codemi.net

实时音视频直播系统

—Linux 版直播服务器

第一版

颜贤时 著

作者的个人网站：codemi.net

目录

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成什么样子，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到在人的眼睛或耳朵的延迟时间稳定在 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**）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。在用户使用直播

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。作者开发的直播客户端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。

章节 2.1-2.13 属于必修内容，章节 2.14-2.22 属于选修内容。

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。

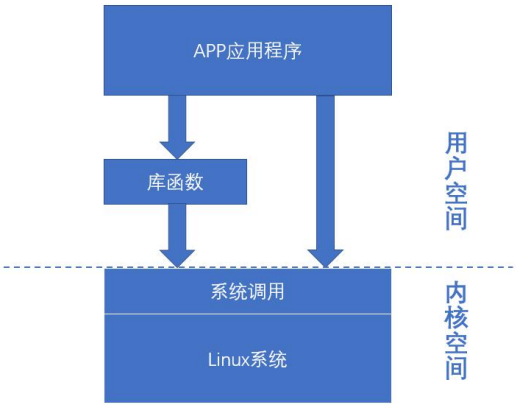


表 2-1 Linux 应用基本原理

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：

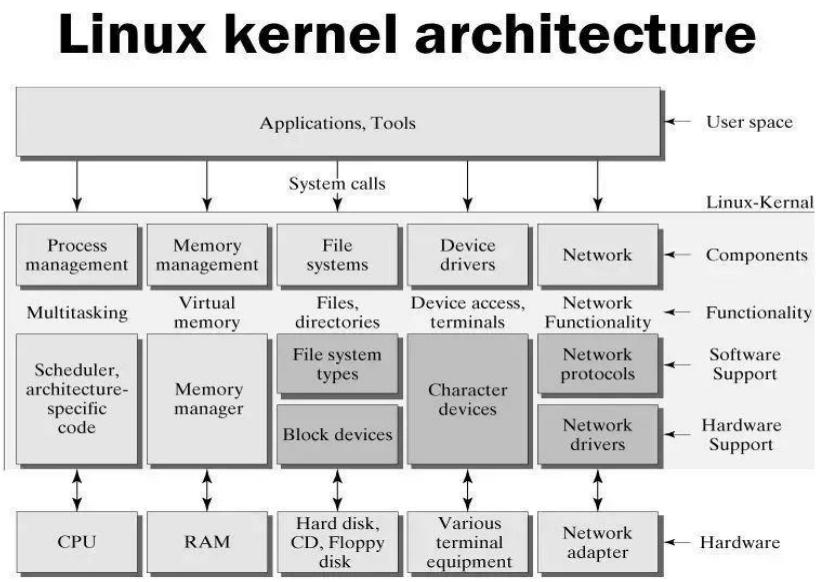


表 2-2 Linux 内核架构

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。

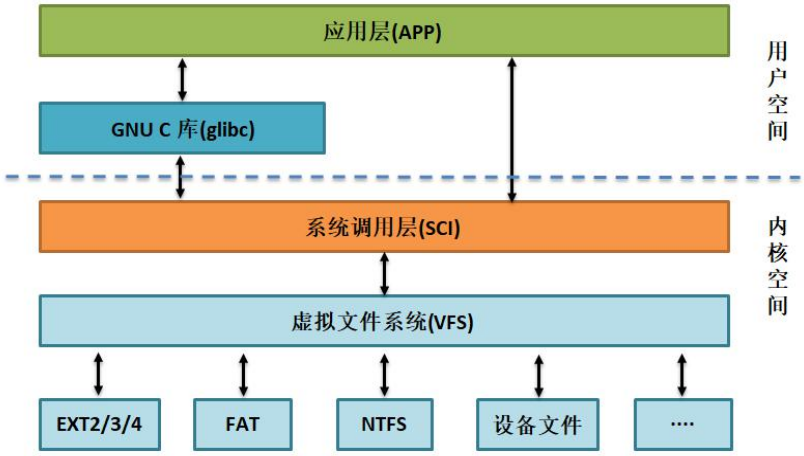


表 2-1 Linux 应用与文件系统

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，Bjame 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 曰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 系统上了已经安装了 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 的本文文件的全部内容：

```
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

作者知道的开源流媒体服务器超过 40 款，本章仅介绍采用 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 Fluendo 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

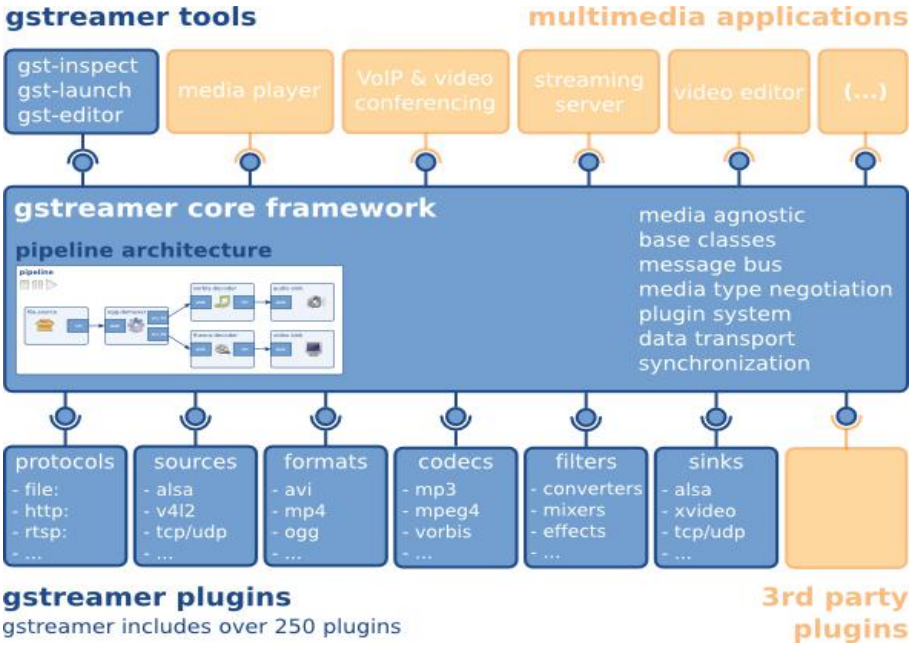


表 3-3 基于 GStreamer 的应用分层

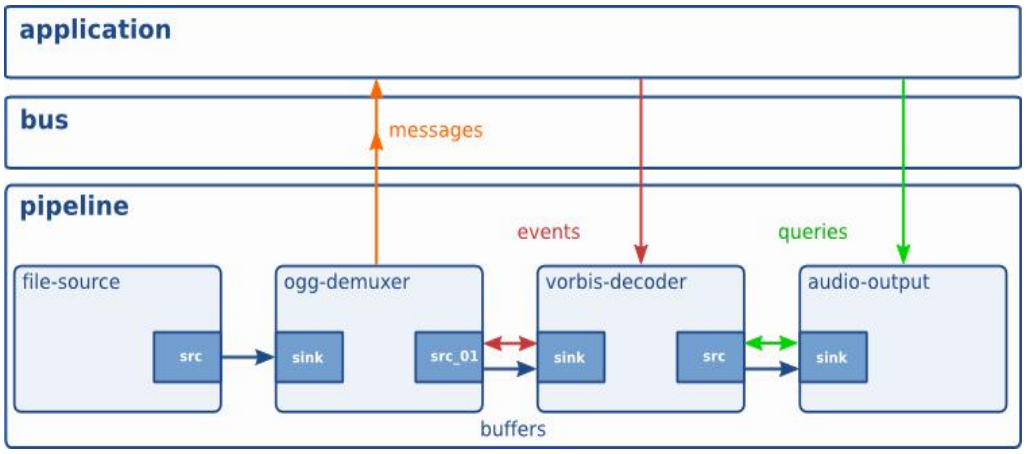


表 3-4 GStreamer 数据处理流程

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/macosforge/dss

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 是其中一位贡献者的名的缩写。

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 xsplayer 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 支持以下流媒体传输协议：

(1) RTSP/RTP/RTCP 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:

Option
OptionParser
mINI ◁- CMD
mINI ◁- CMD ◁- CMD_help
mINI ◁- CMD ◁- CMD_exit
mINI ◁- CMD ◁- CMD_clear
CMDRegister
std::exception ◁- ExitException
File 描述： ① 文件和目录操作。
function_traits<> 描述： ① 函数、lambda 转 functional。
List 描述： ① 单链表模板类。
noncopyable ◁- Logger 描述： ① 各种日志输出通道类的总管。单实例名称 g_defaultLogger。
ostringstream ◁- LogContext 描述： ① 日志输出的上下文（文件名、函数名、行号等）。
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。
