



ZLMediaKit 源码分析

ADM 官方 QQ 群福利版

颜贤时 著

codemi.net

目录

目录.....	1
第 1 章 Linux 操作系统简介.....	3
1.1. Linux 的起源和发展.....	3
1.2. 对读者的基本要求.....	3
1.3. 术语和约定.....	3
第 2 章 Linux 应用编程基础.....	5
2.1. 基本原理.....	5
2.2. C 语言标准.....	7
2.3. C++语言标准.....	7
2.4. 运行时库.....	8
2.5. POSIX 标准.....	8
2.6. 代码编辑器.....	8
2.7. MSYS2 环境.....	9
2.8. 构建工具链.....	9
2.9. gdb 调试器.....	10
2.10. GNU make.....	12
2.11. CMake.....	13
2.12. valgrind.....	14
2.13. Git.....	14
2.14. MySQL.....	14
2.15. Redis.....	14
2.16. nginx.....	15
2.17. HAProxy.....	15
2.18. libuv.....	15
2.19. st.....	16
2.20. ProtoBuf.....	16
2.21. MQTT.....	16
2.22. Docker.....	16
第 3 章 知名开源流媒体服务器分析.....	18
3.1. Live555.....	18
3.2. GStreamer.....	22
3.3. DSS.....	24
3.4. SRS.....	25
3.5. ZLMediaKit.....	26
3.6. xsnode.....	27
3.7. Others.....	27
第 4 章 Linux 版直播服务器需求分析.....	28
4.1. 目标应用场景.....	28
4.2. 运行时硬件环境约束.....	28
4.3. 运行时软件环境约束.....	28

4.4. 功能要求.....	28
4.5. 性能要求.....	28
4.6. 可用性要求.....	28
4.7. 方案选择.....	29
第5章 ZLMediaKit 源码分析.....	30
5.1. 源码目录.....	30
5.2. 源码构建.....	74
5.3. 配置方法.....	75
5.4. 运行方法.....	76
5.5. 顶层设计.....	77
5.6. 底层原理.....	81
5.7. 媒体接入流程.....	95
5.8. 媒体缓冲流程.....	96
5.9. 媒体转发流程.....	96
5.10. 媒体推送流程.....	96
5.11. 媒体播放流程.....	96
5.12. 媒体录制流程.....	96
5.13. 系统集成方案.....	96
第6章 关于.....	97
6.1. 关于作者.....	97

第 1 章 Linux 操作系统简介

1.1. Linux 的起源和发展

1965 年，美国电话电报公司（AT&T）贝尔实验室、美国麻省理工学院和通用电气联合开始了 Multics 工程计划。由于 Multics 工程计划所追求的目标太庞大、太复杂，以至于它的开发人员都不知道要做什么样子，最终以失败收场。AT&T 贝尔实验室的研究人员 Ken Thompson 和 Dennis Ritchie 两人都参与了该计划。

1969 年，Ken Thompson 和 Dennis Ritchie 吸取了 Multics 工程的经验教训，在 DEC 的 PDP-7 计算机上用汇编语言实现了一种分时操作系统，该系统于 1970 被正式命名为 UNIX。此后，UNIX 在商业上蓬勃发展，并衍生出了许多分支，比较知名的有 FreeBSD、Open BSD、SUN Solaris、IBM AIX、HP-UX、UNIX V6 等。

1987 年，荷兰教授 Andrew S. Tanenbaum 发布了一个用于 UNIX 操作系统教学的 Minix 模型操作系统。

1991 年，芬兰赫尔辛基大学的一位名叫 Linus Torvalds 的研究生，受到 Minix 和 UNIX 的启发，决定在自己的购买的 PC 上开发一个自己的操作系统。Linus 将自己开发的这个操作系统命名为 Linux，并将其源代码上传到互联网。

1994 年，1.0 版本的 Linux 发布。

严格来说，Linux 只定义了一个操作系统内核，这个内核由 kernel.org 负责维护。不同的组织在这个内核的基础上开发了许多不同应用软件，并打包发布成自己的“发行版”。二十多年来，比较流行的 Linux 发行版有 Red Hat、Fedora、Red Flag、openSUSE、Debian、CentOS 和 Ubuntu 等。

截至 2020 年 12 月，Linux 在服务器市场占据了 80% 左右的市场份额，在嵌入式设备领域占据了超过 50% 的市场份额，但是在桌面市场仅占据了 3% 左右的市场份额。在 PC 桌面操作系统领域，微软公司的 Windows 仍然占据着主导地位。

1.2. 对读者的基本要求

作者要求读者至少学习过 C++ 语言，能编写简单的控制台应用程序，并且还要熟悉常用的 Linux shell 命令，如软件包管理和文件操作等。

如果读者没有 Linux 后台服务器编程经验，可按照第 2 章的学习路线快速入门。

1.3. 术语和约定

实时 (Realtime) 在本教程中是指音视频数据从采集到呈现给观看者的延迟时间稳定在 500ms 以内的特性。

多媒体 (Multimedia) 是多种媒体的综合，一般包括文本，声音和图像等多种媒体形式。

音视频 (Audio Video) 是声音和图像的复合体，是多媒体概念的子集。

直播 (Live Streaming) 一般指的是播放正在发生的现场画面，相对于点播（或录播）时效性更强。我们并不严格区分直播和点播的功能差别，因为两者仅仅是时效性的差异。用

户使用直播客户端时，可以通过时移操作播放较长时间以前发生的历史音视频片段。作者开发的直播服务器企业版也具备历史媒体点播的功能。

流媒体 (Streaming Media) 指以流方式在网络中传送音频、视频和多媒体文件的媒体形式。

流媒体服务器 (Streaming Media Server) 主要功能是以流式协议 (RTP/RTSP、MMS、RTMP 等) 将视频文件传输到客户端，供用户在线观看；也可从视频采集、压缩软件接收实时视频流，再以流式协议直播给客户端。

直播服务器 (Live Streaming Server) 是指通常部署在云端，负责接收某直播客户端进程推送的音视频直播数据，并将其转发给其它要观看此直播的客户端进程的应用程序。通常情况下，我们可以用术语“**流媒体服务器**”来代替直播服务器。此教程采用术语“**直播服务器**”是为了强调这是一种非常适合直播应用场景的流媒体服务器。

第 2 章 Linux 应用编程基础

本章是为无 Linux 环境应用编程经验的读者快速入门准备的，对于已经熟悉 Linux 开发的读者可以跳过本章，直接阅读下一章。

作者编写本章的目的不是要提供一份详细 Linux 应用开发教程，而是要提供一份可以让 Linux 新手快速上道的学习路线图。新手按照本章的学习路线图，循序渐进，可少走弯路，快速成为 Linux 后台应用开发的行家里手。读者阅读过程中，如果遇到不懂的概念可以通过互联网寻找更详细的资料来快速学习。

本章 1-13 节属于新手必修内容，14-22 节属于新手选修内容。

2.1. 基本原理

在 PC 领域，自 80386 CPU 以后的所有 x86 架构（包括 x86_64）的 CPU 都支持保护模式。CPU 在保护模式下，指令的执行分为 4 个特权级，Windows 和 Linux 操作系统只使用了其中两个特权级。这两个特权级可以理解为内核态和用户态。在内核态下，CPU 可以执行所有指令，而用户态会受到一些限制，仅能运行非特权指令。操作系统内核就是工作在 CPU 的内核态下，因此它能接管整个 CPU 能访问到的所有资源（内存、总线、端口），起到统一调度管理的作用。一般的应用开发程序员写的程序都是工作在用户态的。

我们把工作在用户态的程序所处的内存空间称之为用户空间，把工作在内核态的程序所处的内存空间称之为内核空间。这两个空间是隔离的，处于用户空间的应用程序如果想访问内核空间的操作系统内核提供的功能（或称之为服务），就需要通过系统调用指令来实现。比如，Windows 平台通过 `int 0x2e` 指令来进行系统调用，而 Linux 平台通过 `int 0x80` 指令来实现。

系统调用（System Call）是操作系统提供的服务，是应用程序与内核通信的接口。Linux 提供的系统调用包含的内容有：文件操作、进程控制、系统控制、内存管理、网络管理、socket 套接字、进程间通信、用户管理等。

相对于普通的函数调用来说，系统调用的性能消耗也是巨大的。所以在追求极致性能的程序中，开发者都会尽力减少系统调用。

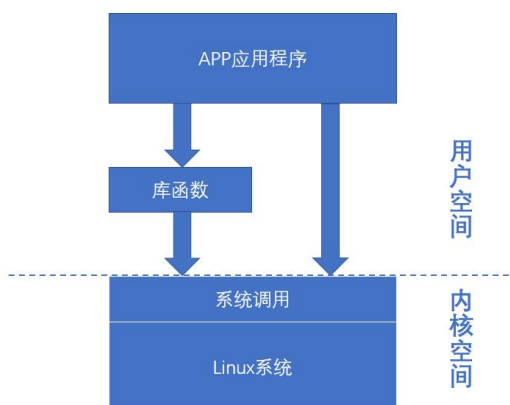


表 2-1 Linux 应用基本原理

Linux 环境下，使用的 C 库一般都是 glibc，它封装了几乎所有的系统调用。追求可移

植性的应用程序应该通过 glibc 库提供的 C 语言风格的 API 来间接地调用 Linux 系统内核中的功能模块。

Linux 内核提供了管理一台计算机的所有资源必备的功能机制，Linux 内核基本架构如下图所示：

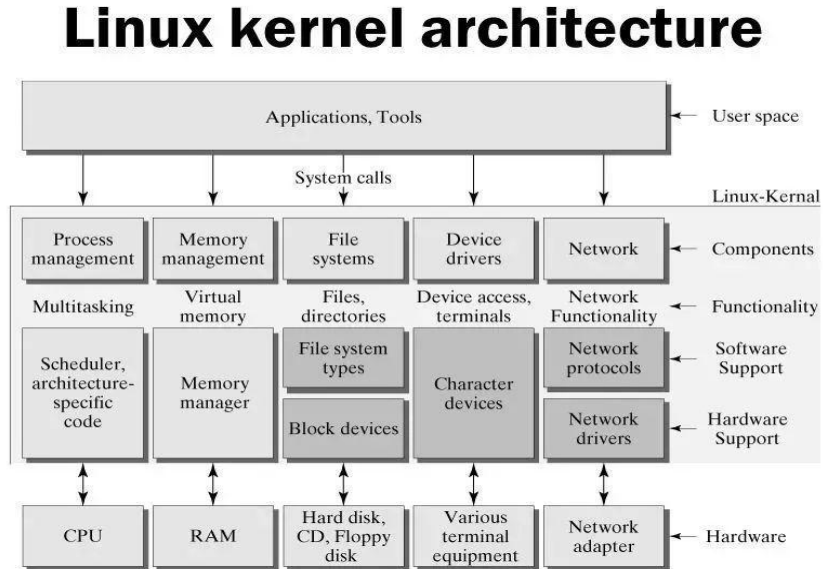


表 2-2 Linux 内核架构

从上图可以看出，Linux 内核的功能非常丰富，涵盖了计算机资源管理的方方面面。Linux 原生应用开发的本质就是内核提供的功能基础上，借助 glibc 库和其它软件开发库，编写运行在用户空间的具备特定功能的应用程序。

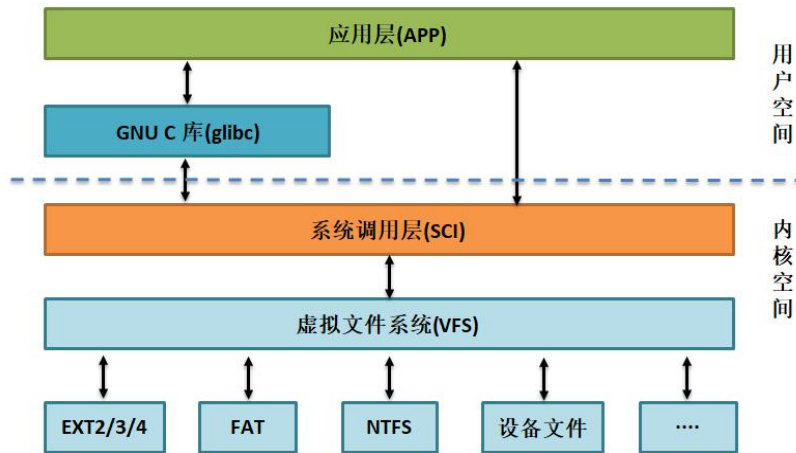


表 2-1 Linux 应用与文件系统

2.2. C 语言标准

C 语言诞生于美国的贝尔实验室，由 D.M.Ritchie 以 B 语言为基础发展而来，在它的主体设计完成后，Thompson 和 Ritchie 用它完全重写了 UNIX，且随着 UNIX 的发展，C 语言也得到了不断的完善。

为了利于 C 语言的全面推广，许多专家学者和硬件厂商联合组成了 C 语言标准委员会，并在之后的 1989 年，诞生了第一个完备的 C 标准，简称“C89”，也就是“ANSI C”。

截至 2020 年 12 月，已正式发布的 C 语言标准有 C89、C90、C94、C95、C99、C11 和 C17。

2.3. C++语言标准

20 世纪 70 年代中期，出生于丹麦的 Bjarne Stroustrup 在英国剑桥大学计算机中心工作。他使用过 Simula 和 ALGOL，接触过 C。他对 Simula 的类体系感受颇深，对 ALGOL 的结构也很有研究，深知运行效率的意义。既要编程简单、正确可靠，又要运行高效、可移植，是 Bjarne Stroustrup 的初衷。以 C 为背景，以 Simula 思想为基础，正好符合他的设想。

1979 年，Bjarne Stroustrup 在 AT&T 贝尔实验室，从事将 C 改良为“C with classes”的工作。

1983 年该语言被正式命名为 C++。

自从 C++ 被发明以来，它经历了 3 次主要的修订，每一次修订都为 C++ 增加了新的特征并作了一些修改。第一次修订是在 1985 年，第二次修订是在 1990 年，而第三次修订发生在 C++ 的标准化过程中。

在 20 世纪 90 年代早期，人们开始为 C++ 建立一个标准，并成立了一个 ANSI 和 ISO 联合标准化委员会。该委员会在 1994 年 1 月 25 日提出了第一个标准化草案。在这个草案中，委员会在保持 Stroustrup 最初定义的所有特征的同时，还增加了一些新的特征。

在完成 C++ 标准化的第一个草案后不久，发生了一件事情使得 C++ 标准被极大地扩展了：Alexander Stepanov 创建了标准模板库 (Standard Template Library, STL)。STL 不仅功能强大，同时非常优雅，然而，它也是非常庞大的。在通过了第一个草案之后，委员会投票并通过了将 STL 包含到 C++ 标准中的提议。STL 对 C++ 的扩展超出了 C++ 的最初定义范围。

C++ 标准化委员会于 1997 年 11 月 14 日通过了该标准的最终草案，1998 年，C++ 的 ANSI/ISO 标准被投入使用。通常，这个版本的 C++ 被认为是标准 C++。所有的主流 C++ 编译器都支持这个版本的 C++ 标准。

截至 2020 年 12 月，已正式发布的 C++ 语言标准有 C++ 98、C++ 03、C++ 11、C++ 14 和 C++ 17。

2.4. 运行时库

运行时库指的是程序运行的时候所依赖的库文件。与运行时概念相对应的还有设计时、编译时等。Linux 系统上最基本的应用层运行时库就是 `glibc` 库。

对现实世界的复杂应用程序开发，我们仅会使用 `glibc` 库是不够的，还需要熟悉一些可以用来提高开发效率的其它运行时库。比如，当我们在开发数据库类应用时，我们需要使用具体的数据库引擎（比如：`MySQL`、`sqlite`、`postgreSQL`、`redis` 等）提供的运行时库，这些运行时库通常包含在数据库引擎提供的软件开发包（SDK）中。软件开发包除了包含基本的运行时库外，通常还包含了调试符号、开发辅助工具、API 头文件、说明文档等，

运行时库分为静态库（`.a`）和动态库（`.so`）。

我们可以通过环境变量 `LD_LIBRARY_PATH` 来指定动态库的搜索路径。

2.5. POSIX 标准

20 世纪 80 年代中期，UNIX 系统发展出了许多分支，不同的厂商试图通过加入新的、往往不兼容的特性来使它们的程序与众不同。这给开发者造成很大的困扰，他们需要针对不同的 UNIX 平台的 API 分别编写功能相同的代码。

为了阻止这种趋势，IEEE(电气和电子工程师协会)开始制定 UNIX 的 API 标准。Richard Stallman 将这套标准命名为“POSIX”。POSIX 是“Portable Operating System Interface of UNIX”的缩写。

POSIX 标准的诞生是为了统一一个操作系统的接口，方便开发者开发程序，写出可移植的代码程序。

POSIX 标准涵盖了很多方面，比如 UNIX 系统调用的 C 语言接口、shell 程序和工具、线程及网络编程。

不仅 UNIX 系统支持 POSIX 标准，在某些非 UNIX 基因的操作系统上也可以支持 POSIX 标准，比如 DEC 的 `Open VMS` 和微软的 `Windows NT`。

Linux 属于类 UNIX 的操作系统，当然也支持 POSIX 标准。

如果我们在 Linux 后台服务器编程中坚持使用符合 POSIX 标准的 C 语言 API，那么我们编写的源代码就能更容易地移植到任何支持 POSIX 标准的操作系统上。

2.6. 代码编辑器

在 Linux 应用开发中，常用的代码编辑器有 `vim`、`Emacs`、`Source Insight`、`Visual Studio Code` 等。

对于单个文本文件的编辑（比如 `shell` 脚本），用 `vim` 是足以应付的。但是如果对于复杂的大型项目，仍然还坚持用这种简单原始的编辑器，有网友认为这是一种反智行为或者可能是一种故弄玄虚的炫耀行为。对于新手，作者建议每个都尝试一下，雨露均沾，最后，还是要看哪个用起来让自己觉得更舒服就用哪个，以提高开发效率为终极目标。

为了顺利学习本章内容，作者建议读者打开 `vim`，编写一个 `hello.c` 文件，仅用 `printf` 输出一行“`hello,world.`”，本章后面的小节会用到这个源文件。

hello.c 文件内容如下：

```
#include <stdio.h>
void main()
{
    printf("hello,world.\n");
    int *p = 0;
    *p = 0;
    printf("goodbye,world.\n");
}
```

2.7. MSYS2 环境

MSYS2 是运行在 Windows 上的一套应用软件。

MSYS2 基于 Cygwin 和 MinGW-w64 构建。

它提供了类似 Linux 下的 bash 的 shell 环境。它让我们在 Windows 平台上就能实现大多数类型的 Linux 应用程序的开发。

如果读者要在 Windows 平台上构建 FFmpeg，那么还是要熟悉 MSYS2 环境。

MSYS2 的具体使用方法请参照 msys2 官网的文档：<https://www.msys2.org/>

尽管 MSYS2 能让我们在 Windows 环境下开发出能在 Linux 上编译运行的流媒体服务器，但作者还是建议读者至少安装一台 Ubuntu 的虚拟机，在真正的 Linux 环境下完成代码构建和调试。

