结果：`button > li > ul > document > window`
- 捕获结果：`window > document > ul > li > button`

## # 20 关于一些兼容性

1. `event`的兼容性

- 其它浏览器`window.event`
- 火狐下没有`window.event`，所以用传入的参数`ev`代替
- 最终写法：`var oEvent = ev || window.event`

2. 事件源的兼容性

- 其它浏览器`event.target`
- `IE`下为`event.srcElement`
- 最终写法：`var target = event.target || event.srcElement`

3. 阻止事件冒泡

- 其它浏览器`event.stopPropagation()`
- `IE`下为`window.event.cancelBubble = true`

4. 阻止默认事件

- 其它浏览器`e.preventDefault()`
- `IE`下为`window.event.returnValue = false`

# 21 如何阻止冒泡和默认事件(兼容写法)

阻止冒泡：

```js
function stopBubble (e) { // 阻止冒泡
  if (e && e.stopPropagation) {
    e.stopPropagation();
  } else {
    // 兼容 IE
    window.event.cancelBubble = true;
  }
}
function stopDefault (e) { // 阻止默认事件
  if (e && e.preventDefault) {
    e.preventDefault();
  } else {
    // 兼容 IE
    window.event.returnValue = false;
    return false;
  }
}
```

# 22 所有的事件都有冒泡吗?

并不是所有的事件都有冒泡的，例如以下事件就没有：

- `onblur`
- `onfocus`
- `onmouseenter`
- `onmouseleave`

# 23 拖拽有哪些知识点

1. 可以通过给标签设置 `draggable` 属性来实现元素的拖拽， `img和a标签` 默认是可以拖拽的

2. 拖拽者身上的三个事件：`ondragstart`、`ondrag`、`ondragend`
3. 拖拽要放到的元素：`ondragenter`、`ondragover`、`ondragleave`、`ondrap`

## # 24 offset、scroll、client的区别

client:

- `oEvent.clientX` 是指鼠标到可视区左边框的距离。
- `oEvent.clientY` 是指鼠标到可视区上边框的距离。
- `clientWidth` 是指可视区的宽度。
- `clientHeight` 是指可视区的高度。
- `clientLeft` 获取左边框的宽度。
- `clientTop` 获取上边框的宽度。

offset:

- `offsetWidth` 是指div的宽度（包括div的边框）
- `offsetHeight` 是指div的高度（包括div的边框）
- `offsetLeft` 是指div到整个页面左边框的距离（不包括div的边框）
- `offsetTop` 是指div到整个页面上边框的距离（不包括div的边框）

scroll:

- `scrollTop` 是指可视区顶部边框与整个页面上部边框的看不到的区域。
- `scrollLeft` 是指可视区左边边框与整个页面左边边框的看不到的区域。
- `scrollWidth` 是指左边看不到的区域加可视区加右边看不到的区域即整个页面的宽度（包括边框）
- `scrollHeight` 是指上边看不到的区域加可视区加右边看不到的区域即整个页面的高度（包括边框）

## # 25 target="_blank"有哪些问题?

**存在问题：**

1. 安全隐患：新打开的窗口可以通过 `window.opener` 获取到来源页面的 `window` 对象即使跨域也可以。某些属性的访问被拦截，是因为跨域安全策略的限制。 但是，比如修改 `window.opener.location` 的值，指向另外一个地址，这样新窗口有可能会把原来的网页地址改了并进行页面伪装来欺骗用户。
2. 新打开的窗口与原页面窗口共用一个进程，若是新页面有性能不好的代码也会影响原页面

**解决方案：**

1. 尽量不用 `target="_blank"`
2. 如果一定要用，需要加上 `rel="noopener"` 或者 `rel="noreferrer"` 。这样新窗口的 `window.openner` 就是 `null` 了，而且会让新窗口运行在独立的进程里，不会拖累原来页面的进程。(不过，有些浏览器对性能做了优化，即使不加这个属性，新窗口也会在独立进程打开。不过为了安全考虑，还是加上吧。)

# # 26 children以及childNodes的区别

- `children` 和只获取该节点下的所有 `element` 节点
- `childNodes` 不仅仅获取 `element` 节点还会获取元素标签中的空白节点
- `firstElementChild` 只获取该节点下的第一个 `element` 节点
- `firstChild` 会获取空白节点

# # 27 HTMLCollection和NodeList的区别

### Node和Element

- `DOM` 是一棵树，所有节点都是 `Node`
- `Node` 是 `Element` 的基类
- `Element` 是其他HTML元素的基类，如 `HTMLDivElement` 、 `HTMLImageElement` 等

- `HTMLCollection` 是 `Element` 的集合
- `NodeList` 是 `Node` 的集合，包含 `Text` 和 `Comment` 节点
- `ele.children` 返回 `HTMLCollection` 集合
- `ele.childNodes` 返回 `NodeList` 集合

- `HTMLCollection` 和 `NodeList` 是类数组
  - 使用 `Array.from(list)` 转化数组
  - 使用 `Array.prototype.slice.call(list)` 转化数组
  - 使用 `[...list]` 转化数组

```html
<p id="p1"><b>node</b> vs <em>element</em><!--注释--></p>

<script>
  const p1 = document.getElementById('p1')
  // console.log(p1.children) // HTMLCollection
  console.log(p1.childNodes) // NodeList

  // p1.tagName // Element类型一定有tagName
  // p1.nodeType/nodeName // node节点

  class Node {}

  // document
  class Document extends Node {}
  class DocumentFragment extends Node {}

  // 文本和注释
  class CharacterData extends Node {}
  class Comment extends CharacterData {}
  class Text extends CharacterData {}

  // elem
  class Element extends Node {}
  class HTMLElement extends Element {}
  class HTMLDivElement extends HTMLElement {}
  class HTMLInputElement extends HTMLElement {}
  // ...
</script>
```