# 前端快照方案：来自大厂的页面「自拍」

每日精选 前端潮咖 2023-02-14 09:07 发表于江苏



作者：ggvswild
文/云音乐前端技术团队

## 1. 背景

将网页保存为图片（以下简称为快照），是用户记录和分享页面信息的有效手段，在各种兴趣测试和营销推广等形式的活动页面中尤为常见。

快照环节通常处于页面交互流程的末端，汇总了用户最终的参与结果，直接影响到用户对于活动的完整体验。因此，生成高质量的页面快照，对于活动的传播和品牌的转化具有十分重要的意义。

本文基于云音乐往期优质活动的相关实践（例如「关于你的画」、「权力的游戏」和「你的使用说明书」等），从快照的**内容完整性**、**清晰度**和**转换效率**等多个方面，讨论将网页转换为高质量图片的实践探索。

## 2. 适用场景

- 适用于将页面转为图片，特别是对实时性要求较高的场景。

- 希望在快照中展示跨域图片资源的场景。
- 针对生成图片内容不完整、模糊或者转换过程缓慢等问题，寻求有效解决方案的场景。

## 3. 原理简析

### 3.1 方案选型

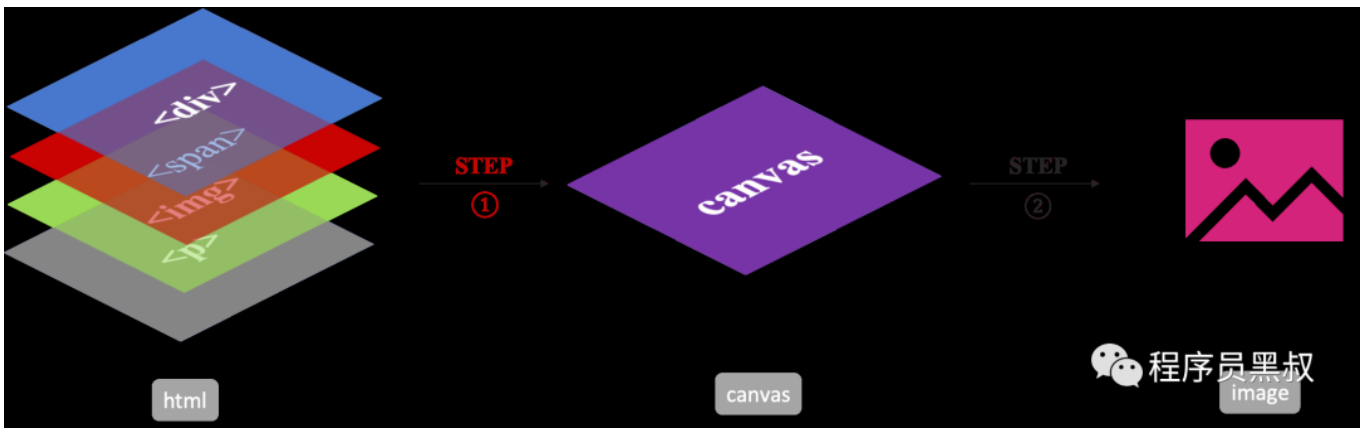
依据图片是否由设备**本地生成**，快照可分为前端处理和后端处理两种方式。

由于后端生成的方案依赖于网络通信，不可避免地存在通信开销和等待时延，同时对于模板和数据结构变更也有一定的维护成本。

因此，出于**实时性**和**灵活性**等综合考虑，我们优先选用前端处理的方式。

### 3.2 基本原理

前端侧对于快照的处理过程，实质上是将 DOM 节点包含的视图信息转换为图片信息的过程。这个过程可以借助 canvas 的原生 API 实现，这也是方案可行性的基础。




theory

具体来说，转换过程是将目标 DOM 节点绘制到 canvas 画布，然后 canvas 画布以图片形式导出。可简单标记为绘制阶段和导出阶段两个步骤：

- **绘制阶段**：选择希望绘制的 DOM 节点，根据 `nodeType` 调用 canvas 对象的对应 API，将目标 DOM 节点绘制到 canvas 画布（例如对于 `<img>` 的绘制使用 drawImage 方法）。
- **导出阶段**：通过 canvas 的 toDataURL 或 getImageData 等对外接口，最终实现画布内容的导出。