读者也可以购买一台安装了 Linux 系统的云服务器，通过 ssh 工具（例如 MobaXterm）在远程 Linux 服务器上进行编程实践。采用云服务器的好处是有外网 IP，便于进行模拟生产环境的测试。

2.8. 构建工具链

用代码编辑器编写好了代码，用什么工具来编译链接呢？这就需要用到构建工具链了。构建工具链是指一套针对特定硬件和软件平台编译、链接、调试、打包的相关工具的集合。

在 Linux 服务器编程中，最常用的工具链有 GCC、Clang 等。如果读者是为特定厂商的开发板（比如某些厂商的 ARM 设备）开发应用程序，还可能需要用到原厂提供的专用交叉编译工具链。不管什么 C/C++ 构建工具链，其构建参数和 GCC 都大同小异。因此，我们只要熟练掌握了 GCC 工具链，那么遇到其它类似的工具链时都可以快速上手。

假设读者的 Linux 系统上已经安装了 gcc。用 GCC 构建一个 hello.c 的命令行如下：

```
gcc hello.c -g
```

此命令的参数 -g 是为了将调试信息包含在生成的可执行文件中。

此行命令如果编译成功，会在当前目录下输出 a.out，然后继续输入如下命令：

```
./a.out
```

不出意外的话，屏幕会显示如下内容：

```
hello,world.
Segmentation fault (core dumped)
```

可以看出，并没有输出“goodbye,world.”，并且还有一行“段错误”的提示，告诉我们程序运行时的时候 core dumped 了，运行时崩溃了。

这是因为作者故意写了个访问空指针的 BUG。

下一章，我们讲解如何使用 gdb 调试本节构建生成的 a.out，如何通过 core dump 文件找出可执行程序运行崩溃时的调用栈。

2.9. gdb 调试器

我们可以用 gdb 命令来调试分析可执行程序中的缺陷。假如我们要调试上一节构建的 a.out 可以在 shell 中输入以下命令：

```
gdb a.out
```

执行上述命令后，进入 gdb 调试器的内部命令解释器的人机交互界面。在这种交互环境下，我们可以输入各种 gdb 调试命令来完成调试分析可执行程序的工作。

常用的 gdb 调试命令如下：

命令	缩写	功能、用法、示例
quit	q	功能：退出当前 gdb 调试环境。 用法：q 示例：q
help	h	功能：显示各种 gdb 命令的解释说明。 用法：h [命令类别] 示例 1：h 示例 2：n breakpoints 示例 3：h bt
file		功能：加载被调试的可执行程序文件。当启动 gdb 调试器时，如果没有带上被调试的可执行程序文件名参数，那么可用此命令来动态加载被调试的可执行文件。也可以用此命令改变当前要调试文件。 用法：file [被调试的可执行程序文件名] 示例：file a.out
run	r	功能：运行被调试的可执行程序文件。 用法：r 示例：r
list	l	功能：列出指定位置的源代码。 用法：l [LINENUM FILE:LINENUM FUNC FILE:FUNC *ADDR] 示例：l main
info	i	功能：显示各种信息，输 i 查看详情。 用法：i [subcommand] 示例 1：i threads 示例 2：i stack 示例 3：i source 示例 4：i sources
step	s	功能：单步执行下一行源码，如果下一行是函数，则 step into it。 用法：s

		示例: s
next	n	功能: 执行下一行源码, 无论下一行是什么, 都 step over it。 用法: n 示例: n
stepi	si	功能: step into 下一条机器指令。参数 N 表示此命令的执行次数。 用法: si [N] 示例 1: si 示例 2: si 2
nexti	ni	功能: step over 下一条机器指令。参数 N 表示此命令的执行次数。 用法: ni [N] 示例 1: ni 示例 2: ni 2
print	p	功能: 打印表达式的值。 用法: p EXP 示例: p 1+1
break	b	功能: 在指定位置设置断点。 用法: b [PROBE_MODIFIER][LOCATION][thread THREADNUM][if CONDITION] 示例: b 2
continue	c	功能: 继续运行, 忽略后续的 N 个断点。 用法: c [N] 示例: c
watch		功能: 监视表达式, 当其值发生变化时触发断点。 用法: watch [-l -location] EXPRESSION 示例: watch p
backtrace	bt	功能: 打印全部或由参数 COUNT 指定的帧数的函数调用栈帧回溯。 用法: bt [COUNT] 示例: bt
...		

表 2-2 gdb 调试命令描述表

上一节的程序 `a.out` 运行时出现了 `core dump`, 这是 Linux 系统的一种程序崩溃转储机制, 它在进程发生未处理异常时, 自动将内存中的进程转储到磁盘的 `core dump` 文件中。在作者使用的 Ubuntu 发行版中, `core dump` 文件默认存放在 `a.out` 进程崩溃时的当前工作目录下, 文件名默认为 `core`。

在 Linux shell 环境下 (非 gdb 调试环境), 可以通过 `ulimit` 命令来检查是否限制了 `core dump` 文件的大小, 如果这个限制阈值很小, 那么当进程使用的内存空间超出这个阈值时, 内核将不能成功生成 `core dump` 文件。我们通过输入 `ulimit -c unlimited` 命令来解除这个限制。

假设我们想分析上文的 `a.out` 程序崩溃后产生的 `core dump` 文件, 看看程序到底崩溃在哪一行, 我们可以这样做:

首先, 在 shell 环境下输入如下命令:

```
gdb ./a.out ./core
```

然后, 在 gdb 调试环境下, 输入 `bt` 命令, 即可查看程序崩溃时的栈帧了。

2. 10. GNU make

当项目源码由许多个文件构成时，还采用 `gcc` 命令的方式来编译是不明智的，这样工作效率很低，而且很容易犯错误。这时候我们需要编写一个 `Makefile` 脚本文件，然后通过 GNU 的 `make` 工具来解释这个 `Makefile` 脚本文件，就自动地构建整个项目了。

`Makefile` 脚本文件的语法规则相当简单，其核心规则如下：

```
target ... : prerequisites ...  
    command
```

“`target ...`”是要构建的目标列表，“`prerequisites ...`”是构建冒号“`:`”前面的目标所需要的依赖项。当 `make` 程序分析这一行规则时，如果发现依赖项文件的修改时间和目标文件的修改时间不一致，就会触发执行下一行的 `command` 命令。

`command` 命令可以是任意合法的 `shell` 命令。

`command` 命令前面需要一个 `tab` 制表符作为前导。

`Makefile` 文件规则就是这么简单，下面再给个简单的示例帮助理解。

以下是一个文件名为 `Makefile` 的文本文件的全部内容：

```
hello:hello.c  
    gcc hello.c -g
```

有了 `Makefile` 文件，我们要构建 `hello` 项目时，只需要在此文件所在的目录下输入 `make` 命令就可以了。

你可以自定义构建脚本文件的文件名，比如 `my-makefile`，那么你构建时只需要在 `make` 命令后带上自定义的文件名作为参数即可。这样非常规的做法属于自找麻烦的行为。

GNU `make` 的官方手册：<https://www.gnu.org/software/make/manual/>

2. 11. CMake

假如我们的项目只需要在 Linux 平台下构建，那么学会使用 `make` 工具就够了。但是，如果我们希望写出来的一套自动化构建脚本，不经任何修改，就能在其它平台（如 Windows、MacOS、Android 等）上使用，那么就需要掌握一种跨平台的自动化构建工具了。

CMake 就是一个跨平台自动化构建工具。

CMake 的官方手册：<https://cmake.org/documentation/>

2. 12. valgrind

valgrind 是一款用于内存泄露检测的开源工具软件，它的名字取自北欧神话英灵殿的入口。它是 Linux 下做内存分析的神器。常用的 Linux 发行版一般都没有自带 valgrind，需要开发者自行下载安装。

valgrind 下载地址：<https://valgrind.org/downloads/current.html>

使用类似如下命令启动被检测的程序：

```
valgrind --tool=memcheck --leak-check=full --track-origins=yes --leak-resolution=high
--show-reachable=yes --log-file=memchecklog ./a.out
```

valgrind 官网手册地址：<https://valgrind.org/docs/manual/manual.html>

2. 13. Git

<https://git-scm.com/docs>

Git 是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持。

在本机使用 git 命令行或者 GUI 工具能很容易的创建一个本地源码仓库，对本机的源码进行版本管理。

2. 14. MySQL

<https://dev.mysql.com/doc/>

MySQL 是 Linux 平台上常用的关系型数据库管理系统。

2. 15. Redis

<https://redis.io/>

Redis 是一个开源（BSD 许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。

Redis 是基于键值对非关系型（NoSQL）数据库。

非关系型数据库还有很多，比如 MongoDB、InfluxDB、Cassandra 等。

MongoDB 是基于文档的 DBMS 系统。

InfluxDB 是优秀的时序数据库，适用于写多读少的时序数据的存储。

2. 16. nginx

<http://nginx.org/>

Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件 (IMAP/POP3) 代理服务器，在 BSD-like 协议下发行。其特点是占有内存少，并发能力强。

Nginx 采用 C 进行编写，其源代码以类 BSD 许可发布。

2. 17. HAProxy

<http://www.haproxy.org/>

HAProxy 是一个使用 C 语言编写的开源软件，其核心功能是负载均衡。

HAProxy 提供了 L4(TCP)和 L7(HTTP)两种负载均衡能力。

就负载均衡这一点而言，HAProxy 比 nginx 更专业。

比 HAProxy 更专业的负载均衡软件还有 LVS。

LVS 具备 L3 (IP) 层的负载均衡能力，能让多台后端业务服务器共享 LVS 网关的同一个 IP 地址。

LVS 由两部分组成，包括 ipvs 和 ipvsadm。ipvs 工作在内核空间，ipvsadm 工作在用户空间。

LVS 有多种工作模式，类型如下：

- NAT: 修改请求报文的目标 IP，多目标 IP 的 DNAT
- DR: 操纵封装新的 MAC 地址
- TUN: 在原请求 IP 报文之外新加一个 IP 首部
- FullNAT: 修改请求报文的源和目标 IP

生产环境中大多使用 DR 模型工作。

2. 18. libuv

<https://libuv.org/>

libuv 是一个高性能的跨平台异步 I/O 库。

libuv 几乎是为 node.js 而发明的。

node.js 最初开始于 2009 年，是一个可以让 Javascript 代码离开浏览器的执行环境也可以执行的项目。

node.js 使用了 Google 的 V8 解析引擎和 Marc Lehmann 的 libev。

node.js 将事件驱动的 I/O 模型与适合该模型的编程语言(Javascript)融合在了一起。

随着 node.js 的日益流行，node.js 需要同时支持 windows，但是 libev 只能在 Unix 环境下运行。

Windows 平台上的性能最好的内核事件通知机制是 IOCP，而 Linux 平台上性能最好的网络 I/O 事件通知机制是 epoll。

libuv 提供了一个跨平台的抽象，并分别针对 IOCP 和 epoll 机制进行了封装实现，因此它无论在 Windows 还是 Linux 平台上都有非常优秀的性能表现。

在 libuv 这样优秀的异步 I/O 库和性能最好 javascript 解释引擎 V8 的加持下，node.js 快速发展成了一个活跃的服务器端应用开发生态。node.js 在跨平台桌面应用开发领域也占有

一席之地，比如 Electron 项目，著名的 Visual Studio Code 就是基于这个框架开发的。

2. 19. st

st 是 State-Thread 的简称，是一个由 C 语言编写的小巧、简洁却高效的开源协程库。

原版的 st 库：<https://sourceforge.net/projects/state-threads>

SRS 的作者 patch 过的 st 库：<https://github.com/ossrs/state-threads>

2. 20. ProtoBuf

<https://developers.google.cn/protocol-buffers?hl=zh-cn>

类似的序列化协议还有 FlatBuffers，详情参考 <https://google.github.io/flatbuffers/>。

这两个协议都是二进制格式编码的，比文本格式编码的 json 和 XML 更加紧凑高效，能有效降低网络带宽需求并同时提高前端（客户端）程序解码速度。

2. 21. MQTT

<https://mqtt.org/>

MQTT（Message Queuing Telemetry Transport，消息队列遥测传输协议），是一种基于发布/订阅（publish/subscribe）模式的消息协议。它工作在 TCP/IP 协议族上，由 IBM 在 1999 年发布。

MQTT 优点在于轻量、简单和易于实现。这些优点使它适用范围非常广泛，如：机器与机器（M2M）通信和物联网（IoT）。

MQTT 在通过卫星链路通信的传感器、偶尔拨号的医疗设备、智能家居、及一些小型化设备中已广泛使用。

<http://mosquitto.org/>

Eclipse Mosquitto 是一个采用 C 语言编写的轻量级的消息中介，它实现了 MQTT 协议版本 5.0, 3.1.1 和 3.1，以 EPL/EDL 许可协议开源。

Mosquitto 适用于从低功耗的单板机到高性能的服务器器的所有设备。

2. 22. Docker

<https://www.docker.org.cn/>

在容器技术出现之前，云计算平台虚拟化的基本粒度是虚拟机，代表性的技术有 VMWare 和 OpenStack。如果用户想拥有一个隔离的云服务运行环境就需要至少拥有一台独占的云主机（一般都是虚拟机）。

有了容器技术，我们可以只在一台公用的虚拟主机上创建许多个轻量级的应用容器。拥有一个应用容器所需的成本相对于拥有一台独占的虚拟主机来说要低得多。这样云服务提供商可以用一台物理云主机来为更多的用户提供隔离的后台应用运行环境，达到降低云服务使用成本的目的。从云服务市场的实际情况来看，容器的租用价格比云主机低很多。容器技术的意义不仅仅是降低成本，它还能让软件的打包、发行、运行过程变得更加标准化和自动化，提高生产效率。

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器或 Windows 机器上。创建一个隔离的 Docker 容器比创建一个比传统的虚拟机所需要的资源（如 CPU、内存、磁盘空间等）开销要低很多。

Docker 本身并不是容器，它是创建容器的工具，是应用容器引擎。Docker 技术的三大核心概念分别是镜像（Image）、容器（Container）和仓库（Repository）。

Kubernetes 是用来管理 Docker 容器的系统，由 Google 开发。Kubernetes 这个单词来自于希腊语，含义是舵手或领航员。K8S 是 Kubernetes 的缩写，用数字“8”字代替这个单词首尾字母中间的 8 个字符。

第 3 章 知名开源流媒体服务器分析

作者知道的流媒体服务器超过 40 款，本章仅介绍采用 C/C++ 语言开发的最常用的几款，将其它的放在 Others 小节中简单地列举一下。

3.1. Live555

Live555 官网：<http://live555.com/>

Live555 是一个实现了 RTSP 协议的开源流媒体框架。

Live555 流媒体框架包含 RTSP 服务器和客户端的实现。

Live555 项目基于自己的流媒体框架实现了一个功能简单的流媒体服务器。我们将这个流媒体服务器称为“Live555 Media Server”。

3.1.1. 支持的文件格式

Live555 Media Server 支持以下常用的（非全部）文件格式的流化（Streaming）：

- H.264 视频裸流文件（扩展名为".264"）
- H.265 视频裸流文件（扩展名为".265"）
- AAC 音频文件（采用 ADTS 编码，扩展名为".aac"）
- Matroska 或 WebM 容器文件（扩展名为".mkv"或".webm"）
- MPEG-1 或 2 (包含 MPEG Audio Layer III) 音频文件（扩展名为".mp3"）
- Ogg 音频文件（扩展名为".ogg"|"ogv"|"opus"）
- WAV (PCM) 音频文件（扩展名为".wav"）
- AMR 音频文件（扩展名为".amr"）

经过 Live555 流化后的视频流或音频流可以通过任何支持 RTSP 协议的媒体播放器（如 VLC）来播放，具体用法可以参考下一节的介绍。

3.1.2. 支持的传输协议

Live555 媒体服务器支持以下流媒体传输协议：

- RTP/RTCP/RTSP
- SIP

3.1.3. 构建和运行方法

以 Windows 平台为例，构建和运行 Live555 媒体服务器的步骤如下：

- (1) 通过 Visual Studio 2019 构建 Live555 项目源码，生成可执行文件 mediaServer.exe。
- (2) 在 mediaServer.exe 目录下，放入你准备流化的媒体文件，比如 test.264。
- (3) 启动 mediaServer.exe。

- (4) 在与 mediaServer.exe 进程相同操作系统环境下（本机环境），启动 VLC 播放器。
- (5) 在 VLC 播放器主菜单中选择“媒体”，在弹出的子菜单中选择“打开网络串流”，然后输入 `rtsp://127.0.0.1:554/test.264` 即可观看由 Live555 媒体服务器流化后在线视频了。

3.1.4. 源码目录分析

本文分析的 Live555 源码的最后修改日期是 2020-12-12。

Live555 源码官网下载地址为：<http://live555.com/liveMedia/public/>

Live555 源码由如下目录构成：

目录	内容描述
BasicUsageEnvironment	包含对 UsageEnvironment 目录中的一些抽象接口类的进一步实现，但是仍然是抽象类，还不能实例化。关键符号：BasicHashTable、BasicUsageEnvironment、BasicTaskScheduler、DelayQueue、HandlerSet。
groupsock	包含网络相关类的定义和实现。groupsock 具备组播的功能，可以将 RTP 数据转发给一组标识客户端会话的套接字。关键符号：createSocket
hlsProxy	包含 HLS 代理服务器项目。它使得用户可以通过浏览器观看位于代理服务器后端的 RTSP/RTP 流。此代理服务器要求在同一台主机上部署一个 Web 服务器程序（比如：nginx）。此项目的基本原理是将 RTSP 流转换为 HLS 切片。
liveMedia	包含了 Live555 流媒体框架的核心功能实现。关键符号：MediaSource、MediaSink。
testProgs	包含许多个独立的测试示例程序，比如：openRTSP、testRTSPClient、testOnDemandRTSPServer 等。这些测试程序可作为研究 Live555 源码的不错的起点。
mediaServer	包含了 Live555 Media Server 的实现。
proxyServer	包含一个 RTSP 代理服务器项目，它可以从其他流媒体服务器（比如支持 RTSP 协议的 IPCamera）拉取实时的视频流，然后转发给多个 RTSP 客户端。此程序主要用于转发 IPCamera 的实时视频流。hlsProxy 可以将 RTSP 流转换成符合 HLS 协议的切片文件，从而让用户通过浏览器访问 IPCamera 的实时视频，而 proxyServer 只能转发给 RTSP 客户端。主流的浏览器都不支持 RTSP 协议的音视频流，但是几乎都支持 HLS 协议的音视频播放。
UsageEnvironment	包含一些基本数据结构以及工具类的定义，如：HashTable、字符串复制函数。还包含几个抽象接口类定义，如：UsageEnvironment、TaskScheduler。
WindowsAudioInputDevice	包含用于实现 Windows 平台下从麦克风采集 PCM 音频的类。

