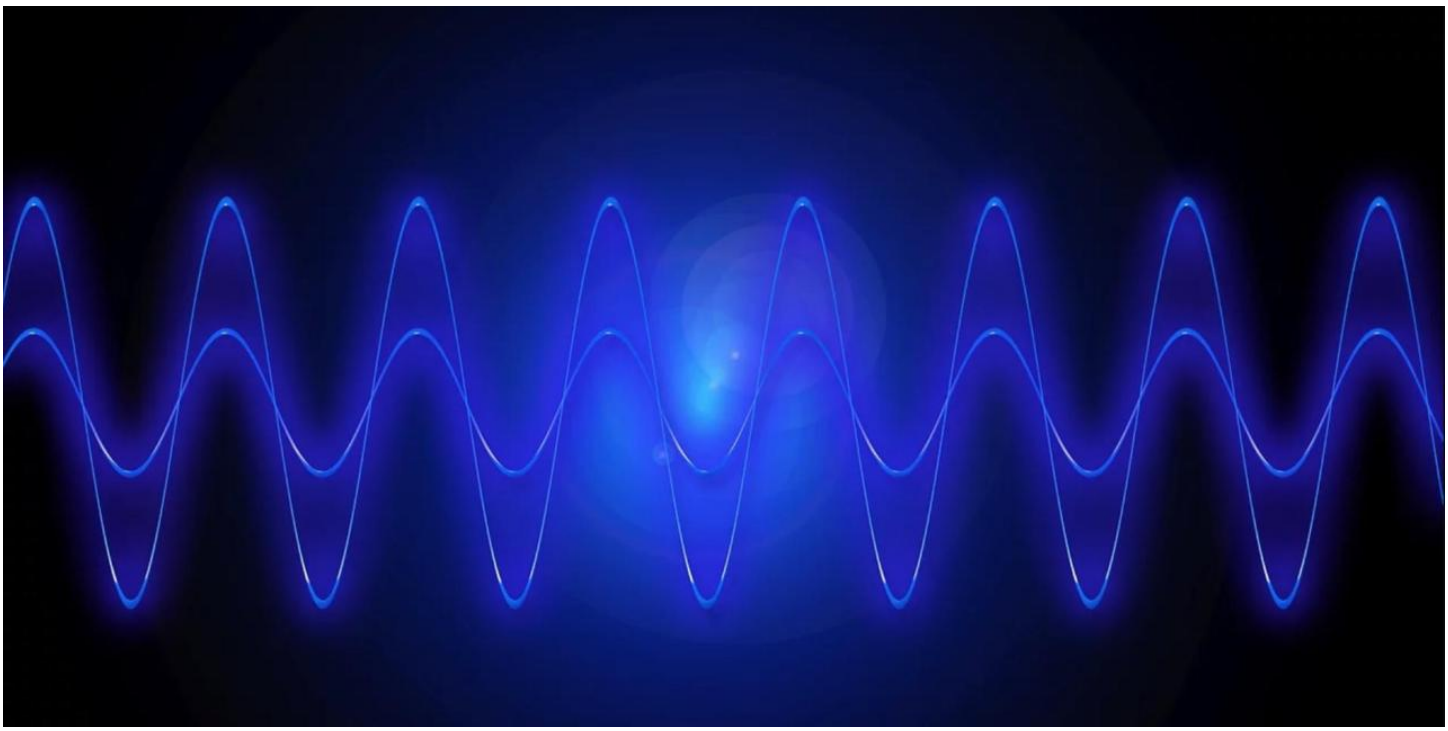


用 Web 实现一个简易的音频编辑器

云音乐前... 已认证的官方帐号 + 关注他

前言

市面上，音频编辑软件非常多，比如 cubase、sonar 等等。虽然它们功能强大，但是在 Web 上的应用却显得心有余而力不足。因为 Web 应用的大多数资源都是存放在网络服务器中的，用 cubase 这些软件，首先要把音频文件下载下来，修改完之后再上传到服务器，最后还要作更新操作，操作效率极其低下。如果能让音频直接在 Web 端进行编辑并更新到服务器，则可以大大提高运营人员的工作效率。下面就为大家介绍一下如何运用 Web 技术实现高性能的音频编辑器。