### 3.3 原生示例

具体地，对于单个 `<img>` 元素可按如下方式生成自身的快照：

**HTML**:

```
<img id="target" src="./music-icon.png" />
```

**JavaScript**:

```
// 获取目标元素
const target = document.getElementById('target');
// 新建canvas画布
const canvas = document.createElement('canvas');
canvas.width = 100;
canvas.height = 100;
const ctx = canvas.getContext("2d");
// 导出阶段：从canvas导出新的图片
const exportNewImage = (canvas) => {
    const exportImage = document.createElement('img');
    exportImage.src = canvas.toDataURL();
    document.body.appendChild(exportImage);
}
// 绘制阶段：待图片内容加载完毕后绘制画布
target.onload = () => {
    // 将图片内容绘入画布
    ctx.drawImage(target, 0, 0, 100, 100);
    // 将画布内容导出为新的图片
    exportNewImage(canvas);
}
```

其中，`drawImage` 是 canvas 上下文对象的实例方法，提供多种方式将 CanvasImageSource 源绘制到 canvas 画布上。`exportNewImage` 用于将 canvas 中的视图信息导出为包含图片展示的 data URI。

## 4. 基础方案

在上一部分中，我们可以看到基于 canvas 提供的相关基础 API，为前端侧的页面快照处理提供了可能。

然而，具体的业务应用往往更加复杂，上面的「低配版」实例显然未能覆盖多数的实际场景，例如：

- canvas 的 `drawImage` 方法只接受 CanvasImageSource，而 `CanvasImageSource` 并不包括文本节点、普通的 `div` 等，将非 `<img>` 的元素绘制到 canvas 需要特定处理。
- 当有多个 DOM 元素需要绘制时，层级优先级处理较为复杂。
- 需要关注 `float` 、 `z-index` 、 `position` 等布局定位的处理。
- 样式合成绘制计算较为繁琐。

因此，基于对综合业务场景的考虑，我们采用社区中认可度较高的方案： `html2canvas` 和 `canvas2image` 作为实现快照功能的基础库。

### 4.1 html2canvas

> 提供将 DOM 绘制到 canvas 的能力

这款来自社区的神器，为开发者简化了将逐个 DOM 绘制到 canvas 的过程。简单来说，其**基本原理**为：

- 递归遍历目标节点及其子节点，收集节点的样式信息；
- 计算节点本身的层级关系，根据一定优先级策略将节点逐一绘制到 canvas 画布中；
- 重复这一过程，最终实现目标节点内容的全部绘制。

在使用方面， `html2canvas` 对外暴露了一个可执行函数，它的第一个参数用于接收待绘制的目标节点(必选)；第二个参数是可选的配置项，用于设置涉及 canvas 导出的各个参数：

```
// element 为目标绘制节点，options为可选参数
html2canvas(element[,options]);
```

简易调用示例如下：

```
import html2canvas from 'html2canvas';

const options = {};

// 输入body节点，返回包含body视图内容的canvas对象
html2canvas(document.body, options).then(function(canvas) {
```

```
    document.body.appendChild(canvas);
});
```

## 4.2 canvas2image

> 提供由 canvas 导出图片信息的多种方法

相比于 `html2canvas` 承担的复杂绘制流程，canvas2image 所要做的事情简单的多。

`canvas2image` 仅用于将输入的 canvas 对象按特定格式转换和存储操作，其中这两类操作均支持 PNG，JPEG，GIF，BMP 四种图片类型：

```
// 格式转换
Canvas2Image.convertToPNG(canvasObj, width, height);
Canvas2Image.convertToJPEG(canvasObj, width, height);
Canvas2Image.convertToGIF(canvasObj, width, height);
Canvas2Image.convertToBMP(canvasObj, width, height);

// 另存为指定格式图片
Canvas2Image.saveAsPNG(canvasObj, width, height);
Canvas2Image.saveAsJPEG(canvasObj, width, height);
Canvas2Image.saveAsGIF(canvasObj, width, height);
Canvas2Image.saveAsBMP(canvasObj, width, height);

```

实质上，`canvas2image` 只是提供了针对 canvas 基础 API 的二次封装（例如 getImageData、toDataURL），而本身并不依赖 `html2canvas` 。

在使用方面，由于目前作者并未提供 ES6 版本的 `canvas2image` (v1.0.5)，暂时不能直接以 import 方式引入该模块。

对于支持现代化构建的工程中（例如 webpack），开发者可以自助 clone 源码并手动添加 export 获得 ESM 支持：

**支持 ESM 导出**：

```
// canvas2Image.js
const Canvas2Image = function () {
```

```
    ...
}();

// 以下为定制添加的内容
export default Canvas2Image;
```

**调用示例**：

```
import Canvas2Image from './canvas2Image.js';

// 其中，canvas代表传入的canvas对象，width, height分别为导出图片的宽高数值
Canvas2Image.convertToPNG(canvas, width, height)
```

