**最全面的前端开发指南**

[前端开发](http://www.imooc.com/article/index/cid/105) 2015-12-16 11:27:21 3603浏览

来源：<http://www.imooc.com/article/2835>

# HTML

## 语义

HTML5为我们提供了很多旨在精确描述内容的语义元素。确保你可以从它丰富的词汇中获益。

<!-- bad -->

<div id="main">

<div class="article">

<div class="header">

<h1>Blog post</h1>

<p>Published: <span>21st Feb, 2015</span></p>

</div>

<p>…</p>

</div>

</div>

<!-- good -->

<main>

<article>

<header>

<h1>Blog post</h1>

<p>Published: <time datetime="2015-02-21">21st Feb, 2015</time></p>

</header>

<p>…</p>

</article>

</main>

你需要理解你正在使用的元素的语义。用一种错误的方式使用语义元素比保持中立更糟糕。

<!-- bad -->

<h1>

<figure>

<img alt=Company src=logo.png>

</figure>

</h1>

<!-- good -->

<h1>

<img alt=Company src=logo.png>

</h1>

## 简洁

保持代码的简洁。忘记原来的XHTML习惯。

<!-- bad -->

<!doctype html>

<html lang=en>

<head>

<meta http-equiv=Content-Type content="text/html; charset=utf-8" />

<title>Contact</title>

<link rel=stylesheet href=style.css type=text/css />

</head>

<body>

<h1>Contact me</h1>

<label>

Email address:

<input type=email placeholder=you@email.com required=required />

</label>

<script src=main.js type=text/javascript></script>

</body>

</html>

<!-- good -->

<!doctype html>

<html lang=en>

<meta charset=utf-8>

<title>Contact</title>

<link rel=stylesheet href=style.css>

<h1>Contact me</h1>

<label>

Email address:

<input type=email placeholder=you@email.com required>

</label>

<script src=main.js></script>

</html>

## 可访问性

可访问性不应该是以后再想的事情。提高网站不需要你成为一个WCAG专家，你完全可以通过修复一些小问题，从而造成一个巨大的变化，例如：

* 学习正确使用alt 属性
* 确保链接和按钮被同样地标记（不允许<div>）
* 不专门依靠颜色来传递信息
* 明确标注表单控件

<!-- bad -->

<h1><img alt="Logo" src="logo.png"></h1>

<!-- good -->

<h1><img alt="My Company, Inc." src="logo.png"></h1>

## 语言

当定义语言和字符编码是可选择的时候，总是建议在文档级别同时声明，即使它们在你的HTTP标头已经详细说明。比任何其他字符编码更偏爱UTF-8。

<!-- bad -->

<!doctype html>

<title>Hello, world.</title>

<!-- good -->

<!doctype html>

<html lang=en>

<meta charset=utf-8>

<title>Hello, world.</title>

</html>

## 性能

除非有正当理由才能在内容前加载脚本，不要阻塞页面的渲染。如果你的样式表很重，开头就孤立那些绝对需要得样式，并在一个单独的样式表中推迟二次声明的加载。两个HTTP请求显然比一个慢，但是感知速度是最重要的因素。

<!-- bad -->

<!doctype html>

<meta charset=utf-8>

<script src=analytics.js></script>

<title>Hello, world.</title>

<p>...</p>

<!-- good -->

<!doctype html>

<meta charset=utf-8>

<title>Hello, world.</title>

<p>...</p>

<script src=analytics.js></script>

# CSS

## 分号

虽然分号在技术上是CSS一个分隔符，但应该始终把它作为一个终止符。

/\* bad \*/

div {

color: red

}

/\* good \*/

div {

color: red;

}

## 盒子模型

盒子模型对于整个文档而言最好是相同的。全局性的\* { box-sizing: border-box; }就非常不错，但是不要改变默认盒子模型的特定元素，如果可以避免的话。

/\* bad \*/

div {

width: 100%;

padding: 10px;

box-sizing: border-box;

}

/\* good \*/

div {

padding: 10px;

}

## 流

不要更改元素的默认行为，如果可以避免的话。元素尽可能地保持在自然的文档流中。例如，删除图像下方的空格而不改变其默认显示：

/\* bad \*/

img {

display: block;