本篇文章总共分为 3 章：

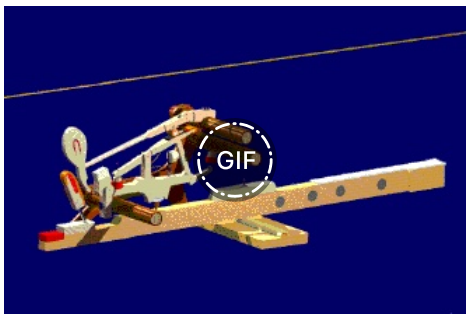
- 第 1 章：声音相关的理论知识
- 第 2 章：音频编辑器的实现方法
- 第 3 章：音频编辑器的性能优化

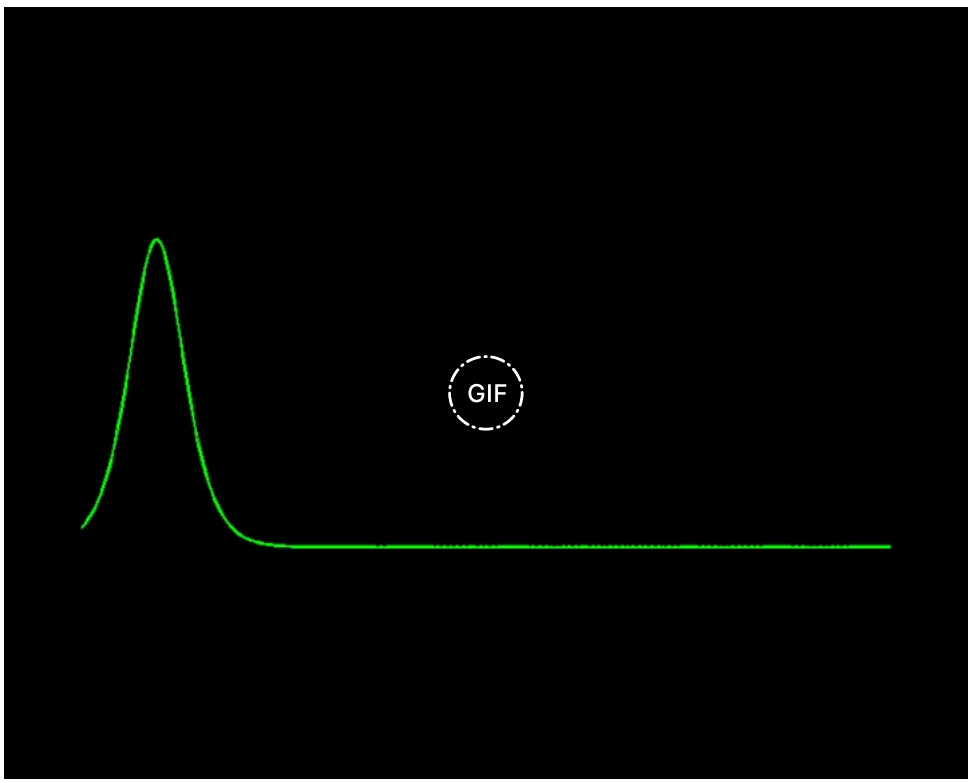
第 1 章 - 声音相关的理论知识

理论是实践的依据和根基，了解理论可以更好的帮助我们实践，解决实践中遇到的问题。

1.1 什么是声音

物体振动时激励着它周围的空气质点振动，由于空气具有可压缩性，在质点的相互作用下，振动物体四周的空气就交替地产生压缩与膨胀，并且逐渐向外传播，从而形成声波。声波通过介质（空气、固体、液体）传入到人耳中，带动听小骨振动，经过一系列的神经信号传递后，被人所感知，形成声音。我们之所以能听到钢琴、二胡、大喇叭等乐器发出的声音，就是因为乐器里的某些部件通过振动产生声波，





1.2 声音的因素

为什么人们的声音都不一样，为什么有些人的声音很好听，有些人的声音却很猥琐呢？这节介绍一下声音的 3 大因素：频率、振幅和音色，了解这些因素之后大家就知道原因了。

1.2.1 频率

声音既然是声波，就会有振幅和频率。频率越大音高越高，声音就会越尖锐，比如女士的声音频率就普遍比男士的大，所以她们的声音会比较尖锐。人的耳朵通常只能听到 20Hz 到 20kHz 频率范围内的声波。

