前端八股文

Pionpill [1]

前端面经

2023 年 3 月 28 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

前端面经，包含 HTML,CSS,JavaScript,TypeScript,React,Redux。

简单的内容只给出代码，不画图。这里只给出结果，至于比较复杂的一些原理与深入解析，请参考我的其他文章。

本文撰写环境:

- React: 18.2.0
- IDE: VSCode 1.72
- Chrome: 91.0
- Node.js: 18.12
- OS: Window11

2023 年 3 月 28 日

目录

I CSS

默认使用 border-box, 即加入如下代码:

```
* {
  box-sizing: border-box
}
```

1 绘制三角形

border

最简单的方法，使用 border 属性:

- 确保内部 width,height 为 0。
- 设置 border 颜色为透明。
- 设置某一边 border 存在颜色。

```
.triangle {
  width: 100px;
  height: 100px;
  border: 100px solid transparent;
  border-bottom: 100px solid #000;
  transform: translateY(-25\%); <!--调整位置-->
}
```

这样有个缺点，是等腰直角三角形，也可以绘制正三角形，无需调整位置:

```
.triangle {
  width: 120px;
  height: 120px;
  border-left: 69px solid transparent;
  border-right: 69px solid transparent;
  border-bottom: 120px solid skyblue;
}
```

clip-path

最推荐的写法，但部分浏览器不支持。利用多边形函数，进行裁剪:

```
.triangle {
  width: 100px;
  height: 80px;
  clip-path: polygon(0 0, 100\% 0, 50\% 100\%);
```

```
  background-color: aquamarine;
}
```

linear-gradient

原理是使用两张背景图片叠在一起。

```
.triangle {
  width: 80px;
  height: 100px;
  outline: 2px solid skyblue;
  background-repeat: no-repeat;
  background-image: linear-gradient(32deg, orangered 50\%, rgba(255, 255, 255, 0) 50\%),
      linear-gradient(148deg, orangered 50\%, rgba(255, 255, 255, 0) 50\%);
  background-size: 100\% 50\%;
  background-position: top left, bottom left;
}
```

实际上中间部分存在一条透明线，不推荐使用。

类似的，还有绘制圆形，也有上述三种思路。

2 ellipsis

2.1 单行省略号

这个比较简单，直接上代码:

```
.ellipsis {
  white-space: nowrap;
  width: 100px;     <!--设置一个宽度-->
  overflow: hidden; <!--超出隐藏-->
  text-overflow: ellipsis; <!--隐藏部分使用省略号-->
}
```

2.2 多行省略号

有两种方案，一是采用 webkit 提供的扩展属性:

```
.ellipse {
  width: 50px;
  height: 65px;
  word-break: break-all;
  overflow: hidden;
  display: -webkit-box;
  -webkit-box-orient: vertical;
  -webkit-line-clamp: 3;
```

```
}
```

这种方案有两个缺陷: 一是并非所有浏览器支持 webkit(虽然绝大部分支持)。而是后续内容其实并没有被隐藏，如果设置足够高，会发现后续内容依然存在，我们只是让第 3 行末尾添加了省略号，仅此而已。

除此以外，我们可以创建为元素:

```
.ellipse {
  width: 50px;
  height: 65px;
  position: relative;
  word-break: break-all;
  overflow: hidden;
  background-color: beige;
}

.ellipse::before {
  content: "...";
  position: absolute;
  bottom: 0;
  right:0;
}
```

但这样存在一个问题，适用性差，如果文本没有超出范围，省略号依然存在。一个解决方案是再添加一个遮蔽块伪元素放在文档后面，背景颜色与容器颜色相同:

```
.ellipse::before {
  content: "...";
  position: absolute;
  bottom: 0;
  right:0;
}

.ellipse::after {
  content: "";
  background-color: blue;
  width: 1em;
  height: 1em;
  position: absolute;
  right: 0;
  margin-top: 5px;
}
```

