[书摘]Android 系统级深入开发之 OpenMax 系统结构和移植内容

2011-03-01 16:36 | 1327 次阅读 | 来源：电子工业出版社 【已有 0 条评论】发表评论
关键词：图书,调试,移植,移动开发,Android | 作者：韩超 梁泉 | 收藏这篇资讯

本文节选于电子工业出版社 北京博文视点资讯有限公司推出的《Android 系统级深入开发——移植与调试》一书第 18 章，这是一本全面介绍 Android 系统级开发的作品，作者韩超和梁泉以实际的开发经验为基础，以软件工程思想为指导，以移植和调试为重点，介绍了从 Android 开源工程到一个基于实际硬件产品中的主要工作，一方面让读者清晰把握各个子系统的架构，另一方面让读者把握移植这个开发核心环节的要点。

OpenMax 系统结构和移植内容

OpenMax 是一个多媒体应用程序的框架标准。其中，OpenMax IL（集成层）技术规格定义了媒体组件接口，以便在嵌入式器件的流媒体框架中快速集成加速编解码器。

在 Android 中，OpenMax IL 层，通常可以用于多媒体引擎的插件，Android 的多媒体引擎 OpenCore 和 StageFright 都可以使用 OpenMax 作为插件，主要用于编解码（Codec）处理。

在 Android 的框架层，也定义了由 Android 封装的 OpenMax 接口，和标准的接口概念基本相同，但是使用 C++类型的接口，并且使用了 Android 的 Binder IPC 机制。Android 封装 OpenMax 的接口被 StageFright 使用，OpenCore 没有使用这个接口，而是使用其他形式对 OpenMax IL 层接口进行封装。

Android OpenMax 的基本层次结构如图 18-1 所示。

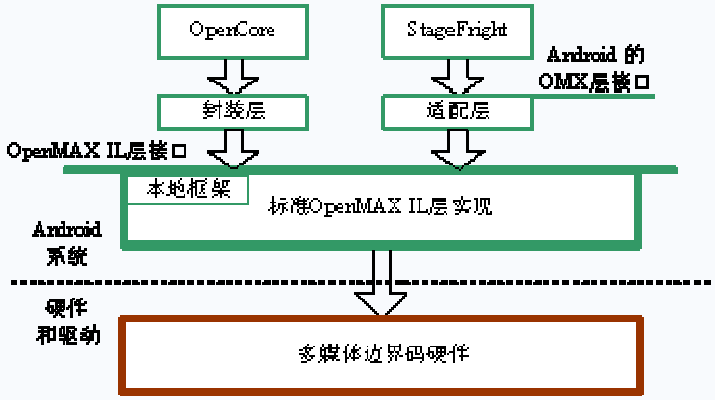

图 18-1 Android 中 OpenMax 的基本层次结构

OpenMax 系统的结构

1. OpenMax 总体层次结构

OpenMax 是一个多媒体应用程序的框架标准，由 NVIDIA 公司和 Khronos 在 2006 年推出。

OpenMax 是无授权费的，跨平台的应用程序接口 API，通过使媒体加速组件能够在开发、集成和编程环节中实现跨多操作系统和处理器硬件平台，提供全面的流媒体编解码器和应用程序便携化。

OpenMax 的官方网站如下所示：

http://www.khronos.org/openmax/

OpenMax 实际上分成三个层次，自上而下分别是，OpenMax DL（开发层），OpenMax IL（集成层）和 OpenMax AL（应用层）。三个层次的内容分别如下所示。

第一层：OpenMax DL（Development Layer，开发层）

OpenMax DL 定义了一个 API，它是音频、视频和图像功能的集合。硅供应商能够在一个新的处理器上实现并优化，然后编解码供应商使用它来编写更广泛的编解码器功能。它包括音频信号的处理功能，如 FFT 和 filter，图像原始处理，如颜色空间转换、视频原始处理，以实现例如 MPEG-4、H.264、MP3、AAC 和 JPEG 等编解码器的优化。

第二层：OpenMax IL（Integration Layer，集成层）

OpenMax IL 作为音频、视频和图像编解码器能与多媒体编解码器交互，并以统一的行为支持组件（例如，资源和皮肤）。这些编解码器或许是软硬件的混合体，对用户是透明的底层接口应用于嵌入式、移动设备。它提供了应用程序和媒体框架，透明的。S 编解码器供应商必须写私有的或者封闭的接口，集成进移动设备。IL 的主要目的是使用特征集合为编解码器提供一个系统抽象，为解决多个不同媒体系统之间轻便性的问题。

第三层：OpenMax AL（Appliction Layer，应用层）

OpenMax AL API 在应用程序和多媒体中间件之间提供了一个标准化接口，多媒体中间件提供服务以实现被期待的 API 功能。

OpenMax 的三个层次如图 18-2 所示。

OpenMax API 将会与处理器一同提供，以使库和编解码器开发者能够高速有效地利用新器件的完整加速潜能，无须担心其底层的硬件结构。该标准是针对嵌入式设备和移动设备的多媒体软件架构。在架构底层上为多媒体的编解码和数据处理定义了一套统一的编程接口，对多媒体数据的处理功能进行系统级抽象，为用户屏蔽了底层的细节。因此，多媒体应用程序和多媒体框架通过 OpenMax IL 可以以一种统一的方式来使用编解码和其他多媒体数据处理功能，具有了跨越软硬件平台的移植性。

提示：在实际的应用中，OpenMax 的三个层次中使用较多的是 OpenMax IL 集成层，由于操作系统到硬件的差异和多媒体应用的差异，OpenMax 的 DL 和 AL 层使用相对较少。

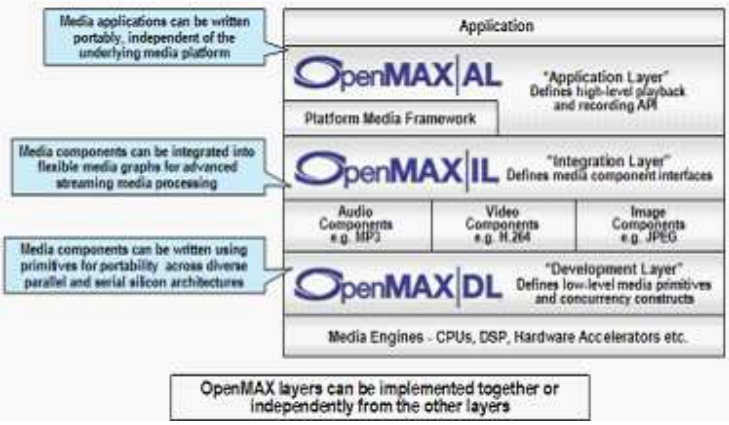

图 18-2 OpenMax 的三个层次

2．OpenMax IL 层的结构

OpenMax IL 目前已经成为了事实上的多媒体框架标准。嵌入式处理器或者多媒体 编解码模块的硬件生产者，通常提供标准的 OpenMax IL 层的软件接口，这样软件的开发者就可以基于这个层次的标准化接口进行多媒体程序的开发。

OpenMax IL 的接口层次结构适中，既不是硬件编解码的接口，也不是应用程序层的接口，因此比较容易实现标准化。

OpenMax IL 的层次结构如图 18-3 所示。

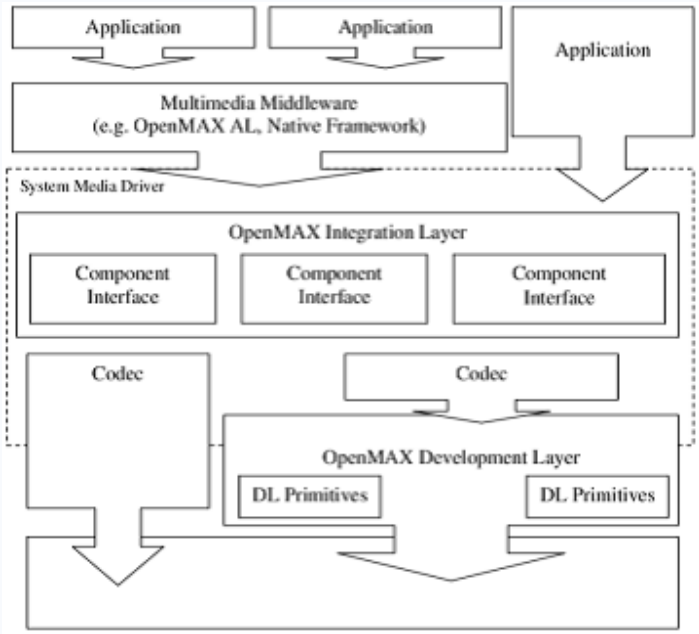

图 18-3 OpenMax IL 的层次结构

图 18-3 中的虚线中的内容是 OpenMax IL 层的内容，其主要实现了 OpenMax IL 中的各个组件（Component）。对下层，OpenMax IL 可以调用 OpenMax DL 层的接口，也可以直接调用各种 Codec 实现。对上层，OpenMax IL 可以给 OpenMax AL 层等框架层（Middleware）调用，也可以给应用程序直接调用。

OpenMax IL 主要内容如下所示。

客户端（Client）：OpenMax IL 的调用者

组件（Component）：OpenMax IL 的单元，每一个组件实现一种功能

端口（Port）：组件的输入输出接口

隧道化（Tunneled）：让两个组件直接连接的方式

OpenMax IL 的基本运作过程如图 18-4 所示。

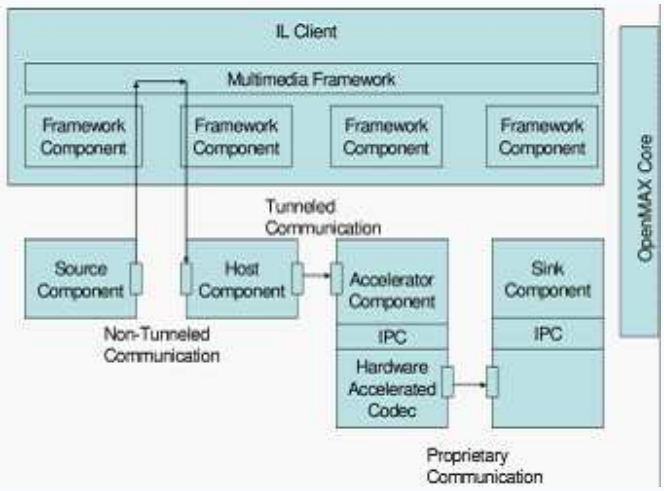

图 18-4 OpenMax IL 的基本运作过程

如图 18-4 所示，OpenMAL IL 的客户端，通过调用四个 OpenMAL IL 组件，实现了一个功能。四个组件分别是 Source 组件、Host 组件、Accelerator 组件和 Sink 组件。Source 组件只有一个输出端口；而 Host 组件有一个输入端口和一个输出端口；Accelerator 组件具有一个输入端口，调用了硬件的编解码器，加速主要体现在这个环节上。Accelerator 组件和 Sink 组件通过私有通讯方式在内部进行连接，没有经过明确的组件端口。

