## 基于FFmpeg的Web音视频处理系统的设计与实现

## 绪论

**1.1 研究背景与意义**

**1.2 研究现状**

Web音视频处理的研究现状

相关框架技术的研究现状

**1.3 论文的研究目标和内容**

本论文的主要工作为：

1. 研究学习音视频处理技术以及相关理论，分析Web音视频处理系统的需求，设计Web音视频处理系统的架构和满足需求的技术实现方案。
2. 研究学习FFmpeg音视频处理库以及WebAssembly二进制字节码编译技术。掌握FFmpeg处理音视频的核心流程和原理，并利用WebAssembly技术编译FFmpeg以实现Web浏览器中强兼容、高性能的音视频处理能力。
3. 设计实现基于FFmpeg的Web多线程音视频编解码模块，进一步设计实现根据数据是否实时性消费划分的弱实时性音视频剪辑处理模块和强实时性音视频播放处理模块。针对弱实时性处理模块实现基于DAG拓扑图的多剪辑任务处理。针对强实时性处理模块实现基于时间戳同步、多级缓存、交替存储数组结构的音视频同步播放处理。
4. 针对本文设计实现的Web音视频处理系统进行功能性测试和非功能性测试。功能性测试主要包括测试音视频编解码功能、弱实时性音视频剪辑处理功能以及强实时性音视频播放处理功能。非功能性测试主要包括系统的性能、代码质量等。

**1.4 论文组织结构**

## 二、相关技术

**2.1音视频技术**

**2.2 FFmpeg**

**2.3 WebAssembly**

**2.4 本章小结**

## 系统需求分析与总体设计

本章内容通过分析系统的功能性需求和非功能性需求进而对系统整体架构和功能进行设计。借助用例图和模块数据流图对系统功能性需求进行分析，同时也对非功能性需求进行说明。最后通过系统模块架构图、整体数据流图、功能分解图清晰地描述系统中各个模块之间的关系。

**3.1 基于FFmpeg的Web音视频处理系统需求分析**

基于FFmpeg的 Web音视频处理系统的最终目标是实现在Web浏览器上下文环境中对本地音视频文件的高性能、强兼容的处理功能，包括基本的音视频编解码处理，以及在此基础上根据解码后数据实时性消费划分的弱实时性剪辑处理和强实时性播放处理。所以本文提出的Web音视频处理系统需要支持在Web浏览器上下文环境中嵌入并调用扩展的FFmpeg音视频处理模块，并通过结合多线程、WebAssembly编译技术的架构来增强原生浏览器对多种音视频格式的兼容识别和解析处理。

因此Web音视频处理系统的功能性需求包括三部分：Web音视频编解码处理功能性需求、弱实时性音视频剪辑处理功能性需求以及强实时性音视频播放处理功能性需求。非功能性需求主要包括安全性、易用性、性能、可靠性以及可维护性五部分。

**3.1.1 系统功能性需求**

1. **Web音视频编解码需求**

Web音视频编解码功能是本系统中最基础、也是架构设计最核心的部分，因为弱实时性剪辑处理模块和强实时性播放处理模块都是基于此功能进行扩展实现的。而Web音视频编解码处理模块的核心思想是在Web浏览器中实现以FFmpeg自身优秀的音视频处理流程和规范为基础的Web编解码功能。但因为Web浏览器执行引擎不能直接运行由C语言组成的FFmpeg音视频处理程序，所以通过引入WebAssembly二进制字节码编译技术来定制化裁剪编译扩展的FFmpeg音视频处理程序为WASM音视频处理模块，再进一步利用Web Worker子线程加载编译模块调用，从而实现在Web浏览器中对本地音视频文件的解封装、解码、编码、封装的基本处理功能。

WebAssembly二进制字节码程序相比较C编译型语言程序可以被Web浏览器执行引擎所执行，同时相比较高级抽象的JavaScript解释型语言程序的执行速度要更快，所以利用WebAssembly编译技术不但解决了FFmpeg扩展音视频处理程序的嵌入执行难题，而且还可以提高音视频处理在Web浏览器中的执行时间。

Web音视频编解码模块通过两个Web Worker子线程来分别读取本地音视频文件和加载初始化编译后的WASM音视频处理模块。在文件Worker子线程中可以利用FileReader对象异步读取存储在用户计算机上的多媒体文件，并以Web浏览器JavaScript主线程作为中间缓冲区将读取到的文件数据转发给处理Worker子线程。处理Worker子线程在加载WASM音视频处理模块时通过初始化已经保存了WASM模块的实例引用，通过此引用可以调用WASM模块暴露的音视频处理接口来进一步实现对音视频数据的解封装、解码、编码、封装四个重要功能。因此Web编解码模块的具体数据流图如图3-1所示：

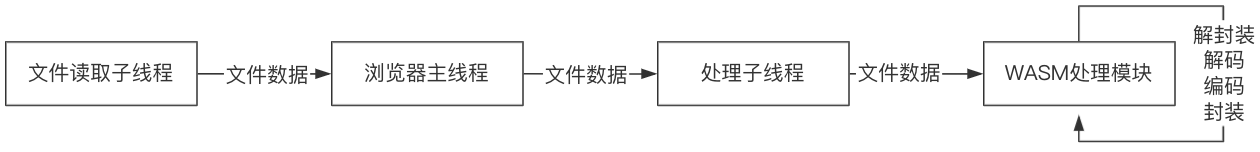


图3-1 Web编解码模块功能的数据流图

因为模块是通过多个Worker子线程来处理异步耗时的文件读取、音视频处理任务，所以还需要设计实现主线程与多个子线程之间的指令数据通讯机制协调实现UI交互操作得到正确响应。此外在Web音视频编解码功能基础之上，根据音视频处理在Web浏览器JavaScript主线程运行侧和WASM音视频处理模块运行侧的数据转移实时性可以划分为弱实时性音视频剪辑处理功能和强实时性音视频播放处理功能。

根据Web音视频编解码处理功能的需求，得到其需求用例图，如图3-2所示：

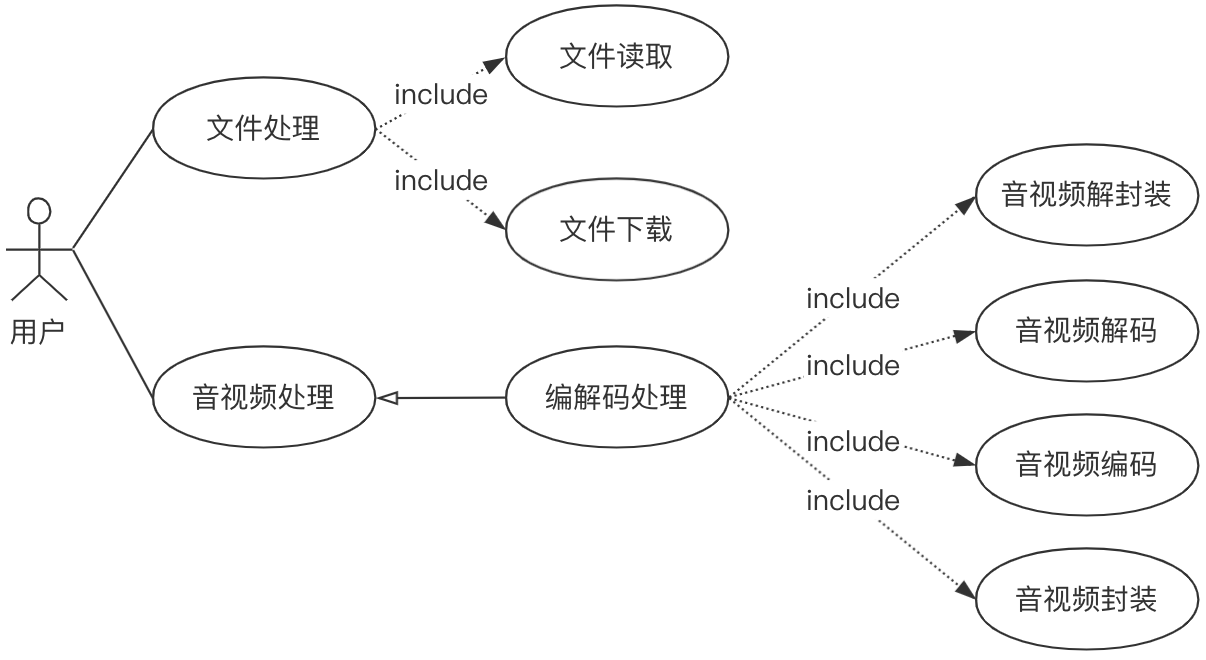


图3-2 Web音视频编解码处理功能需求用例图

用例名称:Web音视频编解码处理。

说明:在Web浏览器上下文环境中，实现文件读取子线程将音视频文件数据流式地传递到音视频处理子线程中FFmpeg定制化编译后的WASM处理模块中，实现高效兼容地解析多种音视频格式，以及包括音视频解封装、音视频解码、音视频编码、音视频封装四个重要基础处理功能。

参与者:Web音视频编解码模块，文件系统模块

前置条件:本地音视频文件读取成功。

后置条件:本地音视频文件可以被正确识别解析、解封装、解码、编码、封装。

**2、弱实时性剪辑处理的需求**

基于上一节Web浏览器上下文环境中对本地音视频文件的读取、解封装、编解码功能的基础上，实现弱实时性的音视频剪辑处理功能。其中弱实时性主要体现在Web浏览器JavaScript运行侧与WASM音视频处理模块运行侧的数据实时性交互不强。整体上音视频数据流向主要是单向流动，由文件子线程读取音视频文件数据通过主线程发送到处理子线程中，再由处理子线程中JavaScript运行侧利用WASM音视频处理模块的实例化引用分配内存空间进行处理。直到WASM音视频处理模块处理完用户定制的音视频剪辑任务后，才会把剪辑后的音视频数据重新编码、封装后再一步步从WASM音视频处理模块的内存空间转移到处理子线程中的JavaScript运行侧，再传递到主线程中进行下载查看。即就是WASM音视频处理模块在处理音视频数据的同时，解码后的音视频帧数据并不会实时地反向传递到Web浏览器主线程中，简化了相应的音视频处理复杂度。因此弱实时性剪辑处理功能的数据流图如图3-3所示：