## 4.3 组合技

接下来，我们基于以上两个工具库，实现一个基础版的快照生成方案。同样是分为两个阶段，对应 3.2 节的基本原理：

- 第一步，通过 `html2canvas` 实现 DOM 节点绘制到 canvas 对象中；
- 第二步，将上一步返回的 canvas 对象传入 `canvas2image`，进而按需导出快照图片信息。

具体地，我们封装一个 `convertToImage` 的函数，用于输入目标节点以及配置项参数，输出快照图片信息。

`JavaScript`：

```
// convertToImage.js
import html2canvas from 'html2canvas';
import Canvas2Image from './canvas2Image.js';
/**
 * 基础版快照方案
 * @param {HTMLElement} container
 * @param {object} options html2canvas相关配置
 */
function convertToImage(container, options = {}) {
    return html2canvas(container, options).then(canvas => {
        const imageEl = Canvas2Image.convertToPNG(canvas, canvas.width, canvas.height);
        return imageEl;
```

```
    });
}
```

## 5. 进阶优化

通过上一节的实例，我们基于 `html2canvas` 和 `canvas2image`，实现了相比原生方案**通用性**更佳的基础页面快照方案。然而面对实际复杂的应用场景，以上基础方案生成的快照效果往往不尽如人意。

快照效果的**差异性**，一方面是由于 `html2canvas` 导出的视图信息是通过各种 DOM 和 canvas 的 API 复合计算二次绘制的结果（并非一键栅格化）。因此不同宿主环境的相关 API 实现差异，可能导致生成的图片效果存在多端不一致性或者显示异常的情况。

另一方面，业务层面的因素，例如对于开发者 `html2canvas` 的配置有误或者是页面布局不当等原因，也会对生成快照的结果带来偏差。

社区中也可以常见到一些对于生成快照质量的讨论，例如：

- 为什么有些内容显示不完整、残缺、白屏或黑屏?
- 明明原页面清晰可辨，为什么生成的图片模糊如毛玻璃?
- 将页面转换为图片的过程十分缓慢，影响后续相关操作，有什么好办法么?
- ...

下面我们从**内容完整性**、**清晰度优化**和**转换效率**，进一步探究高质量的快照解决方案。

### 5.1 内容完整性

> 首要问题：保证目标节点视图信息完整导出

由于真机环境的兼容性和业务实现方式的不同，在一些使用 `html2canvas` 过程中常会出现快照内容与原视图不一致的情况。内容不完整的常见自检 `checklist` 如下：

- **跨域问题**：存在跨域图片污染 canvas 画布。
- **资源加载**：生成快照时，相关资源还未加载完毕。
- **滚动问题**：页面中滚动元素存在偏移量，导致生成的快照顶部出现空白。

### 5.1.1 跨域问题

常见于引入的图片素材相对于部署工程跨域的场景。例如部署在 `https://st.music.163.com/` 上面的页面中引入了来源为 `https://p1.music.126.net` 的图片，这类图片即是属于跨域的图片资源。

由于 canvas 对于图片资源的同源限制，如果画布中包含跨域的图片资源则会污染画布( Tainted canvases )，造成生成图片内容混乱或者 `html2canvas` 方法不执行等异常问题。

对于跨域图片资源处理，可以从以下几方面着手：

#### （1）useCORS 配置

开启 `html2canvas` 的 `useCORS` 配置项，示例如下：

```
// doc: http://html2canvas.hertzen.com/configuration/
const opts = {
    useCORS    : true,    // 允许使用跨域图片
    allowTaint: false    // 不允许跨域图片污染画布
};

html2canvas(element, opts);
```

在 `html2canvas` 的源码中对于 `useCORS` 配置项置为 `true` 的处理，实质上是将目标节点中的 `<img>` 标签注入 crossOrigin 为 `anonymous` 的属性，从而允许载入符合 CORS 规范的图片资源。

其中， `allowTaint` 默认为 `false` ，也可以不作显式设置。即使该项置为 `true` ，也不能绕过 canvas 对于跨域图片的限制，因为在调用 canvas 的 `toDataURL` 时依然会被浏览器禁止。

#### （2）CORS 配置

上一步的 `useCORS` 的配置，只是允许 `<img>` 接收跨域的图片资源，而对于解锁跨域图片在 canvas 上的绘制并导出，需要图片资源本身需要提供 CORS 支持。

