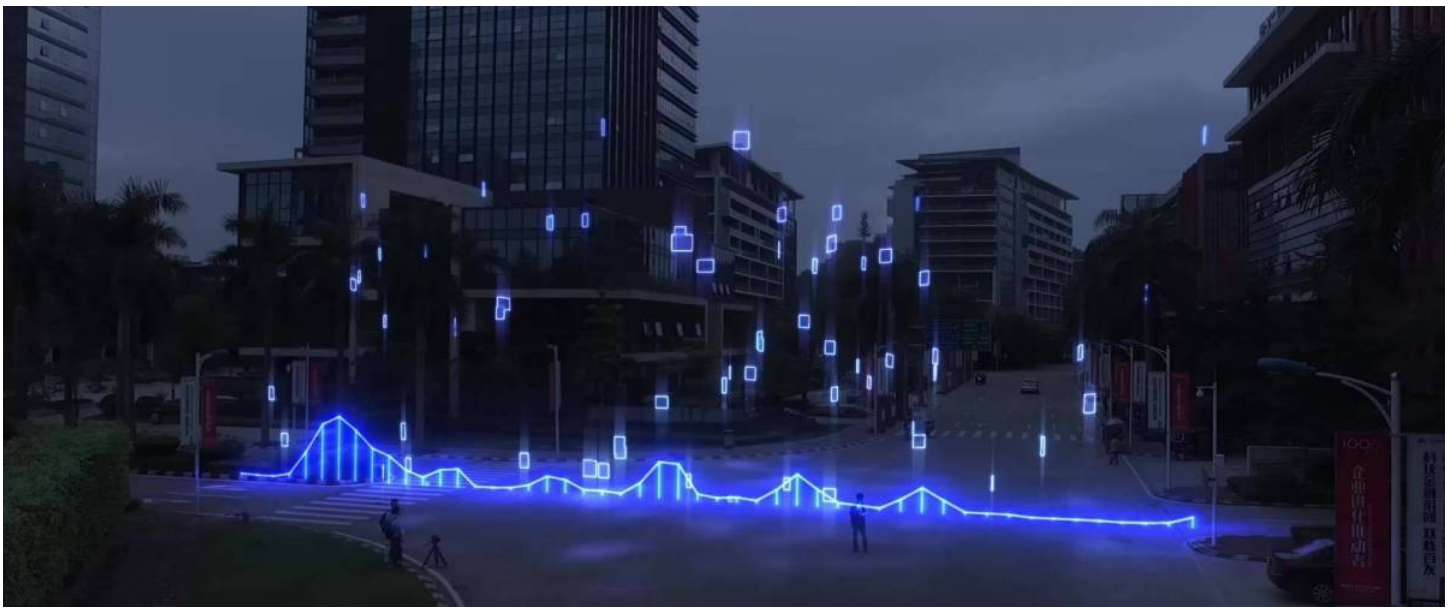


Web Audio在音频可视化中的应用

云音乐前...
已认证的官方帐号

Web Audio在音频可视化中的应用

本文有两个关键词：`音频可视化` 和 `Web Audio` 。前者是实践，后者是其背后的技术支持。 Web Audio 是很大的知识点，本文会将重点放在**如何获取音频数据**这块，对于其 API 的更多内容，可以查看 MDN。

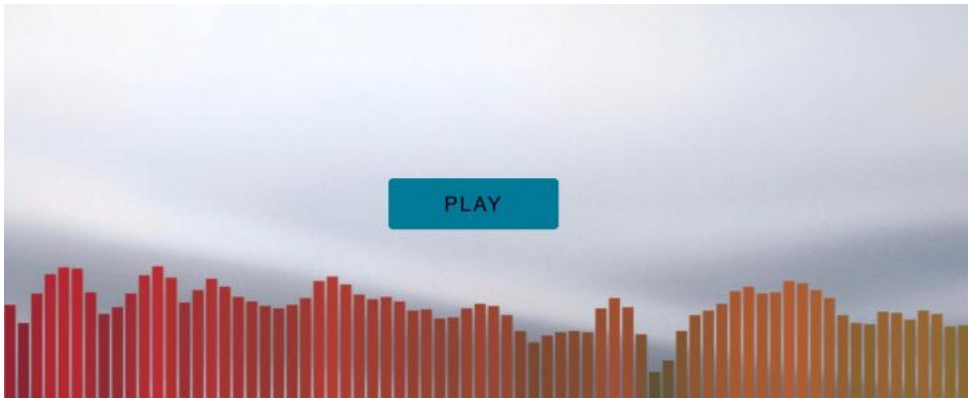
另外，要将音频数据转换成可视化图形，除了了解 Web Audio 之外，还需要对 Canvas （特指2D，下同），甚至 WebGL （可选）有一定了解。如果读者对它们没有任何学习基础，可以先从以下资源入手：

