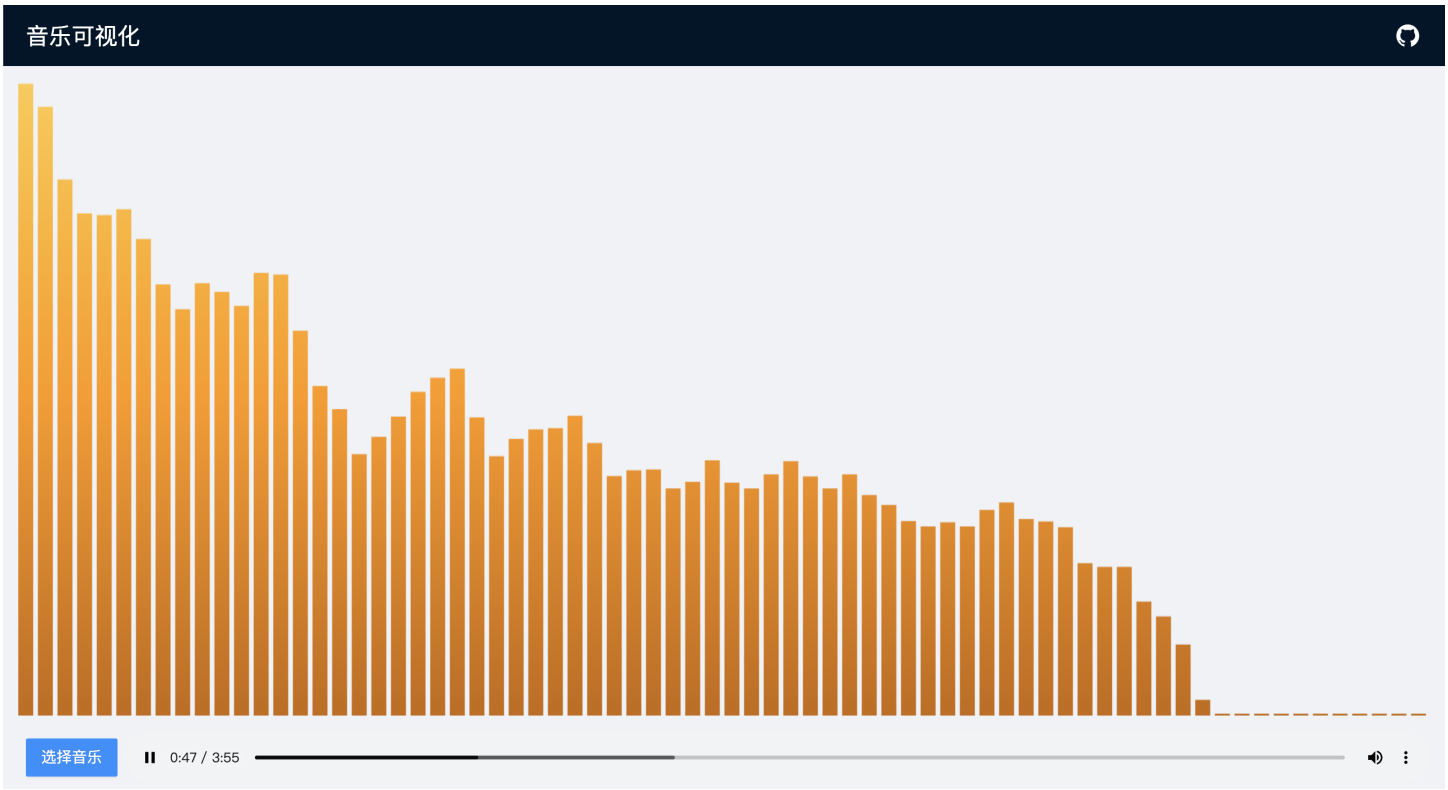
音乐可视化

本项目利用 HTML5 Canvas，实现了对音乐旋律的可视化，使用 TypeScript 编写。



目录

项目报告

1. 程序说明

1.1 线上 Demo

本项目利用 Vercel 部署了一个 线上 Demo，访问可能需要良好的网络环境。因为是线上 Demo，选择好音乐后需要等待音乐加载完毕才能使用。

1.2 本地安装

执行 `./scripts/prebuild.sh` 安装所有依赖，然后执行 `./scripts/build.sh` 以构建本项目。

如果你使用的是 Windows，一种选择是使用 WSL，或者你也可以手动下载安装 Node.js，然后执行以下指令手动安装。

```
npm install -g yarn
yarn global add pm2
yarn && yarn build
```

