

原生 JS 实现最简单的图片懒加载

axuebin 脚本之家 2018-10-25



脚本之家

你与百万开发者在一起

关注

作者：axuebin

<https://segmentfault.com/a/1190000010744417>

懒加载

什么是懒加载

懒加载其实就是延迟加载，是一种对网页性能优化的方式，比如当访问一个页面的时候，优先显示可视区域的图片而不一次性加载所有图片，当需要显示的时候再发送图片请求，避免打开网页时加载过多资源。

什么时候用懒加载

当页面中需要一次性载入很多图片的时候，往往都是需要用懒加载的。

懒加载原理

我们都知道HTML中的 `` 标签是代表文档中的一个图像。。说了个废话。。

`` 标签有一个属性是 `src`，用来表示图像的URL，当这个属性的值不为空时，浏览器就会根据这个值发送请求。如果没有 `src` 属性，就不会发送请求。

嗯？貌似这点可以利用一下？

我先不设置 `src`，需要的时候再设置？

nice，就是这样。

我们先不给 `` 设置 `src`，把图片真正的URL放在另一个属性 `data-src` 中，在需要的时候也就是图片进入可视区域的之前，将URL取出放到 `src` 中。

实现

HTML结构

```
<div class="container">
  <div class="img-area">
    
  </div>
  <div class="img-area">
    
  </div>
  <div class="img-area">
    
  </div>
  <div class="img-area">
    
  </div>
  <div class="img-area">
    
  </div>
</div>
```

仔细观察一下，`` 标签此时是没有 `src` 属性的，只有 `alt` 和 `data-src` 属性。

`alt` 属性是一个必需的属性，它规定在图像无法显示时的替代文本。`data-*` 全局属性：构成一类名称为自定义数据属性的属性，可以通过 `HTMLElement.dataset` 来访问。

如何判断元素是否在可视区域

方法一

网上看到好多这种方法，稍微记录一下。

1. 通过 `document.documentElement.clientHeight` 获取屏幕可视窗口高度
2. 通过 `document.documentElement.scrollTop` 获取浏览器窗口顶部与文档顶部之间的距离，也就是滚动条滚动的距离
3. 通过 `element.offsetTop` 获取元素相对于文档顶部的距离

