Web Audio API

W3C Working Draft 10 October 2013

Web Audio API 官方文档中文版 v0.8

本文档基于 W3C 第 20131010 Web Audio 草案翻译

@音乐前端 2014.01.18

注：本文档仅供大家学习交流，请勿用于任何商业用途.

摘要

此规范描述了一个在 Web 应用中处理与合成音频的高级 Javascript API。其核心思想是音频路由图，在这个路由图中一组 AudioNode 对象连接到一起来决定整体的音频渲染。实际的处理过程将主要在底层实现（通常是对 C 、C++的组合优化），但是同样支持 Javascript 直接处理和合成音频。

”简介”这一节讲述了此规范背后的动机。

设计此 API，并期望和 Web 平台上的其他 API 以及元素一起使用，特别是 XMLHttpRequest（使用 responseType 和 response 属性）。在游戏和交互式应用中，可以将 Web Audio API 和 canvas2d 以及 WebGL 3D 图形接口一起使用。

文档状态

此部分描述了这份文档的状态发布时的状态。后续可能会有文档替代此文档。W3C 当前发布的文档列表以及此技术报告的最新版可以在 W3C technical reports index 上找到 (http://www.w3.org/TR/webaudio/)。

此文档是 Web Audio API 规范的草案。由 W3C Audio Working Group 发布，是 W3C Webapps activity 的一部分。

请发送关于此文档的评论到 public-audio@w3.org， 期待 Web 内容和浏览器开发者来 review 这份草案。

草案并不表示已经受 W3C Membership 的支持。这是一份草案文档并可能在未来某个时间被其他文档更新、替代或者被废除。因此不适合引用此文档（正在进行中的项目除外）。

目录

1．引言 5

1.1．特点 6

1.2. 模组化路由 8

1.3．API 概览 13

2．一致性（官方声明，略） 16

4. Audio API 16

4.1AudioContext 接口 16

4.2 AudioNode 接口 26

4.4 AudioDestinationNode 接口 32

4.5 AudioParam 接口 34

4.7 GainNode 接口 41

4.8 DelayNode 接口 42

4.9 AudioBuffer 接口 43

4.10 AudioBufferSourceNode 接口 45

4.11 MediaElementAudioSourceNode 接口 48

4.12. ScriptProcessorNode 接口 50

4.13. AudioProcessingEvent 接口 51

4.14. PannerNode 接口……53

4.15. AudioListener 接口……57

4.16. ConvolverNode 接口……59

4.17. AnalyserNode 接口……63

4.18. ChannelSplitterNode 接口……66

4.19. ChannelMergerNode 接口……68

4.20. DynamicsCompressorNode 接口……69

4.21 BiquadFilterNode 接口……71

4.22. WaveShaperNode 接口……76

4.23. OscillatorNode 接口……78

4.24. PeriodicWave 接口……81

4.25. MediaStreamAudioSourceNode 接口……81

4.26. MediaStreamAudioDestinationNode 接口……82

6 混音器增益结构……83

7.动态生命周期……89

9.声道的向上混音和向下混音……92

9.1 扬声器声道布局……94

9.2. Channel Rules Examples……99

11.声音定位/位移……100

12.使用卷积的线性效果……111

13.javascript 合成和处理……117

15.性能考虑…… 118

15.1 延迟：是什么以及为什么重要…… 118

15.2 音频毛刺……118

15.3 硬件可扩展性……119

15.4 使用 javascript 进行实时音频处理、合成的问题…… 121

16. 应用举例（直接示例，翻译略）……122

17. 安全考虑（源文档无内容）……122

18. 隐私考虑 122

1．引言

Web 上的音频播放到一直非常原始，不久之前还不得不通过 Flash 或者 QuickTime 的插件形式将声音传递给用户。HTML5 引入 Audio 元素有非常重要的意义，它允许了基本的流式音频播放。但是，它并不能处理更复杂的音频应用。对于复杂的基于 Web 的游戏或者交互式应用，我们需要其他的解决方案。此规范的目标就是介绍实现现代游戏音频引擎和现代桌面音频处理应用里实现的混合、处理、滤波等功能的方法。

这份 API 设计之初就已构想好了广泛的用例。理想的情况下，它应该能支持任何用例，一种合理的实现方案是运行在浏览器中、由 Javascript 控制优化了的 C++引擎。这就是说，因为现代的桌面音频软件可以有非常高级的功能，其中的一些可能很难或者不可能由此 API 系统实现。苹果的 Logic Audio 就是这样高级的应用，它可以支持外部的 MIDI 控制、多种控件式音频效果与合成器、高度优化直达硬盘的音频文件读写、紧密集成的时间轴伸缩等功能。尽管如此，此文档也能支持很大范围的复杂游戏和交互式应用，包括音乐应用。而且它可以在 WebGL 提供的高级图形应用中提供音频功能。此 API 后续会添加更多高级功能。

1.1．特点

此 API 支持下述主要特征：

- 模组化路由用于支持简单或者复杂的混音/音效架构，包括多输入和编组总线。
- 采样精度可调节的音频播放、低延时特征实现那些对于节奏有高精度要求的音乐应用，比如鼓和音序器。还可以动态创建音效。
- 为音频波封、渐入渐出、颗粒感、滤波器扫描、低频振荡器实现音频参数自动化
- 灵活处理音频流的声道，可以分离和合并声道
- 处理来源于 Audio 和 Video 媒体元素的音频源
- 处理来源于 getUserMedia()的 MediaStream 的现场音频
- 和网络实时通信结合
    - 处理通过 MediaStream 获得的接收自远程端的音频
    - 使用 MediaStream 将生成或者处理过的音频流发送到远程端

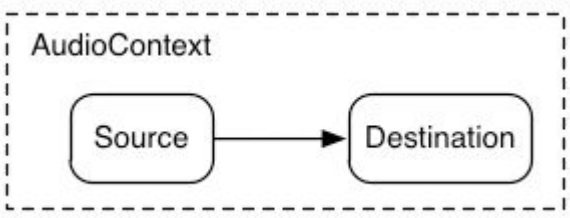
- 直接通过 Javascript 进行音频流的合成和处理
- 空间化音频可以实现大量的 3D 游戏和身临其境的环境：
    - 平移模型（左右声道平移）：同等功率、头相关变换函数（一种音效定位算法）、直通
    - 距离衰减
    - 声锥（用来描述有方向的声音的大小）
    - 声音闭塞或者梗阻
    - 多普勒频移
    - 基于音源/听者的模型
- 一个卷积引擎实现大量的线性效果，尤其是高质量的室内音效。下面列出了一些可能的效果
    - 大/小房间
    - 教堂
    - 音乐厅
    - 山洞
    - 隧道
    - 门厅
    - 森林
    - 露天剧场
    - 通过门口的一个遥远房间的声音
    - 滤波器

    - 奇怪的背景音效
    - 极端梳状滤波器音效
- 动态压缩可用于混音的整体控制和音效优化
- 可实现高效的实时、时域、频率分析/音频视觉化效果
- 可实现高效的低通、高通以及其他常用的滤波器的二阶滤波器
- 波形整形可用来实现失真和其他非线性效果
- 声响振荡器

1.2. 模组化路由

模组化路由允许不同 AudioNode 对象之间任意连接组成。每个节点可以有多个输入或者多个输出。一个源节点没有输入、有唯一输出。一个目的节点有唯一输入、没有输出 ,最常见的例子是 AudioDestinationNode，这是使用音频硬件的最终目的地。 其他节点比如滤波器节点，可以放在源节点和目的节点之间。开发者不用操心两个节点对象相连时低层次的流格式细节，节点会自动完成期望的工作。例如，如果一个单声道音频流和一个立体声输入的节点相连，它们会适当的对左右声道进行混音。

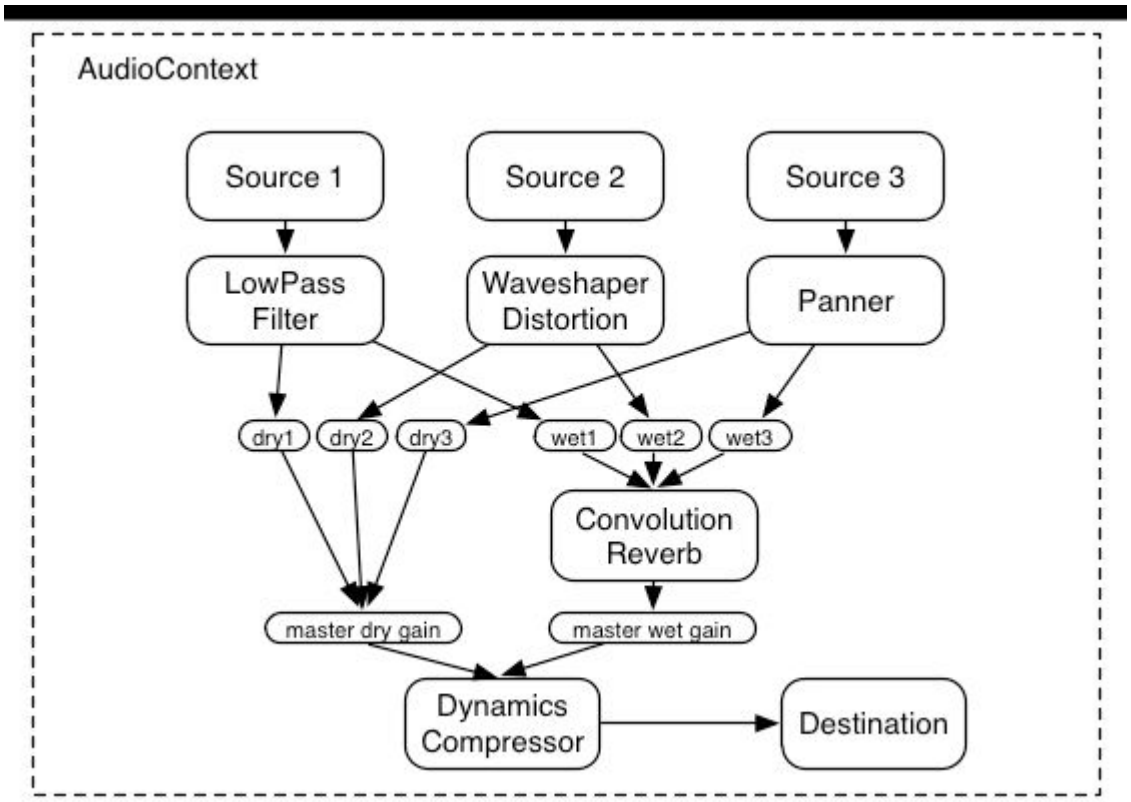
举个最简单的例子，单一源可以被直接路由到音频输出。所有的路由过程发生在一个包含单一 AudioDestinationNode 节点的 AudioContext 里。



下面是这个简单路由的例子，它实现了播放单一声音：

```
var context = new AudioContext();
    function playSound() {
        var source = context.createBufferSounce();
        source.buffer = dogBarkingBuffer;
        source.connect(context.destination);
        source.start(0);
}
```

下图是一个更复杂的例子，它有三个输入源，一个卷积混响，并且在最后输出阶段之前有一个动态压缩器。



```
var context = 0;

var compressor = 0;

var reverb = 0;


var source1 = 0;

var source2 = 0;

var source3 = 0;


var lowpassFilter = 0;

var waveShaper = 0;

var panner = 0;


var dry1 = 0;

var dry2 = 0;

var dry3 = 0;


var wet1 = 0;

var wet2 = 0;

var wet3 = 0;
```

```
var masterDry = 0;

var masterWet = 0 ;


function setupRoutingGraph() {

    context = new AudioContext();


    // Create the effects nodes.

    lowpassFilter = context.createBiquadFilter();

    waveShapter = context.createWaveShapter();

    panner = context.createPanner();

    compressor = context.createDynamicsCompressor();

    reverb = context.createConvolver();



    // Create master wet and dry.

    masterDry = context.createGain();

    masterWet = context.createGain();


    // connect final compressor to final destination

    compressor.connect(context.destination);
```

```
// connect master dry and wet to compressor

masterDry.connect(compressor);

masterWet.connect(compressor);


// connect reverb to master wet

reverb.connect(masterWet);


// create a few sources

source1 = context.createBufferSource();

source2 = context.createBufferSource();

source3 = context.createOscillator();


source1.buffer = manTalkingBuffer;

source2.buffer = footstepsBuffer;

source3.frequency.value = 440;


// connect source1

dry1 = context.createGain();

wet1 = context.createGain();

source1.connect(lowpassFilter);

lowpassFilter.connect(dry1);
```

```
lowpassFilter.connect(wet1);

dry1.connect(masterDry);

wet1.connect(reverb);


// connect source2

dry2 = context.createGain();

wet2 = context.createGain();

source2.connect(waveShaper);

waveShaper.connect(dry2);

waveShaper.connect(wet2);

dry2.connect(masterDry);

wet2.connect(reverb);


// connect source3

dry3 = context.createGain();

wet3 = context.createGain();

source3.connect(panner);

panner.connect(dry3);

panner.connect(wet3);

dry3.connect(masterDry);

wet3.connect(reverb);
```

```
        // start the source now


        source1.start(0);
        source2.start(0);
        source3.start(0);
    }
```

1.3 . API 概览

接口定义如下：

- AudioContext 接口，包含一个音频信号路由图，用来表示 AudioNodes 之间的连接关系
- AudioNode 接口，代表了音频源、音频输出、以及中间处理模块。AudioNodes 可以以模块化方式动态连接在一起。AudioNodes 存在于 AudioContext 上下文中
- AudioDestinationNode 接口，一个 AudioNode 的子类，代表所有渲染后的音频最终目的地
- AudioBuffer 接口，和驻留内存的音频内容相关。它可以表示短暂的声音或者更长的音频剪辑。
- AudioBufferSourceNode 接口 ,一种可以由 AudioBuffer 产生音频的 AudioNode

- MediaElementAudioSourceNode 接口，一种音频资源来源于 Audio、Video、或者其他媒体元素的音频资源 AudioNode
- MeidaStreamAudioSourceNode 接口，一种 AudioNode 其音频资源来源于 MediaStream 比如现场音频输入或者远程端输入的音频资源
- MediaStreamAudioDestinationNode 接口，是将音频传输到 MeidaStream（会被传到远程端）的目的节点，是 AudioNode 的一种
- ScriptProcessorNode 接口，直接用 Javascript 来处理或者产生音频的 AudioNode 的接口
- AudioProcessingEvent 接口 ,和 ScriptProcessorNode 对象一起使用的事件类型
- AudioParam 接口，用来控制的一个 AudioNode 功能的某个单一方面，比如音量
- GainNode 接口，用于显式增益控制。因为 AudioNodes 的输入支持多种连接（作为一个单位增益来汇总这连接），混音器可以用 GainNode 轻松的构建
- BiquadFilterNode 接口，一种低阶滤波器的 AudioNode 节点，比如：
    - 低通滤波器
    - 高通滤波器
    - 带通滤波器
    - 低架滤波器
    - 高架滤波器
    - 峰值滤波器
    - 陷波滤波器（带阻）
    - 全通滤波器

- DelayNode 接口，具有可动态调节延迟参数的 AudioNode
- PannerNode 接口，用来调节音频在三维空间的定位
- AudioListener 接口，在空间定位的时候和 PannerNode 一起工作
- ConvolverNode 接口 ,应用实时线性效果的一种 AudioNode(比如音乐厅的声音)
- AnalyserNode 接口，用于音乐可视化或者其他可视化应用
- ChannelSplitterNode 接口，用来在路由图中访问音频流的单一声道
- ChannelMergerNode 接口，用来将多个音频流的声道合并为一个音频流
- DynamicsCompressorNode 接口，用于动态压缩的一种 AudioNode
- WaveShaperNode 接口，应用一种非线性波形整形效果来实现失真和其他微妙的音效
- OscillatorNode 接口，可以产生周期性波形的音频源

2．一致性（官方声明，略）

3．（源文档无内容）

4. Audio API

4.1． AudioContext 接口

这个接口代表了一组 AudioNode 对象和它们之间的连接。它允许信号用任意方式的路由到达 AudioDestinationNode（最终用户耳朵听到的内容）。节点在这个环境中创建，然后相互连接到一起。大多数情况下，每个 document 只使用一个 AudioContext。

```
callback DecodeSuccessCallback = void (AudioBuffer decodedData);
callback DecodeErrorCallback = void ();

[Constructor]
interface AudioContext : EventTarget {

    readonly attribute AudioDestinationNode destination;
    readonly attribute float sampleRate;
    readonly attribute double currentTime;
    readonly attribute AudioListener listener;

    AudioBuffer createBuffer(unsigned long numberOfChannels, unsigned
long length, float sampleRate);

    void decodeAudioData(ArrayBuffer audioData,
                         DecodeSuccessCallback successCallback,
                         optional DecodeErrorCallback errorCallback);
```

```
        // AudioNode creation
        AudioBufferSourceNode createBufferSource();

        MediaElementAudioSourceNode
createMediaElementSource(HTMLMediaElement mediaElement);

        MediaStreamAudioSourceNode createMediaStreamSource(MediaStream
mediaStream);
        MediaStreamAudioDestinationNode createMediaStreamDestination();

        ScriptProcessorNode createScriptProcessor(optional unsigned long
bufferSize = 0,
                                                   optional unsigned long
numberOfInputChannels = 2,
                                                   optional unsigned long
numberOfOutputChannels = 2);

        AnalyserNode createAnalyser();
        GainNode createGain();
        DelayNode createDelay(optional double maxDelayTime = 1.0);
        BiquadFilterNode createBiquadFilter();
```

```
        WaveShaperNode createWaveShaper();
        PannerNode createPanner();
        ConvolverNode createConvolver();

        ChannelSplitterNode createChannelSplitter(optional unsigned long
numberOfOutputs = 6);
        ChannelMergerNode createChannelMerger(optional unsigned long
numberOfInputs = 6);

        DynamicsCompressorNode createDynamicsCompressor();

        OscillatorNode createOscillator();
        PeriodicWave createPeriodicWave(Float32Array real, Float32Array imag);

    };
```