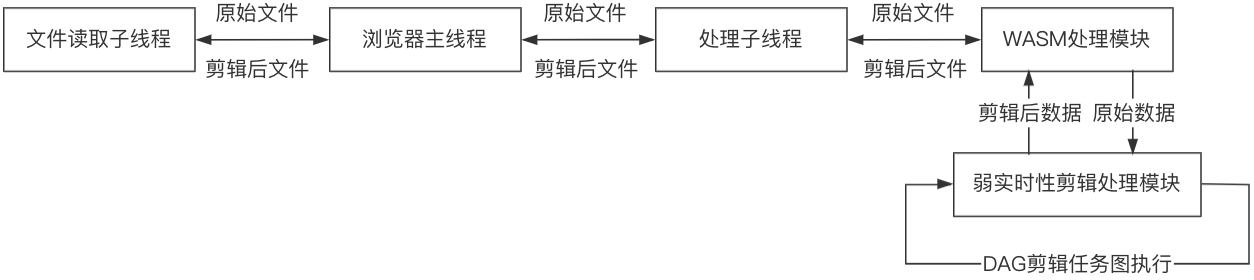


图3-3 弱实时性剪辑处理功能数据流图

音视频剪辑处理通过将剪辑功能操作抽象为五个基本的原子处理单元，分别是剪切处理单元、抽取处理单元、合并处理单元、压缩处理单元和转换处理单元。以及两个基础的辅助处理单元分别是输入处理单元和输出处理单元。输入处理单元是根据文件子线程读取本地文件数量一一对应生成的确保输入的完全可控性；输出处理单元用于标识一个剪辑处理任务的结束标志。之后在画布上成功注册上述的七种处理单元组件后，通过拖拽的组件的方式在画布上灵活组合多个处理单元构建DAG剪辑拓扑图达到最终的剪辑处理需求。最后根据构建好的剪辑拓扑图，执行处理模块再经过以下三个步骤来执行多个剪辑任务：

1、以输入处理单元为无入度根节点，检测多剪辑任务图是否为多个正确的有向无环图。

2、初始化系统任务执行器，并通过拓扑排序算法并行执行多个剪辑任务的各个处理单元。

3、以输出处理单元为无出度结束节点，标志当前音视频剪辑任务的结束，并提供下载查看剪辑后音视频的功能。

根据弱实时性音视频剪辑处理功能的需求，得到其需求用例图，如图3-4所示：

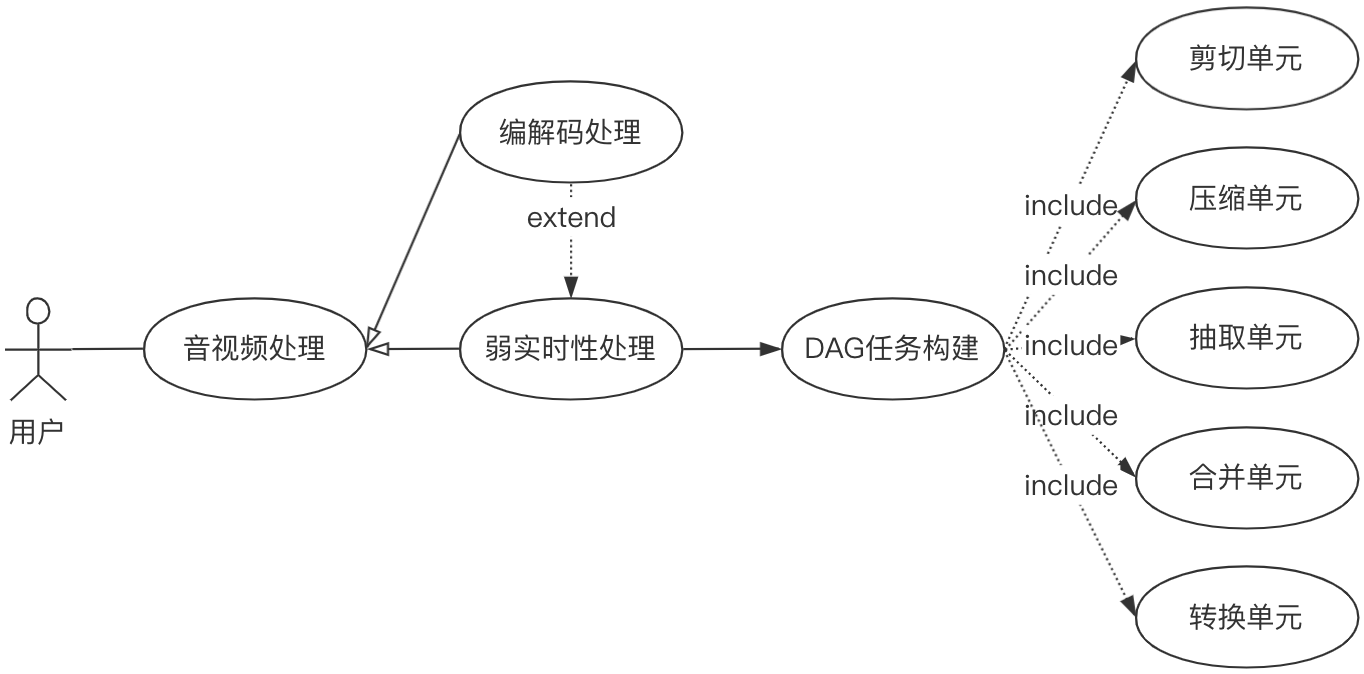


图3-4 弱实时性剪辑处理模块需求用例图

用例名称:弱实时性剪辑处理。

说明:基于Web音视频编解码模块，通过组合抽象的剪辑处理单元和基本输入输出处理单元来构建多DAG剪辑任务图，实现多音视频剪辑处理功能。剪辑处理单元抽象为剪切处理单元、压缩处理单元、抽取处理单元、合并处理单元以及转换处理单元。

参与者:Web音视频编解码模块、弱实时性剪辑处理模块。

前置条件:本地音视频文件成功读取、解析并成功解码。

后置条件:剪辑处理后的音视频文件可以下载查看，并符合预期剪辑效果。

1. **强实时性播放处理的需求**

相比较弱实时性剪辑处理，强实时性播放处理模块同样是基于Web音视频编解码功能进行扩展，不同之处在于播放处理模块要求WASM音视频处理模块运行侧将解码后的每一帧音视频数据实时地通过线程之间的数据通讯机制传递到Web浏览器主线程运行侧，在浏览器主线程中将解码后的视频YUV图像数据和音频采样数据分别通过WebGL和AudioContext进行渲染和播放。同时播放处理模块设计以时间戳同步机制为主的多级缓存策略和回退补偿策略来协调解决音频帧和视频帧的实时同步问题。最终强实时性播放处理模块的数据流图如图3-5所示：

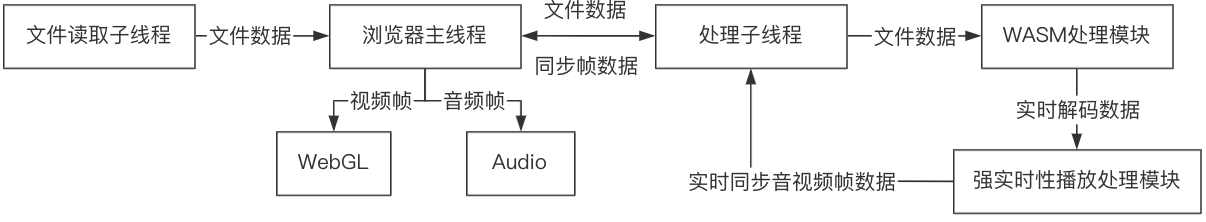


图3-5 强实时性播放处理功能数据流图

强实时性播放处理模块的功能主要分为简单顺序播放、高级播放，高级播放包括倍速播放、全屏播放以及Seek播放。在播放处理过程中，需要考虑协调音频帧和视频帧的实时同步问题，确保在任何播放状态下音频帧和视频帧都会同步被消费，也就是通常所说的音画同步。在播放处理模块中设计音频帧和视频帧的交替存储数据结构，以及设计实现多级缓存机制来确保音频帧和视频帧的同步。同时针对出现音画不同步的问题，强实时性播放处理模块需要一定的回退补偿和丢弃补偿策略来修正音画不同步的情况。

根据强实时性音视频播放处理功能模块的需求，得到其需求用例图，如图3-6所示：

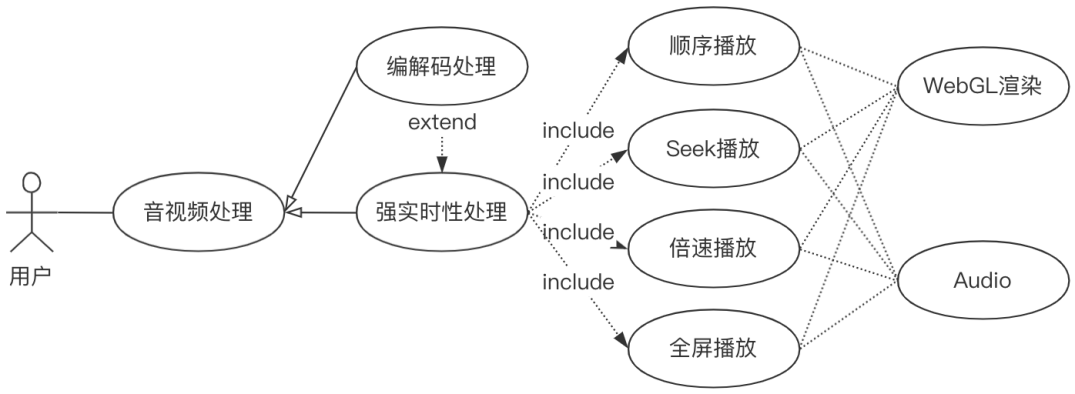


图3-6 强实时性处理模块需求用例图

用例名称:强实时性播放处理。

说明:基于Web音视频编解码模块的，通过将Web编解码模块处理后的解码音视频数据从WASM运行侧转移给JavaScript运行侧，然后通过WebGL来进行yuv图像的绘制、Audio播放音频数据。强实时性播放处理模块包含顺序播放、倍速播放、全屏播放以及Seek播放。