- Canvas Tutorial
- WebGL Tutorial

> 通过获取频率、波形和其他来自声源的数据，将其转换成图形或图像在屏幕上显示出来，再进行交互处理。

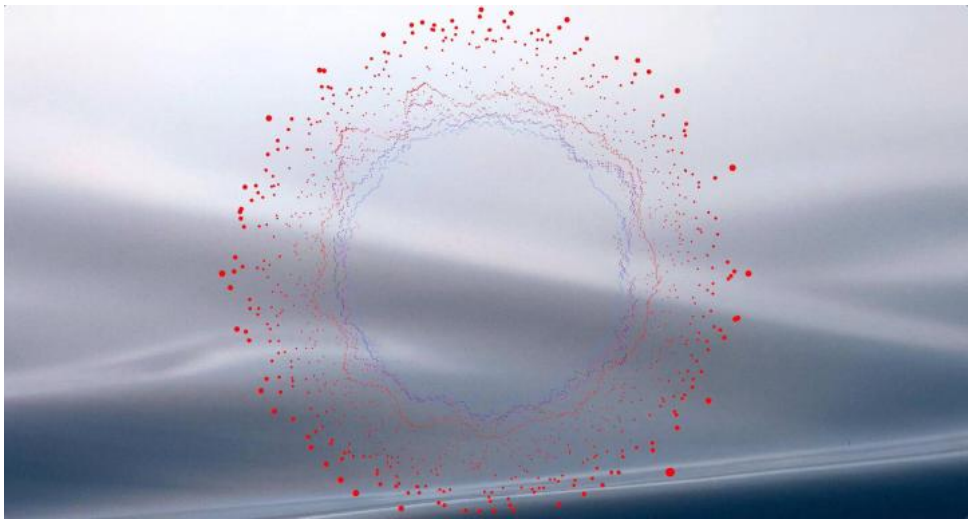
云音乐有不少跟音频动效相关的案例，但其中有些过于复杂，又或者太偏业务。因此这里就现找了两个相对简单，但有代表性的例子。