这里介绍下跨域图片使用 CDN 资源时的注意事项：

验证图片资源是否支持 CORS 跨域，通过 Chrome 开发者工具可以看到图片请求响应头中应含有 `Access-Control-Allow-Origin` 的字段，即坊间常提到的跨域头。

例如，某个来自 CDN 图片资源的响应头示例：

```
// Response Headers
```

```
access-control-allow-credentials: true
access-control-allow-headers: DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type
access-control-allow-methods: GET,POST,OPTIONS
access-control-allow-origin: *
```

不同的 CDN 服务商配置资源跨域头的方式不同，具体应咨询 CDN 服务商。

特殊情况下，部分 CDN 提供方可能会存在图片缓存不含 CORS 跨域头的情况。为保证快照显示正常，建议优先联系 CDN 寻求技术支持，不推荐通过图片链接后缀时间戳等方式强制回源，避免影响源站性能和 CDN 计费。

### （3）服务端转发

在微信等第三方 APP 中，平台的用户头像等图片资源是不直接提供 CORS 支持的。此时需要借助服务端作代理转发，从而绕过跨域限制。

即通过服务端代为请求平台用户的头像地址并转发给客户端(浏览器)，当然这个服务端接口本身要与页面同源或者支持 CORS。

为简洁表述，假设前端与后端针对跨域图片转发作如下约定，且该接口与前端工程部署在相同域名下：

| 请求地址 | 请求方式 | 传入参数 | 返回信息 |
| --- | --- | --- | --- |
| /api/redirect/image | GET | redirect，表示原图地址 | Content-Type为image/png的图片资源 |

页面中的 `<img>` 通过拼接 `/api/redirect/image` 与代表原图地址的查询参数 `redirect`，发出一个 GET 请求图片资源。由于接口与页面同源，因此不会触发跨域限制：

```
<img src="/api/redirect/image?redirect=thirdwx.qlogo.cn/somebody/avatar" alt="user-pic" class="avatar" crossorigin="anonymous">
```

对于服务端接口的实现，这里基于 koa 提供了一则简易示例：

```
const Koa = require('koa');
const router = require('koa-router')();
const querystring = require('querystring');
```

```
const app = new Koa();
/**
 * 图片转发接口
 * - 接收 redirect 入参，即需要代为请求的图片URL
 * - 返回图片资源
 */
router.get('/api/redirect/image', async function(ctx) {
    const querys = ctx.querystring;
    if (!querys) return;
    const { redirect } = querystring.parse(querys);
    const res = await proxyFetchImage(redirect);
    ctx.set('Content-Type', 'image/png');
    ctx.set('Cache-Control', 'max-age=2592000');
    ctx.response.body = res;
})
/**
 * 请求并返回图片资源
 * @param {String} url 图片地址
 */
async function proxyFetchImage(url) {
    const res = await fetch(url);
    return res.body;
}
const res = await proxyFetchImage(redirect);
app.use(router.routes());
```

在浏览器看来，页面请求的图片资源仍是相同域名下的资源，转发过程对前端透明。建议在需求开发前了解图片资源的来源情况，明确是否需要服务端支持。

在云音乐早期的活动「权力的游戏」中，使用了同类方案，实现了微信平台中用户头像的完整绘制和快照导出。

### 5.1.2 资源加载

资源加载不全，是造成快照不完整的一个常见因素。在生成快照时，如果部分资源没有加载完毕，那么生成的内容自然也谈不上完整。

除了设置一定的延迟外，如果要确保资源加载完毕，可以基于 Promise.all 实现。

**加载图片**：

```
const preloadImg = (src) => {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.onload = () => {
            resolve();
        }
        img.src = src;
    });
}
```

**确保在全部加载后生成快照**：

```
const preloadList = [
    './pic-1.png',
    './pic-2.png',
    './pic-3.png',
];

Promise.all(preloadList.map(src => preloadImg(src))).then(async ()
=> {
    convertToImage(container).then(canvas => {
        // ...
    })
});
```

实际上，以上方法只是解决页面图片的显示问题。在真实场景中，即使页面上的图片显示完整，保存快照后依然可能出现内容空白的情况。原因是 html2canvas 库内部处理时，对图片资源仍会做一次加载请求；如果此时加载失败，那么该部分保存快照后即是空白的。

下面介绍图片资源转 Blob 的方案，保证图片的地址来自本地，避免在快照转化时加载失败的情况。这里提到的 Blob 对象表示一个不可变、代表二进制原始数据的类文件对象，在特定的使用场景会使用到。