参与者:Web音视频编解码模块、强实时性播放处理模块。

前置条件:本地音视频文件成功读取、解析并成功解码。

后置条件:解码后的音视频数据可以实时地从WASM处理模块子线程转移至Web浏览器JavaScript主线程中，并成功实时同步对视频帧YUV图像数据渲染和音频帧数据播放。

**3.1.2 系统非功能性需求**

Web音视频处理系统的非功能需求主要体现除了功能性需求以外的其他特性，包括系统的安全性、易用性、性能、可靠性以及可维护性需求，具体说明如下：

1. 安全性

系统的安全性主要体现在两个方面，分别是编码安全和用户内容安全两部分。编码安全指的是系统自身的代码需要经过严格的测试，尽量对编码过程中容易出现的内存泄漏、数据为空进行校验与捕获。用户内容安全是指系统在处理用户本地音视频数据时，保证可见性原则和不侵入原则，安全有效的处理用户指定的音视频文件。通过以上两个方面保障系统自身的安全性。

1. 易用性

Web音视频处理系统借助了B/S架构的易用优势，无需下载其他软件实现音视频在线处理功能。系统面对不同用户群体，尤其是针对不具备音视频相关专业知识基础的用户在使用过程中，提供对应的默认参数以及相关参数的友好提示来帮助用户更好的使用平台功能。同时要做到系统页面逻辑清晰、功能操作简单方便等。

1. 性能

性能需求主要指在Web浏览器中提高音视频处理性能。因为Web浏览器的音视频处理过程是非常消耗系统性能，因为Web浏览器并不适合处理CPU密集型计算任务，并且Web浏览器作为应用软件对操作系统做了封装，导致不能直接利用操作系统的资源。如果可以通过提高音视频处理在Web中的性能，可以缓解服务端压力，更好的利用客户端的计算资源。

1. 可靠性

系统可靠性需求指的是使用该Web音视频处理系统处理音视频时，需要严格按照使用者定义的规则进行处理。同时系统自身的稳定性需要控制在极小的波动范围内，不会对使用者的浏览器应用产生过大的内存负担而导致页面崩溃等问题。

5. 可维护性

系统可维护性需求主要指系统在后期的更新维护过程中，能够根据新的业务场景对原有系统进行扩展。同时系统本身运行过程中，通过埋点收集关键的日志信息，方便根据日志信息排查系统出现的问题并解决。同时在系统功能的设计实现上，尽量运用六大软件设计原则，做到系统中各个模块高内聚、低耦合的关系，提高系统的可维护性。

**3.2 系统总体设计**

根据上一节系统的功能性需求和非功能性需求为基础，以需求分析为设计目标，本节主要针对本文提出的高性能、强兼容性的Web音视频处理系统进行总体设计，具体包括系统架构设计和系统功能设计。

**3.2.1 系统架构设计**

因为Web浏览器中JavaScript执行引擎天然的不适合执行音视频编解码处理这类CPU密集型计算任务，以及无法直接在Web浏览器中运行C语言程序的局限性，所以本系统架构设计上需要考虑怎么样结合WebAssembly二进制字节码编译技术，将基于FFmpeg扩展的音视频处理程序编译后的WASM音视频处理模块通过Web Worker多线程加载到Web浏览器中进行调用来解决低性能、局限性的问题。

基于FFmpeg和WebAssembly的Web音视频处理系统的整体架构主要分为四层：音视频渲染交互层、音视频数据控制层、Web Worker线程加载层以及音视频处理层。接下来分别介绍这四层架构之间的关系以及各层的主要作用：

1、音视频处理层是系统核心功能层，也是Web音视频处理的兼容性和性能保证所在。在音视频处理层需要交叉链接编译FFmpeg裁剪模块和扩展程序为WASM音视频处理模块，并设计WASM处理程序与JavaScipt调用程序之间的交互接口，同时直接用WASM模块实例去分配音视频数据的存储内存来防止不同运行引擎之间非必要的数据拷贝以提高处理性能。

2、Web Worker线程加载层主要通过子线程加载本地音视频文件和WASM音视频处理模块，让本地音视频文件的读取和CPU密集型的音视频计算处理两种耗时操作在子线程中执行而不会阻塞主线程的运行来提高系统的整体性能。最后设计实现主线程和子线程之间的消息通讯机制来实现各个线程之间的交互调用逻辑来协作完成音视频处理任务，通过这种方式可以解耦模块和模块之间的关系，提高了系统的可维护性和稳定性；

3、音视频数据控制层主要作用是通过连接渲染交互层和Worker线程加载层，将渲染交互层的操作逻辑指令解释为Worker线程加载层提供的消息指令组合，再调用下层的核心处理功能。同时也将Worker线程层传递的解码后的音视频数据进行同步控制后再由渲染交互层进行消费来确保音视频帧的实时同步。

4、渲染交互层是使用者直接操控和查看的交互页面的统称，主要作用是抽象音视频处理任务的定制化构建过程以及实时音视频帧数据的渲染播放。

因此Web音视频处理系统的整体架构设计如图3-7所示：

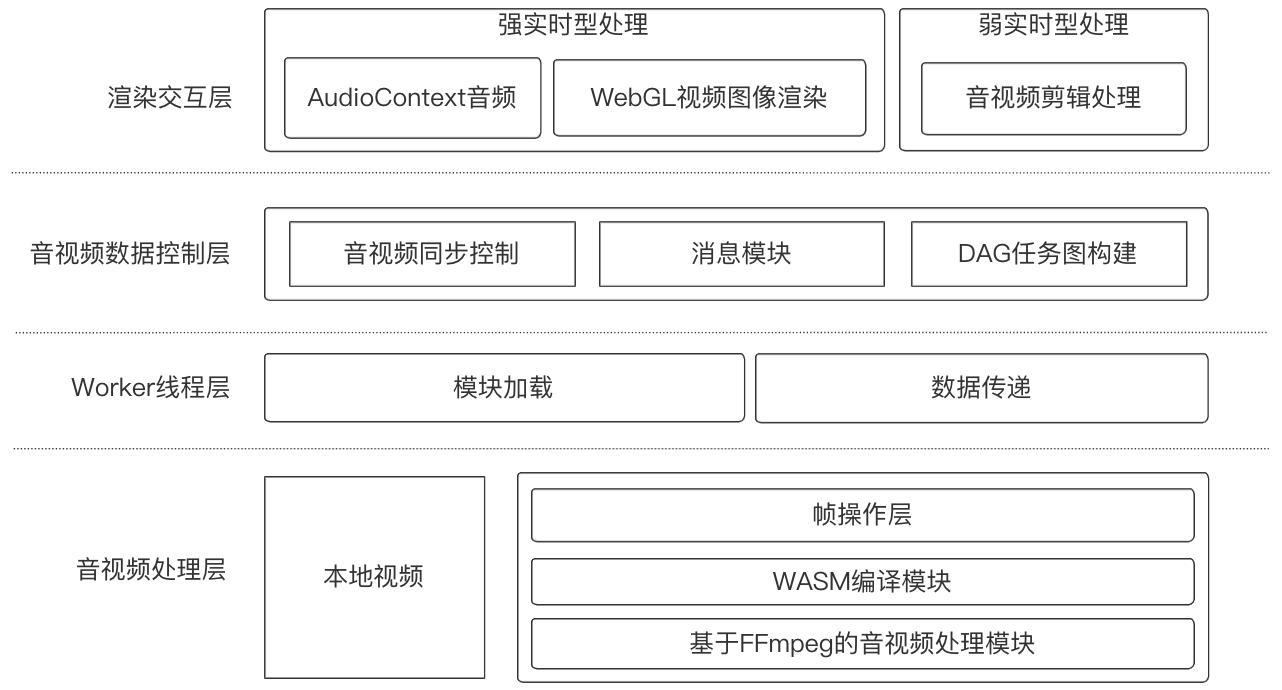


图3-7 音视频处理系统总体架构图

在Web音视频处理系统中，弱实时性音视频剪辑处理模块和强实时性音视频播放处理模块的前端部分都基于Web音视频编解码模块。根据不同的音视频处理需求，Web音视频编解码模块将解码后数据发送给具体的模块来进一步处理。如图3-8展示了Web音视频处理系统针对不同功能需求的具体工作流程。

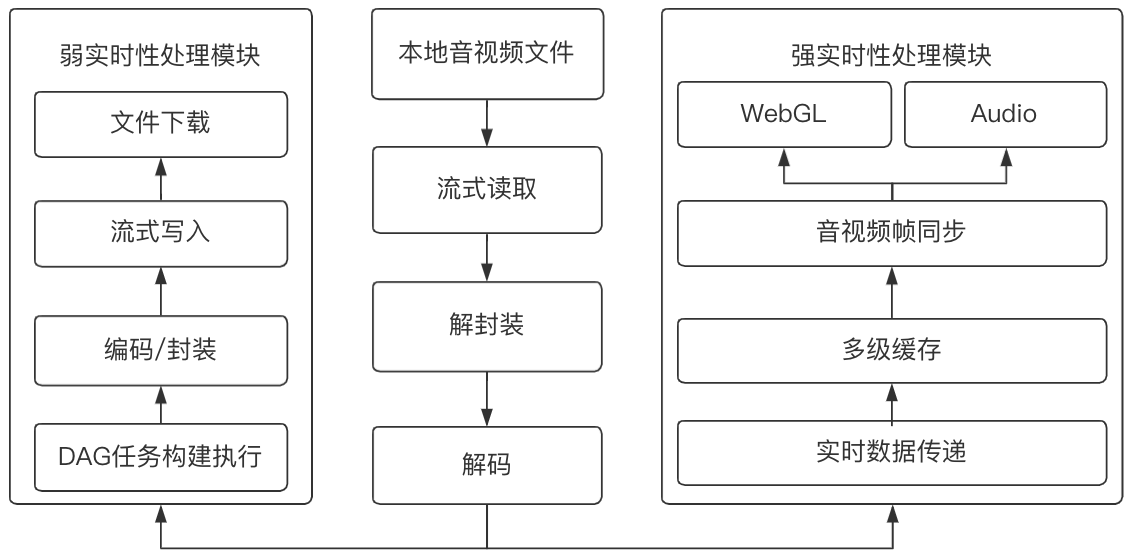


图3-8 系统整体数据流图

首先Web音视频编解码模块第一步是读取本地音视频文件数据，再根据音视频文件的头信息加载对应的解封装器、解码器来保证正确地解码不同格式的音视频文件，最后将解码后的数据发送给弱实时性处理模块或者强实时性处理模块来完成剪辑功能和播放功能。弱实时性处理模块根据DAG剪辑任务图自定义的各个节点处理单元来拓扑执行解码后的音视频数据，将处理完成的音视频数据通过参数指定的编码器和封装器来生成新的音视频文件，最后流式写入本地文件实现文件下载。强实时性处理模块则需要实时地获取到解码后的音频帧和视频帧数据，通过缓存模块和同步控制模块的处理，进一步把帧数据发送给Web浏览器主线程的WebGL模块和Audio模块来消费。