1.3 如何使用

执行 `./scripts/start.sh` 以启动本地服务器，然后在浏览器打开 http://localhost:7070 即可访问。如果出现端口冲突，可以在 server.mjs 里指定 `listenPort` 为其他可用端口。

如果你使用的是 Windows，则执行以下指令：

```
pm2 start server.mjs --watch
```

启动后，点击左下角的 `选择音乐` 按钮，选择需要可视化的音乐，或者提供可用的 mp3 外链地址（可能存在跨域问题，请选择允许跨域的地址），然后点击播放，即可看到可视化的画面。

执行 `./scripts/stop.sh` 以停止本地服务器。

如果你使用的是 Windows，则执行以下指令：

```
pm2 stop server
```

2. 程序原理

核心代码参见 src/components/MusicVisualizer.vue，下面讲讲程序的主要思路。

2.1 音频分析

首先是初始化音频分析器 `audioAnalyser`，将其绑定到我们的音频源上，并设置一些参数。

```ts
// ./src/components/MusicVisualizer.vue

const { FFT_SIZE, SMOOTHING_TIME_CONST } = ANALYZER;

const audioPlayerRef = ref<HTMLAudioElement>();
const audioSrcUrl = ref<string>();
const audioSrc = ref<MediaElementAudioSourceNode>();
const audioContext = ref<AudioContext>();
const audioAnalyser = ref<AnalyserNode>();

const setupAudioAnalyser = (): void => {
  if (audioPlayerRef.value && !audioContext.value) {
    audioContext.value = new AudioContext({ latencyHint: 'interactive' });

    audioPlayerRef.value.onplay = () => audioContext.value?.resume();
    audioSrc.value = audioContext.value.createMediaElementSource(audioPlayerRef.value);

    audioAnalyser.value = audioContext.value.createAnalyser();
    audioAnalyser.value.smoothingTimeConstant = SMOOTHING_TIME_CONST;
    audioAnalyser.value.fftSize = FFT_SIZE;

    audioSrc.value.connect(audioAnalyser.value);
    audioAnalyser.value.connect(audioContext.value.destination);
  }
};
```

这里我们设定了 `AnalyserNode.smoothingTimeConstant` 的值为 `0.8`，以对一定量的历史响度数据取平均，从而让每帧之间的过渡平滑一点。这个值的选取范围为 `[0, 1]` 内的浮点数，`0` 表示仅考虑当前一帧，不进行任何平均化处理。

对于每一帧的音频，我们利用函数 `AnalyserNode.getByteFrequencyData()` 按频率将 `[0, context.sampleRate / 2]` Hz 范围的音频**线性**地切割成 `FFT_SIZE / 2` 个频域，同时得到每个频域上音频的响度（默认范围为 `[-100, -30]` Db）**线性**映射到整数域 `[0, 255]` 上的值。其中 `context.sampleRate` 的默认值取决于外放设备，通常是 `44100`，`FFT_SIZE` 的值在本项目中是 `256`。

```ts
// ./src/components/MusicVisualizer.vue

const spectrum = new Uint8Array(audioAnalyser.value.frequencyBinCount);
audioAnalyser.value.getByteFrequencyData(spectrum);
```

也就是说，我们在这步会得到一个长度为 `128` 的数组 `spectrum`，数组中的值均为 `[0, 255]` 范围内的整数，其中 `0` 表示 `-100` Db，`255` 表示 `-30` Db，中间的值为线性映射。而数组索引则表示频域的范围，例如索引 `0` 就表示 `[0, 44100 / 2 / 128 = 172.27]` Hz 的频域，索引 `i` 就表示 `[172.27 i, 172.27 (i + 1)]` Hz 的频域。

当然，懂乐理的朋友肯定知道，这个分割方式显然是不合理的。因为音频的音高和频率是呈对数关系而不是线性关系，音调每高一个八度，频率翻一倍。因此正确的分割方式应该是按指数关系切割，低频的索引对应的频域小一点，高频的索引对应的频域大一点，这样得到的可视化结果才是根据音调的。但 JS 提供的音频相关 API 就是这样设计的，你也没什么办法。

我尝试过将这个线性切割的数组重新映射到指数切割，但最后发现计算实在是比较复杂。主要难点在于要将不同频域的响度重新合并（不合并而是单纯采样的话，得到的值会非常小），但合并当然不能简单线性相加或者指数相