noncopyable ◁- LogWriter 描述： ① 日志输出器接口类。
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。

noncopyable ◁- **LogChannel** 描述： ① 日志通道抽象类。可运行时设置日志输出级别。
noncopyable ◁- LogChannel ◁- **ConsoleChannel** 描述： ① 带色彩属性输出日志到控制台（cout）。
noncopyable ◁- LogChannel ◁- **FileChannelBase** 描述： ① 异步日志输出器的实现。在独立的工作线程中 run()
noncopyable ◁- LogChannel ◁- FileChannelBase ◁- **FileChannel** 描述： ① 异步日志输出器的实现。在独立的工作线程中 run()
MD5 描述： ① MD5 加密模块。
std::map<key,variant> ◁- **mINI_basic<string, variant>** 描述： ① INI 配置文件的配置项字典。
using **mINI** = mINI_basic<string, variant>
std::string ◁- **variant**
EventDispatcher
NotifyCenter
onceToken
std::shared_ptr<> ◁- **shared_ptr_imp<>**
ResourcePool_1<>
ResourcePool<> **内存资源池**
RingDelegate<> 描述： ① 环形缓冲队列数据包入列前的回调通知接口（委托）。
_RingReader<> 描述： 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 poller 线程中执行。
_RingStorage<>

描述： ① 环形缓冲队列容器类，负责缓冲区的实际存取。
_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 map。 ③ 每绑定一次 EventPoller 对象，就会创建一个 RingReader 对象。 ④ 调用此类的 write 方法时，将会在调用者线程内通知 map 内所有 RingReader。调用者负责在 EventPoller 所属的线程中执行 write 方法。 ⑤ 对使用者提供 map 大小变化通知。
RingBuffer<> 描述： ① 环形缓冲区队列。 ② 具备数据包入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 EventPoller 线程的_RingReader<>对象的能力。 ④ 一个_RingBuffer<>可以 attach 多个 EventPoller 对象引用。 ⑤ 每个 EventPoller 对象指针映射一个_RingReaderDispatcher<>对象。 ⑥ 每个_RingReaderDispatcher<>对象可以 attach 多个_RingReader<>对象。
SHA1
exception ◁- SqlException
SqlConnection
SqlPool
SqlStream
SqlWriter
SSL_Initor
SSL_Box
SSLUtil
Ticker
SmoothTicker
std::string ◁- _StrPrinter
noncopyable
Any
std::unordered_map<string,Any> ◁- AnyStorage
Creator

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:

semaphore 描述： ① 信号量。
ThreadLoadCounter 描述： ① 线程负载计数器。
noncopyable ◁- TaskCancelable 描述： ① 定义了可取消的任务的抽象。

TaskCancelable ◁- TaskCancelableImp<>
typedef TaskCancelableImp<uint64_t(void)> DelayTask;
typedef TaskCancelableImp<void()> Task;
描述：
① 可取消的任务的实现。

TaskExecutorInterface 描述： ① 任务执行器接口，支持同步和异步执行指定任务。
ThreadLoadCounter◁-------TaskExecutor TaskExecutorInterface◁---| 描述： ① 任务执行器的实现。
TaskExecutorGetter 描述： ① 定义了可访问任务执行器的用户类的抽象。
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 实现了具备负载均衡算法的 getExecutor 模板方法。 ③ 作为 EventPollerPool 的基类。 ④ 作为 WorkThreadPool 的基类。
