

FreeSWITCH

源代码分析

杜金房 著

```
INVITE sip:you@xswitch.cn SIP/2.0
Via: SIP/2.0/TLS xswitch.cn:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: "Seven Du" <sip:seven@xswitch.cn>;tag=9fxced76sl
To: You <sip:you@xswitch.cn>
Call-ID: 3848276298220188511@xswitch.cn
CSeq: 2 INVITE
Contact: <sip:seven@xswitch.cn;transport=tls>
```

小樱桃出品

FreeSWITCH 源代码分析

杜金房

图书不在版编目（NCIP）数据

FreeSWITCH 源代码分析/杜金房 著/2016.7

ISBN 7-DU-777777-7

本书主要针对想了解 FreeSWITCH 源代码，以及通过修改 FreeSWITCH 源代码对 FreeSWITCH 进行二次开发并为 FreeSWITCH 开源项目做贡献的读者。

阅读本书前强烈建议阅读《FreeSWITCH 权威指南》以及《FreeSWITCH 实例解析》。

FreeSWITCH 源代码分析

作 者	杜金房
封面设计	杜金房
校 对	杜金房
排 版	杜金房
责任编辑	杜金房
开 本	216 mm × 279 mm
印 张	7.5
印 数	7
版 数	2016 年 7 月第 1 版 2020 年 12 月第 7 次发布
电子邮箱	freeswitch@dujinfang.com

前 言

自《FreeSWITCH 权威指南》出版以来，已经过去两年多了。在这两年多的时间里，我收到很多的反馈，有批评的，有赞扬的，大部分还是赞扬的。

有一部分读者反映《指南》中的内容写得不够深入。我想，这部分读者应该都是比较高级的读者，希望基于某个主题进行更深入地研究。但是，作为 FreeSWITCH 方面的第一本中文书，《指南》已经很厚了。不可否认，其中的某些知识点和案例，如 NAT、SIP、二次开发等，可能需要整整一章或者一本书才能写得详尽，但很显然为每个这样的知识点都写一本书是不现实的。

在设计书稿的时候，《指南》分为三个部分，其实应该分成三本书比较好，考虑到实际情况我们才把所有内容都放到一本书上了。而且，还有些内容放不下，后我们只好以电子版附录的形式发布。

说到电子版，其实还是有很多优势的。一是出版周期短，二是可以随时更新。好多朋友也都想读电子版。与此同时，好多读者也都希望我能专门写一本 FreeSWITCH 源代码方面的书，以便对 FreeSWITCH 内部结构和逻辑有进一步的了解，并更快更好的进行二次开发，并为 FreeSWITCH 开源项目做贡献。

说到为开源项目做贡献，其实很多读者都已经在做了。学习、使用 FreeSWITCH，其实已经在为 FreeSWITCH 在做贡献了；有的人会把自己学习心得共享出来，让很多人受益；有的人把自己在使用 FreeSWITCH 过程中发现的问题，上报到 FreeSWITCH 社区，让更多的人受益；有的人也为 FreeSWITCH 提交补丁，从根本上帮助 FreeSWITCH 壮大和发展，这也是 FreeSWITCH 开放源代码最重要的意义了。

好吧，终于该写写为什么写这本书了。

其实，答案很简单——我一直想写。至于一直想写又一直没写的原因，主观上，感觉看得人太少；客观上，其实能读源代码的人，自己看着看着就懂了，不用看书。写书是一件很花时间和精力事情，看得人少了，就感觉价值不大。但是，总有一些读者告诉我想看到一本关于源代码的书，因此我最终还是决定把它写出来。无非，把价格定得高一点，1240 元一本，而且是预售，坑一个算一个。希望读者在购买本书前能慎重一点，也希望我的书能献给那些真正想掌握源代码的人。

也许本书能帮助更多的开发者快速掌握 FreeSWITCH 代码的流程和设计原则。当然，即使你永远不会修改源代码，你也可以从源代码中看到 FreeSWITCH 内更多的秘密，享受阅读源代码的乐趣。

本书的前三章来自《FreeSWITCH 权威指南》，有增删。当然，原来的内容都是针对 FreeSWITCH 1.2 版的代码写的，至今，已有很多变化，但总体设计思路变化不大，因此，就没有费力去更新这些

内容。读者在阅读时记着不要盲目照搬书上的例子就是了。我们后面会讲到这些变化。当然，如果读者看完本书还不能理解 1.2 和 1.6 版本的不同，那就是本书没有写好，不怪读者。或许，将来我会把这三章代码再更新到 1.6，但到那时候 FreeSWITCH 或许发展到 1.8 或 2.0 了。无论如何，我觉得写新东西更有价值一些，因此，前三章，暂不会更新代码版本。

阅读本书前强烈建议阅读《FreeSWITCH 权威指南》和《FreeSWITCH 实例解析》，或者也要读一下《FreeSWITCH 文集》。在了解 FreeSWITCH 的特性和使用方法后，再去阅读源代码才会事半功倍。有些读者一上来就读 FreeSWITCH 源代码，试图从源代码中读出 FreeSWITCH 是怎么用的，笔者认为有些本末倒置。我不反对，但极不推荐那么做。

阅读本书需要有 C 语言基础和一些基本的 Linux 知识，还有，学源代码嘛，必须要会 Git。

本书主要是在 Mac 上写的，例子基本适用于 Mac 和 Linux。其实 FreeSWITCH 是跨平台的，Windows 用户也很容易能理解，但是，Windows 读者如果需要练习修改或编译源代码时，需要自己学习和掌握 Visual Studio，毕竟，讲述如何使用 Visual Studio 大大超出了本书的范围。

如果你喜欢 FreeSWITCH，你肯定喜欢本书。

本书内容会不断更新，请关注本书的网站：<http://book.dujinfang.com>。你也可以订阅 FreeSWITCH-CN 微信公众号以随时了解 FreeSWITCH 和本书动态。



图 1: FreeSWITCH-CN 微信公众号

目 录

前 言	III
1 源代码导读及编译指南	1
1.1 准备 FreeSWITCH 源代码环境	1
1.2 FreeSWITCH 源代码目录结构	2
1.3 FreeSWITCH 源代码的编译	2
1.3.1 首次编译	3
1.3.2 增量编译	5
1.3.3 常见问题及最佳实践	6
1.4 小结	7
2 FreeSWITCH 源代码导读	8
2.1 核心架构	8
2.1.1 APR	8
2.1.2 SWITCH APR	9
2.1.3 main 函数	11
2.1.4 可加载模块	13
2.1.5 模块的结构	21
2.1.6 Session 和 Channel	23
2.1.7 SWITCH IVR	29
2.1.8 Core IO	30
2.1.9 Core Media	34
2.1.10 Core RTP	35

2.1.11 SWITCH XML	37
2.1.12 SWITCH Event	39
2.1.13 Core Codec 和 Core File	43
2.1.14 Core Video	44
2.2 模块	49
2.2.1 mod_dptools	49
2.2.2 mod_commands	65
2.2.3 mod_sofia	67
2.2.4 小结	78
2.3 Endpoint 接口	79
2.3.1 rtp Endpoint	79
2.3.2 rtc Endpoint	96
2.3.3 null Endpoint	106
3 FreeSWITCH 二次开发	115
3.1 给 FreeSWITCH 汇报 Bug 和打补丁	115
3.1.1 汇报 Bug 时注意的问题	115
3.1.2 修复内存泄露问题	116
3.1.3 给中文模块打补丁	117
3.1.4 给 FreeSWITCH 核心打补丁	118
3.1.5 高手也会犯错误	121
3.1.6 汇报严重的问题	121
3.1.7 给 Sofia-SIP 打补丁	123
3.1.8 给现有 App 增加新功能	125
3.1.9 给 FreeSWITCH 增加一个新的 Interface	127
3.2 写一个新的 FreeSWITCH 编解码模块	129
3.3 从头开始写一个模块	131
3.3.1 初始准备工作	132
3.3.2 写一个简单的 Dialplan	133

3.3.3 增加一个 App	135
3.3.4 写一个 API	136
3.3.5 小结	138
3.4 使用 libfreeswitch	138
3.4.1 自己写一个软交换机	138
3.4.2 使用 libfreeswitch 提供的库函数	140
3.4.3 其它	145
3.5 主要数据结构和函数使用方法	146
3.5.1 通过 UUID 获取 Session	146
3.5.2 JSON	146
3.5.3 JSON API	150
3.5.4 切割字符串	154
3.5.5 散列表	155
3.5.6 绑定事件	156
3.5.7 遍历 Event 消息头	157
3.5.8 将以逗号分隔的字符串转换成事件	157
3.5.9 产生一个 custom 事件	158
3.5.10 队列与线程池	158
3.5.11 Media Bug	160
3.5.12 任务调度	161
3.6 调试跟踪	163
3.7 测试框架	164
3.7.1 switch_ivr_originate.c	166
3.7.2 switch_utils.c	169
3.7.3 test_mod_av.c	170
3.7.4 test_avformat.c	174
3.8 小结	176

4 核心代码详解	177
4.1 g711.c	177
4.2 inet_pton.c	181
4.3 switch.c	181
4.4 switch_apr.c	192
4.5 switch_buffer.c	192
4.6 switch_caller.c	193
4.7 switch_channel.c	198
4.8 switch_config.c	245
4.9 switch_console.c	245
4.10 switch_core.c	258
4.11 switch_core_asr.c	282
4.12 switch_core_cert.c	286
4.13 switch_core_codec.c	288
4.14 switch_core_db.c	295
4.15 switch_core_directory.c	297
4.16 switch_core_event_hook.c	297
4.17 switch_core_file.c	299
4.18 switch_core_hash.c	305
4.19 switch_core_io.c	308
4.20 switch_core_media_bug.c	321
4.21 switch_core_video.c	335
4.22 switch_scheduler.c	386
4.23 switch_core_memory.c	396
4.24 switch_core_timer.c	399
4.25 switch_core_port_allocator.c	400
4.26 switch_core_sqldb.c	402
4.27 switch_ivr.c	413
4.28 switch_loadable_module.c	429

4.29	switch_utils.c	452
4.30	switch_vad.c	470
5	模块代码选析	481
5.1	mod_conference.c	481
5.2	mod_hiredis.c	484
5.3	mod_fifo.c	489
5.4	mod_amqp	493
6	代码修炼之道	515
6.1	虚拟演播室	515
6.1.1	Chroma Key	515
6.1.2	mod_video_filter	518
6.1.3	编译相关	527
6.1.4	精彩继续	528
6.1.5	永无止境	529
6.1.6	小结	530
6.2	使用 Perf 定位性能问题	530
6.3	测试最新版的 FFmpeg	532
6.4	如何维护自己的 FreeSWITCH 分支	534
6.4.1	基础知识	535
6.4.2	冲突解决	539
6.4.3	维护自己的分支	540
6.4.4	小结	541
6.5	解析 SIP 中携带的 ISUP 消息	542
	版本更新历史	552
	写在最后	554
	作者简介	555

FreeSWITCH 源代码分析	目 录
版权声明	556
7 广告	557
7.1 关于广告的广告	557
7.2 XSWITCH 云——我自己的通信助手	557
7.3 RTS 中文社区	557
7.4 烟台小樱桃网络科技有限公司提供商业 FreeSWITCH、Kamailio 及 OpenSIPS 技术支持	557
7.5 知识星球	557
7.6 FreeSWITCH 相关图书推荐	558
	560

第一章 源代码导读及编译指南

有好多朋友在问到如何阅读源代码时，告诉笔者，他们最大的困扰并不是看不懂代码，而是不知道从哪里下手，就好像是老虎吃天——无从下口。是的，FreeSWITCH 的源代码太长了，确实好像从哪里看好像都找不到源头。不过，也不要因此而望而却步。在看比较大型的项目时，尤其是对着不熟悉的功能和代码，总要经过这么一个过程。尤其是，有些读者，在对 FreeSWITCH 本身还不熟悉的时候，就试图通过阅读源代码来了解系统的功能。笔者认为那是本末倒置的方法，不可取。笔者一直坚持，要想阅读 FreeSWITCH 的源代码，先要阅读《FreeSWITCH 权威指南》、《FreeSWITCH 实例解析》等，自己多做实验，掌握了 FreeSWITCH 的基本功能，再来阅读源代码会容易入手一些。

当然，阅读源代码不仅仅是为了满足我们的好奇心——哪些功能是怎么实现的、系统还有何种没有公开（写到文档里）的功能等。更重要的是，如果我们熟悉了源代码，我们就可以修改它——不管是修复 BUG、增加功能并为开源项目做贡献，还是修改源代码以适合你自己的需要，这些都能给你带来很好的成就感。

接下来，我们从准备源代码环境开始，由浅入深地进一步进入 FreeSWITCH 内部的神秘世界。

1.1 准备 FreeSWITCH 源代码环境

首先，Clone FreeSWITCH 源代码：

```
git clone https://freeswitch.org/stash/scm/fs/freeswitch.git
```

要查看源代码，最好选择一个具有语法高亮功能的阅读器或编辑器。作者在 Mac 平台上一般使用 Sublime Text 2，它是跨平台的，也可以在 Windows 上使用。当然也可以使用一些经典工具，如在 UNIX 类系统上使用 vi/vim 或 Emacs，在 Windows 上可以使用 Visual Studio 等。关于在 Windows 平台中的编译方法我们在《FreeSWITCH 权威指南》中已经讲过了，本章及后面章节的例子都是在 Mac 平台上写成的，它适用于大部分的 UNIX 类平台（如 Linux 等）。

1.2 FreeSWITCH 源代码目录结构

FreeSWITCH 的源代码目录中，`src` 目录中包含了绝大部分的源代码；`libs` 目录下是一些第三方的库和模块，如 `libs/sofia-sip` 就是 Nokia 的 SIP 库。

在 `src` 目录中，`include` 目录存放了系统大部分的头文件；不同模块的代码则分门别类的放到 `mod` 目录中不同的子目录中。系统的核心代码则直接在 `src` 目录中。

FreeSWITCH 模块的源代码（`mod` 目录）结构如下表所示：

目录	说明
<code>asr_tts</code>	语音识别及合成相关模块
<code>dialplans</code>	Dialplan 模块
<code>endpoints</code>	Endpoint 模块，如 <code>mod_sofia</code>
<code>formats</code>	文件格式模块，如 <code>mod_sndfile</code>
<code>loggers</code>	日志模块
<code>sdk</code>	一些例子和宏
<code>xml_int</code>	XML 相关的模块
<code>applications</code>	提供各种应用功能的模块，如 <code>mod_dptools</code> 和 <code>mod_commands</code>
<code>directories</code>	LDAP
<code>event_handlers</code>	事件处理模块
<code>languages</code>	嵌入式语言模块
<code>say</code>	不同语种的语言模块
<code>timers</code>	时钟和定时器模块

上面不同的分类也与 FreeSWITCH 内部模块的抽象大致对应，但也有例外的情况，典型地，一些模块可能有多个接口（Interface）实现，这样的模块会根据其主要功能放到对应的目录中，有时就直接放到 `applications` 目录中，相当于该目录中有一些多功能的模块。但随着时间的推移，某些单功能的模块也可能被增加一些新的功能和接口（Interface），变成多功能模块。

1.3 FreeSWITCH 源代码的编译

关于 FreeSWITCH 的编译，我们在《权威指南》第 3 章中的编译安装中已经提到了。在这里，我们再复习一下，并了解一些针对开发者了讲需要注意的问题。在这里，我们主要以在 Linux 系统上的编译为例。Windows 平台上的编译过程及注册事项可以参考《指南》第 3 章的相关内容。

1.3.1 首次编译

在编译源代码前，请确保按第 3 章中所讲的安装相关的依赖库及开发包。FreeSWITCH 在开发中使用经典的 `gcc`、`Makefile` 及 `automake`、`autoconf` 等 GNU 工具链，因而在各种平台上都很容易地进行编译。

为了方便不了解这些 GNU 工具的读者，我们在此也顺便简单讲一下。

首先，大家知道，FreeSWITCH 主要是用 C 和 C++ 写的。编译 C 语言的程序一般需要 `gcc`，如，如下命令会编译 `test.c` 并生成一个可以执行的二进制程序：

```
gcc test.c -o test
```

当源代码数量过多时，一行一行的执行 `gcc` 就比较累了。因此，可以编写简单的 Shell 脚本或 Makefile 实现。Makefile 是 `make` 工具使用的文件，它除了定义源文件到目标文件的编译方法外，还能定义这些文件的依赖关系。通过检查这些依赖关系，如果在下次编译时源文件没有修改过，则可以不用重复编译，因而可以大大加快编译速度。

不同平台上的工具链是不一样的，在 Linux 等开源平台上一般使用 `gcc`，而在其他商业的 UNIX 系统上往往都有各厂商自己的编译工具链，因而一种称为 `automake` 的工具出现了。通过编写 `configure` 脚本，定义一些宏，可以在编译前自动检测当前的平台环境和工具链，以生成适当的 Makefile。

当工程更大的时候，写 `configure` 脚本也是很累人的活，因而又有人发明了 `autoconf`，通过定义更简单的宏，可以自动生成 `configure` 脚本。

总之，大家可以结合 FreeSWITCH 的编译过程深入理解一下。

首先，如果你是从 Git 仓库中 Clone 的源代码，需要先执行一下 `bootstrap.sh`。它会初始化一些文件。

```
./bootstrap.sh
```

如果是直接下载的源代码 Tar 包，则不需要这一步，因为源代码在 `tar` 之前就已经执行过该步骤了。

接下来，执行 `configure`，它会生成 `Makefile`：

```
./configure
```

`configure` 有很多参数，其中比较常用的是 `prefix` 参数，用于将 FreeSWITCH 安装到指定的目录下（FreeSWITCH 默认的安装目录是 `/usr/local/freeswitch`），如：

```
./configure --prefix=/usr/local/freeswitch2
./configure --prefix=/opt/freeswitch
```

`configure` 执行完毕后，将产生 `Makefile`，以及一个 `modules.conf` 文件。`modules.conf` 用于控制在编译阶段要自动编译哪些模块。如果你需要这些模块，则可以编辑该文件，并去掉前面的“#”号注释，如：

```
$ head modules.conf
#applications/mod_abstraction
#applications/mod_avmd
#applications/mod_blacklist
#applications/mod_callcenter
#applications/mod_cidlookup
applications/mod_cluechoo
applications/mod_commands
applications/mod_conference
#applications/mod_curl
applications/mod_db
```

如果不知道哪些模块是干什么的，可以暂且不管这个文件。到以后也可以再单独编译某些模块。

接下来，执行 `make`，它将根据 `Makefile` 进行编译

```
make
```

编译成功后，执行如下命令将程序安装到相应的位置。

```
make install
```

注意，需要确认要安装的目标位置有写入的权限，如果这些命令都是以 `root` 执行的，那你不会遇到权限的问题，但如果你是以普通用户执行的，就可能遇到权限的问题。所以，如果有权限的问题，可以尝试用 `root` 进行安装：

```
sudo make install
```

或者，也可以通过如下方案以普通用户的身份安装，如，以 `freeswitch` 用户安装，假设你现在登录的用户就是 `freeswitch`：

```
sudo mkdir /usr/local/freeswitch          # 用 root 身份创建目录
sudo chown freeswitch /usr/local/freeswitch # 把目录的属主改为 freeswitch
make install                             # 用 freeswitch 普通用户身份安装即可
```

1.3.2 增量编译

有时候，我们修改了源文件，需要再次编译。在没有修改 `autoconf`、`automake` 相关的编译规则的话，直接执行 `make` 就行了：

```
make
```

或者，也可以直接执行

```
make install
```

`make` 会检查全部的规则，并决定哪些需要重新编译，这还是比较耗时的。如果你知道你仅仅修改了哪些模块的话，可以直接编译该模块，如：

```
make mod_sofia
make mod_sofia-install
```

使用这种方法也可以编译默认没有编译过的模块，如 `mod_shout` 模块提供 mp3 录、放音的支持，它默认是不被编译的，可以用以下命令安装：

```
make mod_shout-install
```

当然，在大多数情况下，你也可以直接进入相关的模块目录下，执行 `make`。如：

```
cd src/mod/endpoints/mod_sofia
make install
```

如果你改了核心的代码，则可以执行

```
make core-install
```

1.3.3 常见问题及最佳实践

如果在编译过程中出现某个或某些模块编译不通过的情况，可以先在 `modules.conf` 中将该模块注释掉，等全部的编译通过后，再单独检查该模块有什么问题。

如果跟作者一样，你经常在不同的分支中切来切去，则如果分支差异比较大时编译系统中的目标文件可能会乱掉。这是可能是编译规则设置的问题，但 FreeSWITCH 项目太大了，因而，作者经常在不同的目录中编译，如，下列命令编译最新的 master 版本到默认位置：

```
git clone https://freeswitch.org/stash/scm/fs/freeswitch.git freeswitch-master
cd freeswitch-master
./bootstrap.sh && ./configure && make && make install
```

编译 1.2 版本并安装到 `/usr/local/freeswitch-1.2`：

```
git clone https://freeswitch.org/stash/scm/fs/freeswitch.git freeswitch-1.2
cd freeswitch-1.2
git checkout v1.2.stable
./bootstrap.sh && ./configure --prefix=/usr/local/freeswitch-1.2
make && make install
```

通过这种方式，以后在维护多个分支时就不会混乱了，而且，如果有必要的话，也可以同时在一台主机上同时启动不同版本的 FreeSWITCH 实例。

1.4 小结

在本章，我们主要讲了如何获取及编译 FreeSWITCH 代码，这样读者就可以在阅读源代码的同时尝试改一些地方（至少，可以在某些关于的地方加一些日志输出的语句），并编译执行看一下效果。阅读永远是枯燥乏味的，只有配合一定的实践，让代码“动”起来，才会有意思。

第二章 FreeSWITCH 源代码导读

由于 FreeSWITCH 的代码非常多，为了节省篇幅，我们在本章中尽量不大段列出源代码。读者在阅读本章时可以配合源代码进行阅读。

FreeSWITCH 的更新速度比较快，因此，为了读者能找到正确的行号，我们这里的源代码导读基于 Git 的 Tag `v1.5.7`，它是在本章写作时比较新的版本。读者可以使用下列命令切换到该 Tag：

```
git checkout v1.5.7
```

2.1 核心架构

首先，我们 FreeSWITCH 一些核心的组件、概念，以及代码。

2.1.1 APR

FreeSWITCH 在设计之初就定位于跨平台的，它使用了跨平台较好的 APR 库¹。APR 出身于 Apache²的代码。Apache 是网络上非常流行的 Web 服务器软件，其代码是公认的写的比较好的代码之一。在程序员这一行业，大家一致的观点是——要想提高开发水平，除了大量的练习外，还要大量的阅读其他优秀系统的源代码，而 Apache 就是常被推荐阅读的代码之一。

APR 的主要目的是为应用提供一个可移植的、平台无关的层。它的底层，在不同平台上调用不同的库和函数来向上提供诸如文件系统访问、网络编程、进程和线程管理以及共享内存等一致的功能接口。使用 APR 开发的程序能够在所有被 Apache 所支持的平台上被干净地（最坏的情况也是需要很小程度修改）编译。

除了跨平台支持外，APR 的核心还提供系统内存、数据结构、线程、互斥锁等各种资源的管理和抽象等。大家都知道 C 语言的内存管理是非常令人头痛的，而 APR 通过内存池的管理大大提高了使用的方便性和安全性。

¹参见<http://apr.apache.org/>。

²参见[<httpd.apache.org>](http://httpd.apache.org)。

APR 实用库 (APR-UTIL, 或者 APU) 是 APR 项目的另一个库。它在 APR 基础上, 使用统一标准的编程接口, 提供了一部分功能函数集。APU 并不是每在一个平台上都有一个单独的模块, 但是它为某些其他常用的资源一个类似的方法, 这些资源包括 Base64 编码、MD5/SHA1 加密、UUID 以及队列 (Queue) 管理等。

FreeSWITCH 为了防止潜在的命名空间冲突等因素, 对所有使用到的 APR 函数又进行了一些封装, 这样所有的核心函数就都有了一致的命名空间 “`switch_`”。这些封装在 `switch_apr.c` 里实现。

2.1.2 SWITCH APR

FreeSWITCH 采用了 APR 的代码风格及约定, 非常易于使用。所以如果熟悉 APR 的话, 看 FreeSWITCH 的源代码就容易多了。当然, 反过来, 熟悉了 FreeSWITCH 的源代码也会熟悉 APR。

命名空间

在 APR 和 APR-UTIL 中, 所有的公开接口都使用了字符串前缀 “`apr_`” (数据类型和函数) 和 “`APR_`” (宏)。与此类似, 在 FreeSWITCH 中, 所有的核心接口函数也都使用了 “`switch_`” 前缀的函数和 “`SWITCH_`” 前缀的宏。

在 APR 命名空间中, 也大量使用了二级命名空间, 如 “`apr_socket_`” 等。同理在 FreeSWITCH 中, 也有类似的如 “`switch_file_`”、“`switch_core_session_`” 等二级及三级命名空间。

声明的宏

APR 使用类似于 `APR_DECLARE` 的宏进行声明。例如:

```
APR_DECLARE(apr_status_t) apr_initialize(void);
```

在很多的平台上, 这是一个空声明, 并且扩展为

```
apr_status_t apr_initialize(void);
```

但在某些平台, 如在 Windows 的 Visual C++ 平台上, 需要使用它们特有的、非标准的关键字, 例如 “`_dllexport`”、“`__stdcall`” 等来允许其他的模块使用一个函数, 这些宏就需要扩展以适应这些需要的关键字。

与 APR 此类似, 在 FreeSWITCH 中, 大部分使用 `SWITCH_DECLARE` 或 `SWITCH_DECLARE_DATA` 之类的声明。

另外，FreeSWITCH 中不同类型的模块也都有专用的声明，如声明 Application 的 `SWITCH_STANDARD_APP`、声明 Dialplan 的 `SWITCH_STANDARD_DIALPLAN` 以及声明 API 的 `SWITCH_STANDARD_API` 等。

大部分的宏以及常量、枚举等都在 `switch_types.h` 中定义。

`apr_status_t` 和返回值

在 APR 中广泛采用的一个约定是：函数返回一个状态值，用来为调用者指示成功或者是返回一个错误代码。这个类型便是 `apr_status_t`，它是在 `apr_errno.h` 中定义的，并赋予整数值。因此一个 APR 函数的常见原型就是：

```
APR_DECLARE(apr_status_t) apr_do_something(...function args...);
```

返回值应当在逻辑上进行判断，并且实现一个错误处理函数（进行回复或者对错误进行进一步的描述）。返回值 `APR_SUCCESS` 意味着成功。FreeSWITCH 也定义了类似的返回值，并以 `SWITCH_SUCCESS` 对应 `APR_SUCCESS`。这里注意一点，`SWITCH_SUCCESS` 对应该的枚举值为 0，因此，常见的错误是：

```
apr_status_t status;

status = call_some_function(... args ...);
if (status) { /* 成功? */
    return status;
} else {
    ...
}
```

上面的程序片段是错误的，如果判断是否成功，应该永远使用下面的方式来判断执行结果是否成功：

```
if (status == SWITCH_STATUS_SUCCESS)
```

在 FreeSWITCH 的历史上也曾经因为这个原因出过 Bug（高手也会犯错误）。

另外，有些函数返回一个字符串（`char *` 或者 `const char *`）、一个 `void *` 或者 `void`。这些函数就被认为没有失败条件或者在发生错误时返回一个空指针。

内存池

APR 使用内存池来方便对内存的管理。大家都知道，C 语言中的内存管理是臭名昭著的。而在 APR 中，通过使用内存池，用户在申请内存时可以不用时时刻刻释放申请到的内存，而在可以在用完后一齐释放，极大的方便了内存管理，并能防止产生大量内存碎片。

实际上，APR 中大部分的函数及资源严重依赖于内存池，如创建一个 Socket 需要内存池，创建一个 Thread 也需要内存池。这种情况听起来似乎有些过分，甚至其作者也认为内存这些操作显式的依赖于池是个巨大错误³，并希望能在 2.0 时改变这个问题。

内存池一般设计用于小的内存分配，如果要申请几兆的内存，那么不建议在内存池中申请⁴。

除了这些之外，APR 在使用起来还是相当方便的。在 APR 中通常认为从内存池中申请的内存分配永远不会失败。这个假设成立的原因在于如果内存分配失败，那么系统是不可恢复的，任何错误处理也将会失败。

其他

另外，SWITCH APR 还包装了 APR 中的字符串处理、文件管理、队列、互斥锁、Socket、线程库等。除了使用 `apr_hash` 提供的哈希函数外，FreeSWITCH 还自己实现了相关的哈希函数。

2.1.3 main 函数

在此，回答一下本章开头各位朋友提出的问题。关于看源代码从哪里开始的问题，答案就是——从 main 函数开始看。

大家都知道，C 语言程序的执行都是从 `main` 开始执行的，FreeSWITCH 也不例外。打开 `src/switch.c`，在第 482 行可以看到 `main` 函数。它的主要作用就是解析命令行来的各种参数，然后把一些重要参数记到一个 `switch_core_flag_t` 结构体中。默认系统会启动的前台。在 Linux 平台上，如果执行的时候提供了 “-nc” 参数（第 750 行），则在 UNIX 类系统上会通过 `fork` 系统调用将服务启动到后台（第 1701 行将最终调用 `fork`）；在 Windows 平台上，则是通过 `FreeConsole` WINAPI 实现的。

³Since somebody else said it first, I will admit that APR’ s reliance on pools were my absolute biggest mistake in APR. I wrote an article for Linux Magazine last month where I made it very clear that pools were my biggest mistake. My personal goal for APR 2.0 is to divorce APR from pools completely, so that you can easily use pools if you want to, but you absolutely aren’ t forced to do so. And, it should be on a per allocation basis, not per application. http://mail-archives.apache.org/mod_mbox/apr-dev/200502.mbox/%3C1f1d9820502241330123f955f@mail.gmail.com%3E

⁴REMARK: There is no limitation about memory chunk size that you can allocate by `apr_palloc()`. Nevertheless, it isn’ t a good idea to allocate large size memory chunk in memory pool. That is because memory pool is essentially designed for smaller chunks. Actually, the initial size of memory pool is 8 kilo bytes. If you need a large size memory chunk, e.g. over several mega bytes, you shouldn’ t use memory pool. <http://dev.ariel-networks.com/apr/apr-tutorial/html/apr-tutorial-3.html>

总之，不管是在前台还是后台，它在初始化一些环境并设置好系统相关的路径后，就执行第 1165 行，调用 `switch_core_init_and_modload()` 函数加载各种模块。这些模块是否加载依赖于安装目录中的配置文件 `conf/autoload_configs/modules.conf` 中的设置。

```

482  int main(int argc, char *argv[])
483  {
...
750      else if (!strcmp(local_argv[x], "-nc")) {
751          nc = SWITCH_TRUE;
752      }
...
1066     if (nc) {
1067 #ifdef WIN32
1068         FreeConsole();
1069 #else
1070         if (!nf) {
1071             daemonize(do_wait ? fds : NULL);
1072         }
1073 #endif
...
1165     if (switch_core_init_and_modload(flags, nc ? SWITCH_FALSE : SWITCH_TRUE, &err) !=
↪ SWITCH_STATUS_SUCCESS) {

```

加载完所有模块后，系统核心进入 `switch_core.c:989` 的 `switch_core_runtime_loop()` 对于后台启动的实例来讲，它基本什么都不做，在 Windows 平台上，执行第 1001 行的 `WaitForSingleObject` 以等待服务终止；在 UNIX 类平台上，就是无限循环（第 1005 ~ 1007 行），其中第 1006 行相当于 `sleep 1` 秒；对于从前台启动的系统，它会在第 1011 行进入 `switch_console_loop()` 以启动一个控制台接收用户的键盘输入并打印系统运行的信息（命令输出和日志等）。

```

989  SWITCH_DECLARE(void) switch_core_runtime_loop(int bg)
990  {
...
1001      WaitForSingleObject(shutdown_event, INFINITE);
...
1005      while (runtime.running) {
1006          switch_yield(1000000);
1007      }
...
1011      switch_console_loop();

```

`switch_console_loop()` 函数在 `switch_console.c:1098` 定义。它使用跨平台的 `editline` 库用于接收用户的按键并在控制台上打印信息。在第 1176 行，启动了一个新线程执行 `console_thread` 函数。

```
1098 SWITCH_DECLARE(void) switch_console_loop(void)
1099 {
...
1176     switch_thread_create(&thread, thd_attr, console_thread, pool, pool)
```

在 `console_thread` 中（第 1044 行），也是一个循环用于接收用户输入。如果用户输入一条命令，则在检查命令的合法性后将命令放入命令历史（第 1075 行），以备以后再执行时可以使用键盘上的箭头键翻查命令历史。然后，在第 1076 行调用 `switch_console_process` 执行输入的命令并返回结果。

```
1044 static void *SWITCH_THREAD_FUNC console_thread(switch_thread_t *thread, void *obj)
1045 {
...
1075         history(myhistory, &ev, H_ENTER, line);
1076         running = switch_console_process(cmd);
```

`switch_console_process`（第 134 行）又调用了 `switch_console_execute`（第 348 行），后者最终在第 392 行调用核心提供的 `switch_api_execute`（`switch_loadable_module.c:2282`）执行输入的命令。

```
392     status = switch_api_execute(cmd, arg, NULL, istream);
```

如果用户在命令行上输入 `sofia status`，则上述命令展的结果就是：

```
status = switch_api_execute("sofia", "status", NULL, istream);
```

上述命令的执行结果将存放到 `istream` 中，最终会在某处被取出并打印到命令行上。

2.1.4 可加载模块

通过上一节的学习，我们的 FreeSWITCH 已经启动并可以接收并执行命令了。不过，我们还是往回倒一下，看看 FreeSWITCH 中的可加载模块是如何被加载的。

FreeSWITCH 的核心代码非常紧凑，大部分实际的功能都是由外围的模块实现和扩展的。

我们在上一节提到 `switch_core_init_and_modload()` 函数负责初始化和加载各种模块。它是在 `switch_core.c: 2084` 定义的。在该文件的 2110 行，完成一些初始化后它就调

用 `switch_loadable_module_init()` 进行模块的初始化, 该函数是在 `switch_loadable_module.c:1747` 定义的。在第 1761 行到 1770 行定义了各种不同平台上的动态库的扩展名, 如, 在 Windows 上, 动态链接库的扩展名是 “.dll”、在 Mac 上, 是 “.dylib”、在其它各种 UNIX 类系统上, 是 “.so”。

```
1747 SWITCH_DECLARE(switch_status_t) switch_loadable_module_init(switch_bool_t autoload)
1748 {
...
1761 #ifdef WIN32
1762     const char *ext = ".dll";
1763     const char *EXT = ".DLL";
1764 #elif defined (MACOSX) || defined (DARWIN)
1765     const char *ext = ".dylib";
1766     const char *EXT = ".DYLIB";
1767 #else
1768     const char *ext = ".so";
1769     const char *EXT = ".SO";
1770 #endif
```

在第 1772 行, 初始化了一个结构体变量 `loadable_modules`, 它是一个可加载模块的容器。

```
1772     memset(&loadable_modules, 0, sizeof(loadable_modules));
```

`loadable_modules` 定义如下:

```
struct switch_loadable_module_container {
    switch_hash_t *module_hash;
    switch_hash_t *endpoint_hash;
    switch_hash_t *codec_hash;
    switch_hash_t *dialplan_hash;
    switch_hash_t *timer_hash;
    switch_hash_t *application_hash;
    switch_hash_t *chat_application_hash;
    switch_hash_t *api_hash;
    switch_hash_t *file_hash;
    switch_hash_t *speech_hash;
    switch_hash_t *asr_hash;
    switch_hash_t *directory_hash;
    switch_hash_t *chat_hash;
    switch_hash_t *say_hash;
    switch_hash_t *management_hash;
```

```

switch_hash_t *limit_hash;
switch_mutex_t *mutex;
switch_memory_pool_t *pool;
};

```

可以看出，它主要定义了各种哈希表（`hash`）。将来，新加载的各种模块将统一由不同的哈希表管理，如`mod_sofia`将被记入`endpoint_hash`，`mod_g729`将被记入`codec_hash`等。

此外，它还使用互斥（`mutex`）来防止多线程访问。`pool`则是一个内存池。在接下来的 1773 行就紧接着初始化了这个内存池：

```

1773     switch_core_new_memory_pool(&loadable_modules.pool);

```

1780 ~ 1798 行则初始化了所有的 `hash` 及 `mutex`。其中有些 `hash` 的键（Key）是区分大小写的，用 `switch_core_hash_init()`，有些则是大小写无关的，用 `switch_core_hash_init_nocase()`。这些数据结构初始化时都需要一个内存池，因而可以看出内存池的重要性⁵。

```

1780     switch_core_hash_init(&loadable_modules.module_hash, loadable_modules.pool);
1781     switch_core_hash_init_nocase(&loadable_modules.endpoint_hash, loadable_modules.pool);
...
1798     switch_mutex_init(&loadable_modules.mutex, SWITCH_MUTEX_NESTED, loadable_modules.pool);

```

系统加载完以后，就开始加载各种模块了。在 1802 ~ 1803 行，首先加载的是 `CORE_SOFTTIMER_MODULE` 和 `CORE_PCM_MODULE` 两个模块，这两个模块是直接的核心代码中实现的，因而比较特殊。

```

1802     switch_loadable_module_load_module("", "CORE_SOFTTIMER_MODULE", SWITCH_FALSE, &err);
1803     switch_loadable_module_load_module("", "CORE_PCM_MODULE", SWITCH_FALSE, &err);

```

我们暂且不深入研究这两个模块是如何加载的，继续往下走。第 1806 行，`switch_xml_open_cfg()` 将打开 XML 配置文件中的 `modules.conf`（参见 1753 行）部分（默认在安装目录的 `conf/autoload_configs/modules.conf.xml` 中配置），经过一个 `for` 循环（1809 行）依次取得需要加载的模块的名字，并最终在第 1824 行执行 `switch_loadable_module_load_module_ex()` 加载它们。

⁵如果继续跟踪这些函数，就会看到大部分最终会调用 APR 版本的函数，如 `switch_mutex_init()` 将最终调用 `apr_thread_mutex_create()`（`switch_apr.c:2933`）。唯一例外的是 `hash`，它使用了 SQLite3 提供的 `hash` 库。

```

1806     if ((xml = switch_xml_open_cfg(cf, &cfg, NULL))) {
...
1809         for (ld = switch_xml_child(mods, "load"); ld; ld = ld->next) {
...
1824             if (switch_loadable_module_load_module_ex((char *) path, (char *) val, SWITCH_FALSE,
↳ global, &err) == SWITCH_STATUS_GENERR) {

```

然后，用同样的方法尝试加载 `post_load_modules.conf` (参见 754 行) 中配置的模块 (1839 行起)。

另外，如果上面两个配置文件中都没有找到可加载的模块 (1867)，则尝试加载所有模块 (1872~1897 行)。

```

1867     if (!count) {
1869         all = 1;
1870     }
1871
1872     if (all) {
...
1897         switch_loadable_module_load_module((char *) SWITCH_GLOBAL_dirs.mod_dir, (char *) fname,
↳ SWITCH_FALSE, &err);

```

无论如何，模块加载完毕后，将执行 `switch_loadable_module_runtime()` 函数 (第 1902 行)。该函数在第 114 行定义，关于该函数的作用我们在下一节再讲。

```

1902     switch_loadable_module_runtime();

```

接下来，我们看一下模块是怎么被加载的。模块加载最终是由 1476 行的 `switch_loadable_module_load_module_ex()` 实现的。它会首先计算欲加载的模块对应的文件名，并在 1515 行检查 `loadable_modules.module_hash` 这个哈希表判断该模块是否已被加载，如果未被加载，则在 1519 行调用 `switch_loadable_module_load_file()` 将该模块加载到内存。

```

1476 static switch_status_t switch_loadable_module_load_module_ex(
    char *dir, char *fname, switch_bool_t runtime,
    switch_bool_t global, const char **err)
1477 {
...
1515     if (switch_core_hash_find_locked(loadable_modules.module_hash,

```

```
        file, loadable_modules.mutex)) {  
...  
1519     } else if ((status = switch_loadable_module_load_file(  
        path, file, global, &new_module)) == SWITCH_STATUS_SUCCESS) {
```

`switch_loadable_module_load_file()` 是在第 1328 行定义的。这个函数我们需要好好看一下。

在第 1356 和 1360 行，它会多次尝试使用 `switch_dso_open()` 来打开相应的模块的动态链接库（`switch_dso_open` 在 `switch_dso.c:35` 中定义。在 Windows 平台上，它将使用 `LoadLibraryEx` 来打开相应的 `.dll` 库，在 Mac 和 Linux 上，它将使用 `dlopen()` 打开相应的 `.so` 文件）。

```
1328 static switch_status_t switch_loadable_module_load_file(char *path,  
        char *filename, switch_bool_t global, switch_loadable_module_t **new_module)  
1329 {  
...  
1356     dso = switch_dso_open(lib_path, load_global, &derr);  
...  
1360     dso = switch_dso_open(NULL, load_global, &derr);
```

动态库打开后，通过下面的代码找到动态库里的符号表（第 1379 行）。执行到 1402 行时，如果符号表加载成功，则将该符号表赋值给 `mod_interface_functions` 变量。

```
1379     interface_struct_handle = switch_dso_data_sym(dso, struct_name, &derr);  
...  
1402     mod_interface_functions = interface_struct_handle;
```

`mod_interface_functions` 是一个 `switch_loadable_module_function_table` 结构的结构体，它是在 `switch_type.h:2228` 中定义的：

```
2228 typedef struct switch_loadable_module_function_table {  
2229     int switch_api_version;  
2230     switch_module_load_t load;  
2231     switch_module_shutdown_t shutdown;  
2232     switch_module_runtime_t runtime;  
2233     switch_module_flag_t flags;  
2234 } switch_loadable_module_function_table_t;  
2235
```

该结构体定义了几个指向函数的指针，分别是 `load`、`shutdown` 和 `runtime`。如果被加载的模块中实现了这些函数，则这些指针指向相关的函数入口，如果没有实现，就是 `NULL`。

每个模块都必须实现 `load` 函数，它一般用于模块的初始化操作。因而在第 1403 行，`load` 函数的指针被赋值给 `load_func_ptr` 这个变量。

```
1403         load_func_ptr = mod_interface_functions->load;
```

第 1411 行，`load` 函数将被执行：

```
1411     status = load_func_ptr(&module_interface, pool);
```

`load` 函数的原型是使用 `SWITCH_MODULE_LOAD_FUNCTION(name)` 这个宏来定义的（`switch_types.h:2211`），该宏展开的结果就是

```
switch_status_t mod_xx_load(  
    switch_loadable_module_interface_t **module_interface, switch_memory_pool_t *pool)
```

因而，如果该函数执行成功，将返回 `SWITCH_STATUS_SUCCESS`，并且会初始化一个 `module_interface` 指针。然后，在第 1424 行，初始化一个 `module` 变量。

```
1424     if ((module = switch_core_alloc(pool,  
        sizeof(switch_loadable_module_t))) == 0) {
```

`module` 变量是一个如下所示的结构（在第 44 行定义）。它定义了该模块的一些参数，其中，成员 `module_interface` 就用于存放刚刚在 1411 行初始化的 `module_interface` 指针。

```
44 struct switch_loadable_module {  
45     char *key;  
46     char *filename;  
47     int perm;  
48     switch_loadable_module_interface_t *module_interface;  
49     switch_dso_lib_t lib;  
50     switch_module_load_t switch_module_load;  
51     switch_module_runtime_t switch_module_runtime;  
52     switch_module_shutdown_t switch_module_shutdown;
```

```
53     switch_memory_pool_t *pool;
54     switch_status_t status;
55     switch_thread_t *thread;
56     switch_bool_t shutting_down;
57 };
```

接下来从第 1451 行开始，对 `module` 中的各成员赋值。最后，在第 1463 行初始化 `new_module` 指针，返回到调用该函数的地方，模块加载成功。

```
1451     module->pool = pool;
1452     module->filename = switch_core_strdup(module->pool, path);
1453     module->module_interface = module_interface;
1454     module->switch_module_load = load_func_ptr;
...
1463     *new_module = module;
```

总之，模块加载的流程就是，首先找到模块对应的动态库文件，然后打开并找到符号表，接下来执行模块中的 `load` 函数。另外，如果模块定义了 `runtime` 及 `shutdown` 函数，也将一并记录到 `module` 结构的 `switch_module_runtime` 及 `switch_module_shutdown` 成员变量中。

`switch_loadable_module_load_file()` 执行完毕后，得到了一个 `new_module` 指针，并返回到第 1519 行。紧接着在第 1520 行就执行 `switch_loadable_module_process()` 函数，它使用模块的文件名（`file`）和我们新得到的 `new_module` 结构作为参数传入。

```
1519     } else if ((status = switch_loadable_module_load_file(
        path, file, global, &new_module)) == SWITCH_STATUS_SUCCESS) {
1520         if ((status = switch_loadable_module_process(file, new_module))
```

`switch_loadable_module_process` 函数是在 133 行定义的，在第 138 行，有如下语句：

```
138     new_module->key = switch_core_strdup(new_module->pool, key);
```

该行初始化 `new_module` 的 `key`，它是一个字符串，实际上就是传入的文件名。`switch_core_strdup()` 用于制作一个 `key` 的副本（`duplicate`），它需要的内存是从内存池中申请的，因而后续不需要明确的释放，在模块卸载时直接释放掉内存池就行了。

在第 140 行通过互斥的 `mutex` 来锁定全局的 `loadable_modules` 结构，并在第 141 行向其中的 `loadable_modules.module_hash` 哈希表中插入该模块，以记录该模块被加载了。

```
140     switch_mutex_lock(loadable_modules.mutex);
141     switch_core_hash_insert(loadable_modules.module_hash, key, new_module);
```

第 143 行进行判断，如果被加载的模块实现了一个 `endpoint_interface`，则在第 150 行将它记录到 `loadable_modules.endpoint_hash` 中，

```
143     if (new_module->module_interface->endpoint_interface) {
...
150         switch_core_hash_insert(loadable_modules.endpoint_hash, ptr->interface_name, (const
↪ void *) ptr);
```

并于第 151 行产生一个模块加载的事件——`SWITCH_EVENT_MODULE_LOAD`，并于第 156 行发送出去：

```
151         if (switch_event_create(&event, SWITCH_EVENT_MODULE_LOAD) == SWITCH_STATUS_SUCCESS) {
...
156             switch_event_fire(&event);
```

同理，如果该模块也实现了其他的 `interface`（在同一模块中可以实现多个 `interface`，如第 163 行的 `codec_interface`，第 220 行的 `dialplan_interface` 等），则都记录到相应的哈希表中，产生相关的事件，并针对不同的 `interface` 类型可能还有不同的检查和其他处理等。

```
163     if (new_module->module_interface->codec_interface) {
...
220     if (new_module->module_interface->dialplan_interface) {
```

至此，模块就加载成功了，最后到 551 行会将锁定的临界区的资源解锁，并返回成功的状态码。

```
551     switch_mutex_unlock(loadable_modules.mutex);
552     return SWITCH_STATUS_SUCCESS;
```

不过，模块加载成功并不表示所有工作已完成。上面的函数返回后又回到 1521 行。接下来，判断如果新加载的模块定义了 `runtime` 函数的话，则启动一个新的线程，执行该 `runtime` 函数。

```
1521     if (new_module->switch_module_runtime) {
1522         new_module->thread = switch_core_launch_thread(
            switch_loadable_module_exec, new_module, new_module->pool);
1523     }
```

`runtime` 函数是循环的执行的，即只要 `runtime` 函数不返回 `SWITCH_STATUS_TERM`，则它就会被再次执行，直到该模块被卸载为止，参见第 99 ~ 101 行。

```
99     for (restarts = 0; status != SWITCH_STATUS_TERM && !module->shutting_down; restarts++) {
100         status = module->switch_module_runtime();
101     }
```

至此，模块加载的工作才算完成了。FreeSWITCH 就进入正常运行阶段了。

2.1.5 模块的结构

上一节，我们讲了核心中对可加载模块的处理，在此，我们接着来看一下可加载模块的结构。

在 `src/mod/sdk/autotools/src` 目录下有一个 `mod_example.c`，描述了一个最精减的模块的结构。其它模块可以在这基础上修改。该文件只是一个例子，有些语句默认是注释掉的，在需要的时候可以打开。

该文件不长，因此我们把它全部的内容都列在这里（除了最前面的版权信息及最后面的对编译器的注释）。真正的代码是从第 33 行开始的（为了与实际文件对应，我们对空行了编了行号）。它首先装入 `switch.h`，使得它可以引用 FreeSWITCH 核心中的公用函数（即所谓的 Core Public API）。

然后，它分别使用三个宏（在 `switch_types:2211 ~ 2213` 行定义）声明了三个函数定义：第 36 行的 `mod_example_shutdown`、第 37 行的 `mod_example_runtime` 以及第 40 行的 `mod_example_load`。其中，只有 `load` 函数是必须的，因此，其它两个默认是注释掉的。

上面只是对三个函数的前向声明，真正让这三个函数起作用的行是第 41 行。它的作用是，告诉核心，如果在加载该模块时，就要回调本模块的 `mod_example_load` 函数进行一些初始化操作（即我们上一节讲过的 `switch_loadable_module.c:1411`）。

```
33 #include <switch.h>
34
35 /*
36 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_example_shutdown);
37 SWITCH_MODULE_RUNTIME_FUNCTION(mod_example_runtime);
38 */
```

```
39
40 SWITCH_MODULE_LOAD_FUNCTION(mod_example_load);
41 SWITCH_MODULE_DEFINITION(mod_example, mod_example_load, NULL, NULL);
42
```

在这里，我们仅使用了 `load` 回调函数，如果把其它两个都用上，我们可以这样写：

```
SWITCH_MODULE_DEFINITION(mod_example,
    mod_example_load, mod_example_shutdown, mod_example_runtime);
```

这样的话，当模块被加载的时候就会回调 `load`、启动一个新线程运行 `runtime`，并在模块被卸载的时候执行 `shutdown` 函数。

当该模块被加载时（如在 FreeSWITCH 控制台上执行 `mod_example`），则就回调到下面的函数。该函数在参数中会传过来一个空指针（实际上一对个双重指针）——`module_interface`，我们需要被始化这个指针（第 46 行）。在 `module_interface` 初始化完成后，我们就打印一条日志（第 48 行），并返回 `SWITCH_STATUS_SUCCESS` 值（第 51 行）以表示初始化成功。如果由于任何原因导致初始化失败（如不能连接数据库，不能申请相关资源等），则可以返回 `SWITCH_STATUS_FALSE` 或其它错误值。

```
43 SWITCH_MODULE_LOAD_FUNCTION(mod_example_load)
44 {
45     /* connect my internal structure to the blank pointer passed to me */
46     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
47
48     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Hello World!\n");
49
50     /* indicate that the module should continue to be loaded */
51     return SWITCH_STATUS_SUCCESS;
52 }
53
```

下面是在模块被卸载时的回调函数，可以用于断开数据库连接、释放内存、清理相关现场等。在此，我们的模块没有申请什么资源，因而直接返回成功（第 58 行）。

```
54 /*
55     Called when the system shuts down
56 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_example_shutdown);
57 {
58     return SWITCH_STATUS_SUCCESS;
59 }
```

```
60 */
61
```

如果 `runtime` 函数存在的话，系统核心会启动一个新线程来调用该函数。在这里，可以是一个无限循环（如第 67 行，当然要记住给无限循环终止条件，否则该模块就不能卸载了），也可以在执行一个时间后返回一个状态值，只要返回值不是 `SWITCH_STATUS_TERM`，该函数就会被再次调用。

```
62 /*
63  If it exists, this is called in it's own thread when the module-load completes
64  If it returns anything but SWITCH_STATUS_TERM it will be called again automaticly
65  SWITCH_MODULE_RUNTIME_FUNCTION(mod_example_runtime);
66  {
67      while(looping)
68      {
69          switch_yield(1000);
70      }
71      return SWITCH_STATUS_TERM;
72  }
73 */
```

这就是在 FreeSWITCH 中可加载模块的大体结构。我们在后面将会看到编写一个新模块的实际例子。

2.1.6 Session 和 Channel

在 FreeSWITCH 核心中，与通话最相关的部分莫过于 Session 和 Channel 了。FreeSWITCH 是一个 B2BUA，因此，它的每一条参与通话的腿（Leg）都是一个 Channel。而 Session 则比 Channel 更高级一些，它用于描述一次会话，也就是说，虽然 Session 与 Channel 总是一一对应的，但前者管的事更多一些。也可以这样认为，Session 更关注于控制信令层，而 Channel 更关注于媒体层。

每当有一个电话到来时，或者每次从 FreeSWITCH 中发起一路通话时，便建立一个 Session（同时生成一个 Channel）。

用于标志 Session 的是一个 `struct switch_core_session` 的结构体，它的部分定义如下：

```
107 struct switch_core_session {
108     switch_memory_pool_t *pool;
109     switch_thread_t *thread;
110     switch_thread_id_t thread_id;
111     switch_endpoint_interface_t *endpoint_interface;
112     switch_size_t id;
```

```
113     switch_session_flag_t flags;
114     switch_channel_t *channel;
...
```

可以看出，在 `switch_core_session` 中有一个指向 `channel` 的指针（第 114 行）。上述的定义是在 `src/include/private/switch_core_pvt.h` 中定义的。之所以在 `private`（私有的）下面定义，是因为它不想让 FreeSWITCH 核心之外的应用知道 Session 中的细节。也就是说，其它系统如果使用 FreeSWITCH 的库的话，或者即使用在 FreeSWITCH 内部的模块中，也看不到 Session 内部的东西。如果一段代码需要知道与 Session 相关的 Channel，它只能用 `switch_core_session_get_channel(session)` 函数从 session 变量中取得，而不能直接调用 `session->channel`。当然，如果这样说还不是太明白的话，看一看该文件前面的注释：

```
29  * switch_core.h -- Core Library Private Data (not to be installed into the system)
30  * If the last line didn't make sense, stop reading this file, go away!,
31  * this file does not exist!!!!
```

它的大意是说，该文件的内容是私有的，不需要看，就当它不存在！因而，我们也没必须深入研究了。

Session 是由 `switch_core_session_request_uuid`（第 2259 行）函数生成的。

```
2259 SWITCH_DECLARE(switch_core_session_t *) switch_core_session_request_uuid(
2260     switch_endpoint_interface_t *endpoint_interface,
2261     switch_call_direction_t direction,
2262     switch_originate_flag_t originate_flags,
2263     switch_memory_pool_t **pool, const char *use_uuid)
```

在该函数中，它会检查是使用现有的内存池（第 2319 行）还是创建一个新的内存池（第 2322 行）。然后在该内存池上为 `session` 结构体变量申请内存空间（第 2325 行），并将它的 `pool` 成员变量指向该内存池（第 2326）行。这样，我们就有了一个 Session 了，并且，以后与该 Session 有关的内存申请都可以在该内存池中申请。该内存池会在 Session 消亡时释放，因而，很大程度地方便了内存管理。

```
2318     if (pool && *pool) {
2319         usepool = *pool;
2320         *pool = NULL;
2321     } else {
2322         switch_core_new_memory_pool(&usepool);
```

```
2323     }
2324
2325     session = switch_core_alloc(usepool, sizeof(*session));
2326     session->pool = usepool;
2327
```

接下来，我们看到，它立即在该内存池中又为它对应的 `channel` 申请了内存，并且，对 `channel` 进行了初始化，并将 Channel 的当前状态设为 `CS_NEW`。

```
2330     if (switch_channel_alloc(&session->channel, direction, session->pool) != SWITCH_STATUS_SUCCESS) {
2331         abort();
2332     }
2333
2334     switch_channel_init(session->channel, session, CS_NEW, 0);
```

如果在调用该函数时提供了一个 `uuid`，则使用它（第 2344 行），否则的话，则自动生成一个（第 2346 行），它用于标志一个 Channel。接下来，就可以设置通道变量了，如第 2350 ~ 2351 行。

```
2343     if (use_uuid) {
2344         switch_set_string(session->uuid_str, use_uuid);
2345     } else {
2346         switch_uuid_get(&uuid);
2347         switch_uuid_format(session->uuid_str, &uuid);
2348     }
2349
2350     switch_channel_set_variable(session->channel, "uuid", session->uuid_str);
2351     switch_channel_set_variable(session->channel, "call_uuid", session->uuid_str);
```

接下来就初始化与 Session 相关的各种变量、申请相关内存、初始化 Mutex、锁、队列等。最后，将该 Session 对应的 UUID 记录到一个核心的哈希表中（第 2381 行），至此，Session 的初始化基本上就结束了。

```
2381     switch_core_hash_insert(session_manager.session_table, session->uuid_str, session);
2382     session->id = session_manager.session_id++;
2383     session_manager.session_count++;
```

然后，第 1876 行的 `switch_core_session_thread_launch` 函数会被调用，来启动一个新的线程。

```
1876 SWITCH_DECLARE(switch_status_t) switch_core_session_thread_launch(  
    switch_core_session_t *session)
```

由于启动一个新的线程是比较费时的操作，因而在系统内部维护了一个线程池。在该函数第 1888 行，就判断核心参数是否启用线程池（默认启用），如果是的话，则就在第 1889 行把该 Session 推到线程池队列中去。否则的话，就在第 1906 行启动一个新线程，执行 `switch_core_session_thread` 函数（当然在线程池队列中找到一个可用的线程后，也会执行该函数）。

```
1888     if (switch_test_flag((&runtime), SCF_SESSION_THREAD_POOL)) {  
1889         return switch_core_session_thread_pool_launch(session);  
1890     }  
...  
1906     if (switch_thread_create(&thread, thd_attr, switch_core_session_thread,  
        session, session->pool) == SWITCH_STATUS_SUCCESS) {  
1907         switch_set_flag(session, SSF_THREAD_STARTED);  
1908         status = SWITCH_STATUS_SUCCESS;
```

`switch_core_session_thread` 是在第 1555 行定义的。它有两个传入参数，一个是当前线程的指针 `thread`，另一个是一个无类型的（`void *`）指针 `obj`，该 `obj` 实际上就是我们的 Session 指针，因此，在第 1557 行，初始化了一个 `session` 变量并指向与 `obj` 指针同样的地址。在进行一些初始化操作后，便执行 1565 行的 `switch_core_session_run` 函数。

```
1555 static void *SWITCH_THREAD_FUNC switch_core_session_thread(switch_thread_t *thread, void *obj)  
1556 {  
1557     switch_core_session_t *session = obj;  
...  
1565     switch_core_session_run(session);
```

转了这么一大圈，我们终于找到了关键的地方。`switch_core_session_run` 实际上是一个状态机。该状态机的定义在 `switch_types.h` 中，它是一个枚举类型的定义，内容如下：

```
1179 typedef enum {  
1180     CS_NEW,                // 新建  
1181     CS_INIT,               // 已初始化  
1182     CS_ROUTING,            // 路由  
1183     CS_SOFT_EXECUTE,       // 准备好执行，可由第三方控制  
1184     CS_EXECUTE,            // 执行 Dialplan 中的 App  
1185     CS_EXCHANGE_MEDIA,     // 与另一个 Channel 在交换媒体
```

```

1186     CS_PARK,                // Park, 等待进一步的命令指示
1187     CS_CONSUME_MEDIA,       // 消费掉媒体并丢弃
1188     CS_HIBERNATE,           // 没事可干, Sleep
1189     CS_RESET,               // 重置
1190     CS_HANGUP,              // 挂机, 结束信令和媒体交互
1191     CS_REPORTING,           // 收集呼叫信息 (如写 CDR 等)
1192     CS_DESTROY,             // 待销毁, 退出状态机
1193     CS_NONE                  // 无效
1194 } switch_channel_state_t;

```

`switch_core_session_run` 函数是在 `switch_core_state_machine.c:414` 中定义的。

```

414 SWITCH_DECLARE(void) switch_core_session_run(switch_core_session_t *session)

```

该函数主要的功能就是执行一个循环, 只要该 Session 所对应的 Channel 的状态不是 `CS_DESTROY`, 它就会一直循环。

```

449     while ((state = switch_channel_get_state(session->channel)) != CS_DESTROY) {

```

在循环体内, 就使用了一些 `switch/case` 语句, 还决定在不同的状态执行哪些代码段或函数, 部分代码如下。其中, 有一些关键的代码段被提取出来, 放到一个名为 `STATE_MACRO` 宏中执行了, 而该宏就是状态机中最关键的部分。

```

483     switch (state) {
484     case CS_NEW:
...
487     case CS_DESTROY:
488         goto done;
489     case CS_REPORTING: /* Call Detail */
...
495     case CS_HANGUP: /* Deactivate and end the thread */
...
502     case CS_INIT: /* Basic setup tasks */
503         {
...
506             STATE_MACRO(init, "INIT");
...
521     case CS_ROUTING: /* Look for a dialplan and find something to do */
522         STATE_MACRO(routing, "ROUTING");
523         break;

```

```
524         case CS_RESET:      /* Reset */
525             STATE_MACRO(reset, "RESET");
526             break;
```

`STATE_MACRO` 宏是在第 356 行定义的, 该宏比较复杂, 其实展开以后, 大致就想当于下面的代码片断 (这里用伪代码表示):

```
1 do {
2     switch_log_printf("INIT");
3     if (driver_state_handler->on_init) {
4         driver_state_handler->on_init(session);
5     }
6 } while (0)
```

这里以 `CS_INIT` 状态为例, 从伪代码可以看出, 该宏整个套在一个大的 “`do { ... } while (0)`” 单次循环体中⁶。如果在该 Channel 对应的 Endpoint 的底层驱动中在当前状态上安装了回调函数, 则回执行该回调函数 (第 4 行)。

当然, 实际的情况比这个情况要复杂的多, 它更类似于下面的样子。即, 除了在 Channel 对应的 Endpoint 的底层驱动中可以安装回调函数外, 在其它的模块或第三方应用中也可以在 Channel 上安装回调函数, 以后在 Channel 的状态机状态发生变化时得到回调。那么, 下面这段伪代码就有机会执行额外的回调函数 (第 5 行)。

```
1 do {
2     switch_log_printf("INIT");
3     if (!driver_state_handler->on_init ||
4         (driver_state_handler->on_init(session) == SWITCH_STATUS_SUCCESS) ) {
5         while (do_extra_handlers) {
6             do_extra_handlers(...);
7         }
8     }
9 } while (0)
```

实际上, 实际的情况比这个还要复杂。但这里我们就不深入研究了。总之, 你只需要知道在 Channel 的核心状态机上可以安装回调函数, 并在状态发生变化时得到回到。如果对细节特别感兴趣的读者也可以使用 “`gcc -E`” 命令将该源文件中的宏展开看一看。

⁶这是一个处理宏的技巧, 有如支持定义局部变量、支持复杂的宏定义而不用担心产生副作用等多种好处, 在 Linux 内核中就使用了这种技术, 参见: <http://kernelnewbies.org/FAQ/Dowhile0>。

从这一段的代码我们知道，Session 与 Channel 是息息相关的。初始化了 Session 之后，就有了 Channel，而状态机全部都是在 Channel 上实现的（其中 `CS_INIT` 中的 `CS` 便是 Channel State 的意思）。当然，核心中也定义了很多专门对 Channel 操作的函数，大部分都是在 `switch_channel.c` 中实现的。这些函数的名称和代码看起来都很直观，在这里我们就不多讲了，等到后面用到的时候再个别进行说明。

2.1.7 SWITCH IVR

大部分媒体处理逻辑都是在 `switch_ivr_*.c` 中实现的，有多个源代码实现了不同的 `switch_ivr` 逻辑，如 `switch_ivr_async.c` 进行异步处理，`switch_ivr_bridge.c` 处理话路桥接等。在此，我们先来看一个简单的 `echo` 应用。关于 `echo` App 我们大家都已经很熟悉了，我们知道它的作用就是将收到的媒体（音频或视频）原样再发回去。下面，我们就看一看它是如何实现。

在通话执行到 `echo` App 时，将最终执行到 `switch_ivr_async.c:629` 定义的 `switch_ivr_session_echo` 函数。由于 `echo` 应用是需要媒体的，如果在执行 `echo` 时电话还没有应答（如在 SIP 应用中还没有收到或发送 200 OK），则它会在第 636 行调用 `switch_channel_pre_answer` 试图在电话应答之前建立媒体连接（如果在 SIP 应用中将发送带 SDP 的 183 消息以尝试建立媒体连接）。当然，这是一个小的细节，我们继续往下看。

```

629 SWITCH_DECLARE(switch_status_t) switch_ivr_session_echo(switch_core_session_t *session,
↳ switch_input_args_t *args)
630 {
...
636     if (switch_channel_pre_answer(channel) != SWITCH_STATUS_SUCCESS) {
637         return SWITCH_STATUS_FALSE;
638     }

```

在该函数中，第 644 行执行一个 `while` 循环，只要该 Channel 是正常的（由 `switch_channel_ready` 判断，它会检查一系列参数，在 Channel 正常建立时将返回真，挂机或其它错误情况时将返回假），便会一直循环。然后，在第 645 行，调用核心的函数 `switch_core_session_read_frame` 从该 Channel 中读取一帧的数据（这里的一帧如果在 SIP 应用中就是一个 RTP 包中的数据，如，可能是 20 毫秒的音频数据）。接下来在第 646 行通过一个宏判断读到的数据是否有效，如果无效就跳出循环（647 行）。如果数据有效，就继续进行。第 656 行用于处理该 Channel 上相关的事件，如检查 DTMF 等。在收到 DTMF 的情况下会调用相关的回调函数（第 665 行）。

```

644     while (switch_channel_ready(channel)) {
645         status = switch_core_session_read_frame(session, &read_frame, SWITCH_IO_FLAG_NONE, 0);
646         if (!SWITCH_READ_ACCEPTABLE(status)) {
647             break;

```

```
648         }  
  
656         switch_ivr_parse_all_events(session);  
...  
665         if (switch_channel_has_dtmf(channel)) {  
...
```

我们跳过一些细节，走到第 694 行，可以看到它又调用了 `switch_core_session_write_frame` 将收到的数据写回了 Session 中，然后这些音频数据就会发到远端。当然，如果该 Channel 上有视频的话，它也会进行相关的处理，我们暂时忽略视频的处理代码。

总之，该函数主要的功能就是调用了 `switch_core_session_read_frame` 读取音频数据，并通过 `switch_core_session_write_frame` 写回去。有关这两个函数的实现我们将在下一节讲到。

2.1.8 Core IO

在上一节我们讲到，`switch_core_session_read_frame` 用于读数据，而 `switch_core_session_write_frame` 用于写数据。这两个函数是在 `switch_core_io.c` 中定义的。它们都非常长，因此，我们在此只简单分析一下关键点。

`switch_core_session_read_frame` 是在第 147 行定义的。

```
147 SWITCH_DECLARE(switch_status_t) switch_core_session_read_frame(  
    switch_core_session_t *session, switch_frame_t **frame,  
    switch_io_flag_t flags, int stream_id)
```

一般来说，在 SIP 应用中，都会每 20 毫秒收到一帧音频数据，但也不是绝对的。如果在读的过程中读不到数据但又不至于产生错误，该函数就返回一个静音包（CNG，即 Comfort Noise Generation，可以根据它产生舒适噪音）。

```
167     *frame = &runtime.dummy_cng_frame;  
168     return SWITCH_STATUS_SUCCESS;
```

除此之外，该函数中还有很多检查判断，我们就不详细看了。下面直接跳到第 246 行。该行会调用底层的 Endpoint 提供的 `read_frame` 回调函数来读数据。由于 FreeSWITCH 支持不同的 Endpoint，因此，这里使用回调函数的机制屏蔽各 Endpoint 的不同特性。如，在 SIP 应用中，媒体数据将从 RTP 中读取，在 `mod_portaudio` 中，数据是从本地声卡读取的，而在 `mod_freetdm` 应用中，数据是从硬件的 TDM 板卡驱动中读取的。总之，核心层并不知道这些数据是从哪里来的，只是说，我想要一帧的数据，具体这一帧数据怎么来，得看底层的驱动是怎么实现的。

总之，第 246 行判断 Endpoint 底层的驱动是否实现了 `read_frame` 回调，如果实现了，就在第 249 行调用该回调函数读取数据。然后，判断当前的 Session 是否注册了其它的事件钩子（`event_hook`），如果注册了的话，也调用钩子里的 `read_frame` 回调函数（第 250 行）。也就是说，除了 Endpoint 之外，其它的函数或模块中也可以调用 `switch_core_event_hook_add_*` 一族的函数来安装相关的回调函数，以便在适当的时候得到回调。因此，在第 250 行，它便是检查是否有通过 `switch_core_event_hook_add_read_frame` 函数注册的钩子，如果有的话，就在第 251 行调用。

```

246     if (session->endpoint_interface->io_routines->read_frame) {
...
249         if ((status = session->endpoint_interface->io_routines->read_frame
              (session, frame, flags, stream_id)) == SWITCH_STATUS_SUCCESS) {
250             for (ptr = session->event_hooks.read_frame; ptr; ptr = ptr->next) {
251                 if ((status = ptr->read_frame(session, frame, flags, stream_id)) !=
                    SWITCH_STATUS_SUCCESS) {

```

我们以前也提到过，录音等应用都是使用 Media Bug 来实现的，就是说往一个 Channel 上安装了一个 Media Bug 就是相当于给水管安装了一个“三通”。在第 309 行，就是检查该 Channel 上有没有安装“三通”，如果安装了的话，便在第 316 行一个一个的把它们找出来，并调用它们指定的回调函数（第 341 行）。而在回调函数中可以取到我们在上面第 249 行读到的这一帧的数据，并使用它（在录音应用中典型的是将这些声音数据写到录音文件中去）。

```

309     if (session->bugs && !((*frame)->flags & SFF_CNG) && !((*frame)->flags & SFF_NOT_AUDIO)) {
...
316         for (bp = session->bugs; bp; bp = bp->next) {
...
339             if (bp->callback) {
340                 bp->native_read_frame = *frame;
341                 ok = bp->callback(bp, bp->user_data, SWITCH_ABC_TYPE_TAP_NATIVE_READ);

```

读者不要把这些回调都搞混了。我们简单总结一下，第 249 行的 `io_routines` 中的回调是从底层的驱动中读媒体数据，还其它的如 250 行的 `event_hooks` 中的回调及第 341 行中的 Media Bug 中的回调是使用这些数据。

接着往下看。在第 464 行会判断读到的数据是否需要转码。如果需要的话，程序执行到第 561 行就会执行解码操作。不管读到的数据是什么编码的（PCMU、PCMA、iLBC 等），都会先转换成一种中间的编码格式 L16（16 位的线性编码）。

```

464     if (switch_test_flag(session, SSF_READ_TRANSCODE) &&
        !need_codec && switch_core_codec_ready(session->read_codec)) {

```

```

...
561         status = switch_core_codec_decode(codec,
562             session->read_codec,
563             read_frame->data,
564             read_frame->datalen,
565             session->read_impl.actual_samples_per_second,
566             session->raw_read_frame.data,
                &session->raw_read_frame.datalen,
                &session->raw_read_frame.rate,
567             &read_frame->flags);

```

除了编解码转换外，FreeSWITCH 还支持码率的转换，如将 48000Hz 的高清音频转换成 8000Hz 的窄带音频等。第 792 即是调用转码处理的函数。

```

789         if (session->read_resampler) {
792             switch_resample_process(session->read_resampler,
                data, (int) read_frame->datalen / 2);

```

如果该函数取出的数据用于另外一个 Session，而另外的 Session 可能使用不同的语音编码，则这个时候就需要使用对方的编码将 L16 线性编码的数据编码成对方需要的编码，该操作是在第 835 行调用的。

```

835         status = switch_core_codec_encode(session->read_codec,
836             enc_frame->codec,
837             enc_frame->data,
838             enc_frame->datalen,
839             session->read_impl.actual_samples_per_second,
840             session->enc_read_frame.data,
                &session->enc_read_frame.datalen,
                &session->enc_read_frame.rate, &flag);

```

该函数非常长，里面还有更多的细节，我们就不一一研究了。总之，它的就要作用就是，从底层的 Endpoint 驱动中读取一帧数据，然后调用各种回调函数，并将读到的数据返回（通过 147 行的 `frame` 双重指针返回），如果对方需要返回特定编码的数据，则在函数内部执行转码操作（解码和编码），以返回编码后的数据。

接下来我们也来简单看一下 `switch_core_session_write_frame` 函数，它是在 1025 行定义的：

```

1025 SWITCH_DECLARE(switch_status_t) switch_core_session_write_frame(
    switch_core_session_t *session, switch_frame_t *frame,

```

```
switch_io_flag_t flags,  
1026     int stream_id)
```

如果该 Session 处于 Proxy packet 状态（[SFF_PROXY_PACKET](#)），则它会快速的在第 1072 行调用 [perform_write](#) 函数将数据发送出去：

```
1072     status = perform_write(session, frame, flag, stream_id);
```

否则的话，它也会进行一系列的检查，并进行必要的操作。如它也会在必要的时候对数据进行解码和编码操作：

```
1173     status = switch_core_codec_decode(frame->codec, ...  
...  
1467     status = switch_core_codec_encode(session->write_codec, ...
```

或者进行码率转换：

```
1258     switch_resample_process(session->write_resampler, ...
```

或者对 Media Bug 回调进行处理：

```
1275     if (session->bugs) { ...
```

通过在写（[write](#)）的时候增加一个 Media Bug，可以在媒体数据实际发送出去之前对数据进行修改和替换（如果把读的 Media Bug 理解成录音和监听，那么写的 Media Bug 就可以理解为插话——在一个通话中突然插入另外一个人的语音或者播放一段音乐）。

在后面，它在第 1415 行也是调用 [perform_write](#) 将数据发送出去。

```
1415     status = perform_write(session, write_frame, flags, stream_id);
```

[perform_write](#) 函数是在第 961 行定义的，它最终将会调用 Endpoint 中的 [io_routines](#) 里的 [write_frame](#) 回调函数并数据发送出去（第 1013）行，并在第 1015 行回调相关的 [event_hooks](#) 里的回调函数（如果有的话）。

```

1012     if (session->endpoint_interface->io_routines->write_frame) {
1013         if ((status = session->endpoint_interface->io_routines->write_frame(
            session, frame, flags, stream_id)) == SWITCH_STATUS_SUCCESS) {
1014             for (ptr = session->event_hooks.write_frame; ptr; ptr = ptr->next) {
1015                 if ((status = ptr->write_frame(session, frame, flags, stream_id)) ...