4.1.1 属性

destination

一个 AudioDestinationNode 和一个单一输入组成了所有音频的最终目的地。通常这代表了实际的音频硬件。所有渲染音频的 AudioNode 将直接或者间接的和这个目的地相连。

sampleRate

AudioContext 处理音频的采样频率（每秒采样帧数）。假设在这个环境下的所有的 AudioNode 都用这同一个频率。基于这个假设，采样率转换器或者变速处理器在实时处理中是不支持的。

currentTime

这是以秒为单位的时间，当环境创建的时候它的初始值为 0，并实时增加。所有的预定时间都和它相关。这不是一个可以开始、暂停、和重定位的时间。它总是增加的。基于此可以很轻松的（用 Javascript）构建一个像 GarageBand 的时间轴系统。这个时间和持续增长的硬件时间戳相一致。

listener

AudioListener 用于监听在三维空间定位音频的事件。

4.1.2 方法和参数

createBuffer 方法

按照给定 size 创建一个 AudioBuffer。缓冲初始化的时候音频数据为 0。如果 numberOfChannels 或者 sampleRate 越界或者长度为 0 的时候会抛出一个 NOT_SUPPORTED_ERR 的异常

numberOfChannels 这个参数决定了这个 buffer 有几个声道。实际实现中必须支持至少 32 个声道

length 参数决定了采样帧里 buffer 的尺寸。

sampleRate 参数描述了每秒采样帧的缓冲区中，线性脉冲编码调制数据中的采样率。实现中必须支持最小范围为 22050~96000 的采样率区间

decodeAudioData 方法

异步解码 ArrayBuffer 中的音频文件数据。比如这个 ArrayBuffer 可以通过 XMLHttpRequest 的 response 属性获得 需要将 responseType 设置为“arraybuffer”。音频文件数据可以是 Audio 元素支持的任何格式。

audioData 是包含音频文件数据的 ArrayBuffer

successCallback 是在解码完成时触发的回调函数。这个回调函数的唯一参数是一个 AudioBuffer，表示被解码的线性脉冲编码调制（PCM）数据。

errorCallback 是在解码音频文件数据出现错误的时候触发的回调函数。

下面的步骤必须实现：

1. 临时锁定 audioData ArrayBuffer，保证在这段期间 Javascript 代码不能读取或者修改数据
2. 将解码操作放入另一个线程队列里

3. 解码线程将会把编码了的 audioData 解码为线性 PCM。如果因为音频格式不能识别或者不支持或者因为数据损坏/意外/不一致而抛出解码错误，那么 audioData 的锁定状态将恢复正常，接下来 errorCallback 将会安排到主线程的事件循环中，整个过程终止。
4. 解码进程将产出结果，表示解码了的线性 PCM 音频数据。如果 AudioContext 的采样率和 audioData 的采样率不一致，就需要把数据按照 AudioContext 的采样率重新进行采样。最终的结果（在可能的采样率转换后）将存在一个 AudioBuffer 里
5. audioBuffer 的锁定状态将重新变为正常状态
6. successCallback 函数将安排进主线程的事件循环里等待执行，并且把步骤 4 中的 AudioBuffer 作为参数传入回调函数

createBufferSource 方法

创建一个 AudioBufferSourceNode

createMediaElementSource 方法

创建一个 MediaElementAudioSourceNode 作为一个 HTMLMediaElement。调用此方法之后，MediaStream 播放的音频将被重新路由到 AudioContext 的处理图中

createMediaStreamSource 方法

创建一个 MediaStreamAudioSourceNode 作为一个 MediaStream。调用此方法之后，MediaStream 播放的音频将会重新路由到 AudioContext 的处理图中

createMediaStreamDestination 方法

创建一个 MediaStreamAudioDestinationNode

createScriptProcessor 方法

创建一个 ScriptProcessorNode，可以用 Javascript 直接处理音频。如果 bufferSize 或者 numberOfInputChannels 或者 numberOfOutputChannels 超出有效范围则会抛出 INDEX_SIZE_ERR 的异常

bufferSize 参数决定了采样帧单元的缓冲区大小。如果没有传递这个参数 ,或者值为 0，这具体的实现环境中需要能自动根据环境选择最佳的缓冲大小 ,在节点生命周期中都将保持大小为 2 的常量整数值的数值。如果用户显式传入具体的缓冲大小，数值必须是以下值：256、512、1024、2048、4096、8192、16384。这个值控制了 audioprocess 事件派发的频率以及每个回调里有多少采样帧需要处理。bufferSize 值比较小的时候会有较低的延时。值比较大的时候需要避免出现音频卡断和毛刺。建议用户不要指定 buffer 的大小，而让具体环境的实现去自动选择比较好的 buffer 大小，从而能比较好的平衡延迟和音频质量

numberOfInputChannels 参数（默认为 2），它决定了此节点的输入声道数量。需要能支持数量高达 32 的声道数。

numberOfOutputChannels 参数（默认为 2），它决定了此节点的输出声道数量。需要能支持数量高达 32 的声道数。

numberOfInputChannels 和 numberOfOutputChannels 参数都不可以为 0

createAnalyser 方法

创建 AnalyserNode

createGain 方法

创建 GainNode

createDelay 方法

创建 DelayNode 表示延迟可变的时间轴。初始化默认延迟为 0 秒。

maxDelayTime 参数是可选的，它决定了按秒算的最大延时。如果传了此参数，必须要大于 0 而且要小于 3 分钟，否则会抛出 NOT_SUPPORTED_ERR 的异常

createBiquadFilter 方法

创建 BiquadFilterNode，是二阶滤波器，它可以配置为多种常见滤波器类型的一种。

createWaveShaper 方法

创建 WaveShaperNode，是非线性失真变换

createPanner 方法

创建 PannerNode

createConvolver 方法

创建 ConvolverNode

createChannelSplitter 方法

创建 ChannelSplitterNode，是一个声道分频器。如果参数无效就会抛出异常

numberOfOutput 参数决定了输出的数量。需要支持高达 32 的输出数量。如果没有明确，默认值为 6

createChannelMerger 方法

创建 ChannelMergerNode，是声道合并器。如果参数无效就会抛出异常

numberOfOutput 参数决定了输出的数量。需要支持高达 32 的输出数量。如果没有明确，默认值为 6

createDynamicsCompressor 方法

创建 DynamicsCompressorNode

createOscillator 方法

创建 OsillaterNode

createPeriodicWave 方法

创建 PeriodicWave，是包含有任意谐波含量的波形。参数 real 和 imag 必须是 Float32Array 类型，值大于 0 小于等 4096，不然就要抛出异常。这两个参数表示一个傅立叶级数的傅立叶系数，来产生一个周期波形。这个产生的周期波将和 OscillatorNode 一起

使用，表示一个有绝对值峰值为 1 的归一化了的时域波形。另一个表达方式是 OscillatorNode 产生的波形将在 0dBFS 值处达到最大波峰。方便起见，这个和 Web Audio API 用到的信号值的全范围相对应。因为周期波会在创建的时候归一化，real 和 imag 参数是相对值。

real 参数表示余弦项组成的数组(通常用 A 表示)。在音频的术语里，第一个参数(index 0) 是周期波形的直流偏移，通常设置为 0。第二个元素（index 1）表示基本的频率。第三个元素表示第一个泛音。

imag 参数表示正弦项组成的数组（通常用 B 表示）。第一个参数（index 0）应该设置为 0（而且会被忽略），因为这一项在傅立叶级数里不存在。第二个元素（index 1）表示基本的频率。第三个元素表示第一个泛音。

4.1.3 生命周期

一个 AudioContext 一旦被创建，将会一直播放声音直到没有声音可以播放，或者页面被关闭。

4.1b OfflineAudioContext 接口

OfflineAudioContext 是 AudioContext 的特殊类型，它比实时的渲染/混音更快。它不渲染音频硬件，但是作为替代，通过调用复杂的事件处理并以 AudioBuffer 的形式作为返回值，尽可能快的渲染。

```
[Constructor(unsigned long numberOfChannels, unsigned long length, float sampleRate)]
interface OfflineAudioContext : AudioContext {

    void startRendering();

    attribute EventHandler oncomplete;

};
```

4.1 b.1 属性

oncomplete

OfflineAudioCompletionEvent 的事件处理类型之一

4.1 b.2 方法和参数

startRendering 方法

给定了当前连接、安排好的变化，就开始渲染音频了。oncomplete 句柄会在渲染一结束就调用。这个方法必须只能调用一次，不然会抛异常。

4.1c OfflineAudioCompletionEvent 接口

这个事件对象将派发给 OfflineAudioContext

```
interface OfflineAudioCompletionEvent : Event {

    readonly attribute AudioBuffer renderedBuffer;
```

```
};
```

4.1c.1 属性

renderedBuffer

一旦 OfflineAudioContext 完成渲染了，AudioBuffer 就包含了已经渲染的音频数据。它包含了数量等于 numberOfChannels(OfflineAudioContext 构造器的参数)的声道数。

4.2 AudioNode 接口

AudioNode 是搭建 AudioContext 的基本组成部分。这个接口代表了音频源、音频目的以及中间处理模块。这些模块可以相互连接来构成渲染音频到音频硬件的处理流程图。每个节点可以有输入和输出。一个源节点没有输入、只有一个输出。一个 AudioDestinationNode 只有一个输入、没有输出，而且代表了通向音频硬件的最终目的地。主要的处理节点比如滤波器，会有一个输入、一个输出。每个类型的 AudioNode 都会在处理和合成音频的细节上有所不同。但是，总的来说，AudioNode 会处理输入（如果有的话）、并为输出产生音频（如果有的话）。

每个输出有一个或者多个声道。具体声道的个数取决于 AudioNode 的规范细节。

一个输出可能连接到一个或者多个 AudioNode 的输入，因此支持扇出。输入初始化的时候没有连接，但是可以连接到一个或者多个 AudioNode 输出上，因此支持扇入。当通过调用 connect()方法来连接一个 AudioNode 的输入和另一个 AudioNode 的输出，我们称之为输入的 connection 属性。

在任何时候每个 AudioNode 的输入都有具体的声道数量。数量取决于输入的 connection 属性。如果输入没有 connection 值那它默认有一个声道。

对于每个输入，AudioNode 会混合（通常是向上混合，就是生成的输出的声道数大于输入的声道数）所有连接到输入的连接。请看 Mixer Gain Structure 获得更多细节，以及 Channel up-mixing and down-mixing 部分获得规范性要求。

出于性能考虑，实际实施规范的时候需要使用块处理，每个 AudioNode 处理固定的采样帧数 block-size。为了保证实际实施规范的统一性，我们将明确定义这个值。block-size 定义为 128 的采样帧数，这个值和差不多是 44.1KHZ 的 3ms 采样数。

AudioNode 是 EventTargets，在 DOM 文档中定义了。这表示它可以和其他 EventTargets 一样派发和接收事件。

```
enum ChannelCountMode {
    "max",
    "clamped-max",
    "explicit"
};

enum ChannelInterpretation {
    "speakers",
    "discrete"
};
```

```
interface AudioNode : EventTarget {


        void connect(AudioNode destination, optional unsigned long output = 0,
optional unsigned long input = 0);
        void connect(AudioParam destination, optional unsigned long output = 0);
        void disconnect(optional unsigned long output = 0);


        readonly attribute AudioContext context;
        readonly attribute unsigned long numberOfInputs;
        readonly attribute unsigned long numberOfOutputs;


        // Channel up-mixing and down-mixing rules for all inputs.
        attribute unsigned long channelCount;
        attribute ChannelCountMode channelCountMode;
        attribute ChannelInterpretation channelInterpretation;


};
```

4.2.1 属性

context

AudioNode 所属的 AudioContext

numberOfInputs

输入 AudioNode 的数量，对于源节点，值为 0

numberOfOutputs

AudioNode 的输出数量。对于 AudioDestinationNode 来说此值为 0

channelCount

把任何连接到节点的输入中的数据，进行向上混合和向下混合时，使用的声道的数量。默认值为 2。这个属性对于没有输入的节点无影响。如果值设置为 0，必须抛 NOT_SUPPORTED_ERR 异常

channelCountMode

在将连接到节点的输入进行向上混音和向下混音的时候，决定了如何计算声道数量。此属性对于没有输入的节点无效。

channelInterpretation

在将连接到节点的输入进行向上混音和向下混音的时候，决定了单一声道如何处理。此属性对于没有输入的节点无效。

4.2.2 方法和参数

connect 方法 连接到 AudioNode

destination 参数是连接的目标节点

output 参数是描述了用 AudioNode 的哪个输出进行连接的序号。序号越界会抛出异常

input 参数是描述了用 AudioNode 的哪个输出进行连接的序号。序号越界会抛出异常

可以将 AudioNode 的输出连接到多个输入，通过多次调用 connects()。因此，支持”扇出“。

存在 AudioNode 之间循环连接的情况，前提是其中至少有一个 DelayNode 在此循环中，否则会抛出异常。

对于一个给定节点的输出和另一个给定节点的输入，只允许一个链接。多次连接等同于第一次的连接，后面的声明会被忽略。

nodeA.connect(nodeB);

nodeA.connect(nodeB);

和 nodeA.connect(nodeB);效果一样

connect 连接到 AudioParam 方法

连接 AudioNode 和 AudioParam，参数为音频速率信号

destination 参数表示连接到的 AudioParam 节点。

output 参数（返回值）为序号，表示用 AudioNode 的哪个输出进行连接。越界会抛出异常。

可以将一个 AudioNode 的输出连接到多个 AudioParam 上 ,通过多次调用 connect() 实现。因此支持”扇入“。

AudioParam 会将连接到它的、由任何 AudioNode 的输出已渲染音频数据，转换为单声道数据，如果源数据不是单声道则进行向下混音，然后和其他连接到 AudioParam 的数据进行混音，最后和内部参数值进行混合（该值是当 AudioParam 没有输入连接的时候仍然还有的参数值），包括对此参数的任何时间轴改变的安排

一个 AudioNode 和一个 AudioParam 只有一个有效连接，多的会被忽略

nodeA.connect(param);

nodeA.connect(param);

效果等同于 nodeA.connect(param);

disconnect 方法

断掉一个 AudioNode 的输出的对外连接。

output 参数是一个序号，描述了 AudioNode 的哪个输出会被断掉连接。参数越界会抛出异常。

4.2.3 生命周期

具体的实现环境可以选择任何方法来避免无用节点完成工作的节点带来的不必要的资源使用和内存泄漏导致的增长。下面的描述是指导如何实现节点生命周期的管理的方案。

节点只有还有引用就一直保持生命周期。引用有多种类型

1. 普通的 Javascript 引用遵守正常的垃圾回收机制
2. AudioBufferSourceNodes 和 OscillatorNodes 的运行时应用。当这些节点正在使用的时候，它们维持一个到自身的运行时引用
3. 如果有另一个 AudioNode 连接到某个节点，该节点就有一个连接引用
4. 当 AudioNode 有任何内部处理状态而且没有被取消的时候，该节点会维持一个 tail-time 引用。例如一个 ConvolverNode 有一个 tail，这个 tail 在没有输入的时候也持续运行（类比于在一个很大的音乐厅拍手后，耳朵仍然能听到有声音会在整个大厅回荡）。有些 AudioNode 有此属性，具体的参见各个节点的描述。

任何在循环连接中的 AudioNode，或者之间或者间接连接到它所属的 AudioContext 的 AudioDestinationNode 的 AudioNode ,将一直保持生命只要 AudioContex 也保持生命

当 AudioNode 没有引用的时候就会被删除。但是在删除之前，它会和所有其他所有连接到它的 AudioNode 中断连接。以这种方式它可以释放所有对其他节点的连接引用。