然后判断②-③<①是否成立，如果成立，元素就在可视区域内。

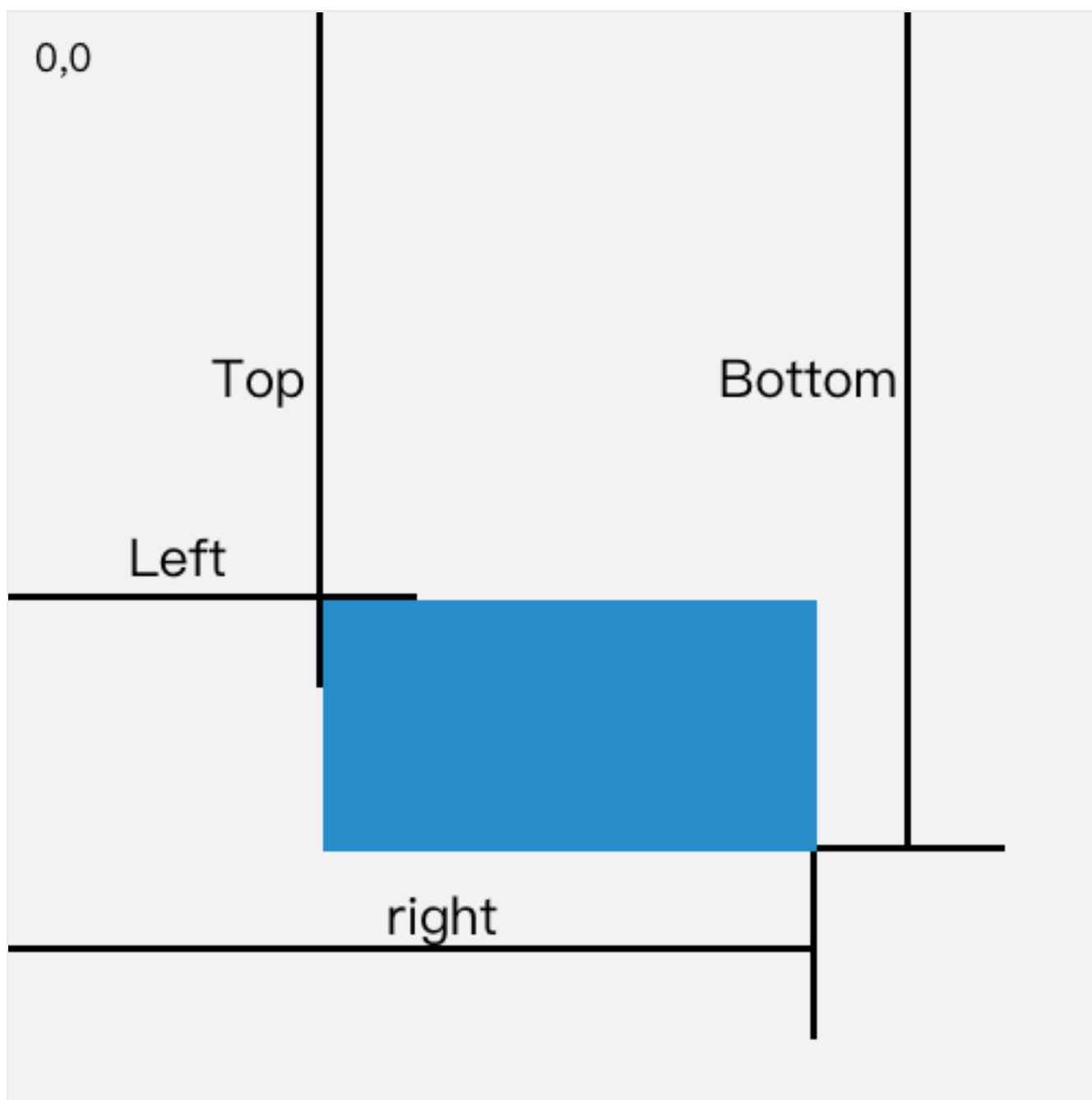
方法二（推荐）

通过 `getBoundingClientRect()` 方法来获取元素的大小以及位置，MDN上是这样描述的：

The `Element.getBoundingClientRect()` method returns the size of an element and its position relative to the viewport.

这个方法返回一个名为 `ClientRect` 的 `DOMRect` 对象，包含了 `top`、`right`、`bottom`、`left`、`width`、`height` 这些值。

MDN上有这样一张图：



可以看出返回的元素位置是相对于左上角而言的，而不是边距。

我们思考一下，什么情况下图片进入可视区域。

假设 `const bound = el . getBoundingClientRect ();` 来表示图片到可视区域顶部距离；
并设 `const clientHeight = window . innerHeight ;` 来表示可视区域的高度。

随着滚动条的向下滚动， `bound . top` 会越来越小，也就是图片到可视区域顶部的距离越来越小，当 `bound . top === clientHeight` 时，图片的上沿应该是位于可视区域下沿的位置的临界点，再滚动一点点，图片就会进入可视区域。

也就是说，在 `bound . top <= clientHeight` 时，图片是在可视区域内的。

我们这样判断：

```
function isInSight(el) {
  const bound = el.getBoundingClientRect();
  const clientHeight = window.innerHeight;
  //如果只考虑向下滚动加载
  //const clientWidth = window.innerWidth;
  return bound.top <= clientHeight + 100;
}
```

这里有个+100是为了提前加载。

加载图片

页面打开时需要对所有图片进行检查，是否在可视区域内，如果是就加载。

```
function checkImgs() {
  const imgs = document.querySelectorAll('.my-photo');
  Array.from(imgs).forEach(el => {
    if (isInSight(el)) {
      loadImg(el);
    }
  })
}

function loadImg(el) {
  if (!el.src) {
    const source = el.dataset.src;
    el.src = source;
  }
}
```