}

/\* good \*/

img {

vertical-align: middle;

}

同样，如果可以避免的话，不要关闭元素流。

/\* bad \*/

div {

width: 100px;

position: absolute;

right: 0;

}

/\* good \*/

div {

width: 100px;

margin-left: auto;

}

## 定位

在CSS中有许多定位元素的方法，但应该尽量限制以下属性/值。按优先顺序排列：

display: block;

display: flex;

position: relative;

position: sticky;

position: absolute;

position: fixed;

## 选择器

最小化紧密耦合到DOM的选择器。当选择器有多于3个结构伪类，后代或兄弟选择器的时候，考虑添加一个类到你想匹配的元素。

/\* bad \*/

div:first-of-type :last-child > p ~ \*

/\* good \*/

div:first-of-type .info

当你不需要的时候避免过载选择器。

/\* bad \*/

img[src$=svg], ul > li:first-child {

opacity: 0;

}

/\* good \*/

[src$=svg], ul > :first-child {

opacity: 0;

}

## 特异性

不要让值和选择器难以覆盖。尽量少用id，并避免!important。

/\* bad \*/

.bar {

color: green !important;

}

.foo {

color: red;

}

/\* good \*/

.foo.bar {

color: green;

}

.foo {

color: red;

}

## 覆盖

覆盖样式使得选择器和调试变得困难。如果可能的话，避免覆盖样式。

/\* bad \*/

li {

visibility: hidden;

}

li:first-child {

visibility: visible;

}

/\* good \*/

li + li {

visibility: hidden;

}

## 继承

不要重复可以继承的样式声明。

/\* bad \*/

div h1, div p {

text-shadow: 0 1px 0 #fff;

}

/\* good \*/

div {

text-shadow: 0 1px 0 #fff;

}

## 简洁

保持代码的简洁。使用简写属性，没有必要的话，要避免使用多个属性。

/\* bad \*/

div {

transition: all 1s;

top: 50%;

margin-top: -10px;

padding-top: 5px;

padding-right: 10px;

padding-bottom: 20px;

padding-left: 10px;

}

/\* good \*/

div {

transition: 1s;

top: calc(50% - 10px);

padding: 5px 10px 20px;

}

## 语言

英语表达优于数学公式。

/\* bad \*/

:nth-child(2n + 1) {

transform: rotate(360deg);

}

/\* good \*/

:nth-child(odd) {

transform: rotate(1turn);

}

## 浏览器引擎前缀

果断地删除过时的浏览器引擎前缀。如果需要使用的话，可以在标准属性前插入它们。

/\* bad \*/

div {

transform: scale(2);

-webkit-transform: scale(2);

-moz-transform: scale(2);

-ms-transform: scale(2);

transition: 1s;

-webkit-transition: 1s;

-moz-transition: 1s;

-ms-transition: 1s;

}

/\* good \*/

div {

-webkit-transform: scale(2);

transform: scale(2);

transition: 1s;

}

## 动画

视图转换优于动画。除了opacity 和transform，避免动画其他属性。

/\* bad \*/

div:hover {

animation: move 1s forwards;

}

@keyframes move {

100% {

margin-left: 100px;

}

}

/\* good \*/

div:hover {

transition: 1s;

transform: translateX(100px);

}

## 单位

可以的话，使用无单位的值。如果使用相对单位，那就用rem 。秒优于毫秒。

/\* bad \*/

div {

margin: 0px;

font-size: .9em;

line-height: 22px;

transition: 500ms;

}

/\* good \*/

div {

margin: 0;

font-size: .9rem;

line-height: 1.5;

transition: .5s;

}

## 颜色

如果你需要透明度，使用rgba。另外，始终使用十六进制格式。

/\* bad \*/

div {

color: hsl(103, 54%, 43%);

}

/\* good \*/

div {

color: #5a3;

}

## 绘画

当资源很容易用CSS复制的时候，避免HTTP请求。

/\* bad \*/

div::before {

content: url(white-circle.svg);