当 AudioContext 被删除的时候 AudioNode 也被删除。

4.4 AudioDestinationNode 接口

这个 AudioNode 是最终的音频目的地，也是用户听到声音的来源。通常我们类比于一个连接到音频输出设备的扬声器。所以渲染了的音频到达耳朵被听到之前都要先被路由到这个目的节点，它是 AudioContext 路由图的终止节点。没有 AudioContext 只有一个 AudioDestinationNode，作为 AudioContext 的 destination 属性的值。

numberOfInputs : 1

numberOfOutputs : 0

channelCount = 2;

channelCountMode = "explicit";

channelInterpretation = "speakers";

Web IDL

```
interface AudioDestinationNode : AudioNode {


    readonly attribute unsigned long maxChannelCount;
```

```
};
```

4.4.1 属性

节点的最大声道数是由节点的 channelCount 属性决定。一个 AudioDestinationNode 表示音频硬件的终点（通常是这样），如果音频硬件是多声道的那么这个节点可以输出多于 2 个声道的音频。maxChannelCount 是硬件支持的最大声道数。如果值为 0，则表示 channelCount 不可以改变值。这个情况会发生在比如 OfflineAudioContext 中的 AudioDestinationNode 上，以及硬件只支持立体声输出的情况。

普通 AudioContext 的终点的 channelCount 默认为 2，也可以设置为任何小于等于 maxChannelCount 的非 0 值。如果值不在有效范围内，则抛出异常。具体的例子：如果音频硬件支持 8 声道输出，则我们会设置 numberOfChannels 为 8，并且渲染 8 声道的输出。

OfflineAudioContext 的 AudioDestinationNode，channelCount 在这个 OfflineAudioContext 创建的时候就定好了，而且不能改变。

4.5 AudioParam 接口

AudioParam 控制 AudioNode 某个方面的功能，比如音量。可以通过设置 value 属性来即时修改值。或者值的变动可以被安排到某个精确的时间发生（AudioContext.currentTime），比如波封、声音渐减、低频振荡器、滤波器扫描等。因

此可以将基于时间轴的任意自动化曲线赋给 AudioParam。除此之外，AudioNode 输出的音频信号可以作为 AudioParam 的输入，和内部参数值汇总。

一些用于合成和处理音频的 AudioNode 有 AudioParam 属性，音频的每个样本采样时都要计算这个值。对于其他 AudioParam，取样精度不那么重要，AudioParam 值的变化可以更粗粒度的采样。

实现时需要使用块处理，每个 AudioNode 在每个块中处理 128 个采样帧。

对于每个 128 样本帧块，k-rate 参数的值需要在第一个采样帧的进行采定，然后值被整个块使用。a-rate 参数需要为块的每个采样帧单独采定。

```
interface AudioParam {

        attribute float value;
        readonly attribute float defaultValue;

        // Parameter automation.
        void setValueAtTime(float value, double startTime);
        void linearRampToValueAtTime(float value, double endTime);
        void exponentialRampToValueAtTime(float value, double endTime);

        // Exponentially approach the target value with a rate having the given
time constant.
```

```
        void setTargetAtTime(float target, double startTime, double
timeConstant);

        // Sets an array of arbitrary parameter values starting at time for the given
duration.
        // The number of values will be scaled to fit into the desired duration.
        void setValueCurveAtTime(Float32Array values, double startTime, double
duration);

        // Cancels all scheduled parameter changes with times greater than or
equal to startTime.
        void cancelScheduledValues(double startTime);

    };
```

4.5.2 方法和参数

AudioParam 维护一个初始化为空的时序事件列表。时间对应于 AudioContext.currentTime 构成的时间坐标系。下面的事件定义了一个时间到值的映射关系。下面的方法可以添加新事件到事件列表。每个事件都有一个和它关联的时间，这些事件在列表中也总是保持一个时间序列。下面这些方法被称为自动化方法

- setValueAtTime() - SetValue

- linearRampToValueAtTime() - LinearRampToValue
- exponentialRampToValueAtTime() - ExponentialRampToValue
- setTargetAtTime() - SetTarget
- setValueCurveAtTime() - SetValueCurve

在调用这些方法的时候需要遵守下面这些规则：

- 如果一个事件添加的时候已经有同类事件在队列里，新事件会替换旧的
- 如果一个事件添加的时候有其他不同类型的事件在队列中，新事件会加到已有事件后面，后面添加的事件按顺序往后添加
- 如果 setValueCurveAtTime()在时间 T 调用、时长 D，如果还有其他事件在时间 T 之后、T+D 之前调用，就会抛出异常。换个说法，在一段时间内如果已经有事件在执行，再安排新曲线的绘制是不允许的
- 类似的，上述任何一个自动化方法调用的时候正处于 setValueCurve 事件（开始于 T，时长为 D）的时间间隔之内，也会抛出异常

setValueAtTime 方法

安排在指定时间改变参数值

value 参数是将会在给定时间变成的值

startTime 参数是和 AudioContext.currentTime 在同一个时间坐标系中的时间值

如果在 SetValue 事件之后没有其他事件，则对于 t>=startTime 时，v(t) = value。换句话说，这个值将保持常数

如果下一个事件起始时间为 T1，它触发在 SetValue 之后，而且不是 LinearRampToValue 或者 ExponentialRampToValue，那么对于 t：startTime<=t <T1，时 v(t)=value。换句话说，在时间间隔之内，值将保持常数

如果 SetValue 之后的执行的是 LinearRampToValue 或者 ExponentialRampToValue，则看下面的细节

linearRampToValueAtTime 方法

将当前的值变为一个线性连续变化的值

value 参数是线性倾斜在给定时间内需要到达的指定值

endTime 参数是和 AudioContext.currentTime 同一个坐标系的时间值

在 T0<=t<T1 之间值的计算为，v(t)= V0 + (V1-V0)*((t-T0)/(T1-T0))

在 t>=T1 之后如果没有其他函数执行，v(t)=V1

exponentialRampToValueAtTime 方法

形成从上一个值到给定值的一个指数连续变化

value 和 endtime 定义同上条

在 T0<=t<T1 之间，v(t) = V0 * (V1 / V0) ^ ((t - T0) / (T1 - T0))

在 t>=T1 之后如果没有其他函数执行，v(t)=V1

setTargetAtTime 方法

数值将以指数的方式接近目标值，速率由给定的时间常数决定。在其他场景中，这个函数适合用于实现 ADSR（Attack，Decay，Sustain，Release）包络的衰减和释放部分

target 参数是起点时间设定的目标值

startTime 参数是和 AudioContext.currentTime 在同一个时间坐标系中的时间值

timeConstant 参数是一阶滤波器（指数型）的时间常量，通过这个滤波器将值变为 target 的值。这个参数越大变化过程越平滑

更精确的说，timeConstant 是在给定一个阶跃输入相应时，用一阶线性连续时变系统达到 1-1/e（大约 63.2%），用的时间。

在这期间（T0<=t<T1），T0 是起始时间，T1 是终点时间：

v(t) = V1 + (V0 - V1) * exp(-(t - T0) / timeConstant)

V0 是初始值，V1 是 target 值

setValueCurveAtTime 方法

给定时间和给定时间长度，在起点设置任意值的 values 参数的数组。values 的数量将变换来适应设置的时间长度。

values 参数是 Float32Array 代表参数值曲线。这些值将在开始到结束都使用

startTime 同上

duration 同上

时间区间内 startTime <= t < startTime + duration

v(t) = values[N * (t - startTime) / duration]，N 是 values 数组的长度

cancelScheduledValues 方法

对于所有时间大于等于 startTime 参数的时间，取消所有的值变化的安排

startTime 参数是取消值变化安排的时间起点。

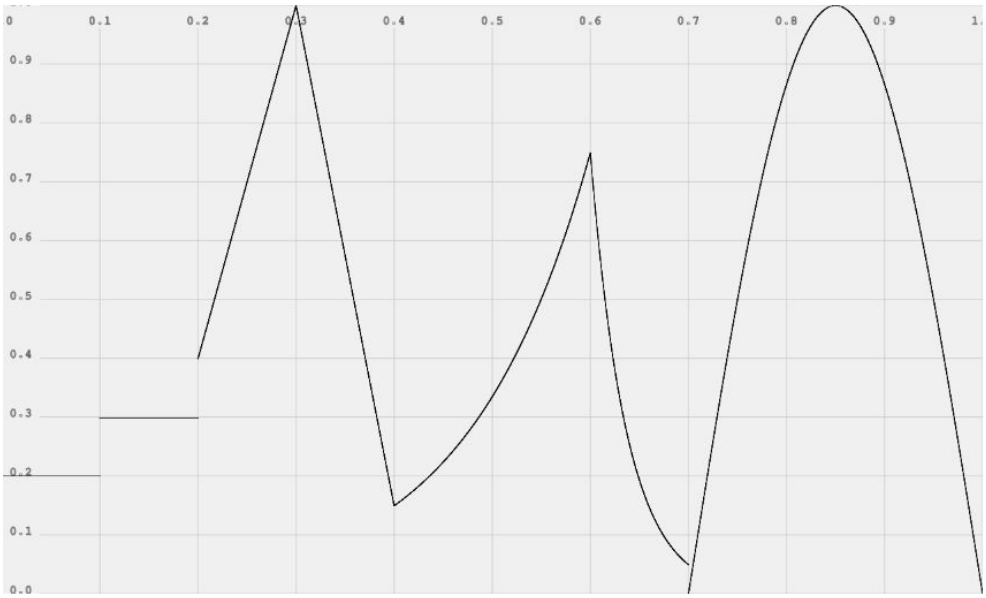
4.5.3 值的计算

computedValue 是用来控制音频数字信号处理的最终值，它音频渲染线程在每个渲染时间单位里计算。内部计算步骤如下：

1. 内部参数值会因为直接修改参数值或者被安排的参数变化（比如自动化事件）而重新计算。如果参数值是在一个自动化事件初始化之后被设置，那么该事件就会被移除。当读取这个参数的时候，返回的始终是内部计算的当前值。如果给定时间范围内自动化事件移除了，内部值将保持不变，直到下一次赋值或者时间范围内的自动化事件触发。
2. AudioParam 将取用任何连接到它的 AudioNode 的输出渲染数据，如果数据不是单声道的则用向下混音的方式将它转换为单声道，然后和其他类似的 AudioNode 输出数据混合到一起。如果没有 AudioNode 连接到其上，这个值会是 0，对 computedValue 没有影响
3. computedValue 是第二步中计算的内部值的和

4.5.4．AudioParam 自动化函数例子

Example




```
var t0 = 0;

var t1 = 0.1;

var t2 = 0.2;

var t3 = 0.3;

var t4 = 0.4;

var t5 = 0.6;

var t6 = 0.7;

var t7 = 1.0;


var curveLength = 44100;

var curve = new Float32Array(curveLength);

for (var i = 0; i < curveLength; ++i)

    curve[i] = Math.sin(Math.PI * i / curveLength);
```

```
param.setValueAtTime(0.2, t0);
param.setValueAtTime(0.3, t1);
param.setValueAtTime(0.4, t2);
param.linearRampToValueAtTime(1, t3);
param.linearRampToValueAtTime(0.15, t4);
param.exponentialRampToValueAtTime(0.75, t5);
param.exponentialRampToValueAtTime(0.05, t6);
param.setValueCurveAtTime(curve, t6, t7 - t6);
```

4.7 GainNode 接口

改变音频信号的增益是音频应用的基本操作。GainNode 是构建混音器的基本组成部分。这个接口是单输入、单输出的 AudioNode。

numberOfInputs : 1

numberOfOutputs : 1

channelCountMode = "max";

channelInterpretation = "speakers";

这个接口将输入音频信号乘以增益属性（可能是时变的），然后将结果复制到输出。默认情况下，此节点会将输入无变化的输出，这时是常量增益 1。

和其他 AudioParam 对比 ,增益参数代表了由时间到浮点值的映射。对输入的每个 PCM（脉冲编码调制）音频采样都会乘以和时间对应的增益值。这个乘积的结果就是 PCM 音频采样的输出。

输出的声道数总是和输入的声道数一致，输入的每个声道都会乘以增益值，然后复制到输出的响应声道中。

实际实现时应该将增益平滑的加入音频流，不应该导致引人注意的打击或者毛刺声。这个过程称作开拉链(?)

```
Web IDL

    interface GainNode : AudioNode {


        readonly attribute AudioParam gain;


    };
```

4.7.1 属性

gain：是增益的数值。默认值为 1（没有增益变化）。名义上它的最小值为 0，但是可以在相位反转的时候设置为负数。名义上最大值为 1，但是最大值也是可以的（不抛异常）。这个参数是 a-rate.

4.8 DelayNode 接口

延迟线是音频应用中的基本组成部分。这个接口是单输入单输出的 AudioNode：

```
        numberOfInputs  : 1
```

```
        numberOfOutputs : 1



        channelCountMode = "max";

        channelInterpretation = "speakers";
```

输入和输出的音频声道总是相等。

这个节点会将到来的音频信号延迟某个时间。比如，对每个时间 t，输入信号 input(t)，延迟时间 delayTime(t)以及输出信号 output(t) ,输出 output(t) = input(t-delayTime(t))。delayTime 的默认值为 0 秒，当延迟时间变化的时候，具体实现的时候需要让变换平滑一些，不应该在音频流中加入引人注意的打击或者毛刺声。

Web IDL

```
    interface DelayNode : AudioNode {



        readonly attribute AudioParam delayTime;



    };
```

4.8.1. Attributes

延迟时间：一个 AudioParam 对象代表了延迟的时间（单位为秒），默认值为 0。最小的值为 0，最大值在 AudioContext 的 createDelay 调用的时候传入的 maxDelayTime 决定。此参数是 a-rate.

4.9 AudioBuffer 接口

这个接口代表了贮存内存的音频内容（为单声音频和其他短音频剪辑）。接口的格式是非交叉的 IEEE32 位范围为-1~+1 的线性 PCM。它可以有一个或者多个声道。一般情况下 PCM 数据的长度都很短（一般都短于 1 分钟）。更长的声音，比如音乐音轨、流应该使用 Audio 元素和 MediaElementAudioSourceNode。

一个 AudioBuffer 可以被一个或者多个 AudioContext 使用。

Web IDL

```
interface AudioBuffer {

    readonly attribute float sampleRate;
    readonly attribute long length;

    // in seconds
    readonly attribute double duration;

    readonly attribute long numberOfChannels;

    Float32Array getChannelData(unsigned long channel);

};
```

4.9.1. Attributes

sampleRate：PCM 音频数据每秒样本的采样率

length：采样帧中的 PCM 音频数据长度

duration：PCM 音频数据秒长度

numberOfChannels：离散音频声道数量

4.9.2 方法和参数

getChannelDate 方法

返回 Float32Array 表示特定声道的 PCM 音频数据

channel 参数是序号，表示取得数据的特定声道。序号 0 表示第一个声道，序号需要小于 numberOfChannels，否则会抛出异常

4.10 AudioBufferSourceNode 接口

这个接口表示一个音频资源，音频是 AudioBuffer 中的贮存内存的音频资源。此接口用来播放那些需要高度灵活播放的短音频资源很有用（可以有节奏的回放）。start()方法在播放音频时调用，播放会在缓冲中的音频数据播放完时停止（如果循环属性设置为 false），或者当 stop()函数调用的时候也会停止播放。

numberOfInputs : 0

numberOfOutputs : 1

输出的声道数总是和赋值给.buffer 属性的 AudioBuffer 的声道数一致，或者当.buffer 为 NULL 时默认为 1 个声道

Web IDL

```
interface AudioBufferSourceNode : AudioNode {
```

```
        attribute AudioBuffer? buffer;

        readonly attribute AudioParam playbackRate;

        attribute boolean loop;
        attribute double loopStart;
        attribute double loopEnd;

        void start(optional double when = 0, optional double offset = 0, optional double duration);
        void stop(optional double when = 0);

        attribute EventHandler onended;

    };
```

4.10.1. 属性

buffer：用于播放的音频资源

playbackRate：渲染音频流的速度。默认为 1，此参数为单一速率

loop：表示音频数据是否应该循环播放。默认值为 false

loopStart：以秒为单位的可选值，当 loop 参数为 true 时表示从什么时间开始循环播放，将其设置为 0 到 duration 之间的值都有效

loopend :以秒为单位的可选值 ,当 loop 参数为 true 时表示从什么时间结束循环播放，将其设置为 0 到 duration 之间的值都有效

onended :用来设置事件句柄的属性 ,在事件派发到类型为 AudioBufferSourceNode 的节点的时候响应。当为 AudioSourceNode 播放 buffer 完毕的时候，将会执行该事件句柄。

4.10.2 方法和参数

start 方法

设置声音在某个时间播放。

when 参数描述声音在什么时候开始播放。它和 AudioContext.currentTime 属于同一个时间坐标系。如果值设置为 0 或者一个小于 currentTime 的值，那么声音会马上播放。start 应该只被调用一次而且需要在 stop 调用前调用，否则会抛出异常。

offset 参数描述了播放开始时间在 buffer 中的偏移时间。如果此参数没有传入，duration 将等于 AudioBuffer 的总 duration 减去 offset 参数。如果 offset 和和 duration 都没有设置，那么 duration 隐式的等于 AudioBuffer 的 duration。

stop 方法

设置声音在某个时间停止。

when 参数描述声音在什么时候停止播放。它和 AudioContext.currentTime 属于同一个时间坐标系。如果值设置为 0 或者一个小于 currentTime 的值，那么声音会马上停止播放。stop 应该只被调用一次而且需要在 stop 或者 start 调用后调用，否则会抛出异常。