TaskQueue 描述：

① 实现了线程安全的任务（函数对象）列队。 ② ThreadPool 类聚合了一个任务队列。
thread_group
TaskExecutor ◁- ThreadPool
... ◁- TaskExecutorGetterImp ◁- WorkThreadPool

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:

...◁- TaskExecutorGetterImp ◁- EventPollerPool 描述： 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。
...◁- TaskExecutor ◁- EventPoller 描述： 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。
Pipe 描述： 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。
PipeWrap 描述： 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。
FdSet
Timer

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:

noncopyable ◁- Buffer 描述: ① 缓存抽象类。
noncopyable ◁- Buffer ◁- BufferOffset 描述: ① 支持偏移基址的缓存类。
noncopyable ◁- Buffer ◁- BufferRaw 描述: ① 可动态扩容的裸指针式缓存类。
noncopyable ◁- Buffer ◁- BufferLikeString 描述: ① 对 std::string 作为容器的缓存类。 ② 提供类似 std::string 的常用操作。
noncopyable ◁- Buffer ◁- BufferSock 描述: ① 保存了 sockaddr 指针的缓存。
noncopyable ◁- Buffer ◁- BufferList 描述: 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。
std::exception ◁- SockException
SockNum
SockFD
MutexWrapper
SockInfo
noncopyable ◁--|-- Socket SockInfo ◁-----|
SockSender
SockSender ◁----------SocketHelper SockInfo ◁---------------| TaskExecutorInterface ◁---|
SockUtil
... ◁- SocketHelper ◁- TcpClient
mINI ◁- TcpServer

...◁- SocketHelper ◁- TcpSession 描述： ③ 服务器端通过监听套接字 accept 得到的 TCP 连接对象。 ④ 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
AudioInfo
... <|- MultiMediaSourceMuxer <|- DevChannel
... <|- FrameWriterInterface <|- MediaSinkInterface
... <|- MediaSinkInterface <|--MediaSink TrackSource <|----------------|
MediaSourceEvent
MediaSourceEvent <|- MediaSourceEventInterceptor
MediaInfo
BytesSpeed
TrackSource <|- MediaSource
FlushPolicy
PacketCache 获得内存资源使用权。
MediaSink <|- MultiMuxerPrivate
... <|- MediaSourceEventIterceptor <|--MultiMediaSourceMuxer ... <|- MediaSinkInterface <|----------| ... <|- MultiMuxerPrivate::Listener <|-|
std::multimap<string, string, StrCaseCompare> <|- StrCaseMap
Parser
DeltaStamp
DeltaStamp <|- Stamp

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
H264Encoder

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：

...◁- AudioTrack ◁- AACTrack
...◁- Sdp ◁- AACSdp