OpenMAL IL 在使用的时候，其数据流也有不同的处理方式：既可以经由客户端，也可以不经由客户端。图 18-4 中，Source 组件到 Host 组件的数据流就是经过客户端的；而 Host 组件到 Accelerator 组件的数据流就没有经过客户端，使用了隧道化的方式；Accelerator 组件和 Sink 组件甚至可以使用私有的通讯方式。

OpenMax Core 是辅助各个组件运行的部分，它通常需要完成各个组件的初始化等工作，在真正运行过程中，重点的是各个 OpenMax IL 的组件，OpenMax Core 不是重点，也不是标准。

OpenMAL IL 的组件是 OpenMax IL 实现的核心内容，一个组件以输入、输出端口为接口，端口可以被连接到另一个组件上。外部对组件可以发送命令，还进行设置/获取参数、配置等内容。组件的端口可以包含缓冲区（Buffer）的队列。

组件的处理的核心内容是：通过输入端口消耗 Buffer，通过输出端口填充 Buffer，由此多组件相联接可以构成流式的处理。

OpenMAL IL 中一个组件的结构如图 18-5 所示。

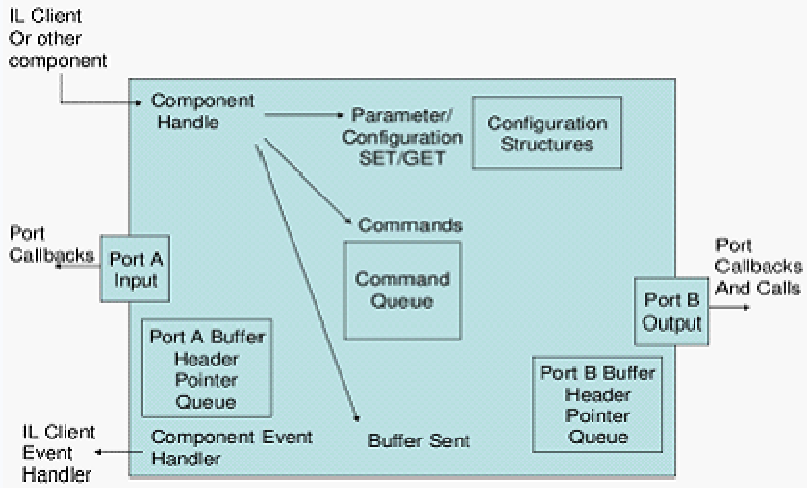

图 18-5 OpenMAL IL 中一个组件的结构

组件的功能和其定义的端口类型密切相关，通常情况下：只有一个输出端口的，为 Source 组件；只有一个输入端口的，为 Sink 组件；有多个输入端口，一个输出端口的为 Mux 组件；有一个输入端口，多个输出端口的为 DeMux 组件；输入输出端口各一个组件的为中间处理环节，这是最常见的组件。

端口具体支持的数据也有不同的类型。例如，对于一个输入、输出端口各一个组件，其输入端口使用 MP3 格式的数据，输出端口使用 PCM 格式的数据，那么这个组件就是一个 MP3 解码组件。

隧道化（Tunneled）是一个关于组件连接方式的概念。通过隧道化可以将不同的组件的一个输入端口和一个输出端口连接到一起，在这种情况下，两个组件的处理过程合并，共同处理。尤其对于单输入和单输出的组件，两个组件将作为类似一个使用。

3．Android 中 OpenMax 的使用情况

Android 系统的一些部分对 OpenMax IL 层进行使用，基本使用的是标准 OpenMax IL 层的接口，只是进行了简单的封装。标准的 OpenMax IL 实现很容易以插件的形式加入到 Android 系统中。

Android 的多媒体引擎 OpenCore 和 StageFright 都可以使用 OpenMax 作为多媒体编解码的插件，只是没有直接使用 OpenMax IL 层提供的纯 C 接口，而是对其进行了一定的封装。

在 Android2.x 版本之后，Android 的框架层也对 OpenMax IL 层的接口进行了封装定义，甚至使用 Android 中的 Binder IPC 机制。Stagefright 使用了这个层次的接口，OpenCore 没有使用。

提示：OpenCore 使用 OpenMax IL 层作为编解码插件在前，Android 框架层封装 OpenMax 接口在后面的版本中才引入。

Android OpenMax 实现的内容

Android 中使用的主要是 OpenMax 的编解码功能。虽然 OpenMax 也可以生成输入、输出、文件解析—构建等组件，但是在各个系统（不仅是 Android）中使用的最多的还是编解码组件。媒体的输入、输出环节和系统的关系很大，引入 OpenMax 标准比较麻烦；文件解析—构建环节一般不需要使用硬件加速。编解码组件也是最能体现硬件加速的环节，因此最常使用。

在 Android 中实现 OpenMax IL 层和标准的 OpenMax IL 层的方式基本，一般需要实现以下两个环节。

编解码驱动程序：位于 Linux 内核空间，需要通过 Linux 内核调用驱动程序，通常使用非标准的驱动程序

OpenMax IL 层：根据 OpenMax IL 层的标准头文件实现不同功能的组件

Android 中还提供了 OpenMax 的适配层接口（对 OpenMax IL 的标准组件进行封装适配），它作为 Android 本地层的接口，可以被 Android 的多媒体引擎调用。

[书摘]Android 系统级深入开发之 OpenMax 系统结构和移植内容

2011-03-01 16:36 | 1327 次阅读 | 来源：电子工业出版社 【已有 0 条评论】发表评论
关键词：图书,调试,移植,移动开发,Android | 作者：韩超 梁泉 | 收藏这篇资讯

本文节选于电子工业出版社 北京博文视点资讯有限公司推出的《Android 系统级深入开发——移植与调试》一书第 18 章，这是一本全面介绍 Android 系统级开发的作品，作者韩超和梁泉以实际的开发经验为基础，以软件工程思想为指导，以移植和调试为重点，介绍了从 Android 开源工程到一个基于实际硬件产品中的主要工作，一方面让读者清晰把握各个子系统的架构，另一方面让读者把握移植这个开发核心环节的要点。

OpenMax 系统结构和移植内容

OpenMax 是一个多媒体应用程序的框架标准。其中，OpenMax IL（集成层）技术规格定义了媒体组件接口，以便在嵌入式器件的流媒体框架中快速集成加速编解码器。

在 Android 中，OpenMax IL 层，通常可以用于多媒体引擎的插件，Android 的多媒体引擎 OpenCore 和 StageFright 都可以使用 OpenMax 作为插件，主要用于编解码（Codec）处理。

在 Android 的框架层，也定义了由 Android 封装的 OpenMax 接口，和标准的接口概念基本相同，但是使用 C++类型的接口，并且使用了 Android 的 Binder IPC 机制。Android 封装 OpenMax 的接口被 StageFright 使用，OpenCore 没有使用这个接口，而是使用其他形式对 OpenMax IL 层接口进行封装。

Android OpenMax 的基本层次结构如图 18-1 所示。

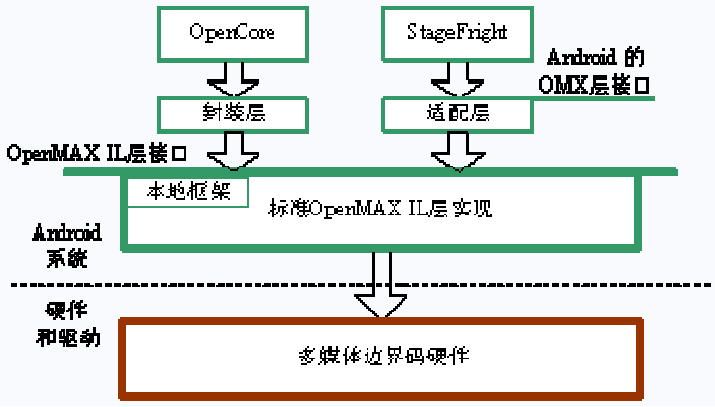

图 18-1 Android 中 OpenMax 的基本层次结构

OpenMax 系统的结构

1．OpenMax 总体层次结构

OpenMax 是一个多媒体应用程序的框架标准，由 NVIDIA 公司和 Khronos 在 2006 年推出。

OpenMax 是无授权费的，跨平台的应用程序接口 API，通过使媒体加速组件能够在开发、集成和编程环节中实现跨多操作系统和处理器硬件平台，提供全面的流媒体编解码器和应用程序便携化。

OpenMax 的官方网站如下所示：

http://www.khronos.org/openmax/

OpenMax 实际上分成三个层次，自上而下分别是，OpenMax DL（开发层），OpenMax IL（集成层）和 OpenMax AL（应用层）。三个层次的内容分别如下所示。

第一层：OpenMax DL（Development Layer，开发层）

OpenMax DL 定义了一个 API，它是音频、视频和图像功能的集合。硅供应商能够在一个新的处理器上实现并优化，然后编解码供应商使用它来编写更广泛的编解码器功能。它包括音频信号的处理功能，如 FFT 和 filter，图像原始处理，如颜色空间转换、视频原始处理，以实现例如 MPEG-4、H.264、MP3、AAC 和 JPEG 等编解码器的优化。