第一个是用 Canvas 实现的音频柱形图。



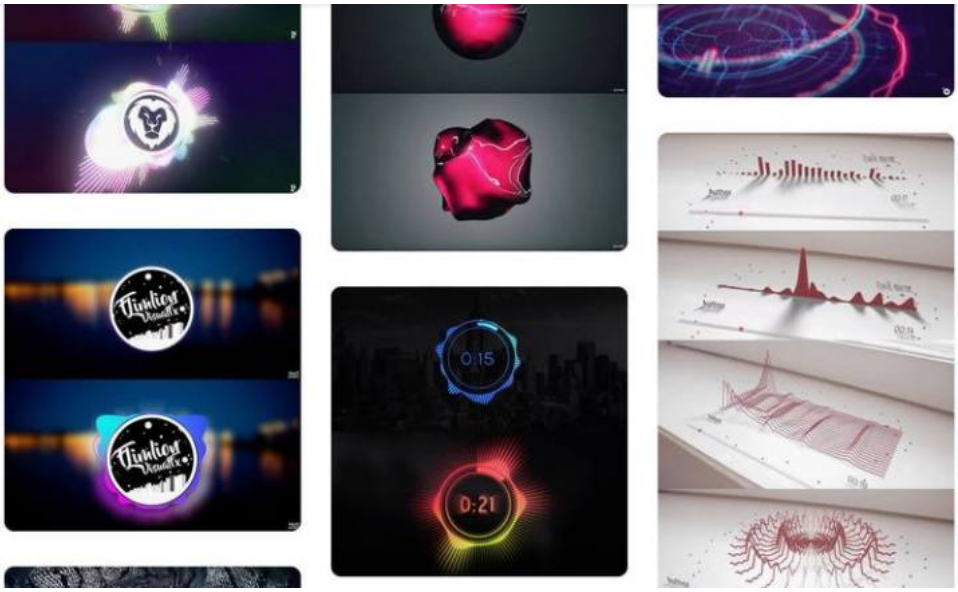
↑点击播放↑

第二个是用 WebGL 实现的粒子效果。



↑点击播放↑

在具体实践中，除了这些基本图形（矩形、圆形等）的变换，还可以把音频和自然运动、3D 图形结合到一起。

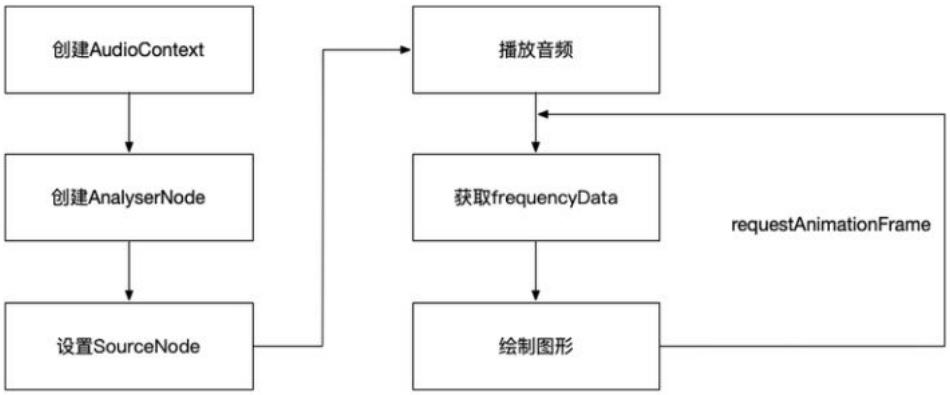


点击查看：pinterest上的一些视觉效果

什么是 Web Audio

> Web Audio 是 Web 端处理和分析音频的一套 API 。它可以设置不同的音频来源（包括 `<audio>` 节点、 ArrayBuffer 、用户设备等），对音频添加音效，生成可视化图形等。

接下来重点介绍 Web Audio 在可视化中扮演的角色，见下图。



简单来说，就是**取数据 + 映射数据**两个过程。我们先把"取数据"这个问题解决，可以按以下5步操作。

1. 创建 AudioContext

在音频的任何操作之前，都必须先创建 AudioContext 。它的作用是关联音频输入，对音频进行解码、控制音频的播放暂停等基础操作。

创建方式如下：

```
const AudioContext = window.AudioContext || window.webkitAudioContext;

const ctx = new AudioContext();
```

AnalyserNode 用于获取音频的频率数据（FrequencyData）和时域数据（TimeDomainData）。从而实现音频的可视化。

它只会对音频进行读取，而不会对音频进行任何改变。

```
const analyser = ctx.createAnalyser();
analyser.fftSize = 512;
```

关于 fftSize，在 MDN 上的介绍可能很难理解，说是快速傅里叶变换的一个参数。

可以从以下角度理解：

1. 它的取值是什么?

fftSize 的要求是 2 的幂次方，比如 256 、 512 等。数字越大，得到的结果越精细。

对于移动端网页来说，本身音频的比特率大多是 128Kbps，没有必要用太大的频率数组去存储本身就不够精细的源数据。另外，手机屏幕的尺寸比桌面端小，因此最终展示图形也不需要每个频率都采到。只需要体现节奏即可，因此 512 是较为合理的值。

2. 它的作用是什么?

fftSize 决定了 frequencyData 的长度，具体为 fftSize 的一半。

至于为什么是 1 / 2，感兴趣的可以看下这篇文章：Why is the FFT “mirrored”?

3. 设置 SourceNode

现在，我们需要将音频节点，关联到 AudioContext 上，作为整个音频分析过程的输入。

在 Web Audio 中，有三种类型的音频源：

- **MediaElementAudioSourceNode** 允许将 `<audio>` 节点直接作为输入，可做到流式播放。
- **AudioBufferSourceNode** 通过 xhr 预先将音频文件加载下来，再用 AudioContext 进行解码。
- **MediaStreamAudioSourceNode** 可以将用户的麦克风作为输入。即通过 `navigator.getUserMedia` 获取用户的音频或视频流后，生成音频源。

这 3 种音频源中，除了 MediaStreamAudioSourceNode 有它不可替代的使用场景（比如语音或视频直播）之外。MediaElementAudioSourceNode 和 AudioBufferSourceNode 相对更容易混用，因此这里着重介绍一下。