...◁- RtmpCodec ◁----------AACRtmpDecoder ...◁- ResourcePoolHelper<FrameImp> ◁---|
...◁- AACRtmpDecoder ◁----------AACRtmpEncoder ...◁- ResourcePoolHelper<RtmpPacket> ◁-|
...◁- RtpCodec ◁----------AACRtpDecoder ...◁- ResourcePoolHelper<FrameImp> ◁---|
...◁- AACRtpDecoder ◁----------AACRtpEncoder ...◁- RtpInfo ◁----------|
...◁- RtmpCodec ◁----------CommonRtmpDecoder ...◁- ResourcePoolHelper<FrameImp> ◁---|
...◁- CommonRtmpDecoder ◁----------CommonRtmpEncoder ...◁- ResourcePoolHelper<RtmpPacket> ◁-|
...◁- RtmpCodec ◁----------CommonRtpDecoder ...◁- ResourcePoolHelper<FrameImp> ◁---|
...◁- CommonRtpDecoder ◁----------CommonRtpEncoder ...◁- RtpInfo ◁----------|
Factory
CodecId
TrackType 描述： ① 轨道码流的一级类型。
CodecInfo 描述： ① 轨道码流的编码信息（二级类型）读取接口。
...◁- Buffer ◁---Frame CodecInfo ◁-------|
...◁- Frame ◁---FrameImp
...◁- FrameInternal
ResourcePoolHelper<>
FrameWriterInterface
FrameWriterInterface ◁- FrameWriterInterfaceHelper
FrameWriterInterface ◁- FrameDispatcher
Frame ◁- FrameFromPtr
...◁- FrameWrapper
...◁- FrameImp ◁- H265Frame
...◁- FrameFromPtr ◁- H265FrameNoCacheAble
...◁- VideoTrack ◁- H265Track 描述： ① 视频轨道大类的二级分类 不 H.265 轨道类。

...◁- Sdp ◁- H265Sdp
...◁- RtmpCodec ◁----------H265RtmpDecoder ...◁- ResourcePoolHelper<H265Frame> ◁---|
...◁- RtpCodec ◁----------H265RtpDecoder ...◁- ResourcePoolHelper<H265Frame> ◁---|
...◁- H265RtpDecoder ◁---H265RtpEncoder ...◁- RtpInfo ◁----------|
...◁- AudioTrackImp ◁- OpusTrack
...◁- Sdp ◁- OpusSdp
T_SPS
T_PPS
PPS
...◁- FrameDispatcher ◁---Track CodecInfo ◁---------------| 描述： ① 具备帧输入事件回调功能和编码信息访问接口的媒体轨道抽象类。
...◁- Track ◁- VideoTrack 描述： ① 视频轨道抽象接口，可获得视频的宽高和 FPS。 ② 不支持视频分层。 ③ 不支持多码率码流并行。
...◁- Track ◁- AudioTrack 描述： ① 音频轨道抽象接口，可获得采样率、位宽和声道数量。
...◁- AudioTrack ◁- AudioTrackImp
TrackSource

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:

...◁- BufferString ◁- **FMP4Packet**
...◁- MediaSource ◁----------**FMP4MediaSource** ...◁- RingDelegate<FMP4Packet::Ptr> ◁--| ...◁- PacketCache<FMP4Packet> ◁--------|
...◁- MP4MuxerMemory ◁----------**FMP4MediaSourceMuxer** ...◁- MediaSourceEventInterceptor ◁----|

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:

ts_segment
HlsParser
...◁- HttpClientImp ◁----HlsPlayer ...◁- PlayerBase ◁-------| HlsParser ◁----------------|