**图片资源转 Blob：**

```
// 返回图片Blob地址
const toBlobURL = (function () {
    const urlMap = {};
    // @param {string} url 传入图片资源地址
```

```
    return function (url) {
        // 过滤重复值
        if (urlMap[url]) return Promise.resolve(urlMap[url]);
        return new Promise((resolve, reject) => {
            const canvas = document.createElement('canvas');
            const ctx = canvas.getContext('2d');
            const img = document.createElement('img');
            img.src = url;
            img.onload = () => {
                canvas.width = img.width;
                canvas.height = img.height;
                ctx.drawImage(img, 0, 0);
                // 关键👇
                canvas.toBlob((blob) => {
                    const blobURL = URL.createObjectURL(blob);
                    resolve(blobURL);
                });
            };
            img.onerror = (e) => {
                reject(e);
            };
        });
    };
}());
```

以上 `toBlobURL` 方法实现将加载 `<img>` 的资源链接转为 blobURL。

进一步地，通过 `convertToBlobImage` 方法，实现对于传入的目标节点中的 `<img>` 批量处理为 `Blob` 格式。

```
// 批量处理
function convertToBlobImage(targetNode, timeout) {
    if (!targetNode) return Promise.resolve();
    let nodeList = targetNode;
    if (targetNode instanceof Element) {
        if (targetNode.tagName.toLowerCase() === 'img') {
            nodeList = [targetNode];
        } else {
            nodeList = targetNode.getElementsByTagName('img');
        }
    } else if (!(nodeList instanceof Array) && !(nodeList instanceof NodeList)) {
        throw new Error('[convertToBlobImage] 必须是Element或NodeList类型');
    }
```

```
    if (nodeList.length === 0) return Promise.resolve();
    // 仅考虑<img>
    return new Promise((resolve) => {
        let resolved = false;
        // 超时处理
        if (timeout) {
            setTimeout(() => {
                if (!resolved) resolve();
                resolved = true;
            }, timeout);
        }
        let count = 0;
        // 逐一替换<img>资源地址
        for (let i = 0, len = nodeList.length; i < len; ++i) {
            const v = nodeList[i];
            let p = Promise.resolve();
            if (v.tagName.toLowerCase() === 'img') {
                p = toBlobURL(v.src).then((blob) => {
                    v.src = blob;
                });
            }
            p.finally(() => {
                if (++count === nodeList.length && !resolved) resolve();
            });
        }
    });
}
export default convertToBlobImage;
```

使用方面，`convertToBlobImage` 应在调用生成快照 `convertToImage` 方法前执行。

### 5.1.3 滚动问题

- 典型特征：生成快照的顶部存在空白区域。
- 原因：一般是保存长图（超过一屏），并且滚动条不在顶部时导致（常见于 SPA 类应用）。
- 解决办法：在调用 `convertToImage` 之前，先记录此时的 `scrollTop`，然后调用 `window.scroll(0, 0)` 将页面移动至顶部。待快照生成后，再调用 `window.scroll(0, scrollTop)` 恢复原有纵向偏移量。