但这样还是存在问题，我们设置的遮蔽块会跟在文本后面，但如果文本只有两行，遮蔽块会出现在第二行末尾，而省略号出现在第三行末尾，并没有起到遮蔽效果。

此外，如果背景颜色不是固定的，比如是图片，或者渐变。那么遮蔽块就无效了。

终极解决方案是：用 js 控制。不过这就不是这里要讨论的内容了。

3 水平居中

文档流

HTML 的元素类型大致可以分为三类:

- 块级元素: 元素创建独立的块，宽度为容器 (父元素) 的整个宽度。常见的块级元素有 `h1,h2...,p,div,ul...`
- 行内元素: 元素不会创建新行，只会在文本中嵌入并与文本同行显示元素。它们只会占据内容所需的宽度，常见的有 `span, img, input, button, strong`
- 行内块级元素: 像行内元素一样显示在同一行上，但可以设置宽度高度属性。行内快元素也经常被归为行内元素。如 `button, input, img`。

简单居中

关于居中要分多种情况讨论:

- 行内元素: 行内元素由于没有宽高，因此需要改变父级属性达到居中效果。
  - 水平居中 (父属性): `text-align: center`。
  - 垂直居中 (父属性): `line-height: xxx`。
- 块级元素: 块级元素本身可以调整位置:
  - 水平居中: `margin: 0 auto`。
  - 垂直居中: 看下文。

水平居中我们解决了，垂直居中就比较难搞了。一种方案是硬编码:

- `margin-left: xxx`: 优点是实现简单，缺点是硬编码，适用性不高。
- 改为 `absolute` 布局，硬编码调整位置 (容器改为 `relative` 布局): 没什么优点，需要看需求。缺点是硬编码，另外会导致元素脱离文档流。

Flex 布局

Flex 布局现在已经推广开来，且有良好的浏览器兼容性。使用 Flex 布局居中非常简单:

```
.flex {
  justify-content: center;
  align-items: center;
}
```

这样就可以达到主轴和副轴居中的效果。

`translate`

`translate` 方法可以调整元素位置，注意这里使用% 调整是指元素本身宽高的百分比。该方法常常用在绝对布局，SVG 标签中。

```
.translate {
  position: absolute;
  left: 50\%;
  top: 50\%;
  transform: translate(-50\%, -50\%);
  }
```

vertical-align

这个方法用的非常少，用来表示普通流垂直方向的基线，即对齐线，Flex 布局也有对应的属性，这里不做过多介绍。

4 BFC

BFC, 块级作用域上下文。主要有两个功能:

- 对外，去除外边距折叠。
- 对内，形成独立的内部区域，所有元素均在 BFC 内。

BFC 创建后由于对内会将内部所有元素包含进来，因此可以清除浮动，清除高度塌陷效果。

建立一个 BFC 区域的方法有:

- 根元素：根元素即 HTML 元素，它默认就是一个 BFC。
- 浮动元素：如果一个元素被设置为浮动（float），那么它会形成一个 BFC。
- 绝对定位元素：如果一个元素被设置为绝对定位（position: absolute/fixed），那么它会形成一个 BFC。
- display 属性值为 inline-block、table-cell、table-caption 的元素：这些元素会形成一个 BFC。
- overflow 属性值不为 visible 的元素：如果一个元素的 overflow 属性值设置为 auto、scroll 或 hidden，那么它会形成一个 BFC。

主动建立 BFC 常常用 `overflow: auto`。因为这样没什么实际影响 (影响很小)。或者使用 Flex 布局。

II JavaScript

5 相等判断

JavaScript 有两种等于判断，相等 (弱等于) == 与全等 (强等于) ===。限制一般都用全等，若等于基本被弃用了。

先说一下全等:

- 简单类型: 判断值。
- 引用类型: 判断栈指针指向的堆地址。
- 基本与引用: false。

然后是弱等于，弱等于不分简单类型与引用类型，因为若等于会进行类型转换。如果类型相同，则按照全等逻辑判断。类型转换的逻辑如下:

- `null == undefined`: 返回 `true`。
- `null undefined`: 不会进行转换，仅有上面一种特殊情况。
- 一个为数字，一个为字符串: 转换为数字。
- 一个为布尔值，一个为非布尔值: 布尔值转换为数字。
- 一个为对象，一个为非对象: 将对象转换为原始值。
- 一个为 NaN, 则一定返回 `false`, 包括 `NaN == NaN`。

字符串转为数字比较特殊，如果不能转换，则转换为 NaN，如果是空字符串 (包括都是空元素的情况)，转换为 0。

有一点需要注意，虽然在弱等运算的时候，字符串会转为数值进行判断，但是在进行加法运算的时候，数值会转化为字符串。

如果加法运算时双方任意一方为字符串，或者双方均不能进行加法运算，则转换为字符串拼接。

6 类型判断

6.1 类型判断的方法

JS 有两个通用的类型判断运算符: `typeof, instanceof`。

- `typeof`: 单目运算符: 返回数据类型，返回值为字符串型。可以接受任意类型的参数。
  - 基本类型: `null` 返回 'object', 其他返回对应类型。
  - 引用类型: `Function` 返回 'function', 其他返回 'object'。
- `instanceof`: 双目运算符: 返回布尔值，左边为引用类型对象，右边为构造函数。

这两个方法都有各自的缺陷，typeof 无法判断明确的类型，除了函数。instanceof 需要两个运算符，无法判断基本类型。

JS 还提供了几种函数用于判断一些常见类型: Array.isArray() 用于判断是否为数组。isNaN() 用于判断是否为 NaN。

此外有一个究极方法: Object.prototype.toString.call()。这个方法会将基本类型转换为对应的包装类，并输出对应的包装类类型对应的字符串。例如 [object Number]。

6.2 手写类型判断方法

手写 typeof

用 Object.prototype.toString.call() 获取类型，然后规定仅输出指定类型就可以了。

```
const customTypeof = (param) => {
  const limitArray = ['undefined', 'number', 'boolean', 'string', 'symbol', 'bigint',
      'object', 'function'];
  const oriOut = Object.prototype.toString.call(param).slice(8, -1).toLowerCase();
  if (limitArray.includes(oriOut)) {
    return oriOut;
  } else {
    return "object"
  }
}
```

手写 instanceof

首先要判断两个参数是不是对应的类型，其次查构造函数的原型链，查到了即返回 true，否则一直查到 null 返回 false。

```
const customInstanceOf = (object, func) => {
  if (object === null || (typeof object !== 'object' && typeof object !== 'function'))
    return false;
  if (typeof func !== 'function')
    return false;
  let objProto = Object.getPrototypeOf(object);
  const funcProto = func.prototype;
  while (true) {
    if (objProto === null)
      return false;
    if (objProto === funcProto)
      return true;
    objProto = Object.getPrototypeOf(objProto);
  }
}
```

7 作用域相关

7.1 var

var 几乎不用，var 有两个特点:

- 声明提升: 在当前函数作用域的任意地方用 var 定义变量都会造成变量的声明提升，赋值不会提升。

```
// 声明提升
console.log(a) // undefined
var a = 1;
// 等价写法
var a;
console.log(a) // undefined
a = 1;
```

- 函数作用域: var 与常规变量声明不同，var 的作用域是函数作用域，不是块作用域。

```
// 直接报错
(() => {
  var a = 1;
})()
console.log(a);
// 不报错
var a = 1;
(() => {
  var a = 2;
  console.log(a); // 2
})()
console.log(a); // 1
```

还有一种特别的，去掉 var。这时候无论是在哪里声明，都是创建一个全局对象的属性。

注意，通过 var 声明的全局作用域的变量会自动成为全局对象 (浏览器是 window, node 是 global) 的属性。但是，全局作用域的变量不等于全局对象的属性。有以下几个区别:

- 全局变量不能通过 delete 关键字删除，全局对象的属性可以。
- 访问未声明的变量会报错，访问未声明的对象属性会返回 undefined。

这是由历史原因造成的，ES6 之前，JS 都是没有块作用域这个概念的。

```
{
  var a = 1;
}
console.log(a); // 1
```