第二层：OpenMax IL（Integration Layer，集成层）

OpenMax IL 作为音频、视频和图像编解码器能与多媒体编解码器交互，并以统一的行为支持组件（例如，资源和皮肤）。这些编解码器或许是软硬件的混合体，对用户是透明的底层接口应用于嵌入式、移动设备。它提供了应用程序和媒体框架，透明的。S 编解码器供应商必须写私有的或者封闭的接口，集成进移动设备。IL 的主要目的是使用特征集合为编解码器提供一个系统抽象，为解决多个不同媒体系统之间轻便性的问题。

第三层：OpenMax AL（Appliction Layer，应用层）

OpenMax AL API 在应用程序和多媒体中间件之间提供了一个标准化接口，多媒体中间件提供服务以实现被期待的 API 功能。

OpenMax 的三个层次如图 18-2 所示。

OpenMax API 将会与处理器一同提供，以使库和编解码器开发者能够高速有效地利用新器件的完整加速潜能，无须担心其底层的硬件结构。该标准是针对嵌入式设备和移动设备的多媒体软件架构。在架构底层上为多媒体的编解码和数据处理定义了一套统一的编程接口，对多媒体数据的处理功能进行系统级抽象，为用户屏蔽了底层的细节。因此，多媒体应用程序和多媒体框架通过 OpenMax IL 可以以一种统一的方式来使用编解码和其他多媒体数据处理功能，具有了跨越软硬件平台的移植性。

提示：在实际的应用中，OpenMax 的三个层次中使用较多的是 OpenMax IL 集成层，由于操作系统到硬件的差异和多媒体应用的差异，OpenMax 的 DL 和 AL 层使用相对较少。

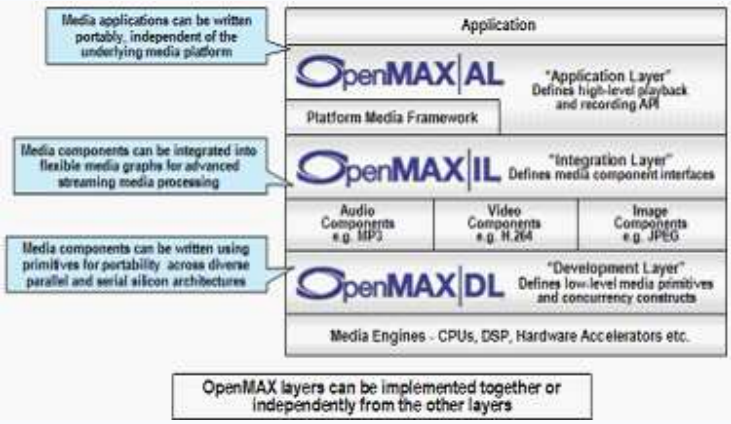

图 18-2 OpenMax 的三个层次

2．OpenMax IL 层的结构

OpenMax IL 目前已经成为了事实上的多媒体框架标准。嵌入式处理器或者多媒体 编解码模块的硬件生产者，通常提供标准的 OpenMax IL 层的软件接口，这样软件的开发者就可以基于这个层次的标准化接口进行多媒体程序的开发。

OpenMax IL 的接口层次结构适中，既不是硬件编解码的接口，也不是应用程序层的接口，因此比较容易实现标准化。

OpenMax IL 的层次结构如图 18-3 所示。

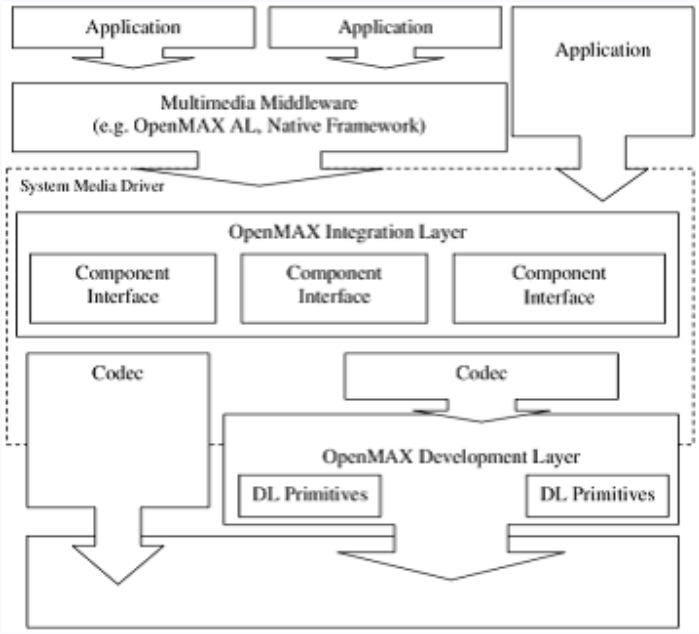

图 18-3 OpenMax IL 的层次结构

图 18-3 中的虚线中的内容是 OpenMax IL 层的内容，其主要实现了 OpenMax IL 中的各个组件（Component）。对下层，OpenMax IL 可以调用 OpenMax DL 层的接口，也可以直接调用各种 Codec 实现。对上层，OpenMax IL 可以给 OpenMax AL 层等框架层（Middleware）调用，也可以给应用程序直接调用。

OpenMax IL 主要内容如下所示。

客户端（Client）：OpenMax IL 的调用者

组件（Component）：OpenMax IL 的单元，每一个组件实现一种功能

端口（Port）：组件的输入输出接口

隧道化（Tunneled）：让两个组件直接连接的方式

OpenMax IL 的基本运作过程如图 18-4 所示。

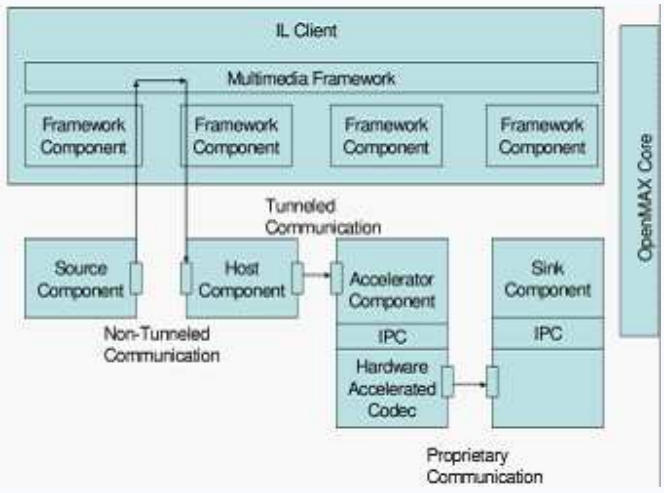

图 18-4 OpenMax IL 的基本运作过程

如图 18-4 所示，OpenMAL IL 的客户端，通过调用四个 OpenMAL IL 组件，实现了一个功能。四个组件分别是 Source 组件、Host 组件、Accelerator 组件和 Sink 组件。Source 组件只有一个输出端口；而 Host 组件有一个输入端口和一个输出端口；Accelerator 组件具有一个输入端口，调用了硬件的编解码器，加速主要体现在这个环节上。Accelerator 组件和 Sink 组件通过私有通讯方式在内部进行连接，没有经过明确的组件端口。

OpenMAL IL 在使用的时候，其数据流也有不同的处理方式：既可以经由客户端，也可以不经由客户端。图 18-4 中，Source 组件到 Host 组件的数据流就是经过客户端的；而 Host 组件到 Accelerator 组件的数据流就没有经过客户端，使用了隧道化的方式；Accelerator 组件和 Sink 组件甚至可以使用私有的通讯方式。

OpenMax Core 是辅助各个组件运行的部分，它通常需要完成各个组件的初始化等工作，在真正运行过程中，重点的是各个 OpenMax IL 的组件，OpenMax Core 不是重点，也不是标准。

OpenMAL IL 的组件是 OpenMax IL 实现的核心内容，一个组件以输入、输出端口为接口，端口可以被连接到另一个组件上。外部对组件可以发送命令，还进行设置/获取参数、配置等内容。组件的端口可以包含缓冲区（Buffer）的队列。

组件的处理的核心内容是：通过输入端口消耗 Buffer，通过输出端口填充 Buffer，由此多组件相联接可以构成流式的处理。

OpenMAL IL 中一个组件的结构如图 18-5 所示。

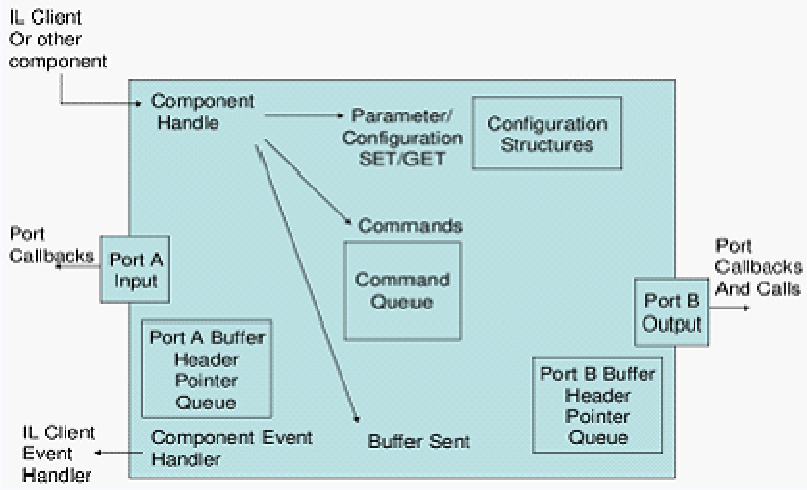

图 18-5 OpenMAL IL 中一个组件的结构

组件的功能和其定义的端口类型密切相关，通常情况下：只有一个输出端口的，为 Source 组件；只有一个输入端口的，为 Sink 组件；有多个输入端口，一个输出端口的为 Mux 组件；有一个输入端口，多个输出端口的为 DeMux 组件；输入输出端口各一个组件的为中间处理环节，这是最常见的组件。

端口具体支持的数据也有不同的类型。例如，对于一个输入、输出端口各一个组件，其输入端口使用 MP3 格式的数据，输出端口使用 PCM 格式的数据，那么这个组件就是一个 MP3 解码组件。

隧道化（Tunneled）是一个关于组件连接方式的概念。通过隧道化可以将不同的组件的一个输入端口和一个输出端口连接到一起，在这种情况下，两个组件的处理过程合并，共同处理。尤其对于单输入和单输出的组件，两个组件将作为类似一个使用。