4.10.3 looping

如果调用 start()的时候 loop 参数为 true，那么会无限播放，直到 stop()调用。我们称之为循环模式。播放总是会在调用 start()时的 offset 参数指定的 buffer 的位置开始播放，而且会一直循环播放下去直到播放到 buffer 的实际循环结束点，这时 buffer 重新回到实际循环起点，然后继续以循环模式播放。

在循环播放模式中，实际的循环点将使用 loopStart 和 loopEnd 属性，通过方式计算：

```
        if ((loopStart || loopEnd) && loopStart >= 0 && loopEnd > 0 && loopStart
< loopEnd) {
                actualLoopStart = loopStart;
                actualLoopEnd = min(loopEnd, buffer.length);
            } else {
                actualLoopStart = 0;
                actualLoopEnd = buffer.length;
        }
```

默认的 loopStart 和 loopEnd 都是 0，表示循环将从 buffer 的头部播放到尾部。

4.11 MediaElementAudioSourceNode 接口

表示一个来自 Audio 或者 Video 元素的音频资源

numberOfInputs : 0

numberOfOutputs : 1

输出的声道数和 HTMLMeidaElement 引用的媒体声道数一致。因此，改变媒体元素的.src 属性可以改变节点输出的声道数。如果.src 属性没有设置,输出的声道数将是默认的 1 个无声声道。

Web IDL

```
interface MediaElementAudioSourceNode : AudioNode {
};
```

MediaElementAudioSourceNode 是利用 HTMLMediaElement 和 AudioContext 的 createMediaElementSource() 方法创造的。

单输入的声道数，等于用参数 HTMLMediaElement 元素调用 createMediaElementSource()时的音频声道数 ,如果 HTMLMediaElement 没有音频的话声道数为 1。

一旦 MediaElementAudioSourceNode 对象创建了 ,HTMLMediaElement 必须保持一致的行为。除非渲染的音频不会再直接被听到，取而代之将变成由连接到 MediaElementAudioSourceNode 的路由图序列处理后的音频。因此暂停、seeking(？)、音量、.src 属性将改变，HTMLMediaElement 的其他方面需要和不使用 MediaElementAudioSourceNode 时保持一致。

```
var mediaElement = document.getElementById('mediaElementID');
```

```
var sourceNode = context.createMediaElementSource(mediaElement);

sourceNode.connect(filterNode);
```

4.12. ScriptProcessorNode 接口

是一个可以直接用 Javascript 产生、处理或者分析音频的接口。

```
numberOfInputs  : 1

numberOfOutputs : 1


channelCount = numberOfInputChannels;

channelCountMode = "explicit";

channelInterpretation = "speakers";
```

ScriptProcessorNode 接口构建时候的 bufferSize 必须是以下值 256, 512, 1024, 2048, 4096, 8192, 16384。这个值控制了 audioprocess 事件多频繁的被触发，以及每次触发的时候有多少采样帧需要处理。

audioprocess 事件只在 ScriptProcessorNode 至少有一个输入或者一个输出连接的时候触发。较小的 bufferSize 会有较小的延迟（更好），较大数字将需要避免音频卡断和毛刺。实际实现中如果 bufferSize 没有在初始化调用 createScriptProcessor 时传入需要设置默认值，或者设置为 0。

numberOfInputChannels 和 numberOfOutputChannels 决定了输入和输出的声道数。这两个数都为 0 是无效的。

```
var node = context.createScriptProcessor(bufferSize, numberOfInputChannels, numberOfOutputChannels);


interface ScriptProcessorNode : AudioNode {


    attribute EventHandler onaudioprocess;


    readonly attribute long bufferSize;


};
```

4.12.1. 属性

onaudioprocess：用来为 ScriptProcessNode 的 audioprocess 事件设置事件句柄。AudioProcessEvent 将派发给这个事件句柄

bufferSize：每次 onaudioprocess 触发的时候，需要处理的 buffer（在采样帧内)的大小。允许的值为(256, 512, 1024, 2048, 4096, 8192, 16384)

4.13. AudioProcessingEvent 接口

在 ScriptProcessorNode 上派发的事件对象。

这个事件的句柄通过读取 inputBuffer 属性里的音频数据来处理输入的音频。处理后的音频数据（如果没有输入的话则是合成的音频）将放到 outputBuffer 中。

Web IDL

```
interface AudioProcessingEvent : Event {

        readonly attribute double playbackTime;
        readonly attribute AudioBuffer inputBuffer;
        readonly attribute AudioBuffer outputBuffer;

};
```

4.13.1. 属性

playbackTime：音频播放的时间。此参数使得用 Javascript 直接处理音频和 context 渲染路由图中的其他事件能紧密同步。

inputBuffer :AudioBuffer 包含了输入音频数据。其声道数和 createScriptProcessor() 的参数 numberOfInputChannels 数量相等。

outputBuffer: 当有音频输出的时候会设置这个属性。声道数量等于 createScriptProcessor()方法的 numberOfOutputChannels 参数值。onaudioprocess 函数作用域内的代码会修改此 AudioBuffer 中的保存声道数据的 Float32Array 数组。任何在回调函数作用域外的代码修改 AudioBuffer 将不会产生听觉上的效果。

4.14. PannerNode 接口

此接口可以对输入的三维空间中的音频流进行处理。这个空间化和 AudioContext 的 AudioListener 相关

```
        numberOfInputs   : 1

        numberOfOutputs : 1



        channelCount = 2;

        channelCountMode = "clamped-max";

        channelInterpretation = "speakers";
```

输入的音频流可以是单声道或者立体声，决定于连接到输入音频。

输出节点硬编码为立体声（2 声道），而且不支持配置

Web IDL

```
    enum PanningModelType {

       "equalpower",

       "HRTF"

    };



    enum DistanceModelType {

       "linear",

       "inverse",

       "exponential"

    };
```

```
interface PannerNode : AudioNode {

    // Default for stereo is HRTF
    attribute PanningModelType panningModel;

    // Uses a 3D cartesian coordinate system
    void setPosition(double x, double y, double z);
    void setOrientation(double x, double y, double z);
    void setVelocity(double x, double y, double z);

    // Distance model and attributes
    attribute DistanceModelType distanceModel;
    attribute double refDistance;
    attribute double maxDistance;
    attribute double rolloffFactor;

    // Directional sound cone
    attribute double coneInnerAngle;
    attribute double coneOuterAngle;
    attribute double coneOuterGain;
```

```
};
```

4.14.2. 属性

"panningModel"：

决定使用哪种空间化算法来定位三维空间中的音频。默认为 HRTF（头相关变换函数）

"equalpower"：

一种简单高效的使用同等功率平移的空间化算法。

"HRTF"：

高质量的空间化算法，它使用测量到的人体反射的脉冲响应进行卷积。这种平移算法渲染的输出是立体声输出。

"distancemodel"：

决定当音源远离听者的时候用哪种算法来减小声源的声音大小。默认是”inverse“。

"linear"：

线性距离模式计算 distanceGain 的方式是：（rolloffFactor 为衰减因子）

1 - rolloffFactor * (distance - refDistance) / (maxDistance - refDistance)

"inverse"：

反转距离模型计算 distanceGain 的方式是：

refDistance / (refDistance + rolloffFactor * (distance - refDistance))

"exponential"：

指数距离模型计算 distanceGain 的方式是：

pow(distance / refDistance, -rolloffFactor)

refDistance：

当音源远离听者的时候用来减小音量的参考距离。默认值为 1

maxDistance：

音源和听者之间的最大距离，到达最大距离之后声音不会再变小。默认值为 10000。

rolloffFactor：

当音源远离听者的时候，描述声音减小的多快。默认值为 1

coneInnerAngle：

方向音源的参数，此参数是角度，此角度之内声音不会衰减。默认值为 360。

coneOuterAngle：

方向音源的参数，此参数是角度，此角度之外的声音将以 coneOuterGain 设定的常量值衰减。默认值是 360

coneOuterGain：

方向音源的参数，描述了 coneOuterAngle 之外的音频衰减的量。默认值为 0。

4.14.3 方法和参数

setPosition 方法

设置音源相对于听者的位置属性。使用三维直角坐标系。

用 x,y,z 参数表示三维坐标系。

默认置为(1,0,0)

setOrientation 方法

设置音源在三维直角坐标系所处的方向。决定于声音的方向（由 cone 属性控制），背向听者的声音可能很小或者听不见。

xyz 参数描述了三维空间的方向向量。

默认值为(1,0,0)

setVelocity 方法

设置音源的速度向量。向量控制了声音在三维空间移动的速度和方向。 音源和听者的相对速度决定了要使用多大的多普勒频移（音高变化）。此向量的单位是米每秒，而且和位置向量以及方向向量的单位无关。

xyz 参数描述一个方向向量，决定了音源移动的方向和强度。

默认值为（1,0,0）

4.15. AudioListener 接口

此接口描述了听到音频的人的位置和方向。所有的 PannerNode 对象都定位在相对于 AudioContext 监听者的位置。

Web IDL

```
interface AudioListener {

        attribute double dopplerFactor;
        attribute double speedOfSound;

        // Uses a 3D cartesian coordinate system
        void setPosition(double x, double y, double z);
        void setOrientation(double x, double y, double z, double xUp, double yUp,
double zUp);
        void setVelocity(double x, double y, double z);

    };
```

4.15.1. 属性

dopplerFactor：此常量决定了在多普勒效应发生的时候音高变化的值。默认为 1.

speedOfSound：用于计算多普勒效应的声速。默认值为 343.3

4.15.2 方法和参数

setPosition 方法：

设置听者在三维直角坐标系中的位置。PannerNode 对象使用这个相对于单一声源的位置值来空间化声音。

xyz 参数表示在三维空间里的坐标

默认值为（0,0,0）

setOrientation 方法：

描述了听者在三维直角坐标系中的方向。由两个向量来定义这个方向，一个是向上矢量、一个是向前矢量。通俗的说，向前矢量表示了人的鼻子正对的方向。向上矢量是人头部正对的方向。这两个向量的值应该是线性不相关的（相互成 90 度）.关于这两个值如何完成空间化,请看空间化部分.

xyz 参数表示了三维空间内的向前向量,默认值为(0,0,-1)

xup,yup,zup 参数表示了三维空间的向上向量.默认值为(0,1,0)

setVelocity 方法：

设置听者的速度向量.向量控制在三维空间中运动的方向和速度.这个速度相对于声源的速度值决定了多普勒频移的值(音高变化).此向量的单位为米每秒,和位置、方向向量的单位无关

xyz 参数描述一个方向向量，决定了音源移动的方向和强度。

默认值为（0,0,0）

4.16. ConvolverNode 接口

接口定义了一个进行音频处理的节点，用给定的脉冲响应函数产生线性卷积效果。对于多声道卷积矩阵参见这里。

numberOfOutputs : 1

```
channelCount = 2;
channelCountMode = "clamped-max";
channelInterpretation = "speakers";
```

Web IDL

```
interface ConvolverNode : AudioNode {

    attribute AudioBuffer? buffer;
    attribute boolean normalize;

};
```

4.16.1. 属性

buffer：

ConvolverNode 使用的单声道、立体声、4 声道的 AudioBuffer ,它包含脉冲响应(可能多声道) 。这个 AudioBuffer 必须和 AudioContext 采样率一致。一旦这个属性设置了值，buffer 和 normalize 属性会和给定归一化配置的脉冲响应一起用来设配置 ConvolverNode。此属性的初始化化值为 null

normalize：

当 buffer 属性设置的时候，控制 buffer 里的脉冲响应是否将按等功率缩放。默认值为 true，可以在不同脉冲响应下让 convolver 的输出的保持同一水平。如果此值为 false，那

么卷积变换将使用不做预处理/缩放的脉冲响应。改变这个值只会影响在此之后 buffer 属性设置。

设置 buffer 的时候，如果 normalize 属性是 false，ConvolverNode 将使用 buffer 里给出的脉冲响应执行线性卷积操作。

设置 buffer 的时候，如果 normalize 属性为 true，ConvolverNode 将首先对 buffer 中的音频数据进行 PMS 功率分析，来计算得出 normalizationScale，算法如下：

```
float calculateNormalizationScale(buffer)
    {
        const float GainCalibration = 0.00125;
        const float GainCalibrationSampleRate = 44100;
        const float MinPower = 0.000125;

        // Normalize by RMS power.
        size_t numberOfChannels = buffer->numberOfChannels();
        size_t length = buffer->length();

        float power = 0;

        for (size_t i = 0; i < numberOfChannels; ++i) {
```

```
            float* sourceP = buffer->channel(i)->data();
            float channelPower = 0;

            int n = length;
            while (n--) {
                float sample = *sourceP++;
                channelPower += sample * sample;
            }

            power += channelPower;
        }

        power = sqrt(power / (numberOfChannels * length));

        // Protect against accidental overload.
        if (isinf(power) || isnan(power) || power < MinPower)
            power = MinPower;

        float scale = 1 / power;

        // Calibrate to make perceived volume same as unprocessed.
```

```
        scale *= GainCalibration;


        // Scale depends on sample-rate.
        if (buffer->sampleRate())
            scale *= GainCalibrationSampleRate / buffer->sampleRate();


        // True-stereo compensation.
        if (buffer->numberOfChannels() == 4)
            scale *= 0.5;


        return scale;
    }
```

处理过程中，ConvolverNode 将此计算得到的 normalizationScale 值和用脉冲响应对输入进行线性卷积变换的结果相乘，产生最终结果。或者任何等量的运算也可以使用 ,比如先将 normalization 乘以输入，或者先将脉冲响应函数乘以 normalizationScale。

4.17. AnalyserNode 接口

此接口定义了一个可以进行实时频域和时域分析信息的节点。音频流将经过输入到节点再输出的过程中不会改变。

numberOfInputs : 1

numberOfOutputs : 1 Note that this output may be left unconnected.

```
channelCount = 1;
channelCountMode = "explicit";
channelInterpretation = "speakers";
```

Web IDL

```
interface AnalyserNode : AudioNode {

    // Real-time frequency-domain data
    void getFloatFrequencyData(Float32Array array);
    void getByteFrequencyData(Uint8Array array);

    // Real-time waveform data
    void getByteTimeDomainData(Uint8Array array);

    attribute unsigned long fftSize;
    readonly attribute unsigned long frequencyBinCount;

    attribute double minDecibels;
    attribute double maxDecibels;

    attribute double smoothingTimeConstant;
```

```
};
```

4.17.1. Attributes

fftSize：

用于进行频域分析的 FFT 快速傅立叶变换的大小。默认值为 2048，

frequencyBinCount：

fftsize 大小的一半

minDecibles：

在转换为无符号字节值时，fft 分析数据的缩放范围的最小指数。默认为-100。如果此属性的置为大于等于 maxDecibels，将抛出 INDEX_SIZE_ERR 的异常

maxDecibles：

在转换为无符号字节值时，fft 分析数据的缩放范围的最大指数。默认为-30。如果此属性的置为小于等于 minDecibels，将抛出 INDEX_SIZE_ERR 的异常

smoothingTimeConstant：

值范围为 0~1，0 表示没有时间和最终分析帧取平均。默认值为 0.8。如果值设置为小于 0 大于 1 会抛出 INDEX_SIZE_ERR 的异常

4.17.2 方法和参数

getFloatFrequencyData 方法：

拷贝当前频率数据到传入的浮点数组。如果这个数组比 frequencyBinCount 元素少，多余的元素将被放弃。如果数组比 frequencyBinCount 元素多，超出的元素将被忽略。

array 参数是频域分析数据将拷贝到的位置。

getByteFrequencyData 方法：

拷贝当前频率数据到传入的无符号字节数组。如果这个数组比 frequencyBinCount 元素少，多余的元素将被放弃。如果数组比 frequencyBinCount 元素多，超出的元素将被忽略。

array 参数是频域分析数据将拷贝到的位置。

getByteTimeDomainData 方法：

将当前的时域（波形）数据拷贝到传入的无符号字节数组。如果数组元素个数小于 fftSize，超出的元素将被放弃。如果数组元素个数大于 fftSize，超出的元素将被忽略。

array 参数是时域分析数据将拷贝到的位置。

4.18. ChannelSplitterNode 接口

ChannelSplitterNode 用作更高级应用，通常和 ChannelMergerNode 一起使用。

numberOfInputs : 1

numberOfOutputs : Variable N (defaults to 6) // number of "active" (non-silent) outputs is determined by number of channels in the input