表 3-1 Live555 源码目录描述表

3.1.5. 关键模块分析

Live555 流媒体服务器 `mediaServer` 由 4 个基础模块构成, 它们分别是 `UsageEnvironment`、`UsageEnvironment`、`groupSock` 和 `liveMedia`。`mediaServer` 工程 (模块) 只是对这 4 个基础模块提供的接口的简单调用和组合。

下文将对 4 个基础模块以及 `mediaServer` 总成模块逐个分析。

3.1.5.1. UsageEnvironment

此模块除了几个基本数据类型和工具函数定义外, 还包含几个重要的抽象类的定义:

- (1) `UsageEnvironment` 抽象类: 代表了整个系统运行的环境, 它提供了错误记录和错误报告的功能, 无论哪一个类要输出错误, 都需要保存 `UsageEnvironment` 的指针。学习过 Android App 开发的读者可将其作用理解为应用程序上下文 (`ApplicationContext`)。
- (2) `TaskScheduler` 抽象类: 负责监听 `socket` 事件、定时触发的事件、手动触发的事件, 当事件可触发时调用相应事件处理回调函数。每个 `UsageEnvironment` 实例都要绑定一个 `TaskScheduler` 实例。学习过 Windows 桌面开发的读者可将其作用理解为 Windows 桌面窗口应用程序的消息循环 (`MessageLoop`)。
- (3) `HashTable` 抽象类: 定义了一个通用的 Hash 表的接口方法。

3.1.5.2. BasicUsageEnvironment

此模块对 `UsageEnvironment` 模块中定义的抽象类进行了派生, 增加了更多的辅助方法。`BasicUsageEnvironment0` 重载了 `UsageEnvironment` 抽象类的部分接口方法, 但仍然是抽象类。`BasicUsageEnvironment` 类继承了 `BasicUsageEnvironment0`, 重载了一组错误输出操作符接口方法, 为上层模块 (如 `mediaServer` 和 `proxyServer`) 提供了一个可实例化的具体类。此模块还实现了几个重要的容器类型, 如: `BasicHashTable`、`DelayQueue`、`HandlerDescriptor`、`HandlerSet`。

3.1.5.3. groupsock

此模块封装了和操作系统相关的 `socket` 网络编程的细节, 为 `liveMedia` 模块提供组播媒体数据的能力。

3.1.5.4. liveMedia

此模块是 Live555 流媒体框架的核心库，整个流媒体框架的业务模型就在这个库中定义的。由于类型非常多，作者只列出对理解业务模型最关键的几个类型的定义。

Medium ◀ --MediaSource ◀ --FramedSource ◀ --FramedFileSource ◀ --ByteStreamFileSource
.Medium ◀ --MediaSource ◀ --FramedSource ◀ --FramedFileSource ◀ --ADTSAudioFileSource
Medium ◀ --MediaSource ◀ --FramedSource ◀ --FrameFilter ◀ --MPEGVideoStreamFramer ◀ --H264or5VideoStreamFramer ◀ --H264or5VideoStreamDiscreteFramer ◀ --H265VideoStreamDiscreteFramer
Medium ◀ --MediaSource ◀ --FramedSource ◀ --FrameFilter ◀ --MPEGVideoStreamFramer ◀ --H264or5VideoStreamFramer ◀ --H265VideoStreamFramer
Medium ◀ --MediaSource ◀ --FramedSource ◀ --RTPSource ◀ --MultiFramedRTPSource ◀ --H265VideoRTPSource
Medium ◀ --MediaSink ◀ --RTPSink ◀ --MultiFramedRTPSink ◀ --VideoRTPSink ◀ --H264or5VideoRTPSink ◀ --H265VideoRTPSink
Medium ◀ --MediaSink ◀ --FileSink ◀ --H264or5VideoFileSink ◀ --VideoRTPSink ◀ --H265VideoFileSink
Medium ◀ --MediaSession
MediaSubSession

表 2-2liveMedia 模块关键类型列表

3.1.5.5. mediaServer

此模块是对前面 4 个基础模块的集成，构造并启动 Live555 媒体框架的基础模块。

3.1.6. 优缺点总结

优点：

- (1) 设计简单，容易学习。
- (2) 容易使用。

缺点：

- (1) 运行时性能一般，在有大量并发客户端的情况下，CPU 占用较高。
- (2) 不支持 HTTP-FLV、HLS、RTMP、GB28181。

3.2. GStreamer

GStreamer 官网: <https://gstreamer.freedesktop.org/>

GStreamer 是用来构建流媒体应用的开源多媒体框架, 以 LGPL 许可发布。

1999 年 Erik Walthinsen 创建了 GStreamer。2004 年, 新公司 Flumotion 成立, 并使用 GStreamer 编写一个流媒体服务器 Flumotion, 并提供多媒体解决方案。

GStreamer 日后在商业上获取巨大成功有许多不同的公司采用 (诺基亚、摩托罗拉、德州仪器、飞思卡尔、英特尔等等), 并已成为一个非常强大的跨平台多媒体框架。

GStreamer 的跨平台设计, 使其能够在 Linux、Solaris、OpenSolaris、FreeBSD、OpenBSD、NetBSD、Mac OS X、Microsoft Windows 和 OS/400 上运行。

GStreamer 可以像 ffmpeg 那样构造出一个完整的流媒体处理组件图, 实现一套从媒体采集、流化到回放的完整处理流程。

gst-rtsp-server 是基于 GStreamer 封装的用于构建 RTSP 服务器的库。

gst-rtsp-server 源码: <https://github.com/GStreamer/gst-rtsp-server>

3.2.1. 基本架构

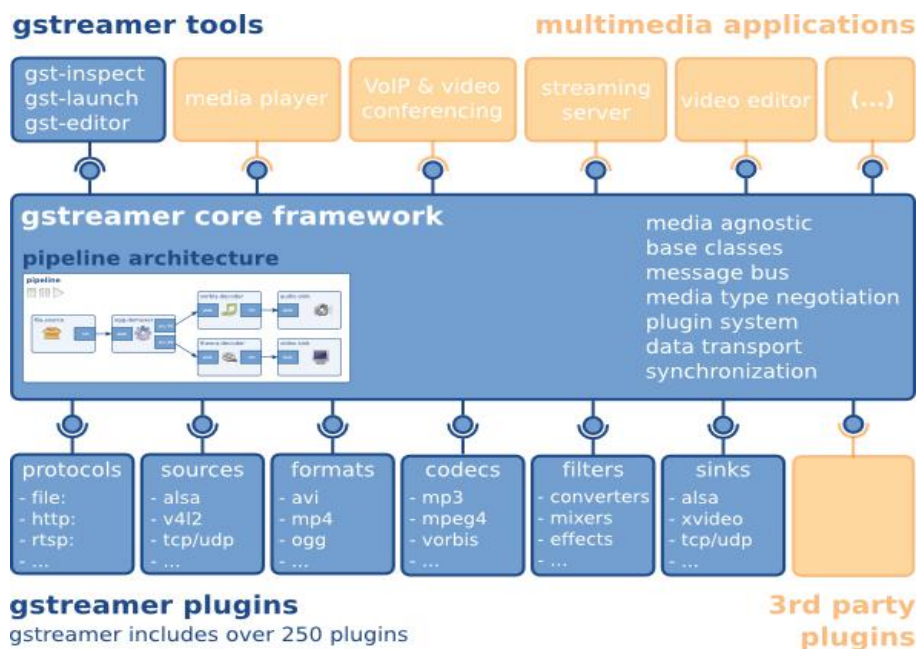


表 3-3 基于 GStreamer 的应用分层

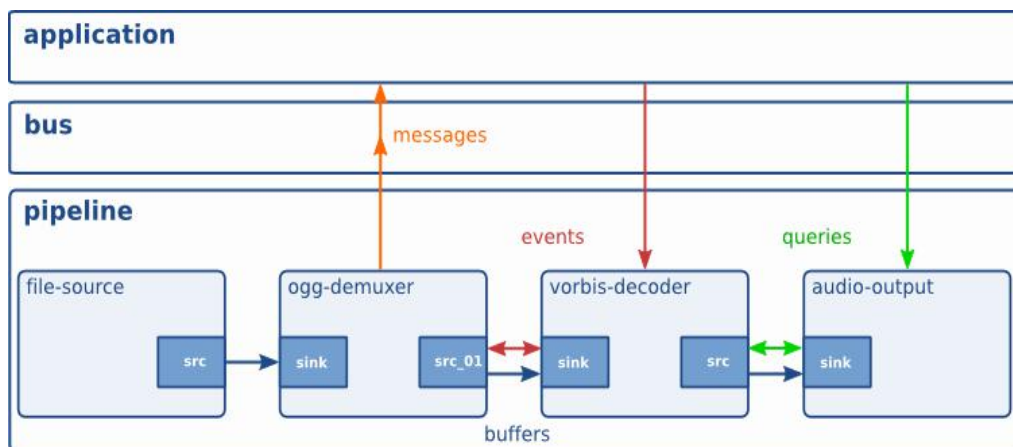


表 3-4 GStreamer 数据处理流程

3.2.2. 支持的文件格式

gst-rtsp-server 支持以下常用的（非全部）文件格式的流化（Streaming）：

- (1) avi 文件（扩展名为".avi"）
- (2) mp4 文件（扩展名为".mp4"）

3.2.3. 支持的传输协议

gst-rtsp-server 支持以下流媒体传输协议：

- (1) RTSP/RTP/RTCP

3.2.4. 优缺点总结

优点：

- (1) 框架设计借鉴了 DirectShow 的设计思想，灵活规范。
- (2) 纯 C 开发。

缺点：

- (1) 此框架的典型应用是播放器。
- (2) 不支持 Web 直播。
- (3) 在国内社区支持不如 ZLMediaKit。

3.3. DSS

DSS 全称是 Darwin Streaming Server。

DSS 项目地址：<https://github.com/macforge/dss>

DSS 是 Apple 公司提供的开源实时流媒体服务器程序，可以运行在 Windows、Mac OS X、Linux 操作系统上。

3.3.1. 支持的文件格式

DSS 支持以下常用的（非全部）文件格式的流化（Streaming）：

- (1) mp4 文件（扩展名为".mp4"）
- (2) mp3 文件（扩展名为".mp3"）

3.3.2. 支持的传输协议

DSS 支持以下流媒体传输协议：

- (1) RTP/RTCP/RTSP

3.3.3. 优缺点总结

优点：

- (1) 模块化设计较好。

缺点：

- (1) 支持的流媒体传输协议不够丰富。
- (2) Web 后台管理技术老旧。

3.4. SRS

SRS 全称是 Simple RTMP Server。

SRS 项目地址：<https://github.com/ossrs/srs>

SRS 是国人写的一款非常优秀的开源流媒体服务器软件，可用于直播/录播/视频客服等多种场景，其定位是运营级的互联网直播服务器集群。

3.4.1. 支持的文件格式

SRS 支持以下常用的（非全部）文件格式的流化（Streaming）：

- (1) H.264 视频裸流文件（文件扩展名为".264"）
- (2) H.265 视频裸流文件（文件扩展名为".265"）
- (3) AAC 音频文件（采用 ADTS 编码，文件扩展名为".aac"）

3.4.2. 支持的传输协议

SRS 支持以下流媒体传输协议：

- (1) RTSP/RTP/RTCP
- (2) RTMP
- (3) HTTP-FLV
- (4) HLS

3.4.3. 优缺点总结

优点：

- (1) 单线程多协程并发模式。
- (2) 单核性能高，支持集群。
- (3) 代码简洁朴素，可读性好。
- (4) 经典的编程风格，传统程序员接受度好。

3.5. ZLMediaKit

ZLMediaKit 项目地址: <https://github.com/xia-chu/ZLMediaKit>

相对于 Live555、GStreamer 和 DSS 流媒体服务器而言, ZLMediaKit 非常年轻, 它是 2018 年前后才流行起来的一套流媒体框架。

ZLMediaKit 中的 ZL 是该项目所有者的名的缩写。

ZLMediaKit 运用了许多 C++11 标准的特性。

ZLMediaKit 是一套通用的流媒体开发框架, 并且基于这套框架开发了一个通用的流媒体服务器, 支持主流的流媒体传输协议。

3.5.1. 支持的文件格式

ZLMediaKit 支持以下常用的 (非全部) 文件格式的流化 (Streaming) :

- (1) MP4 文件 (文件扩展名为 ".mp4")

3.5.2. 支持的传输协议

ZLMediaKit 支持以下流媒体传输协议:

- (1) RTSP/RTP/RTCP
- (2) RTMP
- (3) GB28181
- (4) HTTP-FLV
- (5) HLS

3.5.3. 优缺点总结

优点:

- (1) 多线程并发模式, 单进程并发性能优秀。
- (2) 支持的媒体接入协议齐全, 功能覆盖面广。
- (3) 编程风格现代化, 深受年轻人喜爱。
- (4) 跨平台能力强。
- (5) Web 直播效果好。
- (6) 有活跃的社区支持。

ZLMediaKit 是目前国内商业公司的理想选择。

3.6. xsnode

xsnode 官网: <http://codemi.net>

xsnode 的发明者就是本书的作者, xs 是他的名的缩写。

xsnode 不仅仅是一款十分专业的直播服务器, 还是一个通用的分布式计算平台。

xsnode 是作者为虚拟现实和增强现实领域应用设计的, 它有十分卓越的性能表现。

xsnode 可与作者开发的 xoplayer 播放器和 cos 实时操作系统组合, 形成一个完整的多媒体应用生态系统。

xsnode 的作者遵循极简的原则, 选择仅支持主流的网络传输协议和音视频编解码标准。

xsnode 将根据多媒体技术的发展, 及时地抛弃历史包袱, 坚持使用主流的技术标准, 尽可能地降低多媒体数据传输和存储成本。

xsnode 是一个需要开发者重新思考计算机编程本质的创新平台, 它对开发者的水平要求极高, 普通水平的 C++ 开发人员很难理解其设计思想。

基于大部分商业公司的实际业务需求考虑, 作者在后文中选择了 ZLMediaKit 作为分析目标, 系统地讲解其构建、配置、运行方法和内部实现原理。

作者将会用另一部专著来论述 xsnode 的设计思想。

3.6.1. 支持的传输协议

xsnode 支持以下媒体传输协议:

- (1) RTSP/RTP/RTCP 协议
- (2) 私有 xs 协议

3.6.2. 优缺点总结

优点:

- (1) 多进程或多线程并发模式。
- (2) 低内存占用、低 CPU 占用, 低延时。
- (3) 高性能、高并发、高可用。
- (4) 以开箱即用为设计目标, 而不是二次开发。
- (5) 运维简单, 非技术人员都可以胜任。

缺点:

- (1) 仅支持 AAC、H.264 和 H.265。
- (2) 作者不考虑支持 Web 直播。
- (3) 作者不考虑引入协程机制。
- (4) 对模块开发者的技术水平要求很高。

3.7. Others

除了上述常用的 C/C++ 开源流媒体服务器外, 还有一些其它的比较知名的流媒体服务器, 比如: EasyDarwin(go)、Red5(Java)、Wowza(Java)、FFserver(C)、crtmpserver(C++)等等。

第 4 章 Linux 版直播服务器需求分析

4.1. 目标应用场景

- 智能安防
- 在线教育
- 娱乐直播

4.2. 运行时硬件环境约束

- CPU 架构: x86_64
- 物理内存大小: 至少 4GB

4.3. 运行时软件环境约束

- Linux 内核版本要求: 不低于 4.15.0
- Linux 发行版要求: 64 位的 Ubuntu
- 直播服务器进程可用内存大小: 至少 1GB

4.4. 功能要求

- 支持的视频压缩标准: H.264 | H.265
- 支持的音频压缩标准: AAC | Opus
- 支持的音视频流接入协议: RTMP | RTSP | GB28181

4.5. 性能要求

在带宽不低于 100Mbps 且 ping 值稳定在 10ms 以内的良好 IP 网络环境下:

- 由媒体服务器接入和转发所导致的延时稳定控制在 100ms 以内
稳定是指大于 99.99% 的概率。

4.6. 可用性要求

- 平均无故障运行时间: 大于 365 天

4.7. 方案选择

作者通过对知名开源流媒体服务器的优缺点总结和对目标应用场景的需求调研，发现 ZLMediaKit 完全可以满足上述需求。

由于 ZLMediaKit 项目已经通过商业应用场景的实践检验了，所以作者建议读者直接使用它，而不是重新设计一套功能几乎一模一样的流媒体服务器。

本教程中，作者决定采用 ZLMediaKit 作为“Linux 版直播服务器”的最终解决方案。

我们不仅要掌握如何构建和运行 ZLMediaKit，还需要熟悉其内部实现原理，这样当公司有特殊业务需求时，我们有能力在其基础上做一些扩展开发工作。

下一章，作者将从 ZLMediaKit 源码目录开始，为读者逐层分析它的内部实现原理。

第 5 章 ZLMediaKit 源码分析

5.1. 源码目录

5.1.1. 根目录

后文采用“ZLMediaKit/”来表示 ZLMediaKit 源码的根目录，在这个目录下可以看到 3rdpart、Android、api 等子目录。

ZLMediaKit/目录下的内容描述如下：

子目录	内容描述
3rdpart	第三方开源库，包含三个子目录，内容如下： (1) jsoncpp: 用来处理 json 格式字符串的 C++库。 (2) media-server: 据说是“老陈”提供的 C++媒体服务器，里面包含了若干媒体文件格式、传输协议的封装类。 (3) ZLToolKit: 整个 ZLMediaKit 项目范围内通用的基础工具库，ZLMediaKit 流媒体协议库和 MediaServer 应用程序就是基于这个库实现的。此库封装的功能包括日志、线程池、定时器、任务、缓冲区、套接字、事件处理循环、工具类等，由于它比较通用，因此可以作为“第三方开源库”给其它项目使用（包括 ZLMediaKit 项目）。
Android	采用 ZLMediaKit 框架开发的 Android 版流媒体播放器。
api	将 C++风格的 ZLMediaKit 核心框架封装成纯 C 风格的 API，供 C 程序调用。
cmake	cmake 构建文件的辅助模块。
conf	包含流媒体服务器的配置文件 config.ini
docker	docker 镜像构建脚本
postman	restful 接口测试工具 postman 的测试项目文件
release	项目构建目标输出目录
server	基于 3rdpart 和 src 目录中的模块开发的一个流媒体服务器。
src	ZLMediaKit 流媒体开发库，主要由各种流媒体传输协议实现构成，还有 Player、Pusher 和 Record 等功能模块。
tests	测试代码
www	帮助文件

表 5-1 ZLMediaKit/目录内容描述

5.1.2. Util

ZLMediaKit/3rdpart/ZLToolKit/src/Util 目录类型描述表:

<p>Option</p> <p>描述:</p> <ul style="list-style-type: none"> ① 命令行参数选项封装。 ② 用于定义合法的命令行参数选项的处理规则。
<p>OptionParser</p> <p>描述:</p> <ul style="list-style-type: none"> ① 命令行参数选项解析器。 ② 作为合法的命令行参数选项处理规则的容器。 ③ 支持解析命令行传递给 main 函数的 argc 和 argv 参数。
<p>mINI ◁- CMD</p> <p>描述:</p> <ul style="list-style-type: none"> ① 作为所有扩展命令的基类。 ② 支持命令行字符串信息分离。 ③ 支持字典存储。 ④ 聚合了一个 OptionParser 对象 _parser，用于存储选项处理规则和解析命令行输入。 ⑤ 派生类在构造函数中向 parser 对象输入合法的选项处理规则。
<p>mINI ◁- CMD ◁- CMD_help</p> <p>描述:</p> <ul style="list-style-type: none"> ① 构造帮助命令 “cmd”。
<p>mINI ◁- CMD ◁- CMD_exit</p> <p>描述:</p> <ul style="list-style-type: none"> ① 构造退出命令 “exit”。
<p>mINI ◁- CMD ◁- CMD_clear</p> <p>描述:</p> <ul style="list-style-type: none"> ① 构造清屏命令 “clear”。
<p>CMDRegister</p> <p>描述:</p> <ul style="list-style-type: none"> ① 用于 ZLMediaKit 内部 ShellSession 支持的命令注册。 ② 通过全局静态变量 s_token 的初始化实现命令的自动注册。
<p>std::exception ◁- ExitException</p> <p>描述:</p> <ul style="list-style-type: none"> ① 退出异常类。

<p>File 描述: ① 文件和目录操作。</p>
<p>function_traits<> 描述: ① 函数、lambda 转 functional。</p>
<p>List 描述: ① 单链表模板类。</p>
<p>noncopyable <- Logger 描述: ① 各种日志输出通道类的总管。单实例名称 g_defaultLogger。</p>
<p>ostringstream <- LogContext 描述: ① 日志输出的上下文（文件名、函数名、行号等）。</p>
<p>LogContextCapturer 描述: ① 日志捕获器。构造函数的默认参数自动填写源代码位置上下文。</p>
<p>noncopyable <- LogWriter 描述: ① 日志输出器接口类。</p>
<p>noncopyable <- LogWriter <- AsyncLogWriter 描述: ① 异步日志输出器的实现。 ② 在独立的工作线程中 run(), 向所有日志通道对象写入日志。</p>
<p>noncopyable <- LogChannel 描述: ① 日志通道抽象类。可运行时设置日志输出级别。</p>
<p>noncopyable <- LogChannel <- ConsoleChannel 描述: ① 带色彩属性输出日志到控制台（cout）。</p>
<p>noncopyable <- LogChannel <- FileChannelBase 描述:</p>

① 异步日志输出器的实现。在独立的工作线程中 run()
<pre>noncopyable <- LogChannel <- FileChannelBase <- FileChannel</pre> <p>描述:</p> <p>① 异步日志输出器的实现。在独立的工作线程中 run()</p>
<p>MD5</p> <p>描述:</p> <p>① MD5 加密模块。</p>
<pre>std::map<key, variant> <- mINI_basic<string, variant></pre> <p>描述:</p> <p>① INI 配置文件的配置项字典。</p>
<pre>using mINI = mINI_basic<string, variant></pre>
<pre>std::string <- variant</pre> <p>描述:</p> <p>① 采用 std::string 作为存储的变体类型实现。</p>
<p>EventDispatcher</p>
<p>NotificationCenter</p> <p>描述:</p> <p>① 自定义事件广播通知机制。</p>
<p>onceToken</p> <p>描述:</p> <p>① 用于构造和析构时各执行一次指定的操作。</p>
<pre>std::shared_ptr<> <- shared_ptr_imp<></pre>
<p>ResourcePool_l<></p> <p>描述:</p> <p>① 采用 List 容器实现的对象池。</p>
<p>ResourcePool<></p> <p>描述:</p> <p>① 对象池顶层设计。</p> <p>② 用于高频率分配释放的对象（如 RtpPacket 和 TSPacket）的循环利用。</p>
<p>RingDelegate<></p> <p>描述:</p> <p>① 环形缓冲队列数据包入队列前的回调通知接口（委托）。</p>
<p>_RingReader<></p> <p>描述:</p>

<p>① 环形缓冲队列读取器，对该对象的一切操作都应该在它绑定到的 <code>poller</code> 线程中执行。</p>
<p>_RingStorage<> 描述： ① 环形缓冲队列容器类，负责缓冲区的实际存取。</p>
<p>_RingReaderDispatcher<> 描述： ① 聚合了一个环形缓冲队列容器。 ② 聚合了环形缓冲队列读取器 <code>map</code>。 ③ 每绑定一次 <code>EventPoller</code> 对象，就会创建一个 <code>RingReader</code> 对象。 ④ 调用此类的 <code>write</code> 方法时，将会在调用者线程内通知 <code>map</code> 内所有 <code>RingReader</code>。调用者负责在 <code>EventPoller</code> 所属的线程中执行 <code>write</code> 方法。 ⑤ 对使用者提供 <code>map</code> 大小变化通知。 ⑥ 读者可以将此类理解为一对多的数据流多通连接器。</p>
<p>RingBuffer<> 描述： ① 环形缓冲区队列。 ② 具备数据包入队列前执行委托接口的功能。 ③ 具备将数据包“线程安全地”分发给所有订阅了此对象的 <code>EventPoller</code> 线程的 <code>_RingReader<></code> 对象的能力。 ④ 一个 <code>_RingBuffer<></code> 可以 <code>attach</code> 多个 <code>EventPoller</code> 对象引用。 ⑤ 每个 <code>EventPoller</code> 对象指针映射一个 <code>_RingReaderDispatcher<></code> 对象。 ⑥ 每个 <code>_RingReaderDispatcher<></code> 对象可以 <code>attach</code> 多个 <code>_RingReader<></code> 对象。</p>
<p>SHA1 描述： ① SHA1 散列值计算类。 ② 2017 年 2 月 23 日，SHA-1 算法已经被正式攻破。</p>
<p>exception ◀- SQLException 描述： ① 数据库访问异常信息。</p>
<p>SqlConnection 描述： ① MySQL 连接封装。 ② 实现了常用查询方法。</p>
<p>SqlPool 描述： ① MySQL 连接池封装。</p>

② 聚合了一个 <code>WorkThreadPool</code> 对象，用于异步执行 <code>sql</code> 语句。
SqlStream 描述： ① SQL 语句合成器。 ② 通过用实参替换语句模板中的占位符“？”的方式生成语句。
SqlWriter 描述： ① SQL 查询器。 ② 将数据库持久化机制独立成模块，与流转发核心模块隔离。
SSL_Initor 描述： ① SSL 黑盒模块初始化器。
SSL_Box 描述： ① SSL 加解密黑盒封装。
SSLUtil 描述： ① SSL 证书相关操作封装。
Ticker 描述： ① 用于代码执行时间统计。
SmoothTicker 描述： ① 生成平滑的时间戳。
<code>std::string</code>  <code>_StrPrinter</code> 描述： ① 支持流输入符和流提取运算符的字符串封装。
noncopyable 描述： ① 不可拷贝的对象的基类。
Any 描述： ① 通过堆分配来保存任意对象。


<code>std::unordered_map<string, Any></code>  AnyStorage 描述： ① 属性字典。
Creator 描述： ① 通用类厂模板类。 ② 将对象的创建和销毁流程封装在此类中。

表 5-2 ZLMediaKit/3rdpart/ZLToolkit/src/Util 目录类型描述表

5.1.3. Thread

ZLMediaKit/3rdpart/ZLToolKit/src/Thread 目录类型描述表:

<p>semaphore 描述: ① 信号量。</p>
<p>ThreadLoadCounter 描述: ① 线程负载计数器。</p>
<p>noncopyable <- TaskCancelable 描述: ① 定义了可取消的任务的抽象。</p>
<p>TaskCancelable <- TaskCancelableImp<> typedef TaskCancelableImp<uint64_t(void)> DelayTask; typedef TaskCancelableImp<void()> Task; 描述: ① 可取消的任务的实现。</p>
<p>TaskExecutorInterface 描述: ① 任务执行器接口，支持同步和异步执行指定任务。</p>
<p>ThreadLoadCounter <----- TaskExecutor TaskExecutorInterface <---- 描述: ① 任务执行器的实现。</p>
<p>TaskExecutorGetter 描述: ① 定义了可访问任务执行器的用户类的抽象。</p>
<p>TaskExecutorGetter <- TaskExecutorGetterImp 描述: ① 实现了内部创建的任务执行器的负载情况统计功能。 ② 实现了可获取最小负载的 EventPoller 对象的 getExecutor 方法。 ③ 作为 EventPollerPool 的基类。 ④ 作为 WorkThreadPool 的基类。</p>
<p>TaskQueue 描述:</p>

<ul style="list-style-type: none"> ① 实现了线程安全的任务（函数对象）队列。 ② ThreadPool 类聚合了一个任务队列。
<p>thread_group</p> <p>描述:</p> <ul style="list-style-type: none"> ① 创建并维护一组工作线程。
<p>TaskExecutor ◁- ThreadPool</p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了一个 TaskQueue<Task::Ptr>对象。 ② 聚合了一个 thread_group 对象。 ③ 支持在 thread_group 中的线程上下文中同步执行指定的 Task 对象。 ④ 支持在任意线程上下文中投递 Task 对象，thread_group 中的空闲线程获取到此 Task 对象后，在其线程上下文中同步执行此 Task 对象。 ⑤ 可用于调整指定线程 id 的优先级。 ⑥ 通常用法是创建指定数量的一个 ThreadPool 对象，然后投递需要异步执行的任务，每个拿不到工作任务的工作线程会自动退出，当所有任务执行完成后，所有工作线程都会自动退出，ThreadPool 对象即可成功析构返回。 ⑦ ThreadPool 和 WorkThreadPool 的主要区别是前者的工作线程没有 I/O 事件轮询循环，也没有延时执行任务的机制，使用场景比后者较少。
<p>... ◁- TaskExecutorGetterImp ◁- WorkThreadPool</p> <p>描述:</p> <ul style="list-style-type: none"> ① 借助基类创建并维护一组工作线程。 ② 每个工作线程绑定一个 EventPoller 对象，作为任务执行器。 ③ 具备延时执行任务的能力。

表 5-3 ZLMediaKit/3rdpart/ZLToolkit/src/Thread 目录类型描述表

5.1.4. Poller

ZLMediaKit/3rdpart/ZLToolKit/src/Poller 目录类型描述表:









<p>... <- TaskExecutorGetterImp <- EventPollerPool</p> <p>描述:</p> <ol style="list-style-type: none"> ① 在调用者线程中创建并维护 N 个 EventPoller 实例。 ② EventPoller 实例执行 runLoop 方法时会为自己创建一个工作线程。 ③ 提供迭代调用 EventPoller 执行 TaskIn 函数对象的方法。 ④ 提供 EventPoller 负载均衡算法。
<p>... <- TaskExecutor <- EventPoller</p> <p>描述:</p> <ol style="list-style-type: none"> ① 实现了同步和异步执行任务的接口。 ② 实现了通过管道唤醒的异步任务队列的调度执行。 ③ 实现了延时执行任务的功能。
<p>Pipe</p> <p>描述:</p> <ol style="list-style-type: none"> ① 封装向管道写入消息的方法。 ② 将管道对象与 EventPoller 对象关联, 实现管道事件通知机制。
<p>PipeWrap</p> <p>描述:</p> <ol style="list-style-type: none"> ① 对 Linux 平台的 pipe 接口进行封装。 ② 在 Windows 平台通过 socket 模拟 Linux 的 pipe 行为。
<p>FdSet</p> <p>描述:</p> <ol style="list-style-type: none"> ① 对文件描述符集合的简单封装。
<p>Timer</p> <p>描述:</p> <ol style="list-style-type: none"> ① 与指定 EventPoller 对象绑定的延时执行的回调函数的封装。

表 5-4 ZLMediaKit/3rdpart/ZLToolKit/src/Poller 目录类型描述表

5.1.5. Network

ZLMediaKit/3rdpart/ZLToolKit/src/Network 目录类型描述表:

<p>noncopyable ◀ Buffer</p> <p>描述:</p> <ul style="list-style-type: none"> ① 缓存抽象类。
<p>noncopyable ◀ Buffer ◀ BufferOffset</p> <p>描述:</p> <ul style="list-style-type: none"> ① 支持偏移基址的缓存类。
<p>noncopyable ◀ Buffer ◀ BufferRaw</p> <p>描述:</p> <ul style="list-style-type: none"> ① 可动态扩容的裸指针式缓存类。
<p>noncopyable ◀ Buffer ◀ BufferLikeString</p> <p>描述:</p> <ul style="list-style-type: none"> ① 对 <code>std::string</code> 作为容器的缓存类。 ② 提供类似 <code>std::string</code> 的常用操作。
<p>noncopyable ◀ Buffer ◀ BufferSock</p> <p>描述:</p> <ul style="list-style-type: none"> ① 保存了 <code>sockaddr</code> 指针的缓存。
<p>noncopyable ◀ Buffer ◀ BufferList</p> <p>描述:</p> <ul style="list-style-type: none"> ① 包含了一个缓冲区数组。 ② 支持向指定的 TCP 或 UDP 套接字发送缓冲区队列中的内容。
<p><code>std::exception</code> ◀ SocketException</p> <p>描述:</p> <ul style="list-style-type: none"> ① 用于封装套接字操作错误信息。
<p>SocketNum</p> <p>描述:</p> <ul style="list-style-type: none"> ① 对套接字描述符的简单封装。
<p>SocketFD</p> <p>描述:</p> <ul style="list-style-type: none"> ① 套接字描述符的封装, 绑定了一个 <code>EventPoller</code> 对象。 ② 析构时调用 <code>EventPoller</code> 对象删除对此套接字的事件监听。
<p>MutexWrapper</p>

<p>描述:</p> <ul style="list-style-type: none"> ① 对 <code>std::recursive_mutex</code> 类的封装。
<p>SockInfo</p> <p>描述:</p> <ul style="list-style-type: none"> ① 定义了套接字信息获取接口。
<p>noncopyable  Socket</p> <p>SockInfo  </p> <p>描述:</p> <ul style="list-style-type: none"> ① 异步 IO Socket 对象，包括 TCP 客户端、TCP 服务器和 UDP 套接字。 ② 套接字 <code>connect</code> 调用超过 10 秒后仍变成可写状态即被判定超时错误。 ③ 通过线程池实现阻塞式的 DNS 解析。
<p>SockSender</p> <p>描述:</p> <ul style="list-style-type: none"> ① 定义了各种发送字节流缓冲区的接口方法。
<p>SockSender  SocketHelper</p> <p>SockInfo  </p> <p>TaskExecutorInterface  </p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了一个 Socket 对象。 ② 实现了 SockSender 抽象类中定义的各种花式发送方法。 ③ 实现了 SockInfo 抽象类中的定义的获取套接字相关信息的方法。 ④ 实现了 TaskExecutorInterface 抽象类中的任务执行方法。
<p>SockUtil</p> <p>描述:</p> <ul style="list-style-type: none"> ① 封装了与 OS 实现相关的 socket 操作 API。
<p>DnsCache</p> <p>描述:</p> <ul style="list-style-type: none"> ① 实现了一个主机名 DNS 查询结果缓存。 ② 缓存内的主机名和 sockaddr 映射项在 60 秒内有效。
<p>...  SocketHelper  TcpClient</p> <p>描述:</p> <ul style="list-style-type: none"> ① 通过基类 SocketHelper 创建并聚合一个 Socket 对象。 ② 实现了 TCP 客户端特有的套接字操作流程。
<p>mINI  TcpServer</p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了一个 Socket 对象。

<ul style="list-style-type: none">② 支持配置项字典③ 实现了 TCP 服务器特有的套接字操作流程。
<p>... ◀ SocketHelper ◀ TcpSession</p> <p>描述:</p> <ul style="list-style-type: none">① 服务器端通过监听套接字 <code>accept</code> 得到的 TCP 连接对象。② 在 ZLMediaKit 中一个服务器端 TCP 连接对应一个 RTSP RTMP 会话。

表 5-5 ZLMediaKit/3rdpart/ZLToolKit/src/Network 目录类型描述表

5.1.6. Common

ZLMediaKit/src/Common 目录类型类型描述表:

VideoInfo 描述: ① 视频基本信息。
AudioInfo 描述: ① 音频基本信息。
... <- MultiMediaSourceMuxer <- DevChannel 描述: ① 对 MultiMediaSourceMuxer 对进一步封装。 ② 支持直接输入 YUV 和 PCM 帧, 内部会自动进行压缩后, 再转成各种协议的媒体源。
... <- FrameWriterInterface <- MediaSinkInterface 描述: ① 媒体槽接口。
... <- MediaSinkInterface <- MediaSink TrackSource <- ----- 描述: ① 某个 Track 就绪时通知派生类截取一些信息 (如 vps、sps、pps)。 ② 等待所有 Track 就绪后再通知派生类进行下一步的操作。
MediaSourceEvent 描述: ① 媒体源事件抽象类, 定义了常用媒体控制事件模板方法。
MediaSourceEvent <- MediaSourceEventInterceptor 描述: ① 媒体源事件拦截器。
MediaInfo 描述: ① 解析 URL 获取其中包含的信息, 如协议、主机名、端口、流 id 等。
BytesSpeed 描述: ① 计算和获取字节流速率 (bytes/s)。
TrackSource <- MediaSource 描述:

<ul style="list-style-type: none"> ① 所有协议（RTSP RTMP HLS TS FMP4）直播源的顶层设计。 ② 定义了直播源映射表 SchemaVhostAppStreamMap 类型。 ③ 全局唯一的直播源映射表实例为 s_media_source_map。 ④ 通过 regist 方法可以将 this 对象注册到 s_media_source_map。 ⑤ 通过 find 方法可以从 s_media_source_map 中查找指定名称的流。
<p>FlushPolicy</p> <p>描述:</p> <ul style="list-style-type: none"> ① 缓存刷新策略类。
<p>PacketCache</p> <p>描述:</p> <ul style="list-style-type: none"> ① 用于合并小包的缓存。 ② 当收集到一个完整的帧或者达到某个时长条件后将此缓存中的包批量写入到环形缓存，触发 Reader 将数据投递给下一级节点。 ③ 此类对象被聚合于各种 MediaSource 中。
<p>MediaSink ◀- MultiMuxerPrivate</p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了多种直播协议（RTSP RTMP TS HLS FMP4）的复用器。 ② 若将聚合的多协议复用器对象视为此类的一部分，那么此类显然属于最终的媒体包接收槽，属于 MediaSink 类。
<pre>... ◀- MediaSourceEventInterceptor ◀- MultiMediaSourceMuxer ... ◀- MediaSinkInterface ◀----- ... ◀- MultiMuxerPrivate::Listener ◀- </pre> <p>描述:</p> <ul style="list-style-type: none"> ① 实现了媒体源事件拦截（监听）接口。 ② 聚合了一个 MultiMuxerPrivate 对象，借助它来完成多种流媒体传输协议的媒体源的实际复用流程。 ③ 支持 RTP 代理。
<p>std::multimap<string, string, StrCaseCompare> ◀- StrCaseMap</p> <p>描述:</p> <ul style="list-style-type: none"> ① 大小写敏感的字符串词典。 ② 键值可重复。
<p>Parser</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTSP/HTTP/SIP 协议参数解析器。
<p>DeltaStamp</p> <p>描述:</p> <ul style="list-style-type: none"> ① 计算时间戳增量。