**3.2.2 系统功能设计**

根据Web音视频处理系统各相关功能之间的关系，将系统整体功能划分为文件模块和音视频处理模块，音视频处理模块又分为Web编解码模块、弱实时性剪辑处理模块和强实时性播放处理模块三个模块。最终系统整体功能分解如图3-6所示：

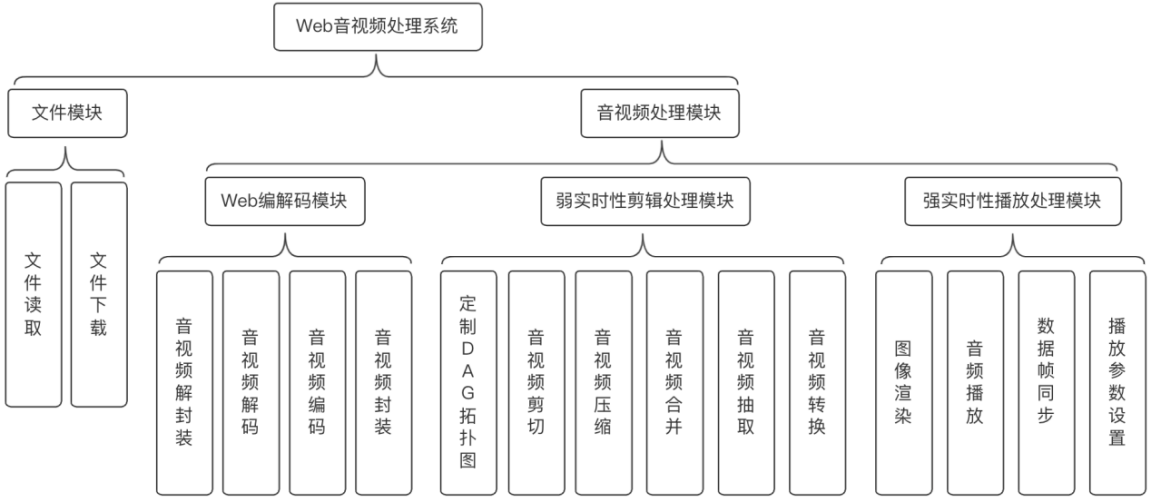


图3-6 音视频处理系统功能分解图

文件模块的主要作用是根据音视频处理模块的指令来控制流式读取文件的开始与停止，并提供处理完成后生成的新音视频文件的下载功能。音视频处理模块中Web编解码模块是核心基础模块，主要负责音视频文件的解封装、解码、编码、封装功能。弱实时性剪辑处理模块的主要功能是通过组合七种音视频剪辑处理单元，包含输入、输出、剪切、压缩、合并、抽取与转换，构建并拓扑执行DAG剪辑处理拓扑图来实现定制化地音视频剪辑功能。最后强实时性播放处理模块是基于WebGL渲染引擎、Audio播放引擎以及数据帧同步功能的基础上实现音视频的播放功能，再通过播放参数设置可以把播放细分为简单顺序播放、倍速播放以及Seek播放，同时也提供播放暂停、开始等基础的播放设置。

**3.4 本章小结**

本章第一部分内容根据Web音视频处理系统的最终目标逐次对三个功能性需求Web音视频编解码、弱实时性剪辑处理、强实时性播放处理进行需求分析，并对系统非功能性提出了五点要求，包括系统的安全性、易用性、性能、可靠性以及可维护性。其次本章第二部分内容是以需求分析为设计目标进行系统总体设计，包括系统架构设计和系统功能设计，清晰地描述了系统各个模块之间的协作关系以及各个功能的划分情况。

## 系统详细设计与实现

本章内容主要是对Web音视频处理系统中三个主要功能模块的设计与实现进行详细介绍，其中包括多线程加载编译的WASM音视频处理模块的设计与实现、基于DAG构建多音视频剪辑任务拓扑图的设计与实现以及基于时间戳同步策略的音视频播放的设计与实现。

## 4.1 Web编解码模块的设计与实现

本小节通过介绍FFmpeg编译移植、多线程加载调用WASM模块的方案以及编解码处理时序实现三方面来详细描述Web编解码模块的设计与实现。

**4.1.1 FFmpeg编译移植**

根据第三章提到的Web浏览器上下文环境不能直接调用执行FFmpeg提供的功能的问题，所以需要利用WebAssembly二进制字节码编译技术来定制化裁剪编译基于FFmpeg扩展的音视频处理程序成WASM音视频处理模块来解决Web浏览器直接调用的局限性。本文采用的编译方案分为两个阶段，第一个阶段是裁剪FFmpeg功能模块后编译为二进制静态链接库文件；第二个阶段即将裁剪编译后的静态链接库文件和系统扩展的音视频处理功能模块链接后再编译为WASM模块。

在MacOS操作系统下实现FFmpeg的定制化编译为WASM模块需要借助相关的编译工具链，FFmpeg的编译依赖于pkg-config，而它又依赖于GLib，而GLib又依赖于gettext，所以必须先安装并编译相关依赖，确保编译过程对相关依赖的可达性。最终具体编译步骤如下：

1. 下载FFmpeg源码、编译安装pkg-config、glib、gettext等依赖包以及fdk-aac、x264、x265等音视频编解码器，确保在编译过程是可以找到相应的依赖包和编解码器进行链接。
2. 下载安装emscripten、emmake，用于替换gcc编译器来编译C语言程序。
3. 裁剪优化编译FFmpeg模块，切换到本地FFmpeg源码目录下，执行裁剪优化配置的编译命令，生成相关头文件和静态链接库文件。
4. 将第三步裁剪编译后静态链接库文件和自定义扩展的音视频处理程序链接后，通过emcc命令编译为可执行的WASM音视频处理模块和自动化加载程序。

接下来详细介绍下第三步采取的裁剪编译为静态链接库文件和第四步链接编译为WASM模块的具体细节。

**1、裁剪编译为静态链接库文件**

因为FFmpeg源代码由多个基础核心模块和上层工具模块共同组成，其中一些模块对于本文Web音视频处理系统的构建是非必需的，所以针对FFmpeg源码模块采取定制化裁剪，只选取核心模块libavformat、libavcodec、libavutil等，并开启多种音视频复用器、解复用器以及编解码器来提高对音视频格式的兼容性。通过这种方式不但可以减少编译后的产物大小，而且还提高了编译时速度、编译后模块的加载速度以及系统整体程序的大小。

在裁剪优化FFmpeg时，通过不同的参数选项组合来实现具体的定制化编译方案。FFmpeg提供的参数选项一般分为以下三个方面：基本参数包含编译输入输出目录、静态库目录动态库目录等设置；开闭参数主要作用是开启或者关闭相关模块、编解码器、复用器与解复用器的支持；高级参数用来指定编译工具、编译宿主系统的CPU、架构信息以及一些编译时内存大小分配等等参数。部分编译时配置的优化参数如表4-1所示：

表4-1 FFmpeg编译配置参数说明

|  |  |
| --- | --- |
| 配置参数 | 参数作用 |
| --disable-ffplay  --disable-ffprobe  --disable-ffmpeg  --disable-avdevice  --disable-doc  --disable-ffserver  --disable-small  --disable-programs  --disable-network  --enable-cross-compile  --enable-gpl  --enable-libx265  –enable-encoder=NAME  –enable-decoder=NAME  –enable-muxer=NAME  –enable-demuxer=NAME | 禁止编译ffplay播放器模块  禁止编译ffprobe嗅探器模块  禁止编译ffmpeg命令工具模块  禁止编译avdevice设备模块  禁止生成doc文件  禁止编译ffserver流媒体服务模块  优化减少FFmpeg的体积  禁止生成可执行文件  禁止编译网络模块  启动交叉编译  允许使用GPL代码  启用x265作为h.265的编解码器  启用NAME编码器  启用NAME解码器  启用NAME复用器  启用NAME解复用器 |

在根据不同配置参数功能的分析后，最后确定如下的编译配置参数。

emconfigure ./configure

# 基本参数

--prefix=$(pwd)/dist

# 高级参数

--cc="emcc" --cxx="em++" --ar="emar" --ranlib="emranlib" -cpu=generic --enable-cross-compile --target-os=none --arch=x86\_32

# 开闭参数

--enable-gpl --enable-version3 --enable-protocol=file

--disable-avdevice --disable-programs --disable-logging --disable-ffplay --disable-ffprobe --disable-ffserver --disable-asm --disable-doc --disable-devices --disable-network --disable-hwaccels --disable-indevs --disable-outdevs --disable-parsers --disable-bsfs --disable-debug

--disable-everything --enable-decoder=hevc --enable-decoder=h264 --enable-decoder=aac --enable-demuxer=mov --enable-demuxer=flv --enable-libx264 --enable-libx265 ...