1.2.2 振幅

声波在空气中传播时，途经的空气会交替压缩和膨胀，从而引起大气压强变化。振幅越大，大气压强变化越大，人耳听到的声波就会越响。人耳可听的声压（声压：声波引起的大气压强变化值）范围为 (2 * 10 ^ - 5)Pa~20Pa，对应的分贝数为 0~120dB。它们之间的换算公式为 `20 * log( X / (2 * 10 ^ -5) )`，其中 X 为声压。相比较用大气压强来表示声音振幅强度，用分贝表示会更加直观。我们平时在形容物体的声音强度时，一般也都会用分贝，而不会说这个大喇叭发出了多少多少帕斯卡的声压（但听起来好像很厉害得样子）。

1.2.3 音色

频率和振幅都不是决定一个人声音是猥琐还是动听的主要因素，决定声音是否好听的主要因素为音色，音色是由声波中的谐波决定的。自然界中物体振动产生的声波，都不是单一频率单一振幅的波（如正弦波），而是可以分解为 1 个基波加上无数个谐波。基波和谐波都是正弦波，其中谐波的频率是基波的整数倍，振幅比基波小，相位也各不相同。如钢琴中央 dou，它的基波频率为 261，其他无数个谐波频率为 261 的整数倍。声音好听的人，在发声时，声带产生的谐波比较“好听”，而声音猥琐的人，声带产生的谐波比较“猥琐”。

1.3 声音的录制、编辑、回放

不管是欧美的钢琴、小提琴，还是中国的唢呐、二胡、大喇叭，我们不可能想听的时候都叫演奏家们去为我们现场演奏，如果能将这些好听声音存储起来，我们就可以在想听的时候进行回放了。传统的声音录制方法是

到磁带或传至音箱等设备发声。这种方法失真较大, 且消除噪音困难, 也不易被编辑和修改，数字化技术可以帮我们解决模拟电流带来的问题。这节我们就来介绍下数字化技术是如何做到的。

1.3.1 录制

声音是一段连续无规则的声波，由无数个正弦波组成。数字化录制过程就是采集这段声波中离散的点的幅值，量化和编码后存储在计算机中。整个过程的基本原理为：声音经过麦克风后根据振幅的不同形成一段连续的电压变化信号，这时用脉冲信号采集到离散的电压变化，最后将这些采集到的结果进行量化和编码后存储到计算机中。采样脉冲频率一般为 44.1kHz，这是因为人耳一般只能听到声波中 20-20kHz 频率正弦波部分，根据采样定律，要从采样值序列完全恢复原始的波形，采样频率必须大于或等于原始信号最高频率的 2 倍。因此，如果要保留原始声波中 20kHz 以内的所有正弦波，采样频率一定要大于等于 40kHz。

1.3.2 编辑

声音数字化后就可以非常方便的对声音进行编辑，如展示声音波形图，截取音频，添加静音效果、渐入淡出效果，通过离散型傅里叶变换查看声音频谱图（各个谐波的分布图）或者进行滤波操作（滤除不想要的谐波部分），这些看似复杂的操作却只需要对量化后的数据简单进行的计算即可实现。

1.3.3 回放

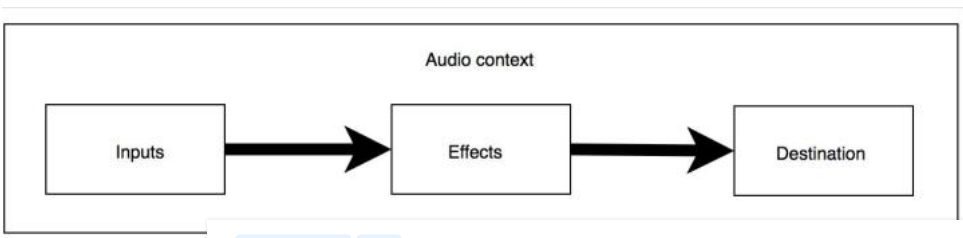
回放过程就是录制过程的逆过程，将录制或者编辑过的音频数据进行解码，去量化还原成离散的电压信号送入大喇叭中。大喇叭如何将电压信号还原成具体的声波振幅，这个没有深入学习，只能到这了。