```

上面,我们讲了音频媒体的读写处理。如果是视频数据,则也有对应的 `switch_core_media_read_video_frame` 以及 `switch_core_media_write_video_frame`, 实现逻辑都差不多。只是,目前 FreeSWITCH 核心中还没有对视频转码的处理,因此,代码要比音频部分简单得多。

总之,系统核心的 IO 操作屏蔽了底层的数据流读、写(收、发)细节,各种需要处理媒体的应用只需要调用核心的 IO 函数进行数据的读、写操作,而不用考虑底层的不同。同时,这种架构使得增加一种新的 Endpoint 非常容易——只需要增加一个 Endpoint 的逻辑结构,安装相应的回调函数,并调用更底层的驱动程序或者协议库进行媒体流读、写即可。

2.1.9 Core Media

Core Media 是用于在核心进行媒体协商和处理的。这些代码原来是在 `mod_sofia` 模块中,但后来为了增加 WebRTC 的支持,把这一部分代码独立出来,放到了 `switch_core_media.c` 中。它目前主要是处理使用 SDP 描述的媒体(如基于 RTP 的媒体)。

如果 Endpoint 中需要 RTP 媒体支持,则它可以在 Session 中建立一个媒体句柄,然后通过 `session->media_handle` 来引用它。通过使用 Core Media,可以隐藏一个 SDP 媒体协商及 RTP 处理的细节,使得开发基于 RTP 的媒体程序更加简单。

在 Core Media 中, `switch_core_media_read_frame` 函数用于从底层的 RTP 中读一帧数据,其中,媒体的类型(`type` 参数)可以是 `SWITCH_MEDIA_TYPE_AUDIO` 或 `SWITCH_MEDIA_TYPE_VIDEO`, 分别表示读音频还是视频数据。在 Core Media 内部,有一个媒体引擎参数,它目前定义了音频和视频两个引擎组成的数组,在第 1414 行可以通过 `engines[type]` 找到所需要的媒体引擎。

```

1395     SWITCH_DECLARE(switch_status_t) switch_core_media_read_frame(
            switch_core_session_t *session, switch_frame_t **frame,
1396     switch_io_flag_t flags, int stream_id, switch_media_type_t type)
    ...
1414     engine = &smh->engines[type];

```

在第 1440 行,它调用 RTP 相关的函数 `switch_rtp_zerocopy_read_frame` 来读取一帧。

```

1440     status = switch_rtp_zerocopy_read_frame(engine->rtp_session, &engine->read_frame, flags);

```

同样，`switch_core_media_write_frame` 函数（第 1767 行）则用于调用底层的 `switch_rtp_write_frame` 函数向发送 RTP 数据（第 1816 行）。

```

1767 SWITCH_DECLARE(switch_status_t) switch_core_media_write_frame(switch_core_session_t *session,
1768                                                                 switch_frame_t *frame, switch_io_flag_t flags, int
↪  stream_id, switch_media_type_t type)
...
1816     if (!switch_rtp_write_frame(engine->rtp_session, frame)) {

```

当然，除了媒体读写以后，Core Media 中还有 `switch_core_media_negotiate_sdp` 函数用于媒体协商、`switch_core_media_activate_rtp` 用于启动 RTP 收发等的函数，我们在此就不多讲了。

2.1.10 Core RTP

FreeSWITCH 中的 RTP 媒体收、发都是在 `switch_rtp.c` 中实现的。我们在上一节提到过，在 Core Media 中，会调用 `switch_rtp zerocopy_read_frame` 来读取一帧数据。下面，我们先来看一下这个函数。

该函数是在第 5502 行定义的，它的输入参数 `rtp_session` 是一个 `switch_rtp_t` 类型的指针，它唯一标志了一个 RTP 连接。第二个参数 `frame` 是一个 `switch_frame_t` 类型的指针，它用于存放读到的数据。它主要是在第 5510 行调用 `rtp_command_read` 从底层的 Socket 中读取数据，并用读到的数据去填充 `frame` 指针指向的结构体。

```

5502 SWITCH_DECLARE(switch_status_t) switch_rtp_zerocopy_read_frame(
    switch_rtp_t *rtp_session, switch_frame_t *frame,
    switch_io_flag_t io_flags)
5503 {
...
5510     bytes = rtp_common_read(rtp_session, &frame->payload, &frame->pmap, &frame->flags, io_flags);

```

从第 5512 行开始，很容易看到 `switch_frame_t` 结构体的结构（该结构在 `switch_frame.h:44` 定义，我们就不再列出具体的定义了）。其中，其成员变量 `data` 指向读到的数据；`packet`（第 5513 行）则指向 RTP 消息的开始（比 `data` 多一个 RTP 包头，一般是 12 个字节）；第 5514 行是 `packet` 的长度；第 5521 行是时间戳；第 5522 行是序号；第 5523 行是同步源标志；第 5524 行是 Marker 标志；第 4485 行是真正的媒体数据长度。

```

5512     frame->data = RTP_BODY(rtp_session);
5513     frame->packet = &rtp_session->recv_msg;

```

```
5514     frame->packetlen = bytes;
5515     frame->source = __FILE__;
5521     frame->timestamp = ntohl(rtp_session->recv_msg.header.ts);
5522     frame->seq = (uint16_t) ntohs((uint16_t) rtp_session->recv_msg.header.seq);
5523     frame->ssrc = ntohl(rtp_session->recv_msg.header.ssrc);
5524     frame->m = rtp_session->recv_msg.header.m ? SWITCH_TRUE : SWITCH_FALSE;
...
5585     frame->datalen = bytes;
5586     return SWITCH_STATUS_SUCCESS;
5587 }
```

`rtp_common_read` 是在第 4674 行定义的，它又根据不同的情况在第 4725 和第 4846 行分别调用 `read_rtp_packet` 来读取数据。

```
4674 static int rtp_common_read(switch_rtp_t *rtp_session,
                             switch_payload_t *payload_type,
4675                             payload_map_t **pmapP, switch_frame_flag_t *flags,
                             switch_io_flag_t io_flags)
...
4725     status = read_rtp_packet(rtp_session, &bytes, flags, SWITCH_FALSE);
...
4846     status = read_rtp_packet(rtp_session, &bytes, flags, SWITCH_TRUE);
```

`read_rtp_packet` 又最终在第 4152 行调用 `switch_socket_recvfrom` 函数从真正的 Socket 中读取数据。`switch_socket_recvfrom` 仅仅是对 APR 库中的 `apr_socket_recvfrom` 的一个简单封装，根据不同的平台调用不同的函数对 Socket 进行读取。

```
4137 static switch_status_t read_rtp_packet(switch_rtp_t *rtp_session,
                                         switch_size_t *bytes, switch_frame_flag_t *flags,
                                         switch_bool_t return_jb_packet)
4138 {
...
4152     status = switch_socket_recvfrom(rtp_session->from_addr,
                                     rtp_session->sock_input, 0,
                                     (void *) &rtp_session->recv_msg, bytes);
```

在上述的这一连串的读取函数中，有一连串的对读取到的数据进行检查的地方，保证了读取到的数据的正确性和安全性。同时，FreeSWITCH 中对有些不规范的 RTP 协议实现也适当进行了一些妥协，以便于跟那些设备对接⁷。

⁷甚至你可以在 `switch_types.h` 中发现一个 `switch_rtp_bug_flag_t` 枚举结构，里面列出了诸多已知的其它设备的 Bug，以及作者的抱怨。

与读数据相反，在写数据时，则需要先初始化好一个 `switch_frame_t` 的 `frame` 帧结构，然后调用 `switch_rtp_write_frame` 进行写（发送，第 6110 行）。当然，根据不同的条件，它或者在第 6514 行调用 `switch_socket_sendto` 直接发送，或者在第 6275 行调用 `rtp_common_write` 再进行深入的设置并发送，具体细节在此我们就不深入研究了。

```

6110 SWITCH_DECLARE(int) switch_rtp_write_frame(switch_rtp_t *rtp_session, switch_frame_t *frame)
6111 {
...
6154     if (switch_socket_sendto(rtp_session->sock_output,
                             rtp_session->remote_addr, 0, frame->packet, &bytes)
...
6275     return rtp_common_write(rtp_session, send_msg, data, len, payload, ts, &frame->flags);

```

前面我们说过，所有的 RTP 连接都是由一个 `switch_rtp_t` 类型的变量（如 `rtp_session`）来标志的。可以使用 `switch_rtp_new` 函数来新建一个新的 `rtp_session`。其函数定义如下：

```

3140 SWITCH_DECLARE(switch_rtp_t *) switch_rtp_new(const char *rx_host,
3141         switch_port_t rx_port,
3142         const char *tx_host,
3143         switch_port_t tx_port,
3144         switch_payload_t payload,
3145         uint32_t samples_per_interval,
3146         uint32_t ms_per_packet,
3147         switch_rtp_flag_t flags[SWITCH_RTP_FLAG_INVALID],
         char *timer_name, const char **err, switch_memory_pool_t *pool)

```

其中，`rx_host` 及 `rx_port` 是本端的 IP 地址和端口号，`tx_host` 和 `tx_port` 则分别为远端的 IP 地址和端口号，并于其它参数的含义和用法可以参考源代码中具体使用该函数的地方（如 `switch_core_media.c:4647`），我们就不再赘述了。

当然，如果要释放一个 `rtp_session`，则可以使用下列函数：

```

3592 SWITCH_DECLARE(void) switch_rtp_destroy(switch_rtp_t **rtp_session)

```

关于 RTP 的代码我们就介绍这么多，有兴趣的读者可以在这些基础上再深入的研究。

2.1.11 SWITCH XML

我们知到，FreeSWITCH 的配置文件中严重依赖 XML。FreeSWITCH 对 XML 的解析是在 `switch_xml` 中实现的。

如果某个程序需要从 XML 中读取配置数据，则它会调用 `switch_xml_open_cfg` 函数首先来打开一个 XML 节点。该函数是在第 2392 行定义的。在该函数内部，它会在第 2400 行调用 `switch_xml_locate` 去查找相关的 XML 节点，并返回相关的 XML 结构指针。

```

2392 SWITCH_DECLARE(switch_xml_t) switch_xml_open_cfg(const char *file_path,
                                                    switch_xml_t *node, switch_event_t *params)
2393 {
...
2400     if (switch_xml_locate("configuration", "configuration", "name", file_path, &xml, &cfg, params,
↪ SWITCH_FALSE) == SWITCH_STATUS_SUCCESS) {
2401         *node = cfg;
2402     }

```

`switch_xml_locate` 在第 1670 行定义。在该函数内部，它会首先判断一个 `BINDINGS` 全局变量中的链表结构。如果该链表非空，那么说明某个地方绑定了 XML 中的一个节点，它能动态地提供 XML。如，在 `mod_xml_curl` 中，就向核心的 XML 绑定了一个节点，然后每当执行到下面的函数需要一个这样的节点时，便回调 `mod_xml_curl` 中绑定的回调函数。这样的回调函数就是在第 1690 行执行的。

```

1670 SWITCH_DECLARE(switch_status_t) switch_xml_locate(const char *section,
1671     const char *tag_name,
1672     const char *key_name,
1673     const char *key_value,
1674     switch_xml_t *root, switch_xml_t *node, switch_event_t *params, switch_bool_t clone)
1675 {
...
1685     for (binding = BINDINGS; binding; binding = binding->next) {
...
1690         if ((xml = binding->function(section, tag_name, key_name, key_value, params, binding-
↪ >user_data))) {

```

当然，如果没有动态绑定，或者获取动态绑定的 XML 资源时发生错误，则该函数还是会尝试从本地的 XML 配置文件中查找（第 1720 行）。

```

1718     for (;;) {
1719         if (!xml) {
1720             if (!(xml = switch_xml_root())) {

```

另外，关于详细的 XML 解析算法以及其它的细节我们在此就不多解释了。

2.1.12 SWITCH Event

FreeSWITCH 中有一些功能是事件驱动的。另外，事件也是 FreeSWITCH 内部与外部进行数据交换的载体。当 FreeSWITCH 中发生状态改变，或者代码执行到某个阶段时，都会触发一些事件。同时，另外一些感兴趣的模块也可以订阅这些事件，以便在收到相应事件时执行相应的动作。从某种意义上说，这种事件机制与我们上面讲过的回调函数和钩子想要达到的效果是一样的，不同的是，事件采用“Pub/Sub”（即发布/订阅机制，也称生产者/消费者模型），建立的是一种更松的耦合关系，在使用起来更方便更自由。另外，外部的第三方系统也可以通过系统提供的接口订阅到事件，从而可以更容易的集成。

在系统初始化时，为先调用 `switch_event_init` 函数进行事件系统的初始化，该函数是在 `switch_event.c:659` 定义的，它会初始化事件系统所需的内存池、哈希表、Mutex、队列等。

```
659 SWITCH_DECLARE(switch_status_t) switch_event_init(switch_memory_pool_t *pool)
```

在 `event` 初始化完成后，核心代码会调用 `switch_event_launch_dispatch_threads`（第 618 行定义）来启动事件分发的线程，这些线程最终会执行 `switch_event_dispatch_thread` 函数（第 645 行）。

```
618 SWITCH_DECLARE(void) switch_event_launch_dispatch_threads(uint32_t max)
619 {
...
645     switch_thread_create(&EVENT_DISPATCH_QUEUE_THREADS[index],
        thd_attr, switch_event_dispatch_thread, EVENT_DISPATCH_QUEUE, pool);
```

`switch_event_dispatch_thread` 函数定义如在第 290 行。它内部执行一个无限循环，不断地从事件队列中取出一个个事件（第 322 行），然后在第 331 行调用 `switch_event_deliver` 分发出去。

```
290 static void *SWITCH_THREAD_FUNC switch_event_dispatch_thread(switch_thread_t *thread, void *obj)
291 {
...
314     for (;;) {
...
322         if (switch_queue_pop(queue, &pop) != SWITCH_STATUS_SUCCESS) {
323             continue;
324         }
329
330         event = (switch_event_t *) pop;
331         switch_event_deliver(&event);
```

```

332         switch_os_yield();
333     }

```

`switch_event_deliver` 在第 391 行定义。它通过一个两层的 `for` 循环，不断判断所有的事件中有哪些事件被哪些节点（`node`）订阅了（第 398 ~ 400 行），如果有人订阅，则调用订阅时提供的回调函数（第 402）行，即相当于把事件分发出去了。

```

391 SWITCH_DECLARE(void) switch_event_deliver(switch_event_t **event)
392 {
...
398     for (e = (*event)->event_id;; e = SWITCH_EVENT_ALL) {
399         for (node = EVENT_NODES[e]; node; node = node->next) {
400             if (switch_events_match(*event, node)) {
401                 (*event)->bind_user_data = node->user_data;
402                 node->callback(*event);

```

当然，有人订阅（消费）事件，就得有人发布（生产）事件。在发布事件之前，首先要产生一个事件。产生事件是用 `switch_event_create_subclass_detailed` 实现的，为了使用方便，在 `switch_event.h`: 381 中也定义了一个 `switch_event_create` 宏：

```

381 #define switch_event_create(event, id)
        switch_event_create_subclass(event, id, SWITCH_EVENT_SUBCLASS_ANY)

```

以及一个 `switch_event_create_subclass` 宏：

```

153 #define switch_event_create_subclass(_e, _eid, _sn)
        switch_event_create_subclass_detailed(__FILE__, (const char *) __SWITCH_FUNC__, __LINE__, _e,
        ↪ _eid, _sn)

```

其中，第一个事件都有一个事件的 ID（一个 `switch_event_type_t` 的枚举值，对应一个事件名称），以及一个可能的子类型（Subclass）。其中，事件 ID 的定义在 `switch_types.h:1754`。其中，只有当事件 ID 为 `SWITCH_EVENT_CUSTOM` 时子类型才有效。

```

1754 typedef enum {
1755     SWITCH_EVENT_CUSTOM,
1756     SWITCH_EVENT_CLONE,
1757     SWITCH_EVENT_CHANNEL_CREATE,

```

```
...
1844     SWITCH_EVENT_ALL
1845 } switch_event_types_t;
```

好了，我们回到 `switch_event.c`。在调用 `switch_event_create_subclass_detailed`（第 707 行定义）创建了一个事件的结构之后，后可以调用 `switch_event_add_header`（第 1142 行定义）添加一系列的头部数据。这些数据是一些“键/值”对。事件中还可以调用 `switch_event_add_body`（第 1190 行定义）加入一个可选的主体数据。

把所有的事件数据准备好以后，就可以通过 `switch_event_fire` 将事件发出去了。该函数实际上是在 `switch_event.h:410` 中定义的一个宏：

```
410 #define switch_event_fire(event) switch_event_fire_detailed(
    __FILE__, (const char *) __SWITCH_FUNC__, __LINE__, event, NULL)
```

该宏最终映射到 `switch_event_fire_detailed`，它是在 `switch_event.c:1944` 定义的。它会根据情况在第 1967 行调用 `switch_event_queue_dispatch_event` 或在第 1972 行调用 `switch_event_deliver_thread_pool` 将事件发送到事件队列中去。

```
1944 SWITCH_DECLARE(switch_status_t) switch_event_fire_detailed(const char *file, const char *func, int
↳ line, switch_event_t **event, void *user_data)
1945 {
...
1964     if (runtime.events_use_dispatch) {
1967         if (switch_event_queue_dispatch_event(event) != SWITCH_STATUS_SUCCESS) {
1968             switch_event_destroy(event);
1969             return SWITCH_STATUS_FALSE;
1970         }
1971     } else {
1972         switch_event_deliver_thread_pool(event);
1973     }
```

如果是第一种情况，则是调在第 383 行将事件推到事件队列里去（旧的方法，为了向后兼容）：

```
383     switch_queue_push(EVENT_DISPATCH_QUEUE, event);
```

如果是第二种情况的话（默认值），则会使用核心的线程池去分发事件。使用核心线程池进行分发的代码也很简单。首先，在第 276 行申请一块内存，用于存放一个 `switch_thread_data_t` 结

构，然后提供欲在线程池中执行的函数名称（第 280 行）以及一个可选的数据对象指针（第 281 行，在此，我们将我们事件的指针作为数据对象的指针传入，然后在后面要讲的第 265 行就能通过 `obj` 指针找到传入的事件）。最后在第 286 行从核心线程池中启动一个线程执行第 280 行指定的 `switch_event_deliver_thread` 函数。

```
272 static void switch_event_deliver_thread_pool(switch_event_t **event)
273 {
274     switch_thread_data_t *td;
275
276     td = malloc(sizeof(*td));
277     switch_assert(td);
278
279     td->alloc = 1;
280     td->func = switch_event_deliver_thread;
281     td->obj = *event;
282     td->pool = NULL;
283
284     *event = NULL;
285
286     switch_thread_pool_launch_thread(&td);
287
288 }
```

`switch_event_deliver_thread` 函数也非常简单，它只是在第 267 行调用 `switch_event_deliver`（该函数我们已经讲过了）将事件分发出去。

```
263 static void *SWITCH_THREAD_FUNC switch_event_deliver_thread(
                                switch_thread_t *thread, void *obj)
264 {
265     switch_event_t *event = (switch_event_t *) obj;
266
267     switch_event_deliver(&event);
268
269     return NULL;
270 }
```

如果一个模块或进程希望从 FreeSWITCH 事件系统中接收事件，则它应该调用 `switch_event_bind_removable` 来绑定（订阅）相关的事件。该函数的定义在第 1978 行。关于它的内部代码我们就不解释了，等到后面我们再讲一下该函数的使用方法。

```
1978 SWITCH_DECLARE(switch_status_t) switch_event_bind_removable(  
    const char *id, switch_event_types_t event, const char *subclass_name,  
1979    switch_event_callback_t callback, void *user_data, switch_event_node_t **node)
```

2.1.13 Core Codec 和 Core File

下面，我们再来看一下 Core Codec 和 Core File。之所以把两者放到一起讲，是因为他们比较类似——没有太多的业务逻辑，只是对不同的编解码和文件格式的抽象和封装。

在 Core Codec 中，提供了初始化（`init`）、编码（`encode`）、解码（`decode`）、释放（`destroy`）等函数的抽象。如 `switch_core_codec_encode` 和 `switch_core_codec_decode` 函数。他们都是在一种编码（`codec`）与另一种编码（`other_codec`）间转换。输入参数 `decoded_data` 表示未编码的（或者说是以 L16 线性编码的）数据缓冲区，而 `decoded_data_len` 则是数据的长度。同理，`encoded_data` 和 `encoded_data_len` 则是编码后的数据缓冲区和长度。如果某种编码有相应的实现代码，则它会向核心注册 `codec->implementation->encode` 和 `codec->implementation->decode` 回调函数，所以，在下面这两个函数中就直接调用这些回调函数进行编码或解码（如第 736 和第 780 行）。

```
712 SWITCH_DECLARE(switch_status_t) switch_core_codec_encode(switch_codec_t *codec,  
713    switch_codec_t *other_codec,  
714    void *decoded_data,  
715    uint32_t decoded_data_len,  
716    uint32_t decoded_rate,  
717    void *encoded_data, uint32_t *encoded_data_len,  
    uint32_t *encoded_rate, unsigned int *flag)  
718 {  
    ...  
736    status = codec->implementation->encode(codec, other_codec,  
        decoded_data, decoded_data_len,  
737        decoded_rate, encoded_data, encoded_data_len, encoded_rate, flag);  
    ...  
  
744 SWITCH_DECLARE(switch_status_t) switch_core_codec_decode(switch_codec_t *codec,  
745    switch_codec_t *other_codec,  
746    void *encoded_data,  
747    uint32_t encoded_data_len,  
748    uint32_t encoded_rate,  
749    void *decoded_data, uint32_t *decoded_data_len,  
    uint32_t *decoded_rate, unsigned int *flag)  
750 {  
    ...  
780    status = codec->implementation->decode(codec, other_codec,
```

```

781         encoded_data, encoded_data_len, encoded_rate,
        decoded_data, decoded_data_len, decoded_rate, flag);

```

与 Core Codec 类似，在 Core File 中，也提供了打开（`open`）、读（`read`）、写（`write`）、关闭（`close`）等对各种不同的文件操作的抽象。以读和写为例，虽然它的代码与 Core Codec 相比要复杂一些，但基本的功能也是回调各功能模块实现的回调函数`fh->file_interface->file_read`（如第295或324行）从文件中读取数据到内存缓冲区，或将内存中的数据通过`fh->file_interface->file_write`回调（如第452、461行）写到文件中去。

```

253 SWITCH_DECLARE(switch_status_t) switch_core_file_read(
        switch_file_handle_t *fh, void *data, switch_size_t *len)
254 {
    ...
295         if ((status = fh->file_interface->file_read(fh, fh->pre_buffer_data, &rlen)) ==
    ↪ SWITCH_STATUS_BREAK) {
    ...
324         if ((status = fh->file_interface->file_read(fh, data, len)) == SWITCH_STATUS_BREAK) {
    ...

387 SWITCH_DECLARE(switch_status_t) switch_core_file_write(
        switch_file_handle_t *fh, void *data, switch_size_t *len)
388 {

452         if ((status = fh->file_interface->file_write(fh,
        fh->pre_buffer_data, &blen)) != SWITCH_STATUS_SUCCESS) {
    ...
461         if ((status = fh->file_interface->file_write(fh, data, len)) ...

```

当然，在 Core File 接口中，除了单纯的读写外还有在缓冲区中对码率的转换以及数据适配等代码，我们就不多讨论了。

总之，在核心中，就是通过这样的抽象与回调机制实现了媒体编解码接口（Codec Interface）、文件接口（File Interface）以及我们前面提到的终点接口（Endpoint Interface），还有我们在本章没有涉及但以前讲过的拨号计划接口（Dialplan Interface）、API 接口（API Interface）、App 接口（Application Interface）等等各种接口。

2.1.14 Core Video

Core Video 实现了 FreeSWITCH 内部支持的视频图像格式的支持。

视频由一帧一帧的图像组成。在 FreeSWITCH 内部按图像进行处理。在 FreeSWITCH 内部，使用原始格式的图像存储。一般来说，原始格式的存储格式有如下几种：

单色图像

每个像素用一个比特表示，0 为黑 1 为白。这样，有多少像素，就需要多少比特。我们以常见的 720p 图像为例子，图像一共有 $1280 * 720 = 921600$ 像素，因此需要占用 $921600 / 8 = 115200$ 个字节。

灰度图像

灰度图像一般使用 256 阶灰度，即用一个字节表示灰度值（0 ~ 255），因此，一帧 720p 的图像会占用 921,600 个字节。

RGB 图像

彩色图像使用 RGB 格式存储，每个像素占 3 个字节（ $3 * 8 = 24$ ，又称 24 位真彩色），因此，一帧图像需要占用 2,764,800 字节。

有的彩色图像有 Alpha 通道，用一个字节代表透明度，0 为透明，255 为不透明，因此一个像素需要占用 4 个字节（正好是一个 32 位整数）。

RGB 图像在内存中，由于计算机体系结构的不同，有不同的内部表示，如：

+--+--+--+--+--+--+--+	
R G B R G B R G B R G B	4 个像素
+--+--+--+--+--+--+--+	
B G R B G R B G R B G R	4 个像素
+--+--+--+--+--+--+--+	
R G B A R G B A	2 个像素，有 Alpha 通道
+--+--+--+--+--+--+--+	
B G R A B G R A	2 个像素，有 Alpha 通道

YUV 色彩空间

通过一定的公式，可以将 RGB 格式的图像换算到 YUV 色彩空间。YUV，是一种颜色编码方法。“Y”表示明亮度（Luminance、Luma），“U”和“V”则是色度、浓度（Chrominance、Chroma）。所以图像可以按如下存储

```
+--+--+--+--+--+--+--+--+
|Y|U|V|Y|U|V|Y|U|V|U|V|
```

YUV 图像与 RGB 图像的主要区别是，如果不管 U 和 V，只看 Y 的话，就是黑白的。所以 YUV 色彩空间广泛用于广播电视中，同样的电视信号，黑白电视和彩电都能播放，只是黑白电视不需要管 U 和 V。

YUV 图像一般来说不是按像素存储，而是按平面（Plane）存储的。如：

Y 平面：

```
+--+--+--+--+--+--+--+--+
|Y|Y|Y|Y|Y|Y|Y|Y|Y|
```

U 平面：

```
+--+--+--+--+--+--+--+--+
|U|U|U|U|U|U|U|U|U|
```

V 平面

```
+--+--+--+--+--+--+--+--+
|V|V|V|V|V|V|V|V|V|
```

按平面存储的好外显而易见，在黑白电视中只需要看 Y 平台而忽略其它平面。

FreeSWITCH 内部主要使用 YUV 格式存储图像。

由于字节序的不同，在内存中的存储方式也有很多种，如：

```
YUVYUVYUVYUV
YVUYVUYVUYVU
YYYYUUUUVVVV
```

其它还有存储格式如 YVU，YUYV，YVYU 等，可以参考<http://www.fourcc.org/yuv.php>上面更多的图像格式。

YUV I420

原始图像占用很大空间，考虑到人类的感知能力，允许降低色度的带宽。常用的节约带宽的格式是 I420 格式。考虑一个 4 像素的正方形图片：

```

+-----+-----+
| Y1 U1 V1 | Y2 U2 V2 |
|-----+-----+
| Y3 U3 V3 | Y4 U4 V4 |
+-----+-----+

```

由于人眼对 U 和 V 不是很敏感，因此，让其它三个像素都使用左上角像素上的 U 和 V，图像变为：

```

+-----+-----+
| Y1 U1 V1 | Y2 U1 V1 |
|-----+-----+
| Y3 U1 V1 | Y4 U1 V1 |
+-----+-----+

```

在这种情况下，如果按平面存储，格式为：

```

Y1 Y2 Y3 Y4
U1 U1 U1 U1
V1 V1 V1 V1

```

去除冗余，图像变为

```

Y1 Y2 Y3 Y4
U1
V1

```

将每 4 个像素作为一组进行这种处理，比原来的存储方式可以节省一半的存储空间。这种存储格式称为 YUV I420。FreeSWITCH 内部主要使用这种方式存储图像。主要的视频编解码器，如 H264 和 VP8，也需要这种格式的图像做编解码。从这种压缩方式也可以看出，按平面存储比按像素存储也更容易计算，因为不同的平面可以有不同的长度。

由于 4 个像素为一组，所以，这种图像一般要求宽和高都为偶数。

YUV I420 的存储格式为: `YYYYYYYY...UU...VV`, 如果 V 在 U 前面, 则称为 YV12, 即 `YYYYYYYY...VV...UU`。

还有两个变种, Y 平面独立, 但 U 和 V 是交错存储的:

- NV12: `YYYYYYYY...UVUV`
- NV21: `YYYYYYYY...VUVU`

值得一提的是, NV12 是 Android 上标准的图像格式。

有时候, 为了能更高效的处理图像 (比如整数计算的数据对齐), 会在存储图像时在每一行结尾预留一些额外的空间, 这些额外的空间就做 `pitch`, 而整个的行宽就叫做 `stride`, 如下图:

按像素存放的 RGB 图像:

```
|<-----stride----->|
|<-----width----->|<-pitch->|
+-----+-----+
|RGB RGB RGB RGB RGB RGB RGB RGB | PPPPPP |
|RGB RGB RGB RGB RGB RGB RGB RGB | PPPPPP |
|RGB RGB RGB RGB RGB RGB RGB RGB | PPPPPP |
```

按平面存放的 RGB 图像:

```
|<-----stride----->|
|<-----width----->|<-pitch->|
+-----+-----+
|YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY | PPPPPP |
|YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY | PPPPPP |
|UUUUUUUUUUUUUU|<-u-pitch->|
|UUUUUUUUUUUUUU|<-u-pitch->|
|VVVVVVVVVVVVVV|<-v-pitch->|
|VVVVVVVVVVVVVV|<-v-pitch->|
```

在 FreeSWITCH 中, 一般来说没有特别的预留 `pitch`。

另外, 还有更高清的图像用两个字节表示一个颜色, 如一个像素可以表示为 `RRGGBB`, FreeSWITCH 暂不支持这种类型。

详细代码可以参阅第4.21节。

2.2 模块

在上一节，我们一起对 FreeSWITCH 核心源代码进行了简单的阅读，了解了 FreeSWITCH 源代码的大致结构和流程。在本节，我们再结合实际模块对源代码进行更深入的分析，来深入了解一下这些代码是如何协同工作的。

在上一节，我们带领大家从 `main` 函数开始看的。但可能还是有的读者觉得比较“绕”——一大堆的代码、一大堆的函数调用，很快就把人绕晕了，看了半天，还是不得要领。其实这也不怪读者，任何项目、任何代码，从不熟悉到熟悉，总要经过一个过程。与代码的作者不同——作者在开发的时候，代码是一行一行写成的，一步一步调试成功的，因此，整个程序的结构全部在他的心里。而对于我们而言，作为一个外来人再去看代码时，就好像只看到一栋盖好的大楼，然后再去想办法了解其结构和建设过程，自然要困难得多。不过，好在，我们在上一章已找到一个突破口——从 `main` 函数开始大致了解了总的框架结构。在此，我们就需要再找一个突破口，一切就比较容易解决了。

2.2.1 mod_dptools

在此，我们找的突破口就是 `mod_dptools`。该模块包含了系统绝大部分的 App，其中就包括我们熟悉的 `answer`（应答）和 `echo`（回声）。从熟悉的地方开始探索，往往比较容易。接下来，看一看我们在源代码里能不能找到 `answer` 和 `echo`。

echo

我们先来找 `echo`。通过使用全文搜索工具搜索源代码，很幸运，我们一下子就在 `mod_dptools.c` 中找到了一个函数定义——`echo_function`，它的代码只有下面短短的三行：

```
2048 SWITCH_STANDARD_APP(echo_function)
2049 {
2050     switch_ivr_session_echo(session, NULL);
2051 }
```

从中，我们可以看出 `echo_function` 这个函数是用 `SWITCH_STANDARD_APP` 这个宏来定义的。接着跟踪这个宏的出外，发现它是在 `switch_types.h:2081` 定义的：

```
2081 #define SWITCH_STANDARD_APP(name) static void name
      (switch_core_session_t *session, const char *data)
```

因此，如果将上面的宏展开，那么 `echo_function` 的定义就是：

```
static void echo_function(switch_core_session_t *session, const char *data){
    switch_ivr_session_echo(session, NULL);
}
```

我们已经知道，每一路通话（一条腿）均有一个 Session（即这里的 `session` 变量），每个 App 都是跟 Session 相关的，因而 FreeSWITCH 在调用每个 App 时，均会把当前的 Session 作为参数传入（一个 `session` 指针）。由于 `echo` App 没有参数，因而这里的 `data` 就是空字符串。当然，如果你在 Dialplan 中传入参数，如：

```
<application action="echo" data="some data"/>
```

那么，这里的 `char *data` 的值就是“`some data`”，只不过，我们在此并不需要用到该参数，因而直接忽略掉了。

该函数就直接调用核心提供的 `switch_ivr_session_echo` 函数，将收到的 RTP 包原样的发回去。而该函数我们在第 20.3.7 节已经详细的看过了。

至此，是不是觉得整个呼叫流程一下子就串起来了？当然，如果还是没有的话，我们继续往下看。

继续在该文件中找 `echo_function`，我们会发现下面一行：

```
5790 SWITCH_ADD_APP(app_interface, "echo", "Echo",
    "Perform an echo test against the calling channel",
    echo_function, "", SAF_NONE);
```

它的作用是将我们刚刚定义的 `echo_function` 加到 `app_interface` 里（即核心的 Application Interface 指针）。

`SWITCH_ADD_APP` 也是一个宏，它是在 `switch_loadable_modules.c:368` 行定义的：

```
368 #define SWITCH_ADD_APP(app_int, int_name, short_descript, \
369     long_descript, funcptr, syntax_string, app_flags) \
370     for (;;) { \
        app_int = (switch_application_interface_t *) \
        switch_loadable_module_create_interface(*module_interface, \
        SWITCH_APPLICATION_INTERFACE); \
371     app_int->interface_name = int_name; \
372     app_int->application_function = funcptr; \
373     app_int->short_desc = short_descript; \
```

```

374     app_int->long_desc = long_descript; \
375     app_int->syntax = syntax_string; \
376     app_int->flags = app_flags; \
377     break; \
378 }

```

这个宏定义的非常巧妙，它使用了一个无限的 for 循环，但由于该循环的最后一条语句是 break，因此它只会执行一次。该循环跟 linux 内核中的“do { ... } while(0)”有异曲同工之妙（参见第 20.3.6 节）。

该宏展开后的结果就相当于（为了易读起见我们去掉了行尾的续行符）：

```

for (;;) {
    app_interface = (switch_application_interface_t *)
        switch_loadable_module_create_interface(
            *module_interface, SWITCH_APPLICATION_INTERFACE);
    app_interface->interface_name = "echo";
    app_interface->application_function = echo_function;
    app_interface->short_desc = "Echo";
    app_interface->long_desc = "Perform an echo test against the calling channel";
    app_interface->syntax = "";
    app_interface->flags = SAF_NONE;
    break;
}

```

所以，一句 SWITCH_ADD_APP 相当于使用 switch_loadable_module_create_interface 函数创建了一个 SWITCH_APPLICATION_INTERFACE 类型的接口（即我们所说的 Application Interface）变量 app_interface，然后给它赋予合适的值。大部分参数都是一些描述信息或帮助字符串，最重要的是下面两行确定了该 echo 这个 app_interface 与我们定义的 echo_function 的对应关系。

```

app_interface->interface_name = "echo";
app_interface->application_function = echo_function;

```

因而，通过 SWITCH_ADD_APP 这个宏，相当于给系统核心添加了一个“echo”App，它对应源代码中的 echo_function。这样，每当系统执行到 Dialplan 中的 echo 程序时，便通过这里的对应关系找到相应的函数入口，进而执行 echo_function 函数。

answer

我们用同样的方法可以找到 answer_function，代码不也不算长，因此，我们也把它全部贴在这里：

```
1210 SWITCH_STANDARD_APP(answer_function)
1211 {
1212     switch_channel_t *channel = switch_core_session_get_channel(session);
1213     const char *arg = (char *) data;
1214
1215     if (zstr(arg)) {
1216         arg = switch_channel_get_variable(channel, "answer_flags");
1217     }
1218
1219     if (!zstr(arg)) {
1220         if (switch_stristr("is_conference", arg)) {
1221             switch_channel_set_flag(channel, CF_CONFERECE);
1222         }
1223     }
1224
1225     switch_channel_answer(channel);
1226 }
```

跟 `echo_function` 类似，该函数也是使用 `SWITCH_STANDARD_APP` 定义的。我们知道——一个 Session 对应一个 Channel。通过 `switch_core_session_get_channel` 函数便可以找当前 Session 对应的 Channel（第 1212 行）。

第 1213 行定义了一个 `arg` 指针，它指向 `answer` 的参数 `data`。如果 `arg`（即传过来的 `data`）为空字符串（第 1215 行，`zstr` 函数用于判断空字符串），则尝试查一下该 Channel 上有没有 `answer_flags` 这个通道变量，如果有的话（第 1219 行。其中 `switch_stristr` 类似于标准的 `strstr`，不区分大小写），就判断该参数中是否包含“`is_conference`”，如果有的话，就把该 Channel 上设置一个 `CF_CONFERECE` 标志（该标志主要用于 RFC4575/RFC4579 描述的会议系统，详见第 TODO 节）。

最后，在第 1225 行调用核心的函数 `switch_channel_answer` 函数来对该 Channel 进行应答。

`switch_channel_answer` 函数实际上是一个宏，在此使用一个宏的作用就是往函数中传入调用者的源文件名和行号信息，以便在日志中打印的文件名和行号是实际调用该函数处的文件名和行号，而不是该函数实际定义处的行号（否则没有什么实际意义）。该宏展开后便是实际调用 `switch_channel.c:3693` 中的 `switch_channel_perform_answer` 函数。

在该函数中，它会首先在第 3695 行初始化一个 `msg` 变量，该变量是 `switch_core_session_message_t` 类型的，用于定义一条消息。然后，第 3714 ~ 3715 行初始化消息的内容，并于第 3716 行将消息发送出去（消息（Message）是与 Core Event 类似的另外一种消息传递（调用）方式，与 Core Event 不同的是，消息的发送总是同步进行的，因此，这里的 `perform_receive_message` 实际上是直接调用各模块中接收消息的回调函数，我们在第 21.1.3 节还会讲到）。

```
3693 SWITCH_DECLARE(switch_status_t) switch_channel_perform_answer(
    switch_channel_t *channel, const char *file, const char *func, int line)
```

```
3694 {
3695     switch_core_session_message_t msg = { 0 };
...
3714     msg.message_id = SWITCH_MESSAGE_INDICATE_ANSWER;
3715     msg.from = channel->name;
3716     status = switch_core_session_perform_receive_message(channel->session, &msg, file, func, line);
```

如果消息发送成功，就在第 3720 行将该 Channel 的状态置为已经应答的状态。

```
3719     if (status == SWITCH_STATUS_SUCCESS) {
3720         switch_channel_perform_mark_answered(channel, file, func, line);
```

实际上，如果这里的 Channel 是一个 SIP 通话的话，FreeSWITCH 中的 `mod_sofia` Endpoint 模块便会调用底层的 Sofia-SIP 协议栈（`libsofia`）给对方发送“200 OK” SIP 消息。

`answer_function` 也是由 `SWITCH_ADD_APP` 宏安装到核心中去。

set

FreeSWITCH 中大量使用通道变量控制通话（Channel）的行为。设置通道变量的操作是由下面的 `set` App 实现的。该函数出奇的简单，是因为它直接调用了另外一个函数 `base_set`。

```
1439 SWITCH_STANDARD_APP(set_function)
1440 {
1441     base_set(session, data, SWITCH_STACK_BOTTOM);
1442 }
```

`base_set` 其它会被多个函数调用，在此，我们只关心它被 `set_function` 调用的情况。为了列直观，我们在实际的例子进行说明。假设我们在 Dialplan 中使用如下配置：

```
<action application="set" data="dialed_extension=$1"/>
```

其中 `$1` 为前面的正则表达式的匹配结果，它是一个变量，我们假设它的值为 `1001`。那么，在下面的函数中，传入的 `data` 参数的值就是一个字符串：“`dialed_extension=$1`”。

```
1375 static void base_set (switch_core_session_t *session,
                        const char *data, switch_stack_t stack)
```

```
1376 {  
1377     char *var, *val = NULL;
```

在第 1385 行, 会将该字符串使用 `switch_core_session_strdup` 复制一份。该函数是在 `session` 上进行操作的, 它会使用该 `session` 的内存池申请字符串空间, 因而申请以后的内存无需释放。至于为什么要重新复制一份, 那是因为我们接下来的操作会修改该字符串的内存 (如第 1392 行), 因而, 复制一份可以避免破坏原来的字符串。到此, 我们的 `var` 变量的值就是 “`dialed_extension=$1`”。

```
1385         var = switch_core_session_strdup(session, data);
```

接下来, 第 1387 行判断字符串是否包含等号, 在我们的例子里有等号, 因此 `val` 指向等号所在的内存位置, 也可以说, `var` 指针所指的字符串值为 “`= $1`”。

```
1387         if (!(val = strchr(var, '='))) {  
1388             val = strchr(var, ',');  
1389         }
```

如果 `val` 非空 (第 1391 行), 则第 1392 行将 `val` 所指的位置写入 “`\0`” (即 C 语言中的字符串结束符), 并将 `val` 指针向后移动一个字节, 此时它的值就是 “`$1`” 了。同时, 由于我们将原字符串中的等号替换改成了 “`\0`”, 因此, `var` 所指向的字符串的值也相当于变短了, 此时, `var` 的值为 “`dialed_extension`”。

```
1391         if (val) {  
1392             *val++ = '\0';  
1393             if (zstr(val)) {  
1394                 val = NULL;  
1395             }  
1396         }
```

第 1398 行, 继续判断如果 `val` 为非空 (因为已经移动了指针, 所以要重新判断), 则执行第 1399 行的函数, 就 `val` 指针中的 “`$1`” 变量替换为它的实际的值。在这里, 我们将会在 `expanded` 变量中得到实际的值 “`1001`”。

```
1398         if (val) {  
1399             expanded = switch_channel_expand_variables(channel, val);  
1400         }
```

第 1404 行将调用函数在 Channel 上设置我们指定的新变量：

```
1404      switch_channel_add_variable_var_check(channel,
          var, expanded, SWITCH_FALSE, stack);
```

它等价于：

```
switch_channel_add_variable_var_check(channel,
    "dialed_extension", "1001", SWITCH_FALSE, stack);
```

当然，最后不要忘记，`expanded` 指针所指向的内存是动态申请的，因此，一定要释放内存，以避免引起内存泄漏。

```
1406      if (expanded && expanded != val) {
1407          switch_safe_free(expanded);
1408      }
```

总之，虽然我们这里讲得比较啰嗦，但实际的过程还是非常简单的。不过，既然我们啰嗦到了这里，就索性啰嗦个够，我们接着看一看第 1404 行调用的 `switch_channel_add_variable_var_check` 函数到底都干了些什么。

该函数定义于“`switch_channel.c:1400`”。它在第 1407 行先对临界区加锁，以防止其它并发的线程同时修改。然后，经过一系列的判断和检查，如果最终所有检查的都通过的话（第 1417 行），则在第 1418 行调用 `switch_event_add_header_string` 函数将通道变量添加到 `channel->variables` 中去。该函数我们在第 18.3.1 节讲过的 `esl_event_add_header_string` 类似，它实际上是往一个 `switch_event_t` 类型的结构体上添加数据，所以，这里可以看到，`channel->variables` 在内部是使用 `switch_event_t` 来存储的。这也不奇怪，因为通道变量本来就是一对“键/值”对（`varname` 和 `value`）。

```
1400 SWITCH_DECLARE(switch_status_t) switch_channel_add_variable_var_check(
    switch_channel_t *channel,
1401     const char *varname, const char *value,
    switch_bool_t var_check, switch_stack_t stack)
1402 {
    ...
1407     switch_mutex_lock(channel->profile_mutex);
    ...
```

```
1417         if (ok) {  
1418             switch_event_add_header_string(channel->variables,  
                                             stack, varname, value);
```

当然，永远不要忘了释放锁：

```
1425     switch_mutex_unlock(channel->profile_mutex);
```

至此，`set` 函数就全部剖析完了。通过它设置的通话变量，以后也可以通过 `switch_channel_get_variable` 再取出来。当然，这就是另外的事情了。

bridge

接下来我们再来看一下 `bridge` 这个 App，从某种意义上讲，它属于 FreeSWITCH 的核心功能，也比较有代表性。

`bridge` App 是由第 3017 行的 `audio_bridge_function` 函数完成的。该函数比较复杂，我们尽量挑简单的部分说。

首先，该 App 在 Dialplan 中的使用方法一般是：

```
<action application="bridge" data="user/1001"/>
```

因而，该函数中的 `data` 参数便是一个指向字符串“`user/1001`”的指针。在第 3052 行，首先检查该字符串的有效性。如果它为空字符串，那就没有必须继续进行了，直接返回（`return`，第 3053 行）。

```
3037 SWITCH_STANDARD_APP(audio_bridge_function)  
3038 {  
...  
3052     if (zstr(data)) {  
3053         return;  
3054     }
```

我们跳过很多“`if/else`”假设，直接跳到第 3194 行（一般来说都会执行到这里）。接下来的第 3195 行将调用核心的 `switch_ivr_orinate` 函数发起一个新的呼叫。

```

3194         if ((status =
3195             switch_ivr_originate(session, &peer_session, &cause,
                                   data, 0, NULL, NULL, NULL, NULL, SOF_NONE, NULL))
              != SWITCH_STATUS_SUCCESS) {
3196             fail = 1;

```

`switch_ivr_originate` 函数是在 `switch_ivr_originate.c: 1850` 定义的。该函数中能是 FreeSWITCH 最长的一个函数，在我们参考的版本中，它足足有 2048 行！因此，笔者在此不准备研究它⁸。但我们下面来看一下它使用的这几个参数的意义。

其中，`session` 就是指当前的 Session，即呼入的那条腿（a-leg），我们执行到此外，调用该函数创建另一条腿（b-leg）。因而，第二个参数 `peer_session` 就将是新建立的 Session。由于我们在该函数执行完成后，需要知道 `peer_session` 指针的值，因此这里我们传入的是指针变量的地址（相当于一个双重指针）。同理，我们也需要在呼叫失败时得到呼叫原因（`cause`），因此把它作为第三个变量。第四个参数便是我们提供的呼叫字符串（`data`）的指针，在本例中该字符串的值是“`user/1001`”。

其它的参数我们就没必要看了，大部分都是空指针。在此，由于我们传入了的当前的 `session` 指针，因此该函数在执行的时候就有参照物了——如，它会将 a-leg（当前 `session`）中的主叫号码（`effective_caller_id_number`）作为主叫号码去呼叫 b-leg 等。当然，b-leg 也不白参数 a-leg，如果 b-leg 的对端回了呼叫进展消息（如 SIP 180 或 183 消息），则 a-leg 也能听到相关的提示音。

如果 b-leg 的对方应答，或者在呼叫进展中返回了媒体消息（如 SIP 中的 183 消息），则上述的 `switch_ivr_originate` 函数就会返回。在接下来的第 3207 行，我们将得到新的 Channel（b-leg 对应的 Channel——`peer_channel`）。

```

3207 switch_channel_t *peer_channel = switch_core_session_get_channel(peer_session);

```

如果我们在呼叫时使用的是 Proxy Media 模式的话（3123 行），则执行 3214 行的函数仅进行信令级的桥接，否则的话（正常情况），就执行第 3236 行的多线程的桥接函数 `switch_ivr_multi_threaded_bridge`。

```

3213         if (switch_channel_test_flag(caller_channel, CF_PROXY_MODE)) {
3214             switch_ivr_signal_bridge(session, peer_session);
3215         } else {
...

```

⁸或许只是这一个函数也够写一本书了。好多读者也是在阅读源代码时，好像是在学会了 `originate` 命令之后，抑或是在知道 `bridge` App 调用了该函数之后，就来看这个函数。然后，得出结论：要不就是一个函数写这么长，代码写得太烂；要不就是一下就被吓住了，看不懂。因此，不建议初学者研究这个函数。笔者也是大致看过，并没有深入研究。

```

3236         switch_ivr_multi_threaded_bridge(session, peer_session,
3237         func, a_key, b_key);
    }

```

接下来我们看 `switch_ivr_multi_threaded_bridge` 函数。它是在 `switch_ivr.c:1270` 实现的。它首先在第 1275 和 1276 行初始化了两个 `switch_ivr_bridge_data_t` 类型的变量 `a_leg` 和 `b_leg`，用于存放两条腿相关的私有数据（我们后面会用到他们）。

```

1270 SWITCH_DECLARE(switch_status_t) switch_ivr_multi_threaded_bridge(
    switch_core_session_t *session,
1271    switch_core_session_t *peer_session,
1272    switch_input_callback_function_t input_callback, void *session_data,
1273    void *peer_session_data)
1274 {
1275     switch_ivr_bridge_data_t *a_leg =
        switch_core_session_alloc(session, sizeof(*a_leg));
1276     switch_ivr_bridge_data_t *b_leg =
        switch_core_session_alloc(peer_session, sizeof(*b_leg));

```

在第 1319 行，它首先在 `peer_channel`（即 b-leg）上安装一些状态回调函数，当 b-leg 的状态发生变化时，将调用相关的回调函数。

```

1319     switch_channel_add_state_handler(peer_channel, &audio_bridge_peer_state_handlers);

```

然后产生一个 `CHANNEL_BRIDGE` 事件（第 1342 行），并发送出去（第 1347 行）。

```

1342     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_BRIDGE) == SWITCH_STATUS_SUCCESS) {
    ...
1347         switch_event_fire(&event);

```

接下来，分别在 a-leg 和 b-leg 上产生一个 `SWITCH_MESSAGE_INDICATE_BRIDGE` 消息（Message，用于标志该 Channel 已经被桥接了），发送给它们（第 1400、1408 行）。

```

1396         msg.message_id = SWITCH_MESSAGE_INDICATE_BRIDGE;
    ...
1400         if (switch_core_session_receive_message(peer_session, &msg) != SWITCH_STATUS_SUCCESS) {
    ...
1408         if (switch_core_session_receive_message(session, &msg) != SWITCH_STATUS_SUCCESS) {

```

在第 1439 行，将一个 `b_leg` 数据指针确定的私有的数据绑定到 b-leg 上（使用私有数据与设置通道变量类似，但后者只能是字符串值，而前者可以绑定做任意值）。然后，在第 1440 行将 b-leg 的状态设为媒体交换的状态（`CS_EXCHANGE_MEDIA`）

```
1439      switch_channel_set_private(peer_channel, "_bridge_", b_leg);
1440      switch_channel_set_state(peer_channel, CS_EXCHANGE_MEDIA);
```

这时候，b-leg 的状态发生了变化，因而会回调在上面 1319 行设置过的回调函数。不过，在讲这些回调函数前，我们先把第 1442 行讲完。接下来的 1442 行很简单，它执行 `audio_bridge_thread` 函数，并将一个 `a_leg` 数据指针传入。该数据指针包含 a-leg 的一些信息。

```
1442      audio_bridge_thread(NULL, (void *) a_leg);
```

第 1442 行的执行是阻塞的，它将阻塞的执行一直到 `bridge` 结束，因此，我们可以倒回头来看 b-leg 上的回调函数了。

我们在第 1319 行就在 b-leg 上安装了一些回调函数，这里回调函数是在一个全局变量中指定的，如下：

```
771 static const switch_state_handler_table_t audio_bridge_peer_state_handlers = {
772     /*.on_init */ NULL,
773     /*.on_routing */ audio_bridge_on_routing,
774     /*.on_execute */ NULL,
775     /*.on_hangup */ NULL,
776     /*.on_exchange_media */ audio_bridge_on_exchange_media,
777     /*.on_soft_execute */ NULL,
778     /*.on_consume_media */ audio_bridge_on_consume_media,
779 };
```

其中，我们在第 1440 行将 Channel 的状态设置为 `CS_EXCHANGE_MEDIA`，Channel 的状态发生了改变，它就会回调在第 776 行指定的 `audio_bridge_on_exchange_media` 函数。

在 `audio_bridge_on_exchange_media` 函数（第 694 行）中，可以看到，它在第 697 行通过 `switch_channel_get_private` 取出了该 Channel 上的一个私有的数据结构，而该私有数据即为我们在上述的第 1439 行设置的。该私有数据里面存储了与 `bridge` 相关的 b-leg 上的数据，有了它以后，我们就可以在第 704 行执行 `audio_bridge_thread` 函数了。注意，在此，第 704 行传入的参数是 b-leg 上的 `switch_ivr_bridge_data_t` 结构的私有数据。

```

694 static switch_status_t audio_bridge_on_exchange_media(switch_core_session_t *session)
695 {
696     switch_channel_t *channel = switch_core_session_get_channel(session);
697     switch_ivr_bridge_data_t *bd =
        switch_channel_get_private(channel, "_bridge_");
    ...
703     if (bd->session == session && *bd->b_uuid) {
704         audio_bridge_thread(NULL, (void *) bd);

```

至此，我们可以看到，a-leg 和 b-leg 分别在他们自己的线程中执行 `audio_bridge_thread` 函数了（这个很重要，我们的思维现在并行化了，即下面我们讲的所有代码都是在两条腿上在两个线程中并行执行的），并且，在该函数中，他们分别传入了自己所在的那条腿上的 `switch_ivr_bridge_data_t` 结构的数据。

在该函数中，它首先在第 207 行将传入的数据从 `obj` 指针赋值给一个 `data` 指针，并在第 236 行将 `data` 指针中的 `session` 成员变量赋值给 `session_a`。注意，到了这里，`session_a` 就不一定是 a-leg 了，而是只在当前线程中的那条腿。即，如果在 a-leg 中调用该函数，它就是 a-leg，如果在 b-leg 中调用该函数，它就是 b-leg。同理，第 237 行的 `session_b` 变量也不一定是 b-leg，而是与本条腿相对的那条腿（桥接中的另一条腿）。注意，该腿相关的 `session_b` 变量不是直接传入的指针，而是传入了一个 Channel 的 UUID（`data->b_uuid`），因此，我们需要使用 `switch_core_session_locate` 来取得该 UUID 对应的 Session 的指针 `session_b`。

```

205 static void *audio_bridge_thread(switch_thread_t *thread, void *obj)
206 {
207     switch_ivr_bridge_data_t *data = obj;
    ...
236     session_a = data->session;
237     if (!(session_b = switch_core_session_locate(data->b_uuid))) {
238         return NULL;
239     }

```

至此，在两个线程中，分别都有当前的 Session 和另一个 Session 的信息了，第 256 ~ 257 行即分别取出它们对应的 Channel。然后，就是一个无限循环（第 341 行），在该循环中，不停地在当前的 Session（`session_a`）中读取一帧媒体数据（第 547 行），然后写入另一个 Session（`session_b`，第 565 行）。这就实现了媒体⁹的交换，也是 `bridge` App 的全部秘密。当然，第 546 行的注释可能更简洁直观一些——“从一个 Channel 中读取音频并写入另一个 Channel”。

```

245     chan_a = switch_core_session_get_channel(session_a);
246     chan_b = switch_core_session_get_channel(session_b);

```

⁹注意，这里所说的媒体就是音频，为了简单起见，我们没有分析视频有关的代码。

```

...
341     for (;;) {
...
546         /* read audio from 1 channel and write it to the other */
547         status = switch_core_session_read_frame(session_a, &read_frame, SWITCH_IO_FLAG_NONE,
↪      stream_id);
...
565         if (switch_core_session_write_frame(session_b, read_frame, SWITCH_IO_FLAG_NONE,
↪      stream_id) != SWITCH_STATUS_SUCCESS) {

```

当读取数据错误、或者检测到挂机时，上述无限循环将终止。在上述函数中，我们在第 237 行使用了 `switch_core_session_locate` 通过一个 UUID 获得了 `session_b` 的指针。而该函数在返回指针的同时会将当前的 Session 加锁，以防止产生竞争条件（Race Condition）。因此，在任何时候使用 `switch_core_session_locate` 函数并获得了非空的指针时，在指针使用完成后都需要明确的解锁，如第 673 行所示：

```

673     switch_core_session_rwlock(session_b);

```

当然，当上面的 `audio_bridge_thread` 函数完成后，后续还有很多事情要做，如发送 `CHANNEL_UNBRIDGE` 事件、检查所有相关的 `after_bridge`（桥接后的）变量（我们常用的 `hangup_after_bridge` 变量就是在这里检查的）等，篇幅所限，我们就不多讲了。

Endpoint Interface

在 `mod_dptools` 模块中，实现了一些常用的“假”的 Endpoint Interface。之所以说是“假”的，是因为它们并没有像 `mod_sofia` 那样即有底层的协议驱动、又有媒体收、发处理，而是为了简化某些操作，或者为了在某些特殊的情况下使用一些一致的命令或接口而实现的。如，我们常用的 `user` 就是一个 Endpoint。一般来说，一个 Endpoint 都会提供一个呼叫字符串、用于外呼，我们对于 `user` 提供的呼叫字符串已经非常熟悉了，如在命令行和 Dialplan 中我们经常使用如下的呼叫字符串：

```

originate user/1000 &echo
<action application="bridge" data="user/1000" />

```

这里的 `user` 就是由 `user` Endpoint 实现的。该 Interface 的指针是在第 3879 行声明的一个全局变量。

```

3879 switch_endpoint_interface_t *user_endpoint_interface;

```

在第 3885 ~ 3887 行，定义了一个 `switch_io_routines_t` 类型的结构体，用于定义回调函数。可以看出，由于该 Endpoint 很简单，它只定义了一个 `outgoing_channel` 回调函数。该回调函数将在有人使用 `user` 呼叫字符串时（如执行 `originate` 和 `bridge` 时）被调用。

```
3885 switch_io_routines_t user_io_routines = {
3886     /*.outgoing_channel */ user_outgoing_channel
3887 };
```

该回调函数的定义如下：

```
3889 static switch_call_cause_t user_outgoing_channel(switch_core_session_t *session,
3890     switch_event_t *var_event,
3891     switch_caller_profile_t *outbound_profile,
3892     switch_core_session_t **new_session,
3893     switch_memory_pool_t **pool, switch_originate_flag_t flags,
3894     switch_call_cause_t *cancel_cause)
3895 {
```

首先，该函数的输入参数中将包含一个 `outbound_profile`，它的成员变量 `destination_number` 即时被叫号码。在第 3909 行，复制该被叫号码并赋值给 `user` 指针。

```
3909     user = strdup(outbound_profile->destination_number);
```

然后获取 `domain` 的值，如果呼叫字符串中未包含 `domain`（如 `user/1000@192.168.1.2` 就包含了 `domain`，而 `user/1000` 则未包含），则在第 3917 行尝试获取默认的 `domain`。

```
3914     if ((domain = strchr(user, '@'))) {
3915         *domain++ = '\0';
3916     } else {
3917         domain = switch_core_get_domain(SWITCH_TRUE);
```

接下来，第 3942 行，从 XML 用户目录中查找该用户，并继续在第 3593 行尝试找到 `dial-string` 配置参数

```
3942     if (switch_xml_locate_user_merged("id", user, domain, NULL,
3943         &x_user, params) != SWITCH_STATUS_SUCCESS) {
```

```
3953         if (!strcasecmp(pvar, "dial-string")) {
```

如果该呼叫字符串是在 `bridge` 中使用的，则第 3992 行的判断成立（即说明有 a-leg），否则（第 4004 行）说明是个腿的呼叫（`originate`）。然后根据不同的情况会有相关的设置，并都会得到一个 `d_dest`（第 4002 或 4022 行）的地址（如 `sip:1000@192.168.1.100:7890` 等）。

```
3992         if (session) {
...
4002             d_dest = switch_channel_expand_variables(channel, dest);
4003
4004         } else {
...
4022             d_dest = switch_event_expand_headers(event, dest);
...
4024         }
```

随后，就调用 `switch_ivr_originate` 去呼叫该地址了，如第 4040 行所示：

```
4040         } else if (switch_ivr_originate(session,
                                         new_session, &cause, d_dest, ...
```

`user` Endpoint 基本上是最简单的一个 Endpoint。它目前仅支持 SIP 呼叫（理论上它还可以扩展支持其它的），实际的呼叫流程还是要转到实际的 `mod_sofia` Endpoint 上进行处理。我们将在第 2.2.1 节再讲 `mod_soifa` 是如何实现 Endpoint Interface 的。

模块框架

在 FreeSWITCH 中，一个模块主要是由 `load`、`runtime` 和 `shutdown` 回调函数组成的。`mod_dptools` 当然也不例外。

该模块的 `load` 函数在第 5578 行定义，它将在模块被加载的时候执行。从第 5580 ~ 第 5584 行可以看出，它实现了包括 API Interface、App Interface、Dialplan Interface 在内的多个 Interface。

```
5578 SWITCH_MODULE_LOAD_FUNCTION(mod_dptools_load)
5579 {
5580     switch_api_interface_t *api_interface;
5581     switch_application_interface_t *app_interface;
```

```
5582     switch_dialplan_interface_t *dp_interface;  
5583     switch_chat_interface_t *chat_interface;  
5584     switch_file_interface_t *file_interface;
```

在初始化了一系列的内存池及其它数据结构后，它在 5593 行向核心注册该模块。

```
5593     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
```

在第 5595 行，它向核心绑定（订阅）了一个 `SWITCH_EVENT_PRESENCE_PROBE` 事件的回调函数，即每当系统中产生该事件后，都会执行回调函数 `pickup_pres_event_handler`。

```
5595     switch_event_bind(modname, SWITCH_EVENT_PRESENCE_PROBE,  
                       SWITCH_EVENT_SUBCLASS_ANY, pickup_pres_event_handler, NULL);
```

另外，它还实现了一些 Endpoint Interface：

```
5621     error_endpoint_interface = ...  
5625     group_endpoint_interface = ...  
5629     user_endpoint_interface = ...  
5633     pickup_endpoint_interface = ...
```

向核心注册 API 和 App：

```
5641     SWITCH_ADD_API(api_interface, "strepoch", ...  
5645     SWITCH_ADD_API(api_interface, "strftime", ...  
5790     SWITCH_ADD_APP(app_interface, "echo", ...  
5791     SWITCH_ADD_APP(app_interface, "park", ...  
5794     SWITCH_ADD_APP(app_interface, "playback", ...
```

还可以看到，`inline` Dialplan 也是在该模块中实现的：

```
5839     SWITCH_ADD_DIALPLAN(dp_interface, "inline", inline_dialplan_hunt);
```

总之，在该函数的最后，返回 `SWITCH_STATUS_SUCCESS` 表明该模块加载成功：

```
5842     return SWITCH_STATUS_SUCCESS;
```

该模块没有 `runtime` 函数。其 `shutdown` 函数也很简单，该模块没有太多要清理的资源，它只需要在第 5573 行向核心取消先前在第 5593 行绑定的事件回调函数：

```
5571 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_dptools_shutdown)
5572 {
5573     switch_event_unbind_callback(pickup_pres_event_handler);
5574
5575     return SWITCH_STATUS_SUCCESS;
5576 }
```

2.2.2 mod_commands

在 `mod_commands` 中，实现了大部分的 API 命令。如常用的 `version`、`status`、`originate` 等。下面，我们先从该模块的 `load` 函数看起。

该函数在第 6388 行定义。它也是定义了一个 `switch_api_interface_t` 类型的指针（第 6390 行）用于实现 API Interface，于 6393 行向核心注册本模块，于第 6420、6444、6460 行等向核心注册它实现的命令的回调函数等。

```
6388 SWITCH_MODULE_LOAD_FUNCTION(mod_commands_load)
6389 {
6390     switch_api_interface_t *commands_api_interface;
6391     ...
6393     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
6394
6395     ...
6420     SWITCH_ADD_API(commands_api_interface, "version", ...
6444     SWITCH_ADD_API(commands_api_interface, "originate", ...
6460     SWITCH_ADD_API(commands_api_interface, "status", ...
```

然后，它使用 `switch_console_set_complete` 添加命令补全信息（如第 6603、6604 行），以便用户在控制台上输入命令时可以使用 Tab 键进行补全。

```
6603     switch_console_set_complete("add show calls");
6604     switch_console_set_complete("add show channels");
```

最后，该函数返回 `SWITCH_STATUS_NOUNLOAD`。与其它模块返回 `SWITCH_STATUS_SUCCESS` 不同，这里的返回值表示该模块是无法被卸载的（由于 `unload` 命令本身是在该模块实现的）。

```
6690     return SWITCH_STATUS_NOUNLOAD;
6691 }
```

我们以 `originate` 命令为例来讲一下。`originate` 命令是在 `originate_function` 中实现的。它在第 4067 行使用 `SWITCH_STANDARD_API` 进行声名。该声名也是一个在 `switch_types.h:2016` 行定义的一个宏，该宏定义在在笔者的电脑上展开的结果如下：

```
static switch_status_t originate_function ( const char *cmd,
                                           switch_core_session_t *session, switch_stream_handle_t *stream)
```

从上面的展开结果可以看出，该函数有三个输入参数。第一个是输入的命令参数；第二个是一个 `session`，但由于大多数的 API 命令都跟 Session 无关，因此该参数一般是一个空指针；第三个参数是一个 `stream`，它是一个流，写入该流中的数据（命令输出）将可以作为命令的结果返回。

第 4089 行，首先，复制了命令字符串。由于 `originate` 命令的参数众多，因此，它使用一个 `switch_separate_string` 将命令字符串进行分隔。该函数将分割后的结果放到一个 `argv` 数组中，并返回数组中参数的个数 `argc`（在这一点上，类似于 C 语言中经典的 `main` 函数的参数）。

```
4067 SWITCH_STANDARD_API(originate_function)
4068 {
...
4089     mycmd = strdup(cmd);
4090     switch_assert(mycmd);
4091     argc = switch_separate_string(mycmd, ' ', argv, (sizeof(argv) / sizeof(argv[0])));
```

在对输入参数进行分析后，它便在第 4128 行调用 `switch_ivr_originate` 发起一个呼叫。读者可以看到，`bridge` App 也是调用了该函数发起呼叫，但不同的是，在这里，它的第一个参数是一个空指针（`NULL`），因而这是一个单腿的呼叫。

```
4128     if (switch_ivr_originate(NULL, &caller_session, &cause, aleg,
                             timeout, NULL, cid_name, cid_num, ...
```

另外一个与 `bridge` App 中调用方法不同的地方在于，在这里，它的大部分参数都不是空指针，因而可以在外呼的同时指定其它参数，如超时（`timeout`）、主叫名称（`cid_name`）、主叫号码（`cid_num`）等。

如果我们在发起呼叫的时候使用“&”指定了一个 App，如 `originate user/1000 &echo`，则它在对方接听后（严格来说是收到媒体后，如收到 SIP 183 消息后），即开始执行第 4153 行，执行 `app_name`（如 `echo`）所指定的函数。

```
4153         if ((extension = switch_caller_extension_new(caller_session, app_name, arg)) == 0) {
```

否则的话，就在第 4160 行转移到相应的 Dialplan，如用户输入“‘originate user/1000 9196 XML default’”的情况。

```
4160     } else {
4161         switch_ivr_session_transfer(caller_session, exten, dp,
```

在第 4165 行，调用输出流 `stream` 的 `write_function` 输出命令的反馈信息，如“+OK UUID”。

```
4165         stream->write_function(stream, "+OK %s\n",
```

使用 `switch_ivr_originate` 所产生的 Session 也是加锁的，因而，我们也要明确的释放它：

```
4170         switch_core_session_rwlock(caller_session);
```

2.2.3 mod_sofia

`mod_sofia` 是 FreeSWITCH 中最大的一个模块，也是最重要的一个模块。所有的 SIP 通话都是它开始和终止的，因而，分析一下该模块的源代码是很有参考意义的。

该模块非常庞大而且复杂，它实现了 SIP 注册、呼叫、Presence、SLA 等一系列的 SIP 特性。在此，我们抓住一条主线，仅研究 SIP 呼叫有关的代码，以避免又陷入庞大代码的海洋。

模块加载

我们还是从该模块的 `load` 函数做为入口。它是在 `mod_sofia.c:5420` 实现的。该函数最开始在第 5423 行定义了一个 `api_interface` 指针，用于往核心中添加 API。第 5427 行，它将一个全局变量 `mod_sofia_globals` 清零。该全局变量在整个模块内是有效的，它用于记录一些模块级的数据和变量。然后，在进行一定的初始化后，它在第 5447 行将全局变量的一个 `running` 成员变量设为 1，标志该模块是在运行的。

```
5420 SWITCH_MODULE_LOAD_FUNCTION(mod_sofia_load)
5421 {
5423     switch_api_interface_t *api_interface;
...
5428     memset(&mod_sofia_globals, 0, sizeof(mod_sofia_globals));
...
5446     switch_mutex_lock(mod_sofia_globals.mutex);
5447     mod_sofia_globals.running = 1;
5448     switch_mutex_unlock(mod_sofia_globals.mutex);
```

在第 5468 行，将启动一个消息处理线程，用于 SIP 消息的处理。

```
5468     sofia_msg_thread_start(0);
```

第 5475 行调用 `config_sofia` 函数来从 XML 中读取该模块的配置并启动相关的 Sofia Profile。第 5549 行，向核心注册本模块。第 5550 行，初始化一个新的 Endpoint，然后接着指定该新的 Endpoint 的名字及绑定相关的回调函数（第 5551 ~ 5554 行）。

```
5549     *module_interface =
        switch_loadable_module_create_module_interface(pool, modname);
5550     sofia_endpoint_interface =
        switch_loadable_module_create_interface(*module_interface,
        SWITCH_ENDPOINT_INTERFACE);
5551     sofia_endpoint_interface->interface_name = "sofia";
5552     sofia_endpoint_interface->io_routines = &sofia_io_routines;
5553     sofia_endpoint_interface->state_handler = &sofia_event_handlers;
5554     sofia_endpoint_interface->recover_callback = sofia_recover_callback;
```

后面的代码还有很多，我们就不继续往下看了。至此，我们还有两个细节还没有研究明白。第一，就是上面刚刚讲到的这些回调函数都是怎么使用的，第二，就是底层的 Sofia 库是在哪儿启动的，又是如何接收 SIP 消息并建立通话的。为了从根本了解一路通话的建立过程，这次，我们先从第二个问题开始看。

Sofia 的加载及通话建立

接下来，我们来看一下 Sofia（即我们的 SIP 服务）到底是从哪里加载的，通话的建立是从哪儿开始，又是如何进行的。

Sofia 的加载 关于 Sofia 的加载，其它我们刚刚已经讲过了，它就隐藏在第 5475 行的 `config_sofia` 函数中。该函数是在 `sofia.c:3585` 定义的。该函数非常长，它解析 XML 配置文件，初始化 Profile 相关的变量的数据结构，并启动相关的 Profile。我们熟知的默认的 `internal` Profile 就是在第 4940 行启动的。

```
3585 switch_status_t config_sofia(sofia_config_t reload, char *profile_name)
3586 {
3587     char *cf = "sofia.conf";
...
4940         launch_sofia_profile_thread(profile);
```

`launch_sofia_profile_thread` 在第 2817 行定义，它将于 2826 行启动一个新线程，并在新线程中执行 `sofia_profile_thread_run`，同时将 `profile` 作为输入参数。

```
2817 void launch_sofia_profile_thread(sofia_profile_t *profile)
2818 {
...
2826     switch_thread_create(&profile->thread, thd_attr,
        sofia_profile_thread_run, profile, profile->pool);
2827 }
```

在新线程中（第 2430 行），将在第 2432 行得到 `profile` 指针的值。然后会在第 2481 行调用 `nua_create` 函数建立一个 UA（User Agent）。该函数是 Sofia-SIP 库提供的函数，它将启动一个 UA，监听相关的端口（如大家熟知的 5060），并等待 SIP 消息到来。一旦收到 SIP 请求，它便会回调 `sofia_event_callback` 回调函数（第 2482 行），该回调函数中将带着对应的 `profile` 作为回调参数。

```
2430 void *SWITCH_THREAD_FUNC sofia_profile_thread_run(
        switch_thread_t *thread, void *obj)
2431 {
2432     sofia_profile_t *profile = (sofia_profile_t *) obj;
...
2481     profile->nua = nua_create(profile->s_root, /* Event loop */
```

```

2482     sofia_event_callback, /* Callback for processing events */
2483     profile, /* Additional data to pass to callback */
2484     ...