**示例**：

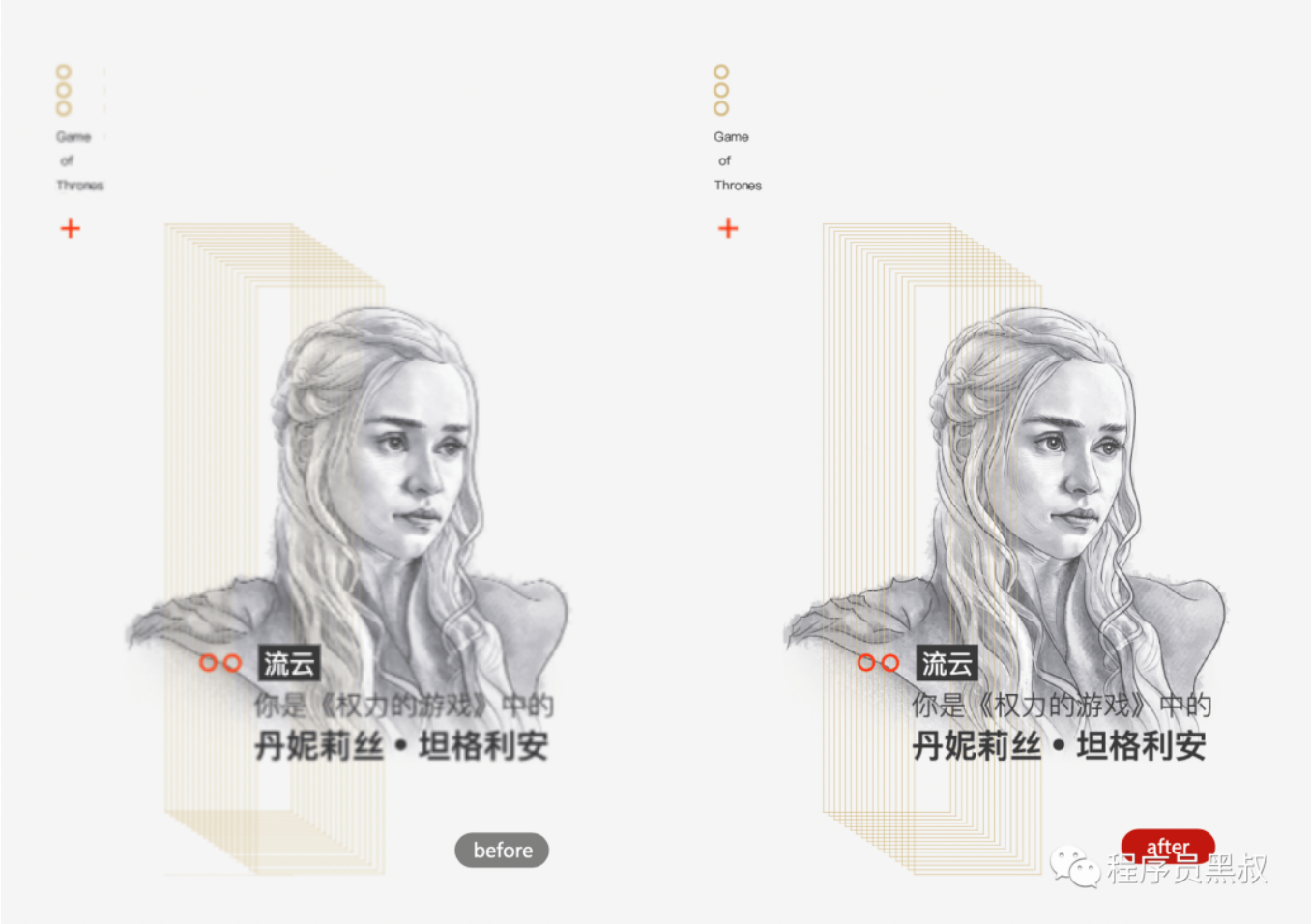
```
// 待保存的目标节点（按实际修改👇）
const container = document.body;
// 实际的滚动元素（按实际修改👇）
const scrollElement = document.documentElement;
// 记录滚动元素纵向偏移量
const scrollTop = scrollElement.scrollTop;

// 针对滚动元素是 body 先作置顶
window.scroll(0, 0);
convertToImage(container)
    .then(() => {
        // ...
    }).catch(() => {
        // ...
    }).finally(() => {
        // 恢复偏移量
        window.scroll(0, scrollTop);
    });
```

特别地，对于存在**局部滚动**布局的情况，也可以操作对应滚动元素置顶避免容器顶部空白的情况。

## 5.2 清晰度优化

> 清晰度是快照质量的分水岭

下图取自「权力的游戏」中两张优化前后的结果页快照对比。可以看到优化前的左图，无论是在文字边缘还是图像细节上，相较优化后的清晰度存在明显可辨的差距。




clear

最终生成快照的清晰度，源头上取决于第一步中 DOM 转换成的 canvas 的清晰度。

以下介绍 5 种行之有效的清晰度优化方法。

### 5.2.1 使用px单位

为了给到 `html2canvas` 明确的整数计算值，避免因小数舍入而导致的拉伸模糊，建议将布局中使用中使用 `%` 、 `vw` 、 `vh` 或 `rem` 等单位的元素样式，统一改为使用 `px` 。

**good:**

```
<div style="width: 100px;"></div>
```

**bad:**

```
<div style="width: 30%;"></div>
```

### 5.2.2 优先使用 img 标签展示图片

很多情况下，导出图片模糊是由原视图中的图片是以 css 中 background 的方式显示的。因为 background-size 并不会反馈一个具体的宽高数值，而是通过枚举值如 contain、cover 等代表图片缩放的类型；相对于 `<img>` 标签， background 方式最终生成的图片会较为模糊。将 background 改为 `<img>` 方式呈现，对于图片清晰度会有一定的改观。对于必须要使用 background 的场景，参见 5.25 节的解决方案。