第2章-音频编辑器的实现方法

通过第 1 章的理论知识，我们知道了什么是声音以及声音的录制和回放，其中录制保存下来的声音数据就叫音频，通过编辑音频数据就能得到我们想要的回放声音效果。这章我们就开始介绍如何用浏览器实现音频编辑工具。浏览器提供了 AudioContext 对象用于处理音频数据，本章首先会介绍下 AudioContext 的基本使用方法，然后介绍如何用 svg 绘制音频波形以及如何对音频数据进行编辑。

2.1 AudioContext 介绍

AudioContext 对音频数据处理过程是一个流式处理过程，从音频数据获取、数据加工、音频数据播放，一步一步流式进行。AudioContext 对象则提供流式加工所需要的方法和属性，如 context.createBufferSource 方法返回一个音频数据缓存节点用于存储音频数据，这是整个流式的起点；context.destination 属性为整个流式的终点，用于播放音频。每个方法都会返回一个 AudioNode 节点对象，通过 AudioNode.connect 方法将所有 AudioNode 节点连接起来。



下面通过一个简单的例子来解锁 AudioContext： - 为了方便起见，我们不使用服务器上的音频文件，而使用 FileReader 读取本地音频文件 - 使用 AudioContext 的 decodeAudioData 方法对读到的音频数据进行解码 - 使用 AudioContext 的 createBufferSource 方法创建音频源节点，并将解码结果赋值给它 - 使用 AudioContext 的 connect 方法连接音频源节点到播放终端节点 - AudioContext 的 destination 属性 - 使用 AudioContext 的 start 方法开始播放

```
// 读取音频文件.mp3 .flac .wav等等
    const reader = new FileReader();
    // file 为读取到的文件，可以通过<input type="file" />实现
    reader.readAsArrayBuffer(file);
    reader.onload = evt => {
        // 编码过的音频数据
        const encodedBuffer = evt.currentTarget.result;
        // 下面开始处理读取到的音频数据
        // 创建环境对象
        const context = new AudioContext();
        // 解码
        context.decodeAudioData(encodedBuffer, decodedBuffer => {
            // 创建数据缓存节点
            const dataSource = context.createBufferSource();
            // 加载缓存
            dataSource.buffer = decodedBuffer;
            // 连接播放器节点destination，中间可以连接其他节点，比如音量调节节点createGa
            // 频率分析节点（用于傅里叶变换）createAnalyser()等等
            dataSource.connect(context.destination);
            // 开始播放
            dataSource.start();
        })
    }
```

2.1 什么是音频波形

音频编辑器通过音频波形图形化音频数据，使用者只要编辑音频波形就能得到对应的音频数据，当然内部实现是将对波形的操作转为对音频数据的操作。所谓音频波形，就是时域上，音频（声波）振幅随着时间的变化情况，即 X 轴为时间，Y 轴为振幅。

2.2 绘制波形

我们知道，音频的采样频率为 44.1kHz，所以一段 10 分钟的音频总共会有 10 * 60 * 44100 = 26460000，超过 2500 万个数据点。 我们在绘制波形时，即使仅用 1 个像素代表 1 个点的振幅，波形的宽度也将近 2500 万像素，不仅绘制速度慢，而且非常不利于波形分析。 因此，下面介绍一种近似算法来减少绘制的像素点：我们首先将每秒钟采集的 44100 个点平均分成 100 份，相当于 10 毫秒一份，每一份有 441 个点， 算出它们的最大值和最小值。用最大值代表波峰，用最小值代表波谷，然后用线连接所有的波峰和波谷。音频数据在被量化后，值的范围为 [-1,1]， 所以我们这里取到的波峰波谷都是在 [-1,1] 的区间内的。 由于数值太小，画出来的波形不美观，我们统一将这些值乘以一个系数比如 64，这样就能很清晰得观察到波形的变化了。 绘制波形可以用 canvas，也可以用 svg，这里我选择使用 svg 进行绘制，因为 svg 是矢量图，可以简化波形缩放算法。