```

关于 Sofia-SIP 底层的库我们就不深入研究了。到此为止，我们的 SIP 服务已经启动了，就等着接收 SIP 消息。

SIP 消息的接收 当我们的服务收到 SIP 消息后，便会调用 `sofia_event_callback` 回调函数。该函数是在第 1789 行。在该行，如果得到回调时，将收到一个 `nua_event_t` 结果的 SIP 事件 `event`。即使不看 Sofia-SIP 库的文档，我们也能从第 1800 行的 `switch` 语句以及后面的 `case` 分支中可以看出——该事件到底是对应什么类型的 SIP 消息了。如果收到 SIP `INVITE` 消息，那么它一定会匹配到第 1840 行。

```

1789 void sofia_event_callback(nua_event_t event,
1790     int status,
1791     char const *phrase,
1792     nua_t *nua, sofia_profile_t *profile,
1793     nua_handle_t *nh, sofia_private_t *sofia_private,
1794     sip_t const *sip,
1795     tagi_t tags[])
1796 {
1797     ...
1800     switch(event) {
1801     case nua_i_terminated:
1802         ...
1840     case nua_i_invite:
1841     case nua_i_register:
1842     case nua_i_options:
1843     case nua_i_notify:
1844     case nua_i_info:

```

不同的消息将进行不同的处理，但大部分都会执行到第 2018 行，将消息通过一个核心的消息队列分发出去（其中，`de` 是一个 `sofia_dispatch_event_t` 的结构体指针，它包含了本次收到的 SIP 消息）。

```

2018     sofia_queue_message(de);

```

Sofia-SIP 库在底层是一个单线程的结构，因此在这里我们使用了消息对列以提高并发量。

接下来我们跟踪到第 1749 行，`sofia_queue_message` 函数将保证我们的消息队列有足够的处理能力。然后，进行必要的检查。如果这是第一次 `INVITE` 请求（第 1881 行），则在第 1943 行（略）或第 1945 行调用 `switch_core_session_request` 生成一个新的 Session，并赋值给 `session` 指针。到这里，`INVITE` 请求在我们系统中起作用了——它导致我们的系统中创建了一个 Session，在以后所有与该 `INVITE` 消息相关的会话消息中，都会与该 Session 相关（当然，具体的关联代码还有很多，我们就不再深入了）。

```
1749 void sofia_queue_message(sofia_dispatch_event_t *de)
1750 {
...

1881     if (event == nua_i_invite && !sofia_private) {
...
1945         session = switch_core_session_request(sofia_endpoint_interface,
↳ SWITCH_CALL_DIRECTION_INBOUND, SOF_NONE, NULL);
```

接下来，在第 1950 行，初始化一个 `tech_pvt` 指针（该指针所指向的结构中将用于保存本 Session 的私有数据，我们后面还会讲到）。第 1962 行，将这些私有数据与 `session` 绑定。然后，在第 1981 行，为该 Session 启动一个新的线程，以执行后续的耗时的操作，避免阻塞当前的线程。

```
1950         tech_pvt = sofia_glue_new_pvt(session);
...
1962         sofia_glue_attach_private(session, profile, tech_pvt, channel_name);
...
1981         if (switch_core_session_thread_launch(session) != SWITCH_STATUS_SUCCESS) {
```

当然，启动了新线程后，对该 SIP 事件的处理还没有完，它还会后续设置 `de` 指针，并将收到的消息通过第 1777 行的 `switch_queue_push` 将推到一个模块级的消息队列中去（即这里的 `msg_queue`）。

```
2003         de->init_session = session;
...
1777     switch_queue_push(mod_sofia_globals.msg_queue, de);
1778 }
```

至此，SIP 事件的接收就完成了。如果后续收到其它 SIP 事件，将进行下次回调，并推到队列中等待处理。

SIP 消息的处理 对 SIP 事件的处理是在单独的线程（组）中执行的。进行事件处理的线程是在模块加载时从 `sofia_msg_thread_start` 函数开始的。该函数定义于 `sofia.c:1715`，它首先会启动一个新线程，并在以后根据 CPU 的数量以及当前的需要决定启动多个消息处理线程。从第 1738 行可以看出，新的事件处理线程中将执行 `sofia_msg_thread_run` 函数。

```
1715 void sofia_msg_thread_start(int idx)
1716 {
...
1736     switch_thread_create(&mod_sofia_globals.msg_queue_thread[i],
1737                          thd_attr,
1738                          sofia_msg_thread_run,
```

我们继续跟踪，就发现 `sofia_msg_thread_run` 是在第 1671 行定义的。它在第 1691 行使用一个无限循环，不断地从消息队列中取出一条消息（事件），然后在第 1700 行使用 `sofia_process_dispatch_event` 函数发送出去。

```
1671 void *SWITCH_THREAD_FUNC sofia_msg_thread_run(
                                switch_thread_t *thread, void *obj)
1672 {
...
1691     for(;;) {
1692
1693         if (switch_queue_pop(q, &pop) != SWITCH_STATUS_SUCCESS) {
...
1699             sofia_dispatch_event_t *de = (sofia_dispatch_event_t *) pop;
1700             sofia_process_dispatch_event(&de);
```

看来还得继续跟踪，`sofia_process_dispatch_event` 的定义是在第 1643 行，当看到第 1652 行时，我们总算看到了一点曙光，它终于好像在调用一个回调函数了。

```
1643 void sofia_process_dispatch_event(sofia_dispatch_event_t **dep)
1644 {
...
1652     our_sofia_event_callback(de->data->e_event, de->data->e_status,
                                de->data->e_phrase, de->nua, de->profile,
1653     de->nh, sofia_private, de->sip, de, (tagi_t *)de->data->e_tags);
```

继续往下跟踪可以确定我们的猜测，在第 1024 行找到 `our_sofia_event_callback` 的定义后，可以看到它确实是在处理 SIP 消息了。在第 1130 行的 `switch` 语句的各个分支中，我们可以看到许多

以 `nua_r_` 和 `nua_i_` 开头的 SIP `event`，其中，前者表示收到一条响应（Response）消息，而后者表示收到一条请求消息。

```

1024 static void our_sofia_event_callback(nua_event_t event,
1025     int status,
1026     char const *phrase,
1027     nua_t *nua, sofia_profile_t *profile, nua_handle_t *nh,
1028     sofia_private_t *sofia_private, sip_t const *sip,
1029     sofia_dispatch_event_t *de, tagi_t tags[])
1029 {
1030     ...
1031     switch (event) {
1032     case nua_r_get_params:
1033     case nua_i_fork:

```

我们集中精力看 `INVITE` 消息，如果收到 `INVITE` 消息，则第 1247 行的 `case` 语句成立。继续判断如果第 1249 行的条件成立，则说明是一个 `re-INVITE` 消息，在第 1250 行进行处理。否则，则说明是一个新的 `INVITE` 消息，在第 1253 行调用 `sofia_handle_sip_i_invite` 处理。

```

1247     case nua_i_invite:
1248         if (session && sofia_private) {
1249             if (sofia_private->is_call > 1) {
1250                 sofia_handle_sip_i_reinvite(...)
1251             } else {
1252                 sofia_private->is_call++;
1253                 sofia_handle_sip_i_invite(session, nua, profile, nh,
1254                                         sofia_private, sip, de, tags);
1255             }
1256         }

```

在 `sofia_handle_sip_i_invite` 中，将更深入的解析 `INVITE` 消息，对 Session 的相关内容进行更新，如果需要对方进行认证，还需要给对方发送 SIP 407 消息进行挑战认证等。该函数是在第 7845 行定义的，有兴趣的读者可以找到它研究一下，在此，我们就不再往里钻了，而是来看一个更重要的 SIP 事件。

SIP 状态机 在 Sofia-SIP 底层，也实现了一个状态机，在 SIP 通话的不同阶段使用不同的状态进行表示和处理。因而，在 SIP 状态发生改变时，它便向上层上报状态变化事件，这些状态变化事件也是在 SIP 事件的形式上报的，因而会经过跟上述的 `INVITE` 消息一致的回调过程一直到同一个回调函数 `our_sofia_event_callback`。在该函数的第 1265 行，我们会看到在收到 Sofia-SIP 底层驱动的状态变化后，是继续回调 `sofia_handle_sip_i_state` 函数来处理的。

```
1265     case nua_i_state:
1266         sofia_handle_sip_i_state(session, status, phrase, nua,
                                profile, nh, sofia_private, sip, de, tags);
```

`sofia_handle_sip_i_state` 函数由于有太多的状态和情况需要处理，因此也非常长。我们很难通过直接阅读源码的方式找到正确的入口。看起来，我们代码剖析到最后，马上就要迷失了。

不过，办法总比困难多。在本章中，我们过多的关注了理论去少了实践。下面让我们拿起一个 SIP 电话，拨打 9196，很快，就可以在日志中看到如下的信息：

```
[DEBUG] sofia.c:5861 ... Channel entering state [received][100]
```

从上一条日志可以看出，在第 5861 行打印了一条日志，表示我们的状态机进入了收到 INVITE 消息后发送 100 Trying 消息的阶段（代码略）。而接着下一条日志则告诉我们 Channel 的状态从 `CS_NEW` 变成了 `CS_INIT`。

```
[DEBUG] sofia.c:6116 ... State Change CS_NEW -> CS_INIT
```

有了上述信息，我们就可以在 `sofia.c` 的第 6116 行很快找到它了。只要满足一定的条件，在该行就会把 Channel 的状态变为 `CS_INIT`，然后，Channel 的核心状态机就会回调相关的状态回调函数了。

```
5773 static void sofia_handle_sip_i_state(...)
5778 {
...
6115         if (switch_channel_get_state(channel) == CS_NEW) {
6116             switch_channel_set_state(channel, CS_INIT);
```

Channel 状态机 只要 Channel 的状态一变成 `CS_INIT`，FreeSWITCH 核心的状态机代码就负责各种状态变化了，因而，各 Endpoint 模块就不需要再自己维护状态机了。也就是说，在一个 Endpoint 模块，首先要有一定的机制可以初始化一个 Session（对应一个 Channel，它的初始状态将为 `CS_NEW`），然后在适当的时候把该 Channel 的状态变成 `CS_INIT`，剩下的事就基本不管了。

当然，这里说的是基本不用管，但一般来说，还是要在 Endpoint 模块中跟踪 Channel 状态机的变化，这就需要靠在核心状态机上注册相应的回调函数实现，如 Sofia Channel 的状态机的回调是在第 4012 行定义的。

```
4012 switch_state_handler_table_t sofia_event_handlers = {
4013     /*.on_init */ sofia_on_init,
4014     /*.on_routing */ sofia_on_routing,
4015     /*.on_execute */ sofia_on_execute,
4016     /*.on_hangup */ sofia_on_hangup,
4017     /*.on_exchange_media */ sofia_on_exchange_media,
4018     /*.on_soft_execute */ sofia_on_soft_execute,
4019     /*.on_consume_media */ NULL,
4020     /*.on_hibernate */ sofia_on_hibernate,
4021     /*.on_reset */ sofia_on_reset,
4022     /*.on_park */ NULL,
4023     /*.on_reporting */ NULL,
4024     /*.on_destroy */ sofia_on_destroy
4025 };
```

这些回调函数是收我们在第 21.3.1 节讲过的和 5553 行的代码注册到核心中去的。回调函数本身都比较简单，感兴趣的读者可以自己看一下代码。为了节省篇幅，在此，我们就不多列举了。

IO 例程 与 Channel 状态机回调相比，Endpoint 模块中更重要的是 IO 例程的回调。IO 例程主要提供媒体数据的输入输出 (IO) 功能。与上一节讲的 Channel 的状态机类似，IO 例程的回调函数是在第 21.3.1 节的第 5552 行注册到核心中去的。其中，IO 例程的回调函数是由一个 `switch_io_routines_t` 类型的结构体变量设置的，该变量的定义在第 3997 行。

```
3997 switch_io_routines_t sofia_io_routines = {
3998     /*.outgoing_channel */ sofia_outgoing_channel,
3999     /*.read_frame */ sofia_read_frame,
4000     /*.write_frame */ sofia_write_frame,
4001     /*.kill_channel */ sofia_kill_channel,
4002     /*.send_dtmf */ sofia_send_dtmf,
4003     /*.receive_message */ sofia_receive_message,
4004     /*.receive_event */ sofia_receive_event,
4005     /*.state_change */ NULL,
4006     /*.read_video_frame */ sofia_read_video_frame,
4007     /*.write_video_frame */ sofia_write_video_frame,
4008     /*.state_run*/ NULL,
4009     /*.get_jb*/ sofia_get_jb
4010 };
```

在 21.1.5 节，我们已经讲过了 `outgoing_channel` 的回调，该回调是在有外呼请求的时候（如，执行“`originate sofia/gateway/...`”时）被回调执行的。在此，我们再来看一下 `mod_sofia` 中的 `outgoing_channel` 有何不同。

该模块的 `outgoing_channel` 回调是在第 4032 行定义的。在第 4057 ~ 4058 行，它也是初始化了一个新的 Session (`nsession`)，然后初始化了一个新的 `tech_pvt` 用于存放私有数据。第 4065 行还是从 `outbound_profile` 中复制被叫号码，第 4071 行得到对应的 Channel (`nchannel`)。如果该外呼是由 `bridge` 发起的，则还会有 a-leg 存在，因而，在第 4074 行将得到 a-leg 对应的 Channel，我们新生成的 `nchannel` 即是 b-leg。

```
4032 static switch_call_cause_t sofia_outgoing_channel(...
...
4057     if (!(nsession = switch_core_session_request_uuid(
        sofia_endpoint_interface, SWITCH_CALL_DIRECTION_OUTBOUND,
4058         flags, pool, switch_event_get_header(var_event,
            "origination_uuid")))) {

4063     tech_pvt = sofia_glue_new_pvt(nsession);
4064
4065     data = switch_core_session_strdup(nsession,
        outbound_profile->destination_number);
...
4071     nchannel = switch_core_session_get_channel(nsession);
4072
4073     if (session) {
4074         o_channel = switch_core_session_get_channel(session);
4075     }
```

接下来的各种判断还是非常冗长。之后，在第 4346 行将 `tech_pvt` 与 `nsession` 关联。

```
4346     sofia_glue_attach_private(nsession, profile, tech_pvt, dest);
```

从第 4428 ~ 4430 行可以看出，`nchannel` 的状态变为了 `CS_INIT`。然后，该 Channel 便进入正常的呼叫流程了。接下来，核心的状态机会接管接下来的状态变化，如将状态机置为 `CS_ROUTING`，然后进行路由查找（即查找 Dialplan），然后进入 `CS_EXECUTE` 状态，执行在 Dialplan 中找到的各种 App 等。

```
4428     if (switch_channel_get_state(nchannel) == CS_NEW) {
4429         switch_channel_set_state(nchannel, CS_INIT);
4430     }
```

当代码中某处调用 `switch_core_session_read_frame` 试图读取一帧音频数据时（如 21.1.4 节中的情况），就会执行 `read_frame` 回调函数，因而会回调第 929 行定义的函数。该回调函数由于将大

部分功能都移动到核心的 Core Media 代码中去了，因而非常简单，它主要就是在第 964 行调用核心的 `switch_core_media_read_frame` 从底层的 RTP 中读取音频数据。

```
929 static switch_status_t sofia_read_frame(switch_core_session_t *session,
      switch_frame_t **frame, switch_io_flag_t flags, int stream_id)
930 {
...
964     status = switch_core_media_read_frame(session, frame, flags,
      stream_id, SWITCH_MEDIA_TYPE_AUDIO);
...
968     return status;
969 }
```

当然，写数据的情况与此差不多，`write_frame` 回调函数在第 971 行定义。它第 1008 行也是调用了 Core Media 中的函数 `switch_core_media_write_frame` 通过 RTP 将音频数据发送出去。

```
971 static switch_status_t sofia_write_frame(switch_core_session_t *session,
      switch_frame_t *frame, switch_io_flag_t flags, int stream_id)
972 {
...
1008     if (switch_core_media_write_frame(session, frame, flags,
      stream_id, SWITCH_MEDIA_TYPE_AUDIO)) {
```

视频的回调函数 `read_video_frame` 和 `write_video_frame` 与此差不多。我们就不多讲了。最后，还有一个比较有意思的 `receive_message` 回调。看第 1096 行定义的回调函数。其中，在第 1123 行和第 1244 行各一个 `switch` 语句用于判断收到的各种消息，并进行相应的处理。我们直接跳到第 2031 行，在收到 `SWITCH_MESSAGE_INDICATE_ANSWER` 消息时，它将在第 2031 行调用 `sofia_answer_channel` 进当前通话进行应答。

```
1096 static switch_status_t sofia_receive_message(switch_core_session_t *session,
      switch_core_session_message_t *msg)
1097 {
...
1123     switch (msg->message_id) {
...
1244     switch (msg->message_id) {
...
2031     case SWITCH_MESSAGE_INDICATE_ANSWER:
2032         status = sofia_answer_channel(session);
```

很容易想象到，应答将会向对方发送 SIP 200 OK 消息。而它就是在第 608 行调用 Sofia-SIP 底层库实现的。

```
602 static switch_status_t sofia_answer_channel(switch_core_session_t *session)
603 {
...
608         nua_respond(tech_pvt->nh, SIP_200_OK, ...
```

让我们再回到第 21.1.2 节的 `answer` App，就可以看到它调用了核心的 `switch_answer_channel` 函数，在第 `switch_channel.c: 3716` 发送了一个 `SWITCH_MESSAGE_INDICATE_ANSWER` 消息，因而 `sofia_receive_message` 函数被回调，代码就执行到了第 2032 行，并最终在第 608 行向对方的 SIP 终端发送 200 OK 消息。

至此，我们所有的呼叫流程就全部都串起来了，我们对源代码的分析也到此结束。

2.2.4 小结

在本章，我们首先讲了 APR 库数据结构、设计理念和原则等相关知识。FreeSWITCH 的源代码依赖于它做跨平台的支持，而且代码风格跟 APR 也非常像，所以很好地了解 APR 对于理解 FreeSWITCH 的源代码是很有帮助的。

在源代码阅读中，我们没有过多地关注有关“互斥”（Mutex）和“锁”的代码。而实际上，尤其是对于 FreeSWITCH 这样的多线程的模型的系统来讲，对临界区（多个线程同时访问的资源）加“锁”是很重要的，而且在使用时一定要非常小心以避免产生竞争条件（Race Condition）或死锁（Deadlock）。不过，在本章，我们更关注代码的设计理念、封装方式、执行逻辑和流程，使读者在阅读时很快找到“切入点”和“头绪”，以便能更加深入的研究下去。

通过对核心源代码架构的把握，相信读者脑子里已经对系统的整体结构有了一个整体的概念。接下来，我们对三个最有代表性的模块的源代码进行了深入剖析。结合上一章所学的内容，从我们最熟悉的 `echo`、`answer` 等 App 开始做为突破口，一步一步的深入跟踪，终于理清了代码的执行流程，了解了各种回调函数的含义及触发时机。同时，我们也从模块启动、网络监听、来话的接收、Session 的生成、各种状态的转移直到应答等全部的流程都进行了跟踪和梳理。

通过本章的学习、然后配合系统的运行日志，更深入的学习和研究源代码应该没有障碍了。当然，FreeSWITCH 的代码非常多，学习源代码更多的是需要细心和耐心。退一万万步讲，这些代码是用了将近十年的时间写成的，不要指望一天就精通。另外，笔者也不可能在短短几章内把所有的概念、方法、流程讲清楚。掌握 FreeSWITCH 的代码还需要阅读代码、多实践、多调试，总之，多下功夫。

2.3 Endpoint 接口

本章基于 Commit Hash [1681db4](#)。

Endpoint 模块是实现各种对接协议的模块，这类模块是最复杂的。本节，我们就来分析几个比较简单的 Endpoint 接口实现，以便于理解更复杂的模块。

在 FreeSWITCH 控制台上使用 `show endpoint` 命令就可以列出已加载的 Endpoint 接口和模块。

```
freeswitch> show endpoint

type,name,ikey
endpoint,error,mod_dptools
endpoint,group,mod_dptools
endpoint,loopback,mod_loopback
endpoint,modem,mod_spandsp
endpoint,null,mod_loopback
endpoint,pickup,mod_dptools
endpoint,rtc,mod_rtc
endpoint,rtp,mod_sofia
endpoint,sofia,mod_sofia
endpoint,user,mod_dptools
endpoint,verto.rtc,mod_verto

11 total.
```

2.3.1 rtp Endpoint

我们先来看 `rtp` 接口。它是在 `mod_sofia` 模块中实现的。这个模块仅有 600 多行，但也算是一个比较完整的模块，且比较便于阅读。

最开始先定义一些参数和数据结构，备用，这些字符串可以作为呼叫字符串的参数。

```
35 #define KLOCALADDR "local_addr"
36 #define KLOCALPORT "local_port"
37 #define KREMOTEADDR "remote_addr"
38 #define KREMOTEPORT "remote_port"
39 #define KCODEC "codec"
40 #define KPTIME "ptime"
41 #define KPT "pt"
42 #define KRFC2833PT "rfc2833_pt"
43 #define KMODE "mode"
44 #define KRATE "rate"
```

```

45
46 static struct {
47     switch_memory_pool_t *pool;
48     switch_endpoint_interface_t *endpoint_interface;
49 } crtp;
50
51 typedef struct {
52     switch_core_session_t *session;
53     switch_channel_t *channel;
54     switch_codec_t read_codec, write_codec;
55     switch_frame_t read_frame;
59     switch_rtp_bug_flag_t rtp_bugs;
60     switch_rtp_t *rtp_session;
61
62     uint32_t timestamp_send;
63
64     const char *local_address;
65     const char *remote_address;
66     const char *codec;
67     int ptime;
68
69     const switch_codec_implementation_t *negotiated_codecs[SWITCH_MAX_CODECS];
70     int num_negotiated_codecs;
71
72     char *origin;
73
74     switch_port_t local_port;
75     switch_port_t remote_port;
76     switch_payload_t agreed_pt; /*XXX*/
77     switch_core_media_dtmf_t dtmf_type;
78     enum {
79         RTP_SENDFONLY,
80         RTP_RECVONLY,
81         RTP_SENDRECV
82     } mode;
83 } crtp_private_t;

```

定义状态机的回调。本模块非常简单，只定义了初始化（`on_init`）和销毁（`on_destroy`）两个回调函数。

```

98 switch_state_handler_table_t crtp_state_handlers = {
99     /*on_init */channel_on_init,
100    /*on_routing */ NULL,
101    /*on_execute */ NULL,
102    /*on_hangup*/ NULL,
103    /*on_exchange_media*/ NULL,

```

```

104     /*on_soft_execute*/ NULL,
105     /*on_consume_media*/ NULL,
106     /*on_hibernate*/ NULL,
107     /*on_reset*/ NULL,
108     /*on_park*/ NULL,
109     /*on_reporting*/ NULL,
110     /*on_destroy*/ channel_on_destroy
111
112 };

```

IO 例程，负责输入输入。本模块仅支持音频。

```

114 switch_io_routines_t crtp_io_routines = {
115     /*outgoing_channel*/ channel_outgoing_channel,
116     /*read_frame*/ channel_read_frame,
117     /*write_frame*/ channel_write_frame,
118     /*kill_channel*/ NULL,
119     /*send_dtmf*/ channel_send_dtmf,
120     /*receive_message*/ channel_receive_message,
121     /*receive_event*/ channel_receive_event,
122     /*state_change*/ NULL,
123     /*read_video_frame*/ NULL,
124     /*write_video_frame*/ NULL,
125     /*read_text_frame*/ NULL,
126     /*write_text_frame*/ NULL,
127     /*state_run*/ NULL
128 };

```

初始化。本函数的 `module_interface` 是一个模块接口，在被调用时由 `mod_sofia` 模块传入。

L135，首先定义一个 `switch_endpoint_interface_t` 类型的接口。L138，使用模块的内存池。创建一个 Endpoint 接口（L139）并设置接口的名称为 `rtp`（L140）。接下来设置 IO 程（L141）和状态回调函数（L142）。

```

133 void crtp_init(switch_loadable_module_interface_t *module_interface)
134 {
135     switch_endpoint_interface_t *endpoint_interface;
136
137     crtp.pool = module_interface->pool;
138     endpoint_interface = switch_loadable_module_create_interface(module_interface,
139     ↪ SWITCH_ENDPOINT_INTERFACE);
140     endpoint_interface->interface_name = "rtp";
141     endpoint_interface->io_routines = &crtp_io_routines;

```

```

142     endpoint_interface->state_handler = &rtp_state_handlers;
143     crtp.endpoint_interface = endpoint_interface;
146 }

```

`outgoing_channel` 是在外呼是回调的，也就是说在比如 `originate rtp/1234 &playback()` 或这样的命令 `bridge rtp/1234` 产生一个新 Channel 进行外呼时调用。

其中第一个参数是一个 `session`，在 `originate` 调用中 `session` 为 `NULL`，因为是单腿的呼叫，而在 `bridge` 调用时，`session` 为 `a-leg`。

如果执行成功，该函数返回一个 `new_session` 指针。

`var_event` 是一个 `switch_event_t` 的数据结构，里面存储了一些“键值对”。如在外呼时可以这样指定 IP 地址和端口：

```

originate {local_addr=192.168.0.1,local_port=4000,remote_addr=192.168.0.2,remote_port=4000}rtp &playback(/
↳ tmp/test.wav)

```

上面是我们熟悉的命令行，FreeSWITCH 会解析 `{}` 中的参数，变成 `var_event`。L163 ~ L172 就是获取这些参数。L175 将字符串转换成整数(`switch_port_t`)的端口值。同样，打包时间 (`ptime`)、采样率 (`rate`) 以及负载类型 (`pt`) 也要转换成整数 (L178 ~ L181)。

```

148 static switch_call_cause_t channel_outgoing_channel(switch_core_session_t *session, switch_event_t
↳ *var_event,
149             switch_caller_profile_t *outbound_profile,
150             switch_core_session_t **new_session,
151             switch_memory_pool_t **pool,
152             switch_originate_flag_t flags, switch_call_cause_t *cancel_cause)
153 {
154     switch_channel_t *channel;
155     char name[128];
156     crtp_private_t *tech_pvt = NULL;
157     switch_caller_profile_t *caller_profile;
158     switch_rtp_flag_t rtp_flags[SWITCH_RTP_FLAG_INVALID] = {0};
159
160     const char *err;
161
162
163     const char *local_addr = switch_event_get_header_nil(var_event, kLOCALADDR),
164             *szlocal_port = switch_event_get_header_nil(var_event, kLOCALPORT),
165             *remote_addr = switch_event_get_header_nil(var_event, kREMOTEADDR),
166             *szremote_port = switch_event_get_header_nil(var_event, kREMOTEPORT),
167             *codec = switch_event_get_header_nil(var_event, kCODEC),
168             *szptime = switch_event_get_header_nil(var_event, kPTIME),

```

```

169         /*mode = switch_event_get_header_nil(var_event, kMODE),
170         /*szrfc2833_pt = switch_event_get_header_nil(var_event, kRFC2833PT),
171         *szrate = switch_event_get_header_nil(var_event, kRATE),
172         *szpt = switch_event_get_header_nil(var_event, kPT);
173
174
175     switch_port_t local_port = !zstr(szlocal_port) ? (switch_port_t)atoi(szlocal_port) : 0,
176         remote_port = !zstr(szremote_port) ? (switch_port_t)atoi(szremote_port) : 0;
177     int ptime = !zstr(szptime) ? atoi(szptime) : 0,
178         //rfc2833_pt = !zstr(szrfc2833_pt) ? atoi(szrfc2833_pt) : 0,
179         rate = !zstr(szrate) ? atoi(szrate) : 8000,
180         pt = !zstr(szpt) ? atoi(szpt) : 0;

```

参数合法性检查。

```

183     if (
184         ((zstr(remote_addr) || remote_port == 0) && (zstr(local_addr) || local_port == 0)) ||
185         zstr(codec) ||
186         zstr(szpt)) {
187
188         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Missing required arguments\n");
189         goto fail;
190     }

```

产生一个新的 Session (L193) 并获取其对应的 Channel (L198)。tech_pvt 为该接口私有的数据结构，直接从 Session 的内存池中申请 (L200)，内存池最终会在 Session 销毁时自动释放。接下来只是记住一时数据 (L201 ~ L209)。

```

193     if (!(*new_session = switch_core_session_request(crtp.endpoint_interface,
↵ SWITCH_CALL_DIRECTION_OUTBOUND, 0, pool))) {
194         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Couldn't request session.\n");
195         goto fail;
196     }
197
198     channel = switch_core_session_get_channel(*new_session);
199
200     tech_pvt = switch_core_session_alloc(*new_session, sizeof *tech_pvt);
201     tech_pvt->session = *new_session;
202     tech_pvt->channel = channel;
203     tech_pvt->local_address = switch_core_session_strdup(*new_session, local_addr);
204     tech_pvt->local_port = local_port;
205     tech_pvt->remote_address = switch_core_session_strdup(*new_session, remote_addr);
206     tech_pvt->remote_port = remote_port;

```

```

207     tech_pvt->ptime = ptime;
208     tech_pvt->agreed_pt = (switch_payload_t)pt;
209     tech_pvt->dtmf_type = DTMF_2833; /* XXX */

```

如果没有远端 IP 地址和端口, 则该 RTP 就只能收数据 (L215), 否则, 则是双向的收发 (L217)。

```

211     if (zstr(local_addr) || local_port == 0) {
212         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "The local address and port must be
↪ set\n");
213         goto fail;
214     } else if (zstr(remote_addr) || remote_port == 0) {
215         tech_pvt->mode = RTP_RECVONLY;
216     } else {
217         tech_pvt->mode = RTP_SENDRXCV;
218     }

```

L220 将私有数据与 Session 相关联。创建一个 `switch_caller_profile_t` 结构的 Profile (L222 ~ L223), 包含主被叫号码等。设置 Channel 的名称 (L226 ~ L227) 并将 Channel 的状态设为初始化状态 (L229)。

```

220     switch_core_session_set_private(*new_session, tech_pvt);
221
222     caller_profile = switch_caller_profile_clone(*new_session, outbound_profile);
223     switch_channel_set_caller_profile(channel, caller_profile);
224
225
226     snprintf(name, sizeof(name), "rtp/%s", outbound_profile->destination_number);
227     switch_channel_set_name(channel, name);
228
229     switch_channel_set_state(channel, CS_INIT);

```

初始化编解码器。分别用于读 (L231) 和写 (L243)。

```

231     if (switch_core_codec_init(&tech_pvt->read_codec,
232                               codec,
233                               NULL,
234                               NULL,
235                               rate,
236                               ptime,
237                               1,

```

```

238             /*SWITCH_CODEC_FLAG_ENCODE |*/ SWITCH_CODEC_FLAG_DECODE,
239             NULL, switch_core_session_get_pool(tech_pvt->session)) !=
↳ SWITCH_STATUS_SUCCESS) {
240         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't load codec?\n");
241         goto fail;
242     } else {
243         if (switch_core_codec_init(&tech_pvt->write_codec,
244                                   codec,
245                                   NULL,
246                                   NULL,
247                                   rate,
248                                   ptime,
249                                   1,
250                                   SWITCH_CODEC_FLAG_ENCODE /*| SWITCH_CODEC_FLAG_DECODE*/,
251                                   NULL, switch_core_session_get_pool(tech_pvt->session)) !=
↳ SWITCH_STATUS_SUCCESS) {
252             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't load codec?\n");
253             goto fail;
254         }
255     }

```

将编码器设置到 Session 上（`switch_core_io`在读写数据的时候要用到）。

```

257     if (switch_core_session_set_read_codec(*new_session, &tech_pvt->read_codec) !=
↳ SWITCH_STATUS_SUCCESS) {
258         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't set read codec?\n");
259         goto fail;
260     }
261
262     if (switch_core_session_set_write_codec(*new_session, &tech_pvt->write_codec) !=
↳ SWITCH_STATUS_SUCCESS) {
263         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't set write codec?\n");
264         goto fail;
265     }

```

初始化一个 RTP Session。

```

267     if (!(tech_pvt->rtp_session = switch_rtp_new(local_addr, local_port, remote_addr, remote_port,
↳ tech_pvt->agreed_pt,
268         tech_pvt->read_codec.implementation->samples_per_packet, ptime * 1000,
269         rtp_flags, "soft", &err, switch_core_session_get_pool(*new_session)))) {
270         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Couldn't setup RTP session: [%s]\n",
↳ err);

```



```
271     goto fail;
272 }
```

L274, 启动一个 Session 线程, 以后 Session 就会由核心接管。由于 `rtp` 只是媒体层的, 缺少信令支持, 在此, 直接把 Session 置为了应答状态 (L279)。

```
274     if (switch_core_session_thread_launch(*new_session) != SWITCH_STATUS_SUCCESS) {
275         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Couldn't start session thread.\n");
276         goto fail;
277     }
278
279     switch_channel_mark_answered(channel);
280
281     return SWITCH_CAUSE_SUCCESS;
```

如果失败则会清理现场并返回错误。

```
283 fail:
284     if (tech_pvt) {
285         if (tech_pvt->read_codec.implementation) {
286             switch_core_codec_destroy(&tech_pvt->read_codec);
287         }
288
289         if (tech_pvt->write_codec.implementation) {
290             switch_core_codec_destroy(&tech_pvt->write_codec);
291         }
292     }
293
294     if (*new_session) {
295         switch_core_session_destroy(new_session);
296     }
297     return SWITCH_CAUSE_DESTINATION_OUT_OF_ORDER;
298 }
```

在 Channel 初始化阶段的回调函数中, 会将 Channel 设置为 `CS_CONSUME_MEDIA` (L305) 状态, 消费媒体。

```
300 static switch_status_t channel_on_init(switch_core_session_t *session)
301 {
302
```

```
303     switch_channel_t *channel = switch_core_session_get_channel(session);
304
305     switch_channel_set_state(channel, CS_CONSUME_MEDIA);
306
307     return SWITCH_STATUS_FALSE;
308 }
```

销毁时清理现场，如私有的数据等。

```
310 static switch_status_t channel_on_destroy(switch_core_session_t *session)
311 {
312     crtp_private_t *tech_pvt = NULL;
313
314     if ((tech_pvt = switch_core_session_get_private(session))) {
315
316         if (tech_pvt->read_codec.implementation) {
317             switch_core_codec_destroy(&tech_pvt->read_codec);
318         }
319
320         if (tech_pvt->write_codec.implementation) {
321             switch_core_codec_destroy(&tech_pvt->write_codec);
322         }
323     }
324
325     return SWITCH_STATUS_SUCCESS;
326 }
```

读音频数据。该函数是在 `switch_core_io` 里回调的，核心会不停地调用该函数收数据。如果在读的过程中检查到 DTMF (L346)，则放到 DTMF 队列里 (L349)。L354 会调用 `switch_rtp.c` 里的函数读从 Socket 上收数据。读到后，会返回一个 `switch_frame_t` 结构的指针 `frame`。`frame->data` 是实际的数据，`frame->datalen` 则为数据的长度。如果读不到数据，则会返回静音 (CNG, L363)，`frame` 上包含 `SFF_CNG` 标志 (L366)。

```
329 static switch_status_t channel_read_frame(switch_core_session_t *session, switch_frame_t **frame,
↪ switch_io_flag_t flags, int stream_id)
330 {
331     crtp_private_t *tech_pvt;
332     switch_channel_t *channel;
333     switch_status_t status;
334
335     channel = switch_core_session_get_channel(session);
336     assert(channel != NULL);
337 }
```

```

338     tech_pvt = switch_core_session_get_private(session);
339     assert(tech_pvt != NULL);
340
341     if (!tech_pvt->rtp_session || tech_pvt->mode == RTP_SENDBY) {
342         switch_yield(20000); /* replace by local timer XXX */
343         goto cng;
344     }
345
346     if (switch_rtp_has_dtmf(tech_pvt->rtp_session)) {
347         switch_dtmf_t dtmf = { 0 };
348         switch_rtp_dequeue_dtmf(tech_pvt->rtp_session, &dtmf);
349         switch_channel_queue_dtmf(channel, &dtmf);
350     }
351
352     tech_pvt->read_frame.flags = SFF_NONE;
353     tech_pvt->read_frame.codec = &tech_pvt->read_codec;
354     status = switch_rtp_zerocopy_read_frame(tech_pvt->rtp_session, &tech_pvt->read_frame, flags);
355
356     if (status != SWITCH_STATUS_SUCCESS && status != SWITCH_STATUS_BREAK) {
357         goto cng;
358     }
359
360     *frame = &tech_pvt->read_frame;
361     return SWITCH_STATUS_SUCCESS;
362
363 cng:
364     *frame = &tech_pvt->read_frame;
365     tech_pvt->read_frame.codec = &tech_pvt->read_codec;
366     tech_pvt->read_frame.flags |= SFF_CNG;
367     tech_pvt->read_frame.datalen = 0;
368
369     return SWITCH_STATUS_SUCCESS;
370 }

```

发送 RTP。FreeSWITCH 核心也是定期调用该回调发送 RTP 数据，L402 行会调用 `switch_rtp.c` 中的函数发送。

```

372 static switch_status_t channel_write_frame(switch_core_session_t *session, switch_frame_t *frame,
↪ switch_io_flag_t flags, int stream_id)
373 {
374     crtp_private_t *tech_pvt;
375     switch_channel_t *channel;
376     //int frames = 0, bytes = 0, samples = 0;
377
378     channel = switch_core_session_get_channel(session);
379     assert(channel != NULL);

```

```
380
381     tech_pvt = switch_core_session_get_private(session);
382     assert(tech_pvt != NULL);
383
398     if (tech_pvt->mode == RTP_RECVONLY) {
399         return SWITCH_STATUS_SUCCESS;
400     }
401
402     switch_rtp_write_frame(tech_pvt->rtp_session, frame);
403
404     return SWITCH_STATUS_SUCCESS;
405 }
```

发送 DTMF。仅支持 RFC2833 类型 (L415)，也是调用 `switch_rtp.c` 中的函数将 DTMF 推送到队列中等待发送 (L418)。

```
407 static switch_status_t channel_send_dtmf(switch_core_session_t *session, const switch_dtmf_t *dtmf)
408 {
409     crtp_private_t *tech_pvt = NULL;
410
411     tech_pvt = switch_core_session_get_private(session);
412     assert(tech_pvt != NULL);
413
414     switch(tech_pvt->dtmf_type) {
415         case DTMF_2833:
416             {
417                 switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Enqueuing RFC2833
↪ DTMF %c of length %d\n", dtmf->digit, dtmf->duration);
418                 return switch_rtp_queue_rfc2833(tech_pvt->rtp_session, dtmf);
419             }
420         case DTMF_NONE:
421             default:
422             {
423                 switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Discarding DTMF %c
↪ of length %d, DTMF type is NONE\n", dtmf->digit, dtmf->duration);
424             }
425         }
426
427     return SWITCH_STATUS_SUCCESS;
428 }
```

比较字符串。

```

430 static switch_bool_t compare_var(switch_event_t *event, switch_channel_t *channel, const char *varname)
431 {
432     const char *chan_val = switch_channel_get_variable_dup(channel, varname, SWITCH_FALSE, -1);
433     const char *event_val = switch_event_get_header(event, varname);
434
435     if (zstr(chan_val) || zstr(event_val)) {
436         return 1;
437     }
438
439     return strcasecmp(chan_val, event_val);
440 }

```

FreeSWITCH 也支持通过事件机制进行通信，事件机制是异步的。可以在通话过程中更改媒体信息。如 L470 更改远端的地址、L511 更新编码等。

```

442 static switch_status_t channel_receive_event(switch_core_session_t *session, switch_event_t *event)
443 {
444     const char *command = switch_event_get_header(event, "command");
445     switch_channel_t *channel = switch_core_session_get_channel(session);
446     crtp_private_t *tech_pvt = switch_core_session_get_private(session);
447     char *codec = switch_event_get_header_nil(event, kCODEC);
448     char *szptime = switch_event_get_header_nil(event, kPTIME);
449     char *szrate = switch_event_get_header_nil(event, kRATE);
450     char *szpt = switch_event_get_header_nil(event, kPT);
451
452     int ptime = !zstr(szptime) ? atoi(szptime) : 0,
453         rate = !zstr(szrate) ? atoi(szrate) : 8000,
454         pt = !zstr(szpt) ? atoi(szpt) : 0;
455
456
457     if (!zstr(command) && !strcasecmp(command, "media_modify")) {
458         /* Compare parameters */
459         if (compare_var(event, channel, kREMOTEADDR) ||
460             compare_var(event, channel, kREMOTEPORT)) {
461             char *remote_addr = switch_event_get_header(event, kREMOTEADDR);
462             char *szremote_port = switch_event_get_header(event, kREMOTEPORT);
463             switch_port_t remote_port = !zstr(szremote_port) ? (switch_port_t)atoi(szremote_port) : 0;
464             const char *err;
465
466
467             switch_channel_set_variable(channel, kREMOTEADDR, remote_addr);
468             switch_channel_set_variable(channel, kREMOTEPORT, szremote_port);
469
470             if (switch_rtp_set_remote_address(tech_pvt->rtp_session, remote_addr, remote_port, 0,
↳ SWITCH_TRUE, &err) != SWITCH_STATUS_SUCCESS) {
471                 switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Error setting
↳ RTP remote address: %s\n", err);

```

```

472         } else {
473             switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Set RTP
↳ remote: %s:%d\n", remote_addr, (int)remote_port);
474             tech_pvt->mode = RTP_SENDRX;
475         }
476     }
477
478     if (compare_var(event, channel, kCODEC) ||
479         compare_var(event, channel, kPTIME) ||
480         compare_var(event, channel, kPT) ||
481         compare_var(event, channel, kRATE)) {
482         /* Reset codec */
483         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_CRIT, "Switching codec
↳ updating \n");
484
485         if (switch_core_codec_init(&tech_pvt->read_codec,
486                                   codec,
487                                   NULL,
488                                   NULL,
489                                   rate,
490                                   ptime,
491                                   1,
492                                   /*SWITCH_CODEC_FLAG_ENCODE */ SWITCH_CODEC_FLAG_DECODE,
493                                   NULL, switch_core_session_get_pool(tech_pvt->session)) != SWITCH_STATUS_SUCCESS) {
494             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't load codec?\n");
495             goto fail;
496         } else {
497             if (switch_core_codec_init(&tech_pvt->write_codec,
498                                       codec,
499                                       NULL,
500                                       NULL,
501                                       rate,
502                                       ptime,
503                                       1,
504                                       SWITCH_CODEC_FLAG_ENCODE /*| SWITCH_CODEC_FLAG_DECODE*/,
505                                       NULL, switch_core_session_get_pool(tech_pvt->session)) !=
↳ SWITCH_STATUS_SUCCESS) {
506                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't load codec?\n");
507                 goto fail;
508             }
509         }
510
511         if (switch_core_session_set_read_codec(session, &tech_pvt->read_codec) != SWITCH_STATUS_SUCCESS)
↳ {
512             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't set read codec?\n");
513             goto fail;
514         }
515

```

```
516     if (switch_core_session_set_write_codec(session, &tech_pvt->write_codec) !=  
↪ SWITCH_STATUS_SUCCESS) {  
517         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Can't set write codec?\n");  
518         goto fail;  
519     }  
520  
521     switch_rtp_set_default_payload(tech_pvt->rtp_session, (switch_payload_t)pt);  
522     //switch_rtp_set_recv_pt(tech_pvt->rtp_session, pt);  
523 }  
524  
525 if (compare_var(event, channel, kRFC2833PT)) {  
526     const char *szpt = switch_channel_get_variable(channel, kRFC2833PT);  
527     int pt = !zstr(szpt) ? atoi(szpt) : 0;  
528  
529     switch_channel_set_variable(channel, kRFC2833PT, szpt);  
530     switch_rtp_set_telephony_event(tech_pvt->rtp_session, (switch_payload_t)pt);  
531 }  
532  
533 } else {  
534     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Received unknown  
↪ command [%s] in event.\n", !command ? "null" : command);  
535 }  
536  
537 return SWITCH_STATUS_SUCCESS;  
538  
539 fail:  
540 if (tech_pvt->read_codec.implementation) {  
541     switch_core_codec_destroy(&tech_pvt->read_codec);  
542 }  
543  
544 if (tech_pvt->write_codec.implementation) {  
545     switch_core_codec_destroy(&tech_pvt->write_codec);  
546 }  
547  
548 switch_core_session_destroy(&session);  
549  
550 return SWITCH_STATUS_FALSE;  
551 }
```

FreeSWITCH 也支持消息机制通信，消息通信是同步调用的，因此在该回调函数中不能阻塞，执行要快。

比如，在命令行上执行 `uuid_debug_media`，该函数就会初回调，并收到 L561 行的消息。然后通过更新 RTP Session 上的标志（L578）让核心里的 `switch_rtp.c` 打印调试信息。

在需要音频同步的场合也会清空缓冲区（L590），以及调整 Jitter Buffer 等（593）。

```

553 static switch_status_t channel_receive_message(switch_core_session_t *session,
↪ switch_core_session_message_t *msg)
554 {
555     crtp_private_t *tech_pvt = NULL;
556
557     tech_pvt = switch_core_session_get_private(session);
558     assert(tech_pvt != NULL);
559
560     switch (msg->message_id) {
561         case SWITCH_MESSAGE_INDICATE_DEBUG_MEDIA:
562             {
563                 if (switch_rtp_ready(tech_pvt->rtp_session) && !zstr(msg->string_array_arg[0]) && !zstr(msg-
↪ >string_array_arg[1])) {
564                     switch_rtp_flag_t flags[SWITCH_RTP_FLAG_INVALID] = {0};
565                     int x = 0;
566
567                     if (!strcasecmp(msg->string_array_arg[0], "read")) {
568                         flags[SWITCH_RTP_FLAG_DEBUG_RTP_READ]++;x++;
569                     } else if (!strcasecmp(msg->string_array_arg[0], "write")) {
570                         flags[SWITCH_RTP_FLAG_DEBUG_RTP_WRITE]++;x++;
571                     } else if (!strcasecmp(msg->string_array_arg[0], "both")) {
572                         flags[SWITCH_RTP_FLAG_DEBUG_RTP_READ]++;x++;
573                         flags[SWITCH_RTP_FLAG_DEBUG_RTP_WRITE]++;
574                     }
575
576                     if (x) {
577                         if (switch_true(msg->string_array_arg[1])) {
578                             switch_rtp_set_flags(tech_pvt->rtp_session, flags);
579                         } else {
580                             switch_rtp_clear_flags(tech_pvt->rtp_session, flags);
581                         }
582                     } else {
583                         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Invalid
↪ Options\n");
584                     }
585                 }
586                 break;
587             }
588         case SWITCH_MESSAGE_INDICATE_AUDIO_SYNC:
589             if (switch_rtp_ready(tech_pvt->rtp_session)) {
590                 rtp_flush_read_buffer(tech_pvt->rtp_session, SWITCH_RTP_FLUSH_ONCE);
591             }
592             break;
593         case SWITCH_MESSAGE_INDICATE_JITTER_BUFFER:
594             {
595                 if (switch_rtp_ready(tech_pvt->rtp_session)) {
596                     int len = 0, maxlen = 0, qlen = 0, maxlen = 50;
597
598                     if (msg->string_arg) {

```



```
599         char *p;
600         const char *s;
601
602         if (!strcasecmp(msg->string_arg, "pause")) {
603             switch_rtp_pause_jitter_buffer(tech_pvt->rtp_session, SWITCH_TRUE);
604             goto end;
605         } else if (!strcasecmp(msg->string_arg, "resume")) {
606             switch_rtp_pause_jitter_buffer(tech_pvt->rtp_session, SWITCH_FALSE);
607             goto end;
608         } else if (!strncasecmp(msg->string_arg, "debug:", 6)) {
609             s = msg->string_arg + 6;
610             if (s && !strcmp(s, "off")) {
611                 s = NULL;
612             }
613             switch_rtp_debug_jitter_buffer(tech_pvt->rtp_session, s);
614             goto end;
615         }
616
617
618         if ((len = atoi(msg->string_arg)) {
619             qlen = len / (tech_pvt->read_codec.implementation->microseconds_per_packet /
620 ↪ 1000);
621             if (qlen < 1) {
622                 qlen = 3;
623             }
624         }
625         if (qlen) {
626             if ((p = strchr(msg->string_arg, ':')) {
627                 p++;
628                 maxlen = atol(p);
629             }
630         }
631
632
633         if (maxlen) {
634             maxqlen = maxlen / (tech_pvt->read_codec.implementation->microseconds_per_packet
635 ↪ / 1000);
636         }
637
638         if (qlen) {
639             if (maxqlen < qlen) {
640                 maxqlen = qlen * 5;
641             }
642             if (switch_rtp_activate_jitter_buffer(tech_pvt->rtp_session, qlen, maxqlen,
643 ↪ tech_pvt->read_codec.implementation-
644 ↪ >samples_per_packet,
```

```

644                                     tech_pvt->read_codec.implementation-
↳ >samples_per_second) == SWITCH_STATUS_SUCCESS) {
645                                     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(tech_pvt->session),
646                                                         SWITCH_LOG_DEBUG, "Setting Jitterbuffer to %dms (%d frames)
↳ (%d max frames)\n",
647                                                         len, qlen, maxlen);
648                                     switch_channel_set_flag(tech_pvt->channel, CF_JITTERBUFFER);
649                                     if (!switch_false(switch_channel_get_variable(tech_pvt->channel,
↳ "rtp_jitter_buffer_plc"))) {
650                                         switch_channel_set_flag(tech_pvt->channel, CF_JITTERBUFFER_PLC);
651                                     }
652                                     } else {
653                                         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(tech_pvt->session),
654                                                         SWITCH_LOG_WARNING, "Error Setting Jitterbuffer to %dms (%d
↳ frames)\n", len, qlen);
655                                     }
656
657                                     } else {
658                                         switch_rtp_deactivate_jitter_buffer(tech_pvt->rtp_session);
659                                     }
660                                     }
661                                     }
662                                     break;
663
664                                     default:
665                                         break;
666                                     }
667     end:
668     return SWITCH_STATUS_SUCCESS;
669 }

```

下面是一些使用实例：

“无”信令模式，将已经进来的呼叫发送到远端一个 RTP 端口上。

```

<action application="bridge"
↳ data="{local_addr=192.168.1.1,local_port=4444,remote_addr=192.168.1.124,remote_port=6666,codec=PCMU,pt=0,rate=8000}"
↳ rtp"

```

监听，将通话中的某一方音频直接发到某一 RTP 端口上：

```

originate
↳ {local_addr=192.168.7.6,local_port=4444,remote_addr=192.168.7.6,remote_port=6666,codec=PCMU,pt=0,rate=8000}rtp
↳ &eavesdrop(uuid)

```

呼叫某一 SIP 用户，`bridge` 到 `rtp`。

```
originate user/1000  
↪ &bridge({local_addr=192.168.7.6,local_port=6666,remote_addr=192.168.7.6,remote_port=4444,codec=PCMU,pt=0,rate=8000})rtp
```

将会议中的音频通过 RTP 发出。

```
conference 3000 dial  
↪ {local_addr=192.168.7.6,local_port=6666,remote_addr=192.168.7.6,remote_port=4444,codec=PCMU,pt=0,rate=8000}rtp
```

通过本节可以看出，收到 `rtp` Endpoint 缺少信令支持，因此需要手工指定各种参数。

在实际使用中，很多同学都问到 FreeSWITCH 怎么将媒体和信令分离。通过，本节的学习，就可以分离了吧？

但话说回来，媒体跟信令实际上是不能分离的。如果没有信令，那么媒体就不知道从哪里收往哪里发。所以，我们上面说的“手工指定各种参数”中的“**手工**”其实也是一种信令。如果不是手工，而这些参数是通过 ESL 来指定的，那 ESL 也就相关于一种信令。

当然，有的同学可能会说，在 FreeSWITCH 中如何“将 SIP 信令和媒体分离”，那就比较精确一点了。不过，SIP 信令是已经被实战证明了的正确的信令，为什么一定要把 SIP 分离掉呢？

2.3.2 rtc Endpoint

与 `rtp` Endpoint 相比，`rtc` Endpoint 更短，只有 400 多行。

严格来说，`rtc` 不是一个完整的 Endpoint，它是在写 `mod_verto` 模块时的一个衍生模块。不过，它也是一个独立的模块，也能独立运行。比如，你可以：

```
bgapi originate rtc/blah &echo
```

但除此之外，它单独没什么实际用处。我们先来看代码。

首先是模块声明及全局变量定义。

```
37 SWITCH_MODULE_LOAD_FUNCTION(mod_rtc_load);  
38 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_rtc_shutdown);  
39 SWITCH_MODULE_DEFINITION(mod_rtc, mod_rtc_load, mod_rtc_shutdown, NULL);
```

```
...  
42 switch_endpoint_interface_t *rtc_endpoint_interface;
```

私有数据结构体。

```
59 typedef struct {  
60     switch_channel_t *channel;  
61     switch_core_session_t *session;  
62     switch_caller_profile_t *caller_profile;  
63     switch_media_handle_t *media_handle;  
64     switch_core_media_params_t mparams;  
65 } private_object_t;
```

模块的全局数据。

```
67 static struct {  
68     switch_memory_pool_t *pool;  
69     switch_mutex_t *mutex;  
70     int running;  
71 } mod_rtc_globals;
```

回调函数。基本只是打印 Log，最后 L133 销毁了 Media Handle。

```
82 static switch_status_t rtc_on_init(switch_core_session_t *session)  
83 {  
84     return SWITCH_STATUS_SUCCESS;  
85 }  
86  
87 static switch_status_t rtc_on_routing(switch_core_session_t *session)  
88 {  
89     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "%s RTC ROUTING\n",  
90                     switch_channel_get_name(switch_core_session_get_channel(session)));  
91  
92     return SWITCH_STATUS_SUCCESS;  
93 }  
94  
95  
96 static switch_status_t rtc_on_reset(switch_core_session_t *session)  
97 {  
98  
99     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "%s RTC RESET\n",  
100                    switch_channel_get_name(switch_core_session_get_channel(session)));
```

```
101
102
103     return SWITCH_STATUS_SUCCESS;
104 }
105
106 static switch_status_t rtc_on_hibernate(switch_core_session_t *session)
107 {
108     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "%s RTC HIBERNATE\n",
109                     switch_channel_get_name(switch_core_session_get_channel(session)));
110
111     return SWITCH_STATUS_SUCCESS;
112 }
113
114 static switch_status_t rtc_on_execute(switch_core_session_t *session)
115 {
116     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "%s RTC EXECUTE\n",
117                     switch_channel_get_name(switch_core_session_get_channel(session)));
118
119     return SWITCH_STATUS_SUCCESS;
120 }
121
122 static switch_status_t rtc_on_destroy(switch_core_session_t *session)
123 {
124     switch_channel_t *channel = switch_core_session_get_channel(session);
125
126     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "%s RTC DESTROY\n",
127                     switch_channel_get_name(channel));
128     switch_media_handle_destroy(session);
129
130     return SWITCH_STATUS_SUCCESS;
131 }
132
133 static switch_status_t rtc_on_hangup(switch_core_session_t *session)
134 {
135     return SWITCH_STATUS_SUCCESS;
136 }
137
138 static switch_status_t rtc_on_exchange_media(switch_core_session_t *session)
139 {
140     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "RTC EXCHANGE_MEDIA\n");
141     return SWITCH_STATUS_SUCCESS;
142 }
143
144 static switch_status_t rtc_on_soft_execute(switch_core_session_t *session)
```

```
151 {  
152     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "RTC SOFT_EXECUTE\n");  
153     return SWITCH_STATUS_SUCCESS;  
154 }
```

读视频，非常单，直接调用了核心里的函数。

```
157 static switch_status_t rtc_read_video_frame(switch_core_session_t *session, switch_frame_t **frame,  
↪ switch_io_flag_t flags, int stream_id)  
158 {  
159     return switch_core_media_read_frame(session, frame, flags, stream_id, SWITCH_MEDIA_TYPE_VIDEO);  
160 }
```

写视频。

```
162 static switch_status_t rtc_write_video_frame(switch_core_session_t *session, switch_frame_t *frame,  
↪ switch_io_flag_t flags, int stream_id)  
163 {  
164     private_object_t *tech_pvt = (private_object_t *) switch_core_session_get_private(session);  
165     switch_assert(tech_pvt != NULL);  
166  
167  
168     if (SWITCH_STATUS_SUCCESS == switch_core_media_write_frame(session, frame, flags, stream_id,  
↪ SWITCH_MEDIA_TYPE_VIDEO)) {  
169         return SWITCH_STATUS_SUCCESS;  
170     }  
171  
172     return SWITCH_STATUS_FALSE;  
173 }
```

读音频。

```
175 static switch_status_t rtc_read_frame(switch_core_session_t *session, switch_frame_t **frame,  
↪ switch_io_flag_t flags, int stream_id)  
176 {  
177     switch_status_t status = SWITCH_STATUS_FALSE;  
178  
179     status = switch_core_media_read_frame(session, frame, flags, stream_id, SWITCH_MEDIA_TYPE_AUDIO);  
180  
181     return status;  
182 }
```

写音频。

```
184 static switch_status_t rtc_write_frame(switch_core_session_t *session, switch_frame_t *frame,
↪ switch_io_flag_t flags, int stream_id)
185 {
186     switch_status_t status = SWITCH_STATUS_SUCCESS;
187
188     status = switch_core_media_write_frame(session, frame, flags, stream_id, SWITCH_MEDIA_TYPE_AUDIO);
189
190     return status;
191 }
```

接收消息，以便在挂机类似的时候解除 RTP 的阻塞状态（FreeSWITCH 内部，音频的读取一般只阻塞很短时间（如 20ms），但视频是一直阻塞的）。

```
193 static switch_status_t rtc_kill_channel(switch_core_session_t *session, int sig)
194 {
195     private_object_t *tech_pvt = switch_core_session_get_private(session);
196
197     if (!tech_pvt) {
198         return SWITCH_STATUS_FALSE;
199     }
200
201     switch (sig) {
202     case SWITCH_SIG_BREAK:
203         if (switch_core_media_ready(tech_pvt->session, SWITCH_MEDIA_TYPE_AUDIO)) {
204             switch_core_media_break(tech_pvt->session, SWITCH_MEDIA_TYPE_AUDIO);
205         }
206         if (switch_core_media_ready(tech_pvt->session, SWITCH_MEDIA_TYPE_VIDEO)) {
207             switch_core_media_break(tech_pvt->session, SWITCH_MEDIA_TYPE_VIDEO);
208         }
209         break;
210     case SWITCH_SIG_KILL:
211     default:
212
213         if (switch_core_media_ready(tech_pvt->session, SWITCH_MEDIA_TYPE_AUDIO)) {
214             switch_core_media_kill_socket(tech_pvt->session, SWITCH_MEDIA_TYPE_AUDIO);
215         }
216         if (switch_core_media_ready(tech_pvt->session, SWITCH_MEDIA_TYPE_VIDEO)) {
217             switch_core_media_kill_socket(tech_pvt->session, SWITCH_MEDIA_TYPE_VIDEO);
218         }
219         break;
220     }
221     return SWITCH_STATUS_SUCCESS;
222 }
```

```
225 static switch_status_t rtc_send_dtmf(switch_core_session_t *session, const switch_dtmf_t *dtmf)
226 {
227     return SWITCH_STATUS_SUCCESS;
228 }
229
230 static switch_status_t rtc_receive_message(switch_core_session_t *session, switch_core_session_message_t
↳ *msg)
231 {
232     switch_channel_t *channel = switch_core_session_get_channel(session);
233     private_object_t *tech_pvt = switch_core_session_get_private(session);
234     switch_status_t status = SWITCH_STATUS_SUCCESS;
235     const char *var;
236
237     if (switch_channel_down(channel) || !tech_pvt) {
238         status = SWITCH_STATUS_FALSE;
239         return SWITCH_STATUS_FALSE;
240     }
241
242     /* ones that do not need to lock rtp mutex */
243     switch (msg->message_id) {
244
245     case SWITCH_MESSAGE_INDICATE_CLEAR_PROGRESS:
246         break;
247     case SWITCH_MESSAGE_INDICATE_ANSWER:
248     case SWITCH_MESSAGE_INDICATE_PROGRESS:
249         {
250
251             if (((var = switch_channel_get_variable(channel, "rtp_secure_media"))) &&
252                 (switch_true(var) || switch_core_media_crypto_str2type(var) != CRYPTO_INVALID)) {
253                 switch_channel_set_flag(tech_pvt->channel, CF_SECURE);
254             }
255         }
256         break;
257
258     default:
259         break;
260     }
261
262     return status;
263 }
264
265
266
267 static switch_status_t rtc_receive_event(switch_core_session_t *session, switch_event_t *event)
268 {
269     return SWITCH_STATUS_SUCCESS;
```



```
270 }
271
272 static switch_jb_t *rtc_get_jb(switch_core_session_t *session, switch_media_type_t type)
273 {
274     private_object_t *tech_pvt = (private_object_t *) switch_core_session_get_private(session);
275
276     return switch_core_media_get_jb(tech_pvt->session, type);
277 }
```

定义 IO 例程及状态回调。

```
279 switch_io_routines_t rtc_io_routines = {
280     /*.outgoing_channel */ rtc_outgoing_channel,
281     /*.read_frame */ rtc_read_frame,
282     /*.write_frame */ rtc_write_frame,
283     /*.kill_channel */ rtc_kill_channel,
284     /*.send_dtmf */ rtc_send_dtmf,
285     /*.receive_message */ rtc_receive_message,
286     /*.receive_event */ rtc_receive_event,
287     /*.state_change */ NULL,
288     /*.read_video_frame */ rtc_read_video_frame,
289     /*.write_video_frame */ rtc_write_video_frame,
290     /*.read_text_frame */ NULL,
291     /*.write_text_frame */ NULL,
292     /*.state_run*/ NULL,
293     /*.get_jb*/ rtc_get_jb
294 };
295
296 switch_state_handler_table_t rtc_event_handlers = {
297     /*.on_init */ rtc_on_init,
298     /*.on_routing */ rtc_on_routing,
299     /*.on_execute */ rtc_on_execute,
300     /*.on_hangup */ rtc_on_hangup,
301     /*.on_exchange_media */ rtc_on_exchange_media,
302     /*.on_soft_execute */ rtc_on_soft_execute,
303     /*.on_consume_media */ NULL,
304     /*.on_hibernate */ rtc_on_hibernate,
305     /*.on_reset */ rtc_on_reset,
306     /*.on_park */ NULL,
307     /*.on_reporting */ NULL,
308     /*.on_destroy */ rtc_on_destroy
309 };
```

设置 Channel 的名字。

```
312 void rtc_set_name(private_object_t *tech_pvt, const char *channame)
313 {
314     char name[256];
315
316     switch_snprintf(name, sizeof(name), "rtc/%s", channame);
317     switch_channel_set_name(tech_pvt->channel, name);
318 }
```

将私有数据与 Session 关联。L333 ~ L335 设置该 Channel 支持 Jitter Buffer、RTP、IO 替代等能力。其中，IO 替代（[CC_IO_OVERRIDE](#)）说明本 Channel 可以与其它 Endpoint 配合使用，替代底层的 IO 能力。关于这部分，可以参考 [mod_verto](#) 中的代码。

L336 创建了一个 Media Handle，用于跟核心（[switch_core_media](#)）交互。

```
322 void rtc_attach_private(switch_core_session_t *session, private_object_t *tech_pvt, const char
↳ *channame)
323 {
324
325     switch_assert(session != NULL);
326     switch_assert(tech_pvt != NULL);
327
328     switch_core_session_add_stream(session, NULL);
329
330     tech_pvt->session = session;
331     tech_pvt->channel = switch_core_session_get_channel(session);
332     switch_core_media_check_dtmf_type(session);
333     switch_channel_set_cap(tech_pvt->channel, CC_JITTERBUFFER);
334     switch_channel_set_cap(tech_pvt->channel, CC_FS_RTP);
335     switch_channel_set_cap(tech_pvt->channel, CC_IO_OVERRIDE);
336     switch_media_handle_create(&tech_pvt->media_handle, session, &tech_pvt->mparams);
337     switch_core_session_set_private(session, tech_pvt);
338
339     if (channame) {
340         rtc_set_name(tech_pvt, channame);
341     }
342 }
```

申请私有数据。

```
345 private_object_t *rtc_new_pvt(switch_core_session_t *session)
346 {
347     private_object_t *tech_pvt = (private_object_t *) switch_core_session_alloc(session,
↳ sizeof(private_object_t));
```

```

348     return tech_pvt;
349 }

```

外呼回调函数。与 `rtp` Endpoint 类似。L365 创建 Session，L371 初始化私有数据结构，L392 将 Session 设为 `INIT` 状态。

```

351 static switch_call_cause_t rtc_outgoing_channel(switch_core_session_t *session, switch_event_t
↪ *var_event,
352                                             switch_caller_profile_t *outbound_profile,
↪ switch_core_session_t **new_session,
353                                             switch_memory_pool_t **pool, switch_originate_flag_t
↪ flags, switch_call_cause_t *cancel_cause)
354 {
355     switch_call_cause_t cause = SWITCH_CAUSE_DESTINATION_OUT_OF_ORDER;
356     switch_core_session_t *nsession = NULL;
357     switch_caller_profile_t *caller_profile = NULL;
358     private_object_t *tech_pvt = NULL;
359     switch_channel_t *nchannel;
360     const char *hval = NULL;
361
362     *new_session = NULL;
363
364
365     if (!(nsession = switch_core_session_request_uuid(rtc_endpoint_interface,
↪ SWITCH_CALL_DIRECTION_OUTBOUND,
366                                                     flags, pool, switch_event_get_header(var_event,
↪ "origination_uuid")))) {
367         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Error Creating Session\n");
368         goto error;
369     }
370
371     tech_pvt = rtc_new_pvt(nsession);
372
373     nchannel = switch_core_session_get_channel(nsession);
374
375     if (outbound_profile) {
376         caller_profile = switch_caller_profile_clone(nsession, outbound_profile);
377         switch_channel_set_caller_profile(nchannel, caller_profile);
378     }
379
380     if ((hval = switch_event_get_header(var_event, "media_webrtc")) && switch_true(hval)) {
381         switch_channel_set_variable(nchannel, "rtc_secure_media", SWITCH_RTP_CRYPT0_KEY_80);
382     }
383
384     if ((hval = switch_event_get_header(var_event, "rtc_secure_media")) {
385         switch_channel_set_variable(nchannel, "rtc_secure_media", hval);

```

```
386     }
387
388     rtc_attach_private(nsession, tech_pvt, NULL);
389
390
391     if (switch_channel_get_state(nchannel) == CS_NEW) {
392         switch_channel_set_state(nchannel, CS_INIT);
393     }
394
395     tech_pvt->caller_profile = caller_profile;
396     *new_session = nsession;
397     cause = SWITCH_CAUSE_SUCCESS;
398
399
400     if (session) {
401         switch_ivr_transfer_variable(session, nsession, "rtc_video_fmtp");
402     }
403
404     goto done;
405
406 error:
407
408     if (nsession) {
409         switch_core_session_destroy(&nsession);
410     }
411
412     if (pool) {
413         *pool = NULL;
414     }
415
416 done:
417
418     return cause;
419 }
```

恢复支持，FreeSWITCH 可以在崩溃重启或在 HA 环境中重建崩溃前的通话。

```
421 static int rtc_recover_callback(switch_core_session_t *session)
422 {
423     private_object_t *tech_pvt = rtc_new_pvt(session);
424     rtc_attach_private(session, tech_pvt, NULL);
425
426     return 1;
427 }
```

模块加载与卸载。

```
429 SWITCH_MODULE_LOAD_FUNCTION(mod_rtc_load)
430 {
431     memset(&mod_rtc_globals, 0, sizeof(mod_rtc_globals));
432     mod_rtc_globals.pool = pool;
433     switch_mutex_init(&mod_rtc_globals.mutex, SWITCH_MUTEX_NESTED, mod_rtc_globals.pool);
434
435     switch_mutex_lock(mod_rtc_globals.mutex);
436     mod_rtc_globals.running = 1;
437     switch_mutex_unlock(mod_rtc_globals.mutex);
438
439     /* connect my internal structure to the blank pointer passed to me */
440     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
441
442     rtc_endpoint_interface = switch_loadable_module_create_interface(*module_interface,
↵ SWITCH_ENDPOINT_INTERFACE);
443     rtc_endpoint_interface->interface_name = "rtc";
444     rtc_endpoint_interface->io_routines = &rtc_io_routines;
445     rtc_endpoint_interface->state_handler = &rtc_event_handlers;
446     rtc_endpoint_interface->recover_callback = rtc_recover_callback;
447
448     /* indicate that the module should continue to be loaded */
449     return SWITCH_STATUS_SUCCESS;
450 }
451
452 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_rtc_shutdown)
453 {
454     switch_mutex_lock(mod_rtc_globals.mutex);
455     if (mod_rtc_globals.running == 1) {
456         mod_rtc_globals.running = 0;
457     }
458     switch_mutex_unlock(mod_rtc_globals.mutex);
459
460     return SWITCH_STATUS_SUCCESS;
461 }
```

2.3.3 null Endpoint

null Endpoint，顾名思义，是一个空的 Endpoint，它只是一个“假”的 Channel，不支持任何实际的媒体收发，一切都是假的，但又是真的。该 Endpoint 主要用于测试。FreeSWITCH 中有一个测试代码框架，我们在以后的章节中再讲。

null Endpoint 跟上面讲的 `rtp` 和 `rtc` 类似，都有一些必须有的要素。具体代码我们就不详细解释了，感兴趣的同学可以对照前两节的内容自己看代码。null Endpoint 是在 `mod_loopback` 中实现的。

```
1211 static switch_endpoint_interface_t *null_endpoint_interface = NULL;
1212
1214 struct null_private_object {
1215     switch_core_session_t *session;
1216     switch_channel_t *channel;
1217     switch_codec_t read_codec;
1218     switch_codec_t write_codec;
1219     switch_timer_t timer;
1220     switch_caller_profile_t *caller_profile;
1221     switch_frame_t read_frame;
1222     int16_t *null_buf;
1223     int rate;
1224 };
1225
1226 typedef struct null_private_object null_private_t;
1239
1240 static switch_status_t null_tech_init(null_private_t *tech_pvt, switch_core_session_t *session)
1241 {
1242     const char *iananame = "L16";
1243     uint32_t interval = 20;
1244     switch_status_t status = SWITCH_STATUS_SUCCESS;
1245     switch_channel_t *channel = switch_core_session_get_channel(session);
1246     const switch_codec_implementation_t *read_impl;
1247
1248     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "%s setup codec %s/%d/
↪ %d\n",
1249                     switch_channel_get_name(channel), ianame, tech_pvt->rate, interval);
1250
1251     status = switch_core_codec_init(&tech_pvt->read_codec,
1252                                   ianame,
1253                                   NULL,
1254                                   NULL,
1255                                   tech_pvt->rate, interval, 1, SWITCH_CODEC_FLAG_ENCODE |
↪ SWITCH_CODEC_FLAG_DECODE, NULL, switch_core_session_get_pool(session));
1256
1257     if (status != SWITCH_STATUS_SUCCESS || !tech_pvt->read_codec.implementation || !
↪ switch_core_codec_ready(&tech_pvt->read_codec)) {
1258         goto end;
1259     }
1260
1261     status = switch_core_codec_init(&tech_pvt->write_codec,
1262                                   ianame,
1263                                   NULL,
1264                                   NULL,
1265                                   tech_pvt->rate, interval, 1, SWITCH_CODEC_FLAG_ENCODE |
↪ SWITCH_CODEC_FLAG_DECODE, NULL, switch_core_session_get_pool(session));
1266
1267
1268     if (status != SWITCH_STATUS_SUCCESS) {
```

```
1269         switch_core_codec_destroy(&tech_pvt->read_codec);
1270         goto end;
1271     }
1272
1273     switch_core_session_set_read_codec(session, &tech_pvt->read_codec);
1274     switch_core_session_set_write_codec(session, &tech_pvt->write_codec);
1275
1276     read_impl = tech_pvt->read_codec.implementation;
1277
1278     switch_core_timer_init(&tech_pvt->timer, "soft",
1279         read_impl->microseconds_per_packet / 1000, read_impl->samples_per_packet * 4,
1280     ↪ switch_core_session_get_pool(session));
1281
1282     switch_core_session_set_private(session, tech_pvt);
1283     tech_pvt->session = session;
1284     tech_pvt->channel = switch_core_session_get_channel(session);
1285     tech_pvt->null_buf = switch_core_session_alloc(session, sizeof(char) * read_impl-
1286     ↪ >samples_per_packet * sizeof(int16_t));
1287
1288     end:
1289
1290     return status;
1291 }
1292
1293 static switch_status_t null_channel_on_init(switch_core_session_t *session)
1294 {
1295     switch_channel_t *channel;
1296     null_private_t *tech_pvt = NULL;
1297
1298     tech_pvt = switch_core_session_get_private(session);
1299     switch_assert(tech_pvt != NULL);
1300
1301     channel = switch_core_session_get_channel(session);
1302     switch_assert(channel != NULL);
1303
1304     switch_channel_set_flag(channel, CF_ACCEPT_CNG);
1305     switch_channel_set_flag(channel, CF_AUDIO);
1306
1307     switch_channel_set_state(channel, CS_ROUTING);
1308
1309     return SWITCH_STATUS_SUCCESS;
1310 }
1311
1312 static switch_status_t null_channel_on_destroy(switch_core_session_t *session)
1313 {
1314     switch_channel_t *channel = NULL;
1315     null_private_t *tech_pvt = NULL;
```

```
1316     channel = switch_core_session_get_channel(session);
1317     switch_assert(channel != NULL);
1318
1319     tech_pvt = switch_core_session_get_private(session);
1320
1321     if (tech_pvt) {
1322         switch_core_timer_destroy(&tech_pvt->timer);
1323
1324         if (switch_core_codec_ready(&tech_pvt->read_codec)) {
1325             switch_core_codec_destroy(&tech_pvt->read_codec);
1326         }
1327
1328         if (switch_core_codec_ready(&tech_pvt->write_codec)) {
1329             switch_core_codec_destroy(&tech_pvt->write_codec);
1330         }
1331     }
1332
1333     return SWITCH_STATUS_SUCCESS;
1334 }
1335
1336
1337 static switch_status_t null_channel_kill_channel(switch_core_session_t *session, int sig)
1338 {
1339     switch_channel_t *channel = NULL;
1340     null_private_t *tech_pvt = NULL;
1341
1342     channel = switch_core_session_get_channel(session);
1343     switch_assert(channel != NULL);
1344
1345     tech_pvt = switch_core_session_get_private(session);
1346     switch_assert(tech_pvt != NULL);
1347
1348     switch (sig) {
1349     case SWITCH_SIG_BREAK:
1350         break;
1351     case SWITCH_SIG_KILL:
1352         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "CHANNEL
↳ SWITCH_SIG_KILL - hanging up\n");
1353         switch_channel_hangup(channel, SWITCH_CAUSE_NORMAL_CLEARING);
1354         break;
1355     default:
1356         break;
1357     }
1358
1359     return SWITCH_STATUS_SUCCESS;
1360 }
1361
1362 static switch_status_t null_channel_on_consume_media(switch_core_session_t *session)
1363 {
```



```
1364     switch_channel_t *channel = NULL;
1365     null_private_t *tech_pvt = NULL;
1366
1367     channel = switch_core_session_get_channel(session);
1368     assert(channel != NULL);
1369
1370     tech_pvt = switch_core_session_get_private(session);
1371     assert(tech_pvt != NULL);
1372
1373     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "CHANNEL CONSUME_MEDIA
↪ - answering\n");
1374
1375     switch_channel_mark_answered(channel);
1376
1377     return SWITCH_STATUS_SUCCESS;
1378 }
1379
1380 static switch_status_t null_channel_send_dtmf(switch_core_session_t *session, const switch_dtmf_t
↪ *dtmf)
1381 {
1382     null_private_t *tech_pvt = NULL;
1383
1384     tech_pvt = switch_core_session_get_private(session);
1385     switch_assert(tech_pvt != NULL);
1386
1387     return SWITCH_STATUS_SUCCESS;
1388 }
1389
1390 static switch_status_t null_channel_read_frame(switch_core_session_t *session, switch_frame_t
↪ **frame, switch_io_flag_t flags, int stream_id)
1391 {
1392     switch_channel_t *channel = NULL;
1393     null_private_t *tech_pvt = NULL;
1394     switch_status_t status = SWITCH_STATUS_FALSE;
1395
1396     channel = switch_core_session_get_channel(session);
1397     switch_assert(channel != NULL);
1398
1399     tech_pvt = switch_core_session_get_private(session);
1400     switch_assert(tech_pvt != NULL);
1401
1402     *frame = NULL;
1403
1404     if (!switch_channel_ready(channel)) {
1405         return SWITCH_STATUS_FALSE;
1406     }
1407
1408     switch_core_timer_next(&tech_pvt->timer);
1409 }
```

```
1410     if (tech_pvt->null_buf) {
1411         int samples;
1412         memset(&tech_pvt->read_frame, 0, sizeof(switch_frame_t));
1413         samples = tech_pvt->read_codec.implementation->samples_per_packet;
1414         tech_pvt->read_frame.codec = &tech_pvt->read_codec;
1415         tech_pvt->read_frame.datalen = samples * sizeof(int16_t);
1416         tech_pvt->read_frame.samples = samples;
1417         tech_pvt->read_frame.data = tech_pvt->null_buf;
1418         switch_generate_sln_silence((int16_t *)tech_pvt->read_frame.data, tech_pvt-
↳ >read_frame.samples, tech_pvt->read_codec.implementation->number_of_channels, 10000);
1419         *frame = &tech_pvt->read_frame;
1420     }
1421
1422     if (*frame) {
1423         status = SWITCH_STATUS_SUCCESS;
1424     } else {
1425         status = SWITCH_STATUS_FALSE;
1426     }
1427
1428     return status;
1429 }
1430
1431 static switch_status_t null_channel_write_frame(switch_core_session_t *session, switch_frame_t
↳ *frame, switch_io_flag_t flags, int stream_id)
1432 {
1433     switch_channel_t *channel = NULL;
1434     null_private_t *tech_pvt = NULL;
1435
1436     channel = switch_core_session_get_channel(session);
1437     switch_assert(channel != NULL);
1438
1439     tech_pvt = switch_core_session_get_private(session);
1440     switch_assert(tech_pvt != NULL);
1441
1442     switch_core_timer_sync(&tech_pvt->timer);
1443
1444     return SWITCH_STATUS_SUCCESS;
1445 }
1446
1447 static switch_status_t null_channel_receive_message(switch_core_session_t *session,
↳ switch_core_session_message_t *msg)
1448 {
1449     switch_channel_t *channel;
1450     null_private_t *tech_pvt;
1451
1452     channel = switch_core_session_get_channel(session);
1453     switch_assert(channel != NULL);
1454
1455     tech_pvt = switch_core_session_get_private(session);
```

```
1456     switch_assert(tech_pvt != NULL);
1457
1458     switch (msg->message_id) {
1459     case SWITCH_MESSAGE_INDICATE_ANSWER:
1460         switch_channel_mark_answered(channel);
1461         break;
1462     case SWITCH_MESSAGE_INDICATE_BRIDGE:
1463     case SWITCH_MESSAGE_INDICATE_UNBRIDGE:
1464     case SWITCH_MESSAGE_INDICATE_AUDIO_SYNC:
1465         switch_core_timer_sync(&tech_pvt->timer);
1466         break;
1467     default:
1468         break;
1469     }
1470
1471     return SWITCH_STATUS_SUCCESS;
1472 }
1473
1474 static switch_call_cause_t null_channel_outgoing_channel(switch_core_session_t *session,
↪ switch_event_t *var_event,
1475                 switch_caller_profile_t *outbound_profile,
1476                 switch_core_session_t **new_session, switch_memory_pool_t **pool,
↪ switch_originate_flag_t flags,
1477                 switch_call_cause_t *cancel_cause)
1478 {
1479     char name[128];
1480     switch_channel_t *ochannel = NULL;
1481
1482     if (session) {
1483         ochannel = switch_core_session_get_channel(session);
1484         switch_channel_clear_flag(ochannel, CF_PROXY_MEDIA);
1485         switch_channel_clear_flag(ochannel, CF_PROXY_MODE);
1486         switch_channel_pre_answer(ochannel);
1487     }
1488
1489     if ((*new_session = switch_core_session_request(null_endpoint_interface,
↪ SWITCH_CALL_DIRECTION_OUTBOUND, flags, pool)) != 0) {
1490         null_private_t *tech_pvt;
1491         switch_channel_t *channel;
1492         switch_caller_profile_t *caller_profile;
1493
1494         switch_core_session_add_stream(*new_session, NULL);
1495
1496         if ((tech_pvt = (null_private_t *) switch_core_session_alloc(*new_session,
↪ sizeof(null_private_t))) != 0) {
1497             const char *rate_ = switch_event_get_header(var_event, "rate");
1498             int rate = 0;
1499
1500             if (rate_) {
```

```

1501         rate = atoi(rate_);
1502     }
1503
1504     if (!(rate > 0 && rate % 8000 == 0)) {
1505         rate = 8000;
1506     }
1507
1508     tech_pvt->rate = rate;
1509
1510     channel = switch_core_session_get_channel(*new_session);
1511     switch_snprintf(name, sizeof(name), "null/%s", outbound_profile->destination_number);
1512     switch_channel_set_name(channel, name);
1513     if (null_tech_init(tech_pvt, *new_session) != SWITCH_STATUS_SUCCESS) {
1514         switch_core_session_destroy(new_session);
1515         return SWITCH_CAUSE_DESTINATION_OUT_OF_ORDER;
1516     }
1517 } else {
1518     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(*new_session), SWITCH_LOG_CRIT, "Hey where
↪ is my memory pool?\n");
1519     switch_core_session_destroy(new_session);
1520     return SWITCH_CAUSE_DESTINATION_OUT_OF_ORDER;
1521 }
1522
1523 if (outbound_profile) {
1524     caller_profile = switch_caller_profile_clone(*new_session, outbound_profile);
1525     caller_profile->source = switch_core_strdup(caller_profile->pool, modname);
1526
1527     switch_snprintf(name, sizeof(name), "null/%s", caller_profile->destination_number);
1528     switch_channel_set_name(channel, name);
1529     switch_channel_set_caller_profile(channel, caller_profile);
1530     tech_pvt->caller_profile = caller_profile;
1531 } else {
1532     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(*new_session), SWITCH_LOG_ERROR, "Doh! no
↪ caller profile\n");
1533     switch_core_session_destroy(new_session);
1534     return SWITCH_CAUSE_DESTINATION_OUT_OF_ORDER;
1535 }
1536
1537 switch_channel_set_state(channel, CS_INIT);
1538 switch_channel_set_flag(channel, CF_AUDIO);
1539 return SWITCH_CAUSE_SUCCESS;
1540 }
1541
1542 return SWITCH_CAUSE_DESTINATION_OUT_OF_ORDER;
1543 }
1544
1545 static switch_state_handler_table_t null_channel_event_handlers = {
1546     /*.on_init */ null_channel_on_init,
1547     /*.on_routing */ NULL,

```

```
1548     /*.on_execute */ NULL,
1549     /*.on_hangup */ NULL,
1550     /*.on_exchange_media */ NULL,
1551     /*.on_soft_execute */ NULL,
1552     /*.on_consume_media */ null_channel_on_consume_media,
1553     /*.on_hibernate */ NULL,
1554     /*.on_reset */ NULL,
1555     /*.on_park */ NULL,
1556     /*.on_reporting */ NULL,
1557     /*.on_destroy */ null_channel_on_destroy
1558 };
1559
1560 static switch_io_routines_t null_channel_io_routines = {
1561     /*.outgoing_channel */ null_channel_outgoing_channel,
1562     /*.read_frame */ null_channel_read_frame,
1563     /*.write_frame */ null_channel_write_frame,
1564     /*.kill_channel */ null_channel_kill_channel,
1565     /*.send_dtmf */ null_channel_send_dtmf,
1566     /*.receive_message */ null_channel_receive_message
1567 };
1568
1569
1570 SWITCH_MODULE_LOAD_FUNCTION(mod_loopback_load)
1571 {
1572     ...
1580     memset(&globals, 0, sizeof(globals));
1581
1582     /* connect my internal structure to the blank pointer passed to me */
1583     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
1584     ...
1589     null_endpoint_interface = switch_loadable_module_create_interface(*module_interface,
1590     ↪ SWITCH_ENDPOINT_INTERFACE);
1591     null_endpoint_interface->interface_name = "null";
1592     null_endpoint_interface->io_routines = &null_channel_io_routines;
1593     null_endpoint_interface->state_handler = &null_channel_event_handlers;
1594
1597     return SWITCH_STATUS_SUCCESS;
1598 }
```

第三章 FreeSWITCH 二次开发

通过本书前面章节的学习，我们熟悉了 FreeSWITCH 的使用方法，熟悉了 FreeSWITCH 的源代码。现在，该轮到我们自己写代码的时候了。当然，在前面我们也写过代码，但都是使用嵌入式脚本及 ESL 接口等在 FreeSWITCH 外部开发的。在本章，我们将从汇报 Bug 开始讲起，以笔者在学习和使用过程中的实际例子为例，讲一下如何修改 FreeSWITCH 的源代码，如何将自己的修改提交到官方的代码库里，为开源项目做贡献。最后，带领大家从头开始开发一个新的模块。

3.1 给 FreeSWITCH 汇报 Bug 和打补丁

大多数商业的系统都是闭源的，有时候出了问题甚至很难跟踪调试，更不容易发现 Bug 的具体位置。而相对来讲，使用开源项目的好处就是，我们可以参照源代码比较容易的找到 Bug。笔者在学习和使用 FreeSWITCH 的过程中，在官方的 Bug 跟踪工具中汇报了很多的 Bug¹，大多数都得到了很及时的修复。接下来，我们就一起看几个真实的例子（为节省篇幅，我们本章中的部分代码使用 `git diff` 格式，其中，行前的“-”表示删除的行，“+”代表添加的行）。

3.1.1 汇报 Bug 时注意的问题

笔者遇到的好多说汉语的朋友，他们在汇报 Bug 时总担心自己英文不好，怕描述不清楚，喜欢找笔者代劳。诚然，对于参与国际性的项目，熟练的英文读写功底还是非常必要的。但是，英语也绝不是一个最重要的障碍，很多情况下，你只要逻辑清晰，用比较简单的语言把问题描述清楚即可。如，你使用的操作系统，编译环境，问题出现的场景，如何再现（很重要）等用比较简单的英语描述清楚了，并附上必要的日志或消息跟踪即可。

之所以说英语绝对不是一个重要障碍，是因为笔者发现很说汉语的朋友在 QQ 群中或邮件列表中其实用汉语也无法把问题描述清楚。比方说，有的朋友问道：“我装了 FreeSWITCH，怎么打不了电话？”，或者“电话能打通，可是却没有声音，这是怎么回事？”。

很容易看到，上述问题是无法回答的，因为缺少太多必要的信息。如，FreeSWITCH 支持很多平台，至少说明一下你是到底在 Linux（CentOS? Debian? Ubuntu?）还是 Windows（XP??