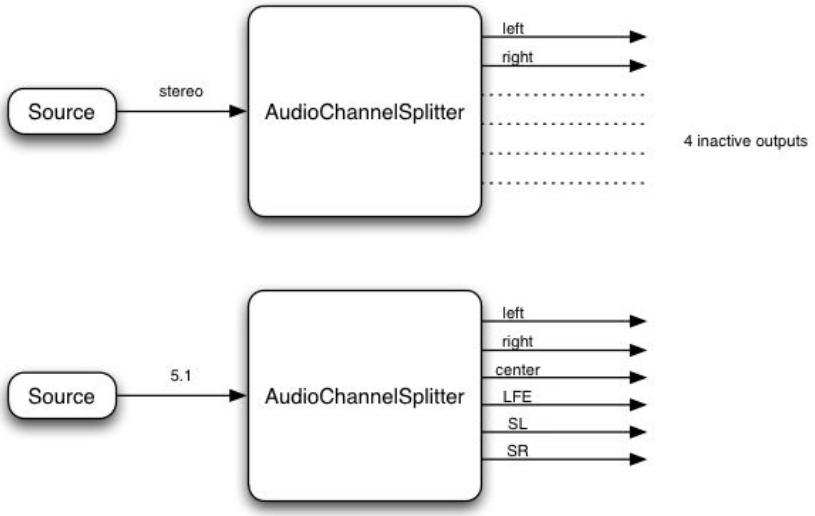
channelCountMode = "max";

channelInterpretation = "speakers";

此接口是一种 AudioNode，在路由图中它用于读取音频流的单一声道。此接口是单输入，有声输出的声道数量等于输入的音频流的声道数。例如，一个立体声输入连接到 ChannelSplitterNode 上，那么有声输出声道为 2 个（一个左声道一个右声道）。总输出

N（取决于 AudioContext 的 createChannelSplitter()调用的时候传入的参数 numberOfs），默认值为 6 如果初始化的时候没有传参数。那些无声的输出声道一般不会再和其他节点连接。

Example:



在这个列子中，分离器并不解释声道特征，只是简单的按照输入顺序分离声道。

ChannelSplitterNode 的一个应用场景是，在进行“矩阵混音”的时候，将分离后的单一声道传给独立增益控制。

Web IDL

```
interface ChannelSplitterNode : AudioNode {


};
```

4.19. ChannelMergerNode 接口

ChannelMergerNode 用于更高级的应用场景。通常和 ChannelSplitterNode 一起使用。

numberOfInputs : Variable N (default to 6) // number of connected inputs may be less than this

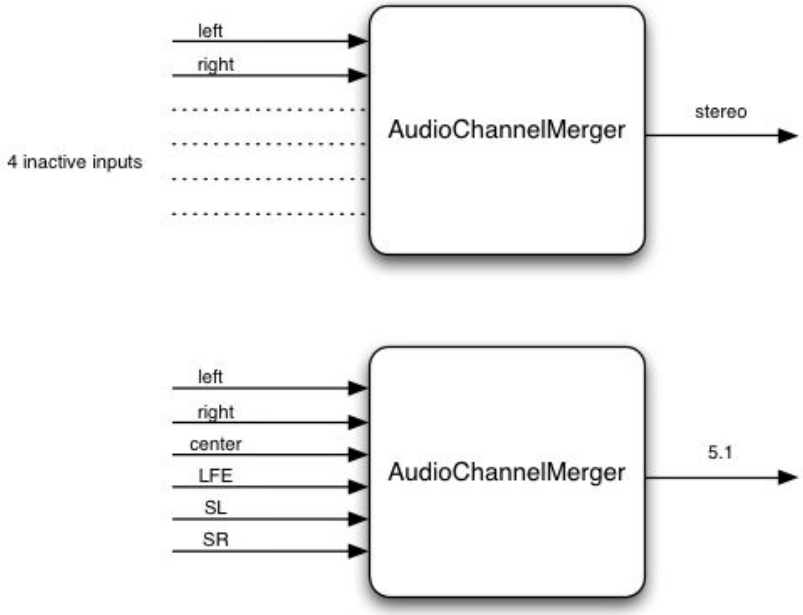
numberOfOutputs : 1

channelCountMode = "max";

channelInterpretation = "speakers";

它一种 AudioNode，可以将多个音频流的声道合并产生一个音频流。它可以有多个输入（默认值为 6），但不是所有这些都需要被连接。输出为单一音频流，它的声道数等于所有连接到输入的音频的声道和。例如，如果一个 ChannelMergerNode 有两个链接（都是立体声），那么输出将会是 4 声道，第一个输入的 2 个声道加上第二个输入的 2 个声道。另一个例子是，对于两个连接的单声道输入，输出为 2 声道（立体声）。第一个输入的声道决定了输出的左声道，第二个输入的声道是输出的右声道。

Example:



注意，上面的例子中，合并器没有解释声道的特征，只是按照顺序合并声道。

有些情况下连接到 ChannelMergerNode 的输入产生的输入音频流会有超过音频硬件最大支持的声道数。这种情况下当输出连接到 AudioContext 的.destination 时（音频硬件），多余的声道将被忽略。因此，ChannelMergerNode 使用的时候应该考虑当前场景是否可以解释产生声道数。

Web IDL

```
interface ChannelMergerNode : AudioNode {


};
```

4.20. DynamicsCompressorNode 接口

DynamicsCompressorNode 是应用了动态压缩效果的一种 AudioNode 处理节点。

动态压缩是在音乐生产和游戏音频中应用的很广泛。降低信号的最高音部分、提升最轻柔部分的音量。总的来说，一个较大声、较丰富、较完整的声音可以压缩。这在游戏和音乐应用中尤其重要，因为在这些场景中，大量的独立声音同时播放来控制整体信号水平以防输出到扬声器的音频失真。

```
numberOfInputs  : 1
numberOfOutputs : 1

channelCount = 2;
channelCountMode = "explicit";
channelInterpretation = "speakers";
```

Web IDL

```
interface DynamicsCompressorNode : AudioNode {

    readonly attribute AudioParam threshold; // in Decibels
    readonly attribute AudioParam knee; // in Decibels
    readonly attribute AudioParam ratio; // unit-less
    readonly attribute AudioParam reduction; // in Decibels
    readonly attribute AudioParam attack; // in Seconds
    readonly attribute AudioParam release; // in Seconds
```

```
};
```

4.20.1. 属性

threshold：分贝高于此值时，将会进行压缩。默认值为-24，标准范围为-100 到 0。

knee：当超出 threshold 设置的值之后，曲线在哪个点开始朝着 ratio 设置的部分平滑变换，默认值为 30，标准范围为 0~40。

ratio:输入增益变化多少来产生 1db 的输出。默认值为 12，标准范围为 1~20。

reduction:只读分贝值，用于计量。表示当前压缩器使用的增益压缩值。没有传入信号的时候值为 0（没有增益减少）。标准值为-20~0

attack:降低增益 10db 的时间（单位为秒），标准范围为 0~1

release:提升增益 10db 的时间（单位为秒），标准范围为 0~1

4.21 BiquadFilterNode 接口

定义了一个有处理功能的 AudioNode 节点，提供了常见的低阶滤波器功能。

低通滤波器是控制音调的基本组成部分（低音、中音、三重音），图形均衡器，以及其他更高级的滤波器。多个 BiquadFilterNode 结合使用可以产生更复杂的滤波器。滤波器参数比如“频率”可以在滤波器扫描的时候改变。每个 BiquadFilterNode 可以配置为下面 IDL 显示的常用滤波器中的一种。默认滤波器为“低通”

numberOfInputs ：1

numberOfOutputs : 1

channelCountMode = "max";

channelInterpretation = "speakers";

输出声道的数量总是等于输入声道的数量。

Web IDL

```
enum BiquadFilterType {
    "lowpass",
    "highpass",
    "bandpass",
    "lowshelf",
    "highshelf",
    "peaking",
    "notch",
    "allpass"
};

interface BiquadFilterNode : AudioNode {

    attribute BiquadFilterType type;
    readonly attribute AudioParam frequency; // in Hertz
    readonly attribute AudioParam detune; // in Cents
    readonly attribute AudioParam Q; // Quality factor
```

```
        readonly attribute AudioParam gain; // in Decibels


        void getFrequencyResponse(Float32Array frequencyHz,
                                  Float32Array magResponse,
                                  Float32Array phaseResponse);


};
```

滤波器类型简述如下。下面这些滤波器都是在音频处理的时候经常用到的。在具体实现的时候，他们都是由标准的模拟滤波器原型衍生出来的。更多的技术细节，我们推荐读者参考 Robert Bristow-Johnson 的手册。

所有的参数是 k-rate(可变？)默认值如下：

frequency：350Hz,标准区间为 10~奈奎斯特频率（采样率的一半）

Q：1, 标准区间为 0.0001 to 1000.

gain：0, 标准区间为 -40 to 40.

4.21.1 lowpass

低通滤波器允许截止频率以下的频率通过，截止频率以上的频率衰减。它应用了标准的二阶共振低通滤波器，每倍频程降低 12 分贝。

frequency：截止频率

Q：控制在截止频率处的响应峰值。更大的值响应峰值更大。注意对应此类滤波器，这个值不是传统的 Q，它是一个单位为分贝的谐振值。

gain：此滤波器型里不使用此参数

4.21.2 "highpass"

高通滤波器和低通滤波器相反。高于截止频率的频率能够通过，但是低于截止频率的频率不能通过。它应用了标准的二阶共振高通滤波器，每倍频程降低 12 分贝。

frequency：截止频率，高于此值的会消除

Q：控制在截止频率处的响应峰值。更大的值响应峰值更大。注意对应此类滤波器，这个值不是传统的 Q，它是一个单位为分贝的谐振值。

gain：此滤波器型里不使用此参数

4.21.3 bandpass

高通滤波器允许高于截止频率范围内的频率能够通过，低于截止频率的频率不能通过。它应用二阶滤波器。

frequency：频率带的中间值

Q：控制带宽度。当 Q 变大的时候宽度变窄。

gain：此滤波器型里不使用此参数

4.21.4 “lowshelf”

低架滤波器允许高频通过，对于低频频率会加上一个激励或者衰减。使用二阶低阻滤波器。

frequency：应用这个激励或者衰减的频率上限

Q：此滤波器型里不使用此参数

gain：激励值，单位为 dB。如果值为负，频率会衰减。

4.21.5 “highshelf”

高架滤波器和低架滤波器相反，允许所有的频率通过，对于高于设定频率的加一个激励。应用二阶高架滤波器。

frequency：会应用这个激励（或者衰减）的频率下限。

Q：此滤波器不使用此参数。

gain：激励，单位为 db。如果值为负，频率将会衰减。

4.21.6 “peaking”

波峰滤波器允许所有的频率通过，对一个范围内的频率加上一个激励或者衰减。

frequency：应用激励的频率范围的中心频率。

Q：控制被处理的频率范围的宽度。值越大应用的范围越窄。

gain：激励值，单位为 dB。如果值为负，频率会被衰减。

4.21.7 “notch”

陷波滤波器（也叫做带阻滤波器）和带通滤波器相反。允许除一个范围之外的所有频率通过。

frequency：陷波应用的频率中心值。

Q：控制影响的频率带宽度。值越大影响的范围越小。

gain：此滤波器不使用这个参数。

4.21.8 “allpass”

全通滤波器允许所有频率通过，会改变不同频率的相位关系。使用二阶全通滤波器。

frequency：进行相位变换的频率中心值。换个角度看，这是应用最大组延迟的频率。

Q：控制在中心频率的相位变换多尖锐。值越大，变换越尖锐、组延迟更大。

gain：此滤波器不使用这个参数。

4.21.9 方法

getFrequencyResponse 方法

给定当前滤波器参数设置，对于特定频率计算频率响应。

frequencyHZ 参数定义了一个要计算响应值的频率组成的数组。

magResponse 参数定义了输出数组，来接收线性幅度响应值。

phaseResponse 参数定义了输出数组，来接收相位响应值的弧度值。

4.22. WaveShaperNode 接口

WaveShaperNode 是一个 AudioNode 处理器，应用了非线性失真效果。

在处理音频，使得音效变得听起来微妙的温暖或者明显的失真效果时，非线性波形整形失真技术很常见。任何非线性整形波都可能使用。

numberOfInputs ：1

numberOfOutputs : 1

```
channelCountMode = "max";
channelInterpretation = "speakers";
```

输出声道数等于输入声道数。

Web IDL

```
enum OverSampleType {
    "none",
    "2x",
    "4x"
};

interface WaveShaperNode : AudioNode {

    attribute Float32Array? curve;
    attribute OverSampleType oversample;

};
```

4.22.1. 属性

curve：用整形波来达到波形整形效果。输入信号标准值为-1~+1。每个在这个范围内的输入采样将序列化到。小于-1 的采样值将。大于+1 的值将。

oversample：定义了在应用波形曲线的时候需要使用哪种过采样类型。默认值为”none“，表示曲线将直接应用到输入的采样中。”2x“和”4x“可以避免干扰信号来改进采样质量，”4x“产出最高的质量。对某些应用，最好不采用过采样来达到精确的整形曲线。

”2x“和”4x“表示一下步骤：

1. 将输入的采样提高到 AudioContext 的采样率的 2 倍或者 4 倍。因此对于每个处理块（128 采样数），产生 256 或者 512 个采样。
2. 使用整形曲线
3. 按照 AudioContext 的采样率将结果采样还原设定的采样数。就是用 256 或者 512 的采样样本，产生 128 的样本作为结果。

具体的向上采样和向下采样滤波器没有指定，可以根据音频质量、低延迟、以及性能调节。这是音频处理 dsp 的基本原则。

4.23. OscillatorNode 接口

OscillatorNode 表示能产生周期波形的音频源。可以设定为多种常用的波形。除此之外，还可以通过使用 PeriodicWave 对象来设定成任意的周期波形。

振荡器是音频合成的基本组成部分。OscillatorNode 将在 start()方法调用的时候产生声音。

用数学术语说，时间连续的周期波形可以有高频率密度的信息（从频域角度讲）。波形由特定的采样率采样为离散的数字音频信号，在将波形转换为数字形式之前需要保证较高的

频率信息比奈奎斯特频率高（采样频率的一半），否则异常的高频信号将变为镜像频率，产生让人反感的声音。这是音频 dsp 的基本原则。

为了避免毛刺音有些实践的方法需要实现。除去这些方法以外，数学上定义了理想的离散时间数字音频信号。实际实现的时候需要在成本（cpu 使用率）和对理想目标保真程度的完成度之间做一个权衡。

属性.frequency 和.detune 都是 a-rate 参数，一起使用来决定 computedFrequency 值：

```
computedFrequency(t) = frequency(t) * pow(2, detune(t) / 1200)

OscillatorNode 时间轴上的的连续相位是按时间聚合 computedFrequency

numberOfInputs  : 0

numberOfOutputs : 1 (mono output)
```

Web IDL

```
enum OscillatorType {
    "sine",
    "square",
    "sawtooth",
    "triangle",
    "custom"
};
```

```
interface OscillatorNode : AudioNode {

    attribute OscillatorType type;

    readonly attribute AudioParam frequency; // in Hertz
    readonly attribute AudioParam detune; // in Cents

    void start(double when);
    void stop(double when);
    void setPeriodicWave(PeriodicWave periodicWave);

    attribute EventHandler onended;

};
```

4.23.1. 属性

type：决定了周期波形的形状，可以直接设置为常量值（非”custom“）。setPeriodicWave()方法可以用来设置一个通用的波形，使得此属性置为”custom“。默认值为”sine“。

frequency：周期波形的频率，单位为赫兹。默认值为 440。此参数为 a-rate。

detune：失谐值将 frequency 偏移给定值。默认值为 0。参数是 a-rate

onended：属性用来设置派发到 OscillatorNode 的 ended 事件的事件句柄。当播放 OscillatorNode 的 buffer 结束的时候，ended 事件将被派发。

4.23.4 方法和参数

setPeriodicWave 方法：设置任意自定义周期波形的 PeriodicWave

start 方法：在 AudioBufferSourceNode 中已定义

stop 方法：在 AudioBufferSourceNode 中已定义

4.24. PeriodicWave 接口

PeriodicWave 表示 OscillatorNode 使用的任意周期波形。参见 createPeriodicWave() 和 setPeriodicWave()的更多细节。

Web IDL

```
interface PeriodicWave {


};
```

4.25. MediaStreamAudioSourceNode 接口

这个接口代表了来源于 MediaStream 的音频源。MediaStream 的第一个 AudioMediaStreamTrack 将用于音频源。

numberOfInputs : 0

numberOfOutputs : 1

输出响应的声道数和 AudioMediaStreamTrack 的声道数相等。如果没有有效音频音轨，输出声道数将为一个静音声道。

Web IDL

```
interface MediaStreamAudioSourceNode : AudioNode {


};
```

4.26. MediaStreamAudioDestinationNode 接口

接口是音频目的节点，是具有单 AudioMediaStreamTrack 的媒体流。媒体流当节点创建的时候创建，并且可以通过 stream 属性读取。这个流可以用 getUserMedia()的方式来获得，比如，可以使用 RTCPeerConnection 的 addStream()方法来发送到远程端。

```
    numberOfInputs  : 1
    numberOfOutputs : 0


    channelCount = 2;
    channelCountMode = "explicit";
    channelInterpretation = "speakers";
```