此外，如果函数作用域中通过 var 声明了局部变量，那么外部的同名变量不会影响到内部的同名变量:

```
var a = "a";
(() => {
```

```
  console.log(a); // undefined
  var a = "b";
  console.log(a); // b
})()
```

7.2 函数声明

我们知道，js 主要有三种创建函数的方法: 函数声明，函数表达式，箭头函数。函数声明会带来一个整体提升的问题。而其他两种定义函数的方法不会:

```
a(); // 111
b(); // 报错
c(); // 报错

function a () {
  console.log("111")
}
const b = function () {
  console.log("222")
}
const c = () => {
  console.log("333")
}
```

7.3 作用域链

JS 的执行上下文主要分为两类: 全局环境，函数环境。(还有一个非常冷门的 `eval()` 环境，几乎不会用)。进入任意一个函数相当于在上下文栈中压入一个新的函数执行上下文，这个栈的栈底永远是全局执行上下文。

当我们的程序进入一个执行上下文的时候会有两个阶段:

- 创建阶段:
  - 创建作用域链 (当前变量对象 + 所有父级变量对象)。
  - 变量对象 (参数，变量，函数声明)。
  - this
- 执行阶段: 变量赋值，函数引用。

利用作用域链产生闭包:

```
const a = "a";
const func1 = () => {
  const b = "b";
  return () => {
    const c = "c";
    console.log(a) // a
```

```
    console.log(b) // b
    console.log(c) // c
  }
}
```

此时我们调用 `func1()` 并执行，内部函数的作用域链就包括: 全局变量 (a) + 外部函数变量 (b) + 内部函数变量 (c)。因此三个变量都能被访问到。且在对对象进行访问时，会优先查找内部变量，再依次向外查询。根据这个原理，我们可以创建闭包来对作用域链上的其他变量进行访问。

8 原型链

原型链有三个关键的概念: 对象，构造函数，原型 (原型本身也是一个对象)。它们之间的关系如下:

- 构造函数的的 `prototype` 属性指向原型。
- 构造函数通过 `new` 创建对象。
- 对象的 `__proto__` 属性指向原型。
- 原型的 `constructor` 属性指向对应的构造函数。
- 原型本身也是对象，因此原型也有一个 `__proto__` 属性指向原型的原型。如果为 `null`，则说明是 `Object` 对象。

8.1 new 过程

假设有如下代码:

```
const Mother = function (name) {
  this.name = name;
}
var son = new Mother("pionpill");
```

通过 `new` 关键字创建对象会发生如下过程:

- 创建一个新的空对象 `son`。
- 新对象的原型被绑定起来:

  ```
  son.__proto__ = Mother.prototype;
  ```

- 新对象和函数调用的 `this` 会绑定起来:

  ```
  Mother.call(son,"pionpill")
  ```

- 执行构造函数中的代码;
- 如果没有返回值，则自动返回这个新对象。

注意，在 JavaScript 中，普通函数和构造函数没有任何区别 (写法上，我们习惯将构造函

数使用大驼峰写法)。如果是通过 new 创建，则会自动返回新对象，但如果不是，则执行普通函数的逻辑，不进行原型绑定，this 绑定，返回新对象等操作。

此外，如果 new 构造函数，且构造函数中有返回值。假如返回值是基本类型，没有什么影响，new 的时候直接忽视，只对非 new 有影响。如果返回引用类型，那么返回的引用类型会替代原来的返回值。

9 事件

9.1 事件处理程序

DOM 事件流分为三个阶段: 事件捕获，事件执行，事件冒泡。也即先深度搜索，再找到节点，再向外冒泡。

DOM 事件何时触发分为捕获阶段触发与冒泡阶段触发，下文简称事件捕获，事件冒泡。

事件处理程序分为几种:

- HTML 事件处理: 事件冒泡。下面代码结果为 3 2 1。

```html
<div onclick="console.log('1')">
  <button onclick="console.log('2')">
    <span onclick="console.log('3')">
      text
    </span>
  </button>
</div>
```