DeltaStamp <- Stamp
描述： <ul style="list-style-type: none">① 解决时间戳回环、回退问题。② 计算相对时间戳。③ 生成平滑的时间戳。

表 5-6 ZLMediaKit/src/Common 目录类型类型描述表

5.1.7. Codec

ZLMediaKit/src/Codec 目录类型类型描述表：

AACEncoder 描述： ① AAC 编码器。
H264Encoder 描述： ① H.264 编码器。

表 5-7 ZLMediaKit/src/Codec 目录类型描述表

5.1.8. Extension

ZLMediaKit/src/Extension 目录类型描述表:

<p>... <- AudioTrack <- AACTrack</p> <p>描述:</p> <p>① AAC 音频轨道描述信息。</p>
<p>... <- Sdp <- AACSDp</p> <p>描述:</p> <p>① AAC 音频的 SDP 封装。</p>
<p>... <- RtmpCodec <- AACRtmpDecoder</p> <p>... <- ResourcePoolHelper<FrameImp> <-</p> <p>描述:</p> <p>① 输入一连串 RtmpPacket 对象, 输出一个完整的 AAC 帧。</p>
<p>... <- AACRtmpDecoder <- AACRtmpEncoder</p> <p>... <- ResourcePoolHelper<RtmpPacket> <-</p> <p>描述:</p> <p>① 输入一个完整的 AAC 帧, 输出一连串 RtpPacket 对象。</p>
<p>... <- RtpCodec <- AACRtmpDecoder</p> <p>... <- ResourcePoolHelper<FrameImp> <-</p> <p>描述:</p> <p>① 输入一连串 RtpPacket 对象, 输出一个完整的 AAC 帧。</p>
<p>... <- AACRtmpDecoder <- AACRtmpEncoder</p> <p>... <- RtpInfo <-</p> <p>描述:</p> <p>① 输入一个完整的 AAC 帧, 输出一连串 RtpPacket 对象。</p>
<p>... <- RtmpCodec <- CommonRtmpDecoder</p> <p>... <- ResourcePoolHelper<FrameImp> <-</p> <p>描述:</p> <p>① 通用 RtmpPacket 对象解码器。</p> <p>② 输入 RtmpPacket, 输出一个完整的帧数据。</p>
<p>... <- CommonRtmpDecoder <- CommonRtmpEncoder</p> <p>... <- ResourcePoolHelper<RtmpPacket> <-</p> <p>描述:</p> <p>① 通用 RtmpPacket 对象解码器。</p> <p>② 将输入的媒体帧分割编码成一连串的 RtmpPacket 对象。</p> <p>③ 将编码生成的 RtmpPacket 包写入_rtmpRing。</p>
<p>... <- RtmpCodec <- CommonRtpDecoder</p> <p>... <- ResourcePoolHelper<FrameImp> <-</p>

<p>描述:</p> <ul style="list-style-type: none"> ① 通用 RtpPacket 对象解码器。 ② 将 RtpPacket 对象的负载部分追加到一块连续的缓冲区中。 ③ 当满一帧数据后, 就产生一帧数据, 触发回调。
<p>... <- CommonRtpDecoder <- CommonRtpEncoder ... <- RtpInfo <- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 通用 RtpPacket 对象编码器。 ② 将输入的媒体帧分割编码成一连串的 RtpPacket 对象。 ③ RtpPacket 对象在发送时的大小不超过一个 MTU 的大小。
<p>Factory</p> <p>描述:</p> <ul style="list-style-type: none"> ① 各种流媒体传输协议相关的辅助工厂方法。
<p>CodecId</p> <p>描述:</p> <ul style="list-style-type: none"> ① 编解码器类型 Id 枚举常量, 如 CodecH264、CodecH265、CodecAAC 等。
<p>TrackType</p> <p>描述:</p> <ul style="list-style-type: none"> ① 轨道码流的一级类型。
<p>CodecInfo</p> <p>描述:</p> <ul style="list-style-type: none"> ① 轨道码流的编码信息 (二级类型) 读取接口。
<p>... <- Buffer <- Frame CodecInfo <- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 帧类型的抽象接口。
<p>... <- Frame <- FrameImp</p> <p>描述:</p> <ul style="list-style-type: none"> ① 包含帧数据缓冲区、dts、pts 和 CodecId 的帧类型接口实现。
<p>... <- FrameInternal<></p> <p>描述:</p> <ul style="list-style-type: none"> ① 用于表示连续缓冲区内多帧数据分离后的多个帧。 ② 分离后的帧是有内置缓冲的拷贝, 还是无拷贝的指针引用, 由类的模板参数决定。
<p>ResourcePoolHelper<></p> <p>描述:</p> <ul style="list-style-type: none"> ① 循环资源池辅助类。

<p>FrameWriterInterface</p> <p>描述:</p> <ul style="list-style-type: none"> ① 帧写入器抽象接口类。
<p>FrameWriterInterface ◁- FrameWriterInterfaceHelper</p> <p>描述:</p> <ul style="list-style-type: none"> ① 帧写入器接口的回调通知模式实现。
<p>FrameWriterInterface ◁- FrameDispatcher</p> <p>描述:</p> <ul style="list-style-type: none"> ① 帧派发器。
<p>Frame ◁- FrameFromPtr</p> <p>描述:</p> <ul style="list-style-type: none"> ① 将指针指向的数据块包装成符合 Frame 接口标准的对象。
<p>... ◁- FrameWrapper</p> <p>描述:</p> <ul style="list-style-type: none"> ① 把 Buffer 对象包装成可缓存的 Frame 对象。
<p>... ◁- AudioTrackImp ◁- G711Track</p> <p>描述:</p> <ul style="list-style-type: none"> ① G711 音频轨道。
<p>... ◁- Sdp ◁- G711Sdp</p> <p>描述:</p> <ul style="list-style-type: none"> ① 合成 G711 类型的 Sdp 字符串。
<p>... ◁- FrameImp ◁- H264Frame</p> <p>描述:</p> <ul style="list-style-type: none"> ① H.264 帧缓冲区封装。
<p>... ◁- FrameFromPtr ◁- H264FrameNoCacheAble</p> <p>描述:</p> <ul style="list-style-type: none"> ① 无拷贝的连续缓冲区内部 H.264 帧引用信息。
<p>... ◁- VideoTrack ◁- H264Track</p> <p>描述:</p> <ul style="list-style-type: none"> ① H.264 视频轨道类。
<p>... ◁- Sdp ◁- H264Sdp</p> <p>描述:</p> <ul style="list-style-type: none"> ① 合成 H.264 类型的 Sdp 字符串。
<p>... ◁- RtmpCodec ◁- H264RtmpDecoder</p>

<p>... <- ResourcePoolHelper<H265Frame> <--- </p> <p>描述:</p> <ol style="list-style-type: none"> ① 包含 H.264 数据的 RTMP 包解码器。 ② 输入 RtmpPacket 对象。 ③ 输出 H264Frame 对象。
<p>... <- H264RtmpDecoder <-----H264RtmpEncoder</p> <p>... <- ResourcePoolHelper<RtmpPacket> <--- </p> <p>描述:</p> <ol style="list-style-type: none"> ① 包含 H.264 数据的 RTMP 包编码器。 ② 输入 H264Frame 对象。 ③ 输出 RtmpPacket 对象。
<p>... <- RtpCodec <-----H264RtpDecoder</p> <p>... <- ResourcePoolHelper<H265Frame> <--- </p> <p>描述:</p> <ol style="list-style-type: none"> ① 包含 H.264 数据的 RTP 包解码器。 ② 输入 RtpPacket 对象。 ③ 输出 H264Frame 对象。
<p>... <- H264RtpDecoder <---H264RtpEncoder</p> <p>... <- RtpInfo <----- </p> <p>描述:</p> <ol style="list-style-type: none"> ① 包含 H.264 数据的 RTP 包编码器。 ② 输入 H264Frame 对象。 ③ 输出 RtpPacket 对象。
<p>... <- FrameImp <- H265Frame</p> <p>描述:</p> <ol style="list-style-type: none"> ① H.265 帧缓冲区封装。
<p>... <- FrameFromPtr <- H265FrameNoCacheAble</p> <p>描述:</p> <ol style="list-style-type: none"> ① 无拷贝的连续缓冲区内部 H265 帧引用信息。
<p>... <- VideoTrack <- H265Track</p> <p>描述:</p> <ol style="list-style-type: none"> ① H265 视频轨道类。
<p>... <- Sdp <- H265Sdp</p> <p>描述:</p> <ol style="list-style-type: none"> ① 合成 H265 类型的 Sdp 字符串。
<p>... <- RtmpCodec <-----H265RtmpDecoder</p> <p>... <- ResourcePoolHelper<H265Frame> <--- </p> <p>描述:</p>

<ul style="list-style-type: none"> ① 包含 H.265 数据的 RTMP 包解码器。 ② 输入 RtmpPacket 对象。 ③ 输出 H265Frame 对象。
<pre>... <- H265RtmpDecoder <-----H265RtmpEncoder ... <- ResourcePoolHelper<RtmpPacket> <--- </pre> <p>描述:</p> <ul style="list-style-type: none"> ① 包含 H.265 数据的 RTMP 包编码器。 ② 输入 H265Frame 对象。 ③ 输出 RtmpPacket 对象。
<pre>... <- RtpCodec <-----H265RtpDecoder ... <- ResourcePoolHelper<H265Frame> <--- </pre> <p>描述:</p> <ul style="list-style-type: none"> ① 包含 H.265 数据的 RTP 包解码器。 ② 输入 RtpPacket 对象。 ③ 输出 H265Frame 对象。
<pre>... <- H265RtpDecoder <---H265RtpEncoder ... <- RtpInfo <----- </pre> <p>描述:</p> <ul style="list-style-type: none"> ① 包含 H.265 数据的 RTP 包编码器。 ② 输入 H265Frame 对象。 ③ 输出 RtpPacket 对象。
<pre>... <- AudioTrackImp <- OpusTrack</pre> <p>描述:</p> <ul style="list-style-type: none"> ① Opus 音频轨道描述信息。 ② 支持合成 OpusSdp 对象。
<pre>... <- Sdp <- OpusSdp</pre> <p>描述:</p> <ul style="list-style-type: none"> ① Opus 音频的 SDP。
<p>T_SPS</p> <p>描述:</p> <ul style="list-style-type: none"> ① NALU 中的 Sequence parameter set 解码后得到的结构
<p>T_PPS</p> <p>描述:</p> <ul style="list-style-type: none"> ① NALU 中的 Picture parameter set 解码后得到的结构
<p>T_HEVCVPS</p> <p>描述:</p> <ul style="list-style-type: none"> ① HEVC 的 NALU 中的 Video parameter set 解码后得到的结构

<p>... ◀- FrameDispatcher ◀- Track</p> <p>CodecInfo ◀----- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 媒体轨道描述信息获取接口类。 ② 支持输入帧派发。 ③ 支持编解码信息获取接口，由具体的媒体轨道类实现。
<p>... ◀- Track ◀- VideoTrack</p> <p>描述:</p> <ul style="list-style-type: none"> ① 视频轨道描述信息获取接口 ② 支持获取视频的宽高和 FPS。 ③ 不支持视频分层。 ④ 不支持多码率码流并行。
<p>... ◀- Track ◀- AudioTrack</p> <p>描述:</p> <ul style="list-style-type: none"> ① 音频轨道描述信息获取接口， ② 支持获取采样率、位宽和声道数量等。
<p>... ◀- AudioTrack ◀- AudioTrackImp</p> <p>描述:</p> <ul style="list-style-type: none"> ① 音频轨道描述信息访问接口实现。
<p>TrackSource</p> <p>描述:</p> <ul style="list-style-type: none"> ① Track 对象获取接口类。

表 5-8 ZLMediaKit/src/Extension 目录类型描述表

5.1.9. FMP4

ZLMediaKit/src/FMP4 目录类型描述表:

<p>... ◀- BufferString ◀- FMP4Packet</p> <p>描述:</p> <p>① FMP4 直播数据包。</p>
<p>... ◀- MediaSource ◀-----FMP4MediaSource</p> <p>... ◀- RingDelegate<FMP4Packet::Ptr> ◀-- </p> <p>... ◀- PacketCache<FMP4Packet> ◀----- </p> <p>描述:</p> <p>① FMP4 媒体源。</p> <p>② FMP4 是一个流式的封装（容器）格式，metadata 信息与数据都存放在一个个 moof、mdat 中。</p> <p>③ DASH 倾向于 FMP4，因为 FMP4 比 TS 更具灵活性。</p> <p>④ FMP4 容易实现多种清晰度码流间无缝切换，因为其切片策略可以自定义为以 GOP 边界对齐。</p>
<p>... ◀- MP4MuxerMemory ◀-----FMP4MediaSourceMuxer</p> <p>... ◀- MediaSourceEventInterceptor ◀---- </p> <p>描述:</p> <p>① FMP4 媒体源复用器。</p> <p>② 聚合了一个 FMP4MediaSource 对象 _media_src。</p> <p>③ 在内存中合成 FMP4 切片。</p> <p>④ 当遇到新的 I 帧或者超过 50ms 则 flush 切片到磁盘文件。</p>

表 5-9 ZLMediaKit/src/FMP4 目录类型描述表

5.1.10. Http

ZLMediaKit/src/Http 目录类型描述表:

<p>ts_segment</p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS ts 段描述信息。
<p>HlsParser</p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS m3u8 文件内容解析器。
<p>... <- HttpClientImp <--- HlsPlayer</p> <p>... <- PlayerBase <----- </p> <p>HlsParser <----- </p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS 播放器的接口实现。
<p>... <- PlayerImp<HlsPlayer, PlayerBase> <--- HlsPlayerImp</p> <p>... <- MediaSink <----- </p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS 播放器的特化实现。 ② 实现了 MediaSink 接口。 ③ 实现了帧缓存功能。
<p>HttpBody</p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP content 部分的抽象。
<p>HttpBody <- HttpFileBody</p> <p>描述:</p> <ul style="list-style-type: none"> ① 文件类型的 HTTP content。
<p>HttpBody <- HttpMultiFormBody</p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP MultiForm 方式提交的 HTTP content。
<p>HttpRequestSplitter <- HttpChunkedSplitter</p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP Chunk 数据分离器。
<p>std::map<string, variant, StrCaseCompare> <- HttpArgs</p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP 请求参数字典。

<p>② 支持合成 URL 编码规范的参数。</p>
<p>TcpClient ◀-----HttpClient HttpRequestSplitter ◀-- 描述: ① HTTP 客户端的实现。</p>
<p>...◀- TcpClientWithSSL<HttpClient> ◀- HttpClientImp 描述: ① 支持 HTTPS 的 HTTP 客户端的实现封装。 ② 比 HttpClient 更先进。</p>
<p>HttpClient 描述: ① HTTP 客户端 cookie 对象。</p>
<p>HttpClientStorage 描述: ① HTTP 客户端 cookie 全局保存器。</p>
<p>...◀- AnyStorage ◀---HttpServerCookie noncopyable ◀----- 描述: ① HTTP 服务器端的 cookie 对象, 用于保存 cookie 的一些信息。</p>
<p>RandStrGenerator 描述: ① cookie 随机字符串生成器。</p>
<p>HttpClientManager 描述: ① 用于 cookie 的生成以及过期管理。</p>
<p>...◀- HttpClientImp ◀- HttpDownloader 描述: ① HTTP 文件下载器。</p>
<p>HttpResponseInvokerImp 描述: ① HTTP 响应处理规则封装。</p>
<p>HttpFileManager 描述: ① 用于控制 HTTP 静态文件夹的访问权限。</p>

<p>... <- HttpClientImp <- HttpRequester</p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP 请求发送和响应处理。
<p>HttpRequestSplitter</p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP 请求分离器。
<p>... <- TcpSession <----- HttpSession</p> <p>FlvMuxer <----- </p> <p>HttpRequestSplitter <----- </p> <p>... <- WebSocketSplitter <-- </p> <p>描述:</p> <ul style="list-style-type: none"> ① HTTP 协议处理器。 ② 支持 WebSocket 协议。
<p>... <- HttpClientImp <- HttpTSPlayer</p> <p>strCoding</p> <p>描述:</p> <ul style="list-style-type: none"> ① 实现了字符串编码转换。 ② 实现了 UTF8 标准的 URL 编解码
<p>... <- ClientTypeImp</p> <p>描述:</p> <ul style="list-style-type: none"> ① 用于 TcpClient 派生类的数据发送拦截。 ② 拦截的目的是为了让此类的用户类可以在数据正式发送前，对数据按照其它协议（如 WebSocket）进行再封装。
<p>... <- HttpClientImp <----- HttpWsClient</p> <p>... <- WebSocketSplitter <- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 实现 WebSocket 客户端的握手协议。
<p>... <- WebSocketClient</p> <p>描述:</p> <ul style="list-style-type: none"> ① 可以将 TcpClient 的派生类包装为 WebSocket 客户端。
<p>SendInterceptor</p> <p>描述:</p> <ul style="list-style-type: none"> ① 发送数据截取器抽象接口。
<p><T> <----- TcpSessionTypeImp</p> <p>SendInterceptor <-- </p>

<p>描述:</p> <ul style="list-style-type: none"> ① 实现了 <code>TcpSession</code> 派生类发送数据的截取。 ② 主要用于发送数据前对数据按照 <code>WebSocket</code> 协议进行打包。
<p>TcpSessionCreator<></p> <p>描述:</p> <ul style="list-style-type: none"> ① 用于构造模板参数指定类型的基于 <code>TCP</code> 传输协议的会话。
<p>... <- WebSocketSessionBase</p> <p>描述:</p> <ul style="list-style-type: none"> ① <code>WebSocket</code> 协议会话基类。 ② 实现了 <code>WebSocket</code> 会话建立、通讯和关闭的基本流程。
<p>... <- WebSocketSession</p> <p>描述:</p> <ul style="list-style-type: none"> ① <code>WebSocket</code> 协议会话模板。 ② 支持通过模板参数构造类似 <code>RTMP over WebSocket</code> 的应用协议栈。
<p>WebSocketHeader</p> <p>描述:</p> <ul style="list-style-type: none"> ① <code>WebSocket</code> 协议消息头封装。
<p>... <- <code>BufferString</code> <- WebSocketBuffer</p> <p>描述:</p> <ul style="list-style-type: none"> ① <code>WebSocket</code> 协议收到的字符串类型缓存。
<p><code>WebSocketHeader</code> <- WebSocketSplitter</p> <p>描述:</p> <ul style="list-style-type: none"> ① <code>WebSocket</code> 协议消息解码器。 ② <code>WebSocket</code> 协议消息编码器。

表 5-10 ZLMediaKit/src/Http 目录类型描述表

5.1.11. Player

ZLMediaKit/src/Player 目录类型描述表:

PlayerImp<PlayerBase, PlayerBase> <- MediaPlayer 描述: ① 媒体传输协议拉流器实现。
TrackSource <- DemuxerBase 描述: ① 解复用器基类。 ② 播放器基类继承了此类, 重载了 <code>getDuration</code> 方法。
... <- DemuxerBase <- PlayerBase mINI <- 描述: ① 播放器基类。 ② 定义了播放器的控制接口方法。
<T><- PlayerImp 描述: ① 播放器控制接口模板化 (多态) 实现。
... <- PlayerBase <- Demuxer 描述: ① 包含一个音频和一个视频 Track。 ② 定义媒体包解复用后的 Track 对象的访问接口。
... <- MediaPlayer <- PlayerProxy MediaSourceEvent <- 描述: ① 媒体播放器拉流代理。

表 5-11 ZLMediaKit/src/Player 目录类型描述表

5.1.12. Pusher

ZLMediaKit/src/Pusher 目录类型描述表:

<p>PusherImp<PusherBase, PusherBase> ◁- MediaPusher</p> <p>描述:</p> <ul style="list-style-type: none"> ① 根据 url 参数动态创建支持特定传输协议的推流器作为委托对象。 ② 将实际的推流接口实现和事件回调处理转发给委托对象。 ③ 为使用者提供简单一致的推流控制接口。
<p>mINI ◁- PusherBase</p> <p>描述:</p> <ul style="list-style-type: none"> ① 推流器抽象类。
<p>... ◁- PusherImp</p> <p>描述:</p> <ul style="list-style-type: none"> ① Pusher 的泛型实现。 ② 支持通过模板参数编译时确定基类和事件委托类。

表 5-12 ZLMediaKit/src/Pusher 目录类型描述表

5.1.13. Rtmp

ZLMediaKit/src/Rtmp 目录类型描述表:

<p>AMFValue 描述: ① AMF 格式消息中的字段值封装。</p>
<p>AMFDecoder 描述: ① AMF 格式消息解码器。</p>
<p>FlvMuxer 描述: ① FLV 文件格式复用器。</p>
<p>FlvMuxer ◀- FlvRecorder 描述: ① FLV 格式录像文件合成器。</p>
<p>RtmpHandshake 描述: ① RTMP 协议握手消息需要的随机数据。</p>
<p>RtmpHeader 描述: ① RTMP 协议消息头。</p>
<p>Buffer ◀- RtmpPacket 描述: ① RTMP 协议消息的封装。</p>
<p>CodecInfo ◀- Metadata 描述: ① RTMP metadata 基类。</p>
<p>... ◀- Metadata ◀- TitleMeta 描述: ① RTMP 非音视频元数据。</p>
<p>RtmpRing 描述: ① 聚合了一个 RTMP 包的 RingBuffer。</p>

<pre>... <- RtmpRing <----- RtmpCodec ... <- FrameDispatcher <-- CodecInfo <----- </pre> <p>描述:</p> <ol style="list-style-type: none"> ① RTMP 协议消息编码器接口。
<pre>... <- Demuxer <- RtmpDemuxer</pre> <p>描述:</p> <ol style="list-style-type: none"> ① RTMP 协议解复用器。
<pre>... <- MediaSource <----- RtmpMediaSource ... <- RingDelegate<RtmpPacket::Ptr> <-- PacketCache<RtmpPacket> <----- </pre> <p>描述:</p> <ol style="list-style-type: none"> ① 纯粹的 RTMP 媒体源的抽象，与其它协议无关。 ② 处理 RTMP 关键三要素：metadata、config 帧和普通帧。 ③ 聚合了一个 List<RtmpPacket::Ptr>对象的环形缓存队列。
<pre>... <- RtmpMediaSource <----- RtmpMediaSourceImp Demuxer::Listener <----- MultiMediaSourceMuxer::Listener <-- </pre> <p>描述:</p> <ol style="list-style-type: none"> ① 聚合了一个 RtmpDemuxer 对象_demuxer，用于 RTMP 包解复用。 ② 聚合了一个 MultiMediaSourceMuxer 对象_muxer 用于其它协议转码。
<pre>... <- RtmpMuxer <----- RtmpMediaSourceMuxer ... <- MediaSourceEventInterceptor <-- </pre> <p>描述:</p> <ol style="list-style-type: none"> ① 聚合了一个 RtmpMediaSource 对象_media_src。 ② 将自身的 RtmpRing 的输入事件委托给_media_src 处理。 ③ 接受上级 MultiMuxerPrivate 对象的数据帧输入。 ④ 借助基类 RtmpMuxer 完成音视频帧的复用。
<pre>... <- MediaSinkInterface <- RtmpMuxer</pre> <p>描述:</p> <ol style="list-style-type: none"> ① 音视频帧的 RTMP 协议复用器。 ② 实现音视频帧的 RTMP 编码。
<pre>... <- PlayerBase <----- RtmpPlayer ... <- TcpClient <----- ... <- RtmpProtocol <-- </pre> <p>描述:</p> <ol style="list-style-type: none"> ① 实现了 RTMP 播放器协议处理部分的功能。

<p>... <- PlayerImp<RtmpPlayer, RtmpDemuxer> <- RtmpPlayerImp</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTMP 协议拉流客户端。 ② 聚合了一个 RtmpMediaSource 对象_rtmp_src;
<p>HttpRequestSplitter <- RtmpProtocol</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTMP 协议实现。
<p>... <- RtmpProtocol <- RtmpPusher</p> <p>... <- TcpClient <----- </p> <p>... <- PusherBase <----- </p> <p>描述:</p> <ul style="list-style-type: none"> ① RTMP 协议推流器。
<p>... <- TcpSession <-----RtmpSession</p> <p>... <- RtmpProtocol <----- </p> <p>MediaSourceEvent <----- </p> <p>描述:</p> <ul style="list-style-type: none"> ① RTMP 服务器端（通过 accept 操作创建的）会话。

表 5-13 ZLMediaKit/src/Rtmp 目录类型描述表

5.1.14. Rtp

ZLMediaKit/src/Rtp 目录类型描述表:

<p>Decoder</p> <p>描述:</p> <p>① MPEG2-TS 或 MPEG2-PS 流的解码器的接口定义。</p>
<p>FrameMerger</p> <p>描述:</p> <p>① 合并时间戳相同的帧。</p>
<p>DecoderImp</p> <p>描述:</p> <p>① MPEG2-TS 和 MPEG2-PS 流解码器实现。</p>
<p>... <- HttpRequestSplitter <- GB28181Process</p> <p>... <- RtpReceiver <----- </p> <p>... <- ProcessInterface <----- </p> <p>描述:</p> <p>① GB28181 数据 RTP 包处理。</p>
<p>ProcessInterface</p> <p>描述:</p> <p>① RTP 包处理接口定义。</p>
<p>Decoder <- PSDecoder</p> <p>描述:</p> <p>① MPEG2-PS 流解析器。</p>
<p>... <- MediaSinkInterface <- PSEncoder</p> <p>描述:</p> <p>① 定义 MPEG2-PS 容器格式的打包接口。</p>
<p>... <- PSEncoder <- PSEncoderImp</p> <p>描述:</p> <p>① 实现 MPEG2-PS 容器格式的打包。</p>
<p>... <- PacketCache<Buffer> <- RtpCache</p> <p>描述:</p> <p>① 缓存原始的 RTP 协议包。</p> <p>② 对缓存的包合并输出给回调函数。</p>
<p>... <- RtpCache <-----RtpCachePS</p>

<p>... <- PSEncoderImp <- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 继承 RtpCache 类的功能。 ② 实现了 PSEncoderImp 接口，实现 MPEG2-PS 流的编码。
<p>SockInfo <------ RtpProcess</p> <p>... <- MediaSinkInterface <------ </p> <p>... <- MediaSourceEventInterceptor <- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了一个 GB281818Process 对象_process。 ② 将输入的 RTP 包委托给_process 对象处理。
<p>... <- MediaSourceEvent <- RtpProcessHelper</p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了一个 RtpProcess 对象_process。 ② 监听 process 对象的媒体源事件。
<p>RtpSelector</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 分流器。 ② 根据 RTP 包的 ssrc 值，将数据分流给不同的 RtpProcess 对象处理。
<p>... <- MediaSinkInterface <- RtpSender</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 发送客户端。 ② 支持发送 GB28181 协议。
<p>RtpServer</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 服务器。 ② 可同时开启 TCP 和 UDP 监听。 ③ 其用途与 GB28181 有关。
<p>... <- TcpSession <------ RtpSession</p> <p>... <- RtpSplitter <------ </p> <p>... <- MediaSourceEvent <- </p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 会话。 ② 用于实现 RTP 代理（转发）功能。
<p>... <- HttpRequestSplitter <- RtpSplitter</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTSP 传输通道上的 RTP 包分离器。 ② 区分 RTSP 信令和 RTP 媒体包。

... ◀- HttpRequestSplitter ◀- TSsegment
描述: ① TS 包分割器。 ② 用于分割一个一个的 TS 包。
Decoder ◀- TSDecoder
描述: ① TS 解析器。

表 5-14 ZLMediaKit/src/Rtp 目录类型描述表

5.1.15. Rtp

ZLMediaKit/src/Rtsp 目录类型描述表:

<p>RtpRing</p> <p>描述:</p> <ul style="list-style-type: none"> ① 聚合了一个 RtpPacket 对象环形队列。
<p>ResourcePoolHelper ◁ RtpInfo</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 数据源信息。 ② 从内部的对象池中获取空闲对象，合成顺序编号的 RtpPacket。
<p>... ◁ TcpSession ◁ RtspSession</p> <p>... ◁ RtpSplitter ◁ </p> <p>RtpReceiver ◁ </p> <p>MediaSourceEvent ◁ </p> <p>描述:</p> <ul style="list-style-type: none"> ① RTSP 服务器端（通过 accept 操作创建的）会话。
<p>MultiCastAddressMaker</p> <p>描述:</p> <ul style="list-style-type: none"> ① 组播地址生成器。
<p>RtpMultiCaster</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 包组播器。
<p>PacketSortor</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 包排序缓存。
<p>RtpReceiver</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 包接收器。 ② 输入 TCP/UDP 层原始 RTP 包指针生成 RtpPacket 对象。 ③ 将生成的 RtpPacket 对象输入到聚合的 RTP 排序缓存。 ④ 音频包和视频包分开独立缓存。
<p>PayloadType</p> <p>描述:</p> <ul style="list-style-type: none"> ① RTP 协议的负载类型的枚举值定义。
<p>BufferRaw ◁ RtpPacket</p> <p>描述:</p>

① RTP 封装。
SdpTrack 描述： <ol style="list-style-type: none"> ① 描述 SDP 中的媒体信息。 ② 一个会话通常包含多种媒体，比如音频和视频。
SdpParser 描述： <ol style="list-style-type: none"> ① SDP 字符串解析器。 ② 提取会话的全部媒体信息保存到 SdpTrack 对象数组。
RtspUrl 描述： <ol style="list-style-type: none"> ① RTSP 协议的 URL 解析。
CodecInfo ◀- Sdp 描述： <ol style="list-style-type: none"> ① SDP 基类。
...◀- Sdp ◀- TitleSdp 描述： <ol style="list-style-type: none"> ① SDP 中去除音视频媒体描述的其它描述部分。 ② 描述的是 Session 的主题，而非媒体内容。
...◀- Demuxer ◀- RtspDemuxer 描述： <ol style="list-style-type: none"> ① 实现了 RTP 包的解复用。分解为多个独立的 Track 对象。
...◀- MediaSource ◀----- RtspMediaSource RingDelegate<RtpPacket::Ptr> ◀---- PacketCache<RtpPacket> ◀----- 描述： <ol style="list-style-type: none"> ① RTSP 媒体源的数据抽象。 ② 保存 SDP 和缓存 RTP 包。 ③ 任何推拉流过程都是先交换 SDP，再协商传输模式（TCP UDP 组播），最后一直发送 RTP 包就行了。
...◀- RtspMediaSource ◀----- RtspMediaSourceImp Demuxer::Listener ◀----- MultiMediaSourceMuxer::Listener ◀- 描述： <ol style="list-style-type: none"> ① RTSP 媒体源实现。 ② 聚合了一个 MultiMediaSourceMuxer 对象用于多协议转换。 ③ 聚合了一个 RtspDemuxer 对象用于 RTP 包的解复用。如果未使能协议

转换那么就不会将 RTP 包输入给解复用器，仅缓存在 this 对象的 PacketCache 链表中，作为媒体源转发。

... <- RtspMuxer <----- RtspMediaSourceMuxer

MediaSourceEventInterceptor <-----|

描述:

- ① RTSP 媒体源复用器。
- ② 聚合一个 RtspMediaSource 对象_media_src。
- ③ 聚合了一个 RtpRing 对象_rtpRing。
- ④ 连接_rtpRing 和_media_src。

... <- MediaSinkInterface <- RtspMuxer

描述:

- ① RTSP 协议复用器。
- ② 聚合了一个 RtpRing 对象_rtpRing。
- ③ 聚合了多个轨道类型的 RTP 编码器（环形缓冲派生类）。
- ④ 对输入的帧进行 RTP 打包并注入聚合的 RTP 编码器。
- ⑤ 保存输入 Track 的 SDP 信息。

... <- PlayerBase <----- RtspPlayer

TcpClient <-----|

RtspSplitter <-----|

RtpReceiver <-----|

描述:

- ① RTSP 协议拉流器。
- ② 仅负责拉取 RTP 协议封装的数据流，不负责音视频解码和渲染。

... <- PlayerImp<RtspPlayer, RtspDemuxer> <- RtspPlayerImp

描述:

- ① RTSP 媒体源播放流程实现。

... <- TcpClient <--- RtspPusher

RtspSplitter <-----|

PusherBase <-----|

描述:

- ① RTSP 协议推流器。

Buffer <- BufferRtp

描述:

- ① 用于 Rtp 包的缓存。

... <- TcpSession <--- RtspSession

RtspSplitter <-----|

RtpReceiver <-----|

MediaSourceEvent <-----|

描述:

① 服务器端 <code>accept</code> 得到的 RTSP 连接。
<code>... <- HttpRequestSplitter <- RtspSplitter</code> 描述： ① RTSP 协议分离器。 ② 用于 RTSP 消息的解析。
UDPServer 描述： ① UDP 接收服务器。 ② 用于 RTP 包的 UDP 方式传输。

表 5-15 ZLMediaKit/src/Rtsp 目录类型描述表

5.1.16. Record

ZLMediaKit/src/Record 目录类型描述表:

<p>HlsMaker</p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS 的切片文件生成器的控制接口。 ② 实现了 m3u8 索引文件的生成。
<p>HlsMaker ◁- HlsMakerImp</p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS 的切片文件生成器实现。 ② 聚合了一个 HlsMediaSource 对象
<p>MediaSource ◁- HlsMediaSource</p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS 直播媒体源实现。
<p>HlsCookieData</p> <p>描述:</p> <ul style="list-style-type: none"> ① HLS 直播时的一些统计信息，如播放器个数和流量计数等。 ② 此类作为 HttpCookieAttachment 类的一部分。
<p>MediaSourceEventInterceptor ◁- HlsRecorder</p> <p>... ◁- TsMuxer ◁----- </p> <p>描述:</p> <ul style="list-style-type: none"> ① 通过基类 TsMuxer 将输入的帧录制为 TS 文件。 ② 通过聚合的 HlsMakerImp 对象生成 m3u8 索引文件。
<p>MP4FileIO</p> <p>描述:</p> <ul style="list-style-type: none"> ① MP4 文件读写类的抽象接口。
<p>MP4FileIO ◁- MP4FileDisk</p> <p>描述:</p> <ul style="list-style-type: none"> ① MP4 文件的磁盘读写实现。 ② 借助 libmov 库实现 mp4 协议解析。
<p>MP4FileIO ◁- MP4FileMemory</p> <p>描述:</p> <ul style="list-style-type: none"> ① MP4 文件的内存读写实现。
<p>... ◁- MP4FileDisk ◁--MP4Demuxer</p> <p>TrackSource ◁----- </p>

<p>描述:</p> <ul style="list-style-type: none"> ① MP4 磁盘文件解复用器。
<p>... ◀- MediaSinkInterface ◀- MP4MuxerInterface</p> <p>描述:</p> <ul style="list-style-type: none"> ① MP4 复用器控制接口。
<p>... ◀- MediaMuxerInterface ◀- MP4Muxer</p> <p>描述:</p> <ul style="list-style-type: none"> ① 根据输入的帧生成 MP4 磁盘文件。
<p>... ◀- MP4MuxerInterface ◀- MP4MuxerMemory</p> <p>描述:</p> <ul style="list-style-type: none"> ① 根据输入的帧生成 MP4 内存文件。
<p>MediaSourceEvent ◀- MP4Reader</p> <p>描述:</p> <ul style="list-style-type: none"> ① 实现 MP4 文件的流化过程（策略）。 ② 把策略推迟到用户开播的时候，根据用户选择的媒体源传输协议和配置参数决定采用哪种文件读取器。
<p>... ◀- MediaSinkInterface ◀- MP4Recorder</p> <p>描述:</p> <ul style="list-style-type: none"> ① 将输入的帧录制为 MP4 文件。
<p>RecordInfo</p> <p>描述:</p> <ul style="list-style-type: none"> ① 录像文件描述信息。
<p>Recorder</p> <p>描述:</p> <ul style="list-style-type: none"> ① 录像控制接口封装。 ② 可创建 HlsRecorder 或 MP4Recorder。
<p>... ◀- MediaSinkInterface ◀- TsMuxer</p> <p>描述:</p> <ul style="list-style-type: none"> ① MPEG2-TS 流复用器抽象类。 ② 定义了 TS 流的输入接口方法，该方法用于输入原始数据包，合成 TSPacket 包，并将合成的包写入到 TSMediaSource 对象。 ③ 实现了帧缓存机制。

表 5-16 ZLMediaKit/src/Record 目录类型描述表

5.1.17. Shell

ZLMediaKit/src/Shell 目录类型描述表:

CMD ◀- CMD_media 描述: ① 实现 MP4 文件的流化过程 (策略)。
... ◀- TcpSession ◀- ShellSession 描述: ① 实现 MP4 文件的流化过程 (策略)。

表 5-27 ZLMediaKit/src/Shell 目录类型描述表

5.1.18. TS

ZLMediaKit/src/TS 目录类型描述表:

<p>... ◀ BufferRaw ◀ TSPacket</p> <p>描述:</p> <p>① MPEG2-TS 数据包。</p>
<p>.. ◀ MediaSource ◀ TSMediaSource</p> <p>RingDelegate<TSPacket::Ptr> ◀</p> <p>PacketCache<TSPacket> ◀</p> <p>描述:</p> <p>① MPEG2-TS 媒体源。</p> <p>② 借助 libmpeg 库实现 MPEG2-TS 流输出。</p>
<p>.. ◀ TsMuxer ◀ TSMediaSourceMuxer</p> <p>.. ◀ MediaSourceEventInterceptor ◀</p> <p>描述:</p> <p>① MPEG2-TS 媒体源复用器。</p> <p>② 创建并聚合 TSMediaSource 媒体源对象。</p> <p>③ 监听聚合的媒体源对象的事件。</p> <p>④ 接收 MediaSink 派生类 MultiMuxerPrivate 的数据输入。</p>