MediaElementAudioSourceNode

MediaElementAudioSourceNode 将 `<audio>` 标签作为音频源。它的 API 调用非常简单。

```
// 获取<audio>节点
const audio = document.getElementById('audio');

// 通过<audio>节点创建音频源
const source = ctx.createMediaElementSource(audio);

// 将音频源关联到分析器
source.connect(analyser);

// 将分析器关联到输出设备（耳机、扬声器）
analyser.connect(ctx.destination);
```

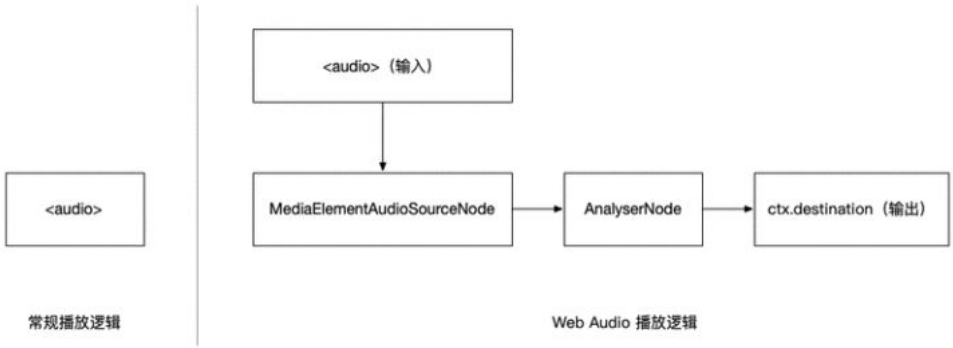
有一种情况是，在安卓端，测试了在 `Chrome/69` （不含）以下的版本，用 MediaElementAudioSourceNode 时，获取到的 frequencyData 是全为 0 的数组。

因此，想要兼容这类机器，就需要换一种预加载的方式，即使用 AudioBufferSourceNode ，加载方式如下：

```
// 创建一个xhr
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/audio.mp3', true);

// 设置响应类型为 arraybuffer
xhr.responseType = 'arraybuffer';

xhr.onload = function() {
    var source = ctx.createBufferSource();

    // 对响应内容进行解码
    ctx.decodeAudioData(xhr.response, function(buffer) {

        // 将解码后得到的值赋给buffer
        source.buffer = buffer;

        // 完成。将source绑定到ctx。也可以连接AnalyserNode
        source.connect(ctx.destination);
    });
};

xhr.send();
```

如果将 AnalyserNode 类比中间件，会不会好理解一些?

可以对比一下常规的 `<audio>` 播放，和 Web Audio 中的播放流程：



4. 播放音频

对于 `<audio>` 节点，即使用 MediaElementAudioSourceNode 的话，播放相对比较熟悉：

```
audio.play();
```

但如果是 AudioBufferSourceNode ，它不存在 play 方法，而是：

```
// 创建AudioBufferSourceNode
const source = ctx.createBufferSource();

// buffer是通过xhr获取的音
```

```
// 调用start方法进行播放
source.start(0);
```

5. 获取 frequencyData

到此，我们已经将音频输入关联到一个 AnalyserNode ，并且开始播放音频。对于 Web Audio 这部分来说，它只剩最后一个任务：获取频率数据。

关于频率， Web Audio 提供了两个相关的 API，分别是：

1. `analyser.getByteFrequencyData`
2. `analyser.getFloatFrequencyData`

两者都是返回 TypedArray ，唯一的区别是精度不同。

getByteFrequencyData 返回的是 0 - 255 的 Uint8Array 。而 getFloatFrequencyData 返回的是 0 - 22050 的 Float32Array 。

相比较而言，如果项目中对性能的要求高于精度，那建议使用 getByteFrequencyData 。下图展示了一个具体例子：

```
Uint8Array(256) [165, 196, 213, 214, 207, 209, 201, 196, 180, 168, 159, 145, 117, 121,
117, 109, 126, 130, 108, 103, 110, 97, 94, 105, 101, 99, 89, 78, 78, 77, 73, 72, 81, 9
2, 90, 79, 76, 69, 66, 65, 65, 71, 73, 74, 72, 72, 77, 76, 68, 62, 63, 58, 59, 58, 60,
59, 58, 57, 55, 48, 53, 59, 64, 61, 57, 58, 60, 57, 57, 60, 60, 59, 61, 58, 51, 51, 5
0, 49, 49, 48, 49, 45, 48, 50, 48, 48, 43, 40, 43, 43, 42, 48, 53, 50, 47, 40, 36, 41,
39, 42, …]
```