从以上配置参数可以看到通过emconfigure工具来执行编译优化的命令，其中基本参数用于确定输出目录的前缀路径。高级参数指定C程序的编译器emcc，最低cpu要求、归档静态库工具emar、微处理器体系结构x86-32以及开启交叉编译等等。开启交叉编译的目的是为了在MacOS操作系统上一次编译出能运行在其他不同的操作系统之上的程序。最后是开闭参数，主要作用是禁止构建系统中不需要的功能模块。最后开启了多种音视频的复用器、解复用器以及编解码器来提高Web音视频处理的兼容性，同时针对FFmpeg源码默认不支持的编解码器，例如FFmpeg默认不支持H.264、H.265的编码，所以通过--enable-libx264、--enable-libx265来静态链接额外依赖的库文件。最后通过执行以上相关优化的配置参数命令编译为二进制静态链接库文件实现对FFmpeg功能模块库以及其他依赖库的裁剪编译工作，

**2、编译为WASM模块**

在上一步裁剪编译产生的FFmpeg二进制链接库文件的基础上，链接系统实现的音视频扩展功能程序，进一步通过emcc编译工具编译成WASM音视频处理功能模块。在Web浏览器环境加载使用WASM音视频处理模块时，需要基于Web浏览器JavaScript运行侧和WASM音视频处理模块运行侧之间的相互调用的功能进一步实现弱实时性剪辑处理和强实时性播放处理功能模块，所以需要在编译WASM模块的时候增加双向的接口暴露来实现双向调用。

如表4-2所示，通过EXPORTED\_FUNCTIONS关键字指定WASM模块可以暴露的自定义函数接口名列表，在Web浏览器JavaScript运行侧可以通过WASM模块的引用直接使用函数名进行调用。通过EXTRA\_EXPORTED\_RUNTIME\_METHODS关键字指定的addFunction函数接口，可以在运行时让WASM模块获取JavaScript运行侧的具体功能函数的引用并调用执行。

表4-2 JavaScript和WASM相互调用实现方式

|  |  |
| --- | --- |
| 功能 | 编译方法 |
| JavaScript调用WASM  WASM调用JavaScript | EXPORTED\_FUNCTIONS=['methodName']  EXTRA\_EXPORTED\_RUNTIME\_METHODS="['addFunction']" |

最后编译WASM模块的编译配置如下所示：

TOTAL\_MEMORY=67108864

EXPORTED\_FUNCTIONS="['\_initDecoder','\_uninitDecoder','\_openDecoder', '\_closeDecoder', '\_sendData', '\_decodeOnePacket', '\_seekTo', '\_main', '\_malloc', '\_free','\_cut', '\_merge', '\_compress', '\_extra','\_transfrom']"

emcc decoder.c dist/lib/libavformat.a dist/lib/libavcodec.a dist/lib/libavutil.a dist/lib/libswscale.a \

-O3 \

-I "dist/include" \

-s WASM=1 \

-s TOTAL\_MEMORY=${TOTAL\_MEMORY} \

-s EXPORTED\_FUNCTIONS="${EXPORTED\_FUNCTIONS}" \

-s EXTRA\_EXPORTED\_RUNTIME\_METHODS="['addFunction']" \

-s RESERVED\_FUNCTION\_POINTERS=14 \

-s FORCE\_FILESYSTEM=1 \

-o libffmpeg.js

-O3 标识最高优化级别减少代码体积，-O表示编译输出，TOTAL\_MEMORY参数在编译时设定内存容量，EXPORTED\_FUNCTIONS用于指定WASM模块暴露函数接口在JavaScript运行侧调用，EXTRA\_EXPORTED\_RUNTIME\_METHODS参数提供一个addFunction方法用于接受一个JavaScript函数，RESERVED\_FUNCTION\_POINTERS参数代表函数指针的预留空间，FORCE\_FILESYSTEM=1参数表示强制启用文件系统。通过以上参数配置就可以编译出与Web浏览器上下文环境双向调用的WASM音视频处理模块。

**4.1.2 多线程加载调用WASM处理模块设计与实现**

在Web浏览器中加载初始化WASM音视频编解码模块、读取本地音视频文件以及编解码音视频文件都是比较耗时的操作，而Web浏览器JavaScript引擎是单线程事件驱动执行的，如果将所有耗时计算都由JavaScript主线程执行就会阻塞其他轻量计算任务的执行。因此设计实现通过Web浏览器提供的Web Worker创建多线程环境来加载调用WASM音视频处理模块，Web Worker允许主线程创建Worker子线程，将一些耗时任务分配给Worker子线程去运行，就不会阻塞主线程、UI渲染线程的执行。同时只需要定义好主线程与多个子线程之间的消息、数据通讯机制，通过事件监听来控制程序执行流程。

完整的编解码模块线程之间的协作关系如图4-1所示，以主线程为中间层，控制UI交互指令消息发送给子线程并协调文件读取子线程和WASM音视频处理子线程的工作。主线程作为主控模块分别与文件读取子线程和音视频处理子线程通过消息、数据进行协作交互。子线程之间不能直接通讯，必须要通过主线程作中间数据指令的转发。

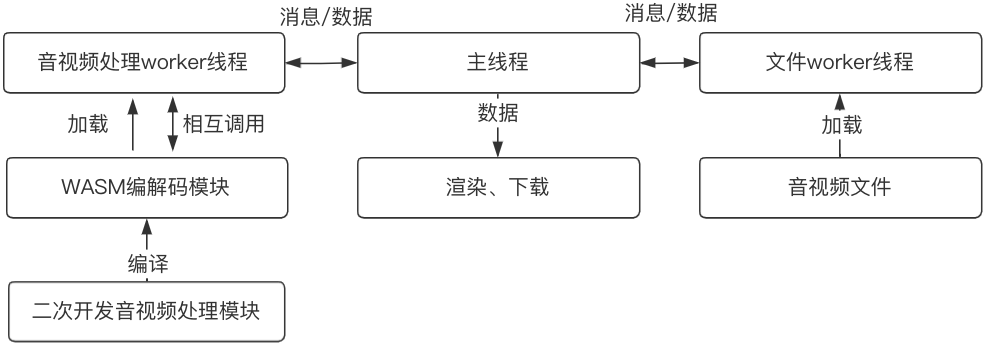


图4-1 Web编解码模块线程关系图

音视频处理子线程和WASM音视频处理模块之间需要设计具体的调用接口来实现相互调用。Web音视频编解码功能的实现更多的是子线程JavaScript运行侧来调用WASM音视频处理模块。具体的编解码核心接口如表4-3所示：

表4-3 部分编解码核心接口函数

|  |  |
| --- | --- |
| 函数名称 | 函数功能 |
| \_initDecoder  \_openDecoder  \_sendData  \_initCoder  \_openCoder  \_decodeOnePacket  \_codeOnePacket  \_malloc  \_free | 初始化解码  打开解码  发送数据  初始化编码  打开编码  解码一个音频或视频packet  编码一个音频或视频packet  分配内存  释放内存 |

音视频处理线程通过WASM模块的引用直接调用相关功能函数就可以将音视频数据进行编解码。通过初始化解码器、打开解码器之后就可以通过\_sendData不断地将读取到的音视频数据buffer发送给WASM模块，WASM模块会把读取的数据写入缓存队列，之后调用\_decodeOnePacket会解码队列中的数据。等处理完解码的数据后再通过codeOnePacket重新编码，openCoder进行重新封装，最后实现整体的Web音视频编解码流程

**4.1.3 WASM模块编解码音视频实现**

Web浏览器JavaScript主线程模块首先向文件子线程发送文件读取指令，文件子线程首先校验文件存在、类型的信息，通过FileReader流式读取文件数据并返回给主线程模块。主线程模块作为中间转发作用，进一步把读取到的文件数据发送给音视频处理子线程中的WASM模块。WASM模块通过相应的解复用器对音视频数据进行解封装并将获取到的音频流和视频流发送给编解码器，由编解码器对流数据进行解码功能和编码功能。同时对于解码后和编码后的数据可以根据主控模块的需求返回给WASM模块，再返回给主线程模块进行进一步处理。