3．Android 中 OpenMax 的使用情况

Android 系统的一些部分对 OpenMax IL 层进行使用，基本使用的是标准 OpenMax IL 层的接口，只是进行了简单的封装。标准的 OpenMax IL 实现很容易以插件的形式加入到 Android 系统中。

Android 的多媒体引擎 OpenCore 和 StageFright 都可以使用 OpenMax 作为多媒体编解码的插件，只是没有直接使用 OpenMax IL 层提供的纯 C 接口，而是对其进行了一定的封装。

在 Android2.x 版本之后，Android 的框架层也对 OpenMax IL 层的接口进行了封装定义，甚至使用 Android 中的 Binder IPC 机制。Stagefright 使用了这个层次的接口，OpenCore 没有使用。

提示：OpenCore 使用 OpenMax IL 层作为编解码插件在前，Android 框架层封装 OpenMax 接口在后面的版本中才引入。

Android OpenMax 实现的内容

Android 中使用的主要是 OpenMax 的编解码功能。虽然 OpenMax 也可以生成输入、输出、文件解析—构建等组件，但是在各个系统（不仅是 Android）中使用的最多的还是编解码组件。媒体的输入、输出环节和系统的关系很大，引入 OpenMax 标准比较麻烦；文件解析—构建环节一般不需要使用硬件加速。编解码组件也是最能体现硬件加速的环节，因此最常使用。

在 Android 中实现 OpenMax IL 层和标准的 OpenMax IL 层的方式基本，一般需要实现以下两个环节。

编解码驱动程序：位于 Linux 内核空间，需要通过 Linux 内核调用驱动程序，通常使用非标准的驱动程序

OpenMax IL 层：根据 OpenMax IL 层的标准头文件实现不同功能的组件

Android 中还提供了 OpenMax 的适配层接口（对 OpenMax IL 的标准组件进行封装适配），它作为 Android 本地层的接口，可以被 Android 的多媒体引擎调用。

[书摘]Android 开发之 OMAP 平台 OpenMax IL 的硬件实现

2011-03-01 16:59 | 1061 次阅读 | 来源：电子工业出版社 【已有 1 条评论】发表评论
关键词：Android,移动开发,移植,调试,图书 | 作者：韩超 梁泉 | 收藏这篇资讯

本文节选于电子工业出版社 北京博文视点资讯有限公司推出的《Android 系统级深入开发——移植与调试》一书第 18 章，这是一本全面介绍 Android 系统级开发的作品，作者韩超和梁泉以实际的开发经验为基础，以软件工程思想为指导，以移植和调试为重点，介绍了从 Android 开源工程到一个基于实际硬件产品中的主要工作，一方面让读者清晰把握各个子系统的架构，另一方面让读者把握移植这个开发核心环节的要点。

一： TI OpenMax IL 实现的结构和机制

Android 的开源代码中，已经包含了 TI 的 OpenMax IL 层的实现代码，其路径如下所示：

hardware/ti/omap3/omx/

其中包含的主要目录如下所示。

system：OpenMax 核心和公共部分

audio：音频处理部分的 OpenMax IL 组件

video：视频处理部分 OpenMax IL 组件

image：图像处理部分 OpenMax IL 组件

TI OpenMax IL 实现的结构如图 18-7 所示。

在 TI OpenMax IL 实现中，最上面的内容是 OpenMax 的管理者用于管理和初始化，中间层是各个编解码单元的 OpenMax IL 标准组件，下层是 LCML 层，供各个 OpenMax IL 标准组件所调用。

TI OpenMax IL 实现的公共部分在 system/src/openmax_il/目录中，主要的内容如下所示。

omx_core/src：OpenMax IL 的核心，生成动态库 libOMX_Core.so

lcml/：LCML 的工具库，生成动态库 libLCML.so

TI OpenMax IL 的视频（Video）相关的组件在 video/src/openmax_il/目录中，主要的内容如下所示。

prepost_processor：Video 数据的前处理和后处理，生成动态库 libOMX.TI.VPP.so

video_decode：Video 解码器，生成动态库 libOMX.TI.Video.Decoder.so

video_encode：Video 编码器，生成动态库 libOMX.TI.Video.encoder.so

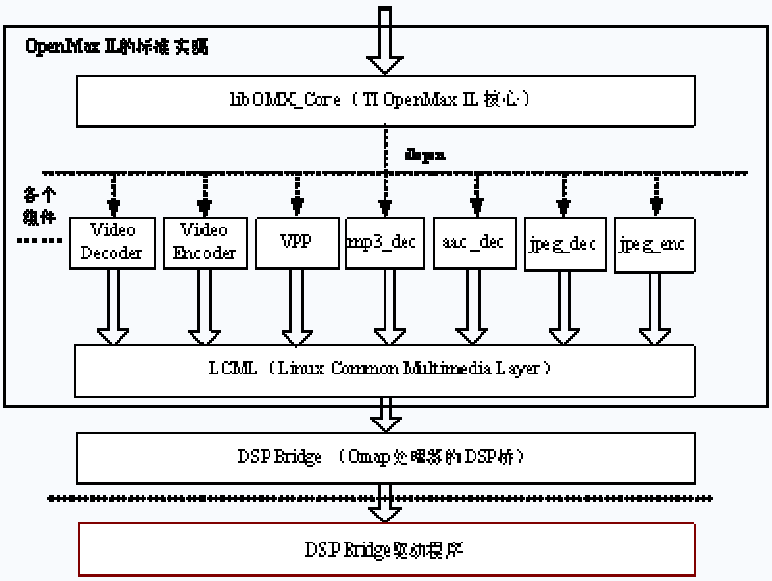

图 18-7 TI OpenMax IL 实现的结构

TI OpenMax IL 的音频（Audio）相关的组件在 audio/src/openmax_il/目录中，主要的内容如下所示。

g711_dec：G711 解码器，生成动态库 libOMX.TI.G711.decode.so

g711_enc：G711 编码器，生成动态库 libOMX.TI.G711.encode.so

g722_dec：G722 解码器，生成动态库 libOMX.TI.G722.decode.so

g722_enc：G722 编码器，生成动态库 libOMX.TI.G722.encode.so

g726_dec：G726 解码器，生成动态库 libOMX.TI.G726.decode.so

g726_enc：G726 编码器，生成动态库 libOMX.TI.G726.encode.so

g729_dec：G729 解码器，生成动态库 libOMX.TI.G729.decode.so

g729_enc：G720 编码器，生成动态库 libOMX.TI.G729.encode.so

nbamr_dec：AMR 窄带解码器，生成动态库 libOMX.TI.AMR.decode.so

nbamr_enc：AMR 窄带编码器，生成动态库 libOMX.TI.AMR.encode.so

wbamr_dec：AMR 宽带解码器，生成动态库 libOMX.TI.WBAMR.decode.so

wbamr_enc：AMR 宽带编码器，生成动态库 libOMX.TI.WBAMR.encode.so

mp3_dec：MP3 解码器，生成动态库 libOMX.TI.MP3.decode.so

aac_dec：AAC 解码器，生成动态库 libOMX.TI.AAC.decode.so

aac_enc：AAC 编码器，生成动态库 libOMX.TI.AAC.encode.so

wma_dec：WMA 解码器，生成动态库 libOMX.TI.WMA.decode.so

TI OpenMax IL 的图像（Image）相关的组件在 image/src/openmax_il/目录中，主要的内容如下所示。

jpeg_enc：JPEG 编码器，生成动态库 libOMX.TI.JPEG.Encoder.so

jpeg_dec：JPEG 解码器，生成动态库 libOMX.TI.JPEG.decoder.so

二： TI OpenMax IL 的核心和公共内容

LCML 的全称是“Linux Common Multimedia Layer”，是 TI 的 Linux 公共多媒体层。在 OpenMax IL 的实现中，这个内容在 system/src/openmax_il/lcml/ 目录中，主要文件是子目录 src 中的 LCML_DspCodec.c 文件。通过调用 DSPBridge 的内容， 让 ARM 和 DSP 进行通信，然 DSP 进行编解码方面的处理。DSP 的运行还需要固件的支持。

TI OpenMax IL 的核心实现在 system/src/openmax_il/omx_core/目录中，生成 TI OpenMax IL 的核心库 libOMX_Core.so。

其中子目录 src 中的 OMX_Core.c 为主要文件，其中定义了编解码器的名称等，其片断如下所示：

```
char *tComponentName[MAXCOMP][2] = {
   {"OMX.TI.JPEG.decoder", "image_decoder.jpeg"},  /* 图像和视频编解码器 */
   {"OMX.TI.JPEG.Encoder", "image_encoder.jpeg"},
   {"OMX.TI.Video.Decoder", "video_decoder.avc"},
   {"OMX.TI.Video.Decoder", "video_decoder.mpeg4"},
   {"OMX.TI.Video.Decoder", "video_decoder.wmv"},
   {"OMX.TI.Video.encoder", "video_encoder.mpeg4"},
   {"OMX.TI.Video.encoder", "video_encoder.h263"},
   {"OMX.TI.Video.encoder", "video_encoder.avc"},
   /* .....省略 ， 语音相关组件*/
#ifdef BUILD_WITH_TI_AUDIO                          /* 音频编解码器 */
   {"OMX.TI.MP3.decode", "audio_decoder.mp3"},
   {"OMX.TI.AAC.encode", "audio_encoder.aac"},
   {"OMX.TI.AAC.decode", "audio_decoder.aac"},
   {"OMX.TI.WMA.decode", "audio_decoder.wma"},
   {"OMX.TI.WBAMR.decode", "audio_decoder.amrwb"},
```

```
    {"OMX.TI.AMR.decode", "audio_decoder.amrnb"},
    {"OMX.TI.AMR.encode", "audio_encoder.amrnb"},
    {"OMX.TI.WBAMR.encode", "audio_encoder.amrwb"},
#endif
    {NULL, NULL},
};
```