代码实现

- 为了方便使用 svg 进行绘制，引入 svg.js，并初始化 svg 对象 draw
- 我们的绘制算法是将每秒钟采集的 44100 个点平均分成 100 份，每份是10毫秒共441个数据点，用它们的最大值和最小值作为这个时间点的波峰和波谷。 然后使用svg.js将所有的波峰波谷通过折线 polyline 连接起来形成最后的波形图。由于音频数据点经过量化处理，范围为[-1,1]，为了让波形更加美观，我
- 初始化变量 perSec

- 以10毫秒为单位获取所有的波峰波谷数据点 peaks，计算方法就是简单得计算出它们各自的最大值和最小值
- 初始化波形图的宽度 svgWidth = 音频时长（buff.duration）* 每秒钟绘制像素点的个数（perSecPx）
- 遍历 peaks，将所有的波峰波谷乘上系数并通过 polyline（折线）连接起来

```
const SVG = require('svg.js');
// 创建svg对象
const draw = SVG(document.getElementById('draw'));
// 波形svg对象
let polyline;
// 波形宽度
let svgWidth;
// 展示波形函数
// buffer – 解码后的音频数据
function displayBuffer(buff) {
    // 每秒绘制100个点，就是将每秒44100个点分成100份，
    // 每一份算出最大值和最小值来代表每10毫秒内的波峰和波谷
    const perSecPx = 100;
    // 波峰波谷增幅系数
    const height = 128;
    const halfHight = height / 2;
    const absmaxHalf = 1 / halfHight;
    // 获取所有波峰波谷
    const peaks = getPeaks(buff, perSecPx);
    // 设置svg的宽度
    svgWidth = buff.duration * perSecPx;
    draw.size(svgWidth);
    const points = [];
    for (let i = 0; i < peaks.length; i += 2) {
        const peak1 = peaks[i] || 0;
        const peak2 = peaks[i + 1] || 0;
        // 波峰波谷乘上系数
        const h1 = Math.round(peak1 / absmaxHalf);
        const h2 = Math.round(peak2 / absmaxHalf);
        points.push([i, halfHight - h1]);
        points.push([i, halfHight - h2]);
    }
    // 连接所有的波峰波谷
    const  polyline = draw.polyline(points);
    polyline.fill('none').stroke({ width: 1 });
}
// 获取波峰波谷
function getPeaks(buffer, perSecPx) {
    const { numberOfChannels, sampleRate, length} = buffer;
    // 每一份的点数=44100 / 100 = 441
    const sampleSize = ~~(sampleRate / perSecPx);
    const first = 0;
    const last = ~~(length / sampleSize)
    const peaks = [];
    // 为方便起见只取左声道
    const chan = buffer.getChannelData(0);
    for (let i = first; i <= last; i++) {
        const start = i * sampleSize;
        const end = start + sampleSize;
        let min = 0;
        let max = 0;
        for (let j = start; j < end; j ++) {
            const value = chan[j];
            if (value > max) {
                max = value;
            }
            if (
```

```
            }
        }
    }
    // 波峰
    peaks[2 * i] = max;
    // 波谷
    peaks[2 * i + 1] = min;
    return peaks;
}
```

2.3 缩放操作

有时候，需要对某些区域进行放大或者对整体波形进行缩小操作。由于音频波形是通过 svg 绘制的，缩放算法就会变得非常简单，只需直接对 svg 进行缩放即可。

代码实现 - 利用svg矢量图特性，我们只要将连接波分波谷的折线宽度乘上系数 scaleX 即可实现缩放功能，scaleX 大于1则放大，scaleX 小于1则缩小。 其实这是一种伪缩放，因为波形的精度始终是10毫秒，只是将折线图拉开了。

```
function zoom(scaleX) {
    draw.width(svgWidth * scaleX);
    polyline.width(svgWidth * scaleX);
}
```