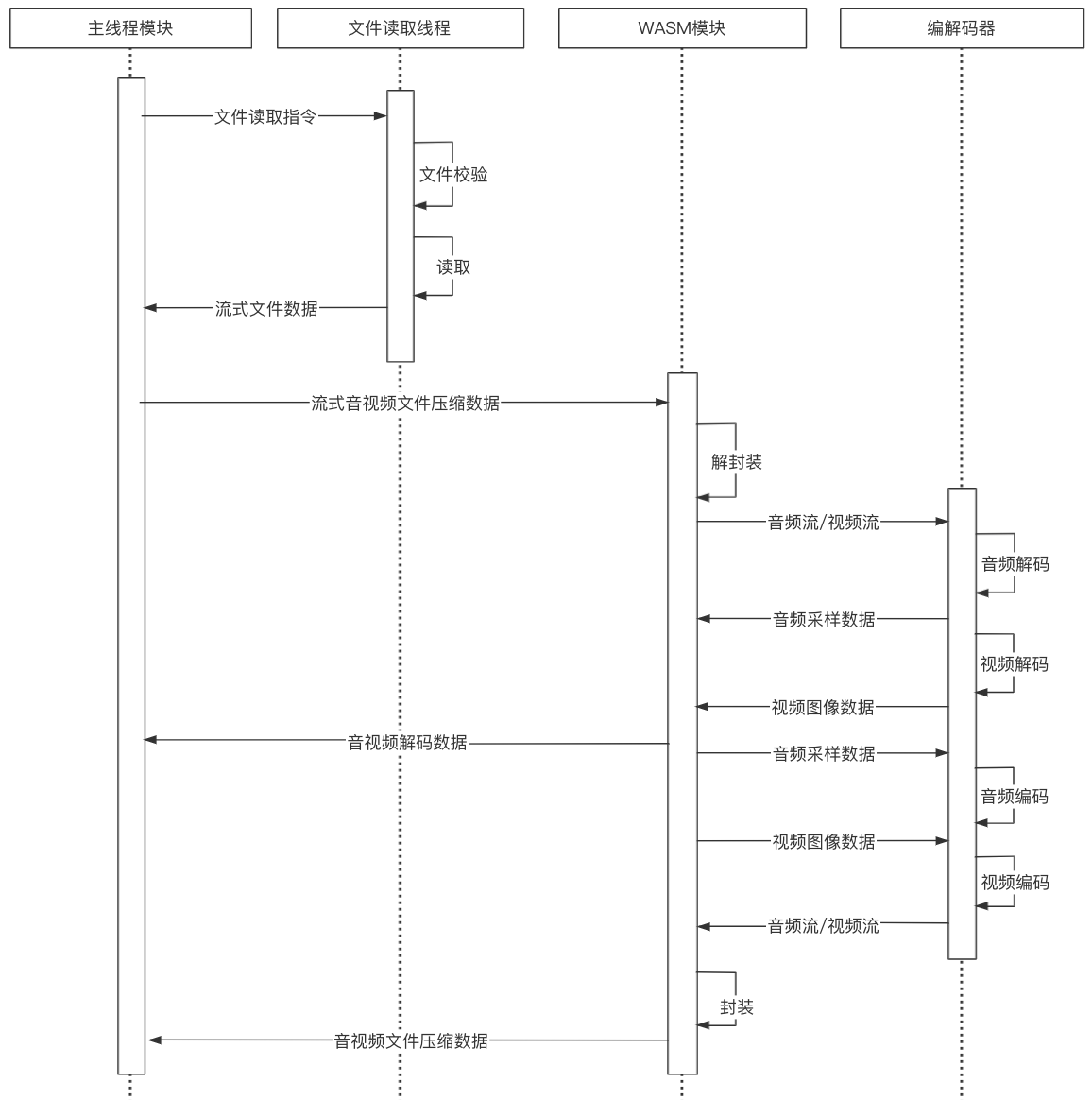


图4-2 Web编解码模块时序图

Web音视频解码模块基于多线程协作完成音视频的解封装、解码、编码、封装流程。首先主线程通过UI交互控制文件子线程加载本地音视频文件，如果加载成功则会预读取音视频文件头信息，一般地通过头信息可以获取到音视频文件的封装格式、多媒体数据的压缩标准和规范信息，通过这些信息可以帮助音视频编解码模块选择合适的解复用器以及编解码器来进行处理。如果正确识别到支持的音视频格式，就可以通过解复用器对音视频进行解封装后获取到独立的音频流和视频流。如果存在音视频没有音频流意味着音频流为空。再通过编解码器就可以对音频或者视频数据进行解码和编码。整体Web音视频编解码流程如图4-3所示：

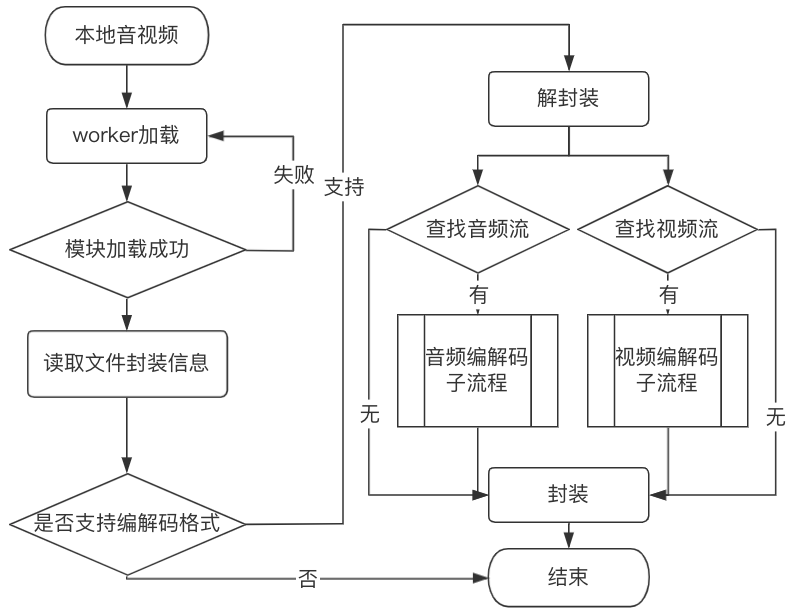


图4-3 音视频编解码概要流程图

在具体的代码逻辑中，依赖FFmpeg提供的相关数据结构扩展实现编解码流程。接下来详细介绍下解码流程，而编码流程相比较简单而且大部分过程与解码流程仅仅相反。首先通过avformat\_open\_input(&avformatContext)获取到整个音视频文件的avformatContext信息，这个数据结构贯穿整个编解码过程，因为它包含了编解码过程中所需要的编解码器信息和压缩信息。通过avformatContext的信息来注册音频和视频的解码器，用于分别生成音频和视频的avcodecContext。之后通过av\_read\_frame()读取一个avpacket，然后发送到解码队列进行排队。因为每一个avpacket可能是一个帧也可能是由多个帧组成，所以通过avcodec\_receive\_frame()获取到已经解码后的帧数据，供后续的扩展功能进行处理。整体上整个编码的实现过程如图4-4所示：

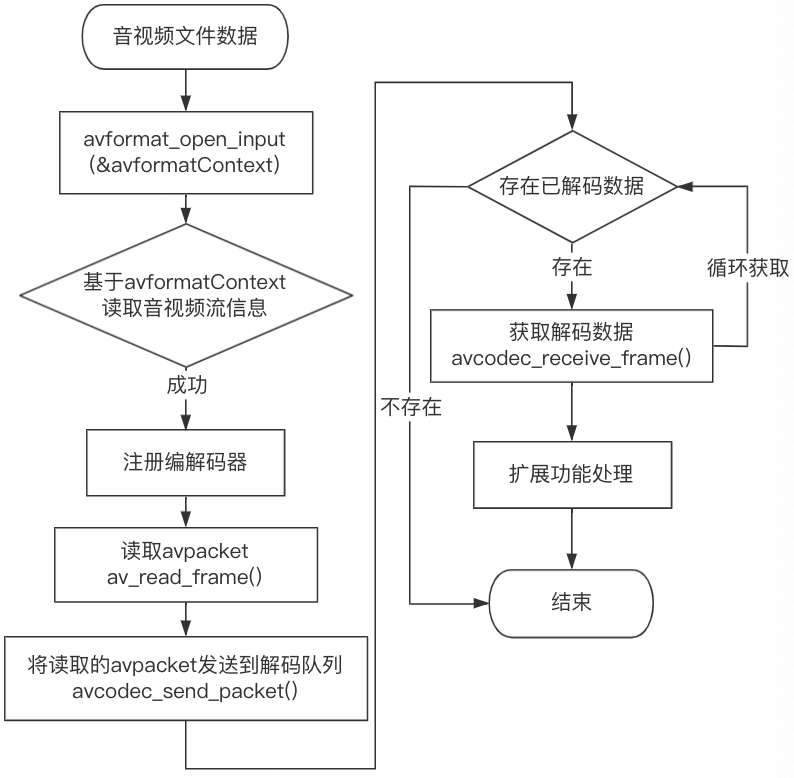


图4-4 Web音视频解码实现过程图

## 4.2弱实时性处理模块的设计与实现

本节基于上一节Web音视频编解码模块的基础上介绍弱实时性音视频剪辑处理模块如何通过构建DAG剪辑处理单元图的方式拓扑执行多剪辑任务，实现对本地音视频的高效剪辑处理。

**4.2.1 抽象剪辑处理单元设计与实现**

在弱实时性剪辑处理模块中，将剪辑处理功能抽象为多个剪辑处理子单元，包括剪切处理单元、压缩处理单元、抽取处理单元、合并处理单元以及转换处理单元。此外，还定义了两个基本单元：输入处理单元和输出处理单元。通过七种处理单元的拓扑组合可以灵活定义多个剪辑任务处理图，每一个完整的剪辑任务处理图都必须至少有一个输入处理单元以及输出处理单元。具体各个处理单元之间关系如图4-5所示：

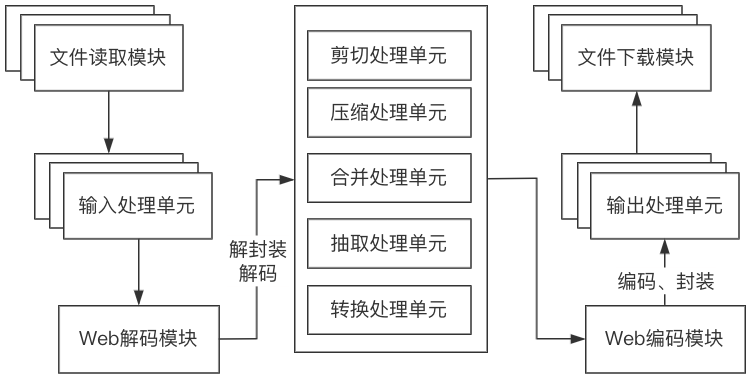


图4-5 剪辑处理单元关系图

弱实时性剪辑处理模块在Web音视频编解码流程的多个关键地方增加处理单元的逻辑执行。剪辑任务执行器会将根据定义的拓扑处理单元图创建剪辑处理pipeline，最后根据拓扑排序来执行每个处理单元的剪辑任务。

1. 输入处理单元

输入基本处理单元是根据音视频文件子线程读取的文件自动生成唯一id映射，可以通过id引用对应的音视频文件。输入处理单元的作用就是标识每个音视频文件，以及探测到音视频文件的大小、格式、类型等信息，其中最重要的是通过avformat\_open\_input函数获取到音视频的avformatContext数据，avformatContext这个结构体数据会贯穿使用在后续剪辑处理单元的各个环节。

1. 输出处理单元

输出基本处理单元用于标识一个完整的剪辑任务拓扑图的终止符号，来表示一个剪辑处理任务的结束。在输出处理单元中，还可以定义输出音视频文件的文件名、下载路径等基本信息。

1. 剪切处理单元

在剪切处理单元中，主要处理对音视频数据的剪切功能。首先改处理单元从上游节点获取到音视频上下文处理信息，通过参数配置实现对音视频数据的裁剪。

以前一个节点的输入avformatContext上下文结构体简称为in\_fmt\_ctx和剪切处理单元输入的相关时间参数开始，首先定义要输出的avformatContext上下文结构体，简称为out\_fmt\_ctx。遍历in\_fmt\_ctx中的每一个音频流、视频流以及字幕流等等，将其完全拷贝至通过avformat\_new\_stream()新生成的流，同时也将codec的context信息拷贝。最后遍历完后，再通过av\_seek\_frame()到达指定的开始时间戳开始读取每一个packet并写入到对应的输出流中，最后直到packet的时间戳大于指定的结束时间。最后再通过av\_write\_trailer()写入一些辅助信息。