这里应该是有一个优化的地方，设一个标识符标识已经加载图片的index，当滚动条滚动时

就不需要遍历所有的图片，只需要遍历未加载的图片即可。

函数节流

在类似于滚动条滚动等频繁的DOM操作时，总会提到“函数节流、函数去抖”。

所谓的函数节流，也就是让一个函数不要执行的太频繁，减少一些过快的调用来节流。

基本步骤：







1. 获取第一次触发事件的时间戳
2. 获取第二次触发事件的时间戳
3. 时间差如果大于某个阈值就执行事件，然后重置第一个时间

```
function throttle(fn, mustRun = 500) {
  const timer = null;
  let previous = null;
  return function() {
    const now = new Date();
    const context = this;
    const args = arguments;
    if (!previous){
      previous = now;
    }
    const remaining = now - previous;
    if (mustRun && remaining >= mustRun) {
      fn.apply(context, args);
      previous = now;
    }
  }
}
```

这里的 `mustRun` 就是调用函数的时间间隔，无论多么频繁的调用 `fn`，只有 `remaining >= mustRun` 时 `fn` 才能被执行。

实验

页面打开时

All XHR JS CSS Img Media Font Doc WS Manifest Other					
20 ms		40 ms		60 ms	
80					
Name	Status	Type	Initiator	Size	Time
 index.html	Finished	docu...	Other	0 B	9 ms
 normalize.css	Finished	styles...	index.html	0 B	1 ms
 index.css	Finished	styles...	index.html	0 B	1 ms
 index.js	Finished	script	index.html	0 B	1 ms
 img1.png	Finished	png	index.js:21	0 B	3 ms
 img2.png	Finished	png	index.js:21	0 B	9 ms

可以看出此时仅仅是加载了img1和img2，其它的img都没发送请求，看看此时的浏览器



第一张图片是完整的呈现了，第二张图片刚进入可视区域，后面的就看不到了~

页面滚动时

当我向下滚动，此时浏览器是这样

















此时第二张图片完全显示了，而第三张图片显示了一点点，这时候我们看看请求情况

All	XHR	JS	CSS	Img	Media	Font	Doc	WS	Manifest	Other
	50000 ms	100000 ms	150000 ms	200000 ms	250000 ms	300000 ms				
Name	Status	Type	Initiator	Size	Time					
index.html	Finished	docu...	Other	0 B	9 ms					
normalize.css	Finished	styles...	index.html	0 B	1 ms					
index.css	Finished	styles...	index.html	0 B	1 ms					
index.js	Finished	script	index.html	0 B	1 ms					
img1.png	Finished	png	index.js:21	0 B	3 ms					
img2.png	Finished	png	index.js:21	0 B	9 ms					
img3.png	Finished	png	index.js:21	0 B	5 ms					

img3的请求发出来，而后面的请求还是没发出~

全部载入时

当滚动条滚到最底下时，全部请求都应该是发出的，如图

All	XHR	JS	CSS	Img	Media	Font	Doc	WS	Manifest	Other
50000 ms	100000 ms	150000 ms	200000 ms	250000 ms	300000 ms	350000 ms	400000 ms			
Name	Status	Type	Initiator	Size	Time					
 index.html	Finished	docu...	Other	0 B	9 ms					
 normalize.css	Finished	styles...	index.html	0 B	1 ms					
 index.css	Finished	styles...	index.html	0 B	1 ms					
 index.js	Finished	script	index.html	0 B	1 ms					
 img1.png	Finished	png	index.js:21	0 B	3 ms					
 img2.png	Finished	png	index.js:21	0 B	9 ms					
 img3.png	Finished	png	index.js:21	0 B	5 ms					
 img4.png	Finished	png	index.js:21	0 B	6 ms					
 img5.png	Finished	png	index.js:21	0 B	5 ms					
 img6.png	Finished	png	index.js:21	0 B	4 ms					
 img7.png	Finished	png	index.js:21	0 B	5 ms					
 img8.png	Finished	png	index.js:21	0 B	11 ms					
 img9.png	Finished	png	index.js:21	0 B	9 ms					
 img10.png	Finished	png	index.js:21	0 B	14 ms					

更新

方法三 IntersectionObserver

经大佬提醒，发现了这个方法

先附上链接：

jjc大大：

<https://github.com/justjavac/the-front-end-knowledge-you-may-dont-know/issues/10>