- DOM0 级: 实现了 HTML 与 JS 分离，事件冒泡，结果为 3 2 1。

```js
const div = document.getElementsByTagName("div")[0];
const button = document.getElementsByTagName("button")[0];
const span = document.getElementsByTagName("span")[0];

div.onclick = () => console.log("1");
button.onclick = () => console.log("2");
span.onclick = () => console.log("3");
```

- DOM2 级: 默认事件冒泡，但可以通过 addEventListener 改为事件捕获，下面代码结果为 DIV, SPAN, BUTTON。

```js
const div = document.getElementsByTagName("div")[0];
const button = document.getElementsByTagName("button")[0];
const span = document.getElementsByTagName("span")[0];

const clickFunc = (element) => {
  return () => {
    console.log(element.nodeName);
  }
}
```

```
10
11 div.addEventListener("click", clickFunc(div), true);
12 button.addEventListener("click", clickFunc(button), false);
13 span.addEventListener("click", clickFunc(span), true);
```

10 this

10.1 普通函数的中的 this

JavaScript 中非箭头函数的 this 有四种绑定规则:

- 默认绑定: 即默认绑定到全局作用域。

```
1 // 浏览器环境
2 (function() {
3   console.log(this === window); // true
4 })()
```

- 隐式绑定: 函数作为对象的方法被调用时，this 指向方法运行时所在的当前对象。

```
1 const obj = {
2   a: 1,
3   foo: function () {
4     console.log(this); // {a: 1, foo: f}
5   }
6 }
```

隐式绑定存在优先级和隐式丢失的问题:

```
1 const obj1 = {
2   a: "a1",
3   b: "b",
4   obj2: {
5     a: "a2",
6     foo: function () {
7       console.log(this.a); // a2
8       console.log(this.b); // undefined
9     }
10  }
11 }
```

这里调用 foo()，this 只会绑定到上一级对象上，获取不到更外层的变量。

- 显示绑定: 使用 call，apply，bind 改变函数的调用对象。

```
1   var name = "window";
2   var outer = "outer";
3
4   const foo = function () {
5     var mark = "mark";
6     console.log(this.name);
```

```
    console.log(this.outer);
    console.log(this.mark);
  }
  const obj = {
    name: "obj",
    mark: "mark",
  }

  foo();
  foo.call(obj);
// window outer, undefined
// obj, undefined, mark
```

这里第一次调用时，this 默认绑定到 window 上，第二次调用时，显示绑定到 obj 对象上。

• new 绑定: 函数通过 new 调用后，会返回一个新对象，并将新对象绑定到函数调用的 this 上。

总的来说，普通函数的 this 指向会出现以下几种情况:

- 以函数形式调用: 永远指向 window。
- 以对象方法形式调用: 指向调用的那个对象。
- 以构造函数形式调用: 指向构造函数创建的实例对象。
- 以事件绑定函数形式调用: 指向绑定事件的对象。
- 显示绑定: 指向绑定的对象。

此外，显示绑定有例外，对于基本类型，浏览器会自动转化为引用类型进行绑定，但如果是 undefined，null，使用默认绑定规则。

绑定优先级: new/显示绑定 > 隐式绑定 > 默认绑定。

10.2 箭头函数中的 this

箭头函数自身没有 this。箭头函数中的 this 拿的是定义函数是外部环境的 this。

```
var obj = {
  data: [],
  getData: function() {
    return (function() {
      var result = ["abc","cba","nba"]
      this.data = result
    })()
  }
}

obj.getData()
console.log(obj.data); // Array(0)
```

上面调用方法时，该方法本身的 this 指向 obj。但是方法内回调的方法在执行时 this

指向的是 window。此时 this 没有 data 属性，因此没有任何影响。

如果要达到给数组赋值的效果，可以这样改进:

```
var obj = {
  data: [],
  getData: function () {
    let self = this;
    return (function() {
      var result = ["abc","cba","nba"]
      self.data = result
    })()
  }
}

obj.getData()
console.log(obj.data); // Array(0)
```