加，而是要先将这个 `[0, 255]` 范围的整数映射到 `[-100, -30]` Db 范围的原始**响度**（当然，也可以利用函数 `AnalyserNode.getFloatFrequencyData()` 直接得到原始响度，但最后还是要映射回去），然后将这个响度按指数关系转化为原始的**能量**大小，将指定范围内的能量相加得到总能量，再转化回响度，最后映射到一个正的线性范围。为了精度还要提高 `FFT_SIZE` 的值，至少要提高到 `2048`，因此数组的长度也变成了 `1024`。而且这样的转化每帧都要做一次，确实是带来了很大的不必要的性能开销，想想还是算了。

接下来，我们对这个数组 `spectrum` 进行可视化，将每个数据点转化为相应的图形。

2.2 Canvas 渲染

由于期末时间过于紧张，这里我们就简单生成一个柱状图。

```vue
// ./src/components/MusicVisualizer.vue

const render = (): void => {
  if (canvasRef.value && canvasContext.value && audioAnalyser.value) {
    const { clientWidth: width, clientHeight: height } = canvasRef.value;
    canvasRef.value.width = width;
    canvasRef.value.height = height;
    setFillStyle();

    const spectrum = new Uint8Array(audioAnalyser.value.frequencyBinCount);
    audioAnalyser.value.getByteFrequencyData(spectrum);

    const barCount = Math.floor(width / (BAR_WIDTH + BAR_GAP));
    const step = spectrum.length / barCount;
    const scale = height / FFT_SIZE;
    for (let i = 0; i < barCount; i += 1) {
      const barValue = average(Array.from(spectrum), Math.floor(i * step), Math.ceil((i + 1) * step));
      const barHeight = Math.max(barValue * scale, MIN_HEIGHT);
      canvasContext.value.fillRect(i * (BAR_WIDTH + BAR_GAP), height - barHeight, BAR_WIDTH, barHeight);
    }

    requestAnimationFrame(render);
  }
}
```

首先我们在页面上生成一个铺满窗口的 canvas，然后根据这个 canvas 的宽度 `width` 和高度 `height` 决定生成多少个柱形、每个柱形的高度缩放比和在 canvas 中的位置。这里我们设置每个柱形的宽度为 `BAR_WIDTH`（此处为 `16`）、间隔为 `BAR_GAP`（此处为 `5`）、最小高度为 `MIN_HEIGHT`（此处为 `2`）。

于是我们就可以算出 canvas 里可以平铺的柱形数量 `barCount`

$$\text{bar\_count} = \left\lfloor \frac{\text{width}}{\text{bar\_width} + \text{bar\_gap}} \right\rfloor$$

和每个柱形的高度缩放比 `scale`

$$\text{scale} = \frac{\text{height}}{\text{fft\_size}}$$

对于第 i 个柱形，我们根据 `barCount` 按线性比例划分其表示的频域范围 $[i \cdot \text{step}, (i+1) \cdot \text{step})$，其中

$$\text{step} = \frac{|\text{spectrum}|}{\text{bar\_count}}$$

然后我们算出这个频域范围内的响度均值 `barValue`

$$\text{bar\_value} = \frac{1}{\lceil \text{step} \rceil} \sum_{k=\lfloor i \cdot \text{step} \rfloor}^{\lceil (i+1) \cdot \text{step} \rceil} \text{spectrum}_k$$

那么这个柱形的高度 `barHeight` 就可以设定为 `barValue` 乘以高度缩放比 `scale`。当这个频域内的响度达到最大值时，柱形就可以有 100% 的 canvas 高度。

柱形的位置即为

$$(x,\, y) = (i \cdot (\text{bar\_width} + \text{bar\_gap}),\, \text{height} - \text{bar\_height})$$

其中原点为 canvas 的**左上角**。

如此依次渲染所有的柱形后，我们就得到了这一帧的柱状图。

这里我们调用函数 `setFillStyle()` 美化了一下输出，设定柱形的填充色为渐变的橙色。

```
// ./src/components/MusicVisualizer.vue

const setFillStyle = (): void => {
  if (canvasRef.value && canvasContext.value) {
    const gradient = canvasContext.value.createLinearGradient(0, 0, 0, canvasRef.value.height);
    gradient.addColorStop(0, '#ffc947');
    gradient.addColorStop(0.5, '#ff9800');
    gradient.addColorStop(1, '#c66900');
    canvasContext.value.fillStyle = gradient;
  }
};
```

最后我们利用函数 `requestAnimationFrame(render)`，每秒调用 60 次这个渲染函数 `render()`，就得到了连贯的 60 fps 可视化动画。

贡献者

- **陈泓宜（18307130003）**<i@hakula.xyz> - 复旦大学

许可协议