¹当然，不要因此误认为开源的系统 Bug 多，其实闭源的通常 Bug 更多，只是你看不到而已。因为开源的项目，全世界的人都在各种应用场景下使用，所有人都能看到源代码，因而更容易发现 Bug，而且，通常开源社区也会更及时地修复。

Win7? 2012?) 上安装的、FreeSWITCH 是什么版本、从安装包装的还是源代码装的等等最基本的信息。而且, 还有出现问题的场景。如: “我在 Windows 上通过安装包安装了最新版的 FreeSWITCH (版本号是 xxxx), 注册了两个电话 1000 和 1001, 从 1000 打 1001 不通, 日志中显示 `INCOMPATIBLE_DESTINATION`, 请问这可能是是什么原因?” 这就是一个比较好的问题, 因为这样信息量比较多, 别人也愿意帮助你 (提问前面两个不好的问题的人通常比较懒, 别人也懒得帮助)。而且很容易翻译成简单的英语, 如:

```
Download from http://files.freeswitch.org/....
Version: xxxxx
Installed on Win7
Call from 1000 to 1001, failed. INCOMPATIBLE_DESTINATION
Log attached, Thanks.
```

然后, 在邮件列表中提问, 或者到 Jira 上汇报 Bug。虽然上面的英语不是很地道, 但也足以让人能看清楚了。

总之, 提问问题或汇报 Bug 并不一定要很高的技能, 或者跟踪要源代码, 把你的问题描述清楚是最重要的。

关于这个问题, 可以参考笔者汇报的 Bug: <http://jira.freeswitch.org/browse/FS-993>。当时笔者在测试 `mod_skypiax` 时 (`mod_skypopen`) 的前身发现有电话挂掉后还有僵尸 Channel 遗留的问题, 通过邮件列表咨询、汇报 Bug、并采取回帖、附件等多种方式提供必要的信息 (有时候问题描述不一定一次能完全提供, 需要多次互动, 但一定要积极, 不要让别人觉得你太懒)。最终问题解决了。

3.1.2 修复内存泄露问题

在笔者早期使用 FreeSWITCH 的过程中, 发现 FreeSWITCH 的内存一直增长。查找了很久没有找到问题原因。而在同时, 也没有发现其它人有这个问题。后来, 笔者考虑到自己使用了 `mod_erlang_event` 模块, 而使用该模块的人比较少。因此就研究了一个该模块的源代码 (当时还不会使用 `valgrind` 工具查找内存泄露问题)。终于发现一处申请了内存没有释放。后来, 在源代码中, 增加了如下一行, 在 `event` 指针用完之后将内存释放问题就解决了。

```
switch_event_destroy(&event);
```

在运行了几天之后, 确认没有问题了, 笔者提交了一个 Jira 描述了发现的问题。然后, 使用 `git diff > erlang_leak.diff` 命令产生了下面的补丁文件, 并将补丁文件附加上去。

```
diff --git a/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
    b/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
index 9a09e80..cd58d95 100644
--- a/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
+++ b/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
@@ -650,6 +650,9 @@ static switch_status_t check_attached_sessions(listener_t *listener)
     }

    switch_thread_rwlock_unlock(listener->session_rwlock);
+
+    switch_event_destroy(&event);
+
    if (prefs.done) {
        return SWITCH_STATUS_FALSE; /* we're shutting down */
    } else {
```

最后，该模块的作者将补丁合并到 FreeSWITCH 代码库中进去了。相关的 Jira 参见：<http://jira.freeswitch.org/browse/FS-3488>。

3.1.3 给中文模块打补丁

由于我们需要在 FreeSWITCH 中支持中文语音，因此我们用到了 `mod_say_zh` 模块。而在使用的过程中我们的团队成员发现了如下的错误：

```
[ERR] mod_say_zh.c:513: Unknown Say type=[18]
```

笔者鼓励同事去源代码里找一找出错的原因，很快就找到它是在第 513 行的一条日志输出语句中输出的。从源代码可以看到，很明显错误的原因是在 `case` 语句中没有对应的分支，进而转到 `default` 语句造成的。相关的部分代码片断如下：

```
490 case SST_NUMBER:
...
511 case SST_CURRENCY:
...
512 default:
513     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR,
        "Unknown Say type=[%d]\n", say_args->type);
```

通过在全部源代码中搜索离它最近的第 511 行的常量定义“`SST_CURRENCY`”，我们找到了对应错误日志中的 18 的是常量 `SST_SHORT_DATE_TIME` (`switch_types.h:418`)。更深入的研究发现其实只需对它做与 `SST_CURRENT_DATE_TIME` 同样的处理即可。因而，我们产生了一个补丁。由于当时笔者已经具有代码库的提交权限，因此就直接将代码提交到了代码库中。而没有提交 Jira。

如果读者有源代码的话，可以用以下命令查看相关的补丁（命令和输出结果如下，为节省篇幅，输出结果有删节）：

```
$ git show f255f6
commit f255f65a82e2cfe1aed1f44aa2459e5faaed150e
Author: Seven Du <dujinfang@gmail.com>
Date: Thu Aug 1 09:50:51 2013 +0800

    add SHORT_DATE_TIME support

diff --git a/src/mod/say/mod_say_zh/mod_say_zh.c ...
      case SST_CURRENT_DATE:
      case SST_CURRENT_TIME:
      case SST_CURRENT_DATE_TIME:
+     case SST_SHORT_DATE_TIME:
          say_cb = zh_say_time;
```

通过这次修复，其它再使用该模块的朋友也不会遇到这个错误了。我们也为我们自己能为 FreeSWITCH 做贡献而感到自豪。

3.1.4 给 FreeSWITCH 核心打补丁

笔者在写本书的时候，阅读了大量的源代码，偶然发现其中对于多个 Channel 进行混音的源代码中可能有问题。该段代码不长，因此我们把它全部贴在后面。

对多个 Channel 进行混音的函数是在第 274 行定义的，所有的音频数据存放到 `data` 指针中。其中，音频数据是 16 位的整形数据 (`int16_t`)；`samples` 代表采样率，如 8000；`channels` 代表有几个声道，如果在双声道中，它的值就是 2。

第 276 行，定义了一个 `buf` 指针，备用；第 277 行，算出需要的缓冲区的字节长度；第 279 行，定义 32 位的整数变量 (`uint32_t`)，以避免在计算中 16 位的整数溢出。

```
274 SWITCH_DECLARE(void) switch_mux_channels(int16_t *data,
      switch_size_t samples, uint32_t channels)
275 {
276     int16_t *buf;
277     switch_size_t len = samples * sizeof(int16_t);
```

```
278     switch_size_t i = 0;
279     uint32_t j = 0, k = 0;
```

第 281 行，申请一个足够大的缓冲区，让 `buf` 指针指向它。然后使用一个双重 `for` 循环将两个声道对应位置的数据相加（第 285 行），并于第 286 行使用 `switch_normalize_to_16bit`² 将 32 位的整数标准化成 16 位的整数，在第 287 行将数据写入缓冲区。

```
281     switch_zmalloc(buf, len);
282
283     for (i = 0; i < samples; i++) {
284         for (j = 0; j < channels; j++) {
285             int32_t z = buf[i] + data[k++];
286             switch_normalize_to_16bit(z);
287             buf[i] = (int16_t) z;
288         }
289     }
```

全部处理完成后，将数据从缓冲区中再使用 `memcpy` 内存拷贝函数将数据复制到原来的数据区域（第 291 行），并释放缓冲区（第 292）行。

```
291     memcpy(data, buf, len);
292     free(buf);
293
294 }
```

具体的算法应该很直观。一个双声道混音的示意图如图 22-1 所示（其中“左”，“右”分代表左、右声道）。

²看起来像一个函数，实际上是一个在 `switch_utils.h:236` 定义的一个宏。

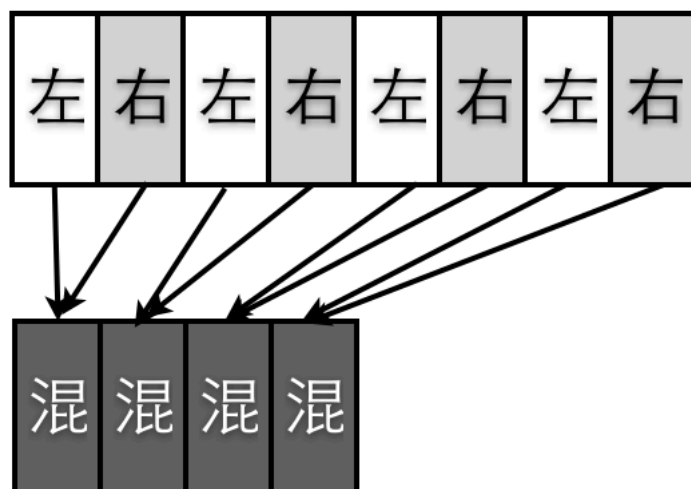


图 3.1: 图 22-1 双声道混音

在笔者刚刚看到该代码时，也是花了一些时间明白了混音的算法。不过，笔者立即想到。既然是混音，那么如果将两个声道混合成一个声道，理论上只占用一半的内存缓冲区，所以，原来的缓冲区如果可以重复利用的话，能不能不申请新的缓冲区呢？通过一番实验，笔者实现了如下改进的算法。

在新的算法中，笔者还是使用一个双重 `for` 循环遍历所有数据，但是，这里，没有申请新的内存缓冲区，而是在第 280 行使用了一个新的中间变量 `z`。将对应的音频数据相加后放到 `z` 中（第 280 行），然后将数据标准化（第 283 行），并直接将最后结果写入原来的 `data` 指针所传入的数据缓冲区即可（第 284 行。因为原来的数据我们已经取出来计算过了，不再需要了）。

```

274 SWITCH_DECLARE(void) switch_mux_channels(int16_t *data,
      switch_size_t samples, uint32_t channels)
275 {
276     switch_size_t i = 0;
277     uint32_t j = 0;
278
279     for (i = 0; i < samples; i++) {
280         int32_t z = 0;
281         for (j = 0; j < channels; j++) {
282             z += data[i * channels + j];
283             switch_normalize_to_16bit(z);
284             data[i] = (int16_t) z;
285         }
286     }
287 }

```

通过使用新的算法，至少节省了一半的内存，而且避免了重复的内存申请造成的内存碎片，节省

了一个内存拷贝操作，以及节省了 6 行代码等。笔者并没有实际测试以比较最终归的效果，不过理论上是这样的，并且在实现了之后也感觉比较有成就感。

在经过反复测试确认没有问题之后，笔者将该改进提到了官方的 Jira 上，见：<http://jira.freeswitch.org/browse/FS-4622>。虽然笔者当时具有直接向代码库中提交代码的权限，但是对于核心代码的改动，毕竟问题比较重大。把它记录到 Jira 上，以后万一出了问题也容易跟踪。而且，Jira 会给原代码的作者一个比较友好的通知，让作者决定是否将代码合并进去，也是一种比较友好的协作方式。

3.1.5 高手也会犯错误

在上一章，我们讲到过，由于 `SWITCH_STATUS_SUCCESS` 的常量值为 0，因此，在使用时需要严格的进行“`==`”判断，否则就容易出现错误。在 FreeSWITCH 代码的历史上，就曾经出现过这样的错误，其中一次是笔者发现的，记录在该 Jira 报告中：<http://jira.freeswitch.org/browse/FS-5351>。

该 Bug 已经修复，不过，感兴趣的读者仍可以使用 `git show c4e7c30` 命令查看当时是如何修复的，以避免自己在后发生同样的错误。如下，可以看出，在“+”一行，增加了“`SWITCH_STATUS_SUCCESS ==`”判断。

```
$ git show c4e7c30
...
diff --git a/src/mod/endpoints/mod_sofia/mod_sofia.c ...
@@ -968,7 +968,7 @@ static switch_status_t sofia_write_video_frame(switch_core_session_t *session, s
     return SWITCH_STATUS_SUCCESS;
}

- if (switch_core_media_write_frame(session, frame, flags, stream_id, SWITCH_MEDIA_TYPE_VIDEO)) {
+ if (SWITCH_STATUS_SUCCESS == switch_core_media_write_frame(
+     session, frame, flags, stream_id, SWITCH_MEDIA_TYPE_VIDEO)) {
     return SWITCH_STATUS_SUCCESS;
}
```

为节省篇幅，上述命令的输出进行了删减，读者可以自己试一下，或者直接访问 Web 版的界面查看：<http://fisheye.freeswitch.org/changelog/freeswitch.git/?cs=c4e7c30>。

当然，我们写这个例子的目的是，即使 FreeSWITCH 的作者，也可能会犯错误。所以，在汇报 Bug 时不要有过多顾虑。

3.1.6 汇报严重的问题

在上一节，我们鼓励大家要勇敢汇报 Bug。许多朋友可能在遇到问题时不确定哪个问题是个 Bug，便在邮件列表中询问，一来二去，耽误了好长时间。FreeSWITCH 的作者 Anthony Minessale

经常在邮件列表中说：“在 Jira 上告诉一个人这不是一个 Bug 比在邮件列表中跟踪这些问题要容易得多”。意思是说，如果感觉类似 Bug 的问题尽管往 Jira 上提，而尽量不要使用邮件列表。因为 Jira 是一个 Bug 跟踪系统，即使你的判断不对，别人也会直接在 Jira 系统上告诉你，流程很清晰。而如果使用邮件列表的话，群里很多的人每天都会收到几百封的邮件，很容易你的信息就被埋没了。

当然，无论如何，如果遇到系统崩溃，那一定是个重大问题。FreeSWITCH 的目标是让它不崩溃，因此，所有的崩溃都应该向 Jira 汇报。

笔者就曾经汇报过一个在使用会议系统时系统崩溃的案例。当发现系统崩溃后，笔者根据崩溃后产生的 `core Dump` 文件（内核转储文件），发现了一些导致问题的可能的原因。下面是当时内核文件的反向跟踪信息（Back Trace，在 GDB 中使用 `bt` 命令得到）：

```
Thread 38 (core thread 37):
#0  switch_core_session_get_channel (session=0x0) at switch_core_session.c:1190
#1  0x00000001033d5e09 in conference_video_thread_run (thread=0x0, obj=0x7fd51c8a1f68) at mod_conference.c:
↳ 1436
#2  0x00007fff887bc8bf in _pthread_start ()
#3  0x00007fff887bfb75 in thread_start ()
```

可以看出，在第“#0”个函数调用中，“`session=0x0`”，即 `session` 指针为空指针，导致后续的操作出错。而该函数是在第“#1”个函数调用的时候出现的，它发生在 `mod_conference:1436`，因此，我们很容易在源代码目录中找到它。如下：

```
1436 switch_channel_t *ichannel =
    switch_core_session_get_channel(imember->session);
```

分析问题的原因，在于并不是所有的会议成员（`imember`）都会对应一个 Channel 的（如录、放音等虚拟成员）。因此笔者简单生成了一个补丁，提到 Jira 上。后来 Anthony 在合并的时候又修改了一些内容，因此产生了如下的补丁：

```
for (imember = conference->members; imember; imember = imember->next) {
-     switch_channel_t *ichannel =
-         switch_core_session_get_channel(imember->session);
+     switch_core_session_t *isession = imember->session;
+     switch_channel_t *ichannel;
+
+     if (!isession || !switch_core_session_read_lock(ises
+         continue;
+     }
+ }
```

```
+         ichannel = switch_core_session_get_channel(imember->ses
...
+         switch_core_session_rwlock(isession);
```

经过测试发现，该补丁虽然解决了崩溃问题，但可能会造成死锁。再次反馈后，发现是又忘了判断 `SWITCH_STATUS_SUCCESS` 了。通过如下补丁把问题修复。

```
-     if (!isession || !switch_core_session_read_lock(isession)) {
+     if (!isession || switch_core_session_read_lock(isession) != SWITCH_STATUS_SUCCESS) {
```

本案例记录在<http://jira.freeswitch.org/browse/FS-4318>。感兴趣的同学可以深入研究一下。

3.1.7 给 Sofia-SIP 打补丁

前面讲的一些例子都是在 FreeSWITCH 代码中打补丁。我们再来看一个更深层次的例子。

为了避免重复发明轮子，FreeSWITCH 大量使用了一些第三方的代码库。同时为了编译方便，以及减少由于不同的不同版本的引起的可能的混乱，FreeSWITCH 尽量将一些协议兼容的第三方代码库也放到 FreeSWITCH 源代码库中。这些库一般放到 FreeSWITCH 源代码的 `libs` 目录中。Sofia-SIP 即是其中之一。

笔者以前做过一个项目，需要在 FreeSWITCH 中增加 MSRP³协议的支持。而经过跟踪发现，Sofia-SIP 底层就不支持该协议，因而如果使用该协议，就需要修改 Sofia-SIP 底层的代码。但 Sofia-SIP 库近几年的维护几乎停滞。经过与 FreeSWITCH 官方的沟通，他们说我们可以先把代码提交到 FreeSWITCH 中，等到需要的时候再提交到上游的 Sofia-SIP 库中。

后来，笔者便在 Jira 上提交了一个补丁<http://jira.freeswitch.org/browse/FS-3748>。下面我们简单看一下补丁的内容。

首先，在 Sofia-SIP 库中提交补丁时，需要更新 “`.update`” 文件。该文件的内容不重要，一般就写上当前的日期。通过修改该文件，当再次执行 “`make mod_sofia`” 进行编译时，它便会感知到 Sofia-SIP 库的变化，进而会重新编译 Sofia-SIP 库⁴。

```
diff --git a/libs/sofia-sip/.update b/libs/sofia-sip/.update
@@ -1,1 @@
```

³MSRP (Message Session Relay Protocol) 称为中继会话中继协定。可用于基于 Session 的即时消息传递或文件传递等。参见：<http://tools.ietf.org/html/rfc4975>。

⁴当然，实际的 Makefile 编译机制应该能自动探测到所有依赖的文件的变化，而不需要这种手工修改一个自定义的文件。但由于 Sofia-SIP 是第三方的库，为了避免过度耦合，因而采用了这种比较简单的方法。FreeSWITCH 中使用的其它的第三方库也有类似的机制。

-Tue Nov 22 18:16:53 CST 2011
+Tue Dec 6 18:12:20 CST 2011

实际的支持代码需要在多个文件中添加，如，首先在 `sdp_parse.c` 中，让它在解析的时候认识 SDP 中 MSRP 相关的内容（否则协议栈会拒绝）：

```
diff --git a/libs/sofia-sip/libsofia-sip-ua/sdp/sdp_parse.c ...

+ else if (su_casematch(s, "TCP/MSRP"))
+   m->m_proto = sdp_proto_msrp, m->m_proto_name = "TCP/MSRP";
+ else if (su_casematch(s, "TCP/TLS/MSRP"))
+   m->m_proto = sdp_proto_msrps, m->m_proto_name = "TCP/TLS/MSRP";
+   else if (su_casematch(s, "UDP"))
+     m->m_proto = sdp_proto_udp, m->m_proto_name = "UDP";
+   else if (su_casematch(s, "TCP"))
```

在生成 SDP 的时候也要加入 MSRP 支持：

```
diff --git a/libs/sofia-sip/libsofia-sip-ua/sdp/sdp_print.c ...

case sdp_proto_rtp: proto = "RTP/AVP"; break;
case sdp_proto_srtp: proto = "RTP/SAVP"; break;
case sdp_proto_udptl: proto = "udptl"; break;
+ case sdp_proto_msrp: proto = "TCP/MSRP"; break;
+ case sdp_proto_msrps: proto = "TCP/TLS/MSRP"; break;
```

最后，我们也把 MSRP 协议相关的常量加到 `sdp_proto_e` 枚举类型中。这样，底层的协议栈就能适当的解析出 MSRP 协议的相关内容，剩下的，我们只需要在上层的 `mod_sofia` 模块中增加相关的支持代码就行了。

```
diff --git a/libs/sofia-sip/libsofia-sip-ua/sdp/sofia-sip/sdp.h

@@ -243,6 +243,8 @@ typedef enum
    sdp_proto_rtp = 256,           /**< RTP/AVP */
    sdp_proto_srtp = 257,         /**< RTP/SAVP */
    sdp_proto_udptl = 258,        /**< UDPTL. @NEW_1_12_4. */
+   sdp_proto_msrp = 259,         /**< TCP/MSRP @NEW_MSRP*/
+   sdp_proto_msrps = 260,        /**< TCP/TLS/MSRP @NEW_MSRP*/
    sdp_proto_tls = 511,          /**< TLS over TCP */
    sdp_proto_any = 512,          /**< * wildcard */
} sdp_proto_e;
```

进行上述修改后, Sofia-SIP 库具备支持 MSRP 协议的能力了。当然, 具体的 MSRP 协议支持和处理还需要上层代码 (如在 `mod_sofia` 中增加相应逻辑) 的支持, 在此, 我们就不多讲了。在本例中, 我们介绍了底层的 Sofia-SIP 库的修改方法以及注意事项。补丁内容的本身并不重要, 重要的是了解这里的方法和流程, 以便在以后遇到类似问题时进行更深入的研究。在后面, FreeSWITCH 开发者还修复了一些 Bug 并增加了 SIP over WebSocket 支持, 以支持 WebRTC。有兴趣的读者也可以看一下这部分的更新历史。

3.1.8 给现有 App 增加新功能

在笔者的某一个咨询项目中, 有个客户提到, 它相与现有的 WebServer 集成, 但又不想使用比方说 ESL 等比较复杂的解决方案。问有没有更好的解决方案。笔者推荐他可以直接在 Dialplan 或 Lua 脚本中调用 `curl` App 跟远程的 HTTP 服务器交互。`curl` App 是在 `mod_curl` 中实现的, 在 Dialplan 中的使用方法如下:

```
<action application="curl" data="http://..." />
```

最初使用起来效果不错, 直到某一天遇到个问题——有些 `curl` 调用由于服务器卡死导致 Channel 老是卡死在那里, 挂掉以后还有残留的僵尸数据。出现该问题的原因是 `curl` 调用远程服务器一直没有返回, 因而进程阻塞。经过对源代码进行研究, 发现 `mod_curl` 是使用 `libcurl` 实现的, 但并没有使用超时机制, 所以导致了上述问题。

使用开源项目的好处就是我们不仅能修复 Bug, 还能随时添加新功能。经过, 查阅 `libcurl` 的文档, 我们发现可以通过 `CURLOPT_CONNECTTIMEOUT` 和 `CURLOPT_TIMEOUT` 选项控制请求超时。其中, 前者是连接超时, 即多长时间连接不到服务器即超时; 后者是执行超时, 即在长时间服务器不返回结果即超时。

我们发现在 `mod_curl` 中连接远程服务器并获取文件的函数是在 `do_lookup_url` 函数中实现的, 因此我们很快实现了如下的补丁。

有时候, 做好事是不需要留名的, 但有时候在开源项目中留下自己的名字也感觉挺不错的, 因而, 笔者在该模块前两位贡献者之后留下了自己的名字:

```
* Rupa Schomaker <rupa@rupa.com>
* Yossi Neiman <mishehu@freeswitch.org>
+ * Seven Du <dujinfang@gmail.com>
```

为了存放我们的超时参数, 我们定义了一个结构体, 并使用 `typedef` 定义了一个新的类型 `curl_options_t`:

```
+struct curl_options_obj {
+    long connect_timeout;
+    long timeout;
+};
+typedef struct curl_options_obj curl_options_t;
```

然后，我们在原来的 `do_lookup_url` 函数的基础上增加了一个 `curl_options_t` 指针类型的参数 `options`：

```
-static http_data_t *do_lookup_url(...,
+static http_data_t *do_lookup_url(..., curl_options_t *options)
```

然后增加如下补丁，判断如果 `options` 参数存在的话（保证向后兼容，如果没有提供超时参数的话，继续保持原来的行为），则调用 `libcurl` 的 `switch_curl_easy_setopt` 函数设置相应的超时参数。

```
+    if (options) {
+        if (options->connect_timeout) {
+            switch_curl_easy_setopt(curl_handle,
+                CURLOPT_CONNECTTIMEOUT, options->connect_timeout);
+        }
+
+        if (options->timeout) {
+            switch_curl_easy_setopt(curl_handle,
+                CURLOPT_TIMEOUT, options->timeout);
+        }
+    }
```

至此，`do_lookup_url` 函数就已经具备超时功能了。但为了使用它，在 `curl` App 对应的函数中我们需要先初始化一个 `options` 结构体，并从当前的通道变量中收集相关的超时参数：

```
+    curl_options_t options = { 0 };
+    const char *curl_timeout;

+    curl_timeout = switch_channel_get_variable(channel, "curl_connect_timeout");
+    if (curl_timeout) options.connect_timeout = atoi(curl_timeout);
+
+    curl_timeout = switch_channel_get_variable(channel, "curl_timeout");
+    if (curl_timeout) options.timeout = atoi(curl_timeout);
```

初始化完了 `options` 参数，我们就可以在原来调用的位置把该参数加上了：

```
- http_data = do_lookup_url(pool, url, method, postdata, content_type);  
+ http_data = do_lookup_url(pool, url, method, postdata, content_type, &options);
```

至此，我们增加的功能就应该完成了。不过，后来我们在编译时发现还有一个错误。由于该模块还同时实现了一个 `curl` API 命令，它也调用了 `do_lookup_url` 函数。而由于我们修改了 `do_lookup_url` 函数的定义，导致无法编译。我们暂时不准备也为该 API 命令增加该功能（我们没有用到它），因此，为了简单起见，我们仅仅给该调用的地方增加了一个空指针作为参数。

```
- http_data = do_lookup_url(pool, url, method, postdata, content_type);  
+ http_data = do_lookup_url(pool, url, method, postdata, content_type, NULL);
```

编译顺利通过。通过在 Dialplan 中增加如下的设置，我们的问题也顺利解决了。

```
<action application="set" data="curl_connect_timeout=3000"/>  
<action application="set" data="curl_timeout=5000"/>  
<action application="curl" data="http://...">
```

综上，该补丁的实现思路典型的遵循 FreeSWITCH 的架构和设计思想——通过通道变量改变 App 的行为。所以，我们增加了两个通道变量（`curl_connect_timeout` 和 `curl_timeout`），并在 `curl` App 执行过程中根据这两个变量的值决定是否启动超时机制，以及控制合理的超时时间。

该补丁的提交哈希是 `d8a02dc`，读者可以通过 “`git show d8a02dc`” 命令查看。

3.1.9 给 FreeSWITCH 增加一个新的 Interface

故事要从若干年前说起。当年，笔者学习 FreeSWITCH 时间不长。在测试中文模块（`mod_say_zh`）时，发现它说出来的中文不符合中文用户的习惯。如，在用英语读美元时，`$10.20` 的习惯读法是 “10 dollar 20 cents”，而在中文模块中，就顺便读成了 “十元二十分”，显然不符合中文习惯。

当时，笔者就进行了一些改进，并提交了一个补丁，见 <http://jira.freeswitch.org/browse/FS-2809>。当时笔者做得比较激进，连厘都写上去了，如 `10.1234` 元将读成 “十元一角二分三厘四”。当然，当时只是为了好玩，因为我相信当时 FreeSWITCH 圈里的人，应该没有人能比笔者更懂中文了。

不过，在补丁提交了之后，原来模块的作者（一个外国人）却说，笔者做的修改太 “中国” 化了，如果那样改了，势必不符合全世界其它地区的习惯。在此之后笔者才意识到，原来中文是全世界的，

而我确实是见识短浅。比方说，如果在美国，即使使用中文读，也确实应该是“十元二十分”啊！也许正是从那以后，笔者考虑问题都会把眼光放远一些了。

但无论如何，中文，一定要支持中国的中文才叫中文。所以笔者与原作者探讨，是否增加一个通道变量检查之类的（类似于 22.1.8 节我们提到的用通道变量控制相关行为），让用户可以酌情选择？不过一直没有得到回应。后来，Mike Jerris（FreeSWITCH 三剑客之一）提议，可以做一个新的 Interface 时，我才恍然大悟——是啊，怎么没想到这一点？！

后来，笔者就修改了补丁，增加了一个新的 Say Interface——“zh_CN”。

其实，增加一个 Say Interface 很简单，只需要在 `mod_say_zh` 中（请注意，我们是在该模块中新增加了一个 Interface，而没有增加新的模块）增加如下的接口定义：

```
+    say_zh_CN_interface = switch_loadable_module_create_interface(  
        *module_interface, SWITCH_SAY_INTERFACE);  
+    say_zh_CN_interface->interface_name = "zh_CN";  
+    say_zh_CN_interface->say_function = zh_CN_say;
```

然后，实现 `zh_CN_say` 回调函数，该函数基本上与原来的 `zh_say` 函数一样（还是调用跟以前一样的函数），只是在读货币（`SST_CURRENCY`）的时候，使用了笔者专门实现的只针对中国的 `zh_CN_say_money` 函数。

```
+static switch_status_t zh_CN_say(...  
+{  
  
+    case SST_CURRENCY:  
+        say_cb = zh_CN_say_money;
```

关于 `zh_CN_say_money` 函数具体的算法在此就不多讲了，有兴趣的读者可以参考 Jira 上的链接。

最后，我们再补充点小知识。当时 FreeSWITCH 出现了两个大的分支，一个是 `master`，一个是 `v1.2.stable`。前者是最新的开发版，并将成为新的 1.4 版，而后者将保持 1.2 版的向后兼容。因此，在这个时期，提交代码时要同时提交到两个分支中。

首先，笔者在本地提交了代码，在提交的 Message 信息中注明了“`FS-2809 --resolved`”，当该提交推到远程的 Git 代码库时，代码库中的钩子程序（`hook`）会自动与“FS-2809”那条 Jira 报告相关联，并将 Jira 报告的状态设为“Resolved”。

```
$ git ci -m 'FS-2809 --resolved' .  
[master 51d3282] FS-2809 --resolved  
1 file changed, 97 insertions(+), 1 deletion(-)
```

然后，通过 `cherry-pick` 将本次修改合并到 `v1.2.stable` 分支中：

```
$ git checkout v1.2.stable
Switched to branch 'v1.2.stable'
$ git cherry-pick 51d3282
[v1.2.stable f90e828] FS-2809 --resolved
1 file changed, 97 insertions(+), 1 deletion(-)
```

最后将本地两个分支的修改推到远程 Git 仓库，让世界了解中国。

```
$ git push ssh master
$ git push ssh v1.2.stable
```

从本例中可以看出，实现一个新的 Interface 也不是很复杂的事。当然，仔细阅读源代码，保持与其它开发者沟通是很重要的。

3.2 写一个新的 FreeSWITCH 编解码模块

我们前面的例子都是在以前的代码上打补丁，在本节，我们看一下如何增加一个新的模块。

在笔者测试 VP8 视频编码时，FreeSWITCH 还不支持 VP8，因而需要自己添加支持。好在，在 FreeSWITCH 中写一个新模块很简单。而且，由于 FreeSWITCH 中的视频模块不支持转码，因而，大部分回调函数什么也不做。

我们首先在 FreeSWITCH 源代码目录中 `src/mod/codecs` 下创建 `mod_vp8` 目录，并在里面创建 `mod_vp8.c`，然后，找一个类似的编解码模块，并把它里面的内容复制过来稍加修改即可。当时笔者发现与 VP8 最像的模块是 `mod_theora`，因此就直接复制了 `mod_theora.c` 里面的内容。修改后的 `mod_vp8.c` 内容如下。

首先，是 `include` 和模块声明。在该模块中，我们只需要其 `load` 函数。

```
33 #include <switch.h>
34
35 SWITCH_MODULE_LOAD_FUNCTION(mod_vp8_load);
36 SWITCH_MODULE_DEFINITION(mod_vp8, mod_vp8_load, NULL, NULL);
```

在编解码模块中，当在核心中初始化一个编码时，首先回调的就是 `init` 回调，即这里的 `switch_vp8_init` 函数。该函数在此要做的事情不多，基本上直接返回了成功——`SWITCH_STATUS_SUCCESS`。

```
38 static switch_status_t switch_vp8_init(switch_codec_t *codec, switch_codec_flag_t flags, const
↳ switch_codec_settings_t *codec_settings)
39 {
...
51     return SWITCH_STATUS_SUCCESS;
```

如果在调用该模块进行编码时或解码时，将调用这里的 `encode` 或 `decode` 函数。由于我们并不支持视频的编、解码，因此直接返回 `SWITCH_STATUS_FALSE`。实际上，收于核心本身不支持编解码，因而永远也不会回调到这里。

```
55 static switch_status_t switch_vp8_encode(switch_codec_t *codec,
...
61 {
62     return SWITCH_STATUS_FALSE;
63 }
64
65 static switch_status_t switch_vp8_decode(switch_codec_t *codec,
...
71 {
72     return SWITCH_STATUS_FALSE;
73 }
```

当然，最后释放编解码器的回调函数 `destroy` 也很简单：

```
75 static switch_status_t switch_vp8_destroy(switch_codec_t *codec)
76 {
77     return SWITCH_STATUS_SUCCESS;
78 }
```

其实该模块最重要的就是 `load` 函数了。在该函数中，第 82 行初始化了一个 `codec_interface`，它是一个 `switch_codec_interface_t` 类型的指针，说明我们想要创建一个 Codec Interface。第 84 行就紧接着创建了它。第 85 行，将该 `codec_interface` 安装到核心中去。

```
80 SWITCH_MODULE_LOAD_FUNCTION(mod_vp8_load)
81 {
82     switch_codec_interface_t *codec_interface;
83     /* connect my internal structure to the blank pointer passed to me */
84     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
85     SWITCH_ADD_CODEC(codec_interface, "VP8 Video (passthru)");
```

第 87 行, 在该 `codec_interface` 上增加了一个实现 (Implementation), 以及实现的回调函数。具体的参数定义我们在此就不多讲了, 总之它定义了四个回调函数, 即我们上面讲过的 `init`、`encode`、`decode` 和 `destroy` (参考 20.3.13 节的内容)。虽然有些回调函数什么也不做, 不过, 我们也最好写上它们。最后, 返回 `SWITCH_STATUS_SUCCESS` 以标明模块加载成功(第 102 行)。

```

87     switch_core_codec_add_implementation(pool, codec_interface,
88         SWITCH_CODEC_TYPE_VIDEO, 99, "VP8", NULL, 90000, 90000, 0,
89         0, 0, 0, 0, 1, 1, switch_vp8_init, switch_vp8_encode,
            switch_vp8_decode, switch_vp8_destroy);
...
102     return SWITCH_STATUS_SUCCESS;

```

后来, Anthony 在做 WebRTC 时, 还用同样的方法增加了“red”和“ulpfec”视频编码, 不过, 那就是后话了 (见第 91 和 96 行)。

```

91     SWITCH_ADD_CODEC(codec_interface, "red Video (passthru)");
96     SWITCH_ADD_CODEC(codec_interface, "ulpfec Video (passthru)");

```

有了上述的目录和模块实现文件后, 在核心进行 `configure` 的时候将会自动生成一个 Makefile, 不过, 我们也可以先自己写一个 Makefile 用于测试。在 Makefile 中加入如下内容后就可以编译该模块了。其中第 1 行指定 FreeSWITCH 源代码的主目录, 第 2 行装入通用的模块编译规则:

```

BASE=../../../../../
include $(BASE)/build/modmake.rules

```

然后, 直接在当前目录下可以使用如下命令编译安装:

```
# make install
```

接下来就可以在 FreeSWITCH 中直接加载使用了。当然, 最后, 笔者把该模块也提交给官方了, 相关 Jira 的地址是: <http://jira.freeswitch.org/browse/FS-4092>。

3.3 从头开始写一个模块

在 22.2 节, 我们给大家讲了编码解码模块的实现方法。本节, 我们再来从头实现一个综合性模块, 实现自己的 Dialplan、自己的 App 以及自己的 API。

3.3.1 初始准备工作

在准备下一步之前，我们先要为我们的模块取一个名字。笔者写书写到这里，脑子几乎用尽了，实在想不出更有创意的名字了，不如，索性就叫 `mod_book` 吧。

我们的 `mod_book` 也将脱离 FreeSWITCH 源代码的环境，单独存放。因此，你可以在任何喜欢的目录下创建目录 `mod_book`，然后在里面创建 `mod_book.c`。内容我们还是参照上面讲的 `mod_vp8.c`（它已经足够简单了），将不需要的函数的功能删除后，并把所有的 `mod_vp8` 替换为 `mod_book`，得到我们新的模块文件如下：

```
01 /* Book Example: Dialplan/App/API Author: Seven Du */
02
03 #include <switch.h>
04
05 SWITCH_MODULE_LOAD_FUNCTION(mod_book_load);
06 SWITCH_MODULE_DEFINITION(mod_book, mod_book_load, NULL, NULL);
07
08 SWITCH_MODULE_LOAD_FUNCTION(mod_book_load)
09 {
10     *module_interface = switch_loadable_module_create_module_interface(
11         pool, modname);
12     return SWITCH_STATUS_SUCCESS;
13 }
```

可以看出，该模块应该是最简单了，它只有短短的 12 行（去掉注释只有 10 行）。我们迅速创建一个 Makefile，内容如下（注意，我们这里的 `BASE` 变量引用的是一个绝对路径，它就是 FreeSWITCH 源代码的路径，如果你的源代码路径与笔者的不同，应该相应地修改它）：

```
BASE=/usr/src/freeswitch
include $(BASE)/build/modmake.rules
```

然后，直接在当前目录执行 `make install`，该模块就安装好了。

然后，到 FreeSWITCH 控制台上，加载该模块，从日志输出中可以看到我们的模块已经加载好了：

```
freeswitch> load mod_book
[CONSOLE] switch_loadable_module.c:1464 Successfully Loaded [mod_book]
```

虽然到此为止，我们的模块还什么都不能做，但至少它顺利加载了。我们要把这阶段性地成果记录下来。执行以下三条命令将这些成果记录到 Git 中。

```
$ git init
$ git add Makefile mod_book.c
$ git ci -m 'initial commit'
```

3.3.2 写一个简单的 Dialplan

为了使我们的模块更有用，我们需要增加一些功能。在此，我们就实现一个自己的 Dialplan Interface——我们仍然起名叫“book”。下面，我们修改 `load` 函数，首先增加一个变量声明：

```
switch_dialplan_interface_t *dp_interface;
```

然后，在 `*module_interface` 一行后，向核心注册我们的 Dialplan，并设置一个回调函数：

```
SWITCH_ADD_DIALPLAN(dp_interface, "book", book_dialplan_hunt);
```

然后实现该回调函数。注意这里我们文件中的行号发生了变化。该回调函数是使用 `SWITCH_STANDARD_DIALPLAN` 声明的。在第 10 行，我们定义了一个 `switch_caller_extension_t` 类型的指针变量，用于定义相关的 `extension`（与 XML Dialplan 中的 `<extension>` 标签相对应）。第 11 行将得到当前的 `channel`。第 14 行将得到一个 `caller_profile`，它里面保存了主叫用户的相关信息。如，在第 20 行我们就在日志中打印出了一些我们关心的信息。该信息跟我们最早在《权威指南》第 6 章讲到的“绿色的行”是一样的，在此，我们自己编程，实现了“绿色的行”。

```
08 SWITCH_STANDARD_DIALPLAN(book_dialplan_hunt)
09 {
10     switch_caller_extension_t *extension = NULL;
11     switch_channel_t *channel = switch_core_session_get_channel(session);
12
13     if (!caller_profile) {
14         caller_profile = switch_channel_get_caller_profile(channel);
15     }
16
17     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_INFO,
18         "Processing %s <%s>->%s in context %s\n",
```

```
19     caller_profile->caller_id_name, caller_profile->caller_id_number,  
20     caller_profile->destination_number, caller_profile->context);
```

当 FreeSWITCH 执行到该回调函数时，说明有一路电话进入了路由（ROUTING）阶段，我们要“查找 Dialplan，返回对应的 Extension（或里面的 Action），以后在后续 Channel”进入执行阶段时（EXECUTE），执行相关的 App。在第 22 行，我们初始化了一个 extension。第 26 行，往该 extension 上增加了一个 App。在此，我们并没有进行任何“查找”，而是直接硬编码了一个“log”App。最后，返回我们生成的 extension（第 29 行）。

```
22     extension = switch_caller_extension_new(session, "book", "book");  
23  
24     if (!extension) abort();  
25  
26     switch_caller_extension_add_application(session, extension,  
27         "log", "INFO Hey, I'm in the book");  
28  
29     return extension;  
30 }
```

进行了这些改变后，我们再次执行“make install”，并在 FreeSWITCH 控制台上使用 reload mod_book 重新加载模块。然后，可以快速的使用如下命令实验一下该 Dialplan 的效果：

```
freeswitch> originate user/1006 9999 book
```

还记得我们在 6.7 节所说的 Dialplan 的三要素吧，其中，9999 就是 Extension、book 就是 Dialplan 的名字，而 Context 由于省略了，默认就是 default，因此，可以在日志中看到如下的“绿色的行”，并且也可以看到我们增加的 App 也如期执行了（输出了对应的日志）。

```
[INFO] mod_book.c:17 Processing <0000000000>->9999 in context default  
[INFO] mod_dptools.c:1595 Hey, I'm in the book
```

至此，我们的 Dialplan 应该可以正常工作了。我们可以在 XML Dialplan 里转向它：

```
<action application="trasfer" data="9999 book default" />
```

也可以在 Sofia Profile 中（如 internal）直接使用它，配置如下：

```
<param name="dialplan" value="book"/>
<param name="context" value="default"/>
```

当然，我们的 Dialplan 功能还不是很强大，有待于进一步加强。不过，到这里，我们也算是一个里程碑了。我们继续将它提交到 Git 中（在提交之前执行一下 `git diff` 是个好习惯）：

```
$ git status
$ git diff
$ git commit -m 'add Dialplan Interface' .
```

到这里，我们应该更深入的理解到 Dialplan 到底是干什么的了——它就是负责找到一组 App，以后 FreeSWITCH 后续能执行这些 App。

3.3.3 增加一个 App

在上述的 Dialplan 的例子中，我们还是使用了 `log` App 作为例子。下面，我们该实现一个自己的 App 了。我们继续将该 App 也取名为“`book`”。实现的步骤如下：

首先，声明一个 `app_interface`：

```
switch_application_interface_t *app_interface;
```

将该 App 向核心注册，并增加一个回调函数 `book_function`：

```
SWITCH_ADD_APP(app_interface, "book", "book example", "book example",
               book_function, "[name]", SAF_SUPPORT_NOMEDIA);
```

实现该回调函数。该函数的参数将从 `data` 指针中传过来，如果为空的话（第 39 行），我们给它指定一个默认的名字；否则，就把传入的参数作为书的名字（第 42 行）。第 45 行输出一条日志，打印自己的名字：

```
35 SWITCH_STANDARD_APP(book_function)
36 {
37     const char *name;
```

```
38
39     if (zstr(data)) {
40         name = "No Name";
41     } else {
42         name = data;
43     }
44
45     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session),
46                     SWITCH_LOG_INFO, "I'm a book, My name is: %s\n", name);
47 }
```

当然，我们也不忘了在我们刚才的 Dialplan 中的 `extension` 上加上我们自己实现的 App：

```
switch_caller_extension_add_application(session, extension,
    "book", "FreeSWITCH - The Definitive Guide");
```

重新编译加载后，再执行一次，日志如下。可以看出，我们的 App 已经执行了。

```
[INFO] mod_dptools.c:1595 Hey, I'm in the book
[INFO] mod_book.c:45 I'm a book, My name is: FreeSWITCH - The Definitive Guide
```

最后，当然我们也不忘了再把我们的改变提交到 Git 里：

```
$ git commit -m 'add book App' .
```

3.3.4 写一个 API

通过上面的例子，相信读者也能想到，自己写一个 API 也是很容易的。为了完整性起见，我们就再来写一个。我们已经完全没有创意了，因此，该 API 的名字还是叫“`book`”。

声明一个 `api_interface`：

```
switch_api_interface_t *api_interface;
```

将 API 注册到核心，并设置回调函数 `book_api_function`：

```
SWITCH_ADD_API(api_interface, "book", "book example", book_api_function, "[name]");
```

实现回调函数。该回调函数与上面的 `book_function` 类似，不同的是，参数是从 `cmd`（第 53 行）获取的，而且这里我们没有打印日志，而是直接将命令的返回结果写到输出流（`stream`）里去了（第 59 行）。

```
49 SWITCH_STANDARD_API(book_api_function)
50 {
51     const char *name;
52
53     if (zstr(cmd)) {
54         name = "No Name";
55     } else {
56         name = cmd;
57     }
58
59     stream->write_function(stream, "I'm a book, My name is: %s\n", name);
60
61     return SWITCH_STATUS_SUCCESS;
62 }
```

重新编译并加载该模块后，我们在日志中看到如下的输出，它分别增加了以 `book` 为名称的 Dialplan、Application、和 API Function（我们在前面还没注意到呢）。

```
[NOTICE] switch_loadable_module.c:227 Adding Dialplan 'book'
[NOTICE] switch_loadable_module.c:269 Adding Application 'book'
[NOTICE] switch_loadable_module.c:315 Adding API Function 'book'
```

然后，我们在 FreeSWITCH 控制台上就可以执行 `book` 命令了：

```
freeswitch> book
I'm a book, My name is: No Name

freeswitch> book FreeSWITCH - The Definitive Guide
I'm a book, My name is: FreeSWITCH - The Definitive Guide
```

再一次，我们将里程碑成果提交到 Git 中：

```
$ git commit -m 'add book API Interface' .
```

3.3.5 小结

在本节的例子中，我们从无到有一步一步的实现了一个模块、添加了 Dialplan、APP 以及 API。在前面的例子中，虽然我们阅读了很多源代码，但是“纸上得来终觉浅”，唯有手工一步一步的做一次才知到每一步是怎么来的。

另外，我们也把每一步的结果都提交到 Git 版本管理系统中了，因而可以随时查阅历史，重温这段美好的回忆。各位读者也可以自己试一下，也可以从本书的 Github 站点上翻阅本书在写作时的“美好回忆”。

3.4 使用 libfreeswitch

FreeSWITCH 不仅有完善的可加载模块支持，而且，它的库 `libfreeswitch` 也可以被连接到其它系统中去，使得其它系统立即具有所有的 FreeSWITCH 中的功能。

3.4.1 自己写一个软交换机

下面，我们就尝试自己写一个软交换机。这里说的自己写，并不是一切都从头写，而是，利用现有的 `libfreeswitch` 库，把它集成到我们的系统中来。

假设我们已有了一个系统，该系统的功能非常强大。不过，为了便于讲解，我们把系统精减到了最简单的程序，精减到它在执行后仅仅打印一条信息就退出。

```
int main(int argc, char **argv)
{
    printf("Hello, MySWITCH is running ...\n");
    return 0;
}
```

下面，我们要将 `libfreeswitch` 集成进我们的系统中，因此，我们将 `main` 函数做了一些改变。代码如下。其中，我们在第 3 行装入了 `switch.h` 头文件，以便我们能引用里面的函数；第 7 行，我们设置一个 `flags` 标志，让它在使用核心数据库；第 8 行，定义一个 `console` 变量并设为 `TRUE`。第 13 行，设置一些默认的全局参数；第 14 行，初始化并加载模块；第 15 行，进入控制台循环。

```
01 /* MySwitch using libfreeswitch */
02
03 #include <switch.h>
04
05 int main(int argc, char** argv)
06 {
07     switch_core_flag_t flags = SCF_USE_SQL;
08     switch_bool_t console = SWITCH_TRUE;
09     const char *err = NULL;
10
11     printf("Hello, MySWITCH is running ...\n");
12
13     switch_core_set_globals();
14     switch_core_init_and_modload(flags, console, &err);
15     switch_core_runtime_loop(!console);
16     return 0;
17 }
```

通过这短短的几行，我们就写了一个功能强大的交换机，它具有 FreeSWITCH 全部的功能。我们通过如下的 Makefile 来编译它：

```
FS = /usr/local/freeswitch
INC = -I$(FS)/include
LIB = -L$(FS)/lib

all: myswitch

myswitch: myswitch.c
    gcc -o myswitch -ggdb $(INC) $(LIB) -lfreeswitch myswitch.c
```

在上面的 Makefile 中，最开始三行我们定义了三个变量。其中 INC 和 LIB 分别指定头文件和库文件的参数。最后一行使用 gcc 进行编译，输出可执行文件为 myswitch；为了调试方便，我们在编译时使用 -ggdb 加入符号表；“-lfreeswitch”为连接 libfreeswitch.so 库文件（在 Mac 上为“freeswitch.dylib”），最后的 myswitch.c 为源文件名。

执行 make 即可进行编译，编译完成后运行结果如下，可以看到与 FreeSWITCH 中类似的日志，一个强大的软交换机诞生了。

```
./myswitch
Hello, MySWITCH is running ...
2013-12-10 20:50:52.349175 [INFO] switch_event.c:669 Activate Eventing Engine.
...
```

3.4.2 使用 libfreeswitch 提供的库函数

在大多数情况下，我们不会重新发明一个 FreeSWITCH，而是想使用它提供的库文件中有用的部分。在下面这个例子中，我们就用到了它的文件接口、编码转换接口，以及 RTP 等功能。

我们程序的功能是从本地音频文件中读取数据，然后用 PCMU 进行编码，并通过 RTP 发送出去。将源文件命名为 `myrtp.c`，它的内容如下。

在 `main` 函数的最开始，我们定义并初始化了很多变量。在此，我们先不介绍这些变量，等后面用到的时候必要的话再讲。

```
1  /* File/Codec/RTP Example Author: Seven Du */
2
3  #include <switch.h>
4
5  int main(int argc, char *argv[])
6  {
7      switch_bool_t verbose = SWITCH_TRUE;
8      const char *err = NULL;
9      const char *fmt = "";
10     int ptime = 20;
11     const char *input = NULL;
12     int channels = 1;
13     int rate = 8000;
14     switch_file_handle_t fh_input = { 0 };
15     switch_codec_t codec = { 0 };
16     char buf[2048];
17     switch_size_t len = sizeof(buf)/2;
18     switch_memory_pool_t *pool = NULL;
19     int blocksize;
20     switch_rtp_flag_t rtp_flags[SWITCH_RTP_FLAG_INVALID] = { 0 };
21     switch_frame_t read_frame = { 0 };
22     switch_frame_t write_frame = { 0 };
23     switch_rtp_t *rtp_session = NULL;
24     char *local_addr = "127.0.0.1";
25     char *remote_addr = "127.0.0.1";
26     switch_port_t local_port = 4444;
27     switch_port_t remote_port = 6666;
28     char *codec_string = "PCMU";
29     int payload_type = 0;
30     switch_status_t status;
```

从命令行参数获取，音频文件名放到 `input` 变量中（第 34 行）。

```
32     if (argc < 2) goto usage;
33
34     input = argv[1];
```

第 36 行初始 `libfreeswitch` 的内核，这里我们使用了 `SCF_MINIMAL` 选项，它将启动最小配置（因为我们这里不需要完整的 FreeSWITCH）。

```
36     if (switch_core_init(SCF_MINIMAL, verbose, &err) != SWITCH_STATUS_SUCCESS) {
37         fprintf(stderr, "Cannot init core [%s]\n", err);
38         goto end;
39     }
```

第 41 行，设置一些全局的参数。第 42 行初始化可加载模块的设置，我们使用了 `SWITCH_FALSE` 参数不记它自动加载模块，而是在后面手工加载。如，第 43 ~ 44 行就加载了两个核心的模块，它们都是在核心中实现的，`CORE_SOFTTIMER_MODULE` 是一个时钟模块，用于定时；`CORE_PCM_MODULE` 即 PCM 编解码模块，用于 PCMU/PCMA 编解码。由于我们这里只用到 PCMU，因此，其它编解码模块就不需要加载了，否则，则需要手工加载对应的编解码模块。由于我们要读取音频文件，因此，我们在第 47 行加载了 `mod_sndfile` 模块，它使用 `libsndfile` 库支持很多类型的声音文件，如 `“.au”`，`“.aiff”` 等。当然，读者通过前面的学习也可能会想到，如果这里我们需要支持 `mp3` 的话就需要加载 `mod_shout` 了。

```
41     switch_core_set_globals();
42     switch_loadable_module_init(SWITCH_FALSE);
43     switch_loadable_module_load_module("", "CORE_SOFTTIMER_MODULE", SWITCH_TRUE, &err);
44     switch_loadable_module_load_module("", "CORE_PCM_MODULE", SWITCH_TRUE, &err);
45
46     if (switch_loadable_module_load_module((char *) SWITCH_GLOBAL_dirs.mod_dir,
47         (char *) "mod_sndfile", SWITCH_TRUE, &err) != SWITCH_STATUS_SUCCESS) {
48         fprintf(stderr, "Cannot init mod_sndfile [%s]\n", err);
49         goto end;
50     }
```

第 52 行初始化一个内存池。第 57 行调用 `switch_core_file_open` 打开输入的音频文件。其参数的值我们都在 `main` 函数的一开始定义了。其中，`channels` 为声道的数量，`rate` 为采样率，读者可以倒回去查看一下。如果音频文件中的参数与这里的不匹配，它将按我们在这里指定的自动进行转换。

```
53     switch_core_new_memory_pool(&pool);
54
```

```

55     fprintf(stderr, "Opening file %s\n", input);
56
57     if (switch_core_file_open(&fh_input, input, channels, rate,
58         SWITCH_FILE_FLAG_READ | SWITCH_FILE_DATA_SHORT, NULL) != SWITCH_STATUS_SUCCESS) {
59         fprintf(stderr, "Couldn't open %s\n", input);
60         goto end;
61     }

```

第 63 行初始化 PCMU 编解码 `codec`。音频数据从文件中读出来后，都是以 L16 编码的线性编码，后面我们需要把它们转成 PCMU。其中，`rate` 为采样率、`ptime` 为打包时间、`channels` 为声道数。`SWITCH_CODEC_FLAG_ENCODE` 标志说明我们只需要用到该编码器的编码器，即不需要用它解码。

```

63     if (switch_core_codec_init(&codec,
64         codec_string, fmt, rate, ptime, channels,
65         SWITCH_CODEC_FLAG_ENCODE, NULL, pool) != SWITCH_STATUS_SUCCESS) {
66         fprintf(stderr, "Couldn't initialize codec for %s@%dh@%di\n", codec_string, rate, ptime);
67         goto end;
68     }

```

我们可以根据采样率和打包时间算出一个数据包的长度 `len` 和需要的内存空间 `blocksize` (第 78 行)，在此，我们使用的 PCMU 编码的数据长度就是 $8000 * 20 / 1000 = 160$ ，即每个 RTP 包有 160 个字节的数据，而原始读取来的数据由于是使用 16 位的存储，因此每个数据有两个字节，所以实际原始数据的长度是 $160 * 2 = 320$ 字节。

```

78     blocksize = len = (rate * ptime) / 1000;
79     switch_assert(sizeof(buf) >= len * 2);
80     fprintf(stderr, "Frame size is %d\n", blocksize);

```

接下来，在第 74 行初始化系统 RTP 环境。然后初始化一个 RTP 的标志参数。第 76 行表示它支持输入输出；第 77 行表示采取非阻塞的方式发送；第 78 行表示允许调试，它将在日志中找印调试信息；第 79 行指定使用时钟，以更好地定时。然后，在第 81 ~ 84 行初始化一个 `rtp_session`，它的参数包含了 RTP 中必要的参数：本地、远程 IP 地址和端口，负载类型 (Payload Type)，采样率以及打包间隔等。另外，`soft` 是一个定时器的名字，它是核心提供的定时器。

```

74     switch_rtp_init(pool);
75
76     rtp_flags[SWITCH_RTP_FLAG_IO] = 1;
77     rtp_flags[SWITCH_RTP_FLAG_NOBLOCK] = 1;
78     rtp_flags[SWITCH_RTP_FLAG_DEBUG_RTP_WRITE] = 1;

```

```

79     rtp_flags[SWITCH_RTP_FLAG_USE_TIMER] = 1;
80
81     rtp_session = switch_rtp_new(local_addr, local_port,
82         remote_addr, remote_port,
83         payload_type, rate / (1000 / ptime), ptime * 1000,
84         rtp_flags, "soft", &err, pool);

```

`libfreeswitch` 默认会捕获各种信号，因此，我们在第 99 行将信号捕获回调设为空值，以后我们在调试的时候随时可以按“`Ctrl+C`”终止程序。

```

91     signal(SIGINT, NULL); /* allow break with Ctrl+C */

```

接下来就是无限循环一直从文件中读取数据。我们每次只读取一帧（`len`）大小的数据，数据将读到 `buf` 缓冲区中（第 93 行）。然后，在第 100 ~ 101 行，将读到的数据进行编码，编码后的数据将存储到 `encode_buf` 中，数据长度可以在 `encoded_len` 中得到。

```

93     while (switch_core_file_read(&fh_input, buf, &len) ==
94         SWITCH_STATUS_SUCCESS) {
95         char encode_buf[2048];
96         uint32_t encoded_len = sizeof(buf);
97         uint32_t encoded_rate = rate;
98         unsigned int flags = 0;
99
100        if (switch_core_codec_encode(&codec, NULL, buf, len*2, rate,
101            encode_buf, &encoded_len, &encoded_rate, &flags) != SWITCH_STATUS_SUCCESS) {
102            fprintf(stderr, "Codec encoder error\n");
103            goto end;
104        }

```

将数据编码成 PCMU 以后，我们就可以把它打包一个数据帧（`frame`）。下面就是设置该数据帧的各种参数：第 107 行，设置帧数据的地址指向我们新编码的数据；第 108 行设置数据的长度；第 109 行设置缓冲区的长度；第 110 行设置采样率；第 111 行设置该数据帧的编解码。然后在第 112 行将该数据帧给发送出去。

```

106        len = encoded_len;
107        write_frame.data = encode_buf;
108        write_frame.datalen = len;
109        write_frame buflen = len;
110        write_frame.rate= 8000;

```

```
111     write_frame.codec = &codec;
112     switch_rtp_write_frame(rtp_session, &write_frame);
```

第 114 行，我们尝试从该 `rtp_session` 中读取一帧数据。其实，由于没人给我们发送数据，它将于 20 毫秒后超时，进入下一次循环。当然，在下入下一次循环前我们要重置 `len` 的值（第 121 行），以避免 `len` 的值可能在某些场合下更改为其它的值引起的错误。

```
114     status = switch_rtp_zerocopy_read_frame(rtp_session,
115                                             &read_frame, 0);
116     if (status != SWITCH_STATUS_SUCCESS &&
117         status != SWITCH_STATUS_BREAK) {
118         goto end;
119     }
120
121     len = blocksize;
122 }
```

如果在前面遇到错误，或前面读文件的循环退出（如，读到文件尾），则代码会执行到第 124 行。后面，第 125 行会释放编解码器；第 126 行关掉文件接口；第 127 行释放内存池；并于第 128 行释放整个 `libfreeswitch` 的核心资源，程序结束。

```
124 end:
125     switch_core_codec_destroy(&codec);
126     if (fh_input.file_interface) switch_core_file_close(&fh_input);
127     if (pool) switch_core_destroy_memory_pool(&pool);
128     switch_core_destroy();
129     return 0;
```

当然，如果用户在命令行上输入错误的参数，程序将跳到 `usage` 标签，打印帮助信息。

```
131 usage:
132     printf("Usage: %s input_file\n\n", argv[0]);
133     return 1;
134 }
```

将上述程序编译运行后，便可以看到它从本地的 4444 端口向外（6666 端口）发送 RTP 数据了。由于数据长度为 160 字节，加上 12 个字节的 RTP 包头，因而日志中显示的一共是 172 字节。部分日志如下：

```
$ ./myrtp /wav/test.wav
Opening file /wav/test.wav
Frame size is 160
[DEBUG] switch_rtp.c:3047 Starting timer [soft] 160 bytes per 20ms
W NoName b= 172 127.0.0.1:4444 127.0.0.1:6666 127.0.0.1:4444 pt=0 ts=160 m=1
W NoName b= 172 127.0.0.1:4444 127.0.0.1:6666 127.0.0.1:4444 pt=0 ts=320 m=0
W NoName b= 172 127.0.0.1:4444 127.0.0.1:6666 127.0.0.1:4444 pt=0 ts=480 m=0
```

如果在运行时，不想要 FreeSWITCH 打印日志，可以在第 7 行将 `verbose` 调为 `SWITCH_FALSE`。读者也可以尝试修改其它参数。

读到这里，也许有的读者会问到，数据是否真的发送出去了？怎么验证呢？最简单的答案是，如果你需要这个程序，也许你已经有方法接收了。当然，如果没有的话，也可以把上述程序稍加改造——我们在第 114 行已经有接收 RTP 的代码，只需照着再写一个程序，将收到的数据保存到声音文件中，或者从声卡中放出来。这些，我们就留给读者自行练习了。

3.4.3 其它

其实，FreeSWITCH 的源代码中，也自带一些例子，其中就包括使用 `libfreeswitch` 的例子。经常有朋友问到——FreeSWITCH 安装目录的 `bin` 目录中，除了 `freeswitch` 和 `fs_cli` 比较熟悉外，其它的几个程序是干什么用的？

其实，回答这个问题很简单，只需不带参数要运行一下那个程序，或者加上 “-h” 参数，就很容易得到一个帮助信息。如，从 `fsxs` 程序的帮助信息看，是帮助编译一个模块的。笔者试了一下用它编译 22.3 节的 `mod_book`，不用 Makefile 也能编译（意味着不用源代码环境也可以编写模块），如，下面的命令将生成 `mod_book.so`

```
$ /usr/local/freeswitch/bin/fsxs build mod_book.so mod_book.c
CC mod_book.c
LD mod_book.so [mod_book.o]
```

使用下列命令就可以安装模块了：

```
/usr/local/freeswitch/bin/fsxs install mod_book.so
```

另外，`fs_encode` 程序可以将一个语音文件从一种编码转到另一种编码，`tone2wav` 则是帮助你从一个 TGML 标记语言描述的铃声转换成一个声音文件。如果要使用这两个程序，读者可以自行参考

一下相关的帮助信息。另外，既然，我们已经学会了查看源代码，自然可以在源代码中发现更多的秘密。这两个程序对应的源代码都在 FreeSWITCH 的源代码目录中（`fs_encode.c` 和 `tone2wav.c`），跟我们 22.4.2 节的实现方式差不多，也都用到了 `libfreeswitch`。读者可以找到这几个程序的源代码自己研究一下，并对比一下与我们在本节的实现有何异同。

3.5 主要数据结构和函数使用方法

FreeSWITCH 核心封装了一些函数和一些主要的数据结构，掌握这些函数和数据结构，可以在阅读源代码时事半功倍，当然，写代码也能事半功倍。

3.5.1 通过 UUID 获取 Session

在 FreeSWITCH 中，每一个 Session 都有一个 UUID 唯一对应，通过 UUID 能获取到相应的 Session 指针。每次获取后，Session 处于 `read-lock` 状态，因此用完后需要释放锁，否则，挂机后会出现 Session 不能释放的情况。

```
session = switch_core_session_locate(uuid);

if (session) {
    switch_channel_t *channel = switch_core_session_get_channel(session);
    const char *cid_number = switch_channel_get_variable(channel, "caller_id_number");
    switch_core_session_rwlock(session);
}
```

3.5.2 JSON

FreeSWITCH 内部封装了 `cJSON`⁵，并有一些扩展。

解析 JSON

```
const char *json_str = "{\"name\": \"Seven Du\", \"age\": 100, \"married\": true,
    \"address\": {\"postcode\": \"264000\", \"city\": \"Yantai\"}}";

// 解析成 cJSON 对象
cJSON *json = cJSON_Parse(json_str);
```

⁵参见<https://github.com/DaveGamble/cJSON>。

```
if (json) {
    const char *name = cJSON_GetObjectCstr(json, "name");
    cJSON *age = cJSON_GetObjectItem(json, "age");
    cJSON *married = cJSON_GetObjectItem(json, "married");
    cJSON *address = cJSON_GetObjectItem(json, "address");

    printf("name: %s\n", name);

    if (age && age->type == cJSON_Number) {
        printf("age: %d\n", cJSON_valueint);
        // or
        printf("age: %lf\n", cJSON_valuedouble);
    }

    if (married) {
        printf("married: %s\n", married->type == cJSON_True ? "true" : "false");
        // or
        printf("married: %s\n", cJSON_isTrue(married) ? "true" : "false");
    }

    if (address) {
        const char *postcode = cJSON_GetObjectCstr(address, "postcode");
        const char *city = cJSON_GetObjectCstr(address, "city");
    }

    // 释放
    cJSON_Delete(result_json);
}
```

JSON 编码

```
char *json_str = NULL;
cJSON *json = cJSON_CreateObject();
cJSON *address = cJSON_CreateObject();

if (json && address) {
    cJSON_AddStringToObject(json, "name", "Seven Du");
    cJSON_AddNumberToObject(json, "age", 100);
    cJSON_AddBooleanToObject(json, "married", cJSON_True);

    cJSON_AddStringToObject(address, "city", "Yantai");
    cJSON_AddStringToObject(address, "postcode", "264000");

    cJSON_AddItemToObject(json, "address", address);

    // 默认输出为“好看”模式，有相关的换行和缩进
```

```
    json_str = cJSON_Print(json);

    // 或者打印成紧凑模式
    // json_str = cJSON_PrintUnformatted(json);

    cJSON_Delete(json);
    free(json_str);
}
```

JSON 数组

```
cJSON *arr = cJSON_CreateArray();

// [1, 2, 3]
cJSON_AddItemToArray(arr, cJSON_CreateNumber(1));
cJSON_AddItemToArray(arr, cJSON_CreateNumber(2));
cJSON_AddItemToArray(arr, cJSON_CreateNumber(3));

cJSON_ArrayForEach(number, arr) {
    if (number->type == cJSON_Number) {
        printf("%d\n", number->valueint);
    }
}

cJSON_Delete(arr);

cJSON *string_arr = cJSON_CreateArray();

// ["1", "2", "3"]
cJSON_AddItemToArray(string_arr, cJSON_CreateString("1"));
cJSON_AddItemToArray(string_arr, cJSON_CreateNumber("2"));
cJSON_AddItemToArray(string_arr, cJSON_CreateNumber("3"));

cJSON_ArrayForEach(str, string_arr) {
    if (str->type == cJSON_String) {
        printf("%d\n", number->valuestring);
    }
}

cJSON_Delete(string_arr);

cJSON *users = cJSON_CreateArray();

cJSON *user1 = cJSON_CreateObject();
cJSON_AddStringToObject(user, "name", "Seven");
cJSON_AddItemToArray(users, user1);
```

```
cJSON *user2 = cJSON_CreateObject();
cJSON_AddStringToObject(user, "name", "Nine");
cJSON_AddItemToArray(users, user1);

cJSON_ArrayForEach(user, users) {
    const char *name = cJSON_GetObjectCstr(user, "name");
    if (name) {
        printf("name: %s\n", name);
    }
}

cJSON_Delete(users);
```

复制与分离

JSON 对象是一个整体的对象，如果 **Delete** 最顶层的对象，则会级连销毁所有对象。在有些情况下，我们只希望使用 JSON 对象的某一部分，则可以进行复制或分离。比如我们有以下对象：

```
cJSON *user1 = cJSON_CreateObject();
switch_assert(user1);

cJSON_AddStringToObject(user1, "name", "user1");

cJSON *address = cJSON_CreateObject();
cJSON_AddStringToObject(address, "city": "Yantai");
cJSON_AddItemToObject(user1, "address", "city");

cJSON *user2 = cJSON_CreateObject();
switch_assert(user2);

cJSON_AddStringToObject(user1, "name", "user1");
```

把 user1 的地址复制到 user2：

```
cJSON *address2 = cJSON_Duplicate(address, cJSON_True);
cJSON_AddItemToObject(user1, "address", "city");
```

删除 user1 的地址：

```
cJSON *address = cJSON_DetachItemFromObject(user1, "address");
cJSON_Delete(address);
```

如果不使用复制的方式，只是从 user1 将地址移动到 user2，则：

```
cJSON *address = cJSON_DetachItemFromObject(user1, "address");
cJSON_AddItemToObject(user2, "address", address);
```

3.5.3 JSON API

与 API 类似，FreeSWITCH 也提供了 JSON API 接口。通过使用 JSON API，可以调用一个 JSON 函数，函数的输入输出都是 JSON（`cJSON *` 类型）。JSON API 可以跨模块调用，也可以在外部应用程序中调用（通过 Websocket）。

定义 JSON API

可以在核心或模块中添加 JSON API 实现，如：

```
SWITCH_STANDARD_JSON_API(my_json_function)
{
    cJSON *reply = cJSON_CreateObject();

    cJSON_AddStringToObject(reply, "test", "Seven's JSON test code.");

    *json_reply = reply;

    return SWITCH_STATUS_SUCCESS;
}
```

`SWITCH_STANDARD_JSON_API` 的定义是：

```
#define SWITCH_STANDARD_JSON_API(name) static switch_status_t name (const cJSON *json, _In_opt_
↪ switch_core_session_t *session, cJSON **json_reply)
```

所以输入参数是 `json`，输出是一个双重指针 `json_reply`。API 的调用者应该负责释放 `json_reply`。

在模块加载的时候可以通过以下方式把实现的 JSON API 注册进去。

```
SWITCH_ADD_JSON_API(json_api_interface, "my_json", "JSON API", my_json_function, "");
```

JSON API 实现后，可以跨模块调用。如，在其它模块中可以这样调用：

```
cJSON *json = cJSON_CreateObject();
cJSON *data = cJSON_CreateObject();
cJSON *reply = NULL;

switch_assert(json);
switch_assert(data);

cJSON_AddStringToObject(data, "param", "whatever");

cJSON_AddStringToObject(json, "command", "my_json");
cJSON_AddItemToObject(json, "data", data);

switch_json_api_execute(json, NULL, &reply);

cJSON_Delete(json);

if (reply) {
    const char *test = cJSON_GetObjectCstr(*reply, "test");
    if (test) printf("%s", test); // 输出 Seven's JSON test code.
    cJSON_Delete(reply);
}
```

在 Lua 中调用 JSON API

`mod_lua` 中也集成了 JSON 接口。下面的 Lua 代码出自 `mod_lua/test/test_json.lua`：

```
json = freeswitch.JSON() -- 初始化 JSON

str = '{"a": "中文"}'      -- 定义字符串
x = json:decode(str)      -- 解析字符串，输出为一个 Lua table
assert(x.a == '中文')

str = '{"a": "1", "b": 2, "c": true, "d": false, "e": [], "f": {}, "g": [1, 2, "3"], "h": {"a": 1, "b": 2}}'
x = json:decode(str)      -- 解析字符串

freeswitch.consoleLog("INFO", serialize(x) .. "\n")
freeswitch.consoleLog("INFO", json:encode(x) .. '\n')
```

```
assert(x.a == "1")
assert(x.b == 2)

x = json:decode('["a", "b", true, false, null]')
freeswitch.consoleLog("INFO", serialize(x) .. "\n")

assert(x[1] == "a")

x = json:decode('[]')
assert(x)
x = json:decode('{}')
assert(x)
x = json:decode('blah')
assert(x == nil)

-- 将 Lua table 轮换为 JSON 字符串
s = json:encode({hello = "blah", seven="7", aa = {bb = "cc", ee="ff", more = {deep = "yes"}}, last="last",
↪ empty={}})
freeswitch.consoleLog("INFO", s .. "\n")

s = json:encode({"a", "b", "c"})
freeswitch.consoleLog("INFO", s .. "\n")

s = json:encode({a = 1, b = 2, c = 3, d=true, e=false, f=nil})
freeswitch.consoleLog("INFO", s .. "\n")

json:return_unformatted_json(true);
s = json:encode({})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == "{}")

json:encode_empty_table_as_object(false);
s = json:encode({})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == "[]")

s = json:encode({[1] = "a"})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == '["a"]')

s = json:encode({"a", "b", "c"})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == '["a","b","c"]')

-- sparse
s = json:encode({[3] = "c"})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == '{"3":"c"}')
```

```

s = json:encode({{name = "seven"}, {name="nine"}})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == '{"name":"seven"}, {"name":"nine"}')

s = json:encode({{name = "中文"}, [{"中文"]="也行"]})
freeswitch.consoleLog("INFO", s .. "\n")
assert(s == '{"name":"中文"}, {"中文":"也行"}')

-- 在 Lua 中执行 JSON API, 输入输出都是 Lua table
json:encode_empty_table_as_object(true);
cmd = {command="status", data={}}
ret = json:execute(cmd)
freeswitch.consoleLog("INFO", serialize(ret) .. "\n")

ret = json:execute(json:encode(cmd))
freeswitch.consoleLog("INFO", serialize(ret) .. "\n")

ret = json:execute2(cmd)
freeswitch.consoleLog("INFO", ret .. "\n")

ret = json:execute2(json:encode(cmd))
freeswitch.consoleLog("INFO", ret .. "\n")

```

JSON 事件广播和订阅

跟普通的 Event 类似，也可以向外广播 JSON Event。如：

```

const char *event_channel = "test.json"

cJSON *data = cJSON_CreateObject();
switch_assert(data);

cJSON_AddStringToObject("eventChannel", event_channel);
cJSON_AddStringToObject("params", "whatever");

switch_event_channel_broadcast(event_channel, &data, "test", NO_EVENT_CHANNEL_ID);

if (data) cJSON_Delete(data);

```

可以使用如下方式订阅事件：

```
static void test_event_channel_handler(const char *event_channel, cJSON *json, const char *key,
↪ switch_event_channel_id_t id, void *user_data)
{
    char *s = cJSON_Print(json);
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "event_channel: %s, key: %s, id: %u, json:
↪ [%s]\n", event_channel, key, id, s);
    free(s);
}

switch_event_channel_id_t event_channel_id;
switch_event_channel_bind("test", test_event_channel_handler, &event_channel_id, NULL);
switch_event_channel_bind("test.json", test_event_channel_handler, &event_channel_id, NULL);
```

默认情况下，在 `eventChannel` 有多个以 `.` 分隔的部分的情况下，FreeSWITCH 只发送第一部分以及完整的 `eventChannel`，如，一个 `eventChannel` 为 `test.test1.test2.test3`，则 FreeSWITCH 只会发送 `test` 和 `test.test1.test2.test3` 两个事件。

在 `switch.conf` 中开启 `event-channel-enable-hierarchy-deliver` 参数后，FreeSWITCH 会发送多个事件，即：

```
test
test.test1
test.test1.test2
test.test1.test2.test3
```

也可以通过 `event-channel-key-separator` 参数将默认的 `.` 隔符修改为其它符号。

3.5.4 切割字符串

在使用中经常需要把字符串按分隔符（如空格，逗号等）切开，FreeSWITCH 也提供了相应的函数。

```
#define MAX 2
int i;
char *words[MAX] = { 0 };

char *string = "this is a string";
char *dup = switch_strdup(string);
int n = switch_split(dup, ' ', tokens);
for (i = 0; i < n; i++) {
    printf("%s\n", words[i]);
}
```

```
}  
free(dup);
```

注意切割字符串会破坏原来的字符串，所以一般需要将原来的字符串复制一份。以上代码会将字符串按空格最多分隔成 `MAX` 份，如果在上面的例子中想得到更多的份数，则需要保证 `MAX` 大于 4（空格数 + 1）。

3.5.5 散列表

散列表（旧称哈希表）是一上非常有用的数据结构，注意散列表不是线程安全的，在遍历时要对散列表加锁。

遍历

遍历散列表并打印所有值：

```
switch_hash_index_t *hi = NULL;  
for (hi = switch_core_hash_first(my_hash); hi; hi = switch_core_hash_next(&hi)) {  
    void *val = NULL;  
    const void *key;  
    switch_core_hash_this(hi, &key, NULL, &val);  
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG,  
        "item: \"%s\" = \"%s\"\\n", (const char *)key, (char *)val);  
}
```