tComponentName 数组的各个项中，第一个表示编解码库内容，第二个表示库所实现的功能。

其中，TIOMX_GetHandle()函数用于获得各个组建的句柄，其实现的主要片断如下所示：

```
OMX_ERRORTYPE TIOMX_GetHandle( OMX_HANDLETYPE* pHandle, OMX_STRING cComponentName,
    OMX_PTR pAppData, OMX_CALLBACKTYPE* pCallBacks)
{
    static const char prefix[] = "lib";
    static const char postfix[] = ".so";
    OMX_ERRORTYPE (*pComponentInit)(OMX_HANDLETYPE*);
    OMX_ERRORTYPE err = OMX_ErrorNone;
    OMX_COMPONENTTYPE *componentType;
    const char* pErr = dlerror();
// ..... 省略错误处理内容
    int i = 0;
    for(i=0; i< COUNTOF(pModules); i++) {       // 循环查找
        if(pModules[i] == NULL) break;
    }
// ..... 省略错误处理内容
    int refIndex = 0;
    for (refIndex=0; refIndex < MAX_TABLE_SIZE; refIndex++) {
```

```
// 循环查找组件列表
        if (strcmp(componentTable[refIndex].name, cComponentName) == 0) {
            if (componentTable[refIndex].refCount >= MAX_CONCURRENT_INSTANCES) {
// ..... 省略错误处理内容
            } else {
                char buf[sizeof(prefix) + MAXNAMESIZE + sizeof(postfix)];
                strcpy(buf, prefix);
                strcat(buf, cComponentName);
                strcat(buf, postfix);
                pModules[i] = dlopen(buf, RTLD_LAZY | RTLD_GLOBAL);
            // ..... 省略错误处理内容
            // 动态取出初始化的符号
                pComponentInit = dlsym(pModules[i], "OMX_ComponentInit");
                pErr = dlerror();
            // ..... 省略错误处理内容
                *pHandle = malloc(sizeof(OMX_COMPONENTTYPE));
            // ..... 省略错误处理内容
                pComponents[i] = *pHandle;
                componentType = (OMX_COMPONENTTYPE*) *pHandle;
                componentType->nSize = sizeof(OMX_COMPONENTTYPE);
                err = (*pComponentInit)(*pHandle);   // 执行初始化工作
            // ..... 省略部分内容
            }
        }
    }
```

```
    err = OMX_ErrorComponentNotFound;
    goto UNLOCK_MUTEX;
// ..... 省略部分内容
    return (err);
}
```

在 TIOMX_GetHandle()函数中，根据 tComponentName 数组中动态库的名称，动态打开各个编解码实现的动态库，取出其中的 OMX_ComponentInit 符号来执行各个组件的初始化。

三： 一个 TI OpenMax IL 组件的实现

TI OpenMax IL 中各个组件都是通过调用 LCML 来实现的，实现的方式基本类似。主要都是实现了名称为 OMX_ComponentInit 的初始化函数，实现 OMX_COMPONENTTYPE 类型的结构体中的各个成员。各个组件其目录结构和文件结构也类似。

以 MP3 解码器的实现为例，在 audio/src/openmax_il/mp3_dec/src 目录中，主要包含以下文件。

OMX_Mp3Decoder.c：MP3 解码器组件实现

OMX_Mp3Dec_CompThread.c：MP3 解码器组件的线程循环

OMX_Mp3Dec_Utils.c：MP3 解码器的相关工具，调用 LCML 实现真正的 MP3 解码的功能

OMX_Mp3Decoder.c 中的 OMX_ComponentInit()函数负责组件的初始化，返回的内容再从参数中得到，这个函数的主要片断如下所示：

```
OMX_ERRORTYPE OMX_ComponentInit (OMX_HANDLETYPE hComp)
{
    OMX_ERRORTYPE eError = OMX_ErrorNone;
    OMX_COMPONENTTYPE *pHandle = (OMX_COMPONENTTYPE*) hComp;
    OMX_PARAM_PORTDEFINITIONTYPE *pPortDef_ip = NULL, *pPortDef_op = NULL;
    OMX_AUDIO_PARAM_PORTFORMATTYPE *pPortFormat = NULL;
    OMX_AUDIO_PARAM_MP3TYPE *mp3_ip = NULL;
    OMX_AUDIO_PARAM_PCMMODETYPE *mp3_op = NULL;
    MP3DEC_COMPONENT_PRIVATE *pComponentPrivate = NULL;
```

```
    MP3D_AUDIODEC_PORT_TYPE *pCompPort = NULL;
    MP3D_BUFFERLIST *pTemp = NULL;
    int i=0;
    MP3D_OMX_CONF_CHECK_CMD(pHandle,1,1);
/* ......省略，初始化 OMX_COMPONENTTYPE 类型的指针 pHandle */
    OMX_MALLOC_GENERIC(pHandle->pComponentPrivate,MP3DEC_COMPONENT_PRIVATE);
    pComponentPrivate = pHandle->pComponentPrivate;  /* 私有指针互相指向 */
    pComponentPrivate->pHandle = pHandle;
/* ......略，初始化似有数据指针 pComponentPrivate */
    /* 设置输入端口（OMX_PARAM_PORTDEFINITIONTYPE 类型）的默认值 */
    pPortDef_ip->nSize                 = sizeof(OMX_PARAM_PORTDEFINITIONTYPE);
    pPortDef_ip->nPortIndex            = MP3D_INPUT_PORT;
    pPortDef_ip->eDir                  = OMX_DirInput;
    pPortDef_ip->nBufferCountActual    = MP3D_NUM_INPUT_BUFFERS;
    pPortDef_ip->nBufferCountMin       = MP3D_NUM_INPUT_BUFFERS;
    pPortDef_ip->nBufferSize           = MP3D_INPUT_BUFFER_SIZE;
    pPortDef_ip->nBufferAlignment      = DSP_CACHE_ALIGNMENT;
    pPortDef_ip->bEnabled              = OMX_TRUE;
    pPortDef_ip->bPopulated            = OMX_FALSE;
    pPortDef_ip->eDomain               = OMX_PortDomainAudio;
    pPortDef_ip->format.audio.eEncoding = OMX_AUDIO_CodingMP3;
    pPortDef_ip->format.audio.cMIMEType = NULL;
    pPortDef_ip->format.audio.pNativeRender          = NULL;
    pPortDef_ip->format.audio.bFlagErrorConcealment = OMX_FALSE;
    /* 设置输出端口（OMX_PARAM_PORTDEFINITIONTYPE 类型）的默认值 */
    pPortDef_op->nSize                 = sizeof(OMX_PARAM_PORTDEFINITIONTYPE);
```

```
    pPortDef_op->nPortIndex          = MP3D_OUTPUT_PORT;
    pPortDef_op->eDir                = OMX_DirOutput;
    pPortDef_op->nBufferCountMin     = MP3D_NUM_OUTPUT_BUFFERS;
    pPortDef_op->nBufferCountActual  = MP3D_NUM_OUTPUT_BUFFERS;
    pPortDef_op->nBufferSize         = MP3D_OUTPUT_BUFFER_SIZE;
    pPortDef_op->nBufferAlignment    = DSP_CACHE_ALIGNMENT;
    pPortDef_op->bEnabled            = OMX_TRUE;
    pPortDef_op->bPopulated          = OMX_FALSE;
    pPortDef_op->eDomain             = OMX_PortDomainAudio;
    pPortDef_op->format.audio.eEncoding      = OMX_AUDIO_CodingPCM;
    pPortDef_op->format.audio.cMIMEType      = NULL;
    pPortDef_op->format.audio.pNativeRender = NULL;
    pPortDef_op->format.audio.bFlagErrorConcealment = OMX_FALSE;
/* .....省略，分配端口 */
    /* 设置输入端口的默认格式 */
    pPortFormat = pComponentPrivate->pCompPort[MP3D_INPUT_PORT]->pPortFormat;
    OMX_CONF_INIT_STRUCT(pPortFormat, OMX_AUDIO_PARAM_PORTFORMATTYPE);
    pPortFormat->nPortIndex        = MP3D_INPUT_PORT;
    pPortFormat->nIndex            = OMX_IndexParamAudioMp3;
    pPortFormat->eEncoding         = OMX_AUDIO_CodingMP3;
    /* 设置输出端口的默认格式 */
    pPortFormat = pComponentPrivate->pCompPort[MP3D_OUTPUT_PORT]->pPortFormat;
    OMX_CONF_INIT_STRUCT(pPortFormat, OMX_AUDIO_PARAM_PORTFORMATTYPE);
    pPortFormat->nPortIndex        = MP3D_OUTPUT_PORT;
    pPortFormat->nIndex            = OMX_IndexParamAudioPcm;
    pPortFormat->eEncoding         = OMX_AUDIO_CodingPCM;
```

```
/* .....省略部分内容 */

    eError = Mp3Dec_StartCompThread(pHandle);   // 启动 MP3 解码线程

/* .....省略部分内容 */

    return eError;

}
```

这个组件是 OpenMax 的标准实现方式，对外的接口的内容只有一个初始化函数。完成 OMX_COMPONENTTYPE 类型的初始化。输入端口的编号为 MP3D_INPUT_PORT（==0），类型为 OMX_PortDomainAudio，格式为 OMX_AUDIO_CodingMP3。输出端口的编号是 MP3D_OUTPUT_PORT（==1），类型为 OMX_PortDomainAudio，格式为 OMX_AUDIO_ CodingPCM。

OMX_Mp3Dec_CompThread.c 中定义了 MP3DEC_ComponentThread()函数，用于创建 MP3 解码的线程的执行函数。

OMX_Mp3Dec_Utils.c 中的 Mp3Dec_StartCompThread()函数，调用了 POSIX 的线程库建立 MP3 解码的线程，如下所示：

```
nRet = pthread_create (&(pComponentPrivate->ComponentThread), NULL,

                       MP3DEC_ComponentThread, pComponentPrivate);
```

Mp3Dec_StartCompThread()函数就是在组件初始化函数 OMX_ComponentInit()最后调用的内容。MP3 线程的开始并不表示解码过程开始，线程需要等待通过 pipe 机制获得命令和数据（cmdPipe 和 dataPipe），在适当的时候开始工作。这个 pipe 在 MP3 解码组件的 SendCommand 等实现写操作，在线程中读取其内容。