这样用的是闭包思想，在函数指向上下文中创建变量，这个变量会被加入到内部函数的作用域链中。

或者用箭头函数:

```
var obj = {
  data: [],
  getData: function () {
    return (() => {
      var result = ["abc","cba","nba"]
      this.data = result
    })()
  }
}

obj.getData()
console.log(obj.data); // Array(0)
```

箭头函数中的 this 是外部函数的 this，外部函数的 this 被隐式绑定到了 obj 对象上。

10.3 手写显式绑定

手写 call，apply

我们知道绑定 this 只有四种方法，如果需要手写显式绑定，自然需要借助其他三种方法中的某一种，默认绑定和 new 绑定自然不行，三个显式绑定都需要传入一个对象作为参数，因此自然是使用隐式绑定。

我们需要将函数放到对象中。这样就可以通过隐式绑定改变 this 指向。有几点需要注意:

- call 是原型上的方法，所以我们也需要写在原型上。

- 调用 newCall 时，是使用函数调用的，函数本身也是对象，因此调用时的 this 指向我们要绑定的函数本身。
- 传入参数为 undefined，null 时，需要默认绑定到 window 对象上。

```
Function.prototype.newApply = function (obj, ...args) {
  const realObj = obj || window;
  realObj.p = this;
  let result = realObj.p(...args);
  delete realObj.p;
  return result;
}
```

有了 call，apply 思路也类似:

```
Function.prototype.newApply = function (obj, ...args) {
  const realObj = obj || window;
  realObj.p = this;
  let result = realObj.p(...args[0]);
  delete realObj.p;
  return result;
}
```

手写 bind

首先最基础的，返回的一个函数:

```
Function.prototype.newBind = function (obj) {
  const self = this;
  return function () {
    return self.apply(obj || window);
  }
}
```

其次，我们知道，原生 bind 方法可以接受多个参数作为函数参数，且需要柯里化:

```
Function.prototype.newBind = function (obj, ...params1) {
  const self = this;
  return function (...params2) {
    return self.apply(obj || window, [...params1, ...params2]);
  }
}
```

另外，原生 bind 方法可以 new 对象，因此需要绑定原型:

```
Function.prototype.newBind = function (obj, ...params1) {
  const self = this;
  const Func = function (...params2) {
    return self.apply(obj || window, [...params1, ...params2]);
  }
  Func.prototype = Object.create(self.prototype);
  Func.prototype.constructor = self;
```

```
  return Func;
}
```

11 setTimeout()

`setInterval()` 的运行机制如下:

- 在执行栈里面执行该方法.
- 回调函数进入执行队列。
- 回到函数执行完毕，执行栈重复上述过程。

如果我们的回调函数执行时间很短，在间隔时间内可以执行完成，那么没什么问题。但如果执行时间超过了间隔时间。回调函数不会在规定时间重新进入执行队列，而是会延迟执行。所以 `setInterval` 是有缺陷的。

因此我们可以用 `setInterval` 代替 `setTimeout`:

```
const newInterval = (func, ms) => {
  const inside = () => {
    func()
    setTimeout(inside, ms);
  }
  setTimeout(inside, ms);
}
```

12 防抖与节流

12.1 防抖

防抖是指事件触发时，相应的函数不会立即执行，而是会被推迟执行。多数情况下，触发的多个任务仅执行一次。

防抖最常见的应用场景是搜索框输入内容，只在最后等待一会后做响应。

```
const debounce = (func, ms) => {
  let process = null;
  return function() {
    clearTimeout(process);
    process = setTimeout(func, ms);
  }
}
```

这是最简单的防抖写法，还有两个问题没有解决:

- `this` 应该指向事件对应的元素本身。
- 函数应该具备传参功能。

在上述代码中，我们会直接返回函数本身，这个函数的 this 会动态绑定指向 元素，但是 setTimeout 回调函数不会 (这里回调的 func 不能是箭头函数，否则无法动态绑定到元素上)。

```
const debounce = (func, ms) => {
  let process = null;
  return function (...args) {
    clearTimeout(process);
    process = setTimeout(() => {
      func.call(this, ...args);
    }, ms);
  }
}
```