关于数组的长度（ 256 ），在上文已经解释过，它是 fftSize 的一半。

现在，我们来看下如何获取频率数组：

```
const bufferLength = analyser.frequencyBinCount;
const dataArray = new Uint8Array(bufferLength);

analyser.getByteFrequencyData(dataArray);
```

需要注意的是， getByteFrequencyData 是对已有的数组元素进行赋值，而不是创建后返回新的数组。

它的好处是，在代码中只会有一个 dataArray 的引用，不用通过函数调用和参数传递的方式来重新取值。

可视化的两种实现方案

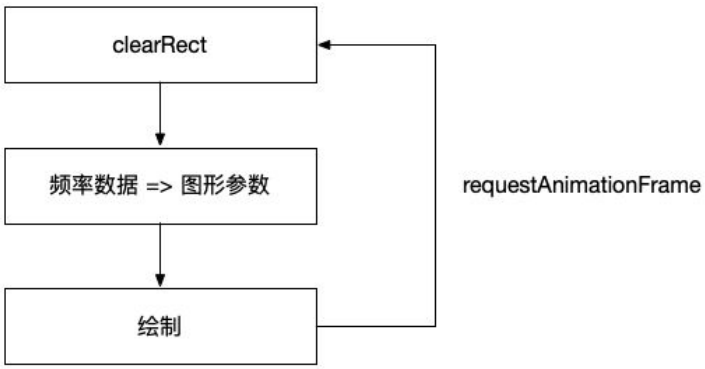
在了解 Web Audio 之后，已经能用 getByteFrequencyData 取到一个 Uint8Array 的数组，暂时命名为 dataArray 。

从原理上讲，可视化所依赖的数据可以是音频，也可以是温度变化，甚至可以是随机数。所以，接下来的内容，我们只需要关心如何将 dataArray 映射为图形数据，不用再考虑 Web Audio 的操作。

（为了简化 Canvas 和 WebGL 的描述，下文提到 Canvas 特指 `Canvas 2D` 。）

1. Canvas 方案

点击查看：第1个示例的源码



以下是从示例代码中摘取的一段：

```
function renderFrame() {
    requestAnimationFrame(renderFrame);

    // 更新频率数据
    analyser.getByteFrequencyData(dataArray);

    // bufferLength表示柱形图中矩形的个数
    for (var i = 0, x = 0; i < bufferLength; i++) {
        // 根据频率映射一个矩形高度
        barHeight = dataArray[i];

        // 根据每个矩形高度映射一个背景色
        var r = barHeight + 25 * (i / bufferLength);
        var g = 250 * (i / bufferLength);
        var b = 50;

        // 绘制一个矩形，并填充背景色
        ctx.fillStyle = "rgb(" + r + "," + g + "," + b + ")";
        ctx.fillRect(x, HEIGHT - barHeight, barWidth, barHeight);

        x += barWidth + 1;
    }
}

renderFrame();
```

对于可视化来说，核心逻辑在于：如何把频率数据映射成图形参数。在上例中，只是简单地改变了柱形图中每一个矩形的高度和颜色。

Canvas 提供了丰富的绘制API，仅从 2D 的角度考虑，它也能实现很多酷炫的效果。类比 DOM 来说，如果只是 `<div>` 的组合就能做出丰富多彩的页面，那么 Canvas 一样可以。

2. WebGL 方案

点击查看：第2个示例的源码

Canvas 是 CPU 计算，对于 for 循环计算 10000 次，而且每一帧都要重复计算， CPU 是负载不了的。所以我们很少看到用 Canvas 2D 去实现粒子效果。取而代之的，是使用 WebGL ，借助 GPU 的计算能力。