输入声道的数量默认为 2（立体声）。任何到输入的连接将按照输入的设置进行向上或者向下混音。

Web IDL

```
interface MediaStreamAudioDestinationNode : AudioNode {

    readonly attribute MediaStream stream;

};
```

4.26.1. Attributes

Stream

媒体流包含了一个单一的 AudioMediaStreamTrack，媒体流的声道数和节点本身的声道数一样

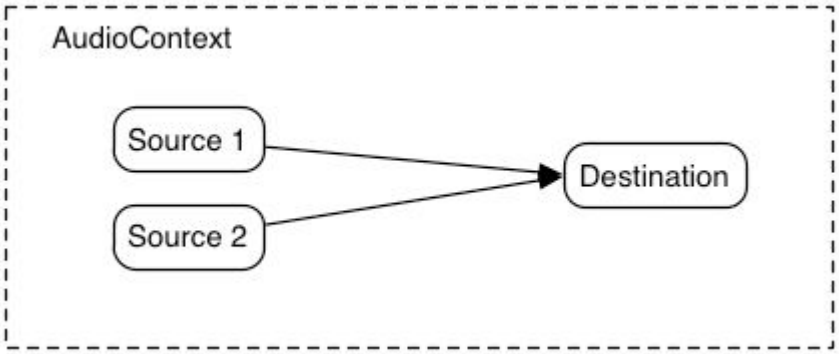
6 混音器增益结构

背景

当处理音频处理图的时候需要考虑的最重要的一点是如何能够在多个点调整增益（音量）。例如，一个标准的混音台模型，每个输入总线都有前置增益、后置增益、发送增益。编组总线和控制输出总线同样有增益控制。这里描述的增益控制可以用于实现标准的混音台以及其他架构。

输入求和

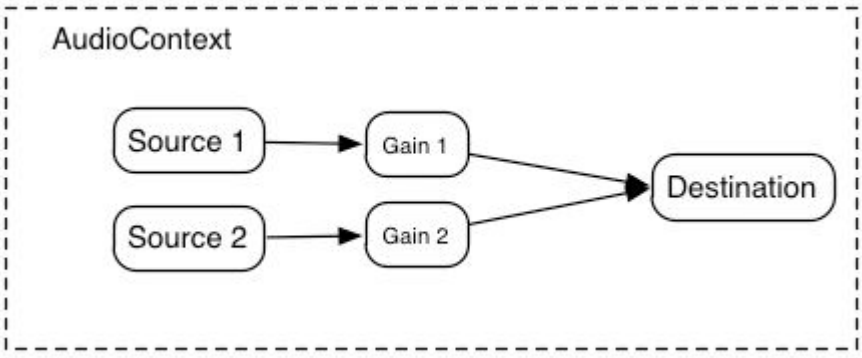
AudioNode 的输入可以和多个节点的输出相连。输入就像单位增益求和联结点，每个输入加在一起产生输出。



如果输出声道布局不一致，则需要使用混音（一般向上混合），参考混音原则。

增益控制

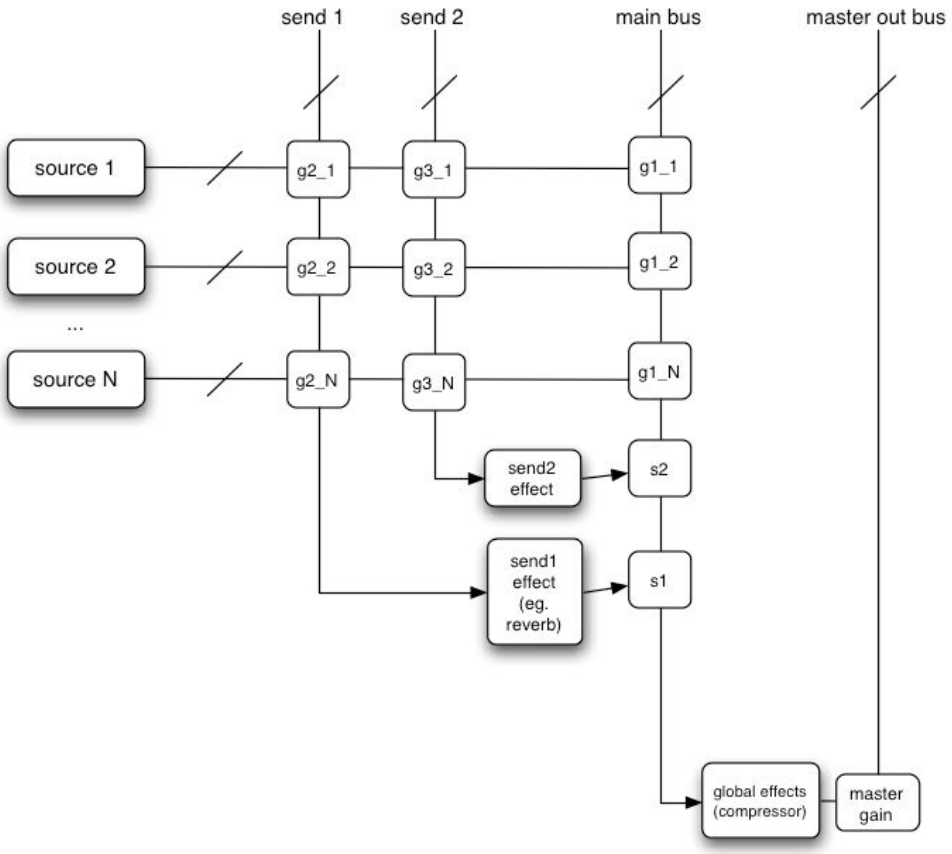
大部分时候需要单独控制输出信号的增益。GainNode 做下面的控制操作：



使用单位增益求和联结点和 GainNode 的概念，可以构建简单或者复杂的混音方案。

例子：混音器和辅助总线

在一个包含多辅助总线混音器、多编组总线混音器的路由场景中。为了简单起见，前置增益控制和插入音效的环节没有列出。



词图使用缩写标志”send1“、”send2“、”main bus“表示实际的输入到 AudioNode 内容，但是这里作为加法总线，交叉点为 g2_1,g3_1,等等，表示对给定混音器输入的音源的增益或者音量。为了操作增益，需要一个 GainNode 节点：

下面的代码表示了如何用 js 构建上面的图：

```
var context = 0;

var compressor = 0;

var reverb = 0;

var delay = 0;

var s1 = 0;
```

```
var s2 = 0;


var source1 = 0;
var source2 = 0;
var g1_1 = 0;
var g2_1 = 0;
var g3_1 = 0;
var g1_2 = 0;
var g2_2 = 0;
var g3_2 = 0;


// Setup routing graph
function setupRoutingGraph() {
    context = new AudioContext();


    compressor = context.createDynamicsCompressor();


    // Send1 effect
    reverb = context.createConvolver();
    // Convolver impulse response may be set here or later
```

```
// Send2 effect
delay = context.createDelay();

// Connect final compressor to final destination
compressor.connect(context.destination);

// Connect sends 1 & 2 through effects to main mixer
s1 = context.createGain();
reverb.connect(s1);
s1.connect(compressor);

s2 = context.createGain();
delay.connect(s2);
s2.connect(compressor);

// Create a couple of sources
source1 = context.createBufferSource();
source2 = context.createBufferSource();
source1.buffer = manTalkingBuffer;
source2.buffer = footstepsBuffer;
```

```
// Connect source1
g1_1 = context.createGain();
g2_1 = context.createGain();
g3_1 = context.createGain();
source1.connect(g1_1);
source1.connect(g2_1);
source1.connect(g3_1);
g1_1.connect(compressor);
g2_1.connect(reverb);
g3_1.connect(delay);

// Connect source2
g1_2 = context.createGain();
g2_2 = context.createGain();
g3_2 = context.createGain();
source2.connect(g1_2);
source2.connect(g2_2);
source2.connect(g3_2);
g1_2.connect(compressor);
g2_2.connect(reverb);
g3_2.connect(delay);
```

```
        // We now have explicit control over all the volumes g1_1, g2_1, ..., s1, s2
        g2_1.gain.value = 0.2;   // For example, set source1 reverb gain

         // Because g2_1.gain is an "AudioParam",
         // an automation curve could also be attached to it.
         // A "mixing board" UI could be created in canvas or WebGL controlling these gains.
    }
```

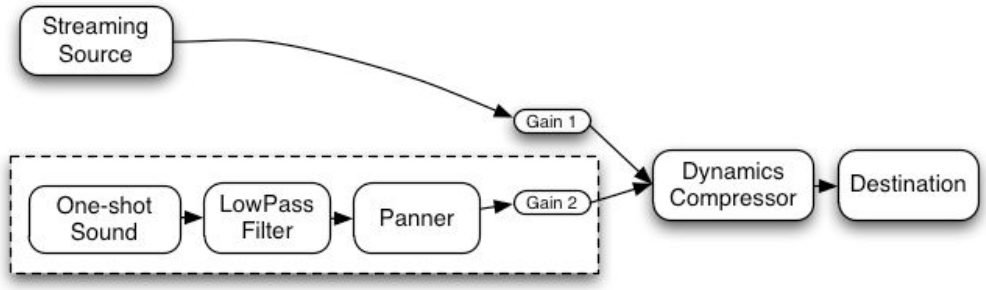
7.动态生命周期

除了静态路由配置，有时候也需要能对动态分配有限生命周期的声音进行自定义音效路由处理。在这个讨论中我们称这个声音为音符。许多音频应用包含了音符的概念，比如鼓乐、序列器、包含很多游戏过程中触发单点音的三维游戏。

传统软件合成中，音符从资源池中动态分配和释放。当收到乐器数字接口的 note-on（音符开启）的信号，音符就被分配。当音符播放完成的时候、或者到达采样终点（非循环模式）时将稳定在波封的 0 相位处、或者当 MIDI（乐器数字接口）的 note-off（音符关闭）信号将音符放到波封的释放相位时，音符将被释放。在 MIDI 的 note-off 情况下，音符不

会立即释放，只有当释放波封的相位到达的时候才释放音符。任何时候，可以同时存在很多音符在播放，但是随着新音符加入路由图和旧音符的释放，音符的组合一直在变化。

音频系统自动处理每个音符事件在路由图中的路径。一个音符由一个 AudioBufferSourceNode 表示，它可以和其他处理节点相连。当音符完成了播放，context 将自动释放对于 AudioBufferSourceNode 的引用，同时也会释放任何和这个节点的连接。节点将自动从路由图中断掉连接，并在没有任何引用的时候被删除。路由图中的长期存在的、被音频共享的节点可以被显式操作。尽管听起来很复杂，这些都不需要额外的 Javascript 处理。



一个低通滤波器、一个相位处理、一个增益节点直接和单点触发的声音连接。所以当该声音完成了播放，context 将会自动释放这些节点（虚线框内的所有节点）。如果没有任何其他 Javascript 引用到这个声音也没有其他节点连接到这个声音节点，它将立即从路由图上删除。音频流资源有一个全局引用，将会保持引用直到显式解除连接。Javascript 的实现如下：

```
var context = 0;
var compressor = 0;
var gainNode1 = 0;
```

```
    var streamingAudioSource = 0;


    // Initial setup of the "long-lived" part of the routing graph
    function setupAudioContext() {
        context = new AudioContext();


        compressor = context.createDynamicsCompressor();
        gainNode1 = context.createGain();


        // Create a streaming audio source.
        var audioElement = document.getElementById('audioTagID');
        streamingAudioSource =
context.createMediaElementSource(audioElement);
        streamingAudioSource.connect(gainNode1);


        gainNode1.connect(compressor);
        compressor.connect(context.destination);
    }


    // Later in response to some user action (typically mouse or key event) // a
one-shot sound can be played.
```

```
function playSound() {
    var oneShotSound = context.createBufferSource();
    oneShotSound.buffer = dogBarkingBuffer;

    // Create a filter, panner, and gain node.
    var lowpass = context.createBiquadFilter();
    var panner = context.createPanner();
    var gainNode2 = context.createGain();

    // Make connections
    oneShotSound.connect(lowpass);
    lowpass.connect(panner);
    panner.connect(gainNode2);
    gainNode2.connect(compressor);

    // Play 0.75 seconds from now (to play immediately pass in 0)
    oneShotSound.start(context.currentTime + 0.75);
}
```

9.声道的向上混音和向下混音

混音器增益结构描述了 AudioNode 节点的输入如何和一个或多个 AudioNode 的输出连接。这些输出的每个连接都是具有非零声道数的流。一个输入有混合的规则来合并连接到它上的所有声道。举个简单的例子，如果一个单声道输出和一个立体声输出连接到同一个输入上，那么单声道将变为一个立体声然后和另一个立体声相加。但是，为每一个连接到 AudioNode 的输入定义具体的混合规则很重要。对于输入来说有默认的混合规则，所以在使用的时候不用太担心细节，尤其是单声道和立体声流混音的时候。但是规则可以在高级应用场景里改变，尤其是对于多声道的混音。

up-mixing 是指将声道数较少的流转变为具有较多声道的流。

down-mixing 是指将声道数较多的流转变为具有较少声道的流。

AudioNode 属性包含了声道向上混音和向下混音的规则，如前面章节描述的。下面是更精确的定义：

- 声道数：用于计算 computeNumberOfChannels
- channelCountMode：决定了 computedNumberOfChannels 如何计算，一旦这个数字计算好了，所有的连接将按照这个数量进行向上或者向下混音。大部分时候，这个默认值都是”max“（取最大值）
    - ”max“computedNumberOfChannels 按照所有连接的最大声道数来计算。此模式下 channelCount 被忽略。
    - ”clamped-max“等于”max“，以 channelCount 为上限
    - ”explicit“：computedNumberOfChannels 取 channelCount 的值

- channelInterpretation：决定每个独立声道如何处理。例如，是按照有特定布局的扬声器处理，还是按照简单的离散声道处理？这个值影响了向上和向下混音如何工作。默认值是”speakers“。
    - ” speakers “:使用下面的混音规则进行处理。如果声道数量不符合混音规则里列出的基本扬声器布局，将其转换为”discrete “
    - ” discrete “：向上混音，填充声道直到没有更多数据填充，则填充 0 到声道里。向下混音，尽可能多的填充声道，然后弃掉多余的声道。

对于 AudioNode 的每个输入，实现步骤是：

- 计算 computedNumberOfChannels
- 对于每个连接的输入：
    - 按照 channelInterpretation 的规则将连接的声道数处理为 computedNumberOfChannels
    - 将每个处理后的输入混合。每个对应位置的声道混合在一起。

9.1 扬声器声道布局

当 channelInterpretation 是”speaker“时，对这种特定声道布局有具体混音的规则。对于此声道布局来说定义声道的顺序很重要。

目前只考虑单声道、立体声、4 声道、5.1.以后还可以定义其他布局。

9.1.1.声道顺序

Mono 单声道

0: M: mono

Stereo 立体声

0: L: left

1: R: right

Quad 四声道

0: L: left

1: R: right

2: SL: surround left

3: SR: surround right

5.1

0: L: left

1: R: right

2: C: center

3: LFE: subwoofer

4: SL: surround left

5: SR: surround right

9.1.2. 向上混音扬声器布局

Mono up-mix:单声道向上混音

1 -> 2 : up-mix from mono to stereo 单声道到立体声

```
output.L = input;
output.R = input;
```

1 -> 4 : up-mix from mono to quad 立体声到四声道

```
output.L = input;
output.R = input;
output.SL = 0;
output.SR = 0;
```

1 -> 5.1 : up-mix from mono to 5.1 单声道到 5.1

```
output.L = 0;
output.R = 0;
output.C = input; // put in center channel
output.LFE = 0;
output.SL = 0;
output.SR = 0;
```

Stereo up-mix:立体声向上混音

2 -> 4 : up-mix from stereo to quad 立体声到四声道

output.L = input.L;

output.R = input.R;

output.SL = 0;

output.SR = 0;

2 -> 5.1 : up-mix from stereo to 5.1 立体声到 5.1

output.L = input.L;

output.R = input.R;

output.C = 0;

output.LFE = 0;

output.SL = 0;

output.SR = 0;

Quad up-mix:四声道向上混音

4 -> 5.1 : up-mix from quad to 5.1 四声道到 5.1

output.L = input.L;

output.R = input.R;

output.C = 0;

output.LFE = 0;

output.SL = input.SL;

output.SR = input.SR;

9.1.3. 向下混音扬声器布局

向下混音实际中也有需要，比如处理 5.1 类型的音源，但是以立体声播放

Mono down-mix:向下混音为单声道

2 -> 1 : stereo to mono

output = 0.5 * (input.L + input.R);

4 -> 1 : quad to mono

output = 0.25 * (input.L + input.R + input.SL + input.SR);

5.1 -> 1 : 5.1 to mono

output = 0.7071 * (input.L + input.R) + input.C + 0.5 * (input.SL + input.SR)