遍历删除

```
switch_hash_index_t *hi = NULL;  
while (hi = switch_core_hash_first(my_hash)) {  
    void *val = NULL;  
    const void *key;  
    switch_core_hash_this(hi, &key, NULL, &val);  
    // free(val); need a way to free the object  
    switch_core_hash_delete(my_hash, key);  
}
```

或

```

switch_hash_index_t *hi = NULL;
while ((item = switch_core_hash_first_iter(my_hash, hi)) {
    void *val = NULL;
    const void *key;
    switch_core_hash_this(item, &key, NULL, &val);
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG,
        "remove hash item: \"%s\" = \"%s\"", (const char *)key, (char *)val);
    switch_core_hash_delete(my_hash, key);
    switch_safe_free(val);
}

```

带有析构函数的删除

可以在插入散列表时提供一个析构函数，则相关的对象会在从散列表中删除时执行该函数以销毁相关的对象并释放资源。

```

switch_core_hash_insert_destructor(my_hash, "first", strdup("first"), free);
switch_core_hash_insert_destructor(my_hash, "second", strdup("second"), free);
switch_core_hash_insert_destructor(my_hash, "third", strdup("third"), free);

// 如果插入同一个 Key 时，则会“替换”掉原来的内容，并且，原来的内容会被正确的释放
switch_core_hash_insert_destructor(my_hash, "first", strdup("new_first"), free);

switch_core_hash_destroy(&my_hash);

```

3.5.6 绑定事件

FreeSWITCH 允许绑定一些特定的事件，不管是 FreeSWITCH 原生的事件而是 Custom 的事件。FreeSWITCH 内部使用多线程分发事件，所以，回调的顺序可能跟事件产生的顺序不同。

```

static void on_channel_originate(switch_event_t *event)
{
    char *event_subclass = switch_event_get_header(event, "Event-Subclass");
    switch_log_printf(SWITCH_CHANNEL_UUID_LOG(uuid), SWITCH_LOG_DEBUG,
        "got event %s %s\n", switch_event_name(event->event_id), zstr(event_subclass) ? "" :
        event_subclass);
}

static void on_conference_maintenance(switch_event_t *event)
{

```

```

char *event_subclass = switch_event_get_header(event, "Event-Subclass");
switch_log_printf(SWITCH_CHANNEL_UUID_LOG(uuid), SWITCH_LOG_DEBUG,
    "got event %s %s\n", switch_event_name(event->event_id), zstr(event_subclass) ? "" :
    ↪ event_subclass);
}

static void bind_to_events(void)
{
    switch_event_bind("my_module_name", SWITCH_EVENT_CHANNEL_ORIGINATE, NULL, on_channel_originate, NULL)
    switch_event_bind("my_module_name", SWITCH_EVENT_CUSTOM, "conference::maintenance",
    ↪ on_conference_maintenance, NULL);
}

static void unbind_from_events(void)
{
    switch_event_unbind_callback(on_channel_originate);
    switch_event_unbind_callback(on_conference_maintenance);
}

```

3.5.7 遍历 Event 消息头

FreeSWITCH 提供 `switch_event_get_header(switch_event_t *event, const char *header_name)` 函数获取相应的消息头域，也可以使用以下代码遍历所有消息头：

```

switch_event_header *hp;
for (hp = event->headers; hp; hp->next) {
    char *var = hp->name;
    char *val = hp->value;
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO,
        "Event header: \"%s\" = \"%s\" \n", var, val);
}

```

3.5.8 将以逗号分隔的字符串转换成事件

我们经常需要解析类似 `"{name=val,name2=val2}"` 的字符串，可以将其解析成 `Event` 结构，进而序列化成字符串，或 JSON 及 XML。

```

// do_something("{foo=bar,foo2=bar2,foo3=bar3}");
static void do_something(const char *args)
{

```



```

    if (!zstr(args)) {
        switch_event_t *params = NULL;
        char *args_dup = strdup(args);
        switch_event_create_brackets(args_dup, '{', '}', '=', &params, &args_dup, SWITCH_FALSE);
        if (params) {
            switch_event_header_t *hp;
            for (hp = params->headers; hp; hp->next) {
                char *var = hp->name;
                char *val = hp->value;
                switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "param: \"%s\" = \"%s\"\\n", var,
↪ val);
            }
            switch_event_free(&params);
        }
        switch_safe_free(args_dup);
    }
}

```

3.5.9 产生一个 custom 事件

经常需要在代码中产生一个事件与外界通信，一般来说，可以产生的事件类型有：

- `SWITCH_EVENT_CUSTOM`: 有 FreeSWITCH 默认的 Core Variables
- `SWITCH_EVENT_CLONE`: 没有 Core Variables

```

switch_event_t *new_event = NULL;
if (switch_event_create_subclass(&new_event, SWITCH_EVENT_CUSTOM, "my-custom-event") ==
↪ SWITCH_STATUS_SUCCESS) {
    switch_event_add_header_string(new_event, SWITCH_STACK_BOTTOM, "my-name", "Seven Du");
    switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Unique-ID", "%s", uuid);
    switch_event_add_body(event, SWITCH_STACK_BOTTOM, "This is really %s!", "Cool");
    switch_event_fire(&new_event);
}

```

3.5.10 队列与线程池

FreeSWITCH 提供一个 `switch_queue_t` 类型的队列，它是线程安全的。也就是说该队列可以在多线程环境下使用。生产者使用 `switch_queue_push()`（阻塞）或 `switch_queue_trypush()`（非阻塞），消费者调用 `switch_queue_pop()` 阻塞地获取队列中的内容，如果队列为空，则消费者会一直等待，除非 `switch_queue_interrupt_all()` 被调用。FreeSWITCH 内部的线程池就是使用了这个队列逻辑。

```

typedef struct {
    switch_queue_t *work_queue;
    switch_thread_rwlock_t *lock;
    int shutdown;
    int started;
} thread_pool;

static void *SWITCH_THREAD_FUNC worker_thread(switch_thread_t *thread, void *obj)
{
    thread_pool *thread_pool_data = (thread_pool *)obj;
    switch_thread_rwlock_rdlock(thread_pool_data->lock);
    thread_pool_data->started = 1;
    while (!thread_pool_data->shutdown || switch_queue_size(thread_pool_data->work_queue)) {
        void *work_item = NULL;
        if (switch_queue_pop_timeout(thread_pool_data->work_queue, &work_item, 100 * 1000) ==
            SWITCH_STATUS_SUCCESS &&
            work_item) {
            /* DO WORK HERE */
            switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Got item: %s\n", (char *)work_item);
            switch_safe_free(work_item);
        }
    }
    switch_thread_rwlock_unlock(thread_pool_data->lock);
}

int main(int argc, char **argv)
{
    switch_memory_pool_t *pool = NULL;
    thread_pool *thread_pool_data = NULL;

    /* CREATE THREAD POOL */
    switch_core_new_memory_pool(&pool);
    thread_pool_data = switch_core_alloc(pool, sizeof(*thread_pool_data));
    switch_thread_rwlock_create(&thread_pool_data->lock, pool);
    switch_queue_create(&thread_pool_data->work_queue, 100, pool);
    for (i = 0; i < 3; i++) {
        switch_thread_t *thread;
        switch_threadattr_t *thd_attr = NULL;
        switch_threadattr_create(&thd_attr, pool);
        switch_threadattr_detach_set(thd_attr, 1);
        switch_thread_create(&thread, thd_attr, worker_thread, thread_pool_data, pool);
    }
    for (i = 0; i < 10 && !thread_pool_data->started; i++) {
        switch_sleep(20 * 1000); // 20 ms
    }

    /* SEND WORK TO THREAD POOL */
}

```

```

    for (i = 0; i < 99; i++) {
        switch_queue_push(thread_pool_data, switch_mprintf("%d", i));
    }

    /* SHUTDOWN THREAD POOL */
    thread_pool_data->shutdown = 1;
    switch_thread_rwlock_wrlock(thread_pool_data->lock);

    switch_core_destroy_memory_pool(&pool);

    return 0;
}

```

3.5.11 Media Bug

FreeSWITCH 内部使用 Media Bug（类似于一个“三通”）在一个通信 Channel 上旁路获取和改变媒体资源。主要的使用场景如录音、监听、插话、ASR、信号音检测等都用到它。

```

/* 当 FreeSWITCH 读到每一个 frame 时都会触发这个回调 */
static switch_bool_t bug_callback(switch_media_bug_t *bug, void *user_data, switch_abc_type_t type)
{
    switch_core_session_t *session = switch_core_media_bug_get_session(bug);

    switch(type) {
        case SWITCH_ABC_TYPE_INIT: {
            switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Media bug starting
↳ up\n");
            /* Initialize anything you want here... store in session or your user data object */
            break;
        }
        case SWITCH_ABC_TYPE_READ_REPLACE: {
            switch_frame_t *rframe = switch_core_media_bug_get_read_replace_frame(bug);

            /* Handle read frame - bug can replace the frame contents here */

            /* Give modified read frame back to FS core */
            switch_core_media_bug_set_read_replace_frame(bug, rframe);
            break;
        }
        case SWITCH_ABC_TYPE_CLOSE: {
            /* Media bug closed - clean up anything you allocated here */
            switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Media bug starting
↳ up\n");

```

```

        break;
    }
    default:
        break;
}
return SWITCH_TRUE; // if you return SWITCH_FALSE, the media bug will be closed
}

static void start_media_bug(switch_core_session_t *session)
{
    switch_media_bug_t *bug = NULL;

    /* the bug flags SMBF_* control which frames FS will send to the bug. Use | operator to combine flags.
       See switch_type.h for all possible flags. These are the most commonly used:
       SMBF_TAP_NATIVE_READ - incoming audio - undecoded frame
       SMBF_READ_REPLACE - incoming audio from caller for replacement
       SMBF_READ_STREAM - copy of incoming audio from caller - after potential replacement from other bugs
       SMBF_TAP_NATIVE_WRITE - outgoing audio - undecoded frame
       SMBF_WRITE_REPLACE - outgoing audio to caller for replacement.
       SMBF_WRITE_STREAM - copy of outgoing audio to caller - after potential replacement from other bugs
    */

    /* create media bug to intercept read frames for possible replacement */
    if (switch_core_media_bug_add(session, "my_bug", NULL, bug_callback, handler, 0, SMBF_READ_REPLACE,
        ↪ &bug) != SWITCH_STATUS_SUCCESS) {
        switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Failed to create media
        ↪ bug\n");
    }
    /* save bug to channel for later removal */
    switch_channel_set_private(switch_core_session_get_channel(session), "__my_bug__", bug);
}

static void stop_media_bug(switch_core_session_t *session)
{
    /* get saved bug from channel */
    switch_media_bug_t *bug = switch_channel_get_private(switch_core_session_get_channel(session),
        ↪ "__my_bug__", bug);
    if (bug) {
        switch_core_media_bug_remove(session, &bug);
    }
}

```

3.5.12 任务调度

FreeSWITCH 内部使用 Scheduler 进行任务调度。

参见 `switch_core.c`，核心定义了两个任务，会定期回调相关的回调函数。

```
switch_scheduler_add_task(switch_epoch_time_now(NULL), heartbeat_callback, "heartbeat", "core", 0, NULL,
↳ SSHF_NONE | SSHF_NO_DEL);
```

```
switch_scheduler_add_task(switch_epoch_time_now(NULL), check_ip_callback, "check_ip", "core", 0, NULL,
↳ SSHF_NONE | SSHF_NO_DEL | SSHF_OWN_THREAD);
```

函数原型如下。设置任务的执行时间 `task_runtime`，回调函数 `func`，描述信息 `desc` 和任务组 `group`（后续可以删除整个任务组）。回调函数的参数由 `cmd_arg` 传进来。

```
SWITCH_DECLARE(uint32_t) switch_scheduler_add_task(time_t task_runtime,
switch_scheduler_func_t func,
const char *desc, const char *group, uint32_t cmd_id, void *cmd_arg, switch_scheduler_flag_t flags)
```

默认回调函数只会被回调一次，如果想在回调后继续执行，则需要在回调函数中重新设置 `runtime`，如 `heartbeat_callback` 所示：

```
SWITCH_STANDARD_SCHED_FUNC(heartbeat_callback)
{
    send_heartbeat();
    /* reschedule this task */
    task->runtime = switch_epoch_time_now(NULL) + runtime.event_heartbeat_interval;
}
```

核心的回调函数都是不可删除的（`SSHF_NO_DEL`）。如果不加这个标志，在任务可以删除。以下函数删除掉一个特定的任务。

```
SWITCH_DECLARE(uint32_t) switch_scheduler_del_task_id(uint32_t task_id)
```

也可以删掉整个任务组（`group`）。

```
SWITCH_DECLARE(uint32_t) switch_scheduler_del_task_group(const char *group)
```

如果 `group` 参数为一个 Channel 的 `UUID`，则相关 Channel 销毁时，会自动调用这个函数删除掉该 `UUID` 相关组的所有任务。

值得注意的是，任务回调函数应该尽可能少占用运行时间，如果占用时间比较多，则应该在独立的线程中运行（加 `SSH_F_OWN_THREAD` 标志），以避免阻塞其它任务。

如果希望在删除任务时能自动回收内存，则添加 `SSH_F_FREE_ARG` 标志。

详情参见 `switch_scheduler.c`（第??节）。

3.6 调试跟踪

在编写程序时，很少会一次写对，笔者也不例外。在笔者写 `myrtp.c` 的例子时，就遇到一个 Bug，导致在程序运行时出现 “`Segmentation fault`”，如下：

```
$ ./myrtp /wav/test.wav
Opening file /wav/vacation.wav
Frame size is 160

Segmentation fault: 11 (core dumped)
```

为了查找失败原因，笔者使用 GDB 进行调试。注意，在调试之前，需要先在编译时过程中使用 “`-ggdb`” 选项，以让 GCC 在产的可执行程序中写入相关的符号表。另外，还要注意，设置 `ulimit` 环境，以允许系统产生内核转储文件（`core dump`）。如，笔者使用如下命令设置允许内核转储文件的大小限制（`unlimited` 即无限制，默认是 0）：

```
ulimit -c unlimited
```

设置完成后，可以使用 `ulimit -a` 命令验证。然后，重新运行程序，产生 `core dump` 文件。在 Linux 系统上，`core dump` 文件一般是在当前目录中产生，在笔者使用的 Mac 每上，它固定产生在 `/cores` 目录中，并以当前的进程号作为扩展名。

在找到 `core dump` 文件中，笔者使用以下命令开启了 `gdb`，装入 `core dump` 文件用于调试：

```
$ gdb -core /cores/core.75886
GNU gdb 6.3.50-20050815 (Apple version gdb-1824)...
Reading symbols for shared libraries . done
#0 apr_palloc (pool=0x0, size=72) at apr_pools.c:603
603     if (pool->user_mutex) apr_thread_mutex_lock(pool->user_mutex);
```

其中，我们可以看到调用堆栈中的第“#0”层有一个 `pool` 变量是空指针（0x0 即 NULL），这可能是我们遇到问题的原因。但上述的命令并没有列出详细的调用栈，`apr_pools.c` 是 APR 底层的库，因而，我们还是需要从更上层查找问题。接着输入 `bt` 命令（Back Trace），我们更到了详细的调用栈，原来在调用堆栈的第“#2”层调用 `switch_rtp_init` 时 `pool` 指针就是空指针。

```
(gdb) bt
#0  apr_palloc (pool=0x0, size=72) at apr_pools.c:603
#1  0x0000000108db8b55 in apr_thread_mutex_create (mutex=0x108f146d0, flags=1, pool=0x0) at thread_mutex.c:
↳ 50
#2  0x0000000108d53373 in switch_rtp_init (pool=0x0) at switch_rtp.c:1328
#3  0x0000000108cce7e0 in main (argc=2, argv=0x7fff56f32760) at myrtp.c:74
```

我们查找源文件，发现该 `pool` 在任何地方都没有初始化，因而它是一个空指针。所以，增加了第 53 行的对内存池初始化的函数后，一切就都正常了。

在本例子中，错误比较明显，因而很容易发现。而在实际编程开发时，可能遇到一些更隐秘的错误，不容易直接从 Back Trace 中看到结果。那就要配合一些在代码中添加日志打印语句，或临时注释掉一些语句等手段以配合调试。有时候，直接使用 GDB 连接（`attach`）到正在运行的进程上进行调试、添加断点等，也是比较有效的调试方法。调试程序是一门细活，也需要有一定的耐心和经验。在此，我们仅通过此简单的例子，给大家讲解一下调试程序的基本原理和方法，剩下的就需要读者自己多加研究和练习了。

3.7 测试框架

本节代码 Commit Hash [1681db4](#)。

FreeSWITCH 从 1.8.4 开始集成了 FCT⁶ 测试框架。FCT 是一个非常简单的 C/C++ 测试框架，它仅有一个头文件，因而非常方便使用。FCT 基本的代码结构如下：

```
/* 首先加载头文件 */
#include "fct.h"

/* 引入更多需要的头文件，这里我们将使用字符串函数 strcmp. */
#include <string.h>

/* 定义测试案例 */
FCT_BGN()
{
    /* 简单的测试代码 */
```

⁶<https://github.com/imb/fctx>。

```
FCT_SUITE_BGN(simple)
{
    /* 一个测试案例，测试字符串相等 */
    FCT_TEST_BGN(strcmp_eq)
    {
        fct_chk(strcmp("FreeSWITCH", "FreeSWITCH") == 0);
    }
    FCT_TEST_END();

    /* 另一个测试案例，测试字符串不相等 */
    FCT_TEST_BGN(chk_neq)
    {
        fct_chk(strcmp("FreeSWITCH", "freeswitch") != 0 );
    }
    FCT_TEST_END();
}
FCT_SUITE_END();
}
FCT_END();
```

直接编译运行测试案例就可以打印结果。

FreeSWITCH 集成了 FCT 并根据 FreeSWITCH 做了扩展，且代码中已经有了一些测试案例。下面我们就来看几个核心的测试案例，它们位于源代码目录下的 `tests/unit` 目录中。

在该目录中执行 `make check` 可以看到类似如下的结果。

```
/Applications/Xcode.app/Contents/Developer/usr/bin/make  check-TESTS
PASS: switch_event
PASS: switch_hash
PASS: switch_ivr_originate
PASS: switch_utils
FAIL: switch_core
```

```
=====
Testsuite summary for freeswitch 1.9.0
=====
```

```
# TOTAL: 5
# PASS: 5
# SKIP: 0
# XFAIL: 0
# FAIL: 0
# XPASS: 0
# ERROR: 0
```

3.7.1 switch_ivr_originate.c

测试 `switch_ivr_originate` 函数。

L32 ~ L35, 加载头文件。定义两个回调函数 (L40, L49), 当发生回调时打印日志, 并更新 `reporting` 或 `destroy` 变量。

```
32 #include <switch.h>
33 #include <stdlib.h>
34
35 #include <test/switch_test.h>
36
37 int reporting = 0;
38 int destroy = 0;
39
40 static switch_status_t my_on_reporting(switch_core_session_t *session)
41 {
42     switch_assert(session);
43     reporting++;
44     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_INFO, "session reporting %d\n",
↳ reporting);
45
46     return SWITCH_STATUS_SUCCESS;
47 }
48
49 static switch_status_t my_on_destroy(switch_core_session_t *session)
50 {
51     switch_assert(session);
52     destroy++;
53     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "session destroy %d\n",
↳ destroy);
54
55     return SWITCH_STATUS_SUCCESS;
56 }
```

定义状态回调表。

```
58 static switch_state_handler_table_t state_handlers = {
59     /*.on_init */ NULL,
60     /*.on_routing */ NULL,
61     /*.on_execute */ NULL,
62     /*.on_hangup */ NULL,
63     /*.on_exchange_media */ NULL,
64     /*.on_soft_execute */ NULL,
```

```

65     /*.on_consume_media */ NULL,
66     /*.on_hibernate */ NULL,
67     /*.on_reset */ NULL,
68     /*.on_park */ NULL,
69     /*.on_reporting */ my_on_reporting,
70     /*.on_destroy */ my_on_destroy,
71     SSH_FLAG_STICKY
72 };

```

L74, FreeSWITCH 核心测试框架开始, 加载 `./conf` 目录下的配置文件 (一个比较精简的 `freeswitch.xml`)。

L76, 开始一个测试套件。L78, 定义一个 `SETUP` 函数, 在每个测试案例之前都需要运行这个函数。L80, 确保 `mod_loopback` 模块已加载。因为该测试依赖这个模块。

L84 定义 `TEARDOWN` 函数, 在每个测试案例执行完后都需要执行一下它, 可以在这里做一些清除工作。

```

74 FST_CORE_BEGIN("./conf")
75 {
76     FST_SUITE_BEGIN(switch_ivr_originate)
77     {
78         FST_SETUP_BEGIN()
79         {
80             fst_requires_module("mod_loopback");
81         }
82         FST_SETUP_END()
83
84         FST_TEARDOWN_BEGIN()
85         {
86         }
87         FST_TEARDOWN_END()

```

L89 定义了一个测试案例。首先使用 `switch_ivr_originate` 函数创建一个 Channel (L96)。`fst_requires` 函数测试 `session` (L97), 如果 `session` 为空, 则会退出并不会继续执行下面的测试。`fst_check` (L98) 则检查表达式是否为真, 不管真或假都会继续执行, 但是测试结束后会打印成功与失败的结果。

如果创建 `session` 成功, 则继续获取 `channel` (L100)。

```

89     FST_TEST_BEGIN(originate_test_early_state_handler)
90     {
91         switch_core_session_t *session = NULL;

```

```
92         switch_channel_t *channel = NULL;
93         switch_status_t status;
94         switch_call_cause_t cause;
95
96         status = switch_ivr_originate(NULL, &session, &cause, "null/+15553334444", 2, NULL, NULL,
↪ NULL, NULL, NULL, SOF_NONE, NULL, NULL);
97         fst_requires(session);
98         fst_check(status == SWITCH_STATUS_SUCCESS);
99
100         channel = switch_core_session_get_channel(session);
101         fst_requires(channel);
```

在当前的 `channel` 上添加状态回调函数（L103）。然后挂机（104）。

```
103         switch_channel_add_state_handler(channel, &state_handlers);
104         switch_channel_hangup(channel, SWITCH_CAUSE_NORMAL_CLEARING);
105         fst_check(!switch_channel_ready(channel));
```

`switch_ivr_originate` 返回的 `session` 是带锁的，因此，用完要解锁（L107）。

```
107         switch_core_session_rwlock(session);
```

最后检查 `reporting` 和 `destroy` 的状态，应该都会恰好调用一次（L110 ~ L111）。

```
109         switch_sleep(1000000);
110         fst_check(reporting == 1);
111         fst_check(destroy == 1);
112     }
113     FST_TEST_END()
```

另一个案例，挂机后（L129）才添加状态回调（131），那么，`reporting` 的值不变，`destroy` 的值增加了变成 2，说明被执行了一次。

```
115     FST_TEST_BEGIN(originate_test_late_state_handler)
116     {
117         switch_core_session_t *session = NULL;
118         switch_channel_t *channel = NULL;
119         switch_status_t status;
```

```

120         switch_call_cause_t cause;
121
122         status = switch_ivr_originate(NULL, &session, &cause, "null/+15553334444", 2, NULL, NULL,
↳ NULL, NULL, NULL, SOF_NONE, NULL, NULL);
123         fst_requires(session);
124         fst_check(status == SWITCH_STATUS_SUCCESS);
125
126         channel = switch_core_session_get_channel(session);
127         fst_requires(channel);
128
129         switch_channel_hangup(channel, SWITCH_CAUSE_NORMAL_CLEARING);
130         switch_sleep(1000000);
131         switch_channel_add_state_handler(channel, &state_handlers);
132
133         switch_core_session_rwlock(session);
134
135         switch_sleep(1000000);
136         fst_check(reporting == 1);
137         fst_check(destroy == 2);
138     }
139     FST_TEST_END()
140 }
141 FST_SUITE_END()
142 }
143 FST_CORE_END()

```

所以，以上两个测试案例在于测试在一个 Channel 上安装状态回调（State Handler），如果安装时当前 Channel 已挂机，则仅会执行 `on_destroy` 回调，否则在此之前会执行 `on_reporting` 回调。

执行 `./switch_ivr_originate` 可以得到如下结果：

```

originate_test_early_state_handler .....
PASS
originate_test_late_state_handler .....
PASS
-----
PASSED (2/2 tests in 0.029939s)

```

3.7.2 switch_utils.c

以下函数测试 `switch_url_encode`。

```

FST_TEST_BEGIN(benchmark)
{
    char encoded[1024];
    char *s = "ABCD";

    switch_url_encode(s, encoded, sizeof(encoded));
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "encoded: [%s]\n", encoded);
    fst_check_string_equals(encoded, "ABCD");

    s = "&bryän#!杜金房";
    switch_url_encode(s, encoded, sizeof(encoded));
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "encoded: [%s]\n", encoded);
    fst_check_string_equals(encoded, "%26bry%C3%A4n%23!%E6%9D%9C%E9%87%91%E6%88%BF");
}
FST_TEST_END()

```

执行结果：

```

$ ./switch_utils
benchmark .....
[INFO] switch_utils.c:56 encoded: [ABCD]
[INFO] switch_utils.c:61 encoded: [%26bry%C3%A4n%23!%E6%9D%9C%E9%87%91%E6%88%BF]
PASS
-----
PASSED (1/1 tests in 0.000046s)

```

3.7.3 test_mod_av.c

除了核心的测试案例外，每个模块也可以有测试案例。这是 `mod_av` 模块的一个测试案例。

L33，初始化全局变量。L36 ~ L49 定义命令行参数。

```

31 #include <test/switch_test.h>
32
33 int loop = 0;
34
35 /* Add our command line options. */
36 static fctl_init_t my_cl_options[] = {
37     {"--disable-hw",          /* long_opt */
38     NULL,                    /* short_opt (optional) */
39     FCTL_STORE_TRUE,         /* action */
40     "disable hardware encoder" /* 禁用硬件编码 */
41 },

```

```
42
43     {"--loop",                /* long_opt */
44     NULL,                    /* short_opt (optional) */
45     FCTCL_STORE_VALUE ,      /* action */
46     "loops to encode a picture" /* 循环次数 */
47     },
48     FCTCL_INIT_NULL /* Sentinel */
49 };
```

L53, 初始化命令行参数。如果在命令行上输入 `--loop` (L55), 则设置 `loop` 变量的值 (L56)。

```
51 FST_CORE_BEGIN("conf")
52 {
53     fctcl_install(my_cl_options);
54
55     const char *loop_ = fctcl_val("--loop");
56     if (loop_) loop = atoi(loop_);
```

L58 初始化一个模块测试, 自动加载该模块, L66 定义测试案例。L74 ~ L76 检查命令行参数以决定是否禁用硬件编码。

```
58 FST_MODULE_BEGIN(mod_av, mod_av_test)
59 {
60     FST_SETUP_BEGIN()
61     {
62     }
63     FST_SETUP_END()
64
65     FST_TEST_BEGIN(encoder_test)
66     {
67         switch_status_t status;
68         switch_codec_t codec = { 0 };
69         switch_codec_settings_t codec_settings = { 0 };
70
71         if (!fctcl_is("--disable-hw")) {
72             codec_settings.video.try_hardware_encoder = 1;
73         }
74     }
```

初始化视频编码。并检查是否成功 (L86)。

```

78         status = switch_core_codec_init(&codec,
79                                         "H264",
80                                         NULL,
81                                         NULL,
82                                         0,
83                                         0,
84                                         1, SWITCH_CODEC_FLAG_ENCODE | SWITCH_CODEC_FLAG_DECODE,
85                                         &codec_settings, fst_pool);
86         fst_check(status == SWITCH_STATUS_SUCCESS);

```

L88, 申请一帧 720p 图像。初始化一个缓冲区用于存放编码后的数据 (L91), 并初始化一个空的 `frame` (L92)。

填充 `frame` 结构。让出 12 个字节的 RTP 头域 (L96), 并设置 `frame` 的图像 (L102)。

```

88         switch_image_t *img = switch_img_alloc(NULL, SWITCH_IMG_FMT_I420, 1280, 720, 1);
89         fst_requires(img);
90
91         uint8_t buf[SWITCH_DEFAULT_VIDEO_SIZE + 12];
92         switch_frame_t frame = { 0 };
93
94         frame.packet = buf;
95         frame.packetlen = SWITCH_DEFAULT_VIDEO_SIZE + 12;
96         frame.data = buf + 12;
97         frame.datalen = SWITCH_DEFAULT_VIDEO_SIZE;
98         frame.payload = 96;
99         frame.m = 0;
100        frame.seq = 0;
101        frame.timestamp = 0;
102        frame.img = img;

```

循环 (L107) 进行编码 (L109)。一帧图像编码后的数据可以很长, 需要进行分包。FreeSWITCH 内分包的长度为 1200 字节左右 (不超过 1500)。如果发生了分包, 则编码函数会返回 `SWITCH_STATUS_MORE_DATA`, 表示编码器中还有后续的数据。最后一个分包的 `marker` 域 (即 `frame.m`) 为 1, 其它为 0 (L113)。打印编码的分片情况 (L122), 直到 `frame.datalen == 0` (L120) 退出。

如果 `loop` 大于 1, 则会继续重复编码。

```

104        int packets = 0;
105        switch_status_t encode_status;
106
107        do {

```

```

108         frame.datalen = SWITCH_DEFAULT_VIDEO_SIZE;
109         encode_status = switch_core_codec_encode_video(&codec, &frame);
110
111         if (encode_status == SWITCH_STATUS_SUCCESS || encode_status == SWITCH_STATUS_MORE_DATA)
112         ↪ {
113             fst_requires((encode_status == SWITCH_STATUS_SUCCESS && frame.m) || !frame.m);
114
115             if (frame.flags & SFF_PICTURE_RESET) {
116                 frame.flags &= ~SFF_PICTURE_RESET;
117                 fst_check(0);
118             }
119
120             if (frame.datalen == 0) break;
121
122             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "[%d]: %02x %02x | m=%d |
123 ↪ %d\n", loop, buf[12], buf[13], frame.m, frame.datalen);
124             packets++;
125         }
126     } while(encode_status == SWITCH_STATUS_MORE_DATA || loop-- > 1);

```

确保最后一帧的 `m` 值为 1 (L128)，以及确实有相应的分片输出 (L129)。最后释放编码器 (L131)。

```

128         fst_check(frame.m == 1);
129         fst_check(packets > 0);
130
131         switch_core_codec_destroy(&codec);
132     }
133     FST_TEST_END()

```

结束时卸载模块 (L139)。

```

135     FST_TEARDOWN_BEGIN()
136     {
137         const char *err = NULL;
138         switch_sleep(1000000);
139         fst_check(switch_loadable_module_unload_module(SWITCH_GLOBAL_dirs.mod_dir, (char
140 ↪ *)"mod_av", SWITCH_TRUE, &err) == SWITCH_STATUS_SUCCESS);
141     }
142     FST_TEARDOWN_END()

```



```

143     FST_MODULE_END()
144 }
145 FST_CORE_END()

```

从本案例也可以看到视频编码函数的用法。

3.7.4 test_avformat.c

本案例测试用 `libavformat` 写文件。L48 初始化图片，L50 初始化静音的音频数据，L58 打开文件准备写入。

```

31 #include <test/switch_test.h>
32
33 #define SAMPLES 160
34
35 FST_CORE_BEGIN("conf")
36 {
37     FST_MODULE_BEGIN(mod_av, mod_av_test)
38     {
39         FST_SETUP_BEGIN()
40         {
41             fst_requires_module("mod_av");
42         }
43         FST_SETUP_END()
44
45         FST_TEST_BEGIN(avformat_test_colorspace_RGB)
46         {
47             switch_status_t status;
48             switch_image_t *img = switch_img_alloc(NULL, SWITCH_IMG_FMT_I420, 1280, 720, 1);
49             switch_file_handle_t fh = { 0 };
50             uint8_t data[SAMPLES * 2] = { 0 };
51             switch_frame_t frame = { 0 };
52             switch_size_t len = SAMPLES;
53             uint32_t flags = SWITCH_FILE_FLAG_WRITE | SWITCH_FILE_DATA_SHORT | SWITCH_FILE_FLAG_VIDEO;
54             int i = 0;
55
56             fst_requires(img);
57
58             status = switch_core_file_open(&fh, "{colorspace=0}./test_RGB.mp4", 1, 8000, flags,
↪   fst_pool);
59             fst_requires(status == SWITCH_STATUS_SUCCESS);
60             fst_requires(switch_test_flag(&fh, SWITCH_FILE_OPEN));

```

L62 写入音频数据。

```
62         status = switch_core_file_write(&fh, data, &len);
63         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "status: %d len: %d\n", status,
↪ (int)len);
64         fst_check(status == SWITCH_STATUS_SUCCESS);
```

L67 构造一个 **frame**，并关联 **img**。L68 将图像数据视频数据（图像会在内部自动被编码成视频）。

```
67         frame.img = img;
68         status = switch_core_file_write_video(&fh, &frame);
69         fst_check(status == SWITCH_STATUS_SUCCESS);
```

读取 PNG 图像到 **ccimg** 中。初始化 RGB 颜色（L74），设为不透明（L75）。循环，每 10 帧换颜色（L77 ~ L85）。

```
71         switch_image_t *ccimg = switch_img_read_png("./cluecon.png", SWITCH_IMG_FMT_ARGB);
72         fst_requires(ccimg);
73
74         switch_rgb_color_t color = {0};
75         color.a = 255;
76
77         for (i = 0; i < 30; i++) {
78             len = SAMPLES;
79
80             if (i == 10) {
81                 color.r = 255;
82             } else if (i == 20) {
83                 color.r = 0;
84                 color.b = 255;
85             }
```

用颜色填充背景图像 **img**（L87），并把 **ccimg** 贴上去（L88，每次循环都变换位置）。写音频（L90），并将新的图像写入视频文件（L91）。

```
87         switch_img_fill(img, 0, 0, img->d_w, img->d_h, &color);
88         switch_img_patch(img, ccimg, i * 10, i * 10);
89
90         status = switch_core_file_write(&fh, data, &len);
91         status = switch_core_file_write_video(&fh, &frame);
```

```
92         switch_yield(100000);
93     }
94
95     switch_core_file_close(&fh);
96     switch_img_free(&img);
97     switch_img_free(&ccimg);
98 }
99 FST_TEST_END()
...
164 }
165 FST_MODULE_END()
166 }
167 FST_CORE_END()
```

本例也可以看出写视频文件的用法。执行完毕后会当前目录下生成一个 mp4 文件。

3.8 小结

本章我们主要讲了基于 FreeSWITCH 进行二次开发的知识。从最简单的汇报 bug 的注意事项开始，到真实的汇报 Bug 的例子。其中，我们精心选择了一些实际的案例，每个案例都涵盖一个或多个知识点，并对前面学到的内容进行了复习和补充。

另外，我们也讲了给 FreeSWITCH 增加新特性以及添加新模块的实例，并带领大家从头开始写了一个新的模块，添加了我们自己实现的各种 Interface。从前面章节中的纸上谈兵到了真正的战场上。

然后，我们还讲了如何把 FreeSWITCH 嵌入其它系统的例子，解答了许多朋友经常问到的一些问题。

最后，我们还配合本书的例子讲解了使用 GDB 进行调试的实例，作为开发中的一种辅助手段，相信也会对广大读者有所帮助。

在所有的实例中，我们都注意穿插讲解了以 Makefile 为主的编译设置，在 Git 作为版本控制系统的版本控制，以帮助读者更快更好地将学到的知识应用于实际项目中去。

总之，这些案例虽然由于本书篇幅的限制，我们写得都不长，但都最大限度的涵盖了相关的知识点和应该注意的问题，并给读者提供了深入学习和研究的方向和思路。

诚然，有时候，对程序代码的一些解释是繁琐和冗长的，有时候甚至再多的解释也比不过一句代码能说明问题。最正确最好的例子都在 FreeSWITCH 的源代码中。所以，希望读者在我们所学的知识的基础上，多看源代码，多多实践。

截至本章，我们的实战演练就到此为止了。“师傅领进门，修行在个人”，预祝大家都能很快精通 FreeSWITCH！

第四章 核心代码详解

从本章开始，我们看核心代码。

FreeSWITCH 的核心代码都在 `src` 目录下。我们按字母顺序一个一个的看。

所有代码均基于 Git commit hash `917d9b44`。读者在阅读时应该有一份 FreeSWITCH 源代码，并且，切换到该 commit：

```
git checkout 917d9b44
```

书中列出的代码大部分均标有行号。正文中引用的行号以“L”表示，如 L10 表示第 10 行。

4.1 g711.c

我们第一个要讲的是 `g711.c`。首先来说什么是 G711。说 G711 前要说什么是 PCM。

PCM¹的全称是 Pulse Code Modulator，即脉冲编码调制，是一种模拟信号的数字化方法。我们知道，在传统电话中，抽样频率是 `8000 Hz`，即每秒钟抽样 `8000` 次，每次抽样得到一个 PCM 抽样值，这个值用一个字（Word，两个字节）来表示。在 FreeSWITCH 中，一般使用 `short` 来定义。这样得到的数据，称为线性编码（Liner），因而，编码方式也称为 L16（16 个比特）。L16 是最基础的编码方式。

PCM 数据有两种基本的压缩方式，分别是 μ 律（*u-law*）和 a 律（*a-law*），希腊字母 μ 简单起见写成 *u*）。它们都是在 ITU G.711²中定义的，经过压缩后的编码称为 PCMA 和 PCMU，其中中国和欧洲默认使用前者，北美和日本默认使用后者。两种编码略有不同，但压缩后，都可以把两个字节的数数据压缩成一个字节，即 8 个比特，在 FreeSWITCH 中的表示是 `uint_8`。

`g711.c` 很简单，它 `include` 了 `g711.h`（我们一会再说）。然后定义了两张表：

¹参见<https://zh.wikipedia.org/wiki/%E8%84%88%E8%A1%9D%E7%B7%A8%E7%A2%BC%E8%AA%BF%E8%AE%8A>。

²参见<https://zh.wikipedia.org/wiki/G.711>。

```
35 #include "g711.h"
36
37 /* Copied from the CCITT G.711 specification */
38 static const uint8_t ulaw_to_alaw_table[256] = {
...
55 };
...
60 static const uint8_t alaw_to_ulaw_table[256] = {
...
77 };
```

后面定义了两个函数，分别实现了 **alaw** 和 **ulaw** 间的转换。很简单，就是查表：

```
79 uint8_t alaw_to_ulaw(uint8_t alaw)
80 {
81     return alaw_to_ulaw_table[alaw];
82 }
...
86 uint8_t ulaw_to_alaw(uint8_t ulaw)
87 {
88     return ulaw_to_alaw_table[ulaw];
89 }
```

其实，这两个函数 FreeSWITCH 并没有用到。而 FreeSWITCH 用到的，是 **g711.h**（在 **src/include** 目录）中的内容。

该文件最开头一是大段注释，读者应该好好读一读。在此，我们就不逐字翻译了。

L42 ~ 43 是 **.h** 文件常用的手法，以便该文件在被多次 **include** 时不出错。

```
42 #if !defined(FREESWITCH_G711_H)
43 #define FREESWITCH_G711_H
```

下面的代码常用在 C 语言实现的函数在 C 和 C++ 中混合编译的情况，防止 C++ 编译器进行名字修饰³。

```
45 #ifdef __cplusplus
46 extern "C" {
47 #endif
```

³参见<https://zh.wikipedia.org/wiki/名字修饰>。

```
...
330     uint8_t alaw_to_ulaw(uint8_t alaw);
...
336     uint8_t ulaw_to_alaw(uint8_t ulaw);
...
338 #ifdef __cplusplus
339 }
340 #endif
```

所谓名字修饰，简单来讲，由于 C++ 支持函数和运算符重载，如：

```
int sum(int a, int b);
int sum(float a, float b);
```

所以，编译器要将两个 `sum` 函数编译成不同的函数名，这种方法称为修饰（Mangling）。至于具体的命名规则，不同的编译器的不同的方法。但是，这样一来，如果源代码通过 C++ 编译器编译成库文件，在其它项目中调用这些代码是没有问题的，但如果是在 C 语言中调用，就找不到这个函数了。所以，上面的代码片断中，如果判断是 C++ 编译器的话（`#ifdef __cplusplus`），就使用 `extern "C"` 禁止编译器进行名字修饰。

下面代码是一些跨平台的支持，不同的平台上有不同的关键字和实现：

```
49 #ifdef _MSC_VER
50 #ifndef __inline__
51 #define __inline__ __inline
52 #endif
53 #if !defined(_STDINT) && !defined(uint32_t)
54     typedef unsigned __int8 uint8_t;
55     typedef __int16 int16_t;
56     typedef __int32 int32_t;
57     typedef unsigned __int16 uint16_t;
58 #endif
59 #endif
```

下面还是跨平台的代码，在不同的平台上进行不同的处理：

```
61 #if defined(__i386__)
...
82 #elif defined(__x86_64__)
...
98 #else
```

实际上这些代码段中定义了两个函数：`top_bit`和`bottom_bit`，用于找到一个字（Word）中的第一个1和最后一个1。这两个函数在*i386*和*x86_64*上都是用汇编代码实现的，速度比较快。如：

```
82 #elif defined(__x86_64__)
83     static __inline__ int top_bit(unsigned int bits) {
84         int res;
85
86         __asm__ __volatile__ (" movq $-1,%%rdx;\n" " bsrq %%rax,%%rdx;\n": "=d"(res)
87                               : "a" (bits));
88         return res;
89     }
```

如果看不懂汇编代码也不要紧，可以看一下它的C语言实现：

```
98 #else
99     static __inline__ int top_bit(unsigned int bits) {
100         int i;
101
102         if (bits == 0)
103             return -1;
104         i = 0;
105         if (bits & 0xFFFF0000) {
106             bits &= 0xFFFF0000;
107             i += 16;
108         }
109         if (bits & 0xFF00FF00) {
110             bits &= 0xFF00FF00;
111             i += 8;
112         }
113         if (bits & 0xF0F0F0F0) {
114             bits &= 0xF0F0F0F0;
115             i += 4;
116         }
117         if (bits & 0xCCCCCCCC) {
118             bits &= 0xCCCCCCCC;
119             i += 2;
120         }
121         if (bits & 0xAAAAAAAA) {
122             bits &= 0xAAAAAAAA;
123             i += 1;
124         }
125         return i;
126     }
```

好了，其实下面两个函数比较重要。前者将 L16 的一个抽样值（在此输入自动转换为 int 型，实际输入应该是 `int16_t` 或 `short`，会自动转换为 `int`）转换成 `ulaw` 的一个字节，后者则相反。这些函数为了效率起见都定义成内联（`inline`）的。

```
204     static __inline__ uint8_t linear_to_ulaw(int linear) {
...
241     static __inline__ int16_t ulaw_to_linear(uint8_t ulaw) {
```

当然，`alaw` 也有对应的函数，我们就不多说了，读者读读源代码自然能明白。

4.2 inet_pton.c

该文件只导入了一个函数 `switch_inet_pton`，用于将 ASCII 形式的 IP 地址转换成内部的二进制格式（如将 `192.168.0.1` 这样的表示转换成一个 32 位的整数）。该函数最终由 `switch_utils.h` 导出，主要是为了支持跨平台。

P to N 的意思在注释里也说得很明白：convert from **P**resentation format (which usually means ASCII printable) to **N**etwork format (which is usually some kind of binary format)。

4.3 switch.c

该文件的有些内容已经在 2.1.3 中讲过了，读者可以先翻回去看看。

本文件包含 `main` 函数，将会被编译成 `freeswitch` 可执行程序。

`handle_SIGILL` 是一个回调，当 `freeswitch` 进程收到某些信号时回调，并关闭 FreeSWITCH。这些信号是在 L1025 以及 L1067 ~ 1068 安装的。

```
84 static void handle_SIGILL(int sig)
85 {
86     int32_t arg = 0;
87     if (sig) {};
88     /* send shutdown signal to the freeswitch core */
89     switch_core_session_ctl(SCSC_SHUTDOWN, &arg);
90     return;
91 }
...
1204 if (nc && nf) {
1205     signal(SIGINT, handle_SIGILL);
```



```

1206     }
...
1067     signal(SIGILL, handle_SIGILL);
1068     signal(SIGTERM, handle_SIGILL);

```

当在命令行上执行 **freeswitch -stop** 时 (L744)，执行 **freeswitch_kill_background** 函数，查找是否有正在运行的 FreeSWITCH 进程，找到进程 PID，关闭 FreeSWITCH。

L101，初始化 **SWITCH_GLOBAL_dirs** 全局变量，里面存放了各种路径信息。L104，拼出 PID 文件的路径，L107 打开 PID 文件，L114 找到 PID。如果是在 UNIX 系统上，直接调用 **kill()** 函数 (L142)，在 Windows 系统上则用 Windows 上的相关函数处理 (L123 ~ 140)，这部分基本每一行都有注释，就不多解释了。

```

93  /* kill a freeswitch process running in background mode */
94  static int freeswitch_kill_background()
95  {
...
101      switch_core_set_globals();
...
104      switch_snprintf(path, sizeof(path), "%S%S%S", SWITCH_GLOBAL_dirs.run_dir, SWITCH_PATH_SEPARATOR,
↪ pfile);
...
107      if ((f = fopen(path, "r")) == 0) {
...
114      if (fscanf(f, "%d", (int *) (intptr_t) & pid) != 1) {
...
119      if (pid > 0) {
...
123 #ifdef WIN32
...
140 #else
141     /* for unix, send the signal to kill. */
142     kill(pid, SIGTERM);
143 #endif
...
744     else if (!strcmp(local_argv[x], "-stop")) {
745         do_kill = SWITCH_TRUE;
746     }
...
1024    if (do_kill) {
1025        return freeswitch_kill_background();
1026    }

```

L159，**ServiceCtrlHandler** 是一个回调函数，用于 Windows 环境。当 FreeSWITCH 以 Service 方式在后台运行时用到，主要用于捕捉到关闭信号时能停止 FreeSWITCH (L165)。

```
158 /* Handler function for service start/stop from the service */
159 void WINAPI ServiceCtrlHandler(DWORD control)
160 {
161     switch (control) {
162     case SERVICE_CONTROL_SHUTDOWN:
163     case SERVICE_CONTROL_STOP:
164         /* Shutdown freeswitch */
165         switch_core_destroy();
```

L198 向 Windows 服务注册了这个回调函数。

```
182 void WINAPI service_main(DWORD numArgs, char **args)
183 {
184     ...
198     hStatus = RegisterServiceCtrlHandler(service_name, &ServiceCtrlHandler);
185     ...
```

初始化一些全局变量。

```
203     switch_core_set_globals();
```

初始化 FreeSWITCH 并加载模块。

```
206     if (switch_core_init_and_modload(flags, SWITCH_FALSE, &err) != SWITCH_STATUS_SUCCESS) {
```

下面的 `check_fd` 函数，使用 poll（synchronous I/O multiplexing）检测当前文件描述符（实际是一个管道）是否可读，如果失败则返回负数，需要继续等待则返回 0，成功返回正数。

```
220 static int check_fd(int fd, int ms)
```

`daemonize` 是 UNIX 类系统上启动后面进程的标准方法，主要是通过调用 `fork()` 函数实现的。

```
1083 daemonize(do_wait ? fds : NULL);
```

一般情况下，如果执行 `freeswitch -nc`，则 FreeSWITCH 会启动到后面模式，L244 的 `fds` 会传入 NULL 指针，因而会执行到第 251 行执行 `fork()`。`fork()` 是一个很特别的函数，它会将当前进程复制出一个，也就是从 251 行起，操作系统上就会有二个进程执行同样的代码。在父进程中，`fork()` 会返回子进程的 PID（进程 ID），在子进程中，则返回 0。

```
244 static void daemonize(int *fds)
245 {
...
250     if (!fds) {
251         switch (fork()) {
```

花开两朵，各表一枝。且说父进程。如果 `fork()` 返回 `-1`，则失败，打印错误消息并退出（L255 ~ 256）。否则，也退出（L259），因为父进程没什么用了，这样，它会释放控制台。

```
254         case -1:
255             fprintf(stderr, "Error Backgrounding (fork)! %d - %s\n", errno, strerror(errno));
256             exit(EXIT_SUCCESS);
257             break;
258         default: /* parent process */
259             exit(EXIT_SUCCESS);
```

再说另一枝—子进程。子进程中，`fork()` 函数会返回 `0`，什么也不做，`break` 后继续往下执行。

```
252         case 0: /* child process */
253             break;
```

L262 调用 `setsid()` 创建一个新的进程组 session⁴。主要是为了防止终端关闭时连子进程一起关闭。

```
262         if (setsid() < 0) {
263             fprintf(stderr, "Error Backgrounding (setsid)! %d - %s\n", errno, strerror(errno));
264             exit(EXIT_SUCCESS);
265         }
```

L268 是一个 `switch_fork()`：

```
268     pid = switch_fork();
```

⁴The `setsid` function creates a new session. The calling process is the session leader of the new session, is the process group leader of a new process group and has no controlling terminal. The calling process is the only process in either the session or the process group. – `man setsid`.

该函数的实现在 `switch_core.c` 里。从代码中可以看出，它只是 `fork()` 函数的一个包装，会自动降低父进程的优先级。

```
SWITCH_DECLARE(pid_t) switch_fork(void)
{
    int i = fork();
    if (!i) set_low_priority();
    return i;
}
```

书接上文。L268 后又出现了两个并行的进程。原来的父进程已经退出了，原来的子进程现在成了父进程，并且又 `fork` 出了一个子进程。又开了两朵花。

这次，我们先看子进程。其中 `fds` 是一对文件描述符，它描述了一个管道，如果有的话就关掉该管道的读的一端（L273），并 `setsid`（L322）。

```
270     switch (pid) {
271     case 0:      /* child process */
272         if (fds) {
273             close(fds[0]);
274         }
275         break;
...
320
321     if (fds) {
322         setsid();
323     }
```

每个进程在开始的时候，会打开三个文件描述符—标准输入（STDIN），标准输出（STDOUT）和标准错误（STDERR），它们对应的值分别是 0、1、2，L324 ~ 344 会把它们重定向到 `/dev/null`（这是一个空设备）。否则的话，在后台进程里访问这三个文件会出错。

```
324     /* redirect std* to null */
325     fd = open("/dev/null", O_RDONLY);
326     switch_assert( fd >= 0 );
327     if (fd != 0) {
328         dup2(fd, 0);
329         close(fd);
330     }
```

至此，子进程。可以悠然地往下运行了。

再说父进程。如果 `fds` 为空指针，L318 就直接退出了，没事了，子进程长大了，脱离了父亲自己生存。

```

280     default:    /* parent process */
281         fprintf(stderr, "%d Backgrounding.\n", (int) pid);
...
318         exit(EXIT_SUCCESS);

```

如果 `fds` 非空，这些还有一些工作要处理。

什么情况下非空呢？FreeSWITCH 启动参数为 `-nc` 的时候会启动 `daemon` 模式，把进程启动到后台。但有时候，需要确认后台进程完全就绪后（全部加载完毕），父进程才退出。这时就需要使用 `-ncwait` 参数。这通常用在 FreeSWITCH 随系统一起启动的场景，某些其它依赖于 FreeSWITCH 的服务需要等待 FreeSWITCH 完全启动完后才能启动。

L295 是一个无限循环，一直读取文件描述符 `fds[0]` 等待子进程就绪。如果超过一定时间子进程还未就绪，就打印错误退出（310 ~ 312 行），否则，子进程一切正常，打印当前进程和子进程的 PID，退出（315 ~ 318 行），全权让位与子进程。

```

283     if (fds) {
...
295         do {
296             system_ready = check_fd(fds[0], 2000);
297
298             if (system_ready == 0) {
299                 printf("FreeSWITCH[%d] Waiting for background process pid:%d to be ready....\n",
↪ (int) getpid(), (int) pid);
300             }
301
302             } while (--sanity && system_ready == 0);
308
309             if (system_ready < 0) {
310                 printf("FreeSWITCH[%d] Error starting system! pid:%d\n", (int) getpid(), (int) pid);
311                 kill(pid, 9);
312                 exit(EXIT_FAILURE);
313             }
314
315             printf("FreeSWITCH[%d] System Ready pid:%d\n", (int) getpid(), (int) pid);
316         }
317
318         exit(EXIT_SUCCESS);
319     }

```

一次执行 `ncwait` 的日志供参考：

```
dujinfang@seven:~/ freeswitch -ncwait
24867 Backgrounding.
FreeSWITCH[24866] Waiting for background process pid:24867 to be ready....
FreeSWITCH[24866] Waiting for background process pid:24867 to be ready....
FreeSWITCH[24866] System Ready pid:24867
```

下面是一段很有趣的代码。这段代码能“再生（复活）”。L361，进程自我 `fork` 了一下，分裂为两个进程。其中，子进程只是在 402 行设置一参数，该参数只在 Linux 上有效，当父进程意外终止时，会给子进程发送 `SIGTERM` 消息。子进程继续往下执行。而父进程，则在 368 行执行 `waitpid` 等待子进程退出。当子进程退出时，父进程就继续往下执行，根据不同的参数，会通过 `execv` 或 `execvp` 等重新载入 `freeswitch` 可执行程序，并具在 396 行又跳回 360 行，重新 `fork` 一个子进程出来。这时候，子进程“重生”了。当然，这种情况适合系统升级的时候用，比如重新编译了更新了 FreeSWITCH。大多数时候，如果不更新 FreeSWITCH，可以不能执行 `exec` 系列的函数，而只是在 397 行跳回 361 行重新 `fork` 一下就好了。

```
355 static void reincarnate_protect(char **argv) {
...
360   refork:
361     if ((i=fork())) { /* parent */
...
367     rewait:
368       r = waitpid(i, &s, 0);
...
383       if (argv) {
384         if (execv(argv[0], argv) == -1) {
...
390         if (execvp(argv[0], argv) == -1) {
...
396         goto refork;
397       } else goto refork;
398     }
399     goto rewait;
400   } else { /* child */
401 #ifdef __linux__
402     prctl(PR_SET_PDEATHSIG, SIGTERM);
403 #endif
404   }
405 }
```

所以，在上面的代码段中，`fork` 后父进程不退出，它时刻监控子进程，一旦子进程退出（崩溃？），它就重新 `fork` 一下，重新启动一个新的子进程。

在启动 FreeSWITCH 时加上 `-reincarnate` 参数, 当 FreeSWITCH 意外终止时, 可以重新自启动, 以及及时恢复服务。

下面, 到 `main` 函数了。C 语言的程序都是从 `main` 函数开始的。第 L510 定义了默认的一些核心标志 (SCF = Switch Core Flags), 这些默认标志可以有其它参数改变。如 `SCF_USE_SQL` 默认使用 SQL 记录所有的 Channel 信息, 但是, 可以在启动时通过 `-nosql` 参数去掉该标志位。与此类似, `SCF_USE_AUTO_NAT` 会自动获取 NAT 信息 (通过 UPnP 之类的), `-nonat` 参数则禁用此功能。

```

479 int main(int argc, char *argv[])
480 {
...
510     switch_core_flag_t flags = SCF_USE_SQL | SCF_USE_AUTO_NAT | SCF_USE_NAT_MAPPING |
↪ SCF_CALIBRATE_CLOCK | SCF_USE_CLOCK_RT;

```

自 L538 开始, 判断命令行参数, 如 L543, 如果命令行参数是 `-help`, 则打印帮助信息 (L544) 并退出 (L545)。

```

538     for (x = 1; x < local_argc; x++) {
...
543         if (!strcmp(local_argv[x], "-help") || !strcmp(local_argv[x], "-h") || !strcmp(local_argv[x],
↪ "-?")) {
544             printf("%s\n", usage);
545             exit(EXIT_SUCCESS);

```

同样, 下列代码也是打印当前版本前退出:

```

676     else if (!strcmp(local_argv[x], "-version")) {
677         fprintf(stdout, "FreeSWITCH version: %s (%s)\n", switch_version_full(),
↪ switch_version_revision_human());

```

其它的参数我们就不多说了。如果读者知道各参数的功能, 对照源代码应该很容易读懂。

在非 Windows 平台上, 设置相关的信号回调, 这块我们在前面已经讲过了。

```

1067     signal(SIGILL, handle_SIGILL);
1068     signal(SIGTERM, handle_SIGILL);

```

在非 Windows 平台上, 如果使用了 `-ncwait`, 则在 L1071 创建一个管道。一个管道是一对文件描述符, 其中, `fds[0]` 用于读, `fds[1]` 用于写, 通过该管道可以在不同的进程间通信。

```
1069 #ifndef WIN32
1070     if (do_wait) {
1071         if (pipe(fds)) {
```

如果有 `-nc`，则 Windows 上使用 `FreeConsole`，其它 UNIX 类系统上使用我们前面讲的 `daemonize` 函数将进程启动到后台（除非有 `-reincarnate`，父进程到这里就结束了，不会再往下执行了）。注意，这里如果有 `-ncwait`，则会将上面创建的管道指针也传到 `daemonize` 函数中。`daemonize` 会 `fork` 子进程，子进程跟父进程就通过管道（`fds`）通信。

```
1078     if (nc) {
1079 #ifdef WIN32
1080         FreeConsole();
1081 #else
1082         if (!nf) {
1083             daemonize(do_wait ? fds : NULL);
1084         }
1085 #endif
```

“复活”这种本领是这样练成的：

```
1088     if (reincarnate)
1089         reincarnate_protect(reincarnate_reexec ? argv : NULL);
```

设置一些进程特权：

```
1092     if (switch_core_set_process_privileges() < 0) {
```

设置 FreeSWITCH 进程优先级：

```
1096     switch (priority) {
1097     case 2:
1098         set_realtime_priority();
1099         break;
1100     case 1:
1101         set_normal_priority();
1102         break;
```

```
1103     case -1:
1104         set_low_priority();
1105         break;
1106     default:
1107         set_auto_priority();
1108         break;
```

设置一些 Limits，如进程可以打开的文件句柄数，堆栈空间等：

```
1111     switch_core_setrlimits();
```

在 UNIX 类系统上，如果上面的一些特权代码部分需要 **root** 用户进行才有效（如普通用户可能没有权限设置 rlimit），所以，FreeSWITCH 进程应该以 **root** 用户来执行才能最好的应用系统资源。当特权代码执行完成后，最好是变成普通用户执行整个进程，这样，万一后面的代码有漏洞，也不至于让黑客取得 **root** 权限。

如果在启动时指定了 **-u freeswitch -g freeswitch**（表示以普通用户 **freeswitch** 的身份执行后面的代码），则 **runas_user** 和 **runas_group** 的值就是 **freeswitch**（L1115），**change_user_group**（**switch_core.c** 中定义）会调用 **setuid** 和 **setgid** 改变当前进程的用户和组权限。当然，Windows 上也有类似的机制（1123 ~ 1140 行），读者可以自行研究。

```
1114 #ifndef WIN32
1115     if (runas_user || runas_group) {
1116         if (change_user_group(runas_user, runas_group) < 0) {
1117             ...
1123         #else
1124             ...
1140         #endif
```

以上大部分都是一个应用程序需要考虑的跟操作系统相关的东西，下面，就是 FreeSWITCH 真正的代码了。

L1142 设置一些全局变量。取得当前进程的 PID（L1144，如果是后台启动模式，只有子进程会执行到这里，父进程在 **daemonize** 那一步已经退出了。接下来计算 PID 文件的路径（L1147）。

```
1142     switch_core_set_globals();
1143
1144     pid = getpid();
1145
```

```

1146     memset(pid_buffer, 0, sizeof(pid_buffer));
1147     switch_snprintf(pid_path, sizeof(pid_path), "%s%s%s", SWITCH_GLOBAL_dirs.run_dir,
↪ SWITCH_PATH_SEPARATOR, pfile);
1148     switch_snprintf(pid_buffer, sizeof(pid_buffer), "%d", pid);
1149     pid_len = strlen(pid_buffer);

```

创建一个内存池（L1151），并确保存放 PID 文件的路径存在（L1153，不存在则创建相关目录），锁定 PID 文件（L1170）并将当前的 PID 写入（L1179），初始化并加载模块（L1181），此后，FreeSWITCH 所有的功能都可以正常运行了。

```

1151     apr_pool_create(&pool, NULL);
1152
1153     switch_dir_make_recursive(SWITCH_GLOBAL_dirs.run_dir, SWITCH_DEFAULT_DIR_PERMS, pool);
...
1170     if (switch_file_lock(fd, SWITCH_FLOCK_EXCLUSIVE | SWITCH_FLOCK_NONBLOCK) != SWITCH_STATUS_SUCCESS)
↪ {
...
1179     switch_file_write(fd, pid_buffer, &pid_len);
...
1181     if (switch_core_init_and_modload(flags, nc ? SWITCH_FALSE : SWITCH_TRUE, &err) !=
↪ SWITCH_STATUS_SUCCESS) {

```

如果在 `-ncwait` 状态下，则向管道中写入一个“1”（L1191），表示子进程已启动完毕了。父进程应该能从管道的另一端读到（`check_fd` 函数，L236）。L1198 将管道写的一端关闭，以便读的一端读到 EOF。

```

1187     if (do_wait) {
1188         if (fds[1] > -1) {
1189             int i, v = 1;
1190
1191             if ((i = write(fds[1], &v, sizeof(v))) < 0) {
...
1198             close(fds[1]);
1199             fds[1] = -1;
1200         }
1201     }
1202 #endif

```

虽然我们前面讨论过很多进程，但实际上 FreeSWITCH 在运行时只有一个进程。FreeSWITCH 多任务是靠线程实现的。在 `switch_core_init_and_modload` 函数中，就启动了很多线程各司其职了。

接下来主线程进入无限循环，等待进程终止。如果有控制台，则在循环中等待键盘输入。

```
1208     switch_core_runtime_loop(nc);
```

如果进程终止，最后就是一些清理现场的工作了（1210 ~ 1215），包括清理内存以及删除 PID 文件。当然，最后还是有一次复活的机会（如执行 `fsctl shutdown restart` 时，通过 `execv` 或 `system` 重新加载 FreeSWITCH）。

```
1210     destroy_status = switch_core_destroy();
1211
1212     switch_file_close(fd);
1213     apr_pool_destroy(pool);
1214
1215     if (unlink(pid_path) != 0) {
1216     ...
1218
1219     if (destroy_status == SWITCH_STATUS_RESTART) {
1224         if (!argv || execv(argv[0], argv) == -1) {
1229             ret = system(buf);
```

收工。为节省篇幅，书中并未列出所有源码，请读者对照源代码阅读。

4.4 switch_apr.c

关于 APR，已经在第 2.1.2 节讲过不少了。该文件绝大部分内容都是将 `apr_` 相关的函数包装成了 `switch_` 函数，只是一种命名空间的转换，部分函数有一些增强。大部分函数也都在 `switch_apr.h` 中有注释，在此，我们就不多讲了。如果后面遇到相关函数，在必要的情况下我们再来解释。

4.5 switch_buffer.c

实现了一些简单的缓冲区读写函数。

缓冲区有固定（默认）、动态（`SWITCH_BUFFER_FLAG_DYNAMIC`）和分区（`SWITCH_BUFFER_FLAG_PARTITION`）三种类型。固定缓冲区是定长的，内存在内存池中申请，一旦申请不可改变大小；动态的缓冲区内存在堆上申请（用 `malloc`），可以根据需要自动增长（`realloc`）；分区型的缓冲区数据指针会指向现有的内存区域，不会自动申请内存。

`switch_buffer_get_head_pointer`，取得 Buffer 头指针的位置。

`switch_buffer_set_partition_data` 和 `switch_buffer_reset_partition_data`，仅对静态 Buffer 有效。用于设置和重设 Buffer 数据指针。

`switch_buffer_create_partition`，创建分区型缓冲区，数据指针指向现有内存区域。

`switch_buffer_create`，创建固定大小的缓冲区。

`switch_buffer_create_dynamic`，创建动态缓冲区，可以设置长度初始值和最大值。

`switch_buffer_add_mutex`，为缓冲区增加一个互斥，以便可以在多线程环境下锁定缓冲区。

`switch_buffer_lock`，锁定缓冲区。

`switch_buffer_unlock`，解锁。

`switch_buffer_len`，返回缓冲区长度。

`switch_buffer_freespace`，返回缓冲区剩余空间。

`switch_buffer_inuse`，返回缓冲区已用空间。

`switch_buffer_toss`，移动当前数据指针。

`switch_buffer_set_loops`，设置循环次数，用于环形缓冲区。

`switch_buffer_read_loop`，如果是环型缓冲区，读到结尾后可以从开头继续读。

`switch_buffer_read`，从缓冲区里读取数据，并复制（memcpy）到指定内存位置，同时移动缓冲区数据指针指向下一个待读的位置。

`switch_buffer_peek`，同上，但不移动数据指针，即下一次 `peak` 或 `read` 时还可以读到同样的内容。这时如果需要移动指针，则需要使用 `switch_buffer_toss`。

`switch_buffer_peek_zerocopy`，与 `peek` 类似，但不复制数据，而直接生成一个新的指针（*ptr）指向缓冲区数据区，避免内存拷贝带来的开销。在新指针使用期间应该避免对缓冲区进行写操作。

`switch_buffer_write`，不适用于分区型的缓冲区，向缓冲区写入数据。如果是动态缓冲区，会自动增长。

`switch_buffer_zero`，重置指针，一切都归零。

`switch_buffer_zwrite`，将缓冲区写入 0。

`switch_buffer_slide_write`，写入一个滑动窗口，缓冲区可以一直写入，但只有最后被写入的一个窗口大小的数据可以被读出。

`switch_buffer_destroy`，销毁缓冲区，释放内存。

4.6 switch_caller.c

以下代码基于 Git Commit Hash c6ece473。

该文件主要处理主叫的数据结构。其中，`switch_caller_profile_new`会创建一个`switch_caller_profile_t`的数据结构，称为 Caller Profile，用于描述主叫的信息，如主、被叫号码、网络地址、Dialplan Context 等。

在下面的代码片断中，L36 会传入一个内存池（`pool`）。L50，`switch_core_alloc`函数会在内存池中申请内存，该内存不需要显示的释放，只要最终可以释放整个的内存池，就不会有内存泄露，方便内存管理。该函数会自动将内容 `memset` 成 0。L51，调用 `switch_assert` 确保内存申请成功。如果申请不成功，程序就会崩溃。这是 FreeSWITCH 里常用的内存处理方式。理论上讲，如果申请内存不成功，应该打印一个友好的消息，或友好地退出程序或等待有足够的内存进行申请，但 FreeSWITCH 认为，内存都没有了，反正什么也干不了了，不如直接崩溃，处理起来还简单许多。

```

36 SWITCH_DECLARE(switch_caller_profile_t *) switch_caller_profile_new(switch_memory_pool_t *pool,
...
39             const char *caller_id_name,
40             const char *caller_id_number,
...
45             const char *source, const char *context, const char
↳ *destination_number)
46 {
47     switch_caller_profile_t *profile = NULL;
48     char uuid_str[SWITCH_UUID_FORMATTED_LENGTH + 1];
49
50     profile = switch_core_alloc(pool, sizeof(*profile));
51     switch_assert(profile != NULL);
52     memset(profile, 0, sizeof(*profile));

```

L54 创建一个 UUID 字符串，这是另一种使用内存的方法，`uuid_str`使用的是上面 L48 的静态内存地址，因此，只需要计算出字符串，填充这段内存即可。标准 UUID 的长度是 36 个字节，C 语言字符串需要有一个 `'\0'` 结尾，因此，L48 多申请一个字节。

但是，静态内存的生存期太短，该函数退出后就失效了，因此，L55 通过 `switch_core_strdup` 在内存池中复制了一份。

L66，设置默认的主叫号码，`SWITCH_DEFAULT_CLID_NUMBER`是一个宏，默认值是“0000000000”，这也就是有时候大家在 `originate` 的时候经常看到这个的主叫号码的原因。

L77，`profile_dup_clean`函数实际上是一个宏（在 `switch_caller.h` 中定义），也是在内存池中复制一份数据，`clean`会清洗掉一些特殊字符。

```

54     switch_uuid_str(uuid_str, sizeof(uuid_str));
55     profile->uuid_str = switch_core_strdup(pool, uuid_str);
...
65     if (zstr(caller_id_number)) {

```

```
66     caller_id_number = SWITCH_DEFAULT_CLID_NUMBER;
67 }
...
77 profile_dup_clean(caller_id_number, profile->caller_id_number, pool);
```

L99, `switch_set_flag` 也是一个宏, 它会将 `profile->flags` (一个 32 位整数) 的某一位 (这里是 `SWITCH_CPF_SCREEN`) 置 1, 相关于设置一个标志位, 备用。与之相对的有一个 `switch_clear_flag` 宏, 将某一标志位置 0。另外, 还有一个 `switch_test_flag`, 用于测试某一标志位。

L100, 记住内存池指针, 备用。L101 最终返回 `profile` 数据结构。

```
99 switch_set_flag(profile, SWITCH_CPF_SCREEN);
100 profile->pool = pool;
101 return profile;
102 }
```

L104, 顾名思义, 复制产生一份新的 Caller Profile。

```
104 SWITCH_DECLARE(switch_caller_profile_t *) switch_caller_profile_dup(switch_memory_pool_t *pool,
    switch_caller_profile_t *tcopy)
```

L178, 与 L104 类似, 但传入参数是一个 Session, 其实它是使用了当前 Session 的内存池 (L182)。

```
178 SWITCH_DECLARE(switch_caller_profile_t *) switch_caller_profile_clone(switch_core_session_t *session,
    switch_caller_profile_t *tcopy)
179 {
180     switch_memory_pool_t *pool;
182     pool = switch_core_session_get_pool(session);
184     return switch_caller_profile_dup(pool, tcopy);
185 }
```

L187, 取得 Caller Profile 中的一些参数, 若找不到则返回 `NULL`。

```
187 SWITCH_DECLARE(const char *) switch_caller_get_field_by_name(switch_caller_profile_t *caller_profile,
    const char *name)
188 {
189     if (!strcasecmp(name, "dialplan")) {
```

```

190     return caller_profile->dialplan;
191 }
...
198 if (!strcasecmp(name, "caller_id_number")) {
199     return caller_profile->caller_id_number;
200 }
...
305 return NULL;
306 }

```

L308, 将 Caller Profile 中的数据, 填充到 `event` 里。 `switch_event_t` 是 FreeSWITCH 中一个基本的数据结构, 它用于描述一个 Event。一个 Event 有一些头域和 Body 组成。头域就是一些 Key-Value 键值对。Event 使用起来比较灵活, 因为可以有无限的头域。

L313, `switch_snprintf` 类似于标准的 `snprintf` 函数, 用于格式化生成字符串。L314, 向 `event` 中从底部 (`SWITCH_STACK_BOTTOM`) 增加一个头域。对比 L317 中可以看出, 一个呼叫可以有一个方向 (Direction) 和一个逻辑方向 (Logical Direction), 前者是相对于 FreeSWITCH 的方向, 后者是逻辑上的呼叫方向, 我们将在后面再详细分析。

L444 用到了 `switch_test_flag` 函数。

```

308 SWITCH_DECLARE(void) switch_caller_profile_event_set_data(switch_caller_profile_t *caller_profile,
    const char *prefix, switch_event_t *event)
309 {
310     char header_name[1024];
311     switch_channel_timetable_t *times = NULL;
312
313     switch_snprintf(header_name, sizeof(header_name), "%s-Direction", prefix);
314     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, header_name,
        caller_profile->direction == SWITCH_CALL_DIRECTION_INBOUND ?
315         "inbound" : "outbound");
316
317     switch_snprintf(header_name, sizeof(header_name), "%s-Logical-Direction", prefix);
318     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, header_name,
        caller_profile->logical_direction == SWITCH_CALL_DIRECTION_INBOUND ?
319         "inbound" : "outbound");
...
443     switch_snprintf(header_name, sizeof(header_name), "%s-Privacy-Hide-Number", prefix);
444     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, header_name,
        switch_test_flag(caller_profile, SWITCH_CPF_HIDE_NUMBER) ? "true" : "false");
445 }

```

L447 用于 Clone 一个 Caller Extension, 从内存池中申请内存。一个 Caller Extension 对应 Dialplan 中的一个 Extension (分支)。

```
447 SWITCH_DECLARE(switch_status_t) switch_caller_extension_clone(switch_caller_extension_t **new_ext,
                                                                    switch_caller_extension_t *orig,
448                                                                    switch_memory_pool_t *pool)
```

L497 创建一个 Caller Extension，从 Session 的内存池中申请内存。

```
497 SWITCH_DECLARE(switch_caller_extension_t *) switch_caller_extension_new(switch_core_session_t *session,
                                                                    const char *extension_name,
498                                                                    const char *extension_number)
```

L512 在 Caller Extension 上增加一个 Application，使用类似 `printf` 的语法格式 `wx.qq.com` 参数。当有电话路由到这个分支时，就可以执行对应的 Application。

```
512 SWITCH_DECLARE(void) switch_caller_extension_add_application_printf(switch_core_session_t *session,
513                                                                    switch_caller_extension_t *caller_extension, const char
↳ *application_name,
514                                                                    const char *fmt, ...)
```

L534 与 L512 类似，只是无须格式化参数。所有的 Application 都会追加到 `caller_extension->applications` 链表尾部（L554 ~ 561）。

```
534 SWITCH_DECLARE(void) switch_caller_extension_add_application(switch_core_session_t *session,
535                                                                    switch_caller_extension_t *caller_extension, const char
↳ *application_name,
536                                                                    const char *application_data)
537 {
...
554     if (!caller_extension->applications) {
555         caller_extension->applications = caller_application;
556     } else if (caller_extension->last_application) {
557         caller_extension->last_application->next = caller_application;
558     }
559
560     caller_extension->last_application = caller_application;
561     caller_extension->current_application = caller_extension->applications;
...
```

读者这里读者大体就会想象到，当有呼叫到达 FreeSWITCH 时，会产生一个 Caller Profile 用于描述主叫，然后查找 Dialplan，找到一个 Caller Extension，Extension 里面会有很多 Action，每个 Action 对应一个 Application，后面我们将会看到这些函数都是哪里执行的。

4.7 switch_channel.c

这是一个很重要的文件，在 FreeSWITCH 里，所有的呼叫都是 Channel，一个 Channel，就是 FreeSWITCH 中的一条腿。

L38 ~ L41 定义了一个原因 (Cause) 表结构。L56 ~ L128 定义了该表，实际上是字符串和数字常量的对应关系（可以看到一些常见的挂机原因，如 `USER_BUSY`、`NO_ANSWER` 等），L189 和 L204 则分别定义了两个函数在两者间转换（其中“2”意为“to”，“a2b”即“a to b”，央视的广播员将“B2B”念成“B 二 B”说明他们不是程序员:D）。

```

38 struct switch_cause_table {
39     const char *name;
40     switch_call_cause_t cause;
41 };
...
56 static struct switch_cause_table CAUSE_CHART[] = {
57     {"NONE", SWITCH_CAUSE_NONE},
58     {"UNALLOCATED_NUMBER", SWITCH_CAUSE_UNALLOCATED_NUMBER},
59     {"NO_ROUTE_TRANSIT_NET", SWITCH_CAUSE_NO_ROUTE_TRANSIT_NET},
60     {"NO_ROUTE_DESTINATION", SWITCH_CAUSE_NO_ROUTE_DESTINATION},
61     {"CHANNEL_UNACCEPTABLE", SWITCH_CAUSE_CHANNEL_UNACCEPTABLE},
62     {"CALL_AWARDED_DELIVERED", SWITCH_CAUSE_CALL_AWARDED_DELIVERED},
63     {"NORMAL_CLEARING", SWITCH_CAUSE_NORMAL_CLEARING},
64     {"USER_BUSY", SWITCH_CAUSE_USER_BUSY},
65     {"NO_USER_RESPONSE", SWITCH_CAUSE_NO_USER_RESPONSE},
66     {"NO_ANSWER", SWITCH_CAUSE_NO_ANSWER},
...
128 };

189 SWITCH_DECLARE(const char *) switch_channel_cause2str(switch_call_cause_t cause)
204 SWITCH_DECLARE(switch_call_cause_t) switch_channel_str2cause(const char *str)

```

```

43 typedef struct switch_device_state_binding_s {
44     switch_device_state_function_t function;
45     void *user_data;
46     struct switch_device_state_binding_s *next;
47 } switch_device_state_binding_t;

```

全局变量统一定义在 `globals` 里，相当于一个命名空间。

```

49 static struct {
50     switch_memory_pool_t *pool;

```

```
51     switch_hash_t *device_hash;
52     switch_mutex_t *device_mutex;
53     switch_device_state_binding_t *device_bindings;
54 } globals;
```

```
130 typedef enum {
131     OCF_HANGUP = (1 << 0)
132 } opaque_channel_flag_t;
133
134 typedef enum {
135     LP_NEITHER,
136     LP_ORIGINATOR,
137     LP_ORIGINATEE
138 } switch_originator_type_t;
```

L140 定义了 `switch_channel` 结构，为了便于阅读，我们全文列在这里。该结构是私有的，在本文之外无法访问。因而，如果要所有需要访问的 Channel 内部属性，都对应一个函数，如 L184、L225、L231 等。

```
140 struct switch_channel {
141     char *name;
142     switch_call_direction_t direction;
143     switch_call_direction_t logical_direction;
144     switch_queue_t *dtmf_queue;
145     switch_queue_t *dtmf_log_queue;
146     switch_mutex_t *dtmf_mutex;
147     switch_mutex_t *flag_mutex;
148     switch_mutex_t *state_mutex;
149     switch_mutex_t *thread_mutex;
150     switch_mutex_t *profile_mutex;
151     switch_core_session_t *session;
152     switch_channel_state_t state;
153     switch_channel_state_t running_state;
154     switch_channel_callstate_t callstate;
155     uint32_t flags[CF_FLAG_MAX];
156     uint32_t caps[CC_FLAG_MAX];
157     uint8_t state_flags[CF_FLAG_MAX];
158     uint32_t private_flags;
159     switch_caller_profile_t *caller_profile;
160     const switch_state_handler_table_t *state_handlers[SWITCH_MAX_STATE_HANDLERS];
161     int state_handler_index;
162     switch_event_t *variables;
163     switch_event_t *scope_variables;
164     switch_hash_t *private_hash;
```

```

165     switch_hash_t *app_flag_hash;
166     switch_call_cause_t hangup_cause;
167     int vi;
168     int event_count;
169     int profile_index;
170     opaque_channel_flag_t opaque_flags;
171     switch_originator_type_t last_profile_type;
172     switch_caller_extension_t *queued_extension;
173     switch_event_t *app_list;
174     switch_event_t *api_list;
175     switch_event_t *var_list;
176     switch_hold_record_t *hold_record;
177     switch_device_node_t *device_node;
178     char *device_id;
179 };
...
184 SWITCH_DECLARE(switch_hold_record_t *) switch_channel_get_hold_record(switch_channel_t *channel)
185 {
186     return channel->hold_record;
187 }
...
225 SWITCH_DECLARE(switch_call_cause_t) switch_channel_get_cause(switch_channel_t *channel)
226 {
227     return channel->hangup_cause;
228 }
...
231 SWITCH_DECLARE(switch_call_cause_t *) switch_channel_get_cause_ptr(switch_channel_t *channel)
232 {
233     return &channel->hangup_cause;
234 }

```

L237 ~ L252, 呼叫状态表, 一个 Channel 有以下的呼叫状态。

```

237 struct switch_callstate_table {
238     const char *name;
239     switch_channel_callstate_t callstate;
240 };
241 static struct switch_callstate_table CALLSTATE_CHART[] = {
242     {"DOWN", CCS_DOWN},
243     {"DIALING", CCS_DIALING},
244     {"RINGING", CCS_RINGING},
245     {"EARLY", CCS_EARLY},
246     {"ACTIVE", CCS_ACTIVE},
247     {"HELD", CCS_HELD},
248     {"RING_WAIT", CCS_RING_WAIT},
249     {"HANGUP", CCS_HANGUP},

```

```

250     {"UNHELD", CCS_UNHELD},
251     {NULL, 0}
252 };

```

L254 ~ L267, 设备状态表。同一个“设备”（终端），可能有多个 Channel。这里定义了“设备”的状态。比较典型的是 **ACTIVE_MULTI** (L262)，表示该设备上有多个活动的呼叫。

```

254 struct switch_device_state_table {
255     const char *name;
256     switch_device_state_t device_state;
257 };
258 static struct switch_device_state_table DEVICE_STATE_CHART[] = {
259     {"DOWN", SDS_DOWN},
260     {"RINGING", SDS_RINGING},
261     {"ACTIVE", SDS_ACTIVE},
262     {"ACTIVE_MULTI", SDS_ACTIVE_MULTI},
263     {"HELD", SDS_HELD},
264     {"UNHELD", SDS_UNHELD},
265     {"HANGUP", SDS_HANGUP},
266     {NULL, 0}
267 };

```

下面的函数通过查表的方法在字符串和内部状态间转换。

```

302 SWITCH_DECLARE(const char *) switch_channel_callstate2str(switch_channel_callstate_t callstate)
317 SWITCH_DECLARE(const char *) switch_channel_device_state2str(switch_device_state_t device_state)
333 SWITCH_DECLARE(switch_channel_callstate_t) switch_channel_str2callstate(const char *str)

```

接下来我们看一个宏 **switch_channel_set_callstate**，通过这种宏定义方式，可以传入函数调用时的真正文件和行号，以便打印到日志里。

具体的宏定义是在 **switch_channel.h** 中定义的，如下：

```

660 #define switch_channel_set_callstate(channel, state)
        switch_channel_perform_set_callstate(channel, state, __FILE__, __SWITCH_FUNC__, __LINE__)

```

后面我们还会看到很多类似的宏定义，都是扩展到 **perform** 版的实际函数。

L270，就是一个 **perform** 版的函数，它用于设置 Channel 当前的状态。我们看到，除了简单设置 **channel->callstate** 的值以外，它还在 L288 创建了一个 Event。Event 是 FreeSWITCH 内部的

异步通信机制，其它地方如果订阅了这种类型事件，就可以收到这个事件。也就是说，每个 Channel 状态变化都会发送一个事件。如果 Event 成功创建，后面加入几个头域（L289 ~ L290），接着 L291 设置从当前 `channel` 上获取一些通用的数据，填充到 `event` 里，并于 L292 将该事件发送出去。

```

270 SWITCH_DECLARE(void) switch_channel_perform_set_callstate(switch_channel_t *channel,
                    switch_channel_callstate_t callstate,
271                    const char *file, const char *func, int line)
272 {
273     switch_event_t *event;
274     switch_channel_callstate_t o_callstate = channel->callstate;
275
276     if (o_callstate == callstate || o_callstate == CCS_HANGUP) return;
277
278     channel->callstate = callstate;
279     if (channel->device_node) {
280         channel->device_node->callstate = callstate;
281     }
282     switch_log_printf(SWITCH_CHANNEL_ID_LOG, file, func, line, switch_channel_get_uuid(channel),
↪ SWITCH_LOG_DEBUG,
283                     "(%s) Callstate Change %s -> %s\n", channel->name,
284                     switch_channel_callstate2str(o_callstate),
↪ switch_channel_callstate2str(callstate));
285
286     switch_channel_check_device_state(channel, channel->callstate);
287
288     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_CALLSTATE) == SWITCH_STATUS_SUCCESS) {
289         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Original-Channel-Call-State",
                    switch_channel_callstate2str(o_callstate));
290         switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Channel-Call-State-Number", "%d",
↪ callstate);
291         switch_channel_event_set_data(channel, event);
292         switch_event_fire(&event);
293     }
294 }

```

L296，获取当关的呼叫状态。

```

296 SWITCH_DECLARE(switch_channel_callstate_t) switch_channel_get_callstate(switch_channel_t *channel)
297 {
298     return channel->callstate;
299 }

```

L353 是音频同步的函数。这里，它使用了另一种通信机制—消息（Message）。它的实现机制是

首先产生一个 Message (L358)，然后设置相关的参数 (L359 ~ 364)，最后将该 Message 推入一个队列 (L366, 这里使用的是 Session 内部的消息队列)。

在 Session 内部, 会有一个线程不断检查该消息队列, 如果有相关的消息到来, 则进行相应的动作 (一般是清掉缓存里已收到的数据包), 这是后话, 暂且不提。

```
353 SWITCH_DECLARE(void) switch_channel_perform_audio_sync(switch_channel_t *channel, const char *file,
↳ const char *func, int line)
354 {
355     if (switch_channel_media_up(channel)) {
356         switch_core_session_message_t *msg = NULL;
357
358         msg = switch_core_session_alloc(channel->session, sizeof(*msg));
359         MESSAGE_STAMP_FFL(msg);
360         msg->message_id = SWITCH_MESSAGE_INDICATE_AUDIO_SYNC;
361         msg->from = channel->name;
362         msg->_file = file;
363         msg->_func = func;
364         msg->_line = line;
365
366         switch_core_session_queue_message(channel->session, msg);
367     }
368 }
```

使用 Message 通信通常比通过 Event 通信及时一些, 而且后者也可以发到 FreeSWITCH 外面去, 但前者, 仅限于 FreeSWITCH 内部线程间通信。

视频的同步跟音频差不多, 不同的是 L379 的消息类型以及 L385 会发送一个 Refresh 请求给对方的 UA, 以便对方产生一个关键帧。

```
371 SWITCH_DECLARE(void) switch_channel_perform_video_sync(switch_channel_t *channel, const char *file,
↳ const char *func, int line)
372 {
...
379     msg->message_id = SWITCH_MESSAGE_INDICATE_VIDEO_SYNC;
...
385     switch_core_session_request_video_refresh(channel->session);
386     switch_core_session_queue_message(channel->session, msg);
388 }
```

L392、L401 返回 Q850 定义的消息。

```

392 SWITCH_DECLARE(switch_call_cause_t) switch_channel_cause_q850(switch_call_cause_t cause)
...
401 SWITCH_DECLARE(switch_call_cause_t) switch_channel_get_cause_q850(switch_channel_t *channel)

```

L406, 获取 Channel 里的时间表, 包含应答、挂机时间等。

```

406 SWITCH_DECLARE(switch_channel_timetable_t *) switch_channel_get_timetable(switch_channel_t *channel)

```

L419, 设置 Channel 的方向, 仅当当前线程与 Session 的线程不是一个线程时 (L421) 才会设置。

```

419 SWITCH_DECLARE(void) switch_channel_set_direction(switch_channel_t *channel, switch_call_direction_t
↳ direction)
420 {
421     if (!switch_core_session_in_thread(channel->session)) {
422         channel->direction = channel->logical_direction = direction;
423         switch_channel_set_variable(channel, "direction",
            switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_OUTBOUND ? "outbound" :
↳ "inbound");
424     }
425 }

```

L437, 创建一个 Channel, 在内存池中申请内存, 并创建相关的队列 (如 L448)、Mutex (如 L451), 初始化相关数据 (L456) 等。

L445, 创建了一个 Event 数据结构, 实际上是为了存通道变量。

```

437 SWITCH_DECLARE(switch_status_t) switch_channel_alloc(switch_channel_t **channel,
            switch_call_direction_t direction, switch_memory_pool_t *pool)
438 {
439     switch_assert(pool != NULL);
440
441     if (((*channel) = switch_core_alloc(pool, sizeof(switch_channel_t))) == 0) {
442         return SWITCH_STATUS_MEMERR;
443     }
444
445     switch_event_create_plain(&(*channel)->variables, SWITCH_EVENT_CHANNEL_DATA);
446
447     switch_core_hash_init(&(*channel)->private_hash);
448     switch_queue_create(&(*channel)->dtmf_queue, SWITCH_DTMF_LOG_LEN, pool);
...