**good:**

```
<img class="u-image" src="./music.png" alt="icon">
```

**bad:**

```
<div class="u-image" style="background: url(./music.png);"></div>
```

### 5.2.3 配置高倍的 canvas 画布

对于高分辨率的屏幕，canvas 可通过将 css 像素与高分屏的物理像素对齐，实现一定程度的清晰度提升（这里对两类像素有详细描述和讨论）。

在具体操作中，创建由 devicePixelRatio 放大的图像，然后使用 css 将其缩小相同的倍数，有效地提高绘制到 canvas 中的图像清晰度表现。

在使用 `html2canvas` 时，我们可以配置一个放缩后的 canvas 画布用于导入节点的绘制。

```
// convertToImage.js
import html2canvas from 'html2canvas';
// 创建用于绘制的基础canvas画布
function createBaseCanvas(scale) {
    const canvas = document.createElement("canvas");
    canvas.width = width * scale;
    canvas.height = height * scale;
    canvas.getContext("2d").scale(scale, scale);
    return canvas;
}
// 生成快照
function convertToImage(container, options = {}) {
    // 设置放大倍数
    const scale = window.devicePixelRatio;
    // 创建用于绘制的基础canvas画布
    const canvas = createBaseCanvas(scale);
```

```
    // 传入节点原始宽高
    const width = container.offsetWidth;
    const height = container.offsetHeight;
    // html2canvas配置项
    const ops = {
        scale,
        width,
        height,
        canvas,
        useCORS: true,
        allowTaint: false,
        ...options
    };
    return html2canvas(container, ops).then(canvas => {
        const imageEl = Canvas2Image.convertToPNG(canvas, canvas.width, canvas.height);
        return imageEl;
    });
}
```

### 5.2.4 关闭抗锯齿

imageSmoothingEnabled 是 `Canvas 2D API` 用来设置图片是否平滑的属性，`true` 表示图片平滑（默认值），`false` 表示关闭 canvas 抗锯齿。

默认情况下，canvas 的抗锯齿是开启的，可以通过关闭抗锯齿来实现一定程度上的图像锐化，提高线条边缘的清晰度。

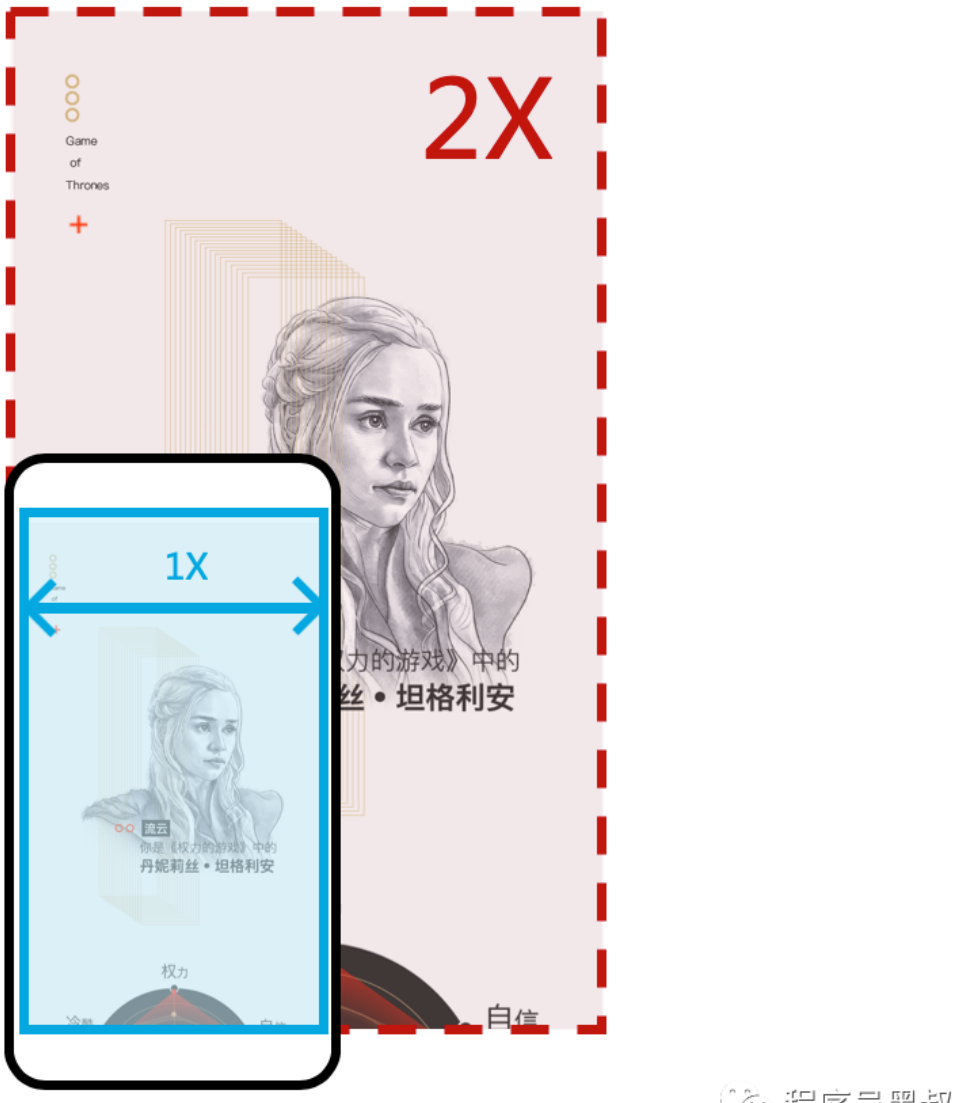
据此，我们将以上 `createBaseCanvas` 方法升级为：

```
// 创建用于绘制的基础canvas画布
function createBaseCanvas(scale) {
    const canvas = document.createElement("canvas");
    canvas.width = width * scale;
    canvas.height = height * scale;
    const context = canvas.getContext("2d");
    // 关闭抗锯齿
    context.mozImageSmoothingEnabled = false;
    context.webkitImageSmoothingEnabled = false;
    context.msImageSmoothingEnabled = false;
    context.imageSmoothingEnabled = false;
    context.scale(scale, scale);
```

```
    context.scale(scale, scale);
    return canvas;
}
```