12.2 节流

节流是指当事件触发时，会执行这个事件的响应函数，但如果事件被频繁触发，则间隔一段频率执行函数。

节流和防抖实现差不多:

```
const throttle = (func, delay) => {
  let process = null;
  return function (...args) {
    if (process)
    return;
    process = setTimeout(() => {
      func.call(this, ...args);
      process = null;
    }, delay);
  }
}
```

也可以不用 setTimeout 直接判断时间间隔。

```
const throttle = (func, delay) => {
  let pre = new Date();
  return function (...args) {
    let now = new Date();
    if (now - pre > delay) {
      func.call(this, args);
      pre = now;
    }
  }
}
```

13 懒加载

HTML 中的很多静态文件，在我们没有滚动到对应位置的时候也会进行加载，我们可以让这些资源在看不到的时候先不进行加载，比如图片:

最简单的方法是监听 window 的 scroll 事件:

```
const imgs = document.querySelectorAll("img");

window.addEventListener("scroll", (e) => {
  imgs.forEach(image => {
    const imgInfo = image.getBoundingClientRect().top;
    if (imgInfo < window.innerHeight) {
      image.setAttribute("src", image.getAttribute("data-src"));
    }
  })
})
```

但这样存在诸多缺陷:

- 滚动才触发，如果图片本身在最上面，需要额外监听 load 事件。
- 每次滚动就会判断，性能开销比较大。
- 引入了没必要的 data-src 属性。

JavaScript 为我们提供了一个好用的 API: IntersectionObserver。这个接口提供了一种异步观察元素与祖先元素或顶级文档 viewport 交集中变化的方法，这个方法接收两个参数:

```
new IntersectionObserver(callback, options)
```

其中，callback 为回调函数，options 为一些配置项。

- 回调函数接收一个参数 entries 是 IntersectionObserverEntry 实例数组，包含所有监听的元素，描述了目标元素与 root 的交叉状态。比较重要的有:
  - intersectionRadio: 返回目标元素出现在可视区的比例。
  - isIntersecting: 返回一个布尔值，目标元素出现在区域，返回 true，否则 false。
- options 是一个对象，用于配置参数，有三个属性如下:
  - root: 所监听的具体祖先元素，如果不传或为 null，表示顶级文档视窗。
  - rootMargin: 计算交叉时添加到 bounding box 的矩形偏移量。
  - threshold: 包含阈值的列表，按升序排列。

IntersectionObserver 的主要方法如下:

- observe(): 开始监听一个目标元素。
- unobserve(): 停止监听特定目标元素。
- takeRecords(): 返回所有观察目标的 IntersectionObserverEntry 对象数组。
- discount(): 停止全部监听工作。

14 深拷贝

首先明确一点，JS 的简单类型放在栈中，只有深拷贝，即复制一份。引用类型在堆中，默认进行浅拷贝，这是深拷贝需要解决的主要问题。

这里只讲自定义函数实现深拷贝，固有的 `JSON.parse(JSON.stringify(obj))` 以及一些其他方法具有很多缺陷，这里不作说明:

首先我们需要判断数据类型是不是对象:

```
const isObject = (obj) => {
  return (obj !== null && (typeof obj === 'object' || typeof obj === 'function'));
}
```

然后我们实现基本功能:

```
const deepClone = (obj) => {
  if (!isObject(obj))
    return obj;
  const newObj = {};
  // 基本类深拷贝
  newObj = Object.assign(obj);
  for (const key of obj) {
    deepClone(obj[key]);
  }
}
```

针对常见的不同类型，需要单独处理:

```
const deepClone = (obj) => {
  if (!isObject(obj))
    return obj;
  if (obj instanceof Array) {
    return obj.concat();
  }
  if (obj instanceof Set) {
    const temp = new Set();
    obj.forEach((item) => {
      temp.add(deepClone(item));
    })
    return obj;
  }
  if (obj instanceof Map) {
    const temp = new Map();
    obj.forEach((item, key) => {
      temp.set(key, deepClone(item));
    })
    return temp;
  }
  if (obj instanceof RegExp) {
    const temp = new RegExp(obj);
    return temp;
  }
```

```
  const newObj = {};
  // 基本类深拷贝
  newObj = Object.assign(obj);
  for (const key of obj) {
    deepClone(obj[key]);
  }
}
```