```

```
451     switch_mutex_init(&(*channel)->dtmf_mutex, SWITCH_MUTEX_NESTED, pool);
456     (*channel)->hangup_cause = SWITCH_CAUSE_NONE;
...
461     return SWITCH_STATUS_SUCCESS;
462 }
```

FreeSWITCH 是多线程的，因此，不同的线程间访问共享资源（临界区）时就需要协调，协调通过互斥（Mutex）实现，实现方式是访问临界区前先对 Mutex 加锁，因而，一般每个临界区都对应一个 Mutex，下面的函数对 `dtmf_mutex` 加锁。

```
464 SWITCH_DECLARE(switch_status_t) switch_channel_dtmf_lock(switch_channel_t *channel)
465 {
466     return switch_mutex_lock(channel->dtmf_mutex);
467 }
```

加锁时，如果 Mutex 已被其它访问者锁定，则会阻塞并一直等待。为了避免等待，下面函数（`try` 版）将在无法锁定时立即返回，以便可以执行一些其它操作，避免阻塞。

```
469 SWITCH_DECLARE(switch_status_t) switch_channel_try_dtmf_lock(switch_channel_t *channel)
470 {
471     return switch_mutex_trylock(channel->dtmf_mutex);
472 }
```

临界区使用完毕后需要记着解锁，否则别人无法再进入临界区。

```
474 SWITCH_DECLARE(switch_status_t) switch_channel_dtmf_unlock(switch_channel_t *channel)
```

如果在一个 Channel 生存期间收到 DTMF，FreeSWITCH 会先将 DTMF 存到一个队列里（`dtmf_queue`），下列函数检测队列中是否有 DTMF。

```
479 SWITCH_DECLARE(switch_size_t) switch_channel_has_dtmf(switch_channel_t *channel)
480 {
481     switch_size_t has;
482
483     switch_mutex_lock(channel->dtmf_mutex);
484     has = switch_queue_size(channel->dtmf_queue);
485     switch_mutex_unlock(channel->dtmf_mutex);
```



```

486
487     return has;
488 }

```

下列函数用于发送 DTMF，发送方式是将 DTMF 推入一个发送队列，等待时机发送出去。L499，要加锁。L506，测试我们是否能正确接收 DTMF。L514，如果不是敏感数据（如密码等），则打印到 Log 里。L539，多次尝试把 DTMF 推到 `dtmf_queue` 队列里。L557，对当前的 RTP Socket 做一个 Break 操作（有些情况下，RTP 是阻塞读取的，这时需要暂时停止阻塞，以便能将 DTMF 发送出去，后话）。

```

490 SWITCH_DECLARE(switch_status_t) switch_channel_queue_dtmf(switch_channel_t *channel, const
↳ switch_dtmf_t *dtmf)
491 {
...
499     switch_mutex_lock(channel->dtmf_mutex);
500     new_dtmf = *dtmf;
...
506     if ((status = switch_core_session_recv_dtmf(channel->session, dtmf) != SWITCH_STATUS_SUCCESS)) {
507         goto done;
508     }
509
510     if (is_dtmf(new_dtmf.digit)) {
...
514         if (!sensitive) {
515             switch_log_printf(SWITCH_CHANNEL_CHANNEL_LOG(channel), SWITCH_LOG_INFO, "RECV DTMF %c:
↳ %d\n", new_dtmf.digit, new_dtmf.duration);
516         }
...
535         switch_zmalloc(dt, sizeof(*dt));
536         *dt = new_dtmf;
537
538
539         while (switch_queue_trypush(channel->dtmf_queue, dt) != SWITCH_STATUS_SUCCESS) {
...
548         }
549     }
550
553 done:
555     switch_mutex_unlock(channel->dtmf_mutex);
557     switch_core_media_break(channel->session, SWITCH_MEDIA_TYPE_AUDIO);

```

同上，只是 DTMF 是字符串。

```

562 SWITCH_DECLARE(switch_status_t) switch_channel_queue_dtmf_string(switch_channel_t *channel, const char
↳ *dtmf_string)
...
610         if (switch_channel_queue_dtmf(channel, &dtmf) == SWITCH_STATUS_SUCCESS) {

```

从 Channel 的 DTMF 队列中读取 DTMF (L633)。读到后, 在 L660 创建一个 Event, 并发送出去 (L684 ~ L687)。发送的方式有两种, 如果 Channel 有 `CF_DIVERT_EVENTS` 标志, 则会在 L685 推入 Channel 的事件队列 (TODO), 否则, 会直接发送出去。

L694 与 `switch_channel_dequeue_dtmf` 类似, 只是返回字符串。

```

623 SWITCH_DECLARE(switch_status_t) switch_channel_dequeue_dtmf(switch_channel_t *channel, switch_dtmf_t
↳ *dtmf)
624 {
...
631     switch_mutex_lock(channel->dtmf_mutex);
633     if (switch_queue_peek(channel->dtmf_queue, &pop) == SWITCH_STATUS_SUCCESS) {
...
658     switch_mutex_unlock(channel->dtmf_mutex);

660     if (!sensitive && status == SWITCH_STATUS_SUCCESS && switch_event_create(&event, SWITCH_EVENT_DTMF)
↳ == SWITCH_STATUS_SUCCESS) {
...
663         switch_event_add_header(event, SWITCH_STACK_BOTTOM, "DTMF-Digit", "%c", dtmf->digit);
...
684         if (switch_channel_test_flag(channel, CF_DIVERT_EVENTS)) {
685             switch_core_session_queue_event(channel->session, &event);
686         } else {
687             switch_event_fire(&event);
688         }
...
694 SWITCH_DECLARE(switch_size_t) switch_channel_dequeue_dtmf_string(switch_channel_t *channel, char
↳ *dtmf_str, switch_size_t len)

```

L709, 很明显, 读出所有 DTMF 并丢弃。

```

709 SWITCH_DECLARE(void) switch_channel_flush_dtmf(switch_channel_t *channel)

```

L723, 销毁 Channel, 释放内存。

```

723 SWITCH_DECLARE(void) switch_channel_uninit(switch_channel_t *channel)

```

L747, Channel 初始化。一个 Channel 对应一个 Session, 把它们关联起来。

```
747 SWITCH_DECLARE(switch_status_t) switch_channel_init(switch_channel_t *channel, switch_core_session_t
↪ *session,
748                                     switch_channel_state_t state, switch_channel_flag_t
↪ flag)
749 {
750     switch_assert(channel != NULL);
751     channel->state = state;
752     switch_channel_set_flag(channel, flag);
753     channel->session = session;
754     channel->running_state = CS_NONE;
755     return SWITCH_STATUS_SUCCESS;
756 }
```

L758, Presense。L776 可以看到, Channel 上 `presence_id` 这个通道变量是很有用的, 可以在 XML 配置文件的 `dial-string` 中看到它。L785 创建一个事件, 并发送出去 (L837)。

```
758 SWITCH_DECLARE(void) switch_channel_perform_presence(switch_channel_t *channel,
               const char *rpid, const char *status, const char *id,
759               const char *file, const char *func, int line)
760 {
761     switch_event_t *event;
762     switch_event_types_t type = SWITCH_EVENT_PRESENCE_IN;
763     ...
776     id = switch_channel_get_variable(channel, "presence_id");
764     ...
785     if (switch_event_create(&event, type) == SWITCH_STATUS_SUCCESS) {
765     ...
837         switch_event_fire(&event);
```

修改 Channel 的 Hold 状态 `on` 和 `off` (L849 ~ L852), 并发送一个事件 (L855 ~ 857)。注意 L863 检查如果有 `flip_record_on_hold` 这个通道变量的话, 如果当前通道正在录音, 则通过 `switch_core_session_get_partner` 函数 (L865) 查找是否有一个与它 Bridge 的另一要腿, 录音是用 Media Bug 实现的, `switch_core_media_bug_transfer_recordings` 可以将 Media Bug 从一条腿转移到另外一条腿 (L866)。

用 `switch_core_session_get_partner` 获取到的 Session 会自动加锁, 所以, 用完要记得解锁 (L867)。

```
841 SWITCH_DECLARE(void) switch_channel_mark_hold(switch_channel_t *channel, switch_bool_t on)
842 {
```

```

...
849     if (on) {
850         switch_channel_set_flag(channel, CF_LEG_HOLDING);
851     } else {
852         switch_channel_clear_flag(channel, CF_LEG_HOLDING);
853     }
854
855     if (switch_event_create(&event, on ? SWITCH_EVENT_CHANNEL_HOLD : SWITCH_EVENT_CHANNEL_UNHOLD) ==
↪ SWITCH_STATUS_SUCCESS) {
856         switch_channel_event_set_data(channel, event);
857         switch_event_fire(&event);
858     }
859
860 end:
862     if (on) {
863         if (switch_true(switch_channel_get_variable(channel, "flip_record_on_hold"))) {
864             switch_core_session_t *other_session;
865             if (switch_core_session_get_partner(channel->session, &other_session) ==
↪ SWITCH_STATUS_SUCCESS) {
866                 switch_core_media_bug_transfer_recordings(channel->session, other_session);
867                 switch_core_session_rwlock(other_session);

```

取得当前 Channel 的保持音乐。如果保持音乐是一个变量（如`hold_music`），则会通过 `switch_channel_expand_variables` 将变量进行扩展（L883），转换为真正的音乐路径（如 `local_stream://moh`）。扩展后返回的内存是动态申请的，因而 L886 行在 Session 的内存池中又复制了一份，并于 L887 释放这段内存，以方便内存生命周期管理。

```

874 SWITCH_DECLARE(const char *) switch_channel_get_hold_music(switch_channel_t *channel)
875 {
876     const char *var;
877
878     if (!(var = switch_channel_get_variable(channel, SWITCH_TEMP_HOLD_MUSIC_VARIABLE))) {
879         var = switch_channel_get_variable(channel, SWITCH_HOLD_MUSIC_VARIABLE);
880     }
881
882     if (!zstr(var)) {
883         char *expanded = switch_channel_expand_variables(channel, var);
884
885         if (expanded != var) {
886             var = switch_core_session_strdup(channel->session, expanded);
887             free(expanded);
888         }
889     }
892     return var;
893 }

```

同上，取得另一条腿上的 Hold Music。

```
895 SWITCH_DECLARE(const char *) switch_channel_get_hold_music_partner(switch_channel_t *channel)
```

Scope Variables 是仅作用于当前 Channel 的通道变量，通常在呼叫字符串里在方括号 “[]” 里表示。

```
908 SWITCH_DECLARE(void) switch_channel_set_scope_variables(switch_channel_t *channel, switch_event_t
↳ **event)
926 SWITCH_DECLARE(switch_status_t) switch_channel_get_scope_variables(switch_channel_t *channel,
↳ switch_event_t **event)
```

获取一个通道变量，并在 Session 的内存池中复制一份。先检查 Scope Variable 里有没有 (L961)，再检查 Caller Profile 里有没有 (L972 ~ 984)，然后检查全局变量 (L985) 里有没有。

```
953 SWITCH_DECLARE(const char *) switch_channel_get_variable_dup(switch_channel_t *channel,
                                                                    const char *varname, switch_bool_t dup, int idx)
954 {
...
960     if (!zstr(varname)) {
961         if (channel->scope_variables) {
962             switch_event_t *ep;
963
964             for (ep = channel->scope_variables; ep; ep = ep->next) {
965                 if ((v = switch_event_get_header_idx(ep, varname, idx))) {
966                     break;
967                 }
968             }
969         }
970
971         if (!v && (!channel->variables || !(v = switch_event_get_header_idx(channel->variables, varname,
↳ idx)))) {
972             switch_caller_profile_t *cp = switch_channel_get_caller_profile(channel);
...
984             if (!cp || !(v = switch_caller_get_field_by_name(cp, varname))) {
985                 if ((vdup = switch_core_get_variable_pdup(varname,
↳ switch_core_session_get_pool(channel->session)))) {
986                     v = vdup;
987                 }
988             }
989         }
990     }
```

与上面的类似，取得另一条腿上的通道变量。

```
1005 SWITCH_DECLARE(const char *) switch_channel_get_variable_partner(switch_channel_t *channel, const char
↪ *varname)
```

以下两个函数用于遍历所有 Channel Variable。

```
1029 SWITCH_DECLARE(void) switch_channel_variable_last(switch_channel_t *channel)
1040 SWITCH_DECLARE(switch_event_header_t *) switch_channel_variable_first(switch_channel_t *channel)
```

以上函数典型的使用方式如下：

```
switch_event_header_t *hi = switch_channel_variable_first(channel);

if (!hi) return;

for (; hi; hi = hi->next) {
    // ...
}
switch_channel_variable_last(channel);
```

设置和读取 Channel 私有的一些信息，就是往 Channel 的一个私有哈希表中插入和读取数据，与通道变量不同的是，通道变量只能存取字符串值，而私有信息可以存取任何类型的指针（void *）。从 L1058 和 L1066 可以看出，在操作哈希表时，都需要锁定一个 Mutex。

```
1055 SWITCH_DECLARE(switch_status_t) switch_channel_set_private(switch_channel_t *channel,
                                                                const char *key, const void *private_info)
1056 {
1057     switch_assert(channel != NULL);
1058     switch_core_hash_insert_locked(channel->private_hash, key, private_info, channel->profile_mutex);
1059     return SWITCH_STATUS_SUCCESS;
1060 }
1061
1062 SWITCH_DECLARE(void *) switch_channel_get_private(switch_channel_t *channel, const char *key)
1063 {
1064     void *val;
1065     switch_assert(channel != NULL);
1066     val = switch_core_hash_find_locked(channel->private_hash, key, channel->profile_mutex);
1067     return val;
1068 }
```

L1070 与 L1062 类似，只是获取另一条腿的私有数据。

```
1070 SWITCH_DECLARE(void) switch_channel_get_private_partner(switch_channel_t
channel, const char *key)
```

设置和读取 Channel 的名字。

```
1088 SWITCH_DECLARE(switch_status_t) switch_channel_set_name(switch_channel_t *channel, const char *name)
1110 SWITCH_DECLARE(char *) switch_channel_get_name(switch_channel_t *channel)
```

设置一个 Channel 的 Caller Profile 变量。

```
1116 SWITCH_DECLARE(switch_status_t) switch_channel_set_profile_var(switch_channel_t *channel,
                           const char *name, const char *val)
1117 {
...
1141     if (!strcasecmp(name, "dialplan")) {
1142         channel->caller_profile->dialplan = v;
...
1147     } else if (!strcasecmp(name, "caller_id_number")) {
1148         channel->caller_profile->caller_id_number = v;
```

设置通道变量，该函数会将 A-leg 上的通道相关变量（L1214，`export_varname`）`export` 到 B-leg 上。

L1218，获取一个以逗号分隔的变量列表，该列表列出都有哪些变量需要 `export` 到 B-leg 上。L1219，将变量复制一份，备用。

L1226 ~ L1229，如果有 `var_event` 的话，将里面的 `export_varname` 替换为新的 `export_vars`。

L1231 ~ L1233，如果有 B-leg 的话，将通道变量设置到 B-leg 上。

L1235，将字符串换逗号切开。由于该函数在切割过程中要破坏内存中原始的数据，所以 L1219 行再复制了一份，以免影响原来的内容。切割完成后 `argc` 就是切开的段数，每一段字符串指针都存到 `argv[]` 数组中。

接着 L1238 会遍历所有字符串，如果字符串以 “`nolocal:`”（L1242）或 “`_nolocal_`”（L1244）开头（前者包含冒号，如果该数据出现在 XML 中，冒号会有特殊含义，所以应该用者后，这也是为什么有注释 “remove this later?”），则说明该变量仅会设置到 B-leg 上（在 A-leg 上不存在），需要移动指针跳过这 8 个（前者，L1243）或 9（后者，L1245）个字符。

```

1214 SWITCH_DECLARE(void) switch_channel_process_export(switch_channel_t *channel, switch_channel_t
↳ *peer_channel,
1215                                     switch_event_t *var_event, const char
↳ *export_varname)
1216 {
1218     const char *export_vars = switch_channel_get_variable(channel, export_varname);
1219     char *cptmp = switch_core_session_strdup(channel->session, export_vars);
1220     int argc;
1221     char *argv[256];
1222     if (zstr(export_vars)) return;
1223     if (var_event) {
1224         switch_event_del_header(var_event, export_varname);
1225         switch_event_add_header_string(var_event, SWITCH_STACK_BOTTOM, export_varname, export_vars);
1226     }
1227
1228     if (peer_channel) {
1229         switch_channel_set_variable(peer_channel, export_varname, export_vars);
1230     }
1231
1232     if ((argc = switch_separate_string(cptmp, ',', argv, (sizeof(argv) / sizeof(argv[0])))) {
1233         int x;
1234
1235         for (x = 0; x < argc; x++) {
1236             const char *vval;
1237             if ((vval = switch_channel_get_variable(channel, argv[x]))) {
1238                 char *vvar = argv[x];
1239                 if (!strncasecmp(vvar, "nolocal:", 8)) { /* remove this later ? */
1240                     vvar += 8;
1241                 } else if (!strncasecmp(vvar, "_nolocal-", 9)) {
1242                     vvar += 9;
1243                 }
1244                 ...
1245                 if (peer_channel) {
1246                     ...
1247                     switch_channel_set_variable(peer_channel, vvar, vval);
1248                 }
1249             }
1250         }
1251     }
1252 }

```

往 B-leg 上 `export` 变量，`var_check` 会检查变量中是否还包含变量。

```

1271 SWITCH_DECLARE(switch_status_t) switch_channel_export_variable_var_check(switch_channel_t *channel,
1272                                     const char *varname, const
↳ char *val,
1273                                     const char *export_varname,
↳ switch_bool_t var_check)

```

还是 `export`，不过用类似 `printf()` 的方法格式化变量值。

```

1319 SWITCH_DECLARE(switch_status_t) switch_channel_export_variable_printf(switch_channel_t *channel, const
↳ char *varname,
1320                                     const char *export_varname,
↳ const char *fmt, ...)
...
1337     status = switch_channel_export_variable(channel, varname, data, export_varname);

```

L1345, 遍历并删除所有以 `prefix` 开头的通道变量。

```

1345 SWITCH_DECLARE(uint32_t) switch_channel_del_variable_prefix(switch_channel_t *channel, const char
↳ *prefix)

```

L1366, 将所有以 `prefix` 开头的变量都从 `orig_channel` 传递到 `new_channel` 上。

```

1366 SWITCH_DECLARE(switch_status_t) switch_channel_transfer_variable_prefix(switch_channel_t
↳ *orig_channel,
                                     switch_channel_t *new_channel, const char *prefix)

```

L1387, 设置 Presence 数据。

```

1387 SWITCH_DECLARE(void) switch_channel_set_presence_data_vals(switch_channel_t *channel, const char
↳ *presence_data_cols)

```

设置通道变量, 并检查通道变量的值是否也是一个变量。L1423 加锁; L1425, 如果值为空则删除变量, 否则, 检查变量值是否包含变量 (L1431), 如果不包含, 则将通道变量加到 `channel->variables` 这个 Event 数据结构里 (1434), 否则报错 (L1436)。

```

1416 SWITCH_DECLARE(switch_status_t) switch_channel_set_variable_var_check(switch_channel_t *channel,
1417                                     const char *varname, const char *value, switch_bool_t var_check)
1418 {
1423     switch_mutex_lock(channel->profile_mutex);
1424     if (channel->variables && !zstr(varname)) {
1425         if (zstr(value)) {
1426             switch_event_del_header(channel->variables, varname);
1427         } else {
1428             int ok = 1;

```

```

1430         if (var_check) {
1431             ok = !switch_string_var_check_const(value);
1432         }
1433         if (ok) {
1434             switch_event_add_header_string(channel->variables, SWITCH_STACK_BOTTOM, varname,
↪ value);
1435         } else {
1436             switch_log_printf(SWITCH_CHANNEL_CHANNEL_LOG(channel), SWITCH_LOG_CRIT, "Invalid data
↪ (${s} contains a variable)\n", varname);
1437         }
1438     }

```

L1447, 与上面的函数类似, 只不过, 多了一个参数, 除了可以增加到 Event 的底部 (SWITCH_STACK_BOTTOM) 外, 还可以选择其它值如 SWITCH_STACK_TOP、SWITCH_STACK_PUSH 等。

```

1447 SWITCH_DECLARE(switch_status_t) switch_channel_add_variable_var_check(switch_channel_t *channel, const
↪ char *varname, const char *value, switch_bool_t var_check, switch_stack_t stack)

```

以 `printf()` 方式格式化设置通道变量的值。

```

1480 SWITCH_DECLARE(switch_status_t) switch_channel_set_variable_printf(switch_channel_t *channel, const
↪ char *varname, const char *fmt, ...)

```

与上面类似, 只是, `printf()` 格式化的是通道变量的名称。

```

1511 SWITCH_DECLARE(switch_status_t) switch_channel_set_variable_name_printf(switch_channel_t *channel,
↪ const char *val, const char *fmt, ...)

```

将通道变量设置在 B-leg 上, 代码中好像没有用到。

```

1541 SWITCH_DECLARE(switch_status_t) switch_channel_set_variable_partner_var_check(switch_channel_t
↪ *channel, const char *varname, const char *value, switch_bool_t var_check)

```

在 Channel 上有一些 `switch_channel_flag_t` (在 `sitch_types.h` 中定义) 类型的标志, 标志 Channel 上的一些特性, 如 `CF_ANSWERED` 表示 Channel 已应答, `CF_VIDEO` 表示 Channel 支持视频等。L1562 测试某一 Channel 上是否有某一标志。

```
1562 SWITCH_DECLARE(uint32_t) switch_channel_test_flag(switch_channel_t *channel, switch_channel_flag_t
↪ flag)
1563 {
1564     uint32_t r = 0;
...
1568     switch_mutex_lock(channel->flag_mutex);
1569     r = channel->flags[flag];
1570     switch_mutex_unlock(channel->flag_mutex);
1572     return r;
1573 }
```

在 B-leg 上设置标志。

```
1575 SWITCH_DECLARE(switch_bool_t) switch_channel_set_flag_partner(switch_channel_t *channel,
↪ switch_channel_flag_t flag)
```

测试 B-leg 上的标志。

```
1593 SWITCH_DECLARE(uint32_t) switch_channel_test_flag_partner(switch_channel_t *channel,
↪ switch_channel_flag_t flag)
```

清除 B-leg 上的标志。

```
1611 SWITCH_DECLARE(switch_bool_t) switch_channel_clear_flag_partner(switch_channel_t *channel,
↪ switch_channel_flag_t flag)
```

无限循环等待某一呼叫状态到来。其中，`switch_cond_next()` 这个函数基本就是 sleep 1 毫秒，给其它线程运行的机会。

```
1629 SWITCH_DECLARE(void) switch_channel_wait_for_state(switch_channel_t *channel, switch_channel_t
↪ *other_channel, switch_channel_state_t want_state)
1630 {
...
1634     for (;;) {
1635         if ((channel->state < CS_HANGUP && channel->state == channel->running_state && channel-
↪ >running_state == want_state) ||
```

```

1636         (other_channel && switch_channel_down_nosig(other_channel)) ||
↪ switch_channel_down(channel)) {
1637         break;
1638     }
1639     switch_cond_next();
1640 }
1641 }

```

与上面类似，如果等待超时则返回。

```

1644 SWITCH_DECLARE(void) switch_channel_wait_for_state_timeout(switch_channel_t
↪ *channel, switch_channel_state_t want_state, uint32_t timeout)
1645 {

```

等待 Channel 上某一标志出现（`pres` 为真，L1667），或消失（`pres` 为假）。如果 `super_channel` 非空，则它必须在 `switch_channel_ready()` 状态才返回真。

```

1665 SWITCH_DECLARE(switch_status_t) switch_channel_wait_for_flag(switch_channel_t
↪ *channel, switch_channel_flag_t want_flag, switch_bool_t pres, uint32_t to, switch_channel_t
↪ *super_channel)

```

设置和清除 Channel 的能力（Capability）值。

```

1704 SWITCH_DECLARE(void) switch_channel_set_cap_value(switch_channel_t *channel, switch_channel_cap_t cap,
↪ uint32_t value)
1714 SWITCH_DECLARE(void) switch_channel_clear_cap(switch_channel_t *channel, switch_channel_cap_t cap)

```

检测 Channel 的能力。

```

1724 SWITCH_DECLARE(uint32_t) switch_channel_test_cap(switch_channel_t *channel, switch_channel_cap_t cap)

```

检测 B-leg 的能力。

```

1730 SWITCH_DECLARE(uint32_t) switch_channel_test_cap_partner(switch_channel_t *channel,
↪ switch_channel_cap_t cap)

```

获取 Channel 上所有的标志,以字符串形式返回。这里,用到了一个 `switch_stream_handle_t` (L1750) 数据结构。该结构会返回一个 `stream`, 是一个流。该内存流可以持续写入,也可以从里面读取。流可以指向一段内存,也可以是一个文件。L1754 是一个宏,它会为该流在堆上申请一段内存,指针存放在 `stream.data` 里, `stream.write_function` 会向这段内存中写入 (L1579)。该函数最后返回了 `stream.data`, 因而,返回值用完后应该由调用者释放。

```

1748 SWITCH_DECLARE(char *) switch_channel_get_flag_string(switch_channel_t *channel)
1749 {
1750     switch_stream_handle_t stream = { 0 };
1754     SWITCH_STANDARD_STREAM(stream);
1755
1756     switch_mutex_lock(channel->flag_mutex);
1757     for (i = 0; i < CF_FLAG_MAX; i++) {
1758         if (channel->flags[i]) {
1759             stream.write_function(&stream, "%d=%d;", i, channel->flags[i]);
1760         }
1761     }
1762     switch_mutex_unlock(channel->flag_mutex);
1764     r = (char *) stream.data;
1766     if (end_of(r) == ';') end_of(r) = '\0';
1770     return r;
1772 }

```

与上面函数类似,取得 Channel 能力值,返回字符串。

```

1774 SWITCH_DECLARE(char *) switch_channel_get_cap_string(switch_channel_t *channel)

```

设置 Channel 标志的值 (L1812 ~ L1815), 同时,根据不同的标志会有一些特殊处理,如 L1819 会向对方请求一个关键帧, L1871 会启动一个视频线程,专门处理该 Channel 的视频数据等。

```

1800 SWITCH_DECLARE(void) switch_channel_set_flag_value(switch_channel_t *channel, switch_channel_flag_t
↪ flag, uint32_t value)
1801 {
...
1812     if (channel->flags[flag] != value) {
1813         just_set = 1;
1814         channel->flags[flag] = value;
1815     }
...
1818     if (flag == CF_VIDEO_READY && just_set) {
1819         switch_core_session_request_video_refresh(channel->session);

```

```
1820     }
1821
...
1870     if (flag == CF_VIDEO_ECHO || flag == CF_VIDEO_BLANK || flag == CF_VIDEO_DECODED_READ || flag ==
↪ CF_VIDEO_PASSIVE) {
1871         switch_core_session_start_video_thread(channel->session);
```

用于多次设置 Channel 标志，每次值加 1。

```
1879 SWITCH_DECLARE(void) switch_channel_set_flag_recursive(switch_channel_t *channel,
↪ switch_channel_flag_t flag)
1880 {
1885     channel->flags[flag]++;
```

设置、清除和检测私有的标志。

```
1898 SWITCH_DECLARE(void) switch_channel_set_private_flag(switch_channel_t *channel, uint32_t flags)
1906 SWITCH_DECLARE(void) switch_channel_clear_private_flag(switch_channel_t *channel, uint32_t flags)
1914 SWITCH_DECLARE(int) switch_channel_test_private_flag(switch_channel_t *channel, uint32_t flags)
```

设置、清除和测试 Channel 标志，标志会存到哈希表里，所以需要指定一个 key。

```
1920 SWITCH_DECLARE(void) switch_channel_set_app_flag_key(const char *key, switch_channel_t *channel,
↪ uint32_t flags)
1921 {
1944 SWITCH_DECLARE(void) switch_channel_clear_app_flag_key(const char *key, switch_channel_t *channel,
↪ uint32_t flags)
1960 SWITCH_DECLARE(int) switch_channel_test_app_flag_key(const char *key, switch_channel_t *channel,
↪ uint32_t flags)
```

设置 Channel 的状态标志。

```
1976 SWITCH_DECLARE(void) switch_channel_set_state_flag(switch_channel_t *channel, switch_channel_flag_t flag)
1977 {
...
1981     channel->state_flags[0] = 1;
1982     channel->state_flags[flag] = 1;
1983     switch_mutex_unlock(channel->flag_mutex);
```

清除状态标志。

```
1986 SWITCH_DECLARE(void) switch_channel_clear_state_flag(switch_channel_t *channel, switch_channel_flag_t
↪ flag)
```

清除 Channel 标志，并根据不同的标志可能有不同的动作。如 L2059 会唤醒视频线程。

```
1995 SWITCH_DECLARE(void) switch_channel_clear_flag(switch_channel_t *channel, switch_channel_flag_t flag)
2058     if (flag == CF_VIDEO_PASSIVE && CLEAR) {
2059         switch_core_session_wake_video_thread(channel->session);
2060     }
```

多次清除标志。使用时理论上 `set_flag_recursive` 应该和 `clear_flag_recursive` 成对出现。

```
2069 SWITCH_DECLARE(void) switch_channel_clear_flag_recursive(switch_channel_t *channel,
↪ switch_channel_flag_t flag)
2070 {
2076     channel->flags[flag]--;
```

获取 Channel 当前的状态。

```
2085 SWITCH_DECLARE(switch_channel_state_t) switch_channel_get_state(switch_channel_t *channel)
```

获取 Channel 当前运行状态。

```
2095 SWITCH_DECLARE(switch_channel_state_t) switch_channel_get_running_state(switch_channel_t *channel)
```

如果当前的 Channel 状态和运行状态不一致，则返回非 0 值。

```
2105 SWITCH_DECLARE(int) switch_channel_state_change_pending(switch_channel_t *channel)
2106 {
...
2111     return channel->running_state != channel->state;
2112 }
```

检查 Channel 上的信号。

```

2114 SWITCH_DECLARE(int) switch_channel_check_signal(switch_channel_t *channel, switch_bool_t
↳ in_thread_only)
2115 {
2116     switch_ivr_parse_signal_data(channel->session, SWITCH_FALSE, in_thread_only);
2117     return 0;

```

检测 Channel 是否准备就绪。这是个非常重要的函数，在很多地方都用到。L2126 检查 Channel 上的信号，这个很不细说。L2128，如果要检查媒体，则必须满足电话已应答或在 Early Media 状态（L2129 ~ L2130），并且 Channel 不在 Proxy Media 或 Bypass Media 状态，并且 Read Codec 和 Write Codec 都非空（能正常读写媒体，L2131）才能返回真。

L2138，如果 `check_ready` 为真则继续检查信号。L2143，如果 `hangup_cause` 非空由说明要挂机了，Channel 的状态也必须满足一定条件，并且 Channel 上没有 Transfer 和 Not Ready 标记，并且 Channel 的状态是一致的（没有待修改的状态），只有满足这些条件才认为 Channel 是 Ready 的，能正常通信。如果该函数返回值为假，则控制当前 Channel 的 Application 就不应该再做其它操作，而应该退出循环立即退出，以便让核心状态机完成一个 Channel 的生命周期。

```

2120 SWITCH_DECLARE(int) switch_channel_test_ready(switch_channel_t *channel, switch_bool_t
↳ check_ready, switch_bool_t check_media)
2121 {
2126     switch_channel_check_signal(channel, SWITCH_TRUE);
2127
2128     if (check_media) {
2129         ret = ((switch_channel_test_flag(channel, CF_ANSWERED) ||
2130             switch_channel_test_flag(channel, CF_EARLY_MEDIA)) && !
↳ switch_channel_test_flag(channel, CF_PROXY_MODE) &&
2131             switch_core_session_get_read_codec(channel->session) &&
↳ switch_core_session_get_write_codec(channel->session));
2133
2134         if (!ret) return ret;
2136     }
2138     if (!check_ready) return ret;
2141     ret = 0;
2143     if (!channel->hangup_cause && channel->state > CS_ROUTING && channel->state < CS_HANGUP && channel-
↳ >state != CS_RESET &&
2144         !switch_channel_test_flag(channel, CF_TRANSFER) && !switch_channel_test_flag(channel,
↳ CF_NOT_READY) &&
2145         !switch_channel_state_change_pending(channel)) {
2146         ret++;
2147     }
2151     return ret;
2152 }

```

Channel 状态的名字数组。

```
2154 static const char *state_names[] = {
2155     "CS_NEW",
2156     "CS_INIT",
2157     "CS_ROUTING",
2158     "CS_SOFT_EXECUTE",
2159     "CS_EXECUTE",
2160     "CS_EXCHANGE_MEDIA",
2161     "CS_PARK",
2162     "CS_CONSUME_MEDIA",
2163     "CS_HIBERNATE",
2164     "CS_RESET",
2165     "CS_HANGUP",
2166     "CS_REPORTING",
2167     "CS_DESTROY",
2168     "CS_NONE",
2169     NULL
2170 };
```

返回 Channel 状态对应的名字。

```
2172 SWITCH_DECLARE(const char *) switch_channel_state_name(switch_channel_state_t state)
2173 {
2174     return state_names[state];
2175 }
```

上一函数的逆向函数，返回 Channel 状态字符串对应的内部表示。

```
2178 SWITCH_DECLARE(switch_channel_state_t) switch_channel_name_state(const char *name)
```

L2190 这段内联代码用于设置 Channel 的状态，它会首先尝试锁定 Channel 级别的线程 Mutex (L2192)，如果不成功则尝试 Session 的 Mutex，如果以锁定，则更新当前状态，如果 100 次后还不能锁定，则无论如何都修改状态。

```
2190 static inline void careful_set(switch_channel_t *channel, switch_channel_state_t
↪ *state, switch_channel_state_t val) {
2191
```

```

2192     if (switch_mutex_trylock(channel->thread_mutex) == SWITCH_STATUS_SUCCESS) {
2193         *state = val;
2194         switch_mutex_unlock(channel->thread_mutex);
2195     } else {
2196         switch_mutex_t *mutex = switch_core_session_get_mutex(channel->session);
2197         int x = 0;
2198
2199         for (x = 0; x < 100; x++) {
2200             if (switch_mutex_trylock(mutex) == SWITCH_STATUS_SUCCESS) {
2201                 *state = val;
2202                 switch_mutex_unlock(mutex);
2203                 break;
2204             } else {
2205                 switch_cond_next();
2206             }
2207         }
2208
2209         if (x == 100) {
2210             *state = val;
2211         }

```

设置运行状态。这里会发送 Channel 状态改变事件（L2260）。

```

2216 SWITCH_DECLARE(switch_channel_state_t) switch_channel_perform_set_running_state(switch_channel_t
↳ *channel, switch_channel_state_t state, const char *file, const char *func, int line)
2218 {
...
2258     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_STATE) == SWITCH_STATUS_SUCCESS) {
2259         switch_channel_event_set_data(channel, event);
2260         switch_event_fire(&event);
2261     }

```

设置 Channel 状态，`last_state` 是 Channel 的上一个状态（L2279），当然，如果上一个状态跟下一个状态（`state`）相同就不必做任何操作了（L2282）。接下来的代码其它是一个状态机，根据 `last_state` 和 `state` 的值来决定怎么做（L2309 ~ L2462）。比如 L2468 行会调用上面提到的 `careful_set` 函数来设置 Channel 的状态。是后返回当前 Channel 的状态（L2488）。

```

2269 SWITCH_DECLARE(switch_channel_state_t) switch_channel_perform_set_state(switch_channel_t
↳ *channel, const char *file, const char *func, int line, switch_channel_state_t state)
2271 {
...
2279     last_state = channel->state;
2281

```

```
2282     if (last_state == state) goto done;
...
2309     switch (last_state) {
2310     case CS_NEW:
2311     case CS_RESET:
2312         switch (state) {
2313         default:
2314             ok++; break;
2316         }
2317         break;
2319     case CS_INIT:
2320         switch (state) {
2321         case CS_EXCHANGE_MEDIA:
2322         case CS_SOFT_EXECUTE:
2323         case CS_ROUTING:
2324         case CS_EXECUTE:
2325         case CS_PARK:
2326         case CS_CONSUME_MEDIA:
2327         case CS_HIBERNATE:
2328         case CS_RESET:
2329             ok++;
2330         default:
2331             break;
2332         }
2333         break;
...
2450     case CS_REPORTING:
2451         switch (state) {
2452         case CS_DESTROY:
2453             ok++;
2454         default:
2455             break;
2456         }
2457         break;
2459     default:
2460         break;
2462     }

2464     if (ok) {
...
2468         careful_set(channel, &channel->state, state);
...
2487     return channel->state;
2488 }
```

加线程锁。

```

2490 SWITCH_DECLARE(void) switch_channel_state_thread_lock(switch_channel_t *channel)
2491 {
2492     switch_mutex_lock(channel->thread_mutex);
2493 }
...
2496 SWITCH_DECLARE(switch_status_t) switch_channel_state_thread_trylock(switch_channel_t *channel)
2497 {
2498     return switch_mutex_trylock(channel->thread_mutex);
2499 }
...
2502 SWITCH_DECLARE(void) switch_channel_state_thread_unlock(switch_channel_t *channel)
2503 {
2504     switch_mutex_unlock(channel->thread_mutex);
2505 }

```

将 Channel 的基础数据设置到 [event](#) 上。如 [Channel-State](#)、[Channel-Call-State](#)、[Unique-ID](#) 等，如果在挂机状态还有 [Hangup-Cause](#)。

```

2507 SWITCH_DECLARE(void) switch_channel_event_set_basic_data(switch_channel_t *channel, switch_event_t
↳ *event)
2508 {
...
2521     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
        "Channel-State", switch_channel_state_name(channel->running_state));
2522     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
        "Channel-Call-State", switch_channel_callstate2str(channel->callstate));
2526     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
        "Unique-ID", switch_core_session_get_uuid(channel->session));
2527
...
2568     if (channel->hangup_cause) {
2569         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
            "Hangup-Cause", switch_channel_cause2str(channel->hangup_cause));
2570     }

```

与上面类似，设置扩展数据，大部分是一些通道变量。大家才编码开发中可能遇到，有一些 Channel 相关的事件中是有所有的通道变量的，但有一些则没有，后者就是因为没有设置这些扩展数据导致的，这主要是为了节约事件的处理。但有时候，为了开发方便，还是希望所有事件中都能得到所有的通道变量，这时候，就可以通过 [verbose-event](#) 参数开启，开启后 L2616 ~ L2617 就返回真，然后所有事件都会带上这些扩展数据。

```

2607 SWITCH_DECLARE(void) switch_channel_event_set_extended_data(switch_channel_t *channel, switch_event_t
↳ *event)
2608 {

```

```

...
2616     if (global_verbose_events ||
2617         switch_channel_test_flag(channel, CF_VERBOSE_EVENTS) ||
2618         switch_event_get_header(event, "presence-data-cols") ||
2619         event->event_id == SWITCH_EVENT_CHANNEL_CREATE ||
2622         event->event_id == SWITCH_EVENT_CHANNEL_ANSWER ||
2625         event->event_id == SWITCH_EVENT_CHANNEL_BRIDGE ||
2629         event->event_id == SWITCH_EVENT_CHANNEL_HANGUP ||
2630         event->event_id == SWITCH_EVENT_CHANNEL_HANGUP_COMPLETE ||
...
2645         event->event_id == SWITCH_EVENT_CUSTOM) {
2649         if (channel->scope_variables) {
2650             switch_event_t *ep;
2652             for (ep = channel->scope_variables; ep; ep = ep->next) {
2653                 for (hi = ep->headers; hi; hi = hi->next) {
...
2663                     if (!switch_event_get_header(event, buf)) {
2664                         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, buf, vval);
2665                     }
2666                 }
2667             }
2668         }
2669
2670         if (channel->variables) {
2671             for (hi = channel->variables->headers; hi; hi = hi->next) {
...
2680                 switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, buf, vval);
2681             }
2682         }
2683     }

```

L2689 设置 Channel 基础和扩展数据。

```

2689 SWITCH_DECLARE(void) switch_channel_event_set_data(switch_channel_t *channel, switch_event_t *event)
2690 {
2691     switch_mutex_lock(channel->profile_mutex);
2692     switch_channel_event_set_basic_data(channel, event);
2693     switch_channel_event_set_extended_data(channel, event);
2694     switch_mutex_unlock(channel->profile_mutex);
2695 }

```

设置 Caller Profile。

```

2697 SWITCH_DECLARE(void) switch_channel_step_caller_profile(switch_channel_t *channel)
2698 {

```

```
2703     cp = switch_caller_profile_clone(channel->session, channel->caller_profile);  
...  
2706     switch_channel_set_caller_profile(channel, cp);  
2707 }  
2708  
2709 SWITCH_DECLARE(void) switch_channel_set_caller_profile(switch_channel_t *channel,  
↪ switch_caller_profile_t *caller_profile)  
2710 {
```

获取 Caller Profile。

```
2760 SWITCH_DECLARE(switch_caller_profile_t *) switch_channel_get_caller_profile(switch_channel_t *channel)
```

设置主叫 Caller Profile。

```
2772 SWITCH_DECLARE(void) switch_channel_set_originator_caller_profile(switch_channel_t *channel,  
↪ switch_caller_profile_t *caller_profile)
```

设置寻路（查找路由）Caller Profile。

```
2791 SWITCH_DECLARE(void) switch_channel_set_hunt_caller_profile(switch_channel_t *channel,  
↪ switch_caller_profile_t *caller_profile)
```

设置呼叫 Caller Profile。

```
2807 SWITCH_DECLARE(void) switch_channel_set_origination_caller_profile(switch_channel_t *channel,  
↪ switch_caller_profile_t *caller_profile)
```

获取各种 Caller Profile。

```
2822 SWITCH_DECLARE(switch_caller_profile_t *)  
↪ switch_channel_get_origination_caller_profile(switch_channel_t *channel)  
  
2837 SWITCH_DECLARE(void) switch_channel_set_originatee_caller_profile(switch_channel_t *channel,  
↪ switch_caller_profile_t *caller_profile)
```

```
2853 SWITCH_DECLARE(switch_caller_profile_t *)  
↪ switch_channel_get_originator_caller_profile(switch_channel_t *channel)  
  
2868 SWITCH_DECLARE(switch_caller_profile_t *)  
↪ switch_channel_get_originatee_caller_profile(switch_channel_t *channel)
```

取得 Channel 的 UUID。

```
2882 SWITCH_DECLARE(char *) switch_channel_get_uuid(switch_channel_t *channel)  
2883 {  
2886     return switch_core_session_get_uuid(channel->session);  
2887 }
```

增加状态处理器 (State Handler)。增加后, State Handler 指定的函数将会在每次 Channel 状态发生变化时有机会被回调。

```
2889 SWITCH_DECLARE(int) switch_channel_add_state_handler(switch_channel_t *channel, const  
↪ switch_state_handler_table_t *state_handler)  
2890 {  
2891     int x, index;  
2892  
2893     switch_assert(channel != NULL);  
2894     switch_mutex_lock(channel->state_mutex);  
2895     for (x = 0; x < SWITCH_MAX_STATE_HANDLERS; x++) {  
2896         if (channel->state_handlers[x] == state_handler) {  
2897             index = x;  
2898             goto end;  
2899         }  
2900     }  
2901     index = channel->state_handler_index++;  
2902  
2903     if (channel->state_handler_index >= SWITCH_MAX_STATE_HANDLERS) {  
2904         index = -1;  
2905         goto end;  
2906     }  
2907  
2908     channel->state_handlers[index] = state_handler;  
2909  
2910     end:  
2911     switch_mutex_unlock(channel->state_mutex);  
2912     return index;  
2913 }
```

获取、清除 State Handler。

```

2915 SWITCH_DECLARE(const switch_state_handler_table_t *) switch_channel_get_state_handler(switch_channel_t
↪ *channel, int index)

2932 SWITCH_DECLARE(void) switch_channel_clear_state_handler(switch_channel_t *channel, const
↪ switch_state_handler_table_t *state_handler)

```

重启 Channel。

```

2969 SWITCH_DECLARE(void) switch_channel_restart(switch_channel_t *channel)
2970 {
2971     switch_channel_set_state(channel, CS_RESET);
2972     switch_channel_wait_for_state_timeout(channel, CS_RESET, 5000);
2973     switch_channel_set_state(channel, CS_EXECUTE);
2974 }

```

Caller Extension 伪装术。实际上就是 Copy 一些数据从原来的 `orig_channel` 到 `new_channel`。用于呼叫转接的场合，试想一下，A 呼 B，B 呼 C，然后 B 挂机，AC 通话时，B 上的一些数据要 Copy 到 C 上。

```

2985 SWITCH_DECLARE(switch_status_t) switch_channel_caller_extension_masquerade(switch_channel_t
↪ *orig_channel, switch_channel_t *new_channel, uint32_t offset)
2986 {
...
2997     if (no_copy) {
2998         dup = switch_core_session_strdup(new_channel->session, no_copy);
2999         argc = switch_separate_string(dup, ',', argv, (sizeof(argv) / sizeof(argv[0])));
3000     }
...
3007     caller_profile = switch_caller_profile_clone(new_channel->session, new_channel->caller_profile);
3008     switch_assert(caller_profile);
3009     extension = switch_caller_extension_new(new_channel->session, caller_profile-
↪ >destination_number, caller_profile->destination_number);
3010     orig_extension = switch_channel_get_caller_extension(orig_channel);
3013     if (extension && orig_extension) {
3014         for (ap = orig_extension->current_application; ap && offset > 0; offset--) {
3015             ap = ap->next;
3016         }
3017
3018         for (; ap; ap = ap->next) {
3019             switch_caller_extension_add_application(new_channel->session, extension,

```

```

        ap->application_name, ap->application_data);
3020     }
3021
3022     caller_profile->destination_number = switch_core_strdup(caller_profile->pool, orig_channel-
↪ >caller_profile->destination_number);
3023     switch_channel_set_caller_profile(new_channel, caller_profile);
3024     switch_channel_set_caller_extension(new_channel, extension);
3025
3026     for (hi = orig_channel->variables->headers; hi; hi = hi->next) {
3038         switch_channel_set_variable(new_channel, hi->name, hi->value);
3039     }

```

翻转主、被叫号码。试想在回呼场景下，FS 呼 A（这时 Channel 上的主叫号码是 FS），然后 Bridge 到 B，这时 B 看到的应该是 A 的号码。

```

3052 SWITCH_DECLARE(void) switch_channel_invert_cid(switch_channel_t *channel)

```

翻转和更新主被叫号码。它会发送 UPDATE 消息通知对端的 UA 更新被叫号码的显示，多用于转接场景中。

```

3082 SWITCH_DECLARE(void) switch_channel_flip_cid(switch_channel_t *channel)
...
3115     if (switch_event_create(&event, SWITCH_EVENT_CALL_UPDATE) == SWITCH_STATUS_SUCCESS) {
...
3122         switch_channel_event_set_data(channel, event);
3123         switch_event_fire(&event);
3124     }
3126
3127     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(channel->session), SWITCH_LOG_INFO, "%s Flipping CID
↪ from \"%s\" <%s> to \"%s\" <%s>\n",

```

确定什么时候需要翻转主被叫号码。L3140，如果 A 是呼入的电话并且有 B-leg，则需要翻转（B-leg 的被叫号码可能已改变，这时候要通知 A 真正的被叫号码）；L3143，如果是直接呼出的电话，并且电话没有到 Dialplan 进行路由，说明是回呼的电话，也需要翻转一下。我们到后面可以看到具体的使用场景。

```

3137 SWITCH_DECLARE(void) switch_channel_sort_cid(switch_channel_t *channel)
3138 {
3139
3140     if (switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_INBOUND &&

```

```
        switch_channel_test_flag(channel, CF_BLEG)) {
3141    switch_channel_flip_cid(channel);
3142    switch_channel_clear_flag(channel, CF_BLEG);
3143    } else if (switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_OUTBOUND &&
        !switch_channel_test_flag(channel, CF_DIALPLAN)) {
3144    switch_channel_set_flag(channel, CF_DIALPLAN);
3145    switch_channel_flip_cid(channel);
3146    }
3147 }
```

取得排队的 Extension。

```
3149 SWITCH_DECLARE(switch_caller_extension_t *) switch_channel_get_queued_extension(switch_channel_t
↳ *channel)
```

转接到某一 Extension。

```
3161 SWITCH_DECLARE(void) switch_channel_transfer_to_extension(switch_channel_t
↳ *channel, switch_caller_extension_t *caller_extension)
3162 {
3163    switch_mutex_lock(channel->profile_mutex);
3164    channel->queued_extension = caller_extension;
3165    switch_mutex_unlock(channel->profile_mutex);
3166
3167    switch_channel_set_flag(channel, CF_TRANSFER);
3168    switch_channel_set_state(channel, CS_ROUTING);
3169 }
```

设置和获取主叫 Extension。

```
3171 SWITCH_DECLARE(void) switch_channel_set_caller_extension(switch_channel_t *channel,
↳ switch_caller_extension_t *caller_extension)

3184 SWITCH_DECLARE(switch_caller_extension_t *) switch_channel_get_caller_extension(switch_channel_t
↳ *channel)
```

设置桥接时间。 `switch_micro_time_now` 取得当前时间，精度是微秒。

```

3198 SWITCH_DECLARE(void) switch_channel_set_bridge_time(switch_channel_t *channel)
3199 {
3200     switch_mutex_lock(channel->profile_mutex);
3201     if (channel->caller_profile && channel->caller_profile->times) {
3202         channel->caller_profile->times->bridged = switch_micro_time_now();
3203     }
3204     switch_mutex_unlock(channel->profile_mutex);
3205 }

```

设置挂机时间。

```

3208 SWITCH_DECLARE(void) switch_channel_set_hangup_time(switch_channel_t *channel)

```

挂机。每个 Channel 都在单独的线程里执行。如果挂机时由于各种原因线程未启动（L3279），则启动之（L3280，当然它会很快终止，启动线程只是为了完成一些标准的流程）。L3283 ~ L3285 会发送挂机事件。L3288 给 Channel 发送 Kill 事件，以便线程能及时终止。L3290，处理一些在挂机状态下应该做的事情。

```

3218 SWITCH_DECLARE(switch_channel_state_t) switch_channel_perform_hangup(switch_channel_t *channel, const
↪ char *file, const char *func, int line, switch_call_cause_t hangup_cause)
...
3279     if (!switch_core_session_running(channel->session) && !switch_core_session_started(channel-
↪ >session)) {
3280         switch_core_session_thread_launch(channel->session);
3281     }
3282
3283     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_HANGUP) == SWITCH_STATUS_SUCCESS) {
3284         switch_channel_event_set_data(channel, event);
3285         switch_event_fire(&event);
3286     }
3287
3288     switch_core_session_kill_channel(channel->session, SWITCH_SIG_KILL);
3289     switch_core_session_signal_state_change(channel->session);
3290     switch_core_session_hangup_state(channel->session, SWITCH_FALSE);
3291 }

```

发送通知 (Indication) 消息的函数。L3302 的 `switch_core_session_perform_receive_message` 是一个同步接收消息的函数，它会回调相关的回调函数处理这个消息。

```

3296 static switch_status_t send_ind(switch_channel_t *channel, switch_core_session_message_types_t msg_id,
↳ const char *file, const char *func, int line)
3297 {
3298     switch_core_session_message_t msg = { 0 };
3299
3300     msg.message_id = msg_id;
3301     msg.from = channel->name;
3302     return switch_core_session_perform_receive_message(channel->session, &msg, file, func, line);
3303 }

```

设置 Channel 的状态为 Ring Ready (一般是收到 SIP 180 消息时)。如果这时候还有另一条腿, 则设置另一条腿的呼叫进展时间 (Progress Time, L3326)。在这里会发送呼叫进展 (Channel Progress) 事件 (L3335 ~ L3337)。如果该 Channel 上有 `execute_on_ring` 的回调, 还会在这里回调相应的函数以执行相应的 Application (L3340), 同理也会执行 `api_on_ring` 设置的 API (L3341)。将 Channel 的状态设为 `CCS_RINGING` (Channel CallState Ringing, L3343)。使用上面提到的 `send_ind` 发送一个通知消息 (L3345)。

```

3306 SWITCH_DECLARE(switch_status_t) switch_channel_perform_mark_ring_ready_value(switch_channel_t
↳ *channel, switch_ring_ready_t rv, const char *file, const char *func, int line)
3309 {
...
3314     switch_channel_set_flag_value(channel, CF_RING_READY, rv);
3316
...
3322     if ((other_session = switch_core_session_locate(channel->caller_profile-
↳ >originator_caller_profile->uuid))) {
...
3326         other_channel->caller_profile->times->progress = channel->caller_profile-
↳ >times->progress;
...
3329     }
3330     channel->caller_profile->originator_caller_profile->times->progress = channel-
↳ >caller_profile->times->progress;
3334
3335     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_PROGRESS) == SWITCH_STATUS_SUCCESS) {
3336         switch_channel_event_set_data(channel, event);
3337         switch_event_fire(&event);
3338     }
3339
3340     switch_channel_execute_on(channel, SWITCH_CHANNEL_EXECUTE_ON_RING_VARIABLE);
3341     switch_channel_api_on(channel, SWITCH_CHANNEL_API_ON_RING_VARIABLE);
3343     switch_channel_set_callstate(channel, CCS_RINGING);
3345     send_ind(channel, SWITCH_MESSAGE_RING_EVENT, file, func, line);

```

检查 `zrtp`。TODO

```

3353 SWITCH_DECLARE(void) switch_channel_check_zrtp(switch_channel_t *channel)
3354 {
3355
3356     if (!switch_channel_test_flag(channel, CF_ZRTP_PASSTHRU)
3357         && switch_channel_test_flag(channel, CF_ZRTP_PASSTHRU_REQ)
3358         && switch_channel_test_flag(channel, CF_ZRTP_HASH)) {
3359         switch_core_session_t *other_session;
3360         switch_channel_t *other_channel;
3361         int doit = 1;
3362
3363         if (switch_core_session_get_partner(channel->session, &other_session) ==
↪ SWITCH_STATUS_SUCCESS) {
3364             other_channel = switch_core_session_get_channel(other_session);
3365
3366             if (switch_channel_test_flag(other_channel, CF_ZRTP_HASH) && !
↪ switch_channel_test_flag(other_channel, CF_ZRTP_PASSTHRU)) {
3367
3368                 switch_channel_set_flag(channel, CF_ZRTP_PASSTHRU);
3369                 switch_channel_set_flag(other_channel, CF_ZRTP_PASSTHRU);
3370
3371                 switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(channel->session), SWITCH_LOG_INFO,
3372                                "%s Activating ZRTP passthru mode.\n",
↪ switch_channel_get_name(channel));
3373
3374                 switch_channel_set_variable(channel, "zrtp_passthru_active", "true");
3375                 switch_channel_set_variable(other_channel, "zrtp_passthru_active", "true");
3376                 switch_channel_set_variable(channel, "zrtp_secure_media", "false");
3377                 switch_channel_set_variable(other_channel, "zrtp_secure_media", "false");
3378                 doit = 0;
3379             }
3380
3381             switch_core_session_rwlock_unlock(other_session);
3382         }
3383
3384         if (doit) {
3385             switch_channel_set_variable(channel, "zrtp_passthru_active", "false");
3386             switch_channel_set_variable(channel, "zrtp_secure_media", "true");
3387             switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(channel->session), SWITCH_LOG_INFO,
3388                            "%s ZRTP not negotiated on both sides; disabling ZRTP passthru mode.\n",
↪ switch_channel_get_name(channel));
3389
3390             switch_channel_clear_flag(channel, CF_ZRTP_PASSTHRU);
3391             switch_channel_clear_flag(channel, CF_ZRTP_HASH);
3392
3393             if (switch_core_session_get_partner(channel->session, &other_session) ==
↪ SWITCH_STATUS_SUCCESS) {

```

```

3394         other_channel = switch_core_session_get_channel(other_session);
3395
3396         switch_channel_set_variable(other_channel, "zrtp_passthru_active", "false");
3397         switch_channel_set_variable(other_channel, "zrtp_secure_media", "true");
3398         switch_channel_clear_flag(other_channel, CF_ZRTP_PASSTHRU);
3399         switch_channel_clear_flag(other_channel, CF_ZRTP_HASH);
3400
3401         switch_core_session_rwlock_unlock(other_session);
3402     }
3403
3404 }
3405 }
3406 }

```

L3408 将 Channel 设为 Early Media 状态（应答之前的媒体状态，一般是在收到 SIP 183 消息时调用）。与 L3306 类似，它会发送 Progress Media 消息（L3446），并执行 `execute_on_pre_answer`、`execute_on_media`（L3451 ~ L3452）和 `api_on_pre_answer`、`api_on_media`（L3454 ~ L3455）回调。

L3457 ~ L3459 是一个特性，在媒体 Passthru 状态下，Bridge 的两个 Channel 的 ptime 可以是不同的。

L3465 ~ L3468，如果我们是一个子 Channel（如 Bridge 状态的 B-leg），同时 A-leg 在阻塞的状态，那么 A-leg 将无从知道我们的状态已经改变了，因此，需要发送一个 Break 消息让对方暂时中断阻塞。

L3475 检查媒体的自动调整功能。自动调整是为了适应 NAT 之类的环境帮助媒体穿越，我们后面再详细分析。

```

3408 SWITCH_DECLARE(switch_status_t) switch_channel_perform_mark_pre_answered(switch_channel_t *channel,
↳ const char *file, const char *func, int line)
3409 {
...
3446     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_PROGRESS_MEDIA) == SWITCH_STATUS_SUCCESS)
↳ {
3449     }
3450
3451     switch_channel_execute_on(channel, SWITCH_CHANNEL_EXECUTE_ON_PRE_ANSWER_VARIABLE);
3452     switch_channel_execute_on(channel, SWITCH_CHANNEL_EXECUTE_ON_MEDIA_VARIABLE);
3453
3454     switch_channel_api_on(channel, SWITCH_CHANNEL_API_ON_PRE_ANSWER_VARIABLE);
3455     switch_channel_api_on(channel, SWITCH_CHANNEL_API_ON_MEDIA_VARIABLE);
3456
3457     if (switch_true(switch_channel_get_variable(channel,
↳ SWITCH_PASSTHRU_PTIME_MISMATCH_VARIABLE))) {

```

```

3458         switch_channel_set_flag(channel, CF_PASSTHRU_PTIME_MISMATCH);
3459     }
3460
3461     if ((uuid = switch_channel_get_variable(channel, SWITCH_ORIGINATOR_VARIABLE))
3462         && (other_session = switch_core_session_locate(uuid))) {
3463         switch_core_session_kill_channel(other_session, SWITCH_SIG_BREAK);
3464         switch_core_session_rwlock_unlock(other_session);
3465     }
3466
3467     switch_channel_set_callstate(channel, CCS_EARLY);
3468     send_ind(channel, SWITCH_MESSAGE_PROGRESS_EVENT, file, func, line);
3469     switch_core_media_check_autoadj(channel->session);

```

在当前 Channel 上执行预应答 (Pre Answer) 功能, 仅对呼入的呼叫有效 (L3502)。实际上就是发送一个 **INDICATE_PROGRESS** 消息, L3505 会调用相关的回调函数处理这件事 (比如向对方发送 SIP 183 消息)。如果发送成功 (L3508), 则将当前 Channel 标记为预应答的状态 (L3509), 并且处理一下媒体同步 (L3510)

```

3483 SWITCH_DECLARE(switch_status_t) switch_channel_perform_pre_answer(switch_channel_t *channel, const
↳ char *file, const char *func, int line)
3484 {
3485     ...
3502     if (switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_INBOUND) {
3503         msg.message_id = SWITCH_MESSAGE_INDICATE_PROGRESS;
3504         msg.from = channel->name;
3505         status = switch_core_session_perform_receive_message(channel->session, &msg, file, func,
↳ line);
3506     }
3507
3508     if (status == SWITCH_STATUS_SUCCESS) {
3509         switch_channel_perform_mark_pre_answered(channel, file, func, line);
3510         switch_channel_audio_sync(channel);

```

与上面类似, 设置 Ring Ready 状态 (如发送 SIP 180 消息)。

```

3518 SWITCH_DECLARE(switch_status_t) switch_channel_perform_ring_ready_value(switch_channel_t
↳ *channel, switch_ring_ready_t rv, const char *file, const char *func, int line)

```

在 Channel 的相关状态下执行 API。API 实际是一个字符串, 如 **uuid_dump <uuid>**, 由命令和参数组成。在此, 首先将传入的字符串在 Session 的内存池中复制一份 (L3561, 因为后面会破坏相关内存)。这里的 **app** 变量名字容易让人想到 Channel 上的 Application, 实际上使用 **cmd** (命

令) 会比较好, 因此读者在此需要知道 **app** 实际上代表了一个 API 命令, 而 **arg** 就是后面的参数 (3564)。L3567 准备了一个标准的 Stream, L3570 是执行一个 API 命令的标准方法, 执行结果会写到 **stream.data** 里。 **stream.data** 是从堆上申请的内存, 用完要释放。

```
3555 static void do_api_on(switch_channel_t *channel, const char *variable)
3556 {
3559     switch_stream_handle_t stream = { 0 };
3560
3561     app = switch_core_session_strdup(channel->session, variable);
3562
3563     if ((arg = strchr(app, ' '))) {
3564         *arg++ = '\\0';
3565     }
3566
3567     SWITCH_STANDARD_STREAM(stream);
3570     switch_api_execute(app, arg, NULL, &stream);
3571     free(stream.data);
3572 }
```

下面就是 **api_on_xxxx** 的回调函数执行机制。取得 (L3582) 并遍历所有通道变量, 如果有匹配 **variable_prefix** 的变量, 则调用我们上面讲的 **do_api_on** 执行之 (L3593, L3597)。L3582 得到的 Event 数据结构最终要释放 (L3602)

```
3575 SWITCH_DECLARE(switch_status_t) switch_channel_api_on(switch_channel_t *channel, const char
↪ *variable_prefix)
3576 {
3582     switch_channel_get_variables(channel, &event);
3583
3584     for (hp = event->headers; hp; hp = hp->next) {
3588         if (!strncasecmp(var, variable_prefix, strlen(variable_prefix))) {
3589             if (...)
3593                 do_api_on(channel, hp->array[i]);
3595             } else {
3597                 do_api_on(channel, val);
3598             }
3599         }
3600     }
3601
3602     switch_event_destroy(&event);
```

在 Channel 上执行 Application, 有异步 (非阻塞, L3632) 和同步 (L3634) 两种。


```
3607 static void do_execute_on(switch_channel_t *channel, const char *variable)
3608 {
3631     if (bg) {
3632         switch_core_session_execute_application_async(channel->session, app, arg);
3633     } else {
3634         switch_core_session_execute_application(channel->session, app, arg);
```

在 Channel 上执行 Application。取得所有全局变量（L3644）以及所有通道变量（L3645），并将他们合并（L3646）。然后遍历（L3648）找到匹配的执行（L3657，L3661）。

```
3638 SWITCH_DECLARE(switch_status_t) switch_channel_execute_on(switch_channel_t *channel, const char
↪ *variable_prefix)
3639 {
3640     ...
3644     switch_core_get_variables(&event);
3645     switch_channel_get_variables(channel, &cevent);
3646     switch_event_merge(event, cevent);
3647
3648     for (hp = event->headers; hp; hp = hp->next) {

3657         do_execute_on(channel, hp->array[i]);
3661         do_execute_on(channel, val);
```

将 Channel 置为应答状态。检查 DTLS 状态（是否已经准备好 RTP 加密传输，L3689），设置应答时间（L3693）。检查 zrtsp（L3697），设置应答标志（L3698）。

L3700 的 `video_mirror_input` 是一个特性，有些终端只能接收特定分辨率的视频（如 352x288），开启这一特性后，如果 FreeSWITCH 给该终端发送的视频分辨率与收到的分辨率不一致，FreeSWITCH 会自动缩放，即收到多大的，发送多大的。

L3706 发送应答事件。如果启用了心跳功能（L3724），会在 L3738 启动心跳功能（如每隔一段时间发送一个 SIP MESSAGE 或 INFO 消息）。

L3755，执行 `execute_on_answer` 回调。L3757 ~ 3760 保证如果 Channel 没经过 Early Media 状态的话（如没收到 183 直接收到 200 OK 应答消息），也可以执行 `on_media` 相关的回调。

L3762，执行 `api_on_answer` 设置的 API。

L3764，发送 Presense 消息。

L3768，如果设置了 `track-calls`，则在数据库中记录当前的呼叫信息，以便 FreeSWITCH 崩溃时可以重启或在另一台服务器上恢复原来的通话。

```

3672 SWITCH_DECLARE(switch_status_t) switch_channel_perform_mark_answered(switch_channel_t *channel, const
↳ char *file, const char *func, int line)
3673 {
...
3689     switch_core_media_check_dtls(channel->session, SWITCH_MEDIA_TYPE_AUDIO);
3690
3693     channel->caller_profile->times->answered = switch_micro_time_now();
3696
3697     switch_channel_check_zrtp(channel);
3698     switch_channel_set_flag(channel, CF_ANSWERED);
3699
3700     if (switch_true(switch_channel_get_variable(channel, "video_mirror_input"))) {
3701         switch_channel_set_flag(channel, CF_VIDEO_MIRROR_INPUT);
3702     }
3703
3704
3705
3706     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_ANSWER) ...
...
3724     if ((var = switch_channel_get_variable(channel, SWITCH_ENABLE_HEARTBEAT_EVENTS_VARIABLE))) {
...
3738         switch_core_session_enable_heartbeat(channel->session, seconds);
3740     }
3741
...
3755     switch_channel_execute_on(channel, SWITCH_CHANNEL_EXECUTE_ON_ANSWER_VARIABLE);
3756
3757     if (!switch_channel_test_flag(channel, CF_EARLY_MEDIA)) {
3758         switch_channel_execute_on(channel, SWITCH_CHANNEL_EXECUTE_ON_MEDIA_VARIABLE);
3759         switch_channel_api_on(channel, SWITCH_CHANNEL_API_ON_MEDIA_VARIABLE);
3760     }
3762     switch_channel_api_on(channel, SWITCH_CHANNEL_API_ON_ANSWER_VARIABLE);
3764     switch_channel_presence(channel, "unknown", "answered", NULL);
3768     switch_core_recovery_track(channel->session);
3770     switch_channel_set_callstate(channel, CCS_ACTIVE);
3772     send_ind(channel, SWITCH_MESSAGE_ANSWER_EVENT, file, func, line);
3774     switch_core_media_check_autoadj(channel->session);

```

应答来话，仅对呼叫电话有效。通过发送 **INDICATE_ANSWER** 消息，L3800 会回调相关的函数执行相应的应答（比如向 SIP 终端发送 200 OK 消息）。

```

3779 SWITCH_DECLARE(switch_status_t) switch_channel_perform_answer(switch_channel_t *channel, const char
↳ *file, const char *func, int line)
3780 {
...
3798     msg.message_id = SWITCH_MESSAGE_INDICATE_ANSWER;
3799     msg.from = channel->name;

```

```
3800     status = switch_core_session_perform_receive_message(channel->session, &msg, file, func, line);
3801
3803     if (status == SWITCH_STATUS_SUCCESS) {
3804         switch_channel_perform_mark_answered(channel, file, func, line);
```