[书摘]Android 系统级深入开发之 OpenMax 的接口与实现

2011-03-01 16:44 | 1137 次阅读 | 来源：电子工业出版社 【已有 2 条评论】发表评论
关键词：图书,OpeMax,移动开发,Android | 作者：杨鹏飞 | 收藏这篇资讯

本文节选于电子工业出版社 北京博文视点资讯有限公司推出的《Android 系统级深入开发——移植与调试》一书第 18 章，这是一本全面介绍 Android 系统级开发的作品，作者韩超和梁泉以实际的开发经验为基础，以软件工程思想为指导，以移植和调试为重点，介绍了从 Android 开源工程到一个基于实际硬件产品中的主要工作，一方面让读者清晰把握各个子系统的架构，另一方面让读者把握移植这个开发核心环节的要点。

OpenMax IL 层的接口

OpenMax IL 层的接口定义由若干个头文件组成，这也是实现它需要实现的内容，它们的基本描述如下所示。

OMX_Types.h：OpenMax II 的数据类型定义

OMX_Core.h：OpenMax IL 核心的 API

OMX_Component.h：OpenMax IL 组件相关的 API

OMX_Audio.h：音频相关的常量和数据结构

OMX_IVCommon.h：图像和视频公共的常量和数据结构

OMX_Image.h：图像相关的常量和数据结构

OMX_Video.h：视频相关的常量和数据结构

OMX_Other.h：其他数据结构（包括 A/V 同步）

OMX_Index.h：OpenMax IL 定义的数据结构索引

OMX_ContentPipe.h：内容的管道定义

提示：OpenMax 标准只有头文件，没有标准的库，设置没有定义函数接口。对于实现者，需要实现的主要是包含函数指针的结构体。

其中，OMX_Component.h 中定义的 OMX_COMPONENTTYPE 结构体是 OpenMax IL 层的核心内容，表示一个组件，其内容如下所示：

```
typedef struct OMX_COMPONENTTYPE
{
    OMX_U32 nSize;                  /* 这个结构体的大小 */
    OMX_VERSIONTYPE nVersion;       /* 版本号 */
    OMX_PTR pComponentPrivate;      /* 这个组件的私有数据指针. */
    /* 调用者（IL client）设置的指针，用于保存它的私有数据，传回给所有的回调函数 */
    OMX_PTR pApplicationPrivate;
```

```
/* 以下的函数指针返回 OMX_core.h 中的对应内容 */
OMX_ERRORTYPE (*GetComponentVersion)(            /* 获得组件的版本*/
    OMX_IN  OMX_HANDLETYPE hComponent,
    OMX_OUT OMX_STRING pComponentName,
    OMX_OUT OMX_VERSIONTYPE* pComponentVersion,
    OMX_OUT OMX_VERSIONTYPE* pSpecVersion,
    OMX_OUT OMX_UUIDTYPE* pComponentUUID);
OMX_ERRORTYPE (*SendCommand)(                    /* 发送命令 */
    OMX_IN  OMX_HANDLETYPE hComponent,
    OMX_IN  OMX_COMMANDTYPE Cmd,
    OMX_IN  OMX_U32 nParam1,
    OMX_IN  OMX_PTR pCmdData);
OMX_ERRORTYPE (*GetParameter)(                   /* 获得参数 */
    OMX_IN  OMX_HANDLETYPE hComponent,
    OMX_IN  OMX_INDEXTYPE nParamIndex,
    OMX_INOUT OMX_PTR pComponentParameterStructure);
OMX_ERRORTYPE (*SetParameter)(                   /* 设置参数 */
    OMX_IN  OMX_HANDLETYPE hComponent,
    OMX_IN  OMX_INDEXTYPE nIndex,
    OMX_IN  OMX_PTR pComponentParameterStructure);
OMX_ERRORTYPE (*GetConfig)(                      /* 获得配置 */
    OMX_IN  OMX_HANDLETYPE hComponent,
    OMX_IN  OMX_INDEXTYPE nIndex,
    OMX_INOUT OMX_PTR pComponentConfigStructure);
OMX_ERRORTYPE (*SetConfig)(                      /* 设置配置 */
```

```
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_IN  OMX_PTR pComponentConfigStructure);
    OMX_ERRORTYPE (*GetExtensionIndex)(          /* 转换成 OMX 结构的索引 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_STRING cParameterName,
        OMX_OUT OMX_INDEXTYPE* pIndexType);
    OMX_ERRORTYPE (*GetState)(                   /* 获得组件当前的状态 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_OUT OMX_STATETYPE* pState);
    OMX_ERRORTYPE (*ComponentTunnelRequest)(     /* 用于连接到另一个组件*/
        OMX_IN  OMX_HANDLETYPE hComp,
        OMX_IN  OMX_U32 nPort,
        OMX_IN  OMX_HANDLETYPE hTunneledComp,
        OMX_IN  OMX_U32 nTunneledPort,
        OMX_INOUT  OMX_TUNNELSETUPTYPE* pTunnelSetup);
    OMX_ERRORTYPE (*UseBuffer)(                  /* 为某个端口使用 Buffer */
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN OMX_U32 nSizeBytes,
        OMX_IN OMX_U8* pBuffer);
    OMX_ERRORTYPE (*AllocateBuffer)(             /* 在某个端口分配 Buffer */
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE** ppBuffer,
```

```
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN OMX_U32 nSizeBytes);
    OMX_ERRORTYPE (*FreeBuffer)(                    /*将某个端口 Buffer 释放*/
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_U32 nPortIndex,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*EmptyThisBuffer)(               /* 让组件消耗这个 Buffer */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*FillThisBuffer)(                /* 让组件填充这个 Buffer */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*SetCallbacks)(                  /* 设置回调函数 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_CALLBACKTYPE* pCallbacks,
        OMX_IN  OMX_PTR pAppData);
    OMX_ERRORTYPE (*ComponentDeInit)(               /* 反初始化组件 */
        OMX_IN  OMX_HANDLETYPE hComponent);
    OMX_ERRORTYPE (*UseEGLImage)(
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN void* eglImage);
    OMX_ERRORTYPE (*ComponentRoleEnum)(
```

```
        OMX_IN OMX_HANDLETYPE hComponent,

 OMX_OUT OMX_U8 *cRole,

 OMX_IN OMX_U32 nIndex);

} OMX_COMPONENTTYPE;
```

OMX_COMPONENTTYPE 结构体实现后，其中的各个函数指针就是调用者可以使用的内容。各个函数指针和 OMX_core.h 中定义的内容相对应。

EmptyThisBuffer 和 FillThisBuffer 是驱动组件运行的基本的机制，前者表示让组件消耗缓冲区，表示对应组件输入的内容；后者表示让组件填充缓冲区，表示对应组件输出的内容。

UseBuffer，AllocateBuffer，FreeBuffer 为和端口相关的缓冲区管理函数，对于组件的端口有些可以自己分配缓冲区，有些可以使用外部的缓冲区，因此有不同的接口对其进行操作。

SendCommand 表示向组件发送控制类的命令。GetParameter，SetParameter，GetConfig，SetConfig 几个接口用于辅助的参数和配置的设置和获取。

ComponentTunnelRequest 用于组件之间的隧道化连接，其中需要制定两个组件及其相连的端口。

ComponentDeInit 用于组件的反初始化。

提示：OpenMax 函数的参数中，经常包含 OMX_IN 和 OMX_OUT 等宏，它们的实际内容为空，只是为了标记参数的方向是输入还是输出。

OMX_Component.h 中端口类型的定义为 OMX_PORTDOMAINTYPE 枚举类型，内容如下所示：

```
typedef enum OMX_PORTDOMAINTYPE {
    OMX_PortDomainAudio,      /* 音频类型端口 */
    OMX_PortDomainVideo,      /* 视频类型端口 */
    OMX_PortDomainImage,      /* 图像类型端口 */
    OMX_PortDomainOther,      /* 其他类型端口 */
    OMX_PortDomainKhronosExtensions = 0x6F000000,
    OMX_PortDomainVendorStartUnused = 0x7F000000
    OMX_PortDomainMax = 0x7fffffff
} OMX_PORTDOMAINTYPE;
```

音频类型，视频类型，图像类型，其他类型是 OpenMax IL 层此所定义的四种端口的类型。

端口具体内容的定义使用 OMX_PARAM_PORTDEFINITIONTYPE 类（也在 OMX_Component.h 中定义）来表示，其内容如下所示：

```
typedef struct OMX_PARAM_PORTDEFINITIONTYPE {
    OMX_U32 nSize; /* 结构体大小 */
    OMX_VERSIONTYPE nVersion; /* 版本*/
    OMX_U32 nPortIndex; /* 端口号 */
    OMX_DIRTYPE eDir; /* 端口的方向 */
    OMX_U32 nBufferCountActual; /* 为这个端口实际分配的 Buffer 的数目 */
    OMX_U32 nBufferCountMin; /* 这个端口最小 Buffer 的数目*/
    OMX_U32 nBufferSize; /* 缓冲区的字节数 */
    OMX_BOOL bEnabled; /* 是否使能 */
    OMX_BOOL bPopulated; /* 是否在填充 */
    OMX_PORTDOMAINTYPE eDomain; /* 端口的类型 */
    union { /* 端口实际的内容，由类型确定具体结构 */
        OMX_AUDIO_PORTDEFINITIONTYPE audio;
        OMX_VIDEO_PORTDEFINITIONTYPE video;
        OMX_IMAGE_PORTDEFINITIONTYPE image;
        OMX_OTHER_PORTDEFINITIONTYPE other;
    } format;
    OMX_BOOL bBuffersContiguous;
    OMX_U32 nBufferAlignment;
} OMX_PARAM_PORTDEFINITIONTYPE;
```

对于一个端口，其重点的内容如下。

端口的方向（OMX_DIRTYPE）：包含 OMX_DirInput（输入）和 OMX_DirOutput（输出）两种

端口分配的缓冲区数目和最小缓冲区数目

端口的类型（OMX_PORTDOMAINTYPE）：可以是四种类型

端口格式的数据结构：使用 format 联合体来表示，具体由四种不同类型来表示，与端口的类型相对应

OMX_AUDIO_PORTDEFINITIONTYPE ， OMX_VIDEO_PORTDEFINITIONTYPE ， OMX_IMAGE_PORTDEFINITIONTYPE 和 OMX_OTHER_PORTDEFINITIONTYPE 等几个具体的格式类型，分别在 OMX_Audio.h，OMX_Video.h，OMX_Image.h 和 OMX_Other.h 这四个头文件中定义。