}

/\* good \*/

div::before {

content: "";

display: block;

width: 20px;

height: 20px;

border-radius: 50%;

background: #fff;

}

## Hacks

不要使用Hacks。

/\* bad \*/

div {

// position: relative;

transform: translateZ(0);

}

/\* good \*/

div {

/\* position: relative; \*/

will-change: transform;

}

# JavaScript

## 性能

可读性，正确性和可表达性优于性能。JavaScript基本上永远不会是你的性能瓶颈。图像压缩，网络接入和DOM重排来代替优化。如果从本文中你只能记住一个指导原则，那么毫无疑问就是这一条。

// bad (albeit way faster)

const arr = [1, 2, 3, 4];

const len = arr.length;

var i = -1;

var result = [];

while (++i < len) {

var n = arr[i];

if (n % 2 > 0) continue;

result.push(n \* n);

}

// good

const arr = [1, 2, 3, 4];

const isEven = n => n % 2 == 0;

const square = n => n \* n;

const result = arr.filter(isEven).map(square);

## 无状态

尽量保持函数纯洁。理论上，所有函数都不会产生副作用，不会使用外部数据，并且会返回新对象，而不是改变现有的对象。

// bad

const merge = (target, ...sources) => Object.assign(target, ...sources);

merge({ foo: "foo" }, { bar: "bar" }); // => { foo: "foo", bar: "bar" }

// good

const merge = (...sources) => Object.assign({}, ...sources);

merge({ foo: "foo" }, { bar: "bar" }); // => { foo: "foo", bar: "bar" }

## 本地化

尽可能地依赖本地方法。

// bad

const toArray = obj => [].slice.call(obj);

// good

const toArray = (() =>

Array.from ? Array.from : obj => [].slice.call(obj)

)();

## 强制性

如果强制有意义，那么就使用隐式强制。否则就应该避免强制。

// bad

if (x === undefined || x === null) { ... }

// good

if (x == undefined) { ... }

## 循环

不要使用循环，因为它们会强迫你使用可变对象。依靠array.prototype 方法。

// bad

const sum = arr => {

var sum = 0;

var i = -1;

for (;arr[++i];) {

sum += arr[i];

}

return sum;

};

sum([1, 2, 3]); // => 6

// good

const sum = arr =>

arr.reduce((x, y) => x + y);

sum([1, 2, 3]); // => 6

如果不能避免，或使用array.prototype方法滥用了，那就使用递归。

// bad

const createDivs = howMany => {

while (howMany--) {

document.body.insertAdjacentHTML("beforeend", "<div></div>");

}

};

createDivs(5);

// bad

const createDivs = howMany =>

[...Array(howMany)].forEach(() =>

document.body.insertAdjacentHTML("beforeend", "<div></div>")

);

createDivs(5);

// good

const createDivs = howMany => {

if (!howMany) return;

document.body.insertAdjacentHTML("beforeend", "<div></div>");

return createDivs(howMany - 1);

};

createDivs(5);

这里有一个通用的循环功能，可以让递归更容易使用。

## 参数

忘记arguments 对象。余下的参数往往是一个更好的选择，这是因为：

你可以从它的命名中更好地了解函数需要什么样的参数

真实数组，更易于使用。

// bad

const sortNumbers = () =>

Array.prototype.slice.call(arguments).sort();

// good

const sortNumbers = (...numbers) => numbers.sort();

## 应用

忘掉apply()。使用操作符。

const greet = (first, last) => `Hi ${first} ${last}`;

const person = ["John", "Doe"];

// bad

greet.apply(null, person);

// good

greet(...person);

## 绑定

当有更惯用的做法时，就不要用bind() 。

// bad

["foo", "bar"].forEach(func.bind(this));

// good

["foo", "bar"].forEach(func, this);

// bad

const person = {

first: "John",

last: "Doe",

greet() {

const full = function() {

return `${this.first} ${this.last}`;

}.bind(this);

return `Hello ${full()}`;

}

}

// good

const person = {

first: "John",

last: "Doe",

greet() {

const full = () => `${this.first} ${this.last}`;

return `Hello ${full()}`;

}

}

## 函数嵌套

没有必要的话，就不要嵌套函数。

// bad

[1, 2, 3].map(num => String(num));