扩展通道变量，如将 `${uuid}` 扩展成实际的 UUID 字符串。

```
3839 SWITCH_DECLARE(char *) switch_channel_expand_variables_check(switch_channel_t *channel, const char
↳ *in, switch_event_t *var_list, switch_event_t *api_list, uint32_t recur)
```

将通道变量转换成字符串，从堆上申请内存。

```
4142 SWITCH_DECLARE(char *) switch_channel_build_param_string(switch_channel_t
↳ *channel, switch_caller_profile_t *caller_profile, const char *prefix)
```

将被叫号码从一个 `channel` 传递到另一个 `other_channel` 上。

```
4272 SWITCH_DECLARE(switch_status_t) switch_channel_pass_callee_id(switch_channel_t *channel,
↳ switch_channel_t *other_channel)
```

将所有通道变量复制到一个新的 `event` 里，产生一个新的 `event`。

```
4298 SWITCH_DECLARE(switch_status_t) switch_channel_get_variables(switch_channel_t *channel, switch_event_t
↳ **event)
4299 {
...
4302     if (channel->variables) {
4303         status = switch_event_dup(event, channel->variables);
4304     } else {
4305         status = switch_event_create(event, SWITCH_EVENT_CHANNEL_DATA);
4306     }
```

取得当前 Channel 对应的 Session。

```
4311 SWITCH_DECLARE(switch_core_session_t *) switch_channel_get_session(switch_channel_t *channel)
4312 {
```

```
4314     return channel->session;
4315 }
```

在 Channel 上设置各种时间。

```
4317 SWITCH_DECLARE(switch_status_t) switch_channel_set_timestamps(switch_channel_t *channel)
```

取得跟本 Channel 桥接的另一个 UUID。

```
4672 SWITCH_DECLARE(const char *) switch_channel_get_partner_uuid(switch_channel_t *channel)
4673 {
4674     if (!(uuid = switch_channel_get_variable(channel, SWITCH_SIGNAL_BOND_VARIABLE))) {
4675         uuid = switch_channel_get_variable(channel, SWITCH_ORIGINATE_SIGNAL_BOND_VARIABLE);
4676     }
4677     return uuid;
4678 }
4679 }
```

如果呼叫 (B-leg) 失败, 根据不同的挂机原因进行相关后续处理。这里主要是处理 `transfer_on_fail` (L4695) 和 `continue_on_fail`, 判断是否需要继续执行还是挂机。其中, `continue_on_fail` 可以取值为 `true` 也可以是具体的挂机原因如 `USER_BUSY` 等, 除 `ATTENDED_TRANSFER` 外 (L4708), FreeSWITCH 会通过各种逻辑组合决定是否 `return` (如 L4742), 直接返回进行后续的处理。

如果有 `transfer_on_fail` (L4756), 则会将当前 Channel 转移 (Transfer, L4792) 到 Dialplan 重新进行路由。

当然, 如果即没有 `continue_on_fail` 也没有 `transfer_on_fail`, FreeSWITCH 还会判断一些额外的条件, 决定是否挂机 (L4818 ~ 4821)。

```
4683 SWITCH_DECLARE(void) switch_channel_handle_cause(switch_channel_t *channel, switch_call_cause_t cause)
4684 {
4685     ...
4695     transfer_on_fail = switch_channel_get_variable(channel, "transfer_on_fail");
4696     tof_data = switch_core_session_strdup(session, transfer_on_fail);
4697     switch_split(tof_data, ' ', tof_array);
4698     transfer_on_fail = tof_array[0];
4699     ...
4708     if (cause != SWITCH_CAUSE_ATTENDED_TRANSFER) {
4711         continue_on_fail = switch_channel_get_variable(channel, "continue_on_fail");
4740         if (continue_on_fail) {
```

```

4741         if (switch_true(continue_on_fail)) {
4742             return;
4743         } else {
4744             for (i = 0; i < argc; i++) {
4745                 if (...) {
4746                     return;
4747                 }
4748             }
4749         }
4750     }
4751 }
4752
4753 ...
4754 if (transfer_on_fail || failure_causes) {
4755     ...
4756     switch_ivr_session_transfer(session, tof_array[1], tof_array[2], tof_array[3]);
4757     ...
4758 }
4759
4760 if (!switch_channel_test_flag(channel, CF_TRANSFER) && !switch_channel_test_flag(channel,
4761 ↪ CF_CONFIRM_BLIND_TRANSFER) &&
4762     switch_channel_get_state(channel) != CS_ROUTING) {
4763     switch_channel_hangup(channel, cause);
4764 }
4765 }
4766 }

```

初始化及销毁全局变量。

```

4824 SWITCH_DECLARE(void) switch_channel_global_init(switch_memory_pool_t *pool)
4825 {
4826     memset(&globals, 0, sizeof(globals));
4827     globals.pool = pool;
4828
4829     switch_mutex_init(&globals.device_mutex, SWITCH_MUTEX_NESTED, pool);
4830     switch_core_hash_init(&globals.device_hash);
4831 }
4832
4833 SWITCH_DECLARE(void) switch_channel_global_uninit(void)
4834 {
4835     switch_core_hash_destroy(&globals.device_hash);
4836 }

```

获取设备状态。下面的函数大部分都是跟设备相关的。

```

4839 static void fetch_device_stats(switch_device_record_t *drec)

```

清除设备记录。

```
4923 SWITCH_DECLARE(void) switch_channel_clear_device_record(switch_channel_t *channel)
4978 }
```

处理设备挂机。

```
4980 SWITCH_DECLARE(void) switch_channel_process_device_hangup(switch_channel_t *channel)
```

检查设备状态。发送 `DEVICE_STATE` 事件（L5132）。

```
5044 static void switch_channel_check_device_state(switch_channel_t *channel, switch_channel_callstate_t
↪ callstate)
5045 {
...
5132     if (switch_event_create(&event, SWITCH_EVENT_DEVICE_STATE) == SWITCH_STATUS_SUCCESS) {
...
5208     if (event) {
5209         switch_event_fire(&event);
5210     }
5212 }
```

往一个设备记录上添加 UUID，假定在被调用时已获得相关的锁。

```
5214 /* assumed to be called under a lock */
5215 static void add_uuid(switch_device_record_t *drec, switch_channel_t *channel)
```

创建设备记录。

```
5242 static switch_status_t create_device_record(switch_device_record_t **drecp, const char *device_id)
5243 {
```

设置设备 ID。

```
5261 SWITCH_DECLARE(const char *) switch_channel_set_device_id(switch_channel_t *channel, const char
↳ *device_id)
```

获取设备记录。

```
5289 SWITCH_DECLARE(switch_device_record_t *) switch_channel_get_device_record(switch_channel_t *channel)
```

释放设备记录。

```
5299 SWITCH_DECLARE(void) switch_channel_release_device_record(switch_device_record_t **drecp)
```

绑定/解除绑定设备状态处理回调函数。

```
5307 SWITCH_DECLARE(switch_status_t)
↳ switch_channel_bind_device_state_handler(switch_device_state_function_t function, void *user_data)

5333 SWITCH_DECLARE(switch_status_t)
↳ switch_channel_unbind_device_state_handler(switch_device_state_function_t function)
```

把 SDP 从一个 Channel (`from_channel`) 传递到另一个 Channel (`to_channel`) 上。传递过程中如果有 `bypass_media_sdp_filter` 过滤器 (L5367)，则将 SDP 处理一下 (L5368) 再传递 (L5373)。

```
5358 SWITCH_DECLARE(switch_status_t) switch_channel_pass_sdp(switch_channel_t
↳ *from_channel, switch_channel_t *to_channel, const char *sdp)
5359 {
...
5364     if (!switch_channel_get_variable(to_channel, SWITCH_B_SDP_VARIABLE)) {
...
5367         if ((var = switch_channel_get_variable(from_channel, "bypass_media_sdp_filter"))) {
5368             if ((patched_sdp = switch_core_media_process_sdp_filter(use_sdp, var, from_channel-
↳ >session))) {
5369                 use_sdp = patched_sdp;
5370             }
5371         }
5373         switch_channel_set_variable(to_channel, SWITCH_B_SDP_VARIABLE, use_sdp);
```

```
5374     }  
...  
5378     return status;  
5379 }
```

至此，Channel 相关的函数都分析完了。这些函数里只有一些简单的业务逻辑（如根据不同的通道变量决定不同的处理策略，处理不同的挂机原因等），具体的业务逻辑是在其它地方实现的，欲知后事如何，且看下节。

4.8 switch_config.c

该文件实现了几个配置相关的函数。貌似仅在 `mod_dialplan_asterisk` 里用到了这些函数。完整性起见，我们不妨也来看下这些代码。

L36 用于打开一个类似于 INI 类型的配置文件。

```
36 SWITCH_DECLARE(int) switch_config_open_file(switch_config_t *cfg, char *file_path)
```

L91 关闭配置文件。

```
91 SWITCH_DECLARE(void) switch_config_close_file(switch_config_t *cfg)
```

找到下一个“键-值”对。

```
101 SWITCH_DECLARE(int) switch_config_next_pair(switch_config_t *cfg, char **var, char **val)
```

具体的实现代码我们就不深究了，毕竟只是一些字符串操作而已。

4.9 switch_console.c

FreeSWITCH 控制台相关代码。默认定义了命令行长度为 1024 字节（L38）。另外，`libedit` 是一个跨平台的库，可以方便支持命令行编辑功能（L40 ~ L41）。当然 `libedit` 也不是必须的，如果没有 `libedit`，FreeSWITCH 也能正常编译运行。

```

38 #define CMD_BUFLen 1024
39
40 #ifdef HAVE_LIBEDIT
41 #include <histedit.h>
48 #else
76 #endif

```

功能键数组，对应 F1 ~ F12 功能键控制（如，在默认配置中按下 F6 会执行 `reloadxml`）。

```

82 static char *console_fnkeys[12];

```

L90，会打开 FreeSWITCH 配置文件读取配置。FreeSWITCH 的配置文件是一个大的 XML，FreeSWITCH 在加载时就已将 XML 解析到一个内存数据结构（`switch_xml_t`）中。L102 打开 XML 配置的并找到 `switch.conf` 节点。默认配置是在 `autoload_configs/switch.conf.xml` 中（注意，文件名和节点名称没有必然的联系，只是它们恰好相同），如：

```

<configuration name="switch.conf" description="Core Configuration">

```

接下来找到 `cli-keybindings` 子节点（L107），遍历子节点找到所有子节点的子节点（有点像绕口令:D），得到一些“键-值”对（`name` 和 `value`，L109 ~ L110），取出这些值并将它们设置到 `console_fnkeys` 数组里（L116）。注意这里使用了 `switch_core_permanent_strdup`，它会在核心内存池中申请内存。`switch_xml_t` 结构用完后需要释放（L121）。

```

90 static switch_status_t console_xml_config(void)
91 {
92     char *cf = "switch.conf";
93     switch_xml_t cfg, xml, settings, param;
94     ...
102     if (!(xml = switch_xml_open_cfg(cf, &cfg, NULL))) {
103         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Open of %s failed\n", cf);
104         return SWITCH_STATUS_TERM;
105     }
106
107     if ((settings = switch_xml_child(cfg, "cli-keybindings"))) {
108         for (param = switch_xml_child(settings, "key"); param; param = param->next) {
109             char *var = (char *) switch_xml_attr_soft(param, "name");
110             char *val = (char *) switch_xml_attr_soft(param, "value");
111             i = atoi(var);

```

```

116         console_fnkeys[i - 1] = switch_core_permanent_strdup(val);
117     }
118 }

```

对比一下实际的 XML 配置文件就能比较容易地理解上述代码了。

```

<configuration name="switch.conf" description="Core Configuration">
  <cli-keybindings>
    <key name="1" value="help"/>
    <key name="2" value="status"/>
    <key name="3" value="show channels"/>
    <key name="4" value="show calls"/>
    <key name="5" value="sofia status"/>
    <key name="6" value="reloadxml"/>
    <key name="7" value="console loglevel 0"/>
    <key name="8" value="console loglevel 7"/>
    <key name="9" value="sofia status profile internal"/>
    <key name="10" value="sofia profile internal siptrace on"/>
    <key name="11" value="sofia profile internal siptrace off"/>
    <key name="12" value="version"/>
  </cli-keybindings>
</configuration>

```

下列函数实现了一个流（Stream）的写函数（回调函数），使用原始格式（raw）写入。L126，`handle` 是一个流句柄，`data` 是要写入的数据，`datalen` 是要写入的数据长度。

写入数据前，先计算需要（`need`）的缓冲区大小（L128），如果缓冲区不够大（L130），则重新申请缓冲区（L134），并重置数据指针和缓冲区大小（L138 ~ L139）。

向一个流写入其实只是简单的 `memcpy`。L142，`handle->data_len` 是流缓冲区中已有数据的长度，因此，`handle->data + handle->data_len` 便是应该写入的位置。

为了保险起见，L145 在数据的最后面写入一个 `'\0'`，因此，即使写入的是一个字符串，也可能正常引用。

```

126 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_stream_raw_write(switch_stream_handle_t *handle,
                                                                    uint8_t *data, switch_size_t datalen)
127 {
128     switch_size_t need = handle->data_len + datalen;
129
130     if (need >= handle->data_size) {
131         void *new_data;

```

```

132     need += handle->alloc_chunk;
133
134     if (!(new_data = realloc(handle->data, need))) {
135         return SWITCH_STATUS_MEMERR;
136     }
137
138     handle->data = new_data;
139     handle->data_size = need;
140 }
141
142 memcpy((uint8_t *) (handle->data) + handle->data_len, data, datalen);
143 handle->data_len += datalen;
144 handle->end = (uint8_t *) (handle->data) + handle->data_len;
145 *(uint8_t *) handle->end = '\0';
146
147 return SWITCH_STATUS_SUCCESS;
148 }

```

与 L126 类似，L150 实现了一个通用的写函数，可以写入类似 `printf()` 方式格式化的字符串。L162~L167 使用 `va_` 一族的函数将字符串格式化，后面的写入方法跟 `_raw_write` 函数差不多，缓冲区不够时也会自动扩展。

```

150 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_stream_write(switch_stream_handle_t *handle,
↪  const char *fmt, ...)
151 {
162     va_start(ap, fmt);
164     if (!(data = switch_vfprintf(fmt, ap))) {
165         ret = -1;
166     }
167     va_end(ap);

```

Stream 的初始化和使用见如下代码段：

```

315     switch_stream_handle_t stream = { 0 };
320     SWITCH_STANDARD_STREAM(stream);
321     switch_assert(stream.data);

```

读者可能注意到上述两个函数是用 `SWITCH_DECLARE_NONSTD` 声明的，这主要是考虑到跨平台遵循 Win32 平台的调用约定，感兴趣的读者可以看一下 `switch_platform.h` 中的宏定义。

其中 L320 是一个宏，我们来看一下 `switch_console.h` 中的宏定义。它首先申请了 `SWITCH_CMD_CHUNK_LEN`（默认为 1024）字节的内存数据缓冲区，并设置了一些数据和回调

函数的指针。我们可以看到，如果在代码中调用 `stream->write_function(...)` 实际上就是调用了 `switch_console_stream_write` 函数。如果缓冲区不够用，就会再申请比所需要内存多 `s.alloc_chunk = SWITCH_CMD_CHUNK_LEN` 大小的内存。另外，从这个宏定义可以看出，L321 的 `switch_assert` 其实是多余的，因为在宏定义里已经存在了。

```
#define SWITCH_STANDARD_STREAM(s) \
    memset(&s, 0, sizeof(s)); s.data = malloc(SWITCH_CMD_CHUNK_LEN); \
    switch_assert(s.data); \
    memset(s.data, 0, SWITCH_CMD_CHUNK_LEN); \
    s.end = s.data; \
    s.data_size = SWITCH_CMD_CHUNK_LEN; \
    s.write_function = switch_console_stream_write; \
    s.raw_write_function = switch_console_stream_raw_write; \
    s.alloc_len = SWITCH_CMD_CHUNK_LEN; \
    s.alloc_chunk = SWITCH_CMD_CHUNK_LEN
```

从文件中读取内容并写入流。L217 打开文件，L221 读，L222 写入 Stream。

```
206 SWITCH_DECLARE(switch_status_t) switch_stream_write_file_contents(switch_stream_handle_t *stream, const
↪ char *path)
207 {
...
217     if ((fd = fopen(path, "r"))) {
...
221         while (switch_fp_read_dline(fd, &line_buf, &llen)) {
222             stream->write_function(stream, "%s", line_buf);
223         }
```

下面是别名 (Alias) 相关的函数。可以通过 `alias` 命令为长的命令行起一个别名，方便输入。如，笔者经常需要在 FreeSWITCH 中打开 SIP Trace 跟踪，就做了个别名：

```
freeswitch> alias add sipf sofia global siptrace on
freeswitch> alias add sipf sofia global siptrace off
```

以后，只需要使用 `sipf` 和 `sipf` 就可以打开和关闭 SIP Trace。

L240 定义了一个函数用于将别名扩展成真正的命令。别名是存储在核心数据库中的。别名没什么值说的，这里值得说的是数据库操作。

L255 打开核心数据库获得一个数据库句柄。如果核心数据库使用 SQLite (L277) 跟使用 ODBC (MySQL 或 PostgreSQL) 的情况下，SQL 语句略微有些差异。这里用到一个 `switch_mprintf`，它

从堆上申请内存并返回格式化后的字符串指针（用完需要记得释放），其中，`%q`和`%w`都类似常用的`%s`，只是前者会对“'”转义，而后者会对“'”和“\”都转义。

L267 执行 SQL 查询。对于查询结果的每一行，都会回调 `alias_callback` 函数。该函数在 L233 定义，其中，`pArg` 是一个返回值，它来自 L267 的 `&r`，`argc` 为列数，`argv` 为列的值数组。这里，将第一列的值（`argv[0]`）返回到 `*r` (L236)，因此，L267 行执行完毕后，`r` 的值就是别名扩展后的字符串。

```

233 static int alias_callback(void *pArg, int argc, char **argv, char **columnNames)
234 {
235     char **r = (char **) pArg;
236     *r = strdup(argv[0]);
237     return -1;
238 }
239
240 SWITCH_DECLARE(char *) switch_console_expand_alias(char *cmd, char *arg)
241 {
242     if (switch_core_db_handle(&db) != SWITCH_STATUS_SUCCESS) ...
243
244     if (switch_cache_db_get_type(db) == SCDB_TYPE_CORE_DB) {
245         sql = switch_mprintf("select command from aliases where alias='%q'", cmd);
246     } else {
247         sql = switch_mprintf("select command from aliases where alias='%w'", cmd);
248     }
249
250     switch_cache_db_execute_sql_callback(db, sql, alias_callback, &r, &errmsg);

```

下面函数执行命令行上输入的命令。L317 获取当前的控制台句柄，L320 初始化一个 Stream，L323 调用 `switch_console_execute` 执行命令（该函数在 L347 实现，后面会讲到）执行结果在 `stream.data` 中，L327 把它打印到控制台上。

```

313 static int switch_console_process(char *xcmd)
314 {
315     FILE *handle = switch_core_get_console();
316
317     SWITCH_STANDARD_STREAM(stream);
318
319     status = switch_console_execute(xcmd, 0, &stream);
320
321     if (status == SWITCH_STATUS_SUCCESS) {
322         if (handle) {
323             fprintf(handle, "\n%s\n", (char *) stream.data);
324             fflush(handle);
325         }
326     }

```

下面就是 `switch_console_execute` 函数。如果命令是一个 `alias`，则扩展成真正的命令（L387），并执行（L389），最终会调用 `switch_api_execute`（L395）执行真正的命令。

```

347 SWITCH_DECLARE(switch_status_t) switch_console_execute(char *xcmd, int rec, switch_stream_handle_t
↪ *istream)
348 {
387     if ((alias = switch_console_expand_alias(cmd, arg)) && alias != cmd) {
389         status = switch_console_execute(alias, ++rec, istream);
392     }
395     status = switch_api_execute(cmd, arg, NULL, istream);

```

下列函数用于打印控制台的输出。可以看到它在日志中加上了当前的时间（L431 ~ L432）。输出可以直接打印到日志里（L434 ~ L435），也可以做为一个日志事件发送出去（L440 ~ L446）。

```

405 SWITCH_DECLARE(void) switch_console_printf(switch_text_channel_t channel, const char *file, const char
↪ *func, int line, const char *fmt, ...)
406 {
431     switch_time_exp_lt(&tm, switch_micro_time_now());
432     switch_strftime_nocheck(date, &retsize, sizeof(date), "%Y-%m-%d %T", &tm);
433
434     if (channel == SWITCH_CHANNEL_ID_LOG) {
435         fprintf(handle, "[%d] %s %s:%d %s() %s", (int) getpid(), date, file, line, func, data);
436         goto done;
437     }
438
439     if (channel == SWITCH_CHANNEL_ID_EVENT &&
440         switch_event_running() == SWITCH_STATUS_SUCCESS && switch_event_create(&event,
↪ SWITCH_EVENT_LOG) == SWITCH_STATUS_SUCCESS) {
441
442         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Log-Data", data);
446         switch_event_fire(&event);
447     }

```

命令补全和回调，暂不多说。

```

470 static int comp_callback(void *pArg, int argc, char **argv, char **columnNames)

588 static int modulename_callback(void *pArg, const char *module_name)

```

```

596 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_available_modules(const char *line,
                                         const char *cursor, switch_console_callback_match_t **matches)

612 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_loaded_modules(const char *line,
                                         const char *cursor, switch_console_callback_match_t **matches)

631 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_interfaces(const char *line, const char
↪ *cursor,
                                         switch_console_callback_match_t **matches)

```

下面我们以 UUID 为例来说一下。有一些命令需要一个 Channel 的 UUID 作为参数,如 `uuid_dump` 命令。在该命令实现时,会增加一个命令补全信息:

```
switch_console_set_complete("add uuid_dump ::console::list_uuid");
```

上面意思说明,在控制台上输入 `uuid_dump` 并按下 TAB 键时,将会调用 `::console::list_uuid` 功能。

该功能是在 L1671 (见下文) 加入的,它映射到一个回调函数 `switch_console_list_uuid`。该函数是在 L667 实现的。当执行到该函数时,它会查询系统中所有的 UUID (L685),或者,如果用户输入了一部分 UUID,则仅查找匹配的 UUID (L682)。总之,L688 执行查询,对查到的每一行,回调 `uuid_callback` 函数 (L658),该函数最终将查询到的结果推到匹配结果列表里去 (L662),进而显示在控制台上。

```

658 static int uuid_callback(void *pArg, int argc, char **argv, char **columnNames)
659 {
660     struct match_helper *h = (struct match_helper *) pArg;
661
662     switch_console_push_match(&h->my_matches, argv[0]);
663     return 0;
664
665 }
666
667 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_uuid(const char *line, const char *cursor,
↪ switch_console_callback_match_t **matches)
668 {

681     if (!zstr(cursor)) {
682         sql = switch_mprintf("select distinct uuid from channels where uuid like '%q%' and
↪ hostname='%q' order by uuid",
683                             cursor, switch_core_get_switchname());
684     } else {

```

```
685     sql = switch_mprintf("select distinct uuid from channels where hostname='%q' order by uuid",
↪   switch_core_get_switchname());
686 }
687
688     switch_cache_db_execute_sql_callback(db, sql, uuid_callback, &h, &errmsg);
```

命令补全函数。在命令行上输入几个字符按 TAB 键后可以自动补全。由于很多数据存在数据库里，因而该函数需要数据库支持。

```
703 SWITCH_DECLARE(unsigned char) switch_console_complete(const char *line, const char *cursor, FILE *
↪   console_out,
704                                                         switch_stream_handle_t *stream, switch_xml_t xml)
```

按下功能键时，执行绑定的命令。

```
952 static unsigned char console_fnkey_pressed(int i)
...
988 static unsigned char console_f1key(EditLine * el, int ch)
989 {
990     return console_fnkey_pressed(1);
991 }
992 static unsigned char console_f2key(EditLine * el, int ch)
993 {
994     return console_fnkey_pressed(2);
995 }
...
```

保存命令历史。

```
976 SWITCH_DECLARE(void) switch_console_save_history(void)
```

命令提示符。

```
1038 char *prompt(EditLine * e)
1039 {
1040     if (*prompt_str == '\0') {
1041         switch_snprintf(prompt_str, sizeof(prompt_str), "freeswitch@%s> ",
↪   switch_core_get_switchname());
```



```
1042     }
1044     return prompt_str;
1045 }
```

控制台线程，无限循环（1053），退出条件是 `running` 变成 0，父进程的 PID 变为 1（L1056，说明我们已经是一个孤儿进程了），或核心已终止（L1061 ~ L1062）。从控制台读取命令（L1066），写入命令历史（L1078），并执行该行命令（L1079）。

```
1047 static void *SWITCH_THREAD_FUNC console_thread(switch_thread_t *thread, void *obj)
1048 {
1053     while (running) {
1056         if (getppid() == 1) break;
1061         switch_core_session_ctl(SCSC_CHECK_RUNNING, &arg);
1062         if (!arg) break;
1065
1066         line = el_gets(el, &count);
1068         if (count > 1) {
1069             if (!zstr(line)) {
1070                 char *cmd = strdup(line);
1078                 history(myhistory, &ev, H_ENTER, line);
1079                 running = switch_console_process(cmd);
1083             }
1084         }
1085         switch_cond_next();
1086     }
```

命令补全函数。

```
1093 static unsigned char complete(EditLine *el, int ch)
1094 {
1095     const LineInfo *lf = el_line(el);
1097     return switch_console_complete(lf->buffer, lf->cursor, switch_core_get_console(), NULL, NULL);
1098 }
```

控制台循环，在有 `libedit` 的情况下（L985）。初始化内存池（L1107），初始化 `editline`（1112 ~ 1114），读取 XML 配置（L1120），绑定功能键（L1122 ~ L1152），绑定命令补全（L1155，回调上面的 `complete` 函数）。初始化命令历史（L1161），启动一个新线程 `console_thread` 处理输入和命令。控制台进入无限循环（L1181 ~ L1188），没什么事可干。如果 FreeSWITCH 从无限循环退出，则清理现场（L1190 ~ L1195，略）。

```

985  #ifdef HAVE_LIBEDIT
...
1101 SWITCH_DECLARE(void) switch_console_loop(void)
1102 {
1107     if (switch_core_new_memory_pool(&pool) != SWITCH_STATUS_SUCCESS) ...
1111
1112     el = el_init(__FILE__, switch_core_get_console(), switch_core_get_console(),
↪ switch_core_get_console());
1113     el_set(el, EL_PROMPT, &prompt);
1114     el_set(el, EL_EDITOR, "emacs");

1120     console_xml_config();
1121     /* Bind the functions to the key */
1122     el_set(el, EL_ADDFN, "f1-key", "F1 KEY PRESS", console_f1key);
1133     el_set(el, EL_ADDFN, "f12-key", "F12 KEY PRESS", console_f12key);
1155     el_set(el, EL_ADDFN, "ed-complete", "Complete argument", complete);
1160
1161     myhistory = history_init();
...
1179     switch_thread_create(&thread, thd_attr, console_thread, pool, pool);
1180
1181     while (running) {
1182         int32_t arg = 0;
1183         switch_core_session_ctl(SCSC_CHECK_RUNNING, &arg);
1184         if (!arg) break;
1187         switch_yield(1000000);
1188     }
1189
...
1196 }

```

上面是在 UNIX 类系统上的函数，在 Windows 系统上，有另外一些系列的函数，我们就不多讲了。

```

1198 #else
1200 #ifdef _MSC_VER
1204
1205 static int console_history(char *cmd, int direction)
...
1244 static int console_bufferInput(char *addchars, int len, char *cmd, int key)
...
1448 static BOOL console_readConsole(HANDLE conIn, char *buf, int len, int *pRed, int *key)
...
1553 #endif

```

如果没有 `libedit`，是使用另外一套循环。初始化 XML 配置（L1567），无限循环（L1575），在

Windows 上（L1593 ~ L1608）和 UNIX 类系统上（L1609 ~ L1649，使用 `select`）使用不同的方法处理键盘输入。获得输入后调用 `switch_console_process`（L1602, L1648）进行处理。

```

1556 SWITCH_DECLARE(void) switch_console_loop(void)
1557 {
...
1567     console_xml_config();
...
1575     while (running) {
1576         int32_t arg;
1577 #ifdef _MSC_VER
1578         int read, key;
1579         HANDLE stdinHandle = GetStdHandle(STD_INPUT_HANDLE);
1580 #else
1581         fd_set rfds, efds;
1582         struct timeval tv = { 0, 20000 };
1583 #endif
1584
1585         switch_core_session_ctl(SCSC_CHECK_RUNNING, &arg);
1586         if (!arg) {
1587             break;
1588         }
1589
1590         if (activity) {
1591             switch_log_printf(SWITCH_CHANNEL_LOG_CLEAN, SWITCH_LOG_CONSOLE, "\nfreeswitch@%s> ",
↵ switch_core_get_switchname());
1592         }
1593 #ifdef _MSC_VER
1601         if (cmd[0]) {
1602             running = switch_console_process(cmd);
1603         }
1607         Sleep(20);
1608 #else
1609         FD_ZERO(&rfds);
1613         if ((activity = select(fileno(stdin) + 1, &rfds, NULL, &efds, &tv
1647         if (cmd[0]) {
1648             running = switch_console_process(cmd);
1649         }
1650 #endif

```

控制台初始化和关闭的函数。

```

1664 SWITCH_DECLARE(switch_status_t) switch_console_init(switch_memory_pool_t *pool)
1665 {
...

```

```

1671     switch_console_add_complete_func("::console::list_uuid",
        (switch_console_complete_callback_t) switch_console_list_uuid);
...
1673 }
1674
1675 SWITCH_DECLARE(switch_status_t) switch_console_shutdown(void)

```

增加和删除命令补全函数。

```

1680 SWITCH_DECLARE(switch_status_t) switch_console_add_complete_func(const char *name,
↪ switch_console_complete_callback_t cb)

1691 SWITCH_DECLARE(switch_status_t) switch_console_del_complete_func(const char *name)

```

清理命令补全过程中的匹配信息。

```

1702 SWITCH_DECLARE(void) switch_console_free_matches(switch_console_callback_match_t **matches)

```

对命令补全结果进行排序。

```

1723 SWITCH_DECLARE(void) switch_console_sort_matches(switch_console_callback_match_t *matches)

```

将匹配结果推到匹配队列里。

```

1781 SWITCH_DECLARE(void) switch_console_push_match_unique(switch_console_callback_match_t **matches, const
↪ char *new_val)

1795 SWITCH_DECLARE(void) switch_console_push_match(switch_console_callback_match_t **matches, const char
↪ *new_val)

```

执行命令补全功能。

```

1818 SWITCH_DECLARE(switch_status_t) switch_console_run_complete_func(const char *func, const char *line,
↪ const char *last_word,
1819                                     switch_console_callback_match_t
↪ **matches)

```

添加补全命令。

```
1836 SWITCH_DECLARE(switch_status_t) switch_console_set_complete(const char *string)
```

设置别名，就是将别名和真正的命令插入数据库。

```
1920 SWITCH_DECLARE(switch_status_t) switch_console_set_alias(const char *string)
```

其实控制台功能主要就是就是命令行编辑，处理用户输入，以及翻看历史命令的功能，而这些使得本文件看起来比较复杂。但它们其实跟 FreeSWITCH 没什么关系，因而，我们也就没有深入解析。无论如何，获得用户输入后，最终会调用 `switch_api_execute` 来执行 API 命令。

4.10 switch_core.c

本文件实现了一些 FreeSWITCH 核心功能函数。

全局变量，全 FreeSWITCH 可见，存储跟 FreeSWITCH 相关的路径（L61）和文件名（L62）。

```
61 SWITCH_DECLARE_DATA switch_directories SWITCH_GLOBAL_dirs = { 0 };
62 SWITCH_DECLARE_DATA switch_filenames SWITCH_GLOBAL_filenames = { 0 };
```

运行时数据，私有数据结构，仅在本文件中可见（L65）。

```
65 struct switch_runtime runtime = { 0 };
```

发送心跳事件。

```
68 static void send_heartbeat(void)
69 {
70     if (switch_event_create(&event, SWITCH_EVENT_HEARTBEAT) == SWITCH_STATUS_SUCCESS) {
104         switch_event_fire(&event);
105     }
106 }
```

检测 IP 和主机名称是否变化。

获取当前的主机名 (L123)，若主机名有变，则发送一个 `SWITCH_EVENT_TRAP` 事件 (L127 ~ L132)。

获取当前的 IPV4 和 IPV6 地址 (L138 ~ L139)。大家可以看到我们常用的 `local_ip_v4` 全局变量其实是在这里设置的 (L156)。若 IP 地址有变，则发送 `SWITCH_EVENT_TRAP` (L174 ~ L184) 事件。若网络中断，也会发送相关事件 (略)。

```

108 static char main_ip4[256] = "";
109 static char main_ip6[256] = "";
110
111 static void check_ip(void)
112 {
123     gethostname(runtime.hostname, sizeof(runtime.hostname));
124     ...
127 } else if (strcmp(hostname, runtime.hostname)) {
128     if (switch_event_create(&event, SWITCH_EVENT_TRAP) == SWITCH_STATUS_SUCCESS) {
132         switch_event_fire(&event);
133     }
135     switch_core_set_variable("hostname", runtime.hostname);
136 }
137
138 check4 = switch_find_local_ip(guess_ip4, sizeof(guess_ip4), &mask, AF_INET);
139 check6 = switch_find_local_ip(guess_ip6, sizeof(guess_ip6), NULL, AF_INET6);
140
156     switch_core_set_variable("local_ip_v4", guess_ip4);
157     switch_core_set_variable("local_mask_v4", inet_ntoa(in));
169     switch_core_set_variable("local_ip_v6", guess_ip6);
173     if (!ok4 || !ok6) {
174         if (switch_event_create(&event, SWITCH_EVENT_TRAP) == SWITCH_STATUS_SUCCESS) {
175             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "condition", "network-address-change");
184             switch_event_fire(&event);

```

心跳回调函数。每当调用到该函数时，发送心跳 (L206)，然后计划下一次应该回调该函数的时间 (L209)，其中，`switch_epoch_time_now` 返回当前时间 (秒)，此处意味着 20 秒后应该再次回调。

```

204 SWITCH_STANDARD_SCHED_FUNC(heartbeat_callback)
205 {
206     send_heartbeat();
207     /* reschedule this task */
209     task->runtime = switch_epoch_time_now(NULL) + 20;
210 }

```

检测 IP 的回调函数，解释同上，每 60 秒执行一次。

```

213 SWITCH_STANDARD_SCHED_FUNC(check_ip_callback)
214 {
215     check_ip();
218     task->runtime = switch_epoch_time_now(NULL) + 60;
219 }

```

心跳和检测 IP 的函数是这样被安装的（L1997 ~ L1999）。`switch_scheduler_add_task` 用于安装一个定时任务，在指定的时间执行相关的回调函数（如 `heartbeat_callback`），在回调函数内部应该设置下一次被回调的时间。

```

1997     switch_scheduler_add_task(switch_epoch_time_now(NULL), heartbeat_callback, "heartbeat", "core", 0,
↪ NULL, SSHF_NONE | SSHF_NO_DEL);
1998
1999     switch_scheduler_add_task(switch_epoch_time_now(NULL), check_ip_callback, "check_ip", "core", 0,
↪ NULL, SSHF_NONE | SSHF_NO_DEL | SSHF_OWN_THREAD);

```

设置和获取当前控制台的文件句柄。

```

222 SWITCH_DECLARE(switch_status_t) switch_core_set_console(const char *console)
223 {
224     if ((runtime.console = fopen(console, "a")) == 0) {

232 SWITCH_DECLARE(FILE *) switch_core_get_console(void)
233 {
234     return runtime.console;
235 }

```

获取当前的控制台窗口大小。

```

240 SWITCH_DECLARE(void) switch_core_screen_size(int *x, int *y)

```

返回数据通道，默认为控制台。主要用于写日志。

```

265 SWITCH_DECLARE(FILE *) switch_core_data_channel(switch_text_channel_t channel)
266 {

```

```
267     FILE *handle = stdout;
268
269     switch (channel) {
270     case SWITCH_CHANNEL_ID_LOG:
271     case SWITCH_CHANNEL_ID_LOG_CLEAN:
272         handle = runtime.console;
273         break;
274     default:
275         handle = runtime.console;
276         break;
277     }
279     return handle;
280 }
```

设置、添加和获取状态处理函数。通过指定一个回调函数，在相应的状态改变时会发生回调。

```
283 SWITCH_DECLARE(void) switch_core_remove_state_handler(const switch_state_handler_table_t
↪ *state_handler)

308 SWITCH_DECLARE(int) switch_core_add_state_handler(const switch_state_handler_table_t *state_handler)

325 SWITCH_DECLARE(const switch_state_handler_table_t *) switch_core_get_state_handler(int index)
```

将核心的变量以（Key=Value）的文本格式写到 Stream 里。

```
335 SWITCH_DECLARE(void) switch_core_dump_variables(switch_stream_handle_t *stream)
336 {
337     switch_event_header_t *hi;
338
339     switch_mutex_lock(runtime.global_mutex);
340     for (hi = runtime.global_vars->headers; hi; hi = hi->next) {
341         stream->write_function(stream, "%s=%s\n", hi->name, hi->value);
342     }
343     switch_mutex_unlock(runtime.global_mutex);
344 }
```

获取主机名（L346），获取 Switch 实例名称（L351），获取 Domain（L357），获取全部（L376）或单个（L385）全局变量。L394 在获取全局变量的时候会自我复制一份（需要释放），L409 则在内存池中复制（无须专门释放）。


```
346 SWITCH_DECLARE(const char *) switch_core_get_hostname(void)
347 {
348     return runtime.hostname;
349 }
350
351 SWITCH_DECLARE(const char *) switch_core_get_switchname(void)
352 {
353     if (!strcmp(runtime.switchname)) return runtime.switchname;
354     return runtime.hostname;
355 }
356
357 SWITCH_DECLARE(char *) switch_core_get_domain(switch_bool_t dup)
375
376 SWITCH_DECLARE(switch_status_t) switch_core_get_variables(switch_event_t **event)
384
385 SWITCH_DECLARE(char *) switch_core_get_variable(const char *varname)
393
394 SWITCH_DECLARE(char *) switch_core_get_variable_dup(const char *varname)
408
409 SWITCH_DECLARE(char *) switch_core_get_variable_pdup(const char *varname, switch_memory_pool_t *pool)
```

取消所有全部变量设置。

```
424 static void switch_core_unset_variables(void)
```

设置全局变量。

```
432 SWITCH_DECLARE(void) switch_core_set_variable(const char *varname, const char *value)
```

有条件的设置全局变量。当且仅当变量 `varname` 的值为 `val2` 时，才将其值设为 `value`。

```
453 SWITCH_DECLARE(switch_bool_t) switch_core_set_var_conditional(const char *varname, const char *value,
↳ const char *val2)
```

上述函数的使用实例如下：

```
freeswitch@seven.local> global_setvar a=b
+OK
```

```
freeswitch@seven.local> global_getvar a
b
freeswitch@seven.local> global_setvar a=c =c
+OK
freeswitch@seven.local> global_getvar a
b
freeswitch@seven.local> global_setvar a=c =b
+OK
freeswitch@seven.local> global_getvar a
c
```

获取核心运行时 UUID。

```
484 SWITCH_DECLARE(char *) switch_core_get_uuid(void)
485 {
486     return runtime.uuid_str;
487 }
```

L490 是一个服务线程，它会不停地读取音频或视频，并丢弃。虽然使用的地方不多，但这里的方法和函数都很有代表性，因此我们也详细介绍一下。

L492 行的 obj 是从 L573 传入的，它实际上是一个 **session** 指针。在读之前，先获取一个 Session 上的读锁（L499），另外，还需要锁定读数据帧的一个锁（L503）。L505 从 Session 中取得当前的 Channel。L507 在 Channel 上设置一个 **CF_SERVICE** 标志，标志我们要读取数据了。只要该标志一直存在，就循环（L508）。

进入循环后，如果要读音频（L510），则读取之（L511），读到后会得到一个 **read_frame** 指针，指向读到的数据帧。如果返回值是（L512 ~ L514）的任何一个，则什么也不做，继续循环；否则，清除 **CF_SERVICE** 标志（L517），退出循环，进而后面会退出整个线程。

读取视频的函数也类似（L522 ~ L530，代码略），其中 **CF_VIDEO** 标志该 Channel 支持视频。

退出循环后，释放锁（L535，L540），清除相关标志（L537 ~ L538），并返回（L542）。

```
490 static void *SWITCH_THREAD_FUNC switch_core_service_thread(switch_thread_t *thread, void *obj)
491 {
492     switch_core_session_t *session = obj;
493     switch_channel_t *channel;
494     switch_frame_t *read_frame;
495
499     if (switch_core_session_read_lock(session) != SWITCH_STATUS_SUCCESS) {
500         return NULL;
501     }
```

```

502
503     switch_mutex_lock(session->frame_read_mutex);
504     channel = switch_core_session_get_channel(session);
505
506     switch_channel_set_flag(channel, CF_SERVICE);
507     while (switch_channel_test_flag(channel, CF_SERVICE)) {
508         if (switch_channel_test_flag(channel, CF_SERVICE_AUDIO)) {
509             switch (switch_core_session_read_frame(session, &read_frame, SWITCH_IO_FLAG_NONE, 0)) {
510                 case SWITCH_STATUS_SUCCESS:
511                 case SWITCH_STATUS_TIMEOUT:
512                 case SWITCH_STATUS_BREAK:
513                     break;
514                 default:
515                     switch_channel_clear_flag(channel, CF_SERVICE);
516                     break;
517             }
518         }
519     }
520
521     if (switch_channel_test_flag(channel, CF_SERVICE_VIDEO) && switch_channel_test_flag(channel,
522 ↪ CF_VIDEO)) {
523     }
524 }
525
526     switch_mutex_unlock(session->frame_read_mutex);
527     switch_channel_clear_flag(channel, CF_SERVICE_AUDIO);
528     switch_channel_clear_flag(channel, CF_SERVICE_VIDEO);
529     switch_core_session_rwlock_unlock(session);
530     return NULL;
531 }

```

上述线程是这样启动的。设置 `CF_SERVICE_AUDIO` 和 `CF_SERVICE_VIDEO` 标志（L570 ~ L571），并启动一个线程（L573），新的线程执行上面的 `switch_core_service_thread` 函数，传入的参数是 `session`。

```

562 SWITCH_DECLARE(void) switch_core_service_session_av(switch_core_session_t *session, switch_bool_t
563 ↪ audio, switch_bool_t video)
564 {
565     if (audio) switch_channel_set_flag(channel, CF_SERVICE_AUDIO);
566     if (video) switch_channel_set_flag(channel, CF_SERVICE_VIDEO);
567
568     switch_core_session_launch_thread(session, (void (*)(switch_thread_t *, void
569 ↪ *))switch_core_service_thread, session);
570 }

```

用于退出上面启动的线程。清除掉 `CF_SERVICE` 标志后（L554），上面的线程会退出循环（L508），

进而退出整个线程，只要循环不会阻塞在 `_read_frame` (L511) 或 `_read_video_frame` 操作上（由 L558 发送一个 `BREAK` 信号保证）。

```
545 /* Either add a timeout here or make damn sure the thread cannot get hung somehow (my preference) */
546 SWITCH_DECLARE(void) switch_core_thread_session_end(switch_core_session_t *session)
547 {
548     switch_channel_clear_flag(channel, CF_SERVICE);
549     switch_channel_clear_flag(channel, CF_SERVICE_AUDIO);
550     switch_channel_clear_flag(channel, CF_SERVICE_VIDEO);
551     switch_core_session_kill_channel(session, SWITCH_SIG_BREAK);
552 }
```

启动一个线程，并回调函数 `func`，可以传入一个参数 `obj`。可以传入一个内存池指针，如果 `pool` 为 `NULL`，则会自动创建一个内存池。

```
590 SWITCH_DECLARE(switch_thread_t *) switch_core_launch_thread(switch_thread_start_t func, void *obj,
↪ switch_memory_pool_t *pool)
```

设置一些全局的路径。具体逻辑略，下面代码供参考该函数都设置了哪些路径。

```
622 SWITCH_DECLARE(void) switch_core_set_globals(void)
623 {
624     ...
625     switch_assert(SWITCH_GLOBAL_dirs.base_dir);
626     switch_assert(SWITCH_GLOBAL_dirs.mod_dir);
627     switch_assert(SWITCH_GLOBAL_dirs.lib_dir);
628     switch_assert(SWITCH_GLOBAL_dirs.conf_dir);
629     switch_assert(SWITCH_GLOBAL_dirs.log_dir);
630     switch_assert(SWITCH_GLOBAL_dirs.run_dir);
631     switch_assert(SWITCH_GLOBAL_dirs.db_dir);
632     switch_assert(SWITCH_GLOBAL_dirs.script_dir);
633     switch_assert(SWITCH_GLOBAL_dirs.htdocs_dir);
634     switch_assert(SWITCH_GLOBAL_dirs.grammar_dir);
635     switch_assert(SWITCH_GLOBAL_dirs.fonts_dir);
636     switch_assert(SWITCH_GLOBAL_dirs.images_dir);
637     switch_assert(SWITCH_GLOBAL_dirs.recordings_dir);
638     switch_assert(SWITCH_GLOBAL_dirs.sounds_dir);
639     switch_assert(SWITCH_GLOBAL_dirs.certs_dir);
640     switch_assert(SWITCH_GLOBAL_dirs.temp_dir);
641     switch_assert(SWITCH_GLOBAL_dirs.data_dir);
642     switch_assert(SWITCH_GLOBAL_dirs.localstate_dir);
```

```
895     switch_assert(SWITCH_GLOBAL_filenames.conf_name);
896 }
```

设置进程权限，仅用于 Solaris 操作系统（L901）。

```
899 SWITCH_DECLARE(int32_t) switch_core_set_process_privileges(void)
900 {
901     #ifdef SOLARIS_PRIVILEGES
902     #endif
903     return 0;
904 }
```

设置 FreeSWITCH 进程为低优先级（L934），或实时优先级（L970），或普通优先级（L1049），或自动设置最优的优先级（1054）。取得当前的 CPU 数目（L1044）。

```
934 SWITCH_DECLARE(int32_t) set_low_priority(void)

970 SWITCH_DECLARE(int32_t) set_realtime_priority(void)

1044 SWITCH_DECLARE(uint32_t) switch_core_cpu_count(void)
1045 {
1046     return runtime.cpu_count;
1047 }
1048
1049 SWITCH_DECLARE(int32_t) set_normal_priority(void)
1050 {
1051     return 0;
1052 }
1053
1054 SWITCH_DECLARE(int32_t) set_auto_priority(void)
1055 {
1056     if (!runtime.cpu_count) runtime.cpu_count = 1;
1057
1058     return set_realtime_priority();
1059 }
```

改变进程运行时使用的有效的用户和组。一般情况下，FreeSWITCH 应该首先以 `root` 用户启动，设置一些优先级之、Limit 等只有 `root` 用户才能进行的操作以后，再切换到普通用户的身份执行后续的代码。这样的话，既然后面的代码有漏洞，破坏者也无法获得 `root` 权限。

下面代码会调用 `setuid` 和 `setgid` 来改变用户和组。

```
1072 SWITCH_DECLARE(int32_t) change_user_group(const char *user, const char *group)
```

MIME⁵类型相关的函数。

L1176, 是 FreeSWITCH 核心的一个无限循环。若系统需要在后台运行 (L1182), 则在 Windows 平台上调用 `WaitForSingleObject` 等待 FreeSWITCH 停止 (L1187), 或在 Linux 等平台上无限循环 (L1190 ~ L1192)。若在前面运行, 则执行 `switch_console_loop` (L1196) 等待键盘输入。

```
1176 SWITCH_DECLARE(void) switch_core_runtime_loop(int bg)
1177 {
1182     if (bg) {
1183 #ifdef WIN32
1184         switch_snprintf(path, sizeof(path), "Global\\Freeswitch.%d", getpid());
1185         shutdown_event = CreateEvent(NULL, FALSE, FALSE, path);
1186         if (shutdown_event) {
1187             WaitForSingleObject(shutdown_event, INFINITE);
1188         }
1189 #else
1190         while (runtime.running) {
1191             switch_yield(1000000);
1192         }
1193 #endif
1194     } else {
1196         switch_console_loop();
1197     }
1198 }
```

L1201 和 L1209, 在扩展名 (如 `.html`) MIME 类型 (如 `text/html`) 间转换。实际上就是查散列表。该散列表是使用 `switch_core_mime_add_type` 函数创建的 (L1222), 系统启动时会调用 `load_mime_types` (L1262) 从 `mime.types` (L1264) 文件中读出 (L1279) 相应的对应关系并插入散列表 (L1298)。

```
1201 SWITCH_DECLARE(const char *) switch_core_mime_ext2type(const char *ext)
1202 {
1206     return (const char *) switch_core_hash_find(runtime.mime_types, ext);
1207 }
1208
1209 SWITCH_DECLARE(const char *) switch_core_mime_type2ext(const char *mime)
1210 {
```

⁵参 见: <https://zh.wikipedia.org/wiki/%E5%A4%9A%E7%94%A8%E9%80%94%E4%BA%92%E8%81%AF%E7%B6%B2%E9%83%B5%E4%BB%B6%E6%93%B4%E5%B1%95>

```
1214     return (const char *) switch_core_hash_find(runtime.mime_type_exts, mime);
1215 }
```

L1217, 返回 MIME 类型的 Hash Index, 以便遍历。

```
1217 SWITCH_DECLARE(switch_hash_index_t *) switch_core_mime_index(void)
1218 {
1219     return switch_core_hash_first(runtime.mime_types);
1220 }
```

L1221, 将 MIME 类型插入哈希表。

```
1222 SWITCH_DECLARE(switch_status_t) switch_core_mime_add_type(const char *type, const char *ext)
1223 {
1224     if (!switch_core_hash_find(runtime.mime_types, ext)) {
1225         switch_core_hash_insert(runtime.mime_types, argv[x], ptype);
1226     }
1227     if (!is_mapped_type) {
1228         switch_core_hash_insert(runtime.mime_type_exts, ptype,
↪ switch_core_permanent_strdup(argv[x]));
1229         is_mapped_type = 1;
1230     }
1231 }
```

L1262 加载 MIME 表。就是从一个配置文件中依次读出第一行 (L1279), 然后解析并调用 L1222 中的函数插入一个哈希表, 备用。

```
1262 static void load_mime_types(void)
1263 {
1264     char *cf = "mime.types";
1265     mime_path = switch_mprintf("%s/%s", SWITCH_GLOBAL_dirs.conf_dir, cf);
1266     fd = fopen(mime_path, "rb");
1267
1268     while ((switch_fp_read_dline(fd, &line_buf, &llen))) {
1269         switch_core_mime_add_type(type, p);
1270     }
1271 }
```

L1316, 设置系统资源限制, 如堆栈大小 (L1331), 最大打开文件数 (L1338)等。FreeSWITCH 本身使用很少的堆栈空间 (主要用于多线程环境), 但我们发现如果某些第三方库需要较大的栈空间才能工作的话, 需要想办法绕过这个限制。

```
1316 SWITCH_DECLARE(void) switch_core_setrlimits(void)
1330     rlp.rlim_cur = SWITCH_THREAD_STACKSIZE;
1331     rlp.rlim_max = SWITCH_SYSTEM_THREAD_STACKSIZE;
1338     setrlimit(RLIMIT_NOFILE, &rlp);
```

以下数据结构和函数处理 FreeSWITCH 内部的 IP 地址列表，主要用于 ACL。

```
1361 typedef struct {
1362     switch_memory_pool_t *pool;
1363     switch_hash_t *hash;
1364 } switch_ip_list_t;
1365
1366 static switch_ip_list_t IP_LIST = { 0 };
```

检查对应的 IP 地址是否在相关的列表中，并返回一个 `token`，成功则返回 `SWITCH_TRUE`。

```
1368 SWITCH_DECLARE(switch_bool_t) switch_check_network_list_ip_token(const char *ip_str, const char
↪ *list_name, const char **token)
```

L1442 ~ 1668，装入 ACL 列表。其中，L1452 找到自己本地的 IP，该函数将尝试连接一个公网地址，并且计算自己应该使用哪个本地 IP（`local_ip_v4` 及 `local_ip_v6`），如果主机无法连接互联网，则返回的可能是本地 `loopback` 地址，如 `127.0.0.1`。

```
1442 SWITCH_DECLARE(void) switch_load_network_lists(switch_bool_t reload)
1443 {
```

先找到本地 IP。

```
1452     switch_find_local_ip(guess_ip, sizeof(guess_ip), &mask, AF_INET);
```

```
1471     tmp_name = "rfc6598.auto";
...
1477     tmp_name = "rfc1918.auto";
1478     switch_network_list_create(&rfc_list, tmp_name, SWITCH_FALSE, IP_LIST.pool);
```

```

1479     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Created ip list %s default (deny)\n",
↪     tmp_name);
1480     switch_network_list_add_cidr(rfc_list, "10.0.0.0/8", SWITCH_TRUE);
1481     switch_network_list_add_cidr(rfc_list, "172.16.0.0/12", SWITCH_TRUE);
1482     switch_network_list_add_cidr(rfc_list, "192.168.0.0/16", SWITCH_TRUE);
1483     switch_network_list_add_cidr(rfc_list, "fe80::/10", SWITCH_TRUE);
1484     switch_core_hash_insert(IP_LIST.hash, tmp_name, rfc_list);

```

自动计算并生成一个 `.auto` 的列表。

```

1486     tmp_name = "wan.auto";
1497     tmp_name = "wan_v6.auto";
1505     tmp_name = "wan_v4.auto";
1517     tmp_name = "any_v6.auto";
1524     tmp_name = "any_v4.auto";
1531     tmp_name = "nat.auto";
1543     tmp_name = "loopback.auto";
1550     tmp_name = "localnet.auto";

```

然后打开 `acl.conf`，根据配置文件进一步配置更多的列表。

```

1560     if ((xml = switch_xml_open_cfg("acl.conf", &cfg, NULL))) {

```

如果 `acl.conf` 中有 `domain`，则查找 `domain` 中所有用户，找到对应用户的 `cidr` 属性，也一起增加到列表里，为该 `cidr` 建立一个列表项。这样，就将该用户与某一 `cidr` 规定的 IP 地址或地址段关联起来，所有来自这些 IP 的呼叫都认为来自这个用户，可以不用再经过 Challenge 验证（即仅验证来源 IP）。

```

1618         if ((ut = switch_xml_child(x_domain, "users"))) {
1622             for (ut = switch_xml_child(x_domain, "user"); ut; ut = ut->next) {
1623                 const char *user_cidr = switch_xml_attr(ut, "cidr");
1624                 const char *id = switch_xml_attr(ut, "id");
1634                 for (gts = switch_xml_child(x_domain, "groups"); gts; gts = gts->next) {
1635                     for (gt = switch_xml_child(gts, "group"); gt; gt = gt->next) {
1636                         for (uts = switch_xml_child(gt, "users"); uts; uts = uts->next) {
1637                             for (ut = switch_xml_child(uts, "user"); ut; ut = ut->next) {
1638                                 const char *user_cidr = switch_xml_attr(ut, "cidr");
1639                                 const char *id = switch_xml_attr(ut, "id");
1640
1641                                 if (id && user_cidr) {

```

设置并返回最大（L1670）或最小（L1710）DTMF 间隔。

```
1670 SWITCH_DECLARE(uint32_t) switch_core_max_dtmf_duration(uint32_t duration)
1687 SWITCH_DECLARE(uint32_t) switch_core_default_dtmf_duration(uint32_t duration)
1710 SWITCH_DECLARE(uint32_t) switch_core_min_dtmf_duration(uint32_t duration)
```

CPU 亲缘性。可以将某一进程固定在某一 CPU 上，避免多 CPU 的时候切换的开销。

```
1729 SWITCH_DECLARE(switch_status_t) switch_core_thread_set_cpu_affinity(int cpu)
```

产生一个 FreeSWITCH 序列字符串，存放到 `conf/freeswitch.serial` 文件中。

```
1758 #ifdef ENABLE_ZRTP
1759 static void switch_core_set_serial(void)
```

检测是否有相应的 flag。

```
1806 SWITCH_DECLARE(int) switch_core_test_flag(int flag)
1807 {
1808     return switch_test_flag(&runtime, flag);
1809 }
```

核心初始化。初始化一些核心的参数和数据结构等。代码比较直观，不多解释。

```
1812 SWITCH_DECLARE(switch_status_t) switch_core_init(switch_core_flag_t flags, switch_bool_t console,
↪ const char **err)
1813 {
    //核心的运行时数据。
1825     memset(&runtime, 0, sizeof(runtime));
1826     gethostname(runtime.hostname, sizeof(runtime.hostname));
    ...
1865     if (sqlite3_initialize() != SQLITE_OK) {
    ...
1871     if (apr_initialize() != SWITCH_STATUS_SUCCESS) {
    ...
1876     if (!(runtime.memory_pool = switch_core_memory_init())) {
    ...
```

```

1882     switch_dir_make_recursive(SWITCH_GLOBAL_dirs.base_dir, SWITCH_DEFAULT_DIR_PERMS,
↪ runtime.memory_pool);
1883     switch_dir_make_recursive(SWITCH_GLOBAL_dirs.mod_dir, SWITCH_DEFAULT_DIR_PERMS,
↪ runtime.memory_pool);
1884     switch_dir_make_recursive(SWITCH_GLOBAL_dirs.conf_dir, SWITCH_DEFAULT_DIR_PERMS,
↪ runtime.memory_pool);
...
1898     switch_mutex_init(&runtime.uuid_mutex, SWITCH_MUTEX_NESTED, runtime.memory_pool);
...
1912     load_mime_types();
...
1922     SSL_library_init();
1923     switch_ssl_init_ssl_locks();
1924     switch_curl_init();
...
1926     switch_core_set_variable("hostname", runtime.hostname);
1927     switch_find_local_ip(guess_ip, sizeof(guess_ip), &mask, AF_INET);
1928     switch_core_set_variable("local_ip_v4", guess_ip);1932
...
1933     switch_find_local_ip(guess_ip, sizeof(guess_ip), NULL, AF_INET6);
1934     switch_core_set_variable("local_ip_v6", guess_ip);
1935     switch_core_set_variable("base_dir", SWITCH_GLOBAL_dirs.base_dir);
...
1939     switch_core_set_variable("conf_dir", SWITCH_GLOBAL_dirs.conf_dir);
...
1955 #ifdef ENABLE_ZRTP
1956     switch_core_set_serial();
1957 #endif
1958     switch_console_init(runtime.memory_pool);
1959     switch_event_init(runtime.memory_pool);
1960     switch_channel_global_init(runtime.memory_pool);
1961
1962     if (switch_xml_init(runtime.memory_pool, err) != SWITCH_STATUS_SUCCESS) {
...
1967     if (switch_test_flag((&runtime), SCF_USE_AUTO_NAT)) {
1968         switch_nat_init(runtime.memory_pool, switch_test_flag((&runtime), SCF_USE_NAT_MAPPING));
1969     }
1970
1971     switch_log_init(runtime.memory_pool, runtime.colorize_console);
...
// 如果设置了 SCF_MINIMAL, 就此返回
1977     if (flags & SCF_MINIMAL) return SWITCH_STATUS_SUCCESS;
1978     // 否则, 解析 switch.conf, 继续初始化
1979     switch_load_core_config("switch.conf");
1980
1981     switch_core_state_machine_init(runtime.memory_pool);
1982
1983     if (switch_core_sqldb_start(...
1987     switch_core_media_init();
1988     switch_scheduler_task_thread_start();

```

```

1990     switch_nat_late_init();
1992     switch_rtp_init(runtime.memory_pool);
...     // 增加定时任务
1997     switch_scheduler_add_task(switch_epoch_time_now(NULL), heartbeat_callback, "heartbeat", "core", 0,
↪ NULL, SSHF_NONE | SSHF_NO_DEL);
1999     switch_scheduler_add_task(switch_epoch_time_now(NULL), check_ip_callback, "check_ip", "core", 0,
↪ NULL, SSHF_NONE | SSHF_NO_DEL | SSHF_OWN_THREAD);
2001     switch_uuid_get(&uuid);
2002     switch_uuid_format(runtime.uuid_str, &uuid);
2003     switch_core_set_variable("core_uuid", runtime.uuid_str);
2006     return SWITCH_STATUS_SUCCESS;
2007 }

```

处理 SIGBUS 信号。

```

2010 #ifdef TRAP_BUS
2011 static void handle_SIGBUS(int sig)
2012 {
2013     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG1, "Sig BUS!\n");
2014     return;
2015 }
2016 #endif

```

SIGHUP 回调函数。如果 FreeSWITCH 进程收到 SIGHUP 信号，则执行该回调。SIGHUP 通常由 `kill -HUP <pid>` 命令产生。实际上，它只是触发了一个 `SWITCH_EVENT_TRAP` 事件，所有订阅该事件的线程都可以进行进一步的处理。

```

2018 static void handle_SIGHUP(int sig)
2019 {
2020     if (sig) {
2021         switch_event_t *event;
2023         if (switch_event_create(&event, SWITCH_EVENT_TRAP) == SWITCH_STATUS_SUCCESS) {
2024             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Trapped-Signal", "HUP");
2025             switch_event_fire(&event);
2026         }
2027     }
2028     return;
2029 }

```

设置默认的打包间隔 (`ptime`)。

```
2032 SWITCH_DECLARE(uint32_t) switch_default_ptime(const char *name, uint32_t number)
2033 {
2034     uint32_t *p;
2036     if ((p = switch_core_hash_find(runtime.ptimes, name))) {
2037         return *p;
2038     }
2040     return 20;
2041 }
```

设置默认的采样率（[rate](#)）。

```
2043 SWITCH_DECLARE(uint32_t) switch_default_rate(const char *name, uint32_t number)
2044 {
2046     if (!strcasecmp(name, "opus")) {
2047         return 48000;
2048     } else if (!strcasecmp(name, "h26", 3)) { // h26x
2049         return 90000;
2050     } else if (!strcasecmp(name, "vp", 2)) { // vp8, vp9
2051         return 90000;
2052     }
2054     return 8000;
2055 }
```

L2057, 有些编码如 iLBC、iSAC、G723 等默认的打包间隔是 30ms。

L2059, 加载核心的设置, 参见 [switch.conf](#) 及 [post_load_switch.conf](#)。

```
2057 static uint32_t d_30 = 30;
2057 static void switch_load_core_config(const char *file)
```

Banner。

```
2348 SWITCH_DECLARE(const char *) switch_core_banner(void)
```

L2370, 初始化并加载模块。L2380 初始化核心参数及数据结构。L2399 加载模块。模块加载完成后, L2407 再加载 [post_load_switch.conf](#)。L2411 产生一个事件。

```

2371 SWITCH_DECLARE(switch_status_t) switch_core_init_and_modload(switch_core_flag_t flags, switch_bool_t
↳ console, const char **err)
2372 {
...
2380     if (switch_core_init(flags, console, err) != SWITCH_STATUS_SUCCESS) {
2381         return SWITCH_STATUS_GENERR;
2382     }
2383
2392     switch_core_set_signal_handlers();
2393     switch_load_network_lists(SWITCH_FALSE);
2395     switch_msrp_init();
2398     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Loading Modules.\n");
2399     if (switch_loadable_module_init(SWITCH_TRUE) != SWITCH_STATUS_SUCCESS) {
2403     }
2404
2405     switch_load_network_lists(SWITCH_FALSE);
2407     switch_load_core_config("post_load_switch.conf");
2409     switch_core_set_signal_handlers();
2416     switch_core_screen_size(&x, NULL);
2443     switch_clear_flag((&runtime), SCF_NO_NEW_SESSIONS);

```

L2445, 如果设置了 `api_on_startup`, 则执行一个 API。

```

2445     if ((cmd = switch_core_get_variable_dup("api_on_startup"))) {
2446         switch_stream_handle_t stream = { 0 };
2447         SWITCH_STANDARD_STREAM(stream);
2448         switch_console_execute(cmd, 0, &stream);
2449         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Startup command [%s] executed.
↳ Output:\n%s\n", cmd, (char *)stream.data);
2450         free(stream.data);
2451         free(cmd);
2452     }

```

计算某一时刻对应的年月日时分秒等。

```

2458 SWITCH_DECLARE(void) switch_core_measure_time(switch_time_t total_ms, switch_core_time_duration_t
↳ *duration)

```

FreeSWITCH 自启动以来运行了多长时间。

```

2474 SWITCH_DECLARE(switch_time_t) switch_core_uptime(void)
2475 {

```

```
2476     return switch_mono_micro_time_now() - runtime.initiated;
2477 }
```

Windows shutdown。

```
2481 #ifdef _MSC_VER
2482 static void win_shutdown(void)
2500 #endif
```

设置信号回调。其中，L2504 设置当收到 **SIGINT** 时忽略，即按下 Ctrl+C 不会停止 FreeSWITCH。另外，L2523 ~ 2526，**SIGUSR1** 和 **SIGHUP** 处理方式相同。

```
2502 SWITCH_DECLARE(void) switch_core_set_signal_handlers(void)
2503 {
2504     /* set signal handlers */
2505     signal(SIGINT, SIG_IGN);
2524 #ifdef SIGUSR1
2525     signal(SIGUSR1, handle_SIGHUP);
2526 #endif
2527     signal(SIGHUP, handle_SIGHUP);
2528 }
```

返回 **debug_level**。

```
2530 SWITCH_DECLARE(uint32_t) switch_core_debug_level(void)
2531 {
2532     return runtime.debug_level;
2533 }
```

L2536 ~ L2858 是核心控制函数，主要完成 **fsctl** 相关的命令。如 L2790 修改调试级别、L2807 修改最大并发数、L2837 修改每秒最大启动的 Session 数量等、L2885 回收内存等。

```
2536 SWITCH_DECLARE(int32_t) switch_core_session_ctl(switch_session_ctl_t cmd, void *val)
2537 {
2549     switch (cmd) {
2550     case SCSC_RECOVER: // recover 相关
2589     case SCSC_DEBUG_SQL: // debug SQL
2600     case SCSC_VERBOSE_EVENTS:
```

```

    // 有些 Channel 相关的事件没有完整的 Channel 信息, 该参数让所有事件都带尽量多的信息。
    // 会增加事件中的头域, 多占用资源
2612     case SCSC_API_EXPANSION: // 是否 expand API
2624     case SCSC_THREADED_SYSTEM_EXEC: // 执行 system 函数时在新的线程中执行还不是 fork 一个新进程
2636     case SCSC_CALIBRATE_CLOCK:
2637         switch_time_calibrate_clock();
2639     case SCSC_FLUSH_DB_HANDLES: // 释放不用的 DB 连接
2640         switch_cache_db_flush_handles();
2642     case SCSC_SEND_SIGHUP: // 相当于自己给自己发 HUP 信号
2643         handle_SIGHUP(1);
2645     case SCSC_SYNC_CLOCK: // 立即与操作系统同步时钟, FreeSWITCH 内部会自己计时
2646         switch_time_sync();
2649     case SCSC_SYNC_CLOCK_WHEN_IDLE: // 在空闲时才同步时钟
2650         newintval = switch_core_session_sync_clock();
2652     case SCSC_SQL: // 停止或继续使用核心的 DB
2659     case SCSC_PAUSE_ALL: // 核心暂停, 不允许建立新的 Session
2666     case SCSC_PAUSE_INBOUND: // 不允许呼入
2673     case SCSC_PAUSE_OUTBOUND: // 不允许呼出
2680     case SCSC_HUPALL: // 挂掉所有通话
2683     case SCSC_CANCEL_SHUTDOWN: // 关闭 FreeSWITCH
2686     case SCSC_SAVE_HISTORY: // 保存命令历史
2689     case SCSC_CRASH: // 自杀
2694     case SCSC_SHUTDOWN_NOW: // 立即关闭
2698     case SCSC_REINCARNATE_NOW: // 立即关闭, 并重启
2702     case SCSC_SHUTDOWN_ELEGANT: // 所有通话完成后再关机
2703     case SCSC_SHUTDOWN_ASAP: // 关闭, 越快越好
2745     case SCSC_PAUSE_CHECK: // 检查暂停状态
2748     case SCSC_PAUSE_INBOUND_CHECK: // 检查暂停呼入状态
2751     case SCSC_PAUSE_OUTBOUND_CHECK: // 检查暂停呼出状态
2754     case SCSC_READY_CHECK: // 检查是否 Ready
2757     case SCSC_SHUTDOWN_CHECK: // 检查是否正在关机
2760     case SCSC_SHUTDOWN: // 关闭
2777     case SCSC_CHECK_RUNNING: // 检查是否正在运行
2780     case SCSC_LOGLEVEL: // 日志级别
2790     case SCSC_DEBUG_LEVEL: // 调试级别
2798     case SCSC_MIN_IDLE_CPU: // 最小的 Idle CPU, 如果系统 Idel CPU 小于该值, 则不允许新通话
2807     case SCSC_MAX_SESSIONS: // 设置最大支持的并发 Session 数
2810     case SCSC_LAST_SPS: // 最后的 SPS
2813     case SCSC_SPS_PEAK: // 峰值 SPS
2819     case SCSC_SPS_PEAK_FIVEMIN: // 最后 5 分钟的峰值 SPS
2822     case SCSC_SESSIONS_PEAK: // 峰值并发数
2825     case SCSC_SESSIONS_PEAK_FIVEMIN: // 最后 5 分钟的峰值并发数
2828     case SCSC_MAX_DTMF_DURATION: // 最大 DTMF 时长
2831     case SCSC_MIN_DTMF_DURATION: // 最小 DTMF 时长
2834     case SCSC_DEFAULT_DTMF_DURATION: // 默认 DTMF 时长
2837     case SCSC_SPS: // 每秒创建的 Session 数
2846     case SCSC_RECLAIM: // 回收内存

```


返回核心的参数、状态等。

```

2860 SWITCH_DECLARE(switch_core_flag_t) switch_core_flags(void)
2865 SWITCH_DECLARE(switch_bool_t) switch_core_running(void)
2870 SWITCH_DECLARE(switch_bool_t) switch_core_ready(void)
2875 SWITCH_DECLARE(switch_bool_t) switch_core_ready_inbound(void)
2880 SWITCH_DECLARE(switch_bool_t) switch_core_ready_outbound(void)

```

L2885 ~ L2989 在 FreeSWITCH 关闭时释放相关内存，关闭 Socket、清理现场等。如 L2901 卸载所有模块、L2910 ~ L2911 释放 RTP 和 MSRP。

```

2885 SWITCH_DECLARE(switch_status_t) switch_core_destroy(void)
2886 {
2901     switch_loadable_module_shutdown();
2910     switch_rtp_shutdown();
2911     switch_msrp_destroy();
2977 }

```

管理接口执行相应动作。

```

2979 SWITCH_DECLARE(switch_status_t) switch_core_management_exec(char *relative_oid,
↪ switch_management_action_t action, char *data, switch_size_t datalen)

```

L2991 ~ L2996，回收内存。FreeSWITCH 大量使用内存池，这几个函数可以用于回收一些内存。它在上面讲的 L2888 被调用。

```

2991 SWITCH_DECLARE(void) switch_core_memory_reclaim_all(void)
2992 {
2993     switch_core_memory_reclaim_logger();
2994     switch_core_memory_reclaim_events();
2995     switch_core_memory_reclaim();
2996 }

```

L2999 定义了一个系统线程句柄结构。

```

2999 struct system_thread_handle {
3000     const char *cmd;

```

```
3001     switch_thread_cond_t *cond;
3002     switch_mutex_t *mutex;
3003     switch_memory_pool_t *pool;
3004     int ret;
3005     int *fds;
3006 };
```

L3008 是一个线程句柄的回调函数，它将会在一个线程中执行。句柄的参数将在 `obj` 参数中传入，并在 L3010 行被强制转换为 `struct system_thread_handle` 结构。它将执行 L3032 的 `system` 函数，执行一个操作系统上的命令，并把执行结果返回给 `sth->ret`。最终，

L3040 获取一个互斥锁，然后给 `sth->cond` 发一个信号（L3041），通知其它线程该函数执行完毕了。

```
3008 static void *SWITCH_THREAD_FUNC system_thread(switch_thread_t *thread, void *obj)
3009 {
3032     sth->ret = system(sth->cmd);
3040     switch_mutex_lock(sth->mutex);
3041     switch_thread_cond_signal(sth->cond);
3042     switch_mutex_unlock(sth->mutex);
```

下面是真正启动线程执行命令的函数。L3058 初始化一个内存池。L3063 则在内存池中申请一个 `struct system_thread_handle` 结构。下面则是对该结构的赋值（L3068~L3074）。注意 L3071 行初始化了一个 `cond`，它是一个条件变量。

```
3050 static int switch_system_thread(const char *cmd, switch_bool_t wait)
3051 {
3058     if (switch_core_new_memory_pool(&pool) != SWITCH_STATUS_SUCCESS) {
3061     }
3062
3063     if (!(sth = switch_core_alloc(pool, sizeof(struct system_thread_handle)))) {
3065     }
3066
3067     sth->pool = pool;
3068     sth->cmd = switch_core_strdup(pool, cmd);
3069
3070     switch_thread_cond_create(&sth->cond, sth->pool);
3071     switch_mutex_init(&sth->mutex, SWITCH_MUTEX_NESTED, sth->pool);
3072     switch_mutex_lock(sth->mutex);
3073
3074     switch_threadattr_create(&thd_attr, sth->pool);
```

一切准备好后，在 L3078 启动一个新线程，新线程将执行 `system_thread` (L3007) 回调函数，并将准备好的 `sth` 结构以参数形式传入。

```

3075     switch_threadattr_create(&thd_attr, sth->pool);
3076     switch_threadattr_stacksize_set(thd_attr, SWITCH_SYSTEM_THREAD_STACKSIZE);
3077     switch_threadattr_detach_set(thd_attr, 1);
3078     switch_thread_create(&thread, thd_attr, system_thread, sth, sth->pool);

```

然后，如果需要等待 (L3080)，它将一直等待 `sth->cond` (L3081)，直到回调函数执行完毕并给它发一个信号 (L3041)，表示 `system` 线程执行完了，然后便可以取到结果。

```

3080     if (wait) {
3081         switch_thread_cond_wait(sth->cond, sth->mutex);
3082         ret = sth->ret;
3083     }
3084     switch_mutex_unlock(sth->mutex);
3085
3086     return ret;

```

返回最大的文件描述符。

```

3089 SWITCH_DECLARE(int) switch_max_file_desc(void)

```

关闭不再使用的文件描述符。该函数在 L3241 调用，用于在 `fork` 环境中关闭不再使用的文件描述符。由于 UNIX 的 `fork` 调用会根据当前进程产生一个一模一样的新进程，原进程中打开的文件描述符在新进程中也会保持打开，在大多数情况下都是不需要继续打开的，因而可以用该函数关闭以避免冲突。

```

3105 SWITCH_DECLARE(void) switch_close_extra_files(int *keep, int keep_ttl)

```

在 `Windows` 中，不支持 `Fork`。所以只能在线程中执行。

```

3129 #ifdef WIN32
3130 static int switch_system_fork(const char *cmd, switch_bool_t wait)
3131 {
3132     return switch_system_thread(cmd, wait);
3133 }

```

在 *NIX 系统下，Fork 一个进程，并执行 `system` 函数。注意，`fork` 函数会将进程分裂为两个进程，该函数在父进程中返回子进程的 `pid`，而在子进程中则返回 `0`。接下来，两个进程会分别接着执行后面的语句。

```
3143 SWITCH_DECLARE(pid_t) switch_fork(void)
3144 {
3145     int i = fork();
3146
3147     if (!i) {
3148         set_low_priority();
3149     }
3150
3151     return i;
3152 }
```

通过 Fork 在新进程中执行 `system` 函数。

```
3156 static int switch_system_fork(const char *cmd, switch_bool_t wait)
3157 {
3158     pid = switch_fork();
3159
3160     if (pid) { // 父进程
3161         if (wait) { // 等待子进程结束
3162             waitpid(pid, NULL, 0);
3163         }
3164         free(dcmd);
3165     } else { // 子进程，执行 system
3166         switch_close_extra_files(NULL, 0);
3167
3168         if (system(dcmd) == -1) {
3169             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR,
3170                             "Failed to execute because of a command error : %s\n", dcmd);
3171         }
3172         free(dcmd);
3173         exit(0); // 返回后，父进程结束等待，
3174     }
3175 }
```

执行 `system` 函数，根据不同场景和设置决定调用 `fork` 还是在新线程中执行。

```
3203 SWITCH_DECLARE(int) switch_system(const char *cmd, switch_bool_t wait)
```

执行 `system` 函数，传入一个 `switch_stream_handle_t` 结构，可以将结果写到 `stera` 中。

```

3215 SWITCH_DECLARE(int) switch_stream_system_fork(const char *cmd, switch_stream_handle_t *stream)
...
3282 SWITCH_DECLARE(int) switch_stream_system(const char *cmd, switch_stream_handle_t *stream)

```

获取堆栈大小。

```

3258 SWITCH_DECLARE(switch_status_t) switch_core_get_stack_sizes(switch_size_t *cur, switch_size_t *max)

```

获取 RTP 端口范围。

```

3293 SWITCH_DECLARE(uint16_t) switch_core_get_rtp_port_range_start_port()
3304 SWITCH_DECLARE(uint16_t) switch_core_get_rtp_port_range_end_port()