OMX_BUFFERHEADERTYPE 是在 OMX_Core.h 中定义的，表示一个缓冲区的头部结构。

OMX_Core.h 中定义的枚举类型 OMX_STATETYPE 命令表示 OpenMax 的状态机，内容如下所示：

```
typedef enum OMX_STATETYPE
{
    OMX_StateInvalid,              /* 组件监测到内部的数据结构被破坏 */
    OMX_StateLoaded,               /* 组件被加载但是没有完成初始化 */
    OMX_StateIdle,                 /* 组件初始化完成，准备开始 */
    OMX_StateExecuting,            /* 组件接受了开始命令，正在树立数据 */
    OMX_StatePause,                /* 组件接受暂停命令*/
    OMX_StateWaitForResources,     /* 组件正在等待资源 */
    OMX_StateKhronosExtensions = 0x6F000000, /* 保留 */
    OMX_StateVendorStartUnused = 0x7F000000, /* 保留 */
    OMX_StateMax = 0X7FFFFFFF
} OMX_STATETYPE;
```

OpenMax 组件的状态机可以由外部的命令改变，也可以由内部发生的情况改变。OpenMax IL 组件的状态机的迁移关系如图 18-6 所示。

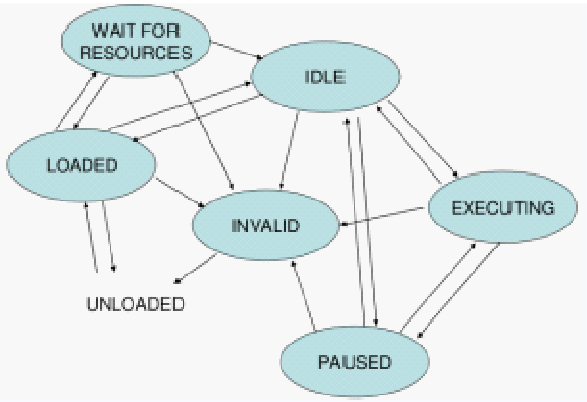

图 18-6 OpenMax IL 组件的状态机的迁移关系

OMX_Core.h 中定义的枚举类型 OMX_COMMANDTYPE 表示对组件的命令类型，内容如下所示：

```
typedef enum OMX_COMMANDTYPE
{
    OMX_CommandStateSet,            /* 改变状态机器 */
    OMX_CommandFlush,               /* 刷新数据队列 */
    OMX_CommandPortDisable,         /* 禁止端口 */
    OMX_CommandPortEnable,          /* 使能端口 */
    OMX_CommandMarkBuffer,          /* 标记组件或 Buffer 用于观察 */
    OMX_CommandKhronosExtensions = 0x6F000000, /* 保留 */
    OMX_CommandVendorStartUnused = 0x7F000000, /* 保留 */
    OMX_CommandMax = 0X7FFFFFFF
} OMX_COMMANDTYPE;
```

OMX_COMMANDTYPE 类型在 SendCommand 调用中作为参数被使用，其中 OMX_CommandStateSet 就是改变状态机的命令。

OpenMax 在 Android 上的实现

分类： Android 研究 多媒体研究 2011-05-15 23:37 1575 人阅读 评论(2) 收藏 举报

摘要：本文简要介绍了 OpenMax 的集成层，并阐述了其在 Android 上的实现和运行过程。

关键字：OMX, 多媒体框架, IL, Android, Stagefright

1、OpenMax 集成层介绍

OpenMax 是一个多媒体应用程序的框架标准。它自上而下分为三层，Application Layer, Integration Layer 和 Development Layer。应用层规定了应用程序和多媒体中间层的标准接口，使应用程序的移植性更好。集成层定义了多

媒体组件的接口，使得多媒体框架能以一种统一的方式访问多媒体 Codec 和组件，以便在嵌入式流媒体框架中快速集成加速编解码器。。开发层为 Codec 厂商和硬件厂商提供了一套 API，使开发更加便捷。

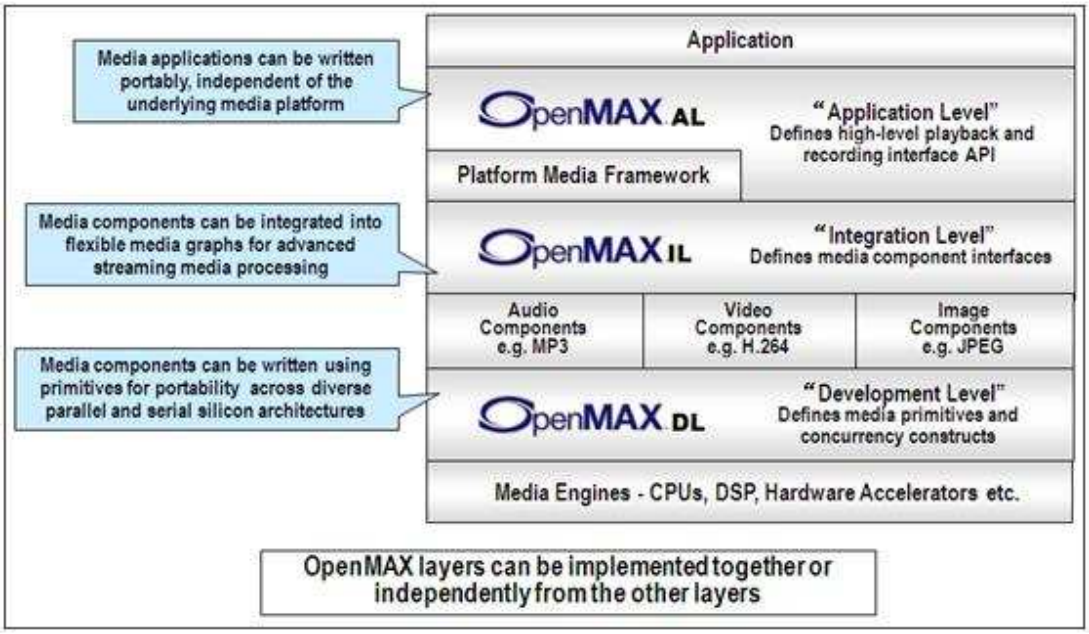

图 1 OpenMax 的分层结构

OMX 集成层由 Client、Core、Component 和 Port 组成，Client 通过 Core 得到对应 Component 的 Handle，而后通过命令直接和 Component 进行交互。每个 Component 至少有一个 Port 进行数据交互，如 Decoder 有一个输入 Port 接收码流，一个输出 Port 输出 YUV 序列。Component 内部可能通过消息处理机制完成 Client 要求的任务。

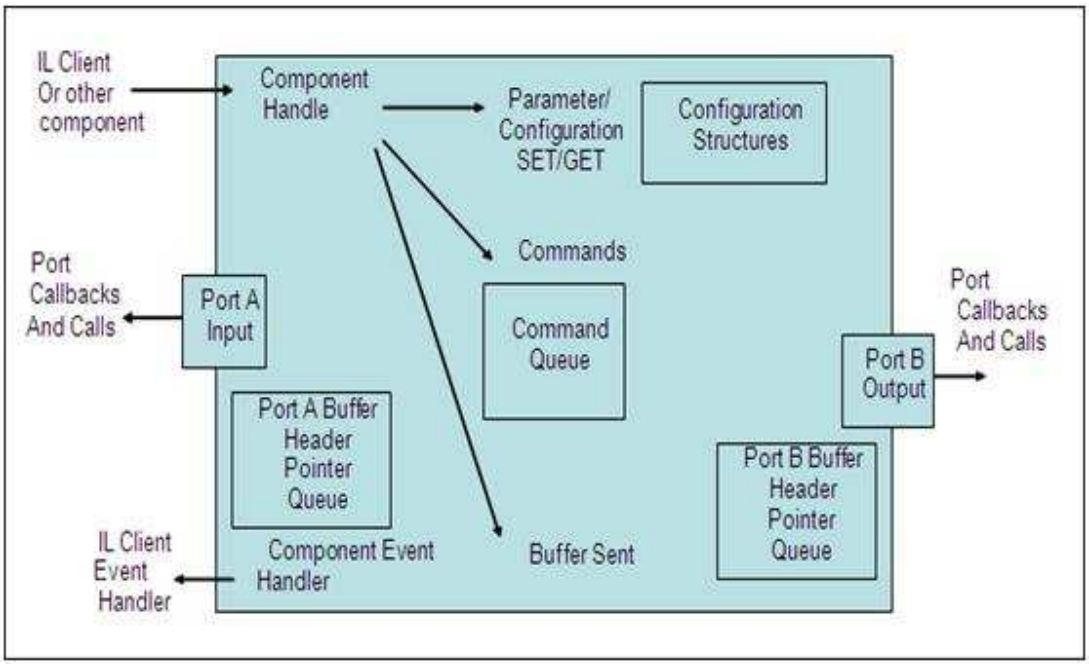

图 2 OMX IL 层的组成

在 Android 中，OpenMax IL 层，通常可以用于多媒体引擎的插件，Android2.2 的多媒体引擎 StageFright 都可以使用 OpenMax 作为插件，主要用于编解码（Codec）处理。

在 Android 的框架层，定义了由 Android 封装的 OpenMax 接口，和标准的接口概念基本相同，使用 C++类型的接口，并且使用了 Android 的 Binder IPC 机制实现了函数远程调用。Android 封装 OpenMax 的接口被 StageFright 使用。