阮一峰大大：

http://www.ruanyifeng.com/blog/2016/11/intersectionobserver_api.html

API Sketch for Intersection Observers：

IntersectionObserver 可以自动观察元素是否在视口内。

```
var io = new IntersectionObserver(callback, option);
// 开始观察
io.observe(document.getElementById('example'));
// 停止观察
io.unobserve(element);
// 关闭观察器
io.disconnect();
```

callback的参数是一个数组，每个数组都是一个 IntersectionObserverEntry 对象，包括以下属性：

属性	描述
time	可见性发生变化的时间，单位为毫秒
rootBounds	与getBoundingClientRect()方法的返回值一样
boundingClientRect	目标元素的矩形区域的信息
intersectionRect	目标元素与视口（或根元素）的交叉区域的信息
intersectionRatio	目标元素的可见比例，即intersectionRect占boundingClientRect的比例，完全可见时为1，完全不可见时小于等于0
target	被观察的目标元素，是一个 DOM 节点对象

我们需要用到 intersectionRatio 来判断是否在可视区域内，当 intersectionRatio > 0 && intersectionRatio <= 1 即在可视区域内。

代码

```
function checkImgs() {
  const imgs = Array.from(document.querySelectorAll(".my-photo"));
  imgs.forEach(item => io.observe(item));
}

function loadImg(el) {
  if (!el.src) {
```

```
const source = el.dataset.src;
el.src = source;
}
}

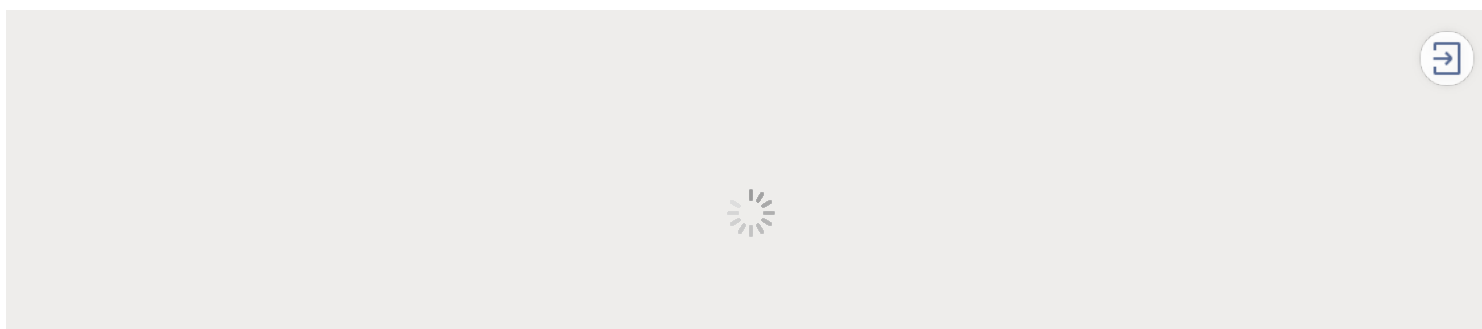
const io = new IntersectionObserver(ioes => {
  ioes.forEach(ioe => {
    const el = ioe.target;
    const intersectionRatio = ioe.intersectionRatio;
    if (intersectionRatio > 0 && intersectionRatio <= 1) {
      loadImg(el);
    }
    el.onload = el.onerror = () => io.unobserve(el);
  });
});
```

想了解更多前端知识？欢迎关注↓↓↓



- 关注后直接回复【电子书】，即可免费获取 27本 精选的前端电子书！
- 关注后直接回复【web100】，免费获取 100本 最棒的前端电子书！

今日头条回顾：



80后网警受贿两千万...



写了8年的代码，做过的项目都下线了...



送给你优秀的技术学习资源



- **考研早知道：** 如何加强记忆，比如背了忘不了（。ò ∨ ó。）
- Chrome 的哪些功能改变了我们浏览网页的方式？
- **脚本之家粉丝福利，请查看！**
- 大量学习视频、编程资源，欢迎收藏~
- 作为程序员，最起码要知道的几个公众号
- 99%的程序员都会收藏的书单，你读过几本？



小贴士

返回 上一级 搜索“[Java](#) [女程序员](#) [大数据](#) [留言送书](#) [运维](#) [算法](#) [Chrome](#) [黑客](#) [Python](#) [JavaScript](#) [人工智能](#) [女朋友](#) [MySQL](#) [书籍](#) 等关键词获取相关文章推荐。