Stereo down-mix:向下混音为立体声

4 -> 2 : quad to stereo

output.L = 0.5 * (input.L + input.SL);

output.R = 0.5 * (input.R + input.SR);

5.1 -> 2 : 5.1 to stereo

output.L = L + 0.7071 * (input.C + input.SL)

output.R = R + 0.7071 * (input.C + input.SR)

Quad down-mix:向下混音为四声道

5.1 -> 4 : 5.1 to quad

output.L = L + 0.7071 * input.C

output.R = R + 0.7071 * input.C

output.SL = input.SL

output.SR = input.SR

9.2. Channel Rules Examples

此部分是陈述类的。

```
// Set gain node to explicit 2-channels (stereo).
gain.channelCount = 2;
gain.channelCountMode = "explicit";
gain.channelInterpretation = "speakers";

// Set "hardware output" to 4-channels for DJ-app with two stereo output busses.
```

```
context.destination.channelCount = 4;
context.destination.channelCountMode = "explicit";
context.destination.channelInterpretation = "discrete";

// Set "hardware output" to 8-channels for custom multi-channel speaker array
// with custom matrix mixing.
context.destination.channelCount = 8;
context.destination.channelCountMode = "explicit";
context.destination.channelInterpretation = "discrete";

// Set "hardware output" to 5.1 to play an HTMLAudioElement.
context.destination.channelCount = 6;
context.destination.channelCountMode = "explicit";
context.destination.channelInterpretation = "speakers";

// Explicitly down-mix to mono.
gain.channelCount = 1;
gain.channelCountMode = "explicit";
gain.channelInterpretation = "speakers";
```

11.声音定位/位移

三维游戏的常见特征是动态定位声音以及能在三维空间中移动多个音频源。游戏音频引擎比如 OpenAL, FMOD, Creative's EAX, Microsoft's XACT Audio。

使用 PannerNode，音频流可以设定空间内相对于 AudioListener 的位置。AudioContext 包含一个确定的 AudioListener。平移节点和听者都在一个三维空间的直角坐标系中。直角坐标系的单位没有定义，也不需要定义因为计算出的效果独立于单位。PannerNode 对象（代表音频流）有代表声音发射的方向向量，同时还有一个音锥。比如声音可以全向发射的，表示任何方向都可以听到；也可以是方向声音，表示只有在正对着听者的时候才能听到。AudioListener 对象（代表人耳）有一个方向和向上的方向向量表示人脸对着的方向。因为音源和听者都在移动，他们都有速度向量表示速度值和移动方向。相互作用下，两个向量可以产生多普勒频移从而影响音高。

在渲染的时候，PannerNode 会计算方向角和仰角。这两个值是内部变量来渲染空间定位效果。参见 Panning Algorithm 节来看这些值如何使用。

计算仰角和方向角的算法如下：（cross 为向量叉乘，dot 为向量点乘）

```
// Calculate the source-listener vector.
vec3 sourceListener = source.position - listener.position;

if (sourceListener.isZero()) {
    // Handle degenerate case if source and listener are at the same point.
    azimuth = 0;
```

```
        elevation = 0;
        return;
    }

    sourceListener.normalize();

    // Align axes.
    vec3 listenerFront = listener.orientation;
    vec3 listenerUp = listener.up;
    vec3 listenerRight = listenerFront.cross(listenerUp);
    listenerRight.normalize();

    vec3 listenerFrontNorm = listenerFront;
    listenerFrontNorm.normalize();

    vec3 up = listenerRight.cross(listenerFrontNorm);

    float upProjection = sourceListener.dot(up);

    vec3 projectedSource = sourceListener - upProjection * up;
    projectedSource.normalize();
```

```
azimuth = 180 * acos(projectedSource.dot(listenerRight)) / PI;


// Source in front or behind the listener.
double frontBack = projectedSource.dot(listenerFrontNorm);
if (frontBack < 0)
    azimuth = 360 - azimuth;


// Make azimuth relative to "front" and not "right" listener vector.
if ((azimuth >= 0) && (azimuth <= 270))
    azimuth = 90 - azimuth;
else
    azimuth = 450 - azimuth;


elevation = 90 - 180 * acos(sourceListener.dot(up)) / PI;


if (elevation > 90)
    elevation = 180 - elevation;
else if (elevation < -90)
    elevation = -180 - elevation;
```

- 平移算法

支持单声道—>立体声和立体声—>立体声平移。

只有所有连接都是单声道的时候才使用单声道—>立体声。

- 等指数平移（基于向量）

这是一个简单的相对代价较低的算法，可以产出基本而有效的结果。这个算法在平移音频源的时候比较常用。

这个平移算法忽略了仰角值。

用下面的步骤进行处理:

```
azimuth(方向角) 的值初始范围为 -90 <= azimuth <= +90 :      // Clamp azimuth to allowed range of -180 -> +180.
azimuth = max(-180, azimuth);
azimuth = min(180, azimuth);

// Now wrap to range -90 -> +90.
if (azimuth < -90)
    azimuth = -180 - azimuth;
else if (azimuth > 90)
    azimuth = 180 - azimuth;
```

0 -> 1 的归一化值是由单声道到立体声处理过程中由 azimuth 计算得出的：

```
x = (azimuth + 90) / 180
```

对于 stereo->stereo 是：

```
if (azimuth <= 0) { // from -90 -> 0
    // inputL -> outputL and "equal-power pan" inputR as in mono case
    // by transforming the "azimuth" value from -90 -> 0 degrees into the range -90 -> +90.
    x = (azimuth + 90) / 90;
} else { // from 0 -> +90
    // inputR -> outputR and "equal-power pan" inputL as in mono case
    // by transforming the "azimuth" value from 0 -> +90 degrees into the range -90 -> +90.
    x = azimuth / 90;
}
```

然后计算左右声道的增益值：

```
gainL = cos(0.5 * PI * x);
gainR = sin(0.5 * PI * x);
```

对于 mono->stereo，输出值计算如下:

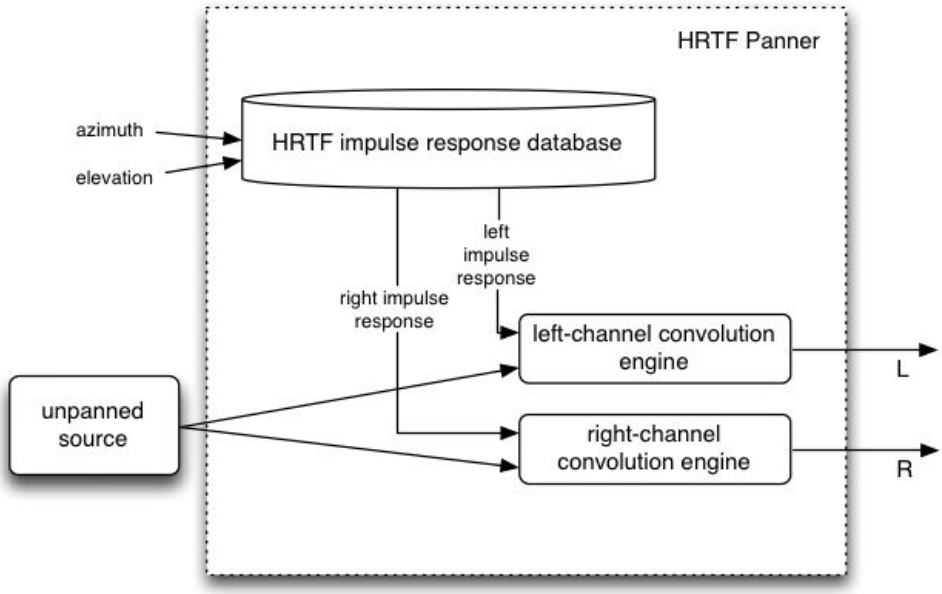
```
outputL = input * gainL

outputR = input * gainR
```

对于 stereo->stereo,输出计算如下:

```
if (azimuth <= 0) { // from -90 -> 0

    outputL = inputL + inputR * gainL;

    outputR = inputR * gainR;

} else { // from 0 -> +90

    outputL = inputL * gainL;

    outputR = inputR + inputL * gainR;

}
```

- HRTF 平移 (仅用于立体声)

需要在仰角和方向角里记录一系列 HRTF 脉冲响应。有一些开源的脉冲响应可以使用。实现的时候需要使用一个高度优化的卷积函数。比等功率更消耗资源，但是提供了更空间化的声音。



距离效果

越近的声音越大，越远的声音越小。声音音量如何按照离听者的距离变化决定于 distanceModel 属性。

在音频渲染的过程中，距离值将基于平移和听者的位置计算：

v = panner.position - listener.position

distance = sqrt(dot(v, v))

这个距离然后将用于计算 distanceGain，它取决于 distanceModel 属性。

作为处理的一部分，PannerNode 会用 distanceGain 缩放/乘以输入音频信号，来让远距离的声音更安静或者近距离的声音更大。

音锥

听者和每个音源都有方向向量，来描述他们面朝的方向。每个音源的声音辐射特征由内部和外部的锥来描述，音锥将声音的密度描述为音源/听者相对于音源方向向量的角度。因此，直接指向听者的音源将比其他方向大。音源也可以是全范围辐射。

下面的算法需要用来计算音锥效果的增益贡献，给定音源（PannerNode）和听者：

```
    if (source.orientation.isZero() || ((source.coneInnerAngle == 360) &&
(source.coneOuterAngle == 360)))
        return 1; // no cone specified - unity gain

    // Normalized source-listener vector
    vec3 sourceToListener = listener.position - source.position;
    sourceToListener.normalize();

    vec3 normalizedSourceOrientation = source.orientation;
    normalizedSourceOrientation.normalize();

    // Angle between the source orientation vector and the source-listener vector
    double dotProduct = sourceToListener.dot(normalizedSourceOrientation);
    double angle = 180 * acos(dotProduct) / PI;
    double absAngle = fabs(angle);
```

```
// Divide by 2 here since API is entire angle (not half-angle)
double absInnerAngle = fabs(source.coneInnerAngle) / 2;
double absOuterAngle = fabs(source.coneOuterAngle) / 2;
double gain = 1;


if (absAngle <= absInnerAngle)
    // No attenuation
    gain = 1;
else if (absAngle >= absOuterAngle)
    // Max attenuation
    gain = source.coneOuterGain;
else {
    // Between inner and outer cones
    // inner -> outer, x goes from 0 -> 1
    double x = (absAngle - absInnerAngle) / (absOuterAngle - absInnerAngle);
    gain = (1 - x) + source.coneOuterGain * x;
}


return gain;
```

多普勒频移

- 引入了音高调节来模拟真实的移动音源

- 决定于音源和听者的速度向量，声音的速度、多普勒因子

需要用下面的算法来计算多普勒频移，用来作为所有连接到 AudioNodePannerNode 的 AudioNodeBufferSourceNode 的附加的播放比例控制器。

```
double dopplerShift = 1; // Initialize to default value
double dopplerFactor = listener.dopplerFactor;

if (dopplerFactor > 0) {
    double speedOfSound = listener.speedOfSound;
    // Don't bother if both source and listener have no velocity.
    if (!source.velocity.isZero() || !listener.velocity.isZero()) {
        // Calculate the source to listener vector.
        vec3 sourceToListener = source.position - listener.position;

        double sourceListenerMagnitude = sourceToListener.length();

        double listenerProjection = sourceToListener.dot(listener.velocity) /
sourceListenerMagnitude;
        double sourceProjection = sourceToListener.dot(source.velocity) /
sourceListenerMagnitude;
```

```
                listenerProjection = -listenerProjection;

                sourceProjection = -sourceProjection;


                double scaledSpeedOfSound = speedOfSound / dopplerFactor;

                listenerProjection = min(listenerProjection, scaledSpeedOfSound);

                sourceProjection = min(sourceProjection, scaledSpeedOfSound);


                dopplerShift = ((speedOfSound - dopplerFactor * listenerProjection) /
(speedOfSound - dopplerFactor * sourceProjection));

                fixNANs(dopplerShift); // Avoid illegal values


                // Limit the pitch shifting to 4 octaves up and 3 octaves down.

                dopplerShift = min(dopplerShift, 16);

                dopplerShift = max(dopplerShift, 0.125);

            }

        }
```

12.使用卷积的线性效果

背景：

卷积是数学处理过程，可以应用到音频信号上来产生有趣的高质量线性效果.通常,这些效果用来模拟声学的空间比如音乐厅、教堂、或者露天剧场。它还可以用来产生复杂的滤波效果，像是从封闭空间内部传出的包裹住的声音效果，水下的声音，电话里来的声音，或者老式的箱式扬声器的声音。这个技术通常用在大部分电影、音乐作品里，产生多种多样并且高质量的效果。

每种独特的效果都是由某种脉冲响应决定的。一种脉冲响应可以是一个音频文件，并且可以从实际的声学空间比如山洞里录音得到，也可以由多种技术合成产生。

作为标准使用的目的：

许多游戏音频引擎的主要特征（OpenAL, FMOD, Creative's EAX, Microsoft's XACT Audio, etc.）是用混响效果来模拟声学空间里的声音。但是用于产生效果的代码一般都是自定义或者算法化的（一般使用手动调整的延迟线以及全通滤波器相互连接和反馈）。在近乎所有的情况下，不仅仅是自定义的实现，代码是专属并且闭源的，每个公司都在实现其独特的质量的时候施加了神秘的魔法。每种实现都有不同参数以至于很难到达统一的效果。代码的版权使得很难将其变为标准。除此之外算法上的混响效果只能实现比较小范围的不同音效，不管参数如何微调。

卷积效果使用精确的数学算法作为其处理基础来解决这类问题。脉冲响应是用于音频流的特定音效，并且可以保存为音频文件，进而可以由 url 引用。并且卷积可以产生非常多种类的效果。