2.4 裁剪操作

这节主要介绍下裁剪操作的实现，其他的操作也都是类似的对音频数据作计算。 所谓裁剪，就是从原始音频中去除不要的部分，如噪音部分，或者截取想要的部分，如副歌部分。要实现对音频文件进行裁剪， 首先我们需要对它 有足够的认识。 解码后的音频数据其实是一个 AudioBuffer对象 ， 它会被赋值给 AudioBufferSourceNode 音频源节点的 buffer 属性，并由 AudioBufferSourceNode 将其带进 AudioContext 的处理流里，其中 AudioBufferSourceNode 节点可以通过 AudioContext 的 createBufferSource 方法生成。 看到这里有点懵的同学可以回到 2.1 一节再回顾一下 AudioContext 的基本用法。 AudioBuffer 对象有 sampleRate（采样速率，一般为44.1kHz）、numberOfChannels（声道数）、duration（时长）、length（数据长度）4 个属性，还有 1 个比较重要的方法 getChannelData ，返回 1 个 Float32Array 类型的数组。我们就是通过改变这个 Float32Array 里的数据来对 音频进行裁剪或者其他操作。裁剪的具体步骤： - 首先获取到待处理音频的通道数和采样率 - 根据裁剪的开始时间点、结束时间点、采样率算出被裁剪的长度：长度 lengthInSamples = (endTime - startTime) * sampleRate，然后通过 AudioContext 的 createBuffer 方法创建一个长度为 lengthInSamples 的 AudioBuffer cutAudioBuffer 用于存放裁剪下来的音频数据，再创建一个长度为原始音频长度减去 lengthInSamples 的 AudioBuffer newAudioBuffer 用于存放裁剪后的音频数据 - 由于音频往往是多声道的，裁剪操作需要对所有声道都作裁剪，所以我们遍历所有声道，通过 AudioBuffer 的 getChannelData 方法返回各个声道 Float32Array 类型的音频数据 - 通过 Float32Array 的 subarray 方法获取需要被裁剪的音频数据，并通过 set 方法将数据设置到 cutAudioBuffer，同时将被裁剪之后的音频数据 set 到 newAudioBuffer中 - 返回 newAudioBuffer 和 cutAudioBuffer

```
function cut(originalAudioBuffer, start, end) {
    const { numberOfChannels, sampleRate } = originalAudioBuffer;
    const lengthInSamples = (end - start) * sampleRate;
    // offlineAudioContext相对AudioContext更加节省资源
    const offlineAudioContext = new OfflineAudioContext(numberOfChannels, numbe
    // 存放截取的数据
    const cutAud
        numberOf
```

```
        lengthInSamples,
        sampleRate
    );
    // 存放截取后的数据
    const newAudioBuffer = offlineAudioContext.createBuffer(
        numberOfChannels,
        originalAudioBuffer.length - cutSegment.length,
        originalAudioBuffer.sampleRate
    );
    // 将截取数据和截取后的数据放入对应的缓存中
    for (let channel = 0; channel < numberOfChannels; channel++) {
        const newChannelData = newAudioBuffer.getChannelData(channel);
        const cutChannelData = cutAudioBuffer.getChannelData(channel);
        const originalChannelData = originalAudioBuffer.getChannelData(channel)
        const beforeData = originalChannelData.subarray(0,
            start * sampleRate - 1);
        const midData = originalChannelData.subarray(start * sampleRate,
            end * sampleRate - 1);
        const afterData = originalChannelData.subarray(
            end * sampleRate
        );
        cutChannelData.set(midData);
        if (start > 0) {
            newChannelData.set(beforeData);
            newChannelData.set(afterData, (start * sampleRate));
        } else {
            newChannelData.set(afterData);
        }
    }
    return {
        // 截取后的数据
        newAudioBuffer,
        // 截取部分的数据
        cutSelection: cutAudioBuffer
    };
};
```

2.5 撤销和重做操作

每一次操作前，把当前的音频数据保存起来。撤销或者重做时，再把对应的音频数据加载进来。这种方式有不小的性能开销，在第 3 章 - 性能优化章节中作具体分析。

第 3 章-音频编辑器的性能优化

3.1 存在的问题

通过第 2 章介绍的近似法用比较少的点来绘制音频波形，已基本满足波形查看功能。但是仍存在以下 2 个性能问题：

1. 如果对波形进行缩放分析，比如将波形拉大 10 倍或者更大的时候，即使 svg 绘制的波形可以自适应不失真放大，但由于整个波形放大了 10 倍以上，需要绘制的像素点也增加了 10 倍，导致整个缩放过程非常得卡顿。
2. 撤销和重做功能此每次操作都需要保存修改后音频数据。一份音频数据，一般都在几 M 到十几 M 不等，每次操作都保存的话，势必会撑爆内存。