具体的以时间维度剪切音视频处理单元的流程如图所示：

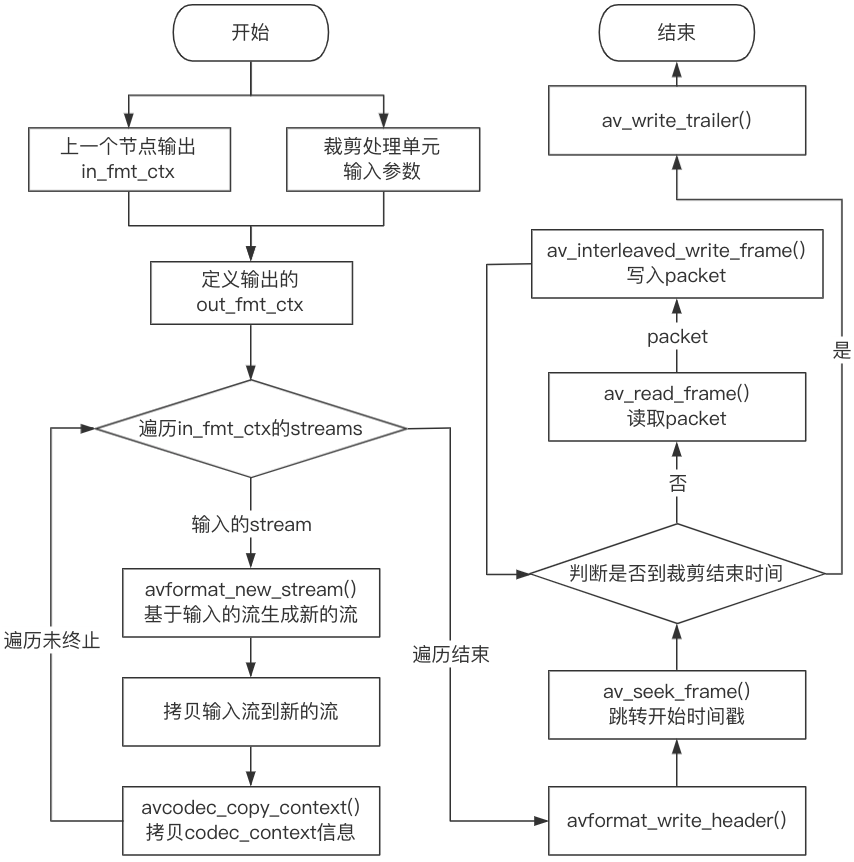


图4-7 剪切处理单元流程图

1. 抽取处理单元

抽取处理单元的主要功能是对音视频文件中的某个音频流、视频流或者字幕流进行单独抽取。FFmpeg通过宏定义了三种流的类型，分别是音频流类型AVMEDIA\_TYPE\_AUDIO、视频流类型AVMEDIA\_TYPE\_VIDEO以及字幕流 AVMEDIA\_TYPE\_SUBTITLE。

音视频抽取某一类型的流数据，首先通过av\_find\_best\_stream函数根据用户指定的流类型参数获取到对应的索引id，之后读取流中所有的packet，并根据packet.stream\_idx和流索引id比较判断，对需要抽取的流抽取到新的avformatContext中并输出。在抽取流的过程中，最重要的就是获取到对应流的索引然后对读取到的每一个packet进行比较判断即可。整个过程中，不需要对packet解码成帧，只需要拷贝对应的流即可。（缺少拷贝编解码器，及时释放之前的avformatcontext）

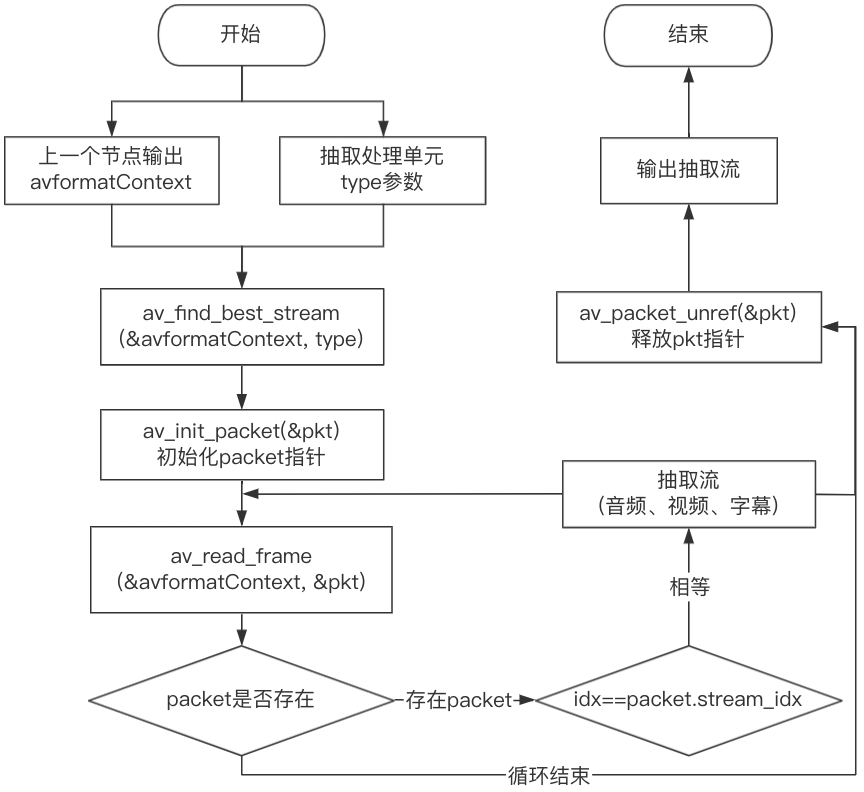
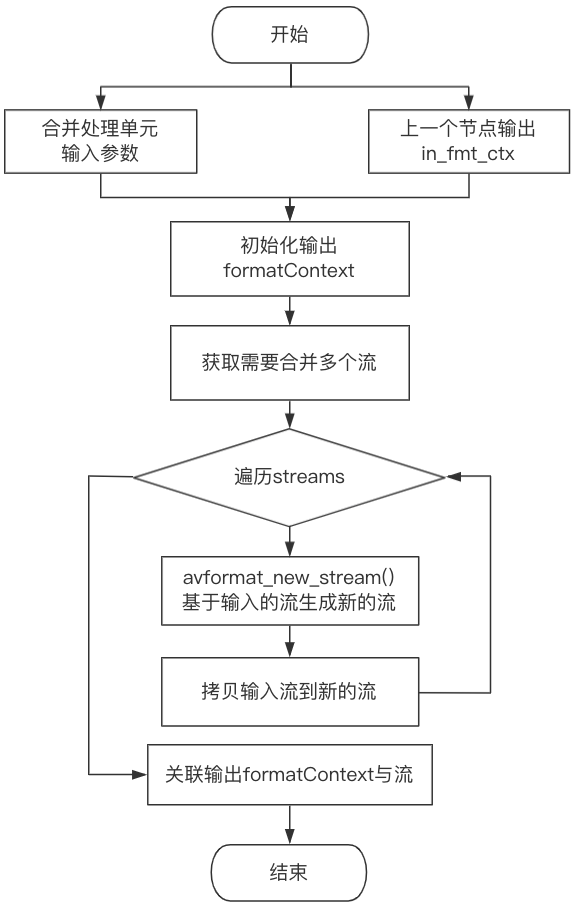


图4-8 抽取处理单元流程图

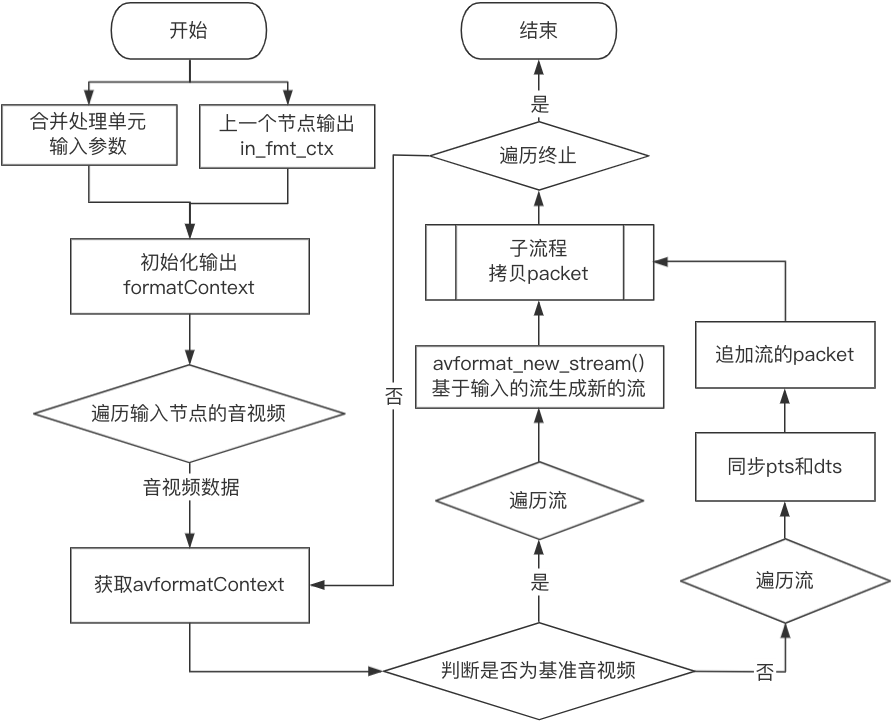
1. 合并处理单元

合并处理音视频前面的节点通常有两个及两个以上的音视频处理单元节点。同时对于合并处理可以分为两种，一种是将音频流、视频流或者字幕流进行合并，也就是对于一个视频增加音频或者增加字幕的操作；第二种是将多个音视频以次序进行前后合并的操作。

首先介绍第一种对视频增加音频或者增加字幕的操作。当该处理单元节点的输出就有两个一个是视频的video\_format\_context，另一个可以是audio\_format\_context也可以是subtitle\_format\_context。首先我们定义一个需要输出的format\_context，将几种format\_context中的流根绝类型划分拷贝到输出的format\_context中，之后只需要输出format\_context即可。