实现指南：

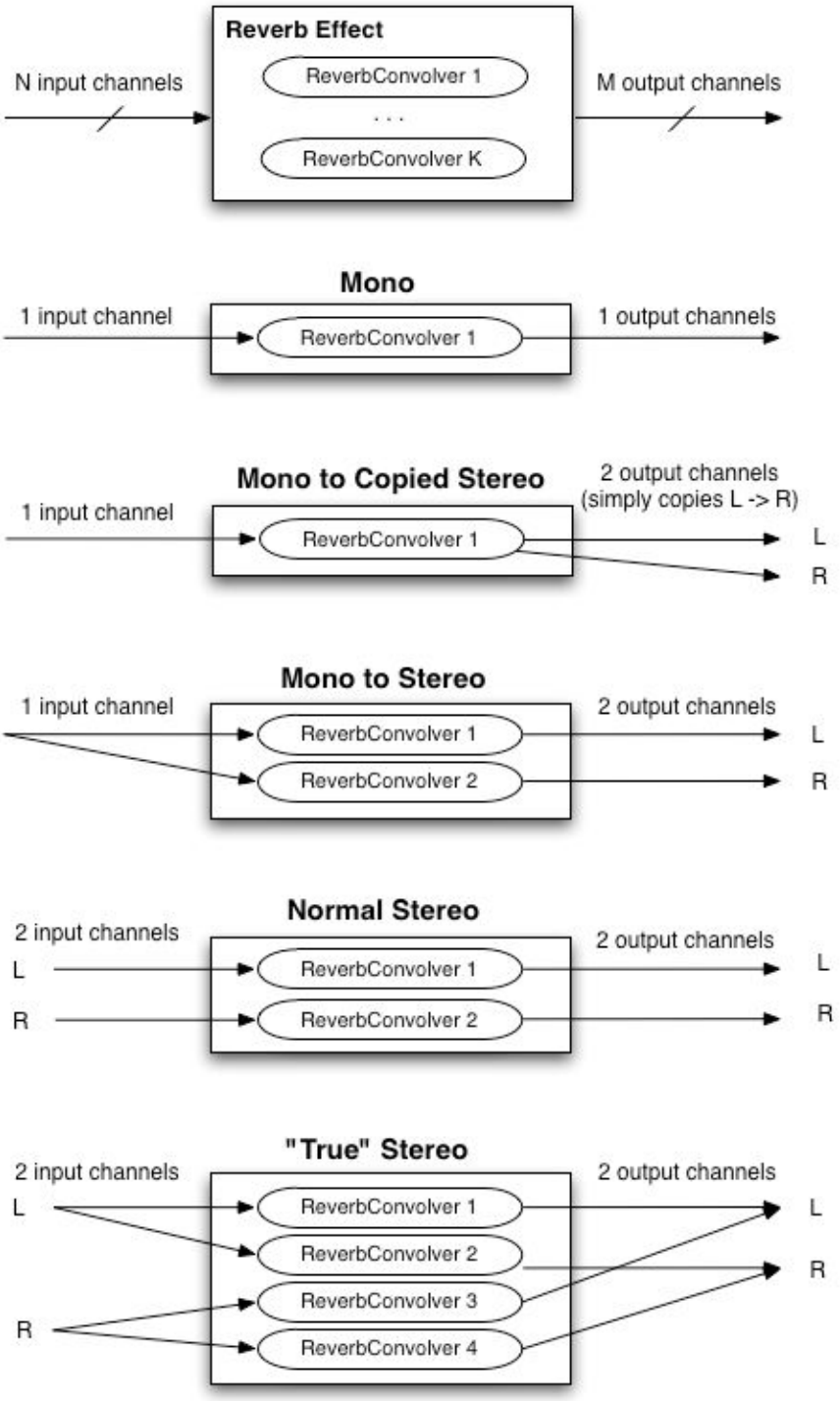
线性卷积可以高效的实现。 notes 这里描述了具体如何实现。

混响效果：（使用矩阵）

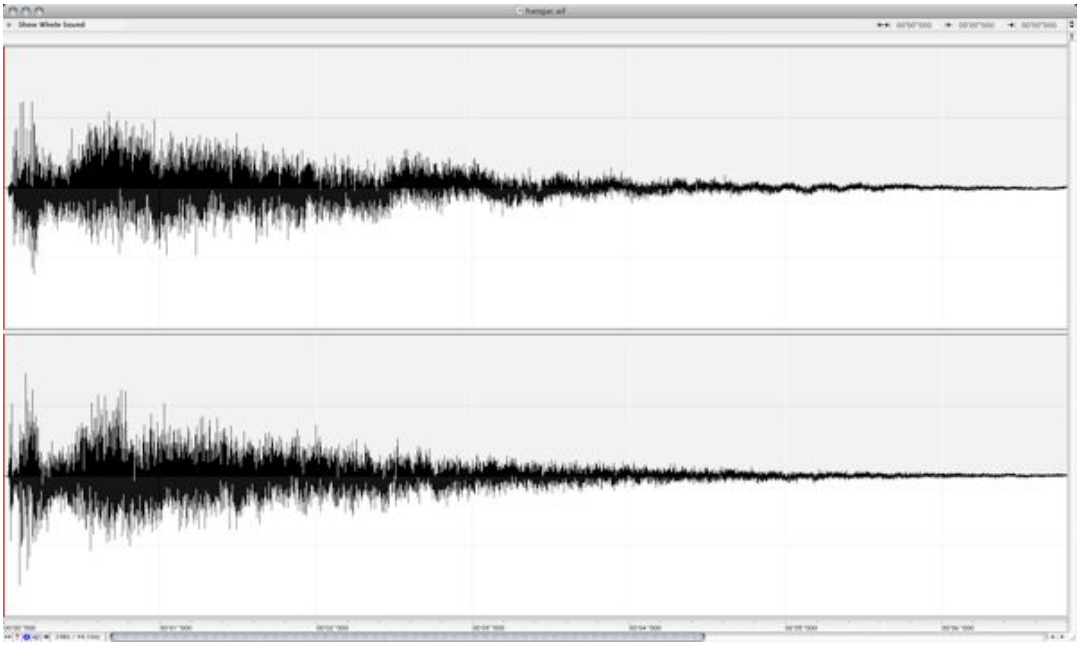
输入为 N 声道的声源，脉冲响应为 K 声道，播放系统为 M 输出声道。因此如何对这些声道进行矩阵相乘来得出最终的记过矩阵是个问题。

n\m\k 的如下子集在实际中必须实现（注意下面第一个图仅仅用来显示总体的情况，后面的几张图才是规范）。不失一般性，开发者想要实现更复杂和任意矩阵，可以使用多个 ConvolverNode 对象和 ChannelMergerNode 相连。

单一声道卷积作用在一个单声道输入中，使用单声道脉冲响应，产生单声道输出。但是为了产生更多空间化的声音，需要考虑使用 2 声道音频输入以及，1,2，4 声道脉冲响应。下面的图显示了立体声播放的常见例子，n、m 为 1,2，或者 4。


Reverb Effect
N input channels
ReverbConvolver 1
. . .
ReverbConvolver K
M output channels
Mono
1 input channel
ReverbConvolver 1
1 output channels
Mono to Copied Stereo
2 output channels
(simply copies L -> R)
1 input channel
ReverbConvolver 1
L
R
Mono to Stereo
1 input channel
ReverbConvolver 1
ReverbConvolver 2
2 output channels
L
R
Normal Stereo
2 input channels
L
R
ReverbConvolver 1
ReverbConvolver 2
2 output channels
L
R
"True" Stereo
2 input channels
L
R
ReverbConvolver 1
ReverbConvolver 2
ReverbConvolver 3
ReverbConvolver 4
2 output channels
L
R

记录脉冲响应



最现代的、精确的方式来记录实际声音空间的脉冲响应的方法是 ,使用长指数正弦扫描。测试声音可以长达 20~30 秒或者更长。

许多测试声音的录音同时通过扬声器播放 ,还可以和许多麦克风相连放置在房间的各个位置。记录扬声器的位置和方向、麦克风的类型、它们的设置、相对于录音机的方向很重要。

在录音之后需要进行后置处理，在测试声音运用反向卷积，针对响应的麦克风位置产生脉冲响应信号.脉冲响应等待加载到卷积混响引擎里来重建室内的声音.

工具:

写了两个指令工具:

generate_testtones 产生指数型的正弦扫描测试音以及它的逆。另个一个工具是后置处理模块。使用这些工具，任何人可以记录设备的他们自己的脉冲响应。实际测试这个工具，首先记录货舱空间里的有趣信息。然后用这些命令行来处理这些信号。

% generate_testtones -h

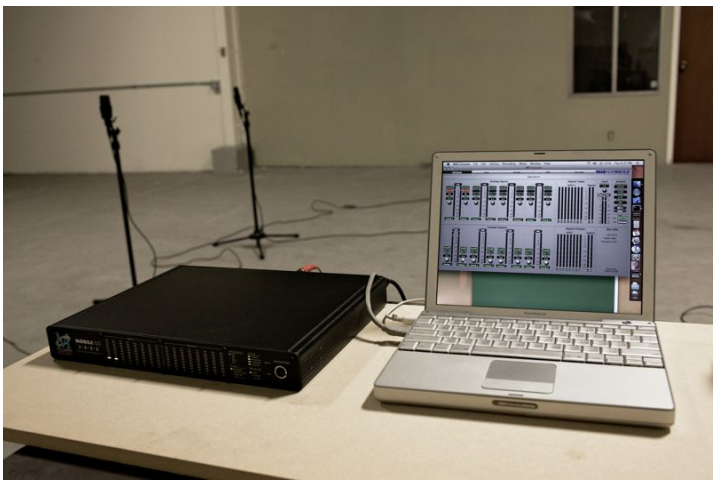
用法：产生测试音频

```
[-o /Path/To/File/To/Create] Two files will be created: .tone
and .inverse
[-rate <sample rate>] sample rate of the generated test tones
[-duration <duration>] The duration, in seconds, of the generated files
[-min_freq <min_freq>] The minimum frequency, in hertz, for the sine
sweep
```

% convolve -h

用法: 对 input_file impulse_response_file output_file 进行卷积

录音设备搭建



Audio Interface: Metric Halo Mobile I/O 2882

Microphones: AKG 414s, Speaker: Mackie HR824

The Warehouse Space

13.Javascript 合成和处理

Mozilla 的项目发起了直接用 Javascript 合成和处理音频信号的实验。一些类型的音频处理 demo 给人们留下了深刻的印象。本文档也叙述了一些方法使用 ScriptProcessorNode 来用 Javascript 来直接处理和合成音频。

定制 dsp 效果：

可以直接使用 js 来进行有趣的定制化音频处理。这也是新算法原型的测试方法。

教育行业的应用：

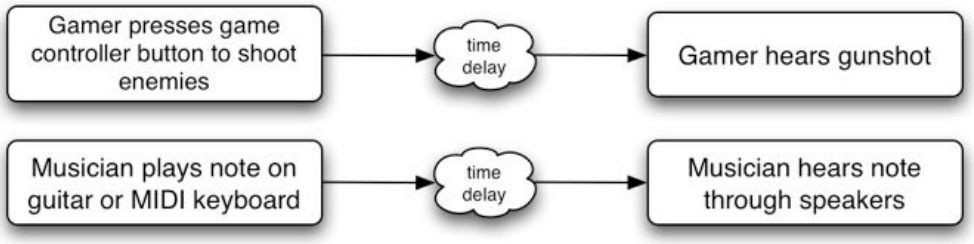
js 处理是理想的展示计算机音频处理和合成的方式，比如将一个方波分解为谐波成分，FM 处理技术。

Javascript 性能：

Javascript 有很多的性能问题，所以并不适用于所有类型的音频处理。此文档提出的方法包含了较高频度计算的音频处理方式（实时计算对于 Javascript 来说太费性能了）。比如多数输入源的 3D 定位以及用 C++代码优化了的卷积效果。不管是直接 Javascript 处理还是 C++优化的代码都由模块化方式集成进 API 里

15.性能考虑

15.1 延迟：是什么以及为什么重要



对于 Web 应用，鼠标、键盘（mousedown、keydown）事件到声音被听到之间的延时值得注意。

时间滞后也叫做延迟，产生的原因有多种（输入设备延迟，内部缓冲延迟，DSP 处理延迟，输入设备延迟，人耳相对于发声体的距离等），并且这些因素会相互叠加。延迟越大用户体验越差。极端情况下，会导致不能正常播音乐或者游戏不能玩。一般情况下会延迟会影响计时 ,并产生音乐滞后或者游戏响应不及时的印象。对于音乐应用时间问题会影响节奏。对于游戏，时间问题将影响游戏的精确操作。对于交互式应用，延迟将和低动画帧率一样给用户交差的体验。对于不同的应用，延迟可能从 3-6ms 到 25-50ms

15.2 音频毛刺

音频毛刺是由于连续音频流中有干扰，导致了声音较大的打击声。可能是多媒体系统的环境出现问题，应该尽可能的避免这样的情况发生。可能因为负责传输音频流到硬件的线程除了问题导致的，比如由于优先级不够或者时间限制导致的计划外延时。它同样可能因为音频 dsp 要实时处理的工作量超出了 cpu 速度。

15.3 硬件可扩展性

系统应该能优雅的降级，来使得在资源有限的情况下进行音频处理不会丢失音频帧。

首先，在任何平台下，音频处理过程不应该完全卡死机器。第二，音频渲染需要产生一个干净的、没有中断、没有毛刺的音频流。

系统应该可以支持很多硬件，从移动端到平板设备到桌面电脑。但是手机对计算资源进行更多的限制是有必要的，因此需要在技术实现的时候降低音频渲染的复杂性。例如，声音降质算法可以降低给定时间内播放的音符总数。

下面列出的技术可以用来限制 CPU 的使用：

15.3.1 CPU 监控

为了防止音频的中断，CPU 使用量比如保持低于 100%。

相对的 CPU 使用情况可以通过动态测量每个 AudioNode（连接的节点链）渲染时间比例来测量。单线程应用中，整体的 CPU 使用比如始终保持在 100%以下。测量方法可以根据渲染来动态调整。也可以用 Javascript 读取 AudioNode 的 cpuUsage。

当测量到 CPU 使用量 100%的时候（或者任意设定的高门槛值），如果再加入更多的 AudioNode 到渲染图片将导致音频质量降低。

15.3.2 声音质量降低

声音降质是限制同样时间内音符播放的数量，来保证 CPU 使用量维持在合理范围内。可以在给定时间内限制音符播放总量，或者当 CPU 超量使用时执行声音降质来降低 CPU 使用量并动态监控，或者同时使用这两种方法。当监控每个音符播放时的 CPU 使用时，可以监控音频路由图中的每一个节点。

当需要“丢弃”一个音符来降低音质时，必须以这样的方式来进行：“丢弃”的音符不会发出能听见的击键声或直接进入到已渲染的音频流里，而是在那个音符在渲染频谱里被抹掉前，迅速地结束淡出这个音符在音频流里的渲染。

当决定好丢弃一个或多个音符时，有一些方法可以帮你挑选在当前播放的音频里哪些音符应该被丢弃。 以下这几点可以帮助你做决定：

- 相对较老的音符： 比起近期的音符，那些被用的最久的音符更应该被丢弃；
- 相对较为安静的音符： 比起那些声音较大的音符，那些对总体混音贡献较小的音符更应该被丢弃；
- 比起那些占用 CPU 用量较小的音符， 那些消耗相对更多 CPU 资源的音符更应该被丢弃；
- 对于音频节点，其包含一个高优先级节点来帮助你判断这个音符的重要性。

15.3.3 简化处理效果

文档里描述的效果都是相对性能消耗不那么大，所以在低性能的移动设备上应该也可以运行。然而，卷积效果可以由各种脉冲响应进行配置，有些对于移动设备会消耗较大。总的来说，CPU 使用量伴随着脉冲响应的长度和声道数的增加而增加。因此，对超过一定长度的脉冲响应应该不做响应是合理的。具体的限制由设备的速度决定。对比于完全拒绝长响应的卷积，更合适的是将响应截断到允许的最大长度或者降低脉冲响应的声道数。

除了卷积效果，使用 HRTF 的平移模型实现 PannerNode 也很消耗性能。对于较慢的设备，可以采用 EQUALPOWER 来节省计算带来的资源开销。

15.3.4 采样率

对于低性能设备，需要考虑以低于常规的采样率来运行渲染。例如，采样率可以由 44.1KHZ 降低到 22.05KHZ。这样降级决定必须在 AudioContext 创建的时候就定下，在运行中再改变采样率会很难实现，而且会在变换的时候引起可听到的毛刺。

15.3.5 预启动

应该可以通过某种形式来激发“pre-flighting”代码（通过 Javascript）来大致决定设备的能力。Javascript 代码可以接着利用这些信息，来降级处理对于通常在更高端设备上运行的密集处理逻辑。同样的，基础的实施方案也可以作为声音降质算法的信息之一。

15.3.6 对于不同 user agents 设备给予不同的权限

Javascript 代码可以使用不同的 useragent ,来降级处理那些在更高性能设备上运行的密集操作。

15.3.7 直接 Javascript 合成、处理的可扩展性

任何音频 DSP 或者 Javascript 直接写的音频处理代码需要考虑可扩展性。考虑可用范围，Javascript 代码需要监控 CPU 使用量，并且在低性能设备上需要能降级处理更密集的处理操作。如果是一种全有或者全无的处理，需要进行 useragent 检查或者 pre-flighting 操作，来避免生成的音频流包含中断的信息。

15.4 使用 Javascript 进行实时音频处理、合成的问题

当用 Javascript 进行音频处理时，既想获得可靠的、无毛刺的音频，同时又想尽可能的低延迟，是很有挑战的事情，尤其当处理器负载很高时。

- Javascript 比高度优化的 C++慢许多，而且不能利用 SSE 优化和多线程，这些都是现在处理器获得高性能的方法。对于处理快速傅立叶变换来说优化了的本地代码比 Javascript 性能高处高出二十倍。因此，Javascript 对于处理诸如针对大数据量的音频进行卷积和 3D 空间定位之类的高负荷处理，会不那么高效。
- setInterval()和 XHR 句柄将占用音频处理的时间。在一个较复杂的游戏中，一些 Javascript 资源将用来执行游戏物理效果和动画效果。因为音频渲染是最后执行的因此会面临大挑战（避免毛刺或者高延迟）
- Javascript 不是运行在一个实时处理线程中，因此会被系统中其他线程抢占线程资源
- 垃圾回收（MAC OS X 上的自动释放的线程池）会造成不可预见的 Javascript 线程延迟

- 多 Javascript 上下文可能正在主线程运行，会占用正在进行音频处理的上下文的时间
- 主线程运行的其他代码（除了 Javascript）比如页面渲染
- 可能系统在 Javascript 线程上加了锁或者分配了内存，这将带来额外的线程占用

对于现在的移动设备来说这些问题可能更困难，这些设备上的处理器性能较低、有电池续航的问题。

16. 应用举例（直接示例，翻译略）

Demo 汇总页：

http://chromium.googlecode.com/svn/trunk/samples/audio/index.html

17 安全考虑（官方无内容）

18 隐私考虑

当有了 AudioNode 的各种信息时，Web Audio API 也暴露了客户端的各种特征（比如音频硬件的采样率）给使用 AudioNode 接口的各种页面。除此之外，时间信息可以通过 RealtimeAnalyzerNode or ScriptProcessorNode 接口采集。基于这些信息可以了解客户端的很多特征。

当前此文档没有明确描述音频输入 ,输入处理将涉及到读取客户端的音频输入或者麦克风输入。这将需要向用户询问是否允许程序如此操作，比如通过 getUSerMedia()接口。

19 最后

1. @音乐前端(weibo.com/musicfe) 基于 WebAudio 探索尝试的相关示例如下：

   http://labs.music.baidu.com/demo/interactions/visualization/

   http://labs.music.baidu.com/demo/audiolab/

2. 感谢@jiexuangao(weibo.com/jiexuangao)同学完成文档初稿翻译，@enimo(weibo.com/enimo)协助初稿校对整理。因文档发布前仅作内部参考使用，必有纰漏之处还请见谅。

3. 本文档仅供大家学习交流，请勿用于商业用途。