其次我们需要解决循环引用问题:

```
const deepClone = (obj) => {
  const isObject = (obj) => {
    return (obj !== null && (typeof obj === 'object' || typeof obj === 'function'));
  }

  const map = new WeakMap();

  const copy = (obj) => {
    if (!isObject(obj))
      return obj;

    if (map.get(obj))
      return map.get(obj);

    if (obj instanceof Array) {
      map.set(obj, obj.concat());
      return obj.concat();
    }
    if (obj instanceof Set) {
      const temp = new Set();
      map.set(obj, temp);
      obj.forEach((item) => {
        temp.add(deepClone(item));
      })
      return obj;
    }
    if (obj instanceof Map) {
      const temp = new Map();
      map.set(obj, temp);
      obj.forEach((item, key) => {
        temp.set(key, deepClone(item));
      })
      return temp;
    }
    if (obj instanceof RegExp) {
      const temp = new RegExp(obj);
      return temp;
    }
    const newObj = {};
    // 基本类深拷贝
    newObj = Object.assign(obj);
    for (const key of obj) {
      copy(obj[key]);
```

```
    }
  }
}
```

注意，只有容器 (集合，列表这些) 才会产生循环引用问题。

15 Promise

手写 Promise, 比较复杂，自行理解，几个关键点如下:

- 为了防止在 new 时，异步确认状态，导致 then 等方法调用时无法确认状态，需要保存一个回调函数执行队列。

```
class MyPromise {
  static PENDING = "pending";
  static FULFILLED = "fulfilled";
  static REJECTED = "reject";
  constructor(func) {
    this.resolveCallbacks = [];
    this.rejectCallbacks = [];
    this.status = MyPromise.PENDING;
    this.result = null;
    try {
      return func(this.resolve.bind(this), this.reject.bind(this));
    } catch {
      this.reject(this.result);
    }
  }

  resolve(result) {
    setTimeout(() => {
      if (this.status === MyPromise.PENDING) {
        this.status = MyPromise.FULFILLED;
        this.result = result;
        this.resolveCallbacks.forEach((func) => func(result));
      }
    });
  }

  reject(result) {
    setTimeout(() => {
      if (this.status === MyPromise.PENDING) {
        this.status = MyPromise.REJECTED;
        this.result = result;
        this.resolveCallbacks.forEach((func) => func(result));
      }
    });
  }

  then(onFulfilled, onRejected) {
```

```
    return new MyPromise((resolve, reject) => {
      onFulfilled = typeof onFulfilled === "function" ? onFulfilled : () => {};
      onRejected = typeof onRejected === "function" ? onRejected : () => {};
      // 待定状态判断
      if (this.status === MyPromise.PENDING) {
        this.resolveCallbacks.push(onFulfilled);
        this.rejectCallbacks.push(onRejected);
      }
      if (this.status === MyPromise.FULFILLED) {
        setTimeout(() => onFulfilled(this.result));
      }
      if (this.status === MyPromise.REJECTED) {
        setTimeout(() => onRejected(this.result));
      }
    });
  }

  catch(onRejected) {
    return new MyPromise((resolve, reject) => {
      onRejected = typeof onRejected === "function" ? onRejected : () => {};
      if (this.status === MyPromise.PENDING) {
        this.rejectCallbacks.push(onRejected);
      }
      if (this.status === MyPromise.REJECTED) {
        setTimeout(() => onRejected(this.result));
      }
    });
  }

  finally(onFinally) {
    onFinally = typeof onFinally === "function" ? onFinally : () => {};
    if (this.status === MyPromise.PENDING) {
      this.resolveCallbacks.push(onFinally);
      this.rejectCallbacks.push(onFinally);
    }
    if (this.status !== MyPromise.PENDING) {
      setTimeout(() => onFinally(this.result));
    }
  }
}
```