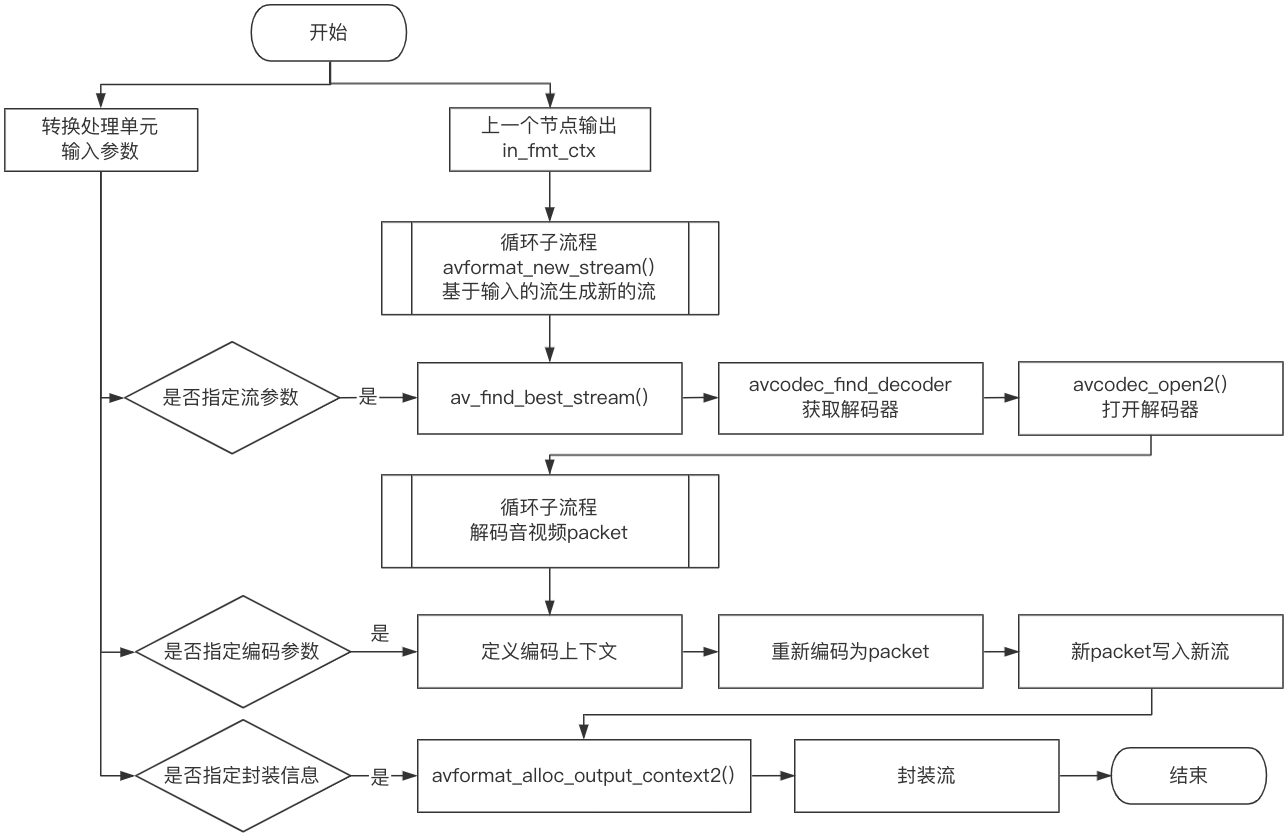
第二种是以时间为维度的合并，将多个音视频按照次序依次合并为一个音视频。具体流程如下，同样先初始化一个输出的format\_context，之后先获取第一个音视频遍历所有的流拷贝至输出上下文中，之后其他次序的音视频文件的流经过pts、dts时间戳校验后统一依次追加到之前的流信息里面。如果之前的流信息有空缺，也会因为时间戳校验会空出来。保证合并的一致性。



1. 转换处理单元

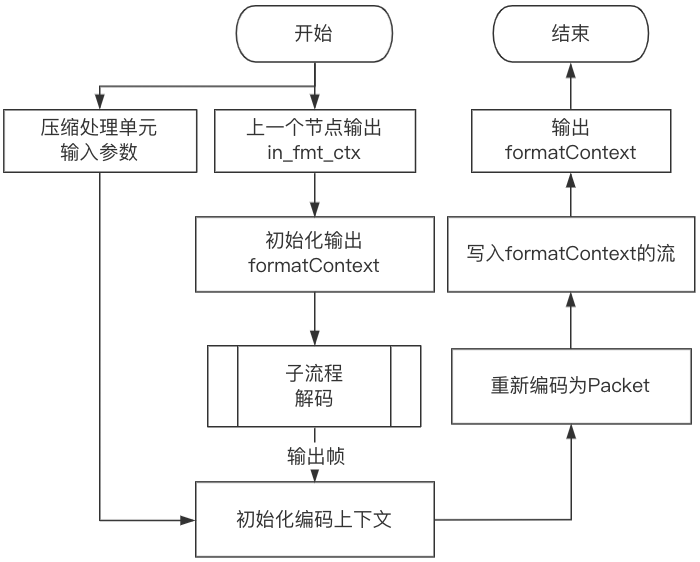
转换处理单元主要用于对音频或视频格式、编解码协议等进行转换。

通过转换处理单元输入参数来控制各个环节的流程。首先针对上个节点输出的音视频formatcontext信息，循环遍历其流信息。通过是否指定流参数进行过滤并拿到其编解码器对流中的每一个packet开始解码。根据参数指定的编解码参数来重新编码packet，最后将新的packet写入到新的音视频格式上下文，最后再根据啊是否指定封装参数信息来确定封装格式等。



1. 压缩处理单元

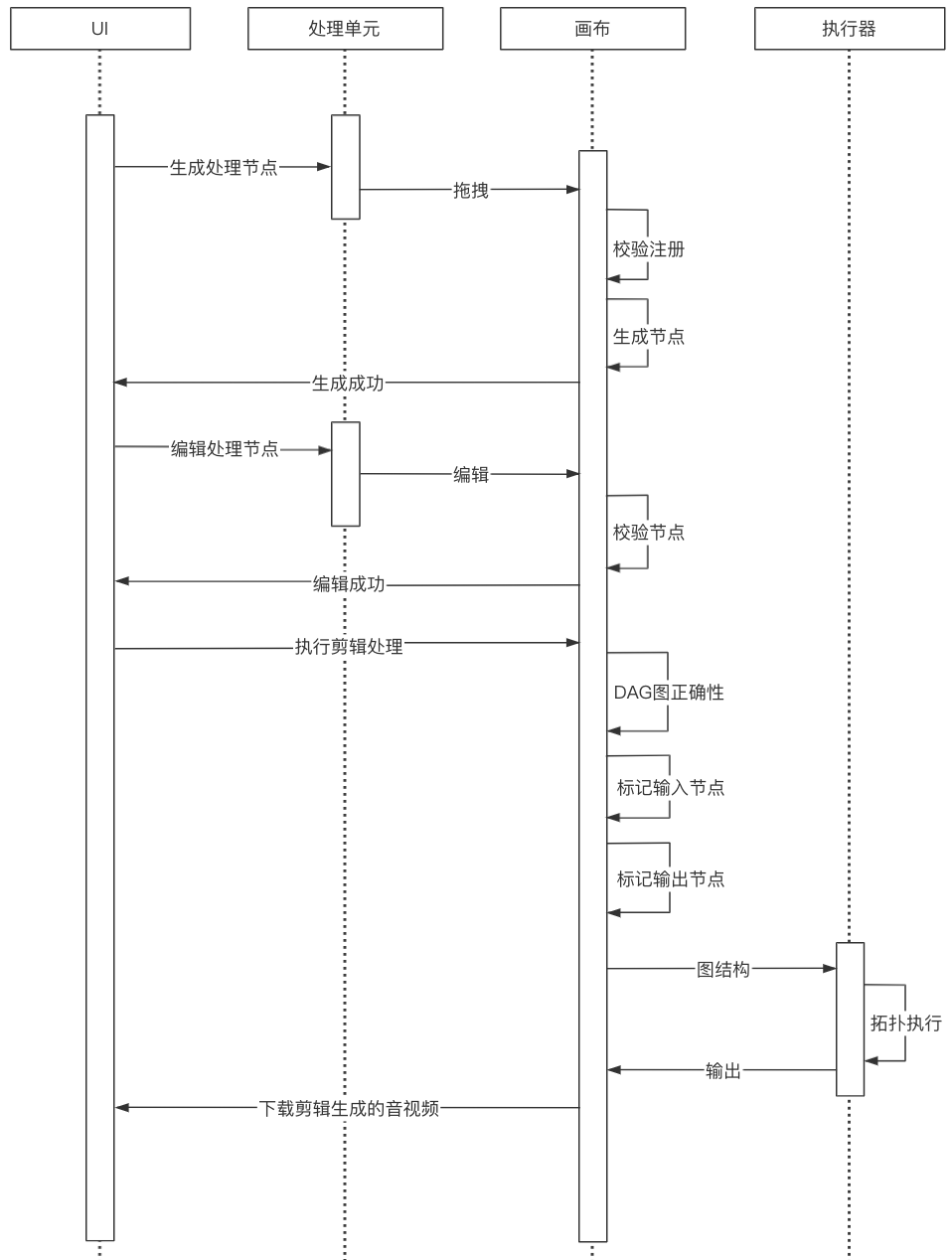
压缩处理单元主要是对音视频流中每个packet解码后，重新根据用户定义的压缩参数进行编码。



**4.2.2 基于DAG的多剪辑处理实现**

剪辑处理模块通过自定义拖拽处理单元组件，在画布上构建剪辑处理任务图。通过拓扑执行有向无环图，来逐次的执行各个处理单元。在前端页面利用react-flow-render拖拽库实现可拖拽的组件与画布。首先要注册七个处理单元并使其拖拽化，同时在画布上监听相应的拖拽事件，根据每个处理单元的唯一id生成不同的处理节点。最后将节点进行连接，使其形成一个以多个输入处理单元为开始节点，以多个输出处理单元为结束节点的有向无环图。

拓扑执行DAG剪辑任务图的时序图如下所示：



## 4.3强实时性处理模块的设计与实现

本节内容主要介绍在弱实时性处理流程之上，增加实时地将音视频处理子线程中WASM模块运行侧解码后的音视频帧数据转移到Web浏览器JavaScript主线程运行侧进行同步播放。并设计以时间戳同步、多级缓存、交替存储数组的同步策略为基础来实现强实时性播放处理模块。

**4.3.1 音视频帧同步模块的设计与实现**

交替数组

**4.3.2 简单播放的设计与实现**

**4.3.1 高级播放的设计与实现**

倍速播放

Seek播放

## 系统测试与分析

## 总结与展望

## 致谢

## 参考文献