// good

[1, 2, 3].map(String);

## 合成函数

避免调用多重嵌套函数。使用合成函数来替代。

const plus1 = a => a + 1;

const mult2 = a => a \* 2;

// bad

mult2(plus1(5)); // => 12

// good

const pipeline = (...funcs) => val => funcs.reduce((a, b) => b(a), val);

const addThenMult = pipeline(plus1, mult2);

addThenMult(5); // => 12

## 缓存

缓存功能测试，大数据结构和任何奢侈的操作。

// bad

const contains = (arr, value) =>

Array.prototype.includes

? arr.includes(value)

: arr.some(el => el === value);

contains(["foo", "bar"], "baz"); // => false

// good

const contains = (() =>

Array.prototype.includes

? (arr, value) => arr.includes(value)

: (arr, value) => arr.some(el => el === value)

)();

contains(["foo", "bar"], "baz"); // => false

## 变量

const 优于let ，let 优于var。

// bad

var me = new Map();

me.set("name", "Ben").set("country", "Belgium");

// good

const me = new Map();

me.set("name", "Ben").set("country", "Belgium");

## 条件

IIFE 和return 语句优于if， else if，else和switch语句。

// bad

var grade;

if (result < 50)

grade = "bad";

else if (result < 90)

grade = "good";

else

grade = "excellent";

// good

const grade = (() => {

if (result < 50)

return "bad";

if (result < 90)

return "good";

return "excellent";

})();

## 对象迭代

如果可以的话，避免for…in。

const shared = { foo: "foo" };

const obj = Object.create(shared, {

bar: {

value: "bar",

enumerable: true

}

});

// bad

for (var prop in obj) {

if (obj.hasOwnProperty(prop))

console.log(prop);

}

// good

Object.keys(obj).forEach(prop => console.log(prop));

## map对象

在对象有合法用例的情况下，map通常是一个更好，更强大的选择。

// bad

const me = {

name: "Ben",

age: 30

};

var meSize = Object.keys(me).length;

meSize; // => 2

me.country = "Belgium";

meSize++;

meSize; // => 3

// good

const me = new Map();

me.set("name", "Ben");

me.set("age", 30);

me.size; // => 2

me.set("country", "Belgium");

me.size; // => 3

## Curry

Curry虽然功能强大，但对于许多开发人员来说是一个外来的范式。不要滥用，因为其视情况而定的用例相当不寻常。

// bad

const sum = a => b => a + b;

sum(5)(3); // => 8

// good

const sum = (a, b) => a + b;

sum(5, 3); // => 8

## 可读性

不要用看似聪明的伎俩混淆代码的意图。

// bad

foo || doSomething();

// good

if (!foo) doSomething();

// bad

void function() { /\* IIFE \*/ }();

// good

(function() { /\* IIFE \*/ }());

// bad

const n = ~~3.14;

// good

const n = Math.floor(3.14);

## 代码重用

不要害怕创建小型的，高度可组合的，可重复使用的函数。

// bad

arr[arr.length - 1];

// good

const first = arr => arr[0];

const last = arr => first(arr.slice(-1));

last(arr);

// bad

const product = (a, b) => a \* b;

const triple = n => n \* 3;

// good

const product = (a, b) => a \* b;

const triple = product.bind(null, 3);

## 依赖性

最小化依赖性。第三方是你不知道的代码。不要只是因为几个可轻易复制的方法而加载整个库：

// bad

var \_ = require("underscore");

\_.compact(["foo", 0]));

\_.unique(["foo", "foo"]);

\_.union(["foo"], ["bar"], ["foo"]);

// good

const compact = arr => arr.filter(el => el);

const unique = arr => [...Set(arr)];

const union = (...arr) => unique([].concat(...arr));

compact(["foo", 0]);

unique(["foo", "foo"]);

union(["foo"], ["bar"], ["foo"]);

译文链接：<http://www.codeceo.com/article/full-frontend-guidelines.html>  
英文原文：[Frontend Guidelines](https://github.com/bendc/frontend-guidelines)  
翻译作者：码农网 – 小峰