表 5-37 ZLMediaKit/src/TS 目录类型描述表

5.2. 源码构建

构建源码方法请阅读 ZLMediaKit 官方的 Wiki，其地址为：

<https://github.com/xia-chu/ZLMediaKit/wiki/快速开始>

以下是作者对原文的摘录：

ZLMediaKit 源码的获取方法：

不要使用 github 下载 zip 包的方式下载源码，务必使用 git 克隆 ZLMediaKit 的代码，因为 ZLMediaKit 依赖于第三方代码，zip 包不会下载第三方依赖源码。

正确的做法如下：

```
#国内用户推荐从同步镜像网站 gitee 下载
git clone --depth 1 https://gitee.com/xia-chu/ZLMediaKit
cd ZLMediaKit
#千万不要忘记执行这句命令
git submodule update --init
```

ZLMediaKit 源码在 Linux 平台下的构建方法：

- (1) 进入 “ZLMediaKit/build” 目录。
- (2) 输入 “cmake ..” 以生成 makefile 文件。
- (3) 输入 “make -j4” 可启动 4 进程并发构建。
- (4) 构建成功后，会在 “ZLMediaKit/release/linux/Debug” 目录下生成 MediaServer 程序。

ZLMediaKit 源码在 Windows 平台下的构建方法：

- (1) 直接在图形界面的 Windows 资源管理器中打开 “ZLMediaKit/” 文件夹。
- (2) 在这个录下打开 cmd 命令行或者 PowerShell 窗口，输入 “cmake CMakeList.txt” 以生成 ZLMediaKit.sln 解决方案文件。也可以通过 cmake-gui 工具选择 “ZLMediaKit/” 目录来生成.sln 文件。
- (3) 假设你已经安装了 VS 2019，直接双击 ZLMediaKit.sln 打开解决方案，构建名称为 MediaServer 的项目。
- (4) 构建成功后，会在 “ZLMediaKit/release/windows/Debug” 目录下生成 MediaServer.exe。

其它平台的构建方法请参考 ZLMediaKit 的 Wiki 原文。

5.3. 配置方法

MediaServer 程序的配置文件模板是 ZLMediaKit/conf/config.ini。

MediaServer 启动时，如果在其所在目录下找不到 config.ini，那么它会通过 mINI 类在其所在目录下自动生成一个默认的配置模板文件 config.ini，这个文件中没有配置文件模板中的那些注释。

新手若无特殊需求，采用程序自动生成的 config.ini 中的默认配置即可。

在程序运行前，可以将 ZLMediaKit/conf/config.ini 复制一份到 MediaServer 可执行程序所在的目录下，根据实际需求进行个性化配置。

config.ini 由多个节区 (section) 构成，每个节区有一个中括号标记的节区标题，主要节区内容描述如下：

节区名	内容描述
api	HTTP API 相关的配置。
ffmpeg	将外部 FFMPEG 可执行程序作为拉流代理程序时，FFMPEG 命令行相关配置。
general	各种直播协议的总开关、自动断流时间、合并发送开关等。
hls	HLS 直播协议相关配置。
hook	直播流程关键事件回调地址配置。 回调地址 (hook) 采用 RESTful 风格的 HTTP 接口。 回调的返回结果可以影响直播的正常流程。
http	HTTP 和 HTTPS 服务器相关配置。 ZLMediaKit 实现了 HTTP 和 HTTPS 服务器端协议。 ZLMediaKit 也实现了 HTTP 和 HTTPS 客户端请求模块。
multicast	基于 UDP 的 RTP 包组播地址配置。
record	MP4 录制相关配置。
rtmp	RTMP 直播协议相关配置，如监听端口、握手、保活、时间戳重写等。
rtp	RTP 协议相关配置，如与 MTU 有关的音视频包、视频包最大尺寸设置。
rtp_proxy	RTP 代理功能相关配置。
rtsp	RTSP 协议相关配置，如监听端口、握手和保活时间阈值等。
shell	用于诊断目的类似 shell 的命令解释功能配置，如监听端口、命令最大长度。

表 5-4 config.ini 的节区内容描述

配置文件模板 ZLMediaKit/conf/config.ini 中已经对各个配置节区内的配置项进行了详细的注释，本节不再作更多的解释。

5.4. 运行方法

运行方法请阅读 ZLMediaKit 官方的 Wiki，其地址为：

<https://github.com/xia-chu/ZLMediaKit/wiki/快速开始>

以下是作者对原文的摘录：

在 Linux 平台下的运行方法：

- (1) 进入 “ZLMediaKit/release/linux/Debug” 目录。
- (2) 输入 “./MediaServer -d &” 以守护进程模式启动。
- (3) 输入 “./MediaServer -h” 可了解启动参数。

在 Windows 平台下的运行方法：

- (1) 进入 “ZLMediaKit/release/windows/Debug” 目录。
- (2) 用鼠标左键双击 MediaServer 启动。
- (3) 你也可以在 cmd 或 PowerShell 中启动，通过 MediaServer -h 了解启动参数。

其它平台（如 Android、iOS、Mac OS）下的运行方法请参考 Wiki。

5.5. 顶层设计

5.5.1. 模块划分

以下是 ZLMediaKit 作者提供的架构图：

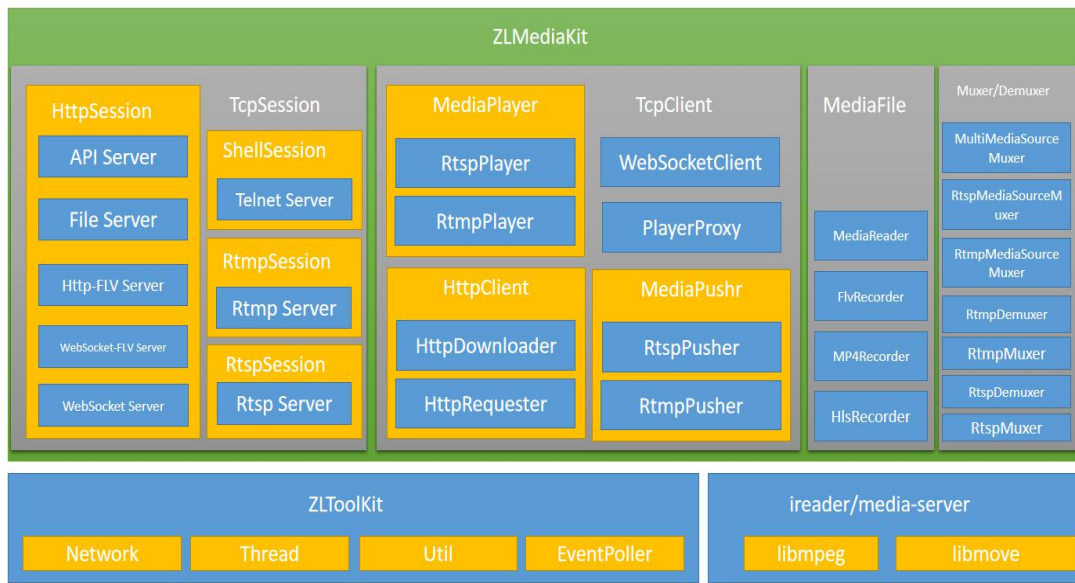


图 1 ZLMediaKit 架构图

根据上图，我们可以把 ZLMediaKit 的模块做如下分类：

- (1) 事件轮询器。
- (2) 线程池。
- (3) 传输协议服务器端会话类。
- (4) 媒体源复用器。
- (5) 媒体源解复用器。
- (6) 多协议媒体源复用器。
- (7) 媒体容器文件读写器。
- (8) 传输协议客户端。
- (9) 媒体推送器。
- (10) 媒体播放器。

如果以自顶向下的方式来分析 MediaServer 应用程序的设计，那么所有模块可以构成一棵关系树，树的根节点就是应用程序。

顶层模块是指模块层次关系树中距离根节点较近的那些模块。

以下是本书作者绘制的 MediaServer 顶层模块分布图：

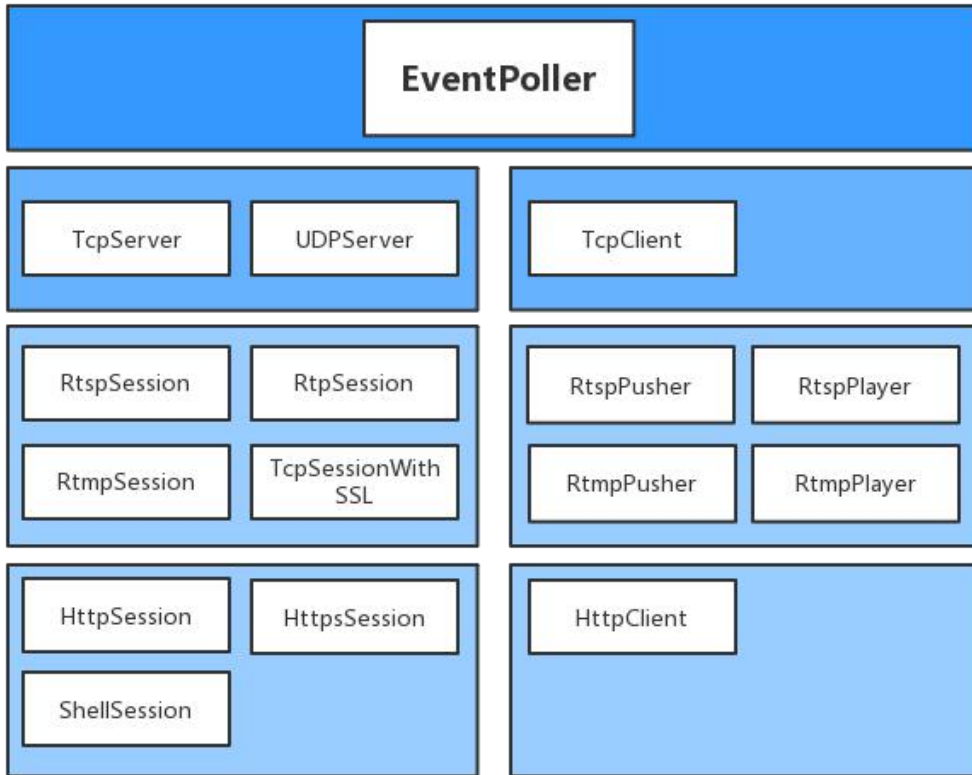


表 5-5 MediaServer 程序顶层模块分布图

5.5.2. 线程划分

MediaServer 进程包含了五类常驻线程。

常驻线程是指从被创建到退出进程阶段之前一直存在的线程。

我们将常驻线程应有的职责与功能称为它们的“职能”。

以下是 MediaServer 进程的常驻线程的职能描述表：

线程入口名称	线程职能
main	(1) 分析和设置进程的运行环境。 (2) 启动日志机制。 (3) 加载配置文件和 SSL 证书。 (4) 根据配置启动各种协议栈。 (5) 等待操作系统的退出信号。
AsyncLogWriter::run	(1) 等待 LogContext 写入事件。 (2) 将 LogContext 队列交换到临时变量 tmp。 (3) 通知所有 LogChannel 对象将 tmp 中的日志写入到目标设备。
initMillisecondThread	(1) 进程流逝时间统计。 (2) 防止时间戳回退。 (3) 提供毫秒级精度的参考时钟。
由 EventPollerPool 创建的 M 个 EventPoller::runLoop	(1) 等待和处理 I/O 事件。 (2) 延时执行任务。 (3) 通过事件驱动各种媒体传输协议处理。
由 WorkerThreadPool 创建的 N 个 EventPoller::runLoop	(1) 等待其它线程投递任务。 (2) 同步执行任务队列中的任务。

表 5-6 MediaServer 进程的常驻线程职能描述表

在上述表格中，数量符号 M 和 N 在默认值为 CPU 逻辑核心数，用户可通过 setPoolSize 方法修改默认值，其它未明确说明数量的线程入口只有单一实例。

5.5.3. 启动流程

本人将 MediaServer 进程的启动过程分为几个阶段。

MediaServer 进程的每个启动阶段的详细流程如下表所示：

启动阶段	详细流程
设置环境	<ol style="list-style-type: none"> (1) 分析命令行参数。 (2) 创建日志通道。 (3) 启动守护机制。 (4) 启动崩溃捕获机制。 (5) 启动日志异步写入线程。 (6) 加载 ini 配置文件。 (7) 加载 SSL 证书。 (8) 读取各种协议的服务端口配置。
启动服务	<ol style="list-style-type: none"> (1) 为支持的协议栈预创建用于底层传输的 TcpServer 对象,但不立即开始监听。 (2) 根据协议服务端口的配置值,创建该类协议服务端会话的类厂并开始监听 TCP 端口。
等待信号	<ol style="list-style-type: none"> (1) 构造静态退出信号量 sem。 (2) 安装 SIGINT 信号处理函数。 (3) 一直等待 SIGINT 信号.....
退出进程	<ol style="list-style-type: none"> (1) 收到 SIGINT 信号,减少退出信号量 sem 的计数。 (2) 退出进程。

表 5-7 MediaServer 进程的启动流程

5.6. 底层原理

5.6.1. 网络 I/O

ZLMediaKit 框架与网络 I/O 密切相关的源码目录如下：

- ZLMediaKit/3rdpart/ZLToolKit/src/**Network**
- ZLMediaKit/3rdpart/ZLToolKit/src/**Poller**

网络 I/O 事件处理模型封装在 Poller 目录下的 EventPoller 类中。

与套接字相关的操作细节主要封装在 Network 目录下的 Socket、TcpServer、TcpSession 和 TcpClient 类中。

对 Linux 平台，EventPoller 类采用 epoll_wait 函数来等待系统的网络 I/O 事件通知。

对 Windows 平台，EventPoller 类采用 select 函数来等待系统的网络 I/O 事件通知。

EventPoller 属于简单通用 I/O 事件通知机制，网上的资料很多，作者不重复讲解。

本节我们从 TcpServer 和 TcpClient 入手，分析 ZLMediaKit 独有的网络通讯流程。作者将前后文衔接作用的关键符号用粗体进行了标记。

TcpServer 接受客户端 TCP 连接过程代码片段（省略了细节）：

```
TcpServer::TcpServer(const EventPoller::Ptr &poller = nullptr) {
    _poller = poller ? poller : EventPollerPool::Instance().getPoller();
    _socket = createSocket();
    _socket->setOnAccept(bind(&TcpServer::onAcceptConnection_I, this, placeholders::_1));
}

void TcpServer::onAcceptConnection_I(const Socket::Ptr &sock) {
    onAcceptConnection(sock);
}

void TcpServer::onAcceptConnection(const Socket::Ptr &sock) {
    //创建由模板参数决定类型的服务会话实例指针封装器 helper。
    auto helper = _session_alloc(shared_from_this(), sock); //基于新 Socket 对象构建新会话。
    auto &session = helper->session(); //服务会话实例指针封装在 helper 对象之中。
    //会话接收数据事件
    sock->setOnRead([weak_session](const Buffer::Ptr &buf, struct sockaddr *, int) {
        strong_session->onRecv(buf); //TcpSession 派生类对象接收数据的开端
    });
}

//TcpServer 类聚合了一个 Socket 对象。
//main 函数调用了 TcpServer::start。
//TcpServer::start 方法调用了 Socket::listen。
bool Socket::listen(const SockFD::Ptr &sock){
    _poller->addEvent(sock->rawFd(), Event_Read | Event_Error, [...](int event) {
```

```

        strong_self->onAccept(strong_sock, event);
    });
}

```

TcpSession 是一个抽象类，它定义了数据接收的模板方法 OnRecv。

RtspSession 派生自 TcpSession。

RtspSession 类的数据接收过程代码片段：

```

void RtspSession::onRecv(const Buffer::Ptr &buf) {
    _bytes_usage += buf->size();
    if (_on_recv) {
        _on_recv(buf); //将 HTTP poster 收到的数据转发给已有的 HTTP getter 对象处理。
    } else {
        input(buf->data(), buf->size()); //调用基类的处理方法。
    }
}

//HttpRequestSplitter 是 RtspSession 的基类。
void HttpRequestSplitter::input(const char *data,uint64_t len) {
    //解析 HTTP 请求
}

```

RtspSession 类的数据发送过程代码片段：

```

void RtspSession::handleReq_Options(const Parser &parser) {
    sendRtspResponse("200 OK",{"Public" , "OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE,
ANNOUNCE, RECORD, SET_PARAMETER, GET_PARAMETER"});
}

//省略了细节的伪代码
bool RtspSession::sendRtspResponse(const string &res_code, const StrCaseMap &header_const,
const string &sdp, const char *protocol){
    auto header = header_const;
    header.emplace("CSeq",StrPrinter << _cseq);
    header.emplace("Server",SERVER_NAME);
    header.emplace("Date",dateStr());
    if(!sdp.empty()){
        header.emplace("Content-Length",StrPrinter << sdp.size());
        header.emplace("Content-Type","application/sdp");
    }
    _StrPrinter printer;
    printer << protocol << " " << res_code << "\r\n";
    for (auto &pr : header){
        printer << pr.first << ": " << pr.second << "\r\n";
    }
    if(!sdp.empty()){
        printer << "\r\n";
    }
}