...◁- PlayerImp<HlsPlayer,PlayerBase> ◁---HlsPlayerImp ...◁- MediaSink ◁-------------------------|
HttpBody
HttpBody ◁- HttpFileBody
HttpBody ◁- HttpMultiFormBody
HttpRequestSplitter ◁- HttpChunkedSplitter
std::map<string,variant,StrCaseCompare> ◁- HttpArgs
TcpClient ◁-------------HttpClient HttpRequestSplitter ◁--|
...◁- TcpClientWithSSL<HttpClient> ◁- HttpClientImp
HttpCookie
HttpCookieStorage
...◁- AnyStorage ◁---HttpServerCookie noncopyable ◁----------|
RandStrGenerator
HttpCookieManager
...◁- HttpClientImp ◁- HttpDownloader
HttpResponseInvokerImp
HttpFileManager
...◁- HttpClientImp ◁- HttpRequester
HttpRequestSplitter
...◁- TcpSession ◁----------HttpSession FlvMuxer ◁--------------------| HttpRequestSplitter ◁-------| ...◁- WebSocketSplitter ◁--|
...◁- HttpClientImp ◁- HttpTSPlayer
strCoding
...◁- ClientTypeImp
...◁- HttpClientImp ◁------HttpWsClient ...◁- WebSocketSplitter ◁-|
...◁- WebSocketClient
SendInterceptor
...◁----------------TcpSessionTypeImp SendInterceptor ◁--|

TcpSessionCreator<>
...◁- WebSocketSessionBase
...◁- WebSocketSession
WebSocketHeader
...◁- BufferString ◁- WebSocketBuffer
WebSocketHeader ◁- WebSocketSplitter

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**
TrackSource ◁- **DemuxerBase**
...◁- DemuxerBase ◁- **PlayerBase** mINI ◁----------|
...◁- **PlayerImp**
...◁- 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 ◁----|

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:

PusherImp<PusherBase,PusherBase> ◁- MediaPusher
mINI ◁- PusherBase
... ◁- PusherImp

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:

AMFValue
AMFDecoder
FlvMuxer
FlvMuxer ◁- FlvRecorder
RtmpHandshake
RtmpHeader
Buffer ◁- RtmpPacket
CodecInfo ◁- Metadata
...◁- Metadata ◁- TitleMeta
RtmpRing 描述: ① 聚合了一个 RingBuffer 对象用于缓冲 RTMP 协议包。
...◁- RtmpRing ◁---------RtmpCodec ...◁- FrameDispatcher ◁--| CodecInfo ◁----------|
...◁- Demuxer ◁- RtmpDemuxer
...◁- MediaSource ◁--------------------RtmpMediaSource ...◁- RingDelegate<RtmpPacket::Ptr> ◁--| PacketCache<RtmpPacket> ◁----------|
...◁- RtmpMediaSource ◁----------RtmpMediaSourceImp Demuxer::Listener ◁----------| MultiMediaSourceMuxer::Listener ◁--|