### 5.2.5 锐化特定元素

受到 canvas 画布放缩的启发，我们对特定的 DOM 元素也可以采用类似的优化操作，即设置待优化元素宽高设置为 2 倍或 `devicePixelRatio` 倍，然后通过 css 缩放的方式控制其展示大小不变。




scale

例如，对于必须用背景图 `background` 的元素，采用以下方式可明显提高快照的清晰度：

```
.box {
    background: url(/path/to/image) no-repeat;
    width: 100px;
```

```
    height: 100px;
    transform: scale(0.5);
    transform-origin: 0 0;
}
```

其中，`width`和`height`为实际显示宽高的 2 倍值，通过`transform: scale(0.5)`实现了元素大小的缩放，`transform-origin`根据实际情况设置。

## 5.3 转换效率

快照的转换效率直接关系到用户的等待时长。我们可以在目标节点传入阶段和快照导出两个阶段对其进行一定优化。

### 5.3.1 传入阶段

> 传入节点的视图信息越精简，生成快照处理的计算量就越小

以下方式适用于传入视图信息“瘦身”：

- 减少 DOM 规模，降低`html2canvas`递归遍历的计算量。
- 压缩图片素材本身的体积，使用 tinypng 或 ImageOptim 等工具压缩素材。
- 如果使用了自定义字体，请使用 fontmin 工具对文字进行按需裁剪，避免动辄数兆的无效资源引入。
- 传入合适的`scale`值以缩放 canvas 画布（5.2.3节）。通常情况下 2~3 倍就已经满足一般的场景，不必要传入过大的放大倍数。
- 在 5.1.2 节中提到的图片资源转 blob，可将图片资源本地化，避免了生成快照时 html2canvas 的二次图片加载处理，同时所生成的资源链接具备 URL 长度较短等优势。

### 5.3.2 导出优化

`canvas2image`提供了多个 API 用于导出图片信息，上文已有介绍。包括：

- convertToPNG
- convertToJPEG
- convertToGIF

- convertToBMP

不同的导出格式，对于生成快照的文件体积存在较大的影响。通常对于没有透明度展示要求的图片素材，可以使用 `jpeg` 格式的导出。在我们的相关实践中，`jpeg` 相比于 `png` 甚至能够节约 80% 以上的文件体积。

实际场景中的的图片导出格式，按业务需求选用即可。

## 6. 小结

本文基于 `html2canvas` 和 `canvas2image`，从快照的内容完整性、清晰度和转换效率等多个方面，介绍了前端页面生成高质量快照的解决方案。

由于实际应用的复杂性，以上方案可能无法覆盖到每一处具体场景，欢迎大家交流和探讨。





收录于合集 #前端技巧 91

超全面的前端工程化配置指南！

‹ 上一篇 · 超全面的前端工程化能查指南！

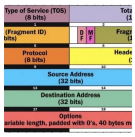
喜欢此内容的人还喜欢

【包真】我的第一次webpack优化，首屏渲染从9s到1s

前端潮咖


最简明的 Tcpdump 抓包入门指南

Coding测试



腾讯一面：如何正确停止一个线程?

Java知音