解码 Component 通过输入 Port 和输出 Port 来进行交互，可以通过和 OMXCodec 共享 buffer 来进行编解码。

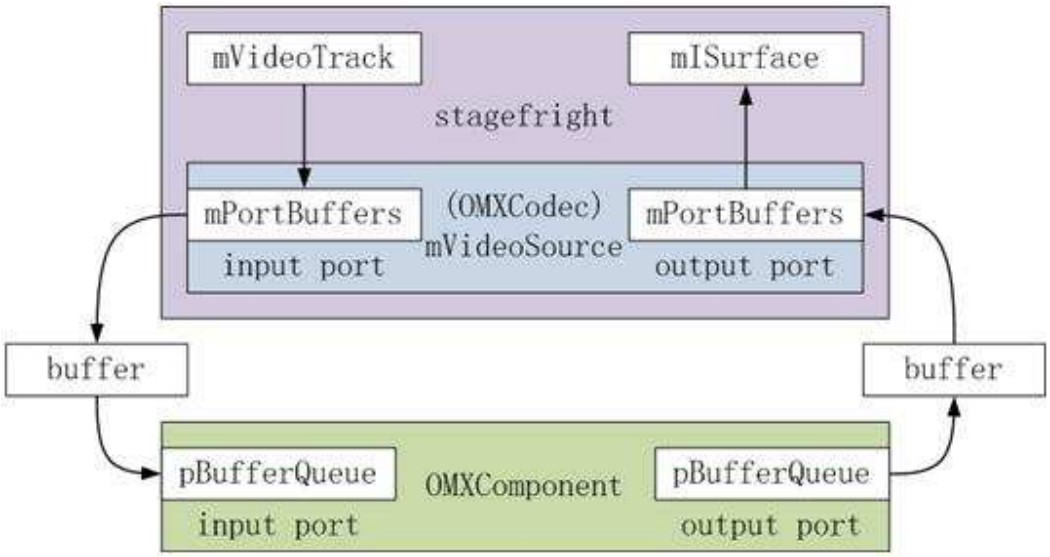

图 3 StageFright 的 OMX 结构

2、OMX 相关的类关系

StageFright 的核心播放器 AwesomePlayer 中包含了 sp<MediaSource>型的 mVideoSource 指针，初始化时指向 OMXCodec 的实际对象。OMXCodec 使用了 Binder 机制，实现了远程函数调用，且函数调用模式于本地函数无异，其中 IOMX 作为接口类定义了 OMX 的大部分接口函数。OMX 的具体实现时，OMXMaster 类用于管理 OMX 的插件，OMXNodeInstance 类代表 OMX 的具体实例，完成和 Component 的调用和交互，内部类 CallbackDispatcher 是一个主动类，它用于调度处理回调函数传回的消息。OMXNodeInstance 和 CallbackDispatcher 一一对应，协同工作，完成不同实例的消息处理。

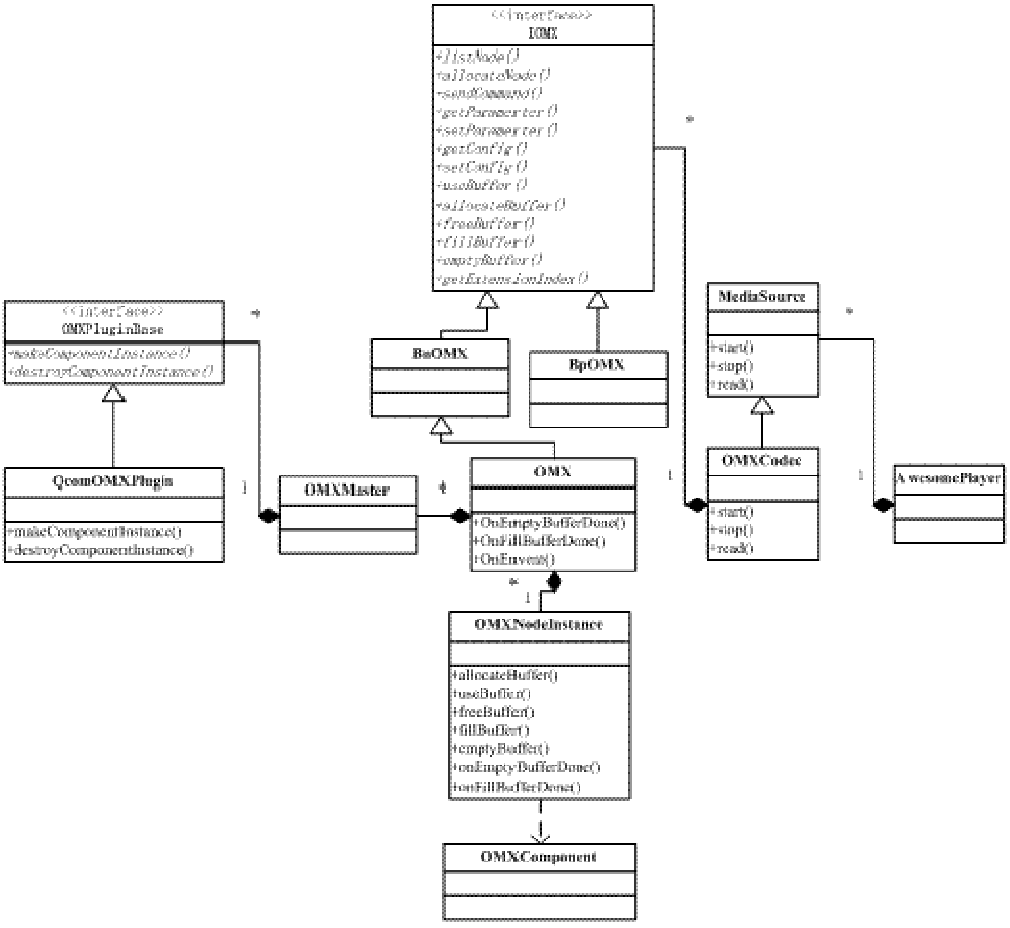

图 4 OMX 相关类关系图

OMXNodeInstance 类中最重要的几个函数为：

allocateBuffer: Client 通过调用此函数让 Component 分配 Buffer。

useBuffer: Client 通过此函数将已分配好的 Buffer 传给 Component，让其使用。

freeBuffer: Client 通过调用此函数让 Component 释放 allocateBuffer()分配的 Buffer。

fillBuffer: Client 通过调用此函数传递空的 Buffer 给 Component，让其将处理好的数据填入其中。此函数会调用 OMX 标准接口 OMX_FillThisBuffer()。

emptyBuffer: Client 通过调用此函数传递输入 Buffer 给 Component，让其读取其中的数据进行编解码等处理。此函数会调用 OMX 标准接口 OMX_ EmptyThisBuffer ()。

OnEmptyBufferDone: Component 完成对输入 buffer 的读取后，调用此回调函数，向 Client 发送 EmptyBufferDone 消息。

OnFillBufferDone: Component 完成相应处理将输出数据填入输出 Buffer 后，调用此回调函数，向 Client 发送 FillBufferDone 消息。

以 Decoder 为例，说明 OMX Codec 执行解码的过程，

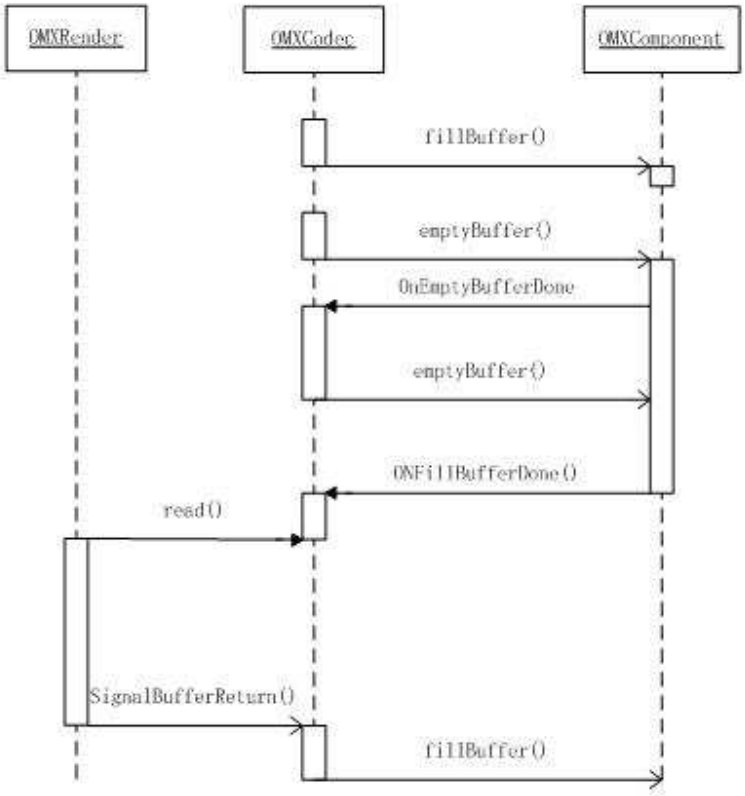

图 5 OMX Codec 解码序列图

OMXCodec 使用 emptyBuffer()函数(IL 层中为 OMX_EmptyThisBuffer()传递未解码的 buffer 给 component,component 收到该命令后会读取 input port buffer 中的数据,将其组装成帧进行解码,读取 buffer 中的数据完成后会调用 EmptyBufferDone 通知 OMXCodec。

Compoment 使用 EmptyBufferDone 消息通知 OMXCodec 已完成 input buffer 的读取, 具体的实现是通过调用回调函数 OnEmptyBufferDone()实现的。OMXCodec 收到该命令后会通过 mVideoTrack 读取新的视频码流到 input port 的 buffer 中,并调用 OMX_EmptyThisBuffer 通知 component。

OMXCodec 使用 OMX_FillThisBuffer 传递空的 bffer 给 component 用于存储解码后的帧，Component 收到该命令后将解码好的帧数据复制到该 buffer 上,然后调用 FillBufferDone 通知 OMXCodec。

Compoment 使用 FillBufferDone 通知 OMXCodec 已完成 output port buffer 的填充，具体的实现是通过调用回调函数 OnFillBufferDone()实现的。OMXCodec 收到该命令后将解码好的帧存入可显示队列中，AwesomePlayer 调用 OMXCodec::read()函数读出可显示队列的对头送给 Renderer 完成颜色转换等操作再传递给 mISurface 进行图像绘制，同时 Render 调用 release()函数,其中的 SignalBufferDone()会用 OMX_FillThisBuffer 通知 component 有空的 buffer 可填充。