...◁- RtmpMuxer ◁--------------------RtmpMediaSourceMuxer ...◁- MediaSourceEventInterceptor ◁--|
...◁- MediaSinkInterface ◁- RtmpMuxer
...◁- PlayerBase ◁----RtmpPlayer ...◁- TcpClient ◁-----| ...◁- RtmpProtocol ◁--|
...◁- PlayerImp<RtmpPlayer,RtmpDemuxer> ◁- RtmpPlayerImp
HttpRequestSplitter ◁- RtmpProtocol
...◁- RtmpProtocol ◁--RtmpPusher ...◁- TcpClient ◁-----| ...◁- PusherBase ◁----|
...◁- TcpSession ◁----RtmpSession ...◁- RtmpProtocol ◁--| MediaSourceEvent ◁-----|

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:

Decoder
FrameMerger
DecoderImp
...◁- HttpRequestSplitter ◁--RGB28181Process ...◁- RtpReceiver ◁----------| ...◁- ProcessInterface ◁-----|
ProcessInterface
PSDecoder ◁- Decoder
...◁- MediaSinkInterface ◁- PSEncoder
...◁- PSEncoder ◁- PSEncoderImp
...◁- PacketCache<Buffer> ◁- RtpCache
...◁- RtpCache ◁------RtpCachePS ...◁- PSEncoderImp ◁--|
SockInfo ◁----------RtpProcess ...◁- MediaSinkInterface ◁----------| ...◁- MediaSourceEventInterceptor ◁--|
...◁- MediaSourceEvent ◁- RtpProcessHelper
RtpSelector
...◁- MediaSinkInterface ◁- RtpSender
RtpServer
...◁- TcpSession ◁--------RtpSession ...◁- RtpSplitter ◁-------| ...◁- MediaSourceEvent ◁--|
...◁- HttpRequestSplitter ◁- RtpSplitter
...◁- HttpRequestSplitter ◁- TSSegment
Decoder ◁- TSDecoder

表 5-14 ZLMediaKit/src/Rtp 目录类型描述表

5.1.15. Rtsp

ZLMediaKit/src/Rtsp 目录类型描述表:

RtpRing
ResourcePoolHelper ◁- RtpInfo
RtpRing ◁----------RtspSession FrameDispatcher ◁--| CodecInfo ◁--------|
MultiCastAddressMaker
RtpMultiCaster
PacketSortor
RtpReceiver
PayloadType
BufferRaw ◁- RtpPacket
SdpTrack
SdpParser
RtspUrl
CodecInfo ◁- Sdp
...◁- Sdp ◁- TitleSdp
...◁- Demuxer ◁- RtspDemuxer 描述: ② 实现了 RTP 包的解复用。分解为多个独立的 Track 对象。
...◁- MediaSource ◁--RtspMediaSource RtspSplitter ◁-------| RtpReceiver ◁--------|
...◁- RtspMediaSource ◁----------RtspMediaSourceImp Demuxer::Listener ◁---------------| MultiMediaSourceMuxer::Listener ◁-|
...◁- RtspMuxer ◁----------------RtspMediaSourceMuxer MediaSourceEventInterceptor ◁-----|
...◁- MediaSinkInterface ◁- RtspMuxer
...◁- PlayerBase ◁---------RtspPlayer TcpClient ◁-----------------| RtspSplitter ◁----------------| RtpReceiver ◁----------------|
...◁- PlayerImp<RtspPlayer,RtspDemuxer> ◁- RtspPlayerImp