```

```
    }
    return send(std::make_shared<BufferString>(std::move(printer))) > 0 ;
}

int RtspSession::send(Buffer::Ptr pkt){
    _bytes_usage += pkt->size();
    return TcpSession::send(std::move(pkt));
}

int SocketHelper::send(Buffer::Ptr buf) {
    return _sock->send(std::move(buf), nullptr, 0, _try_flush); //委托给套接字发送
}

//此处以 TCP 套接字为例
int Socket::send(Buffer::Ptr buf, struct sockaddr *addr, socklen_t addr_len, bool try_flush){
    auto size = buf ? buf->size() : 0;
    {
        LOCK_GUARD(_mtx_send_buf_waiting);
        _send_buf_waiting.emplace_back(buf); //将用户数据投递到一级发送缓存。
    }
    if(try_flush){
        if (_sendable) { //该 socket 可写
            return flushData(sock, false) ? size : -1;
        }
    }
    return size;
}

//请读者考虑：
//（1）是否可以在不同线程上下文中同时触发 Socket::flushData 方法？
//（2）_on_flush()回调是否可能在不同线程上下文中同时触发？
//（3）一级缓存中的数据经过多级转移，能否按照最初投递的顺序发送给客户端？
bool Socket::flushData(const SockFD::Ptr &sock, bool poller_thread) {
    decltype(_send_buf_sending) send_buf_sending_tmp;
    {
        //将二级缓存全部转移线程安全的局部变量
        LOCK_GUARD(_mtx_send_buf_sending);
        if(!_send_buf_sending.empty()){
            send_buf_sending_tmp.swap(_send_buf_sending);
        }
    }

    if (send_buf_sending_tmp.empty()) {
        _send_flush_ticker.resetTime();
    }
}
```

```
do {
    {
        //二级发送缓存为空，那么我们接着消费一级缓存中的数据
        LOCK_GUARD(_mtx_send_buf_waiting);
        if (!_send_buf_waiting.empty()) {
            //把一级缓存中数据放置到二级缓存中并清空
            send_buf_sending_tmp.emplace_back(std::make_shared<BufferList>(_send_buf_waiting));
            break;
        }
    }
    //如果一级缓存也为空,那么说明所有数据均写入 socket 了
    if (poller_thread) { //当时为空，离开了临界区，此刻仍然一定为空吗？
        //假如此刻一级缓存不为空，关闭监听可写事件是明智的吗？
        stopWriteAbleEvent(sock);
        onFlushed(sock); //通知用户所有数据均已写入 socket 了。
    }
    return true;
} while (0);
}

int fd = sock->rawFd();
bool is_udp = sock->type() == SockNum::Sock_UDP;
while (!send_buf_sending_tmp.empty()) {
    //一次取出一个缓冲区，进行发送。
    auto &packet = send_buf_sending_tmp.front();
    int n = packet->send(fd, _sock_flags, is_udp); //委托给缓冲区对象发送？
    if (n > 0) {
        if (packet->empty()) {
            //全部发送成功
            send_buf_sending_tmp.pop_front();
            continue;
        }
        //部分发送成功
        if (!poller_thread) {
            //该函数不是 poller 线程触发的，需要通过 poller 监听 sock 的可写事件。
            startWriteAbleEvent(sock);
        }
        break;
    }
}

//一个都没发送成功
int err = get_uv_error(true);
if (err == UV_EAGAIN) {
```

```
//等待下一次发送
if (!poller_thread) {
    //该函数不是 poller 线程触发的，需要通过 poller 监听 sock 的可写事件。
    startWriteAbleEvent(sock);
}
break;
}
//其他错误代码，发生异常
onError(sock);
return false;
}

//二级缓存已经全部发送完毕，说明该 socket 还可写，我们尝试继续写
//如果是 poller 线程，再次重入此函数，其他线程可能调用了 send 函数又有新数据。
return poller_thread ? flushData(sock, poller_thread) : true;
}

//缓冲区列表如何发送数据？
int BufferList::send(int fd,int flags,bool udp) {
    while (_remainSize && send_l(fd,flags,udp) != -1); //循环发送内部数据
    ...
}

//以 TCP 套接字为例
int BufferList::send_l(int fd, int flags,bool udp) {
    int n;
    do {
        struct msghdr msg;
        msg.msg_name = NULL;
        msg.msg_namelen = 0;
        msg.msg_iov = &(_iovec[_iovec_off]);
        msg.msg_iovlen = _iovec.size() - _iovec_off;
        int max = IOV_MAX;
        if(msg.msg_iovlen > max){
            msg.msg_iovlen = max;
        }
        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        msg.msg_flags = flags;
        n = sendmsg(fd,&msg,flags); //这是一个 Linux 系统 API，Windows 可用 sendto 模拟。
    } while (-1 == n && UV_EINTR == get_uv_error(true));
}
```

TcpClient 实现了 TCP 客户端的通讯机制。

RtspPusher 派生自 TcpClient。

RtspPusher 的连接过程代码片段(省略了次要细节)：

```
void RtspPusher::publish(const string &url_str) {
    teardown();
    _url = url_str;
    _publish_timer.reset(new Timer(publish_timeout_sec, [weak_self]() {
        strong_self->onPublishResult(SocketException(Err_timeout, "publish rtsp timeout"), false);
        return false;
    }, getPoller()));
    startConnect(url._host, url._port, publish_timeout_sec);
}

void TcpClient::startConnect(const string &url, uint16_t port, float timeout_sec) {
    setSock(createSocket());
    getSock()->connect(url, port, [weakSelf](const SocketException &err) {
        strongSelf->onSocketConnect(err);
    }, timeout_sec, _net_adapter.data());
}

void Socket::connect(const string &url, uint16_t port, onErrCB con_cb_in, float timeout_sec, const
string &local_ip, uint16_t local_port) {
    //重置当前 socket
    closeSock();
    auto con_cb = [con_cb_in, weak_self](const SocketException &err) {
        con_cb_in(err);
    };
    auto async_con_cb = std::make_shared<function<void(int)>>([weak_self, con_cb](int sock) {
        auto sock_fd = strong_self->makeSock(sock, SocketNum::Socket_TCP);
        weak_ptr<SocketFD> weak_sock_fd = sock_fd;
        //监听该 socket 是否可写，可写表明已经连接服务器成功
        int result = strong_self->poller->addEvent(sock, Event_Write, [weak_self, weak_sock_fd,
con_cb](int event) {
            //socket 可写事件，说明已经连接服务器成功
            strong_self->onConnected(strong_sock_fd, con_cb);
        });
    });

    auto poller = _poller;
    weak_ptr<function<void(int)>> weak_task = async_con_cb;

    ThreadPool::Instance().getExecutor()->async([url, port, local_ip, local_port, weak_task,
poller]() {
        //阻塞式 dns 解析放在后台线程执行
        int sock = SocketUtil::connect(url.data(), port, true, local_ip.data(), local_port);
    });
}
```

```
poller->async([sock, weak_task]() {
    auto strong_task = weak_task.lock();
    if (strong_task) {
        (*strong_task)(sock); //在 poller 线程执行 onConnected 回调
    } else {
        CLOSE_SOCKET(sock);
    }
});

_async_con_cb = async_con_cb; //标记回调函数。
}

int SocketUtil::connect(const char *host, uint16_t port, bool bAsync, const char *localIp, uint16_t
localPort) {
    int sockfd = socket(addr.sa_family, SOCK_STREAM, IPPROTO_TCP);
    bindSock(sockfd, localIp, localPort);
    if (::connect(sockfd, &addr, sizeof(struct sockaddr)) == 0) {
        //同步连接成功
        return sockfd;
    }
    if (bAsync && get_uv_error(true) == UV_EAGAIN) {
        //异步连接成功
        return sockfd;
    }
    close(sockfd);
    return -1;
}

void Socket::onConnected(const SocketFD::Ptr &sock, const onErrCB &cb) {
    //先删除之前的可写事件监听
    _poller->delEvent(sock->rawFd());
    attachEvent(sock, false); //绑定读写事件回调
    sock->setConnected();
    //连接成功
    cb(err); //此回调会转发调用到 TcpClient::onSockConnect
}

//在 TcpClient::startConnect 中设置到 Socket 对象的回调
void TcpClient::onSockConnect(const SocketException &ex) {
    getSocket()->setOnRead([weakSelf, sock_ptr](const Buffer::Ptr &pBuf, struct sockaddr *, int) {
        strongSelf->onRecv(pBuf); //RtspPusher 数据接收过程的开端
    });
    onConnect(ex); //连接服务器结果回调
}
}
```



```

void RtspPusher::onConnect(const SockException &err) {
    if (err) {
        onPublishResult(err, false);
        return;
    }
    sendAnnounce();
}

```

RtspPusher 的数据发送过程代码片段:

```

void RtspPusher::sendAnnounce() {
    _on_res_func = std::bind(&RtspPusher::handleResAnnounce, this, placeholders::_1);
    sendRtspRequest("ANNOUNCE", _url, {}, src->getSdp());
}

void RtspPusher::sendRtspRequest(const string &cmd, const string &url, const
std::initializer_list<string> &header,const string &sdp ) {
    sendRtspRequest(cmd, url, header_map, sdp);
}

void RtspPusher::sendRtspRequest(const string &cmd, const string &url,const StrCaseMap
&header_const,const string &sdp ) {
    auto header = header_const;
    header.emplace("CSeq", StrPrinter << _cseq++);
    header.emplace("User-Agent", SERVER_NAME);
    if (!_session_id.empty()) {
        header.emplace("Session", _session_id);
    }

    _StrPrinter printer;
    printer << cmd << " " << url << " RTSP/1.0\r\n";
    for (auto &pr : header) {
        printer << pr.first << ": " << pr.second << "\r\n";
    }

    printer << "\r\n";

    if (!sdp.empty()) {
        printer << sdp;
    }
    SocketHelper::send(std::move(printer));
}

int SocketHelper::send(string buf) {
    return send(std::make_shared<BufferString>(std::move(buf)));
}

int SocketHelper::send(Buffer::Ptr buf) {
    return _sock->send(std::move(buf), nullptr, 0, _try_flush);
}

```

```

}
int Socket::send(Buffer::Ptr buf, struct sockaddr *addr, socklen_t addr_len, bool try_flush){
    //后续过程在 RtspSession 的数据发送过程中已有分析
}

```

RtspPusher 的数据接收过程代码片段:

```

void Socket::onConnected(const SockFD::Ptr &sock, const onErrCB &cb) {
    //先删除之前的可写事件监听
    _poller->delEvent(sock->rawFd());
    attachEvent(sock, false);
}

bool Socket::attachEvent(const SockFD::Ptr &sock, bool is_udp) {
    int result = _poller->addEvent(sock->rawFd(), Event_Read | Event_Error | Event_Write,
[weak_self,weak_sock,is_udp](int event) {
        if (event & Event_Read) {
            strong_self->onRead(strong_sock, is_udp);
        }
        if (event & Event_Write) {
            strong_self->onWriteAble(strong_sock);
        }
        if (event & Event_Error) {
            strong_self->onError(strong_sock);
        }
    });
}

int Socket::onRead(const SockFD::Ptr &sock, bool is_udp) {
    while (_enable_rcv) {
        do {
            nread = recvfrom(sock_fd, data, capacity, 0, &addr, &len);//系统 API
        } while (-1 == nread && UV_EINTR == get_uv_error(true));

        //触发回调
        LOCK_GUARD(_mtx_event);
        _on_read(_read_buffer, &addr, len);//TcpClient::onSockConnect 设置 TcpClient::onRecv。
    }
    return 0;
}

void RtspPusher::onRecv(const Buffer::Ptr &buf){
    input(buf->data(), buf->size()); //由基类 RtspSplitter 的基类 HttpRequestSplitter 处理。
}

```

```
void HttpRequestSplitter::input(const char *data,uint64_t len) {
    _content_len = onRecvHeader(header_ptr, header_size);//通知派生类收到 HTTP 头。
    //已经找到 http 头了
    if(_content_len > 0){
        //收到 content 数据，并且接受 content 完毕
        onRecvContent(ptr,_content_len); //通知派生类收到 HTTP 内容。
        _remain_data_size -= _content_len;
        ptr += _content_len;
        //content 处理完毕,后面数据当做请求头处理
        _content_len = 0;
        _remain_data.clear();
        return;
    }
    //_content_len < 0;数据按照不固定长度 content 处理
    onRecvContent(ptr,_remain_data_size);//消费掉所有剩余数据
    _remain_data.clear();
}

int64_t RtspSplitter::onRecvHeader(const char *data, uint64_t len) {
    if(_isRtpPacket){
        onRtpPacket(data,len); //采用 TCP 传输 RTP 包时会进入此分支。
        return 0;
    }
    _parser.Parse(data);
    auto ret = getContentLength(_parser);
    if(ret == 0){
        onWholeRtspPacket(_parser); //收到来自服务器的 RTSP 包
        _parser.Clear();
    }
    return ret;
}

void RtspSplitter::onRecvContent(const char *data, uint64_t len) {
    _parser.setContent(string(data,len));
    onWholeRtspPacket(_parser);//收到来自服务器的 RTSP 包
    _parser.Clear();
}

void RtspPusher::onWholeRtspPacket(Parser &parser) {
    decltype(_on_res_func) func;
    _on_res_func.swap(func); //根据当前状态预置的响应处理函数指针
    if (func) {
        func(parser);
    }
    parser.Clear();
}
```

```
}
```

5.6.2. 线程池

ZLMediaKit 框架的线程池类型有如下两种:

- 工作线程常驻的 `WorkThreadPool` 类。
- 工作线程执行完任务就退出的 `ThreadPool` 类。

在 `MediaServer` 项目中,主要用到的是工作线程实例固定的 `WorkThreadPool` 类,以静态单例模式创建 `WorkThreadPool` 类实例。`ThreadPool` 的通常用法是根据异步任务需要,动态构造一个指定数量工作线程的实例,然后向这个实例投递需要异步执行的任务项。

`WorkThreadPool` 类创建了多个 `EventPoller` 对象。每个 `EventPoller` 对象都会创建独立的工作线程,在工作线程中通过 `runLoop` 方法来实现异步任务队列中任务项的分发执行。

`WorkThreadPool` 默认创建的 `EventPoller` 对象个数为 CPU 逻辑核心数量。

我们必须在 `WorkThreadPool` 单例构造之前,通过其 `setPoolSize` 静态方法来修改线程池的固定线程数量。`WorkThreadPool::Instance` 方法一旦被调用,线程池的线程数量就固定下来了,此后再调用 `setPoolSize` 就没有意义了。

采用 `WorkThreadPool` 异步执行阻塞式任务的代码片段:

```
WorkThreadPool::Instance().getExecutor()->async([]() {  
    //执行一些阻塞式的任务,例如:  
    usleep(500); //让当前线程睡眠 500ms  
});
```

5.6.3. 文件 I/O

ZLMediaKit 框架与文件系统 I/O 密切相关的源码目录如下：

➤ ZLMediaKit/3rdpart/ZLToolKit/src/Util

ZLMediaKit 没有采用操作系统的异步 I/O 机制读写文件，而是采用同步方式直接读写文件，通过将文件读写代码封装成任务的方式投递给线程池来实现异步文件 I/O。知名的异步 I/O 库 libuv 也是采用类似的方式来实现文件异步 I/O 的。

文件创建的代码片段：

```
std::shared_ptr<FILE> HlsMakerImp::makeFile(const string &file, bool setbuf) {
    File::create_file(file.data(), "wb"); //此处省略了次要细节。
}

FILE *File::create_file(const char *file, const char *mode) {
    std::string path = file;
    if (path[path.size() - 1] != '/') {
        ret = fopen(file, mode);
    }
    return ret;
}
```

文件读取的代码片段：

```
Buffer::Ptr HttpFileBody::readData(uint32_t size) {
    if(!_map_addr){
        //fread 模式
        int iRead;
        auto ret = _pool.obtain();
        ret->setCapacity(size + 1);
        do{
            iRead = fread(ret->data(), 1, size, _fp.get()); //以同步的方式读取
        }while(-1 == iRead && UV_EINTR == get_uv_error(false));

        if(iRead > 0){
            //读到数据了
            ret->setSize(iRead);
            _offset += iRead;
            return ret;
        }
        //读取文件异常，文件真实长度小于声明长度
        _offset = _max_size;
        return nullptr;
    }

    //mmap 模式
```

```
auto ret = std::make_shared<BufferMmap>(_map_addr,_offset,size);
_offset += size;
return ret;
}
```

文件写入的代码片段:

```
void HlsMaker::makeIndexFile(bool eof) {
    onWriteHls(m3u8.data(), m3u8.size());//此处省略了次要细节。
}

void HlsMakerImp::onWriteHls(const char *data, int len) {
    auto hls = makeFile(_path_hls);
    if (hls) {
        fwrite(data, len, 1, hls.get());
    }
}
```

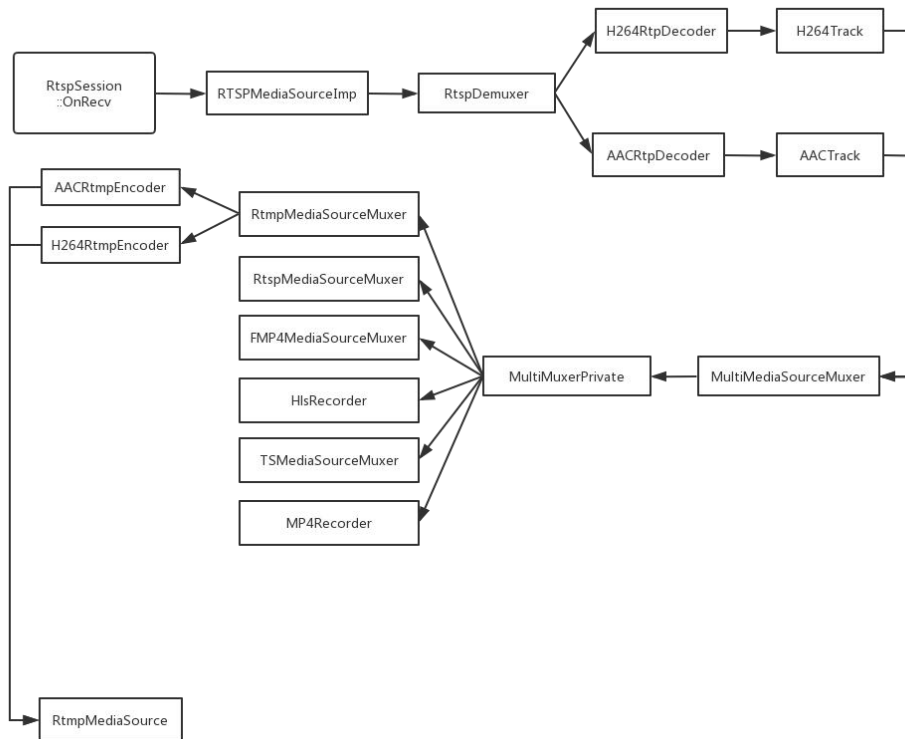
5.7. 媒体接入流程

5.7.1. RTMP 接入

TODO:

5-2 RTMP 协议接入流程

5.7.2. RTSP 接入



5-3 RTSP 转 RTMP 流程

5.8. 媒体缓冲流程

5.9. 媒体转发流程

5.9.1. RTMP 直播

5.9.2. RTSP 直播

5.10. 媒体推送流程

5.11. 媒体播放流程

5.12. 媒体录制流程

5.13. 系统集成方案

第 6 章 关于

6.1. 关于作者

2020 年 5 月，作者结束了外出务工的职业生涯，决心成为一名专业技术作家和一名独立开发者。

2020 年 12 月，作者创立了 ADM 实验室，并开办了自己的个人网站 codemi.net，尝试通过这个窗口来发布自己的技术书籍和研究成果。

作者计划在未来十年内创作十本技术书籍。这十本书，可以帮助经验不足的程序员快速成长，帮助中小企业降低技术预研成本、加快研发进度和提升产品质量。

目前，作者正在独立开发一套多媒体应用平台。该平台的目标是让部分多媒体应用开发者能够使用一套共享的平台和大量可复用的组件，避免重复劳动，提高软件生产效率。

作者计划在未来的一段时间内，独立定制一款专门用于虚拟现实和增强现实领域的实时操作系统，让每个程序员足不出户就能顺畅地工作和交流，开创一个以虚拟现实协作平台为中心的远程办公时代。

作者的最新版本技术书籍发布 QQ 群：**462178798**

作者的最新研究成果发布 QQ 群：**462178798**

读者之间的技术交流 QQ 群：**462178798**