3.2 性能优化方案

3.2.1 懒加载

缩放波形卡顿的主要原因就是所需要绘制的像素点太多，因此我们可以通过懒加载的形式减少每次绘制波形时所需要绘制的像素点。 具体方案就是，根据当前波形的滚动位置，实时计算出当前视口需要绘制波形范围。 因此，需要对第 2 章获取波峰波谷的函数 getPeaks 进行一下改造, 增加 2 个参数：

- buffer：解码后的音频数据 AudioBuffer
- pxPerSec：每秒钟音频数据横向需要的像素点，这里为 100，每 10 毫秒数据对应 1 组波峰波谷
- start：当前波形视口滚动起始位置 scrollLeft
- end：当前波形视口滚动结束位置 scrollLeft + viewWidth。
- 具体计算时，我们只会取当前视口内对应时间段的音频的波峰和波谷。
- 比如 start 等于 10，end 等于 100，根据我们 1 个像素对应 1 个 10 毫秒数据量波峰波谷的近似算法，就是取第 10 个 10 毫秒到第 100 个 10 毫秒的波峰波谷，即时间段为 100 毫秒到 1 秒。

```
function getPeaks(buffer, pxPerSec, start, end) {
    const { numberOfChannels, sampleRate } = buffer;
    const sampleWidth = ~~(sampleRate / pxPerSec);
    const step = 1;
    const peaks = [];
    for (let c = 0; c < numberOfChannels; c++) {
        const chanData = buffer.getChannelData(c);
        for (let i = start, z = 0; i < end; i += step) {
            let max = 0;
            let min = 0;
            for (let j = i * sampleWidth; j < (i + 1) * sampleWidth; j++) {
                const value = chanData[j];
                max = Math.max(value, max);
                min = Math.min(value, min);
            }
            peaks[z * 2] = Math.max(max, peaks[z * 2] || 0);
            peaks[z * 2 + 1] = Math.min(min, peaks[z * 2 + 1] || 0);
            z++;
        }
    }
    return peaks;
}
```

3.2.2 撤销操作的优化

其实我们只需要保存一份原始未加工过的音频数据，然后在每次编辑前，把当前执行过的指令集全部保存下来，在撤销或者重做时，再把对应的指令集对原始音频数据操作一遍。比如：对波形进行 2 次操作：第 1 次操作时裁剪掉 0-1 秒的部分，保存指令集 A 为裁剪 0-1 秒；第二次操作时，再一次裁剪 2-3 秒的部分，保存指令集 B 为裁剪 0-1 秒、裁剪 2-3 秒。撤销第 2 次操作，只要用前一次指令集 A 对原始波形作一次操作即可。通过这种保存指令集的方式，极大降低了内存的消耗。

总结

声音实质就是声波在人耳中振动被人脑感知，决定音质的因素包括振幅、频率和音色（谐波），人耳只能识别 20-20kHz 频率和 0-120db 振幅的声音。 音频数字化处理过程为：脉冲抽样，量化，编码，解码，加工，回放。 用 canvas 或者 svg 绘制声音波形时，会随着绘制的像素点上升，性能急剧下降，通过懒加载按需绘制的方式可以有效的提高绘制性能。 通过保存指令集的方式进行撤销和重做操作，可以有效的节省内存消耗。 Web Audio API 所能做的事情还有很多很多，期待大家一起去深挖。

参考

- 什么是声波
- 一文看懂音频原理
- Basic concepts beh
- Using the Web Aud

- Web Audio API
- AudioBufferSourceNode
- AudioBuffer
- svg.js
- 傅里叶分析之掐死教程（完整版）

> 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

Web Audio

文章被以下专栏收录

云音乐前端技术团队专栏 关注专栏

推荐阅读

动漫女生头像——小姐姐该有的样子我都有

里昂的树洞

最近最久未使用算法（LRU）介绍与实现

概念LRU 全称是 Least Recently Used，即最近最久未使用算法，它是页面置换算法的一种。原理LRU 算法根据数据的历史访问记录来进行淘汰数据，其核心思想是「如果数据最近被访问过，那么将来...

王天笑

2017全国职业院校信息化教学大赛作品述评

创壹100VR

成分越多

陈伟东

还没有评论

写下你的评论...