...◁- TcpClient ◁----------RtspPusher RtspSplitter ◁----------------| PusherBase ◁-----------------|
Buffer ◁- BufferRtp
...◁- TcpSession ◁--RtspSession RtspSplitter ◁-------| RtpReceiver ◁--------| MediaSourceEvent ◁---|

描述： ① 服务器端 accept 得到的 RTSP 连接。
...◁- HttpRequestSplitter ◁- RtspSplitter
UDPServer

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:

HlsMaker
HlsMaker ◁- HlsMakerImp
MediaSource ◁- HlsMediaSource
HlsCookieData
MediaSourceEventInterceptor ◁- HlsRecorder ...◁- TsMuxer ◁---------- |
MP4FileIO
MP4FileIO ◁- MP4FileDisk
MP4FileIO ◁- MP4FileMemory
...◁- MP4FileDisk ◁--MP4Demuxer TrackSource ◁--------- |
...◁- MediaSinkInterface ◁- MP4MuxerInterface
...◁- MediaMuxerInterface ◁- MP4Muxer
...◁- MP4MuxerInterface ◁- MP4MuxerMemory
MediaSourceEvent ◁- MP4Reader
...◁- MediaSinkInterface ◁- MP4Recorder
RecordInfo
Recorder
...◁- MediaSinkInterface ◁- TsMuxer

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
...◁- TcpSession ◁- ShellSession

表 5-27 ZLMediaKit/src/Shell 目录类型描述表

5.2. 源码构建

TODO：

5.3. 配置方法

TODO：

5.4. 运行方法

TODO：

5.5. 总体分析

ZLMediaKit 主要类型

ZLMediaKit 的核心部件
// 去除所有不是必须的辅助宇宙。

ZLMediaKit 的外围部件
// 运行时典型的平行宇宙空间分布。
(1) 主线程 main。
(2) 异步日志输出线程 AsyncLogWriter。
(3) 时间戳记录线程 initMillisecondThread。
(4) I/O 事件轮询线程 EventPoller::runLoop。

ZLMediaKit 的启动过程
// 启动过程分析。

5.6. 基础模块分析

5.6.1. 网络 I/O

TODO：

5.6.2. 线程池

TODO：

5.6.3. 文件系统 I/O

TODO：

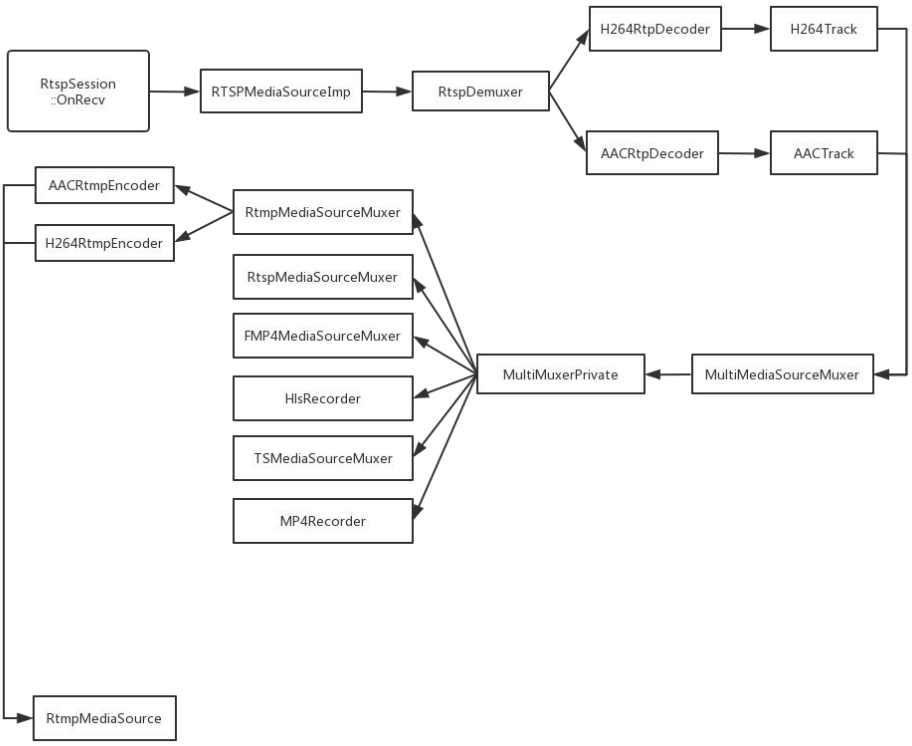
5.7. 媒体接入流程

5.7.1. RTMP 接入

TODO：

5-1 RTMP 协议接入流程

5.7.2. RTSP 接入



5-2 RTP 包接入流程

5.7.3. GB28181 接入

在全球基于 IP 网络的实时音视频通信领域有两套比较重要的行业标准，分别是 1996 年提出的 H.323 和 1999 年提出的 SIP。

SIP 全称是 Session Initiation Protocol，中文名为“会话初始协议”。

SIP 标准是 IETF 组织在 1999 年提出的，其应用目标是在基于 internet 环境，实现音视频和数据的实时通讯。

SIP 标准的信令信息是基于文本的，采用符合 iso10646 的 utf-8 编码。

GB28181 协议的通信信令部分采用的是 SIP 标准。

5.7.4.GB28181 协议

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

5.9.3. GB28181 直播

5.10. 媒体推送流程

5.11. 媒体拉取流程

5.12. 媒体录制流程

第 6 章　测试与总结

6.1. 测试环境

6.2. 测试数据

6.3. 测试总结

6.4. 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，开办了自己的个人网站 codemi.net，尝试通过这个窗口来传播自己的专业知识和设计思想。

作者计划在未来十年内，创作十本书。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。

官方技术交流 QQ 群：**462178798**