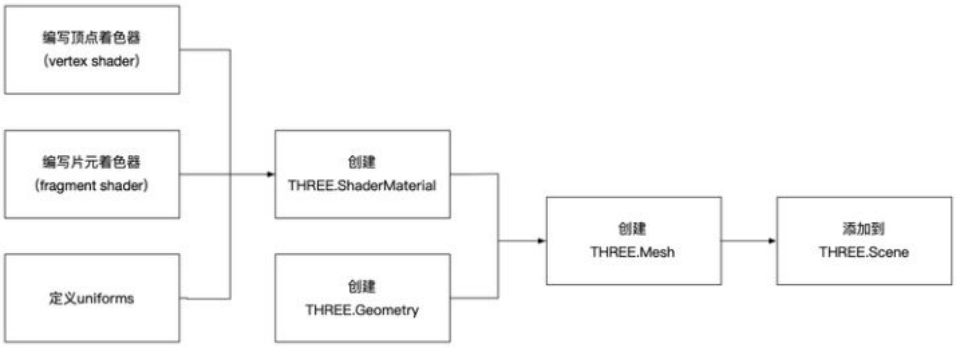
在 WebGL 中，有一个概念相对比较陌生——着色器。它是运行在 GPU 中负责渲染算法的一类总称。它使用 GLSL （ OpenGL S
简单的示例：

```
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

关于着色器更详细的介绍，可以查看这篇文章。

WebGL 的原生 API 是非常复杂的，因此我们使用 `Three.js` 作为基础库，它会让业务逻辑的编写变得简单。

先来看下整个开发流程中做的事情，如下图：



在这个过程中， uniforms 的类型是简单 Object ，我们会将音频数组作为 uniforms 的一个属性，传到着色器中。至于着色器做的事情，可以简单理解为，它将 uniforms 中定义的一系列属性，映射为屏幕上的顶点和颜色。

顶点着色器和片元着色器的编写往往不需要前端开发参与，对于学过 Unity3D 等技术的游戏同学可能会熟悉一些。读者可以到 ShaderToy 上寻找现成的着色器。

然后介绍以下3个 Three.js 中的类：

1. THREE.Geometry

可以理解为形状。也就是说，最后展示的物体是球体、还是长方体、还是其他不规则的形状，是由这个类决定的。

因此，你需要给它传入一些顶

2. THREE.ShaderMaterial

可以理解为颜色。还是以三角形为例，一个三角形可以是黑色、白色、渐变色等，这些颜色是由 ShaderMaterial 决定的。

ShaderMaterial 是 Material 的一种，它由顶点着色器和片元着色器进行定义。

3. THREE.Mesh

定义好物体的形状和颜色后，需要把它们组合在一起，称作 Mesh （网格）。有了 Mesh 之后，便可以将它添加到画布中。然后就是常规的 requestAnimationFrame 的流程。

同样的，我们摘取了示例中比较关键的代码，并做了标注。

i. 创建 Geometry （这是从 THREE.BufferGeometry 继承的类）：

```
var geometry = ParticleBufferGeometry({
    // TODO 一些参数
});
```

ii. 定义 uniforms：

```
var uniforms = {
    dataArray: {
        value: null,
        type: 't' // 对应THREE.DataTexture
    },
    // TODO 其他属性
};
```

iii. 创建 ShaderMaterial：

```
var material = new THREE.ShaderMaterial({
    uniforms: uniforms,
    vertexShader: '', // TODO 传入顶点着色器
    fragmentShader: '', // TODO 传入片元着色器
    // TODO 其他参数
});
```

iv. 创建 Mesh：

```
var mesh = new THREE.Mesh(geometry, material);
```

v. 创建 Three.js 中一些必须的渲染对象，包括场景和摄像头：

```
var scene, camera, renderer;

renderer = new THREE.WebGLRenderer({
    antialias: true,
    alpha: true
});

camera = new THREE.PerspectiveCamera(45, 1, .1, 1e3);

scene = new THREE.Scene();
```

vi. 常规的渲染逻辑：

```
    // TODO 此处可以触发事件，用于更新频率数据

    renderer.render(scene, camera);
}
```

小结

本文首先介绍了如何通过 Web Audio 的相关 API 获取音频的频率数据。

然后介绍了 Canvas 和 WebGL 两种可视化方案，将频率数据映射为图形数据的一些常用方式。

另外，云音乐客户端上线鲸云动效已经有一段时间，看过本文之后，有没有同学想尝试实现一个自己的音频动效呢?

最后附上文中提到的两段 codepen 示例：

1. codepen.io/jchenn/pen/L...
2. codepen.io/jchenn/pen/W...

> 本文发布自 网易云音乐前端团队，文章未经授权禁止任何形式的转载。我们一直在招人，如果你恰好准备换工作，又恰好喜欢云音乐，那就 加入我们!

编辑于 2020-01-09

HTML5 网页应用 可视化

2 条评论 切换为时间排序

写下你的评论...

oodex 5 个月前

App里的鲸云动效也是用WebAudio做的?

赞

陈佳辉 回复 oodex 5 个月前

不是，是客户端原生写的，但在很多H5中使用过WebGL做音频动效。

赞