```

4.11 switch_core_asr.c

本章基于 Commit Hash 1681db4。

本文件实现了自动语音识别功能相关的接口函数。

L40 定义了打开 ASR 的函数，传入的参数有模块名称（`module_name`）及音频编码（`codec`）等。其中，`module_name` 可能会包含冒号（L48），冒号后面是模块相关的参数（L50 ~ L51）。

```

40 SWITCH_DECLARE(switch_status_t) switch_core_asr_open(switch_asr_handle_t *ah,
41     const char *module_name,
42     const char *codec, int rate, const char *dest,
43     switch_asr_flag_t *flags, switch_memory_pool_t *pool)
44 {
...
48     if (strchr(module_name, ':')) {
49         switch_set_string(buf, module_name);
50         if ((param = strchr(buf, ':')) {
51             *param++ = '\0';
52             module_name = buf;
53         }
54     }

```

根据 `module_name` 找到相应的模块并打开，如 `module_name = unimrcp` 时，则打开 `mod_unimrcp` 里的 `asr_open` 接口函数。

```

58     if ((ah->asr_interface = switch_loadable_module_get_asr_interface(module_name)) == 0) {
59         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Invalid ASR module [%s]!\n",
↪ module_name);
60         return SWITCH_STATUS_GENERR;
61     }
...
83     status = ah->asr_interface->asr_open(ah, codec, rate, dest, flags);
84
...
91 }

```

加载/卸载相应的语法文件，有些模块（如 `mod_pocketsphinx` 和 `mod_unimrcp`）需要语法文件辅助识别。

```

93 SWITCH_DECLARE(switch_status_t) switch_core_asr_load_grammar(switch_asr_handle_t *ah, const char
↪ *grammar, const char *name)
...
156 SWITCH_DECLARE(switch_status_t) switch_core_asr_unload_grammar(switch_asr_handle_t *ah, const char
↪ *name)

```

启动/停用相关的语法。

```

166 SWITCH_DECLARE(switch_status_t) switch_core_asr_enable_grammar(switch_asr_handle_t *ah, const char
↪ *name)
...
179 SWITCH_DECLARE(switch_status_t) switch_core_asr_disable_grammar(switch_asr_handle_t *ah, const char
↪ *name)
...
192 SWITCH_DECLARE(switch_status_t) switch_core_asr_disable_all_grammars(switch_asr_handle_t *ah)

```

暂停 ASR 解析。

```

205 SWITCH_DECLARE(switch_status_t) switch_core_asr_pause(switch_asr_handle_t *ah)
206 {
...
209     return ah->asr_interface->asr_pause(ah);
210 }

```

继续 ASR 解析。

```
212 SWITCH_DECLARE(switch_status_t) switch_core_asr_resume(switch_asr_handle_t *ah)
213 {
...
216     return ah->asr_interface->asr_resume(ah);
217 }
```

关闭。

```
219 SWITCH_DECLARE(switch_status_t) switch_core_asr_close(switch_asr_handle_t *ah, switch_asr_flag_t *flags)
```

向 ASR 引擎“喂”数据，在 FreeSWITCH 中收到的音频数据时，调用该函数。数据的类型为 `int16`，即每个数据以 16 位整数（2 个字节）来表示，`len` 的单位为实际的字节数。

```
239 SWITCH_DECLARE(switch_status_t) switch_core_asr_feed(switch_asr_handle_t *ah, void *data, unsigned int
↪ len, switch_asr_flag_t *flags)
240 {
```

如果当前 Channel 的采样率与 ASR 模块的不匹配，则自动创建转换器并转换采样率。一般来说，PSTN 通话为 `8000Hz`，而 WebRTC 相关的通信则默认为 `48000Hz`，一般来说 `16000Hz` 的采样率会有较好的识别结果，但是为了适配 PSTN 的通话质量，一般的 ASR 引擎都会专门提供 `8000Hz` 的窄带识别模型。

```
244     if (ah->native_rate && ah->samplerate && ah->native_rate != ah->samplerate) {
245         if (!ah->resampler) {
246             if (switch_resample_create(&ah->resampler,
247                 ah->samplerate, ah->native_rate, (uint32_t) orig_len, SWITCH_RESAMPLE_QUALITY, 1) !=
↪ SWITCH_STATUS_SUCCESS) {
248                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Unable to create resampler!\n");
249                 return SWITCH_STATUS_GENERR;
250             }
251         }
```

执行采样率转换。

```
253     switch_resample_process(ah->resampler, data, len / 2);
```

最后“喂”给实际的引擎进行识别。

```
272     return ah->asr_interface->asr_feed(ah, data, len, flags);
```

FreeSWITCH 自己能识别 DTMF，但是为了完整，有时也需要将 DTMF 发送给 ASR 引擎。

```
275 SWITCH_DECLARE(switch_status_t) switch_core_asr_feed_dtmf(switch_asr_handle_t *ah, const switch_dtmf_t
↳ *dtmf, switch_asr_flag_t *flags)
```

检查是否有相应的识别结果。一般来说，ASR 引擎缺少通知机制，需要核心不断调用该函数以检测是否有相应的识别结果。

```
288 SWITCH_DECLARE(switch_status_t) switch_core_asr_check_results(switch_asr_handle_t *ah, switch_asr_flag_t
↳ *flags)
289 {
...
292     return ah->asr_interface->asr_check_results(ah, flags);
293 }
```

如果上一个函数返回了 **SUCCESS**，表示有相应的识别结果，则该函数用于取回识别结果。由于历史原因，ASR 引擎都会返回 XML，因此返回是 XML 字符串（**xmlstr**）。但新的引擎有的会返回 JSON 字符串数据。一般来说，该字符串内是在堆上申请的，应该在调用者处销毁。

```
295 SWITCH_DECLARE(switch_status_t) switch_core_asr_get_results(switch_asr_handle_t *ah, char **xmlstr,
↳ switch_asr_flag_t *flags)
296 {
...
299     return ah->asr_interface->asr_get_results(ah, xmlstr, flags);
300 }
```

获取额外的识别结果头域。

```
302 SWITCH_DECLARE(switch_status_t) switch_core_asr_get_result_headers(switch_asr_handle_t *ah,
↳ switch_event_t **headers, switch_asr_flag_t *flags)
```

开启 ASR 引擎内部的计数器，以计算相应的超时等。

```

314 SWITCH_DECLARE(switch_status_t) switch_core_asr_start_input_timers(switch_asr_handle_t *ah)
315 {
...
320     if (ah->asr_interface->asr_start_input_timers) {
321         status = ah->asr_interface->asr_start_input_timers(ah);
322     }
...
325 }

```

向 ASR 引擎传递相应的参数。

```

327 SWITCH_DECLARE(void) switch_core_asr_text_param(switch_asr_handle_t *ah, char *param, const char *val)
336 SWITCH_DECLARE(void) switch_core_asr_numeric_param(switch_asr_handle_t *ah, char *param, int val)
345 SWITCH_DECLARE(void) switch_core_asr_float_param(switch_asr_handle_t *ah, char *param, double val)

```

4.12 switch_core_cert.c

本章基于 Commit Hash [1681db4](#)。

证书相关，FreeSWITCH 使用了 OpenSSL 支持证书。读这段代码需要有 OpenSSL 以及加密解密相关的知识。

在多线程环境中，需要处理有相关的锁支持。

```

39 static inline void switch_ssl_ssl_lock_callback(int mode, int type, char *file, int line)
49 static inline unsigned long switch_ssl_ssl_thread_id(void)
54 SWITCH_DECLARE(void) switch_ssl_init_ssl_locks(void)
79 SWITCH_DECLARE(void) switch_ssl_destroy_ssl_locks(void)

```

根据不同的加密算法字符串获取不同的加密函数。

```

96 static const EVP_MD *get_evp_by_name(const char *name)
97 {
98     if (!strcasecmp(name, "md5")) return EVP_md5();
99     if (!strcasecmp(name, "sha1")) return EVP_sha1();
100    if (!strcasecmp(name, "sha-1")) return EVP_sha1();

```

```
101     if (!strcasecmp(name, "sha-256")) return EVP_sha256();  
102     if (!strcasecmp(name, "sha-512")) return EVP_sha512();
```

检查证书指纹。

```
133 SWITCH_DECLARE(int) switch_core_cert_verify(dtls_fingerprint_t *fp)
```

扩展指纹。

```
152 SWITCH_DECLARE(int) switch_core_cert_expand_fingerprint(dtls_fingerprint_t *fp, const char *str)
```

将指纹字符串扩展内存中的格式。

```
168 SWITCH_DECLARE(int) switch_core_cert_extract_fingerprint(X509 *x509, dtls_fingerprint_t *fp)
```

从 X509 证书中提取指纹。

```
189 SWITCH_DECLARE(int) switch_core_cert_gen_fingerprint(const char *prefix, dtls_fingerprint_t *fp)
```

从文件中提取指纹。

```
SWITCH_DECLARE(int) switch_core_cert_gen_fingerprint(const char *prefix, dtls_fingerprint_t *fp)
```

生成证书文件。产生 .pem 或 .key 和 .crt 文件。

```
238 static int mkcert(X509 **x509p, EVP_PKEY **pkeyp, int bits, int serial, int days);  
239  
240 SWITCH_DECLARE(int) switch_core_gen_certs(const char *prefix)
```

4.13 switch_core_codec.c

本章基于 Commit Hash 1681db4。

编解码相关。

L39 ~ L44, 为每个 Codec 都生成一个 ID。

```
39 static uint32_t CODEC_ID = 1;
40
41 SWITCH_DECLARE(uint32_t) switch_core_codec_next_id(void)
42 {
43     return CODEC_ID++;
44 }
```

从当前 Session 时取消掉之前设置的读 codec 及写 codec。读 codec 用于从远端收到数据时解码，写 codec 用于将 FreeSWITCH 内部数据编码并发送到远端。

```
46 SWITCH_DECLARE(void) switch_core_session_unset_read_codec(switch_core_session_t *session)
82 SWITCH_DECLARE(void) switch_core_session_unset_write_codec(switch_core_session_t *session)
```

多线程环境中加读写锁。

```
62 SWITCH_DECLARE(void) switch_core_session_lock_codec_write(switch_core_session_t *session)
67 SWITCH_DECLARE(void) switch_core_session_unlock_codec_write(switch_core_session_t *session)
72 SWITCH_DECLARE(void) switch_core_session_lock_codec_read(switch_core_session_t *session)
77 SWITCH_DECLARE(void) switch_core_session_unlock_codec_read(switch_core_session_t *session)
```

设置 real_read_codec。

```
94 SWITCH_DECLARE(switch_status_t) switch_core_session_set_real_read_codec(
```

设置 read_codec。

```
197 SWITCH_DECLARE(switch_status_t) switch_core_session_set_read_codec(switch_core_session_t *session,
↪ switch_codec_t *codec)
```

获取当前 Session 上有效的读 `codec`。

```
305 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_effective_read_codec(switch_core_session_t  
↪ *session)
```

获取当前 Session 上的读 `codec`。

```
312 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_read_codec(switch_core_session_t *session)
```

获取本 `codec` 相应的实现信息，如采样率等。同样的 `codec` 可能会有不同的实现，如不同的采样率，不同的打包间隔等。

```
319 SWITCH_DECLARE(switch_status_t) switch_core_session_get_read_impl(switch_core_session_t *session,  
↪ switch_codec_implementation_t *impp)
```

获取 `read_read_codec` 的实现。

```
331 SWITCH_DECLARE(switch_status_t) switch_core_session_get_real_read_impl(switch_core_session_t *session,  
↪ switch_codec_implementation_t *impp)
```

写。

```
341 SWITCH_DECLARE(switch_status_t) switch_core_session_get_write_impl(switch_core_session_t *session,  
↪ switch_codec_implementation_t *impp)
```

视频。

```
353 SWITCH_DECLARE(switch_status_t) switch_core_session_get_video_read_impl(switch_core_session_t *session,  
↪ switch_codec_implementation_t *impp)  
365 SWITCH_DECLARE(switch_status_t) switch_core_session_get_video_write_impl(switch_core_session_t *session,  
↪ switch_codec_implementation_t *impp)
```

设置相关实现。

```

378 SWITCH_DECLARE(switch_status_t) switch_core_session_set_read_impl(switch_core_session_t *session, const
↳ switch_codec_implementation_t *impp)
384 SWITCH_DECLARE(switch_status_t) switch_core_session_set_write_impl(switch_core_session_t *session, const
↳ switch_codec_implementation_t *impp)
390 SWITCH_DECLARE(switch_status_t) switch_core_session_set_video_read_impl(switch_core_session_t *session,
↳ const switch_codec_implementation_t *impp)
396 SWITCH_DECLARE(switch_status_t) switch_core_session_set_video_write_impl(switch_core_session_t *session,
↳ const switch_codec_implementation_t *impp)

```

以下函数意思非常明了。

```

403 SWITCH_DECLARE(switch_status_t) switch_core_session_set_write_codec(switch_core_session_t *session,
↳ switch_codec_t *codec)
468 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_effective_write_codec(switch_core_session_t
↳ *session)
476 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_write_codec(switch_core_session_t *session)
486 SWITCH_DECLARE(switch_status_t) switch_core_session_set_video_read_codec(switch_core_session_t *session,
↳ switch_codec_t *codec)
526 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_video_read_codec(switch_core_session_t
↳ *session)
535 SWITCH_DECLARE(switch_status_t) switch_core_session_set_video_write_codec(switch_core_session_t
↳ *session, switch_codec_t *codec)
571 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_video_write_codec(switch_core_session_t
↳ *session)

```

解析 `fntp`，`fntp`一般可以在 SDP 里描述，可以为 Codec 设置不同的参数。

L591，获取 Codec 接口，该函数同时会获得读锁，所以使用完毕后需要释放相关的锁（L597）。实际在解析函数在 Codec 模块中实现（L594）。

```

580 SWITCH_DECLARE(switch_status_t) switch_core_codec_parse_fntp(const char *codec_name, const char *fntp,
↳ uint32_t rate, switch_codec_fntp_t *codec_fntp)
581 {
591     if ((codec_interface = switch_loadable_module_get_codec_interface(codec_name, NULL))) {
592         if (codec_interface->parse_fntp) {
593             codec_fntp->actual_samples_per_second = rate;
594             status = codec_interface->parse_fntp(fntp, codec_fntp);
595         }
596
597         UNPROTECT_INTERFACE(codec_interface);
598     }
600     return status;
601 }

```

重置 Codec。

```
603 SWITCH_DECLARE(switch_status_t) switch_core_codec_reset(switch_codec_t *codec)
```

由现有 Codec 复制生成一个新的 Codec。

```
614 SWITCH_DECLARE(switch_status_t) switch_core_codec_copy(switch_codec_t *codec,
```

初始化一个 Codec。其中：

- `codec_name` 为 Codec 的名字，如 PCMU。
- `mod_name` 为使用哪个模块的实现，有些 Codec 在多个模块中都有实现，如 H264 在 `mod_av` 和 `mod_openh264` 中都有实现。如果该值为 `NULL`，则由 FreeSWITCH 选择使用哪一个。
- `fntp` 为 Codec 的参数，可以为 `NULL`。
- `rate` 是 Codec 的采样率，如 8000。
- `ms` 是打包时间，如 20 ms。
- `channels` 是 Codec 支持的通道数，通常为 1，立体声为 2。
- `bitrate` 是速率，一些编码如 PCMU 速率是恒定的，即 64kbps，而有些编码如 OPUS 速率就是变化的，该参数指定最大速率。
- `flags`，一些标志，如支持编码还是解码，还是两者都支持。
- `codec_settings` 为一些 Codec 设置，视频编码通常有更多的设置。

```
633 SWITCH_DECLARE(switch_status_t) switch_core_codec_init_with_bitrate(switch_codec_t *codec, const char
↳ *codec_name, const char *modname, const char *fntp,
634     uint32_t rate, int ms, int channels, uint32_t bitrate, uint32_t flags,
635     const switch_codec_settings_t *codec_settings, switch_memory_pool_t *pool)
```

如果 `codec_name` 是以 “.” 分隔的（L649 ~ L656），则前端为模块名，后面为实际的 Codec 名称，如 `mod_av.H264`。

```
649     if (strchr(codec_name, '.')) {
650         char *p = NULL;
651         codec_name = switch_core_strdup(pool, codec_name);
652         if ((p = strchr(codec_name, '.'))) {
653             *p++ = '\0';
```

```

654         modname = codec_name;
655         codec_name = p;
656     }
657 }
```

L659, 查找哪个模块实现了该编码器。

```

659     if ((codec_interface = switch_loadable_module_get_codec_interface(codec_name, modname)) == 0) {
660         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Invalid codec %s!\n", codec_name);
661         return SWITCH_STATUS_GENERR;
662     }
```

G.722 编码标准实际上有个 Bug, 在 SDP 里, 采样率写的是 8000, 但实际上应该发送 16000, L678 ~ L682 对其进行特殊处理, 使用 `samples_per_second` 而不是 `actual_samples_per_second`。所以需要一些特殊处理。

```

678         uint32_t crate = !strcasecmp(codec_name, "g722") ? iptr->samples_per_second :
679         uint32_t crate = !strcasecmp(codec_name, "g722") ? iptr->samples_per_second : iptr-
↳ >actual_samples_per_second;
...
697     found: // 如果找到了相应的实现
698
699     if (implementation) { // 则调用模块的`init`接口进行初始化
718         implementation->init(codec, flags, codec_settings);
```

编码接口, 其中 `other_codec` 未使用, 输入数据为未编码的数据 `decoded_data`, 16 位 PCM 格式, 长度为 `decoded_data_len1`; 输出数据为编码后的数据 `encoded_data`, 长度为 `encodec_data_len`。注意长度的单位为 Sample (数据样本) 的数量, 由于每个 Sample 为 16 位 PCM 数据, 占两个字节, 所以输入数据占用缓冲区实际的长度字节数应该为 `decodec_data_len * 2`。输出数据的长度为字节数。

```

733 SWITCH_DECLARE(switch_status_t) switch_core_codec_encode(switch_codec_t *codec,
734                                                           switch_codec_t *other_codec,
735                                                           void *decoded_data,
736                                                           uint32_t decoded_data_len,
737                                                           uint32_t decoded_rate,
738                                                           void *encoded_data, uint32_t *encoded_data_len,
↳ uint32_t *encoded_rate, unsigned int *flag)
739 {
    // 实际调用相关编码模块里的函数进行 encode
```

```

757     status = codec->implementation->encode(codec, other_codec, decoded_data, decoded_data_len,
758         decoded_rate, encoded_data, encoded_data_len, encoded_rate, flag);
760
761     return status;
763 }

```

解码原理差不多，解码后的数据放到 `decoded_data` 中，长度为 `decoded_data_len`。

```

765 SWITCH_DECLARE(switch_status_t) switch_core_codec_decode(switch_codec_t *codec,
766     switch_codec_t *other_codec,
767     void *encoded_data,
768     uint32_t encoded_data_len,
769     uint32_t encoded_rate,
770     void *decoded_data, uint32_t *decoded_data_len,
771     ↪ uint32_t *decoded_rate, unsigned int *flag)
772 {
773     status = codec->implementation->decode(codec, other_codec, encoded_data, encoded_data_len,
774     ↪ encoded_rate,
775     decoded_data, decoded_data_len, decoded_rate, flag);

```

L808，视频编码。调用相关实现中的 `encode_video` 函数对图像进行编码。输入参数为一帧图像，存放在 `frame->img` 中。通常，一帧图像编码会产生比较长的数据，如果这些数据超过一定大小（如网络的 MTU，最大 1500）值，则可能会在网络层造成分片，并可能产生数据丢失，错乱的情况。因此，为了避免这种情况，一般会将生成的数据切片。不同的编码有不同的切片方案，在 FreeSWITCH 中，默认切片的大小为最大 1200 字节。

如果编码过程中发生了数据切片，则 `encode_video` 函数会返回 `SWITCH_STATUS_MORE_DATA` (L829)，表明编码器中还会有更多的数据。此时，设置 `SFF_SAME_IMAGE` 标志，上层的应用应该继续使用同一帧图像调用 `switch_core_codec_encode_video` 函数，以便取出后续更多的切片数据。

获取数据后，将 `packetlen` 设为数据长度+12 (L835)，以便为 RTP 头留出空间。

```

808 SWITCH_DECLARE(switch_status_t) switch_core_codec_encode_video(switch_codec_t *codec, switch_frame_t
809     ↪ *frame)
810 {
811     if (codec->implementation->encode_video) {
812         status = codec->implementation->encode_video(codec, frame);
813
814         if (status == SWITCH_STATUS_MORE_DATA) {
815             frame->flags |= SFF_SAME_IMAGE;
816         } else {
817             frame->flags &= ~SFF_SAME_IMAGE;
818         }
819     }

```



```

834
835     frame->packetlen = frame->datalen + 12;
836 }
837
838 if (codec->mutex) switch_mutex_unlock(codec->mutex);
839
840 return status;
842 }

```

L844, 视频解码。对收到一个数据帧（`frame` 是一个 RTP 数据帧），调用 `decode_video` (L864) 进行解码。通常，由于数据分片，解码器会先将 RTP 缓存，等到收到足够的 RTP 数据后才进行解码。所以，并不是每次解码都能得到图像。如果解码器需要更多的数据才能解码，解码器会返回 `status = SWITCH_STATUS_MORE_DATA`。

解码成功后，得到的图像存放到 `frame->img` 中。不管是编码还是解码，`frame->img` 中的图像格式都是 I420 格式的。

```

844 SWITCH_DECLARE(switch_status_t) switch_core_codec_decode_video(switch_codec_t *codec, switch_frame_t
↪ *frame)
845 {
846
863     if (codec->implementation->decode_video) {
864         status = codec->implementation->decode_video(codec, frame);
865     }
869 }

```

L872, 对 Codec 进行控制，如，在编码过程中强制编码器产生一个关键帧，或者根据需要改变带宽等。

```

872 SWITCH_DECLARE(switch_status_t) switch_core_codec_control(switch_codec_t *codec,
873                                                            switch_codec_control_command_t cmd,
874                                                            switch_codec_control_type_t ctype,
875                                                            void *cmd_data,
876                                                            switch_codec_control_type_t atype,
877                                                            void *cmd_arg,
878                                                            switch_codec_control_type_t *rtype,
879                                                            void **ret_data)
880 {
897     if (codec->implementation->codec_control) {
898         status = codec->implementation->codec_control(codec, cmd, ctype, cmd_data, atype, cmd_arg,
↪ rtype, ret_data);
899     }
905 }

```

销毁编码器。

```
907 SWITCH_DECLARE(switch_status_t) switch_core_codec_destroy(switch_codec_t *codec)
```

4.14 switch_core_db.c

本章基于 Commit Hash [1681db4](#)。

核心数据库接口。核心数据库默认使用 SQLite。

选择路径，打开、关闭数据库。

```
38 static void db_pick_path(const char *dbname, char *buf, switch_size_t size)
48 SWITCH_DECLARE(int) switch_core_db_open(const char *filename, switch_core_db_t **ppDb)
53 SWITCH_DECLARE(int) switch_core_db_close(switch_core_db_t *db)
```

返回列的值为字符串。

```
58 SWITCH_DECLARE(const unsigned char *) switch_core_db_column_text(switch_core_db_stmt_t *stmt, int iCol)
```

返回列名。

```
71 SWITCH_DECLARE(const char *) switch_core_db_column_name(switch_core_db_stmt_t *stmt, int N)
```

返回列数。

```
76 SWITCH_DECLARE(int) switch_core_db_column_count(switch_core_db_stmt_t *pStmt)
```

返回错误信息。

```
81 SWITCH_DECLARE(const char *) switch_core_db_errmsg(switch_core_db_t *db)
```

执行 `sql` 语句，对返回的每一行，执行 `callback` 回调函数，`data` 可以为回调函数设置参数。查询时如果遇到数据库忙等情况会多次重试（L92）。

```

86 SWITCH_DECLARE(int) switch_core_db_exec(switch_core_db_t *db, const char *sql,
↪ switch_core_db_callback_func_t callback, void *data, char **errmsg)
87 {
...
92 while (--sane > 0) {
93     ret = sqlite3_exec(db, sql, callback, data, &err);
94     if (ret == SQLITE_BUSY || ret == SQLITE_LOCKED) {
95         if (sane > 1) {
96             switch_core_db_free(err);
97             switch_yield(100000);
98             continue;
99         }
100     } else {
101         break;
102     }
103 }

```

关闭结果数据集。

```

116 SWITCH_DECLARE(int) switch_core_db_finalize(switch_core_db_stmt_t *pStmt)

```

以下函数基本上是对 SQLite 原始函数的封装，不多解释。

```

121 SWITCH_DECLARE(int) switch_core_db_prepare(switch_core_db_t *db, const char *zSql, int nBytes,
↪ switch_core_db_stmt_t **ppStmt, const char **pzTail)
126 SWITCH_DECLARE(int) switch_core_db_step(switch_core_db_stmt_t *pStmt)
131 SWITCH_DECLARE(int) switch_core_db_reset(switch_core_db_stmt_t *pStmt)
136 SWITCH_DECLARE(int) switch_core_db_bind_int(switch_core_db_stmt_t *pStmt, int i, int iValue)
141 SWITCH_DECLARE(int) switch_core_db_bind_int64(switch_core_db_stmt_t *pStmt, int i, int64_t iValue)
146 SWITCH_DECLARE(int) switch_core_db_bind_text(switch_core_db_stmt_t *pStmt, int i, const char *zData, int
↪ nData, switch_core_db_destructor_type_t xDel)
151 SWITCH_DECLARE(int) switch_core_db_bind_double(switch_core_db_stmt_t *pStmt, int i, double dValue)
156 SWITCH_DECLARE(int64_t) switch_core_db_last_insert_rowid(switch_core_db_t *db)
161 SWITCH_DECLARE(int) switch_core_db_get_table(switch_core_db_t *db, const char *sql, char ***resultp, int
↪ *nrow, int *ncolumn, char **errmsg)
166 SWITCH_DECLARE(void) switch_core_db_free_table(char **result)
171 SWITCH_DECLARE(void) switch_core_db_free(char *z)
176 SWITCH_DECLARE(int) switch_core_db_changes(switch_core_db_t *db)
181 SWITCH_DECLARE(int) switch_core_db_load_extension(switch_core_db_t *db, const char *extension)

```

```

198 SWITCH_DECLARE(switch_core_db_t *) switch_core_db_open_file(const char *filename)
230 SWITCH_DECLARE(void) switch_core_db_test_reactive(switch_core_db_t *db, char *test_sql, char *drop_sql,
↳ char *reactive_sql)
273 SWITCH_DECLARE(switch_status_t) switch_core_db_persistent_execute_trans(switch_core_db_t *db, char *sql,
↳ uint32_t retries)
344 SWITCH_DECLARE(switch_status_t) switch_core_db_persistent_execute(switch_core_db_t *db, char *sql,
↳ uint32_t retries)

```

4.15 switch_core_directory.c

本章基于 Commit Hash [1681db4](#)。

企业目录服务相关，与目录服务器相连，如 LDAP 等。目前实际应用较少。

```

38 SWITCH_DECLARE(switch_status_t) switch_core_directory_open(switch_directory_handle_t *dh,
61 SWITCH_DECLARE(switch_status_t) switch_core_directory_query(switch_directory_handle_t *dh, char *base,
↳ char *query)
66 SWITCH_DECLARE(switch_status_t) switch_core_directory_next(switch_directory_handle_t *dh)
71 SWITCH_DECLARE(switch_status_t) switch_core_directory_next_pair(switch_directory_handle_t *dh, char
↳ **var, char **val)
76 SWITCH_DECLARE(switch_status_t) switch_core_directory_close(switch_directory_handle_t *dh)

```

4.16 switch_core_event_hook.c

本章基于 Commit Hash [1681db4](#)。

该文件比较简单，只是定义了一些 Hook 函数。

```

34 NEW_HOOK_DECL(outgoing_channel)
35 NEW_HOOK_DECL(receive_message)
36 NEW_HOOK_DECL(receive_event)
37 NEW_HOOK_DECL(state_change)
38 NEW_HOOK_DECL(state_run)
39 NEW_HOOK_DECL(read_frame)
40 NEW_HOOK_DECL(write_frame)
41 NEW_HOOK_DECL(video_read_frame)
42 NEW_HOOK_DECL(video_write_frame)
43 NEW_HOOK_DECL(text_read_frame)
44 NEW_HOOK_DECL(text_write_frame)
45 NEW_HOOK_DECL(kill_channel)

```

```

46 NEW_HOOK_DECL(send_dtmf)
47 NEW_HOOK_DECL(recv_dtmf)

```

其中，`NEW_HOOK_DECL` 是一个宏定义，在 `switch_event_hook.h` 中定义，主要的定义是下面两行：

```

switch_io_event_hook_##_NAME##_t *hook, *ptr;
session->event_hooks._NAME = hook;

```

上面的定义扩展开就是：

```

switch_io_event_hook_outgoing_channel_t *hook, *ptr;
session->event_hooks.outgoing_channel = hook;

```

所以，如果在看代码时全文搜索相关的定义时，可能搜不到，记得来这个文件里找。

Event Hook 是 FreeSWITCH 内部的一种回调机制，在通话的不同阶段执行不同的回调，如 `read_frame` 回调会在收到一帧 RTP 数据时执行。

把 `read_frame` 函数宏扩展以后的代码如下：

```

switch_status_t switch_core_event_hook_add_read_frame (switch_core_session_t *session,
↳ switch_read_frame_hook_t read_frame)
{
    switch_io_event_hook_read_frame_t *hook, *ptr;

    for (ptr = session->event_hooks.read_frame; ptr && ptr->next; ptr = ptr->next) {
        if (ptr->read_frame == read_frame) return SWITCH_STATUS_FALSE;
    }

    if (ptr && ptr->read_frame == read_frame) return SWITCH_STATUS_FALSE;
    if ((hook = switch_core_perform_session_alloc(session, sizeof(*hook),
        "src/switch_core_event_hook.c", (const char *)__func__, 39)) != 0) {
        hook->read_frame = read_frame ;
        if (! session->event_hooks.read_frame ) {
            session->event_hooks.read_frame = hook;
        } else {
            ptr->next = hook;
        }
        return SWITCH_STATUS_SUCCESS;
    }
}

```

```
    return SWITCH_STATUS_MEMERR;
}
```

当 `switch_core_event_hook_add_read_frame` 函数被调用时，向当前 Session 上面加一个 `event_hook` 回调函数。

在 `switch_core_io.c` 中，读到 RTP 数据时，会检查 Session 上所有的 `event_hooks`，并依次执行相关的回调函数（L177）。

```
176  for (ptr = session->event_hooks.read_frame; ptr; ptr = ptr->next) {
177      if ((status = ptr->read_frame(session, frame, flags, stream_id)) != SWITCH_STATUS_SUCCESS) {
178          break;
179      }
180  }
```

4.17 switch_core_file.c

本章基于 Commit Hash `1681db4`。

本文件定义了 FreeSWITCH 内部支持的各种音、视频文件的格式，以及读写函数。

获取文件大小：

```
40 static switch_status_t get_file_size(switch_file_handle_t *fh, const char **string)
```

打开文件。实际使用时一般使用 `switch_core_file_open` 这个宏定义，`_perform_` 版的函数传入的参数有源代码的文件名（`file`）、函数（`func`）、行号（`line`）等，便于在 Log 中找到实际调用的位置（下同）。

文件正确打开后，返回一个 File Handle（`*fh`）指针。如果实际文件的音轨（`channels`）数或采样频率（`rate`）与打开时参数要求的不一致（如打开一个 `44100Hz` 的 MP3 文件，但当前的 Channel 是 `8000Hz` 的 PSTN 呼叫），则会自动转换为需要的格式（当然动态转换会带来一些额外的开销）。

如果文件名参数中有“`{}`”，则尝试解析其中的参数。类似 Codec，文件接口也可能在多个模块中实现，因此，可以指定使用哪个具体模块的实现（L152）。其它可以指定的参数如（L124 ~ L176）`samplerate`（采样率）、`channels`（声道数）、`ab`、`vb`（音频及视频速率）、`try_hardware_encoder`（尝试使用硬件解码等）。

```

63 SWITCH_DECLARE(switch_status_t) switch_core_perform_file_open(const char *file, const char *func, int
↳ line,
64     switch_file_handle_t *fh,
65     const char *file_path,
66     uint32_t channels, uint32_t rate, unsigned int flags, switch_memory_pool_t *pool)
67 {
...
125     if (*file_path == '{') {
...
152         if ((modname = switch_event_get_header(fh->params, "modname"))) {
153             fh->modname = switch_core_strdup(fh->memory_pool, modname);
154         }
155
156         if ((val = switch_event_get_header(fh->params, "samplerate"))) {
163         if ((val = switch_event_get_header(fh->params, "force_channels"))) {
170         if ((val = switch_event_get_header(fh->params, "ab"))) {
177         if ((val = switch_event_get_header(fh->params, "cbr"))) {
182         if ((val = switch_event_get_header(fh->params, "vb"))) {
194         if ((val = switch_event_get_header(fh->params, "vw"))) {
201         if ((val = switch_event_get_header(fh->params, "vh"))) {
208         if ((val = switch_event_get_header(fh->params, "try_hardware_encoder"))) {
212         if ((val = switch_event_get_header(fh->params, "auth_username"))) {
216         if ((val = switch_event_get_header(fh->params, "auth_password"))) {
220         if ((val = switch_event_get_header(fh->params, "fps"))) {
227         if ((val = switch_event_get_header(fh->params, "vbuf"))) {
243         if ((val = switch_event_get_header(fh->params, "vencspd"))) {
255         if ((val = switch_event_get_header(fh->params, "vprofile"))) {

```

L303, 查找文件接口的具体实现, 该函数也会默认加锁, 因此在 L305 要释放锁。

找到具体实现后, 在 L354 调用模块中的 `file_open` 函数打开文件。

如果打开文件时指定了 `pre_buffer_datalen`, 则初始化一个 `pre_buffer`, 该 Buffer 用于预读一定长度的数据。

```

303     if ((fh->file_interface = switch_loadable_module_get_file_interface(ext, fh->modname)) == 0) {
305         switch_goto_status(SWITCH_STATUS_GENERR, fail);
306     }
...
354     if ((status = fh->file_interface->file_open(fh, file_path)) != SWITCH_STATUS_SUCCESS) {
355         if (fh->spool_path) {
356             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Spool dir is set. Make sure [%s]
↳ is also a valid path\n", fh->spool_path);
357         }
358         UNPROTECT_INTERFACE(fh->file_interface);
359         goto fail;

```

```

360     }
...
399     if (fh->pre_buffer_data_len) {
401         switch_buffer_create_dynamic(&fh->pre_buffer, fh->pre_buffer_data_len * fh->channels, fh-
↪ >pre_buffer_data_len * fh->channels, 0);
402         fh->pre_buffer_data = switch_core_alloc(fh->memory_pool, fh->pre_buffer_data_len * fh->channels);
403     }

```

L431, 从文件中读取音频。读到的都是 `int16` (`short`) 格式的, `len` 也是以 Sample 为单位的, 实际返回的字节数为 `Sample * channels * 2`。

L396, 首先尝试从缓存中读取数据。

```

431 SWITCH_DECLARE(switch_status_t) switch_core_file_read(switch_file_handle_t *fh, void *data,
↪ switch_size_t *len)
432 {
...
450     if (fh->buffer && switch_buffer_inuse(fh->buffer) >= *len * 2 * fh->channels) {
451         *len = switch_buffer_read(fh->buffer, data, orig_len * 2 * fh->channels) / 2 / fh->channels;
452         return *len == 0 ? SWITCH_STATUS_FALSE : SWITCH_STATUS_SUCCESS;
453     }

```

L466, 如果使用了文件预读功能, 则会先读取一定长度的数据到 `pre_buffer` 中 (L474, L486)。L468, `asis` 代表该文件是否是一个 Native File。然后, 再从 `pre_buffer` 中读取 (491)。

FreeSWITCH 支持 Native File。比如, 由于专利原因, FreeSWITCH 默认不支持 G729 转码。但可以直接从 G729 编码的文件中读取数据, 直接通过 RTP 发送出去, 这样就省了将 G729 编码转换为 PCM 数据的过程。

L484, 如果文件中的声道数大于当前需要的声道数, 则会尝试将多个声道合并。

```

464 more:
465
466     if (fh->pre_buffer) {
467         switch_size_t rlen;
468         int asis = switch_test_flag(fh, SWITCH_FILE_NATIVE);
469
470         if (!switch_test_flag(fh, SWITCH_FILE_BUFFER_DONE)) {
471             rlen = asis ? fh->pre_buffer_data_len : fh->pre_buffer_data_len / 2 / fh->real_channels;
472
473             if (switch_buffer_inuse(fh->pre_buffer) < rlen * 2 * fh->channels) {
474                 if ((status = fh->file_interface->file_read(fh, fh->pre_buffer_data, &rlen)) ==
↪ SWITCH_STATUS_BREAK) {

```



```

475         return SWITCH_STATUS_BREAK;
476     }
477
478
479     if (status != SWITCH_STATUS_SUCCESS || !rlen) {
480         switch_set_flag_locked(fh, SWITCH_FILE_BUFFER_DONE);
481     } else {
482         fh->samples_in += rlen;
483         if (fh->real_channels != fh->channels && !switch_test_flag(fh, SWITCH_FILE_NOMUX)) {
484             switch_mux_channels((int16_t *) fh->pre_buffer_data, rlen, fh->real_channels,
↳ fh->channels);
485         }
486         switch_buffer_write(fh->pre_buffer, fh->pre_buffer_data, asis ? rlen : rlen * 2 *
↳ fh->channels);
487     }
488 }
489 }
490
491 rlen = switch_buffer_read(fh->pre_buffer, data, asis ? *len : *len * 2 * fh->channels);
492 *len = asis ? rlen : rlen / 2 / fh->channels;
493
494 if (*len == 0) {
495     switch_set_flag_locked(fh, SWITCH_FILE_DONE);
496     goto top;
497 } else {
498     status = SWITCH_STATUS_SUCCESS;
499 }

```

如果不使用预读缓冲区，则代码简单得多，直接从文件读取数据（L503）。

如果不是 Native File 并且采样率不匹配（L519），则尝试启动一个 **resampler**（L521）进行采样率转换（L528）。

```

501     } else {
502
503         if ((status = fh->file_interface->file_read(fh, data, len)) == SWITCH_STATUS_BREAK) {
504             return SWITCH_STATUS_BREAK;
505         }
506
507         ...
519         if (!switch_test_flag(fh, SWITCH_FILE_NATIVE) && fh->native_rate != fh->samplerate) {
520             if (!fh->resampler) {
521                 if (switch_resample_create(&fh->resampler,
522                                         fh->native_rate, fh->samplerate, (uint32_t) orig_len,
↳ SWITCH_RESAMPLE_QUALITY, fh->channels) != SWITCH_STATUS_SUCCESS) {
523                     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Unable to create resampler!\n");
524                     return SWITCH_STATUS_GENERR;

```

```

525         }
526     }
527
528     switch_resample_process(fh->resampler, data, (uint32_t) *len);

```

检测文件是否支持视频。

```

563 SWITCH_DECLARE(switch_bool_t) switch_core_file_has_video(switch_file_handle_t *fh, switch_bool_t
↪ check_open)

```

向文件中写入。也会根据情况决定是否启动 **resampler** (L604)，也支持 **pre_buffer** (预写, L640, L655)，或直接写入文件 (L664)。

```

568 SWITCH_DECLARE(switch_status_t) switch_core_file_write(switch_file_handle_t *fh, void *data,
↪ switch_size_t *len)
569 {
...
604     if (!switch_test_flag(fh, SWITCH_FILE_NATIVE) && fh->native_rate != fh->samplerate) {
605         if (!fh->resampler) {
606             if (switch_resample_create(&fh->resampler,
607                                     fh->native_rate,
608                                     fh->samplerate,
609                                     (uint32_t) orig_len * 2 * fh->channels, SWITCH_RESAMPLE_QUALITY,
↪ fh->channels) != SWITCH_STATUS_SUCCESS) {
610                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Unable to create resampler!\n");
611                 return SWITCH_STATUS_GENERR;
612             }
613         }
614
615         switch_resample_process(fh->resampler, data, (uint32_t) * len);
...
640     if (fh->pre_buffer) {
...
655         if ((status = fh->file_interface->file_write(fh, fh->pre_buffer_data, &blen)) !=
↪ SWITCH_STATUS_SUCCESS) {
656             *len = 0;
657         }
...
662     } else {
663         switch_status_t status;
664         if ((status = fh->file_interface->file_write(fh, data, len)) == SWITCH_STATUS_SUCCESS) {
665             fh->samples_out += orig_len;
666         }

```

```
667         return status;
668     }
669 }
```

写视频数据。其中，视频在一个视频帧（`frame`）中，原始图像格式（`frame->img`，I420 格式）。

```
671 SWITCH_DECLARE(switch_status_t) switch_core_file_write_video(switch_file_handle_t *fh, switch_frame_t
↳ *frame)
```

从文件中读取一帧，成功后返回 `frame->img`，I420 格式。

```
692 SWITCH_DECLARE(switch_status_t) switch_core_file_read_video(switch_file_handle_t *fh, switch_frame_t
↳ *frame, switch_video_read_flag_t flags)
```

移动文件指针（以支持快进、快退等），其中，移动的步长是以 `sample`（即抽样数据的个数）计的。

```
716 SWITCH_DECLARE(switch_status_t) switch_core_file_seek(switch_file_handle_t *fh, unsigned int *cur_pos,
↳ int64_t samples, int whence)
```

设置和读取文件的元数据，如歌手，专辑等。

```
767 SWITCH_DECLARE(switch_status_t) switch_core_file_set_string(switch_file_handle_t *fh, switch_audio_col_t
↳ col, const char *string)
783 SWITCH_DECLARE(switch_status_t) switch_core_file_get_string(switch_file_handle_t *fh, switch_audio_col_t
↳ col, const char **string)
```

清空文件。

```
813 SWITCH_DECLARE(switch_status_t) switch_core_file_truncate(switch_file_handle_t *fh, int64_t offset)
```

文件控制，在打开文件后，对文件指针进行控制，如对音视频编码和缓冲区进行控制等，如清空预读缓冲区（L855），暂停（`SCFC_PAUSE_READ`）等。

```
843 SWITCH_DECLARE(switch_status_t) switch_core_file_command(switch_file_handle_t *fh, switch_file_command_t  
↪ command)  
855     case SCFC_FLUSH_AUDIO:
```

预关闭文件。在某些情况下（如录音），需要知道文件的大小，所以可以将文件置于预关闭状态，以获取文件的大小。

```
873 SWITCH_DECLARE(switch_status_t) switch_core_file_pre_close(switch_file_handle_t *fh)
```

关闭文件。

```
918 SWITCH_DECLARE(switch_status_t) switch_core_file_close(switch_file_handle_t *fh)
```

以上函数只是对实际模块实现的一些抽象和封装，实际的文件操作功能都由具体的功能模块实现。

4.18 switch_core_hash.c

本章基于 Commit Hash [1681db4](#)。

散列表（旧称哈希表）相关函数。它实际上是封装了一个开源的散列表实现（Copyright (C) 2002, 2004 Christopher Clark）。FreeSWITCH 没有用 APR 的散列表函数，主要是因为 APR 的散列表会对 Key 进行缓存，而 FreeSWITCH 中的 Channel UUID 是随机的，缓存显然是不适合的。

初始化，`case_sensitive` 指定 Key 是否区分大小写。

```
39 SWITCH_DECLARE(switch_status_t) switch_core_hash_init_case(switch_hash_t **hash, switch_bool_t  
↪ case_sensitive)
```

销毁。

```
49 SWITCH_DECLARE(switch_status_t) switch_core_hash_destroy(switch_hash_t **hash)
```

插入数据，同时插入一个销毁回调函数，当相关的项目从表中删除时，自动执行回调函数销毁相关的对象。

```
58 SWITCH_DECLARE(switch_status_t) switch_core_hash_insert_destructor(switch_hash_t *hash, const char *key,  
↪ const void *data, hashtable_destructor_t destructor)
```

插入，同时加锁。

```
67 SWITCH_DECLARE(switch_status_t) switch_core_hash_insert_locked(switch_hash_t *hash, const char *key,  
↪ const void *data, switch_mutex_t *mutex)
```

插入，同时加写锁。

```
84 SWITCH_DECLARE(switch_status_t) switch_core_hash_insert_wrlock(switch_hash_t *hash, const char *key,  
↪ const void *data, switch_thread_rwlock_t *rwlock)
```

从表中删除数据。

```
101 SWITCH_DECLARE(void *) switch_core_hash_delete(switch_hash_t *hash, const char *key)
```

删除，同时加锁。

```
106 SWITCH_DECLARE(void *) switch_core_hash_delete_locked(switch_hash_t *hash, const char *key,  
↪ switch_mutex_t *mutex)
```

删除，加写锁。

```
123 SWITCH_DECLARE(void *) switch_core_hash_delete_wrlock(switch_hash_t *hash, const char *key,  
↪ switch_thread_rwlock_t *rwlock)
```

删除多个数据项。

```
140 SWITCH_DECLARE(switch_status_t) switch_core_hash_delete_multi(switch_hash_t *hash,  
↪ switch_hash_delete_callback_t callback, void *pData) {
```

查找。

```
176 SWITCH_DECLARE(void *) switch_core_hash_find(switch_hash_t *hash, const char *key)
```

查找，事先加锁。

```
181 SWITCH_DECLARE(void *) switch_core_hash_find_locked(switch_hash_t *hash, const char *key, switch_mutex_t  
↪ *mutex)
```

查找，加读锁。

```
199 SWITCH_DECLARE(void *) switch_core_hash_find_rdlock(switch_hash_t *hash, const char *key,  
↪ switch_thread_rwlock_t *rwlock)
```

是否为空？

```
216 SWITCH_DECLARE(switch_bool_t) switch_core_hash_empty(switch_hash_t *hash)
```

遍历，取第一个。

```
229 SWITCH_DECLARE(switch_hash_index_t *) switch_core_hash_first_iter(switch_hash_t *hash,  
↪ switch_hash_index_t *hi)
```

遍历，取下一个。

```
234 SWITCH_DECLARE(switch_hash_index_t *) switch_core_hash_next(switch_hash_index_t **hi)
```

取得当前项目。

```
239 SWITCH_DECLARE(void) switch_core_hash_this(switch_hash_index_t *hi, const void **key, switch_ssize_t
↳ *klen, void **val)
```

取得当前项目的值。

```
244 SWITCH_DECLARE(void) switch_core_hash_this_val(switch_hash_index_t *hi, void *val)
```

整数散列表相关函数，Key 为整数。

```
250 SWITCH_DECLARE(switch_status_t) switch_core_inthash_init(switch_inthash_t **hash)
255 SWITCH_DECLARE(switch_status_t) switch_core_inthash_destroy(switch_inthash_t **hash)
262 SWITCH_DECLARE(switch_status_t) switch_core_inthash_insert(switch_inthash_t *hash, uint32_t key, const
↳ void *data)
274 SWITCH_DECLARE(void *) switch_core_inthash_delete(switch_inthash_t *hash, uint32_t key)
279 SWITCH_DECLARE(void *) switch_core_inthash_find(switch_inthash_t *hash, uint32_t key)
```

4.19 switch_core_io.c

本章基于 Commit Hash [1681db4](#)。

本文件主要是 FreeSWITCH 核心的输入输出。

产生编码后的静音包。对于 G729 而言，就直接定义了相应的值（L41~L56），对 PCM 数据，直接设为 [255](#)（L62）。

```
39 SWITCH_DECLARE(void) switch_core_gen_encoded_silence(unsigned char *data, const
↳ switch_codec_implementation_t *read_impl, switch_size_t len)
40 {
41     unsigned char g729_filler[] = {
42         114, 170, 250, 103, 54, 211, 203, 194, 94, 64,
43         ...
44     };
45     if (read_impl->ianacode == 18 || switch_stristr("g729", read_impl->iananame)) {
46         memcpy(data, g729_filler, len);
47     }
```

```

61     } else {
62         memset(data, 255, len);
63     }
64 }
65 }

```

读取一帧。其中 `tap_only` (L79) 是在直接读 RTP 的数据时用的，如在 G729 的情况下，没有开源的编解码器可用，就可以直接读取 G729 数据，如果直接写到文件里也可以录音，只不过在播放的时候需要有 G729 解码器。

L81，避免长期占用 CPU，让出一会儿，一般会 Sleep 1 秒。

L83~L89，如果无法锁定（多线程环境下被其它进程占用），则返回静音包。

```

67 SWITCH_DECLARE(switch_status_t) switch_core_session_read_frame(switch_core_session_t *session,
↪ switch_frame_t **frame, switch_io_flag_t flags,
68                                     int stream_id)
69 {
70     tap_only = switch_test_flag(session, SSF_MEDIA_BUG_TAP_ONLY);
71
72     switch_os_yield();
73
74     if (switch_mutex_trylock(session->codec_read_mutex) == SWITCH_STATUS_SUCCESS) {
75         switch_mutex_unlock(session->codec_read_mutex);
76     } else {
77         switch_cond_next();
78         *frame = &runtime.dummy_cng_frame;
79         return SWITCH_STATUS_SUCCESS;
80     }
81 }

```

L91~L98，如果编码没有协商成功，也返回静音包。

```

91     if (!(session->read_codec && session->read_codec->implementation && switch_core_codec_ready(session-
↪ >read_codec))) {
92         if (switch_channel_test_flag(session->channel, CF_PROXY_MODE) ||
↪ switch_channel_get_state(session->channel) == CS_HIBERNATE) {
93             switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_CRIT, "%s reading on a
↪ session with no media!\n",
94                             switch_channel_get_name(session->channel));
95             switch_cond_next();
96             *frame = &runtime.dummy_cng_frame;
97             return SWITCH_STATUS_SUCCESS;
98         }

```

L112, L122, 加锁。

L126~L131, 如果在读的过程中读到 DTMF, 则 Ping 一下状态机 (L129), 通知有 DTMF。

```
112     switch_mutex_lock(session->codec_read_mutex);
122     switch_mutex_lock(session->read_codec->mutex);
123
124     top:
125
126     for(i = 0; i < 2; i++) {
127         if (session->dmachine[i]) {
128             switch_channel_dtmf_lock(session->channel);
129             switch_ivr_dmachine_ping(session->dmachine[i], NULL);
130             switch_channel_dtmf_unlock(session->channel);
131         }
132     }
```

在当前 Channel 上触发一个 `SWITCH_MESSAGE_HEARTBEAT_EVENT` 事件。

```
144     *frame = NULL;
145
146     if (session->read_codec && !session->track_id && session->track_duration) {
147         if (session->read_frame_count == 0) {
148             switch_event_t *event;
149             switch_core_session_message_t msg = { 0 };
150
151             session->read_frame_count = (session->read_impl.samples_per_second / session-
↵ >read_impl.samples_per_packet) * session->track_duration;
152
153             msg.message_id = SWITCH_MESSAGE_HEARTBEAT_EVENT;
154             msg.numeric_arg = session->track_duration;
155             switch_core_session_receive_message(session, &msg);
156
157             switch_event_create(&event, SWITCH_EVENT_SESSION_HEARTBEAT);
158             switch_channel_event_set_data(session->channel, event);
159             switch_event_fire(&event);
160         } else {
161             session->read_frame_count--;
162         }
163     }
```

如果 Channel 处于 Hold 状态, 则返回 `BREAK`。

```

166     if (switch_channel_test_flag(session->channel, CF_HOLD)) {
167         switch_yield(session->read_impl.microseconds_per_packet);
168         status = SWITCH_STATUS_BREAK;
169         goto even_more_done;
170     }

```

L172, 如果相应的 Endpoint 实现了 `io_routines->read_frame` 回调函数（如 `mod_sofia` 中的 `sofia_read_frame`），则执行相关回调函数（L175）读取数据。如果读取成功，则检查当前 Channel 上是否安装了 Event Hook，如果安装了的话，回调每一个 Hook 上的回调函数（L176 ~ L177）。

```

172     if (session->endpoint_interface->io_routines->read_frame) {
173         switch_mutex_unlock(session->read_codec->mutex);
174         switch_mutex_unlock(session->codec_read_mutex);
175         if ((status = session->endpoint_interface->io_routines->read_frame(session, frame, flags,
↵ stream_id)) == SWITCH_STATUS_SUCCESS) {
176             for (ptr = session->event_hooks.read_frame; ptr; ptr = ptr->next) {
177                 if ((status = ptr->read_frame(session, frame, flags, stream_id)) !=
↵ SWITCH_STATUS_SUCCESS) {
178                     break;
179                 }
180             }
181         }

```

如果 Endpoint 返回 `SWITCH_STATUS_INUSE`，则返回静音包。

```

183     if (status == SWITCH_STATUS_INUSE) {
184         *frame = &runtime.dummy_cng_frame;
185         switch_yield(20000);
186         return SWITCH_STATUS_SUCCESS;
187     }

```

检查状态，如果读到非预期的结果，则返回失败。

```

189     if (!SWITCH_READ_ACCEPTABLE(status) || !session->read_codec || !switch_core_codec_ready(session-
↵ >read_codec)) {
190         *frame = NULL;
191         return SWITCH_STATUS_FALSE;
192     }

```

更多检查略.....

如果当前 Channel 处于 `SFF_PROXY_PACKET` 状态，则直接返回成功。

```

222     if (switch_test_flag(*frame, SFF_PROXY_PACKET)) {
223         /* Fast PASS! */
224         status = SWITCH_STATUS_SUCCESS;
225         goto done;
226     }

```

否则，检查编码器，下面要进行转码了。

```

228     switch_assert((*frame)->codec != NULL);
229
230     if (!(session->read_codec && (*frame)->codec && (*frame)->codec->implementation) &&
↪ switch_core_codec_ready((*frame)->codec)) {
231         status = SWITCH_STATUS_FALSE;
232         goto done;
233     }

```

如果当前 Channel 上有 Media Bug（如录音），则循环（L242）执行每一个 Media Bug 的回调（L265），这里只是处理了 TAP 状态（不转码）的情况（L258）。

```

235     if (session->bugs && !((*frame)->flags & SFF_CNG) && !((*frame)->flags & SFF_NOT_AUDIO)) {
242         for (bp = session->bugs; bp; bp = bp->next) {
257             if (bp->ready) {
258                 if (switch_test_flag(bp, SMBF_TAP_NATIVE_READ)) {
259                     if ((*frame)->codec && (*frame)->codec->implementation &&
260                         (*frame)->codec->implementation->encoded_bytes_per_packet &&
261                         (*frame)->datalen != (*frame)->codec->implementation->encoded_bytes_per_packet)
↪ {
262                         switch_set_flag((*frame), SFF_CNG);
263                         break;
264                     }
265                     if (bp->callback) {
266                         bp->native_read_frame = *frame;
267                         ok = bp->callback(bp, bp->user_data, SWITCH_ABC_TYPE_TAP_NATIVE_READ);
268                         bp->native_read_frame = NULL;
269                     }
270                 }
271             }

```

判断是否需要转码。如果当前 Channel 的编码跟预期的不一致就需要转码。

```
287     if (session->read_codec->implementation->impl_id != codec_impl.impl_id) {
288         need_codec = TRUE;
289         tap_only = 0;
290     }
```

判断是否需要 Resample（转换采样频率）。

```
292     if (codec_impl.actual_samples_per_second != session->read_impl.actual_samples_per_second) {
293         do_resample = 1;
294     }
```

L361, 如果读到静音包, 则判断是否是 Bridge 状态 (L366), 如果是, 则查找另一条腿 (L367) 上是否有 Media Bug (L368), 进而决定是否继续处理下面的逻辑 (375)。

```
361     if (switch_test_flag(*frame, SFF_CNG)) {
362         if (!session->bugs && !session->plc) {
363             /* Check if other session has bugs */
364             unsigned int other_session_bugs = 0;
365             switch_core_session_t *other_session = NULL;
366             if (switch_channel_test_flag(switch_core_session_get_channel(session), CF_BRIDGED) &&
367                 switch_core_session_get_partner(session, &other_session) == SWITCH_STATUS_SUCCESS) {
368                 if (other_session->bugs && !switch_test_flag(other_session, SSF_MEDIA_BUG_TAP_ONLY)) {
369                     other_session_bugs = 1;
370                 }
371                 switch_core_session_rwlock_unlock(other_session);
372             }
373
374             /* Don't process CNG frame */
375             if (!other_session_bugs) {
376                 status = SWITCH_STATUS_SUCCESS;
377                 goto done;
378             }
379         }
380         is_cng = 1;
381         need_codec = 1;
382     } else if (switch_test_flag(*frame, SFF_NOT_AUDIO)) {
383         do_resample = 0;
384         do_bugs = 0;
385         need_codec = 0;
386     }
```

重置转码条件，用于在转码中后来又不需要转码的情况。

```

388     if (switch_test_flag(session, SSF_READ_TRANSCODE) && !need_codec && switch_core_codec_ready(session-
↪ >read_codec)) {
389         switch_core_session_t *other_session;
390         const char *uuid = switch_channel_get_partner_uuid(switch_core_session_get_channel(session));
391         switch_clear_flag(session, SSF_READ_TRANSCODE);
392
393         if (uuid && (other_session = switch_core_session_locate(uuid))) {
394             switch_set_flag(other_session, SSF_READ_CODEC_RESET);
395             switch_set_flag(other_session, SSF_READ_CODEC_RESET);
396             switch_set_flag(other_session, SSF_WRITE_CODEC_RESET);
397             switch_core_session_rwlock_unlock(other_session);
398         }
399     }
400
401     if (switch_test_flag(session, SSF_READ_CODEC_RESET)) {
402         switch_core_codec_reset(session->read_codec);
403         switch_clear_flag(session, SSF_READ_CODEC_RESET);
404     }

```

L416，在当前 Channel 上发送一个转码的消息。

```

407     if (status == SWITCH_STATUS_SUCCESS && need_codec) {
408         switch_frame_t *enc_frame, *read_frame = *frame;
409
410         switch_set_flag(session, SSF_READ_TRANSCODE);
411
412         if (!switch_test_flag(session, SSF_WARN_TRANSCODE)) {
413             switch_core_session_message_t msg = { 0 };
414
415             msg.message_id = SWITCH_MESSAGE_INDICATE_TRANSCODING_NECESSARY;
416             switch_core_session_receive_message(session, &msg);
417             switch_set_flag(session, SSF_WARN_TRANSCODE);
418         }

```

如果读到 CNG（静音包），并且需要补偿（L424），则产生实际的静音数据（L425）。

```

420     if (read_frame->codec || (is_cng && session->plc)) {
421         session->raw_read_frame.datalen = session->raw_read_frame.buflen;
422
423         if (is_cng) {

```

```

424         if (session->plc) {
425             plc_fillin(session->plc, session->raw_read_frame.data, read_frame->codec-
↳ >implementation->decoded_bytes_per_packet / 2);
426             is_cng = 0;
427             flag &= ~SFF_CNG;
428         } else {
429             memset(session->raw_read_frame.data, 255, read_frame->codec->implementation-
↳ >decoded_bytes_per_packet);
430         }
431
432         session->raw_read_frame.timestamp = 0;
433         session->raw_read_frame.datalen = read_frame->codec->implementation-
↳ >decoded_bytes_per_packet;
434         session->raw_read_frame.samples = session->raw_read_frame.datalen / sizeof(int16_t) /
↳ session->read_impl.number_of_channels;
435         session->raw_read_frame.channels = read_frame->codec->implementation-
↳ >number_of_channels;
436         read_frame = &session->raw_read_frame;
437         status = SWITCH_STATUS_SUCCESS;

```

L438, 不是静音包, 如果处理处理 Media Bug (L440), 则解码 (L498), 如果解码失败则返回 (L506)。

```

438     } else {
439         switch_codec_t *use_codec = read_frame->codec;
440         if (do_bugs) {
441             ...
483             switch_codec_t *codec = use_codec;
484             ...
496             codec->cur_frame = read_frame;
497             session->read_codec->cur_frame = read_frame;
498             status = switch_core_codec_decode(codec,
499                 session->read_codec,
500                 read_frame->data,
501                 read_frame->datalen,
502                 session->read_impl.actual_samples_per_second,
503                 session->raw_read_frame.data,
504                 &session->raw_read_frame.datalen, &session->raw_read_frame.rate,
505                 &read_frame->flags);
506             if (status == SWITCH_STATUS_NOT_INITIALIZED) {
507                 switch_thread_rwlock_unlock(session->bug_rwlock);
508                 goto done;
509             }

```

如果需要 Resample，则转换采样率。

```

533         if (do_resample && ((status == SWITCH_STATUS_SUCCESS) || is_cng)) {
534             status = SWITCH_STATUS_RESAMPLE;
535         }
536
537         /* mux or demux to match */
538         if (session->raw_read_frame.channels != session->read_impl.number_of_channels) {
539             uint32_t rlen = session->raw_read_frame.datalen / 2 / session->raw_read_frame.channels;
540             switch_mux_channels((int16_t *) session->raw_read_frame.data, rlen, session-
↪ >raw_read_frame.channels, session->read_impl.number_of_channels);
541             session->raw_read_frame.datalen = rlen * 2 * session->read_impl.number_of_channels;
542             session->raw_read_frame.samples = session->raw_read_frame.datalen / 2;
543             session->raw_read_frame.channels = session->read_impl.number_of_channels;
544         }
545         ...
546
547         switch (status) {
548             case SWITCH_STATUS_RESAMPLE:
549                 if (!session->read_resampler) {
550                     switch_mutex_lock(session->resample_mutex);
551
552                     status = switch_resample_create(&session->read_resampler,
553                                                     read_frame->codec->implementation->actual_samples_per_second,
554                                                     session->read_impl.actual_samples_per_second,
555                                                     session->read_impl.decoded_bytes_per_packet, SWITCH_RESAMPLE_QUALITY,
556                                                     session->read_impl.number_of_channels);
557
558                     ...
559
560                 case SWITCH_STATUS_SUCCESS:
561                     ...
562
563                 case SWITCH_STATUS_NOOP:
564                     ...
565
566                     }
567                     status = SWITCH_STATUS_SUCCESS;
568                     break;
569                 case SWITCH_STATUS_BREAK:
570                     ...
571
572                 case SWITCH_STATUS_NOT_INITIALIZED:
573                     ...
574
575             }
576         }
577     }
578 }

```

继续处理 Media Bug，循环（L652）执行相关回调（L674，L678）。

```

646         if (session->bugs) {
647             switch_media_bug_t *bp;

```

```

648         switch_bool_t ok = SWITCH_TRUE;
649         int prune = 0;
650         switch_thread_rwlock_rdlock(session->bug_rwlock);
651
652         for (bp = session->bugs; bp; bp = bp->next) {
653             ok = SWITCH_TRUE;
654             ...
655             if (ok && switch_test_flag(bp, SMBF_READ_REPLACE)) {
656                 do_bugs = 0;
657                 if (bp->callback) {
658                     bp->read_replace_frame_in = read_frame;
659                     bp->read_replace_frame_out = read_frame;
660                     bp->read_demux_frame = NULL;
661                     if ((ok = bp->callback(bp, bp->user_data, SWITCH_ABC_TYPE_READ_REPLACE)) ==
662                         SWITCH_TRUE) {
663                         read_frame = bp->read_replace_frame_out;
664                     }
665                 }
666             }
667             ...
668         }

```

继续处理 Media Bug。

```

697         if (session->bugs) {
698             switch_media_bug_t *bp;
699             switch_bool_t ok = SWITCH_TRUE;
700             int prune = 0;
701             switch_thread_rwlock_rdlock(session->bug_rwlock);
702
703             for (bp = session->bugs; bp; bp = bp->next) {
704                 ok = SWITCH_TRUE;
705                 ...
706                 if (ok && bp->ready && switch_test_flag(bp, SMBF_READ_STREAM)) {
707                     ...
708                     if (bp->callback) {
709                         ok = bp->callback(bp, bp->user_data, SWITCH_ABC_TYPE_READ);
710                     }
711                     switch_mutex_unlock(bp->read_mutex);
712                 }
713                 ...
714             }

```

处理读到的数据，写到一个 `session->raw_read_buffer` 里 (L785)。

```

762     if (session->read_codec) {
775         if (read_frame->datalen == session->read_impl.decoded_bytes_per_packet) {
776             perfect = TRUE;
777         } else {
778             if (!session->raw_read_buffer) {
779                 switch_size_t bytes = session->read_impl.decoded_bytes_per_packet;
780                 switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Engaging
↪ Read Buffer at %u bytes vs %u\n",
781                                 (uint32_t) bytes, (uint32_t) (*frame)->datalen);
782                 switch_buffer_create_dynamic(&session->raw_read_buffer, bytes *
↪ SWITCH_BUFFER_BLOCK_FRAMES, bytes * SWITCH_BUFFER_START_FRAMES, 0);
783             }
784
785             if (read_frame->datalen && (!switch_buffer_write(session->raw_read_buffer, read_frame-
↪ >data, read_frame->datalen))) {
786                 status = SWITCH_STATUS_MEMERR;
787                 goto done;
788             }
789         }

```

L791, 如果缓冲区中有足够的数据可以发送, 则编码 (L812) 存放到 `session->enc_read_frame`。该函数的更多逻辑略。

```

791     if (perfect || switch_buffer_inuse(session->raw_read_buffer) >= session-
↪ >read_impl.decoded_bytes_per_packet) {
...
812         status = switch_core_codec_encode(session->read_codec,
813                                           enc_frame->codec,
814                                           enc_frame->data,
815                                           enc_frame->datalen,
816                                           session->read_impl.actual_samples_per_second,
817                                           session->enc_read_frame.data, &session-
↪ >enc_read_frame.datalen, &session->enc_read_frame.rate, &flag);
818         switch_assert(session->enc_read_frame.datalen <= SWITCH_RECOMMENDED_BUFFER_SIZE);
...
946 }

```

L956, 向 Channel 发一个信号。如果 Endpoint 实现了相关回调的情况下 (L956), 比如可以通知相关的 Channel 从阻塞中退出等。

```

948 static char *SIG_NAMES[] = {
949     "NONE",

```

```

950     "KILL",
951     "XFER",
952     "BREAK",
953     NULL
954 };
955
956 SWITCH_DECLARE(switch_status_t) switch_core_session_perform_kill_channel(switch_core_session_t *session,
957     const char *file, const char
    ↪ *func, int line, switch_signal_t sig)
958 {
959     switch_io_event_hook_kill_channel_t *ptr;
960     switch_status_t status = SWITCH_STATUS_FALSE;
961
962     switch_log_printf(SWITCH_CHANNEL_ID_LOG, file, func, line, switch_core_session_get_uuid(session),
    ↪ SWITCH_LOG_DEBUG10, "Send signal %s [%s]\n",
963         switch_channel_get_name(session->channel), SIG_NAMES[sig]);
964
965     if (session->endpoint_interface->io_routines->kill_channel) {
966         if ((status = session->endpoint_interface->io_routines->kill_channel(session, sig)) ==
    ↪ SWITCH_STATUS_SUCCESS) {
967             for (ptr = session->event_hooks.kill_channel; ptr; ptr = ptr->next) {
968                 if ((status = ptr->kill_channel(session, sig)) != SWITCH_STATUS_SUCCESS) {
969                     break;
970                 }
971             }
972         }
973     }
974     return status;
975 }

```

接收 DTMF，执行相关的回调函数（L1016）。

```

977 SWITCH_DECLARE(switch_status_t) switch_core_session_recv_dtmf(switch_core_session_t *session, const
    ↪ switch_dtmf_t *dtmf)
978 {
979     ...
1015     for (ptr = session->event_hooks.recv_dtmf; ptr; ptr = ptr->next) {
1016         if ((status = ptr->recv_dtmf(session, &new_dtmf, SWITCH_DTMF_RECV)) !=
    ↪ SWITCH_STATUS_SUCCESS) {
1017             return status;
1018         }
1019     }
1021
1022     return fed ? SWITCH_STATUS_FALSE : SWITCH_STATUS_SUCCESS;
1023 }

```

发送 DTMF (L1055)，**w**和**W**不是实际的 DTMF，而且暂停相应的时间（一般分别是**500**毫秒和**1**秒）。同时调用 Event Hooks (L1085) 及 Ping 相应的状态机 (L1092)。

```

1025 SWITCH_DECLARE(switch_status_t) switch_core_session_send_dtmf(switch_core_session_t *session, const
↳ switch_dtmf_t *dtmf)
1026 {
...
1052     for(p = digits; p && *p; p++) {
1053         if (is_dtmf(*p)) {
1054             x_dtmf.digit = *p;
1055             switch_core_session_send_dtmf(session, &x_dtmf);
1056         }
1057     }
...
1067
1068     if (new_dtmf.digit != 'w' && new_dtmf.digit != 'W') {
...
1078     }
1079
1080     if (!new_dtmf.duration) {
1081         new_dtmf.duration = switch_core_default_dtmf_duration(0);
1082     }
1083
1084     if (!switch_test_flag(dtmf, DTMF_FLAG_SKIP_PROCESS)) {
1085         for (ptr = session->event_hooks.send_dtmf; ptr; ptr = ptr->next) {
1086             if ((status = ptr->send_dtmf(session, dtmf, SWITCH_DTMF_SEND)) != SWITCH_STATUS_SUCCESS)
↳ {
1087                 return SWITCH_STATUS_SUCCESS;
1088             }
1089         }
1090         if (session->dmachine[1]) {
1091             char str[2] = { new_dtmf.digit, '\0' };
1092             switch_ivr_dmachine_feed(session->dmachine[1], str, NULL);
1093             return SWITCH_STATUS_SUCCESS;
1094         }
1095     }
1096
1097
1098     if (session->endpoint_interface->io_routines->send_dtmf) {
1099         int send = 0;
1100         status = SWITCH_STATUS_SUCCESS;
1101
1102         if (switch_channel_test_cap(session->channel, CC_QUEUEABLE_DTMF_DELAY) && (dtmf->digit ==
↳ 'w' || dtmf->digit == 'W')) {
1103             send = 1;
1104         } else {
1105             if (dtmf->digit == 'w') {

```

```
1106         switch_yield(500000);
1107     } else if (dtmf->digit == 'W') {
1108         switch_yield(1000000);
1109     } else {
1110         send = 1;
1111     }
1112 }
1113
1114 if (send) {
1115     status = session->endpoint_interface->io_routines->send_dtmf(session, &new_dtmf);
1116 }
1117 }
1118 return status;
1119 }
```

发送 DTMF 字符串。

```
1121 SWITCH_DECLARE(switch_status_t) switch_core_session_send_dtmf_string(switch_core_session_t *session,
↪ const char *dtmf_string)
```

4.20 switch_core_media_bug.c

本章基于 Commit Hash [1681db4](#)。

销毁 Media Bug。

```
38 static void switch_core_media_bug_destroy(switch_media_bug_t **bug)
```

暂停及继续 Media Bug。

```
92 SWITCH_DECLARE(void) switch_core_media_bug_pause(switch_core_session_t *session)
93 {
94     switch_channel_set_flag(session->channel, CF_PAUSE_BUGS);
95 }
96
97 SWITCH_DECLARE(void) switch_core_media_bug_resume(switch_core_session_t *session)
98 {
99     switch_channel_clear_flag(session->channel, CF_PAUSE_BUGS);
100 }
```

设置清除相关的 `flag` 和 `param`。

```
102 SWITCH_DECLARE(uint32_t) switch_core_media_bug_test_flag(switch_media_bug_t *bug, uint32_t flag)
107 SWITCH_DECLARE(uint32_t) switch_core_media_bug_set_flag(switch_media_bug_t *bug, uint32_t flag)
115 SWITCH_DECLARE(uint32_t) switch_core_media_bug_clear_flag(switch_media_bug_t *bug, uint32_t flag)
120 SWITCH_DECLARE(void) switch_core_media_bug_set_media_params(switch_media_bug_t *bug, switch_mm_t *mm)
125 SWITCH_DECLARE(void) switch_core_media_bug_get_media_params(switch_media_bug_t *bug, switch_mm_t *mm)
```

获取当前 Session。

```
130 SWITCH_DECLARE(switch_core_session_t *) switch_core_media_bug_get_session(switch_media_bug_t *bug)
```

获取 RTT 文本。

```
135 SWITCH_DECLARE(const char *) switch_core_media_bug_get_text(switch_media_bug_t *bug)
136 {
137     return bug->text_framedata;
138 }
```

获取视频帧。

```
140 SWITCH_DECLARE(switch_frame_t *) switch_core_media_bug_get_video_ping_frame(switch_media_bug_t *bug)
141 {
142     return bug->video_ping_frame;
143 }
```

获取写视频帧，用于替换内容。

```
145 SWITCH_DECLARE(switch_frame_t *) switch_core_media_bug_get_write_replace_frame(switch_media_bug_t *bug)
146 {
147     return bug->write_replace_frame_in;
148 }
```

设置写视频帧，用于替换内容。

```
150 SWITCH_DECLARE(void) switch_core_media_bug_set_write_replace_frame(switch_media_bug_t *bug,  
↪ switch_frame_t *frame)  
151 {  
152     bug->write_replace_frame_out = frame;  
153 }
```

获取读音频帧，用于替换。

```
155 SWITCH_DECLARE(switch_frame_t *) switch_core_media_bug_get_read_replace_frame(switch_media_bug_t *bug)  
156 {  
157     return bug->read_replace_frame_in;  
158 }
```

获取 Native 帧，不转码。

```
160 SWITCH_DECLARE(switch_frame_t *) switch_core_media_bug_get_native_read_frame(switch_media_bug_t *bug)  
161 {  
162     return bug->native_read_frame;  
163 }
```

获取 Native 写帧，不转码。

```
165 SWITCH_DECLARE(switch_frame_t *) switch_core_media_bug_get_native_write_frame(switch_media_bug_t *bug)  
166 {  
167     return bug->native_write_frame;  
168 }
```

设置读帧，用于替换。

```
170 SWITCH_DECLARE(void) switch_core_media_bug_set_read_replace_frame(switch_media_bug_t *bug,  
↪ switch_frame_t *frame)  
171 {  
172     bug->read_replace_frame_out = frame;  
173 }
```

设置读帧，不混音。

```
175 SWITCH_DECLARE(void) switch_core_media_bug_set_read_demux_frame(switch_media_bug_t *bug, switch_frame_t
↳ *frame)
176 {
177     bug->read_demux_frame = frame;
178 }
```

获取 Media Bug 上的用户自定义数据。

```
180 SWITCH_DECLARE(void *) switch_core_media_bug_get_user_data(switch_media_bug_t *bug)
181 {
182     return bug->user_data;
183 }
```

清空数据。

```
185 SWITCH_DECLARE(void) switch_core_media_bug_flush(switch_media_bug_t *bug)
```

返回在用的数据指针。

```
206 SWITCH_DECLARE(void) switch_core_media_bug_inuse(switch_media_bug_t *bug, switch_size_t *readp,
↳ switch_size_t *writep)
207 {
208     if (switch_test_flag(bug, SMBF_READ_STREAM)) {
209         switch_mutex_lock(bug->read_mutex);
210         *readp = bug->raw_read_buffer ? switch_buffer_inuse(bug->raw_read_buffer) : 0;
211         switch_mutex_unlock(bug->read_mutex);
212     } else {
213         *readp = 0;
214     }
215
216     if (switch_test_flag(bug, SMBF_WRITE_STREAM)) {
217         switch_mutex_lock(bug->write_mutex);
218         *writep = bug->raw_write_buffer ? switch_buffer_inuse(bug->raw_write_buffer) : 0;
219         switch_mutex_unlock(bug->write_mutex);
220     } else {
221         *writep = 0;
222     }
223 }
```

设置预缓冲的帧数。

```

225 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_set_pre_buffer_framecount(switch_media_bug_t *bug,
↪ uint32_t framecount)
226 {
227     bug->record_pre_buffer_max = framecount;
228
229     return SWITCH_STATUS_SUCCESS;
230 }

```

从 Media Bug 的缓冲区里读数据 (L340, L356)。核心 (`switch_core_io`) 会将数据读到 Media Bug 的缓冲区里。

```

232 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_read(switch_media_bug_t *bug, switch_frame_t
↪ *frame, switch_bool_t fill)
233 {
...
246     bytes = read_impl.decoded_bytes_per_packet;
247
...
338     if (do_read) {
340         frame->datalen = (uint32_t) switch_buffer_read(bug->raw_read_buffer, frame->data, do_read);
351     }
352
353     if (do_write) {
356         datalen = (uint32_t) switch_buffer_read(bug->raw_write_buffer, bug->data, do_write);
367     }

```

读到两侧（读和写侧分别是 从该 Channel 是听到的和发出去的数据）的数据后，进行下一步计算。其中 L370 为写侧，L371 为读侧。如果需要读立体声 (L376)，则将数据分别写入左声道和右声道。注意左声道和右声道的数据是交错存放的，每个 Sample 占两个字节 (`short` 型)。如果设置了 `SMBF_STEREO_SWAP` (L379)，还会将左右声道互换。

L = Left, R = Right

```

+-+--+--+--+--+
|L|R|L|R|L|R|...
+-+--+--+--+--+

```

```

369     tp = bug->tmp;
370     dp = (int16_t *) bug->data;
371     fp = (int16_t *) frame->data;
372     rlen = frame->datalen / 2;
373     wlen = datalen / 2;
374     blen = (uint32_t)(bytes / 2);
375
376     if (switch_test_flag(bug, SMBF_STEREO)) {
377         int16_t *left, *right;
378         size_t left_len, right_len;
379         if (switch_test_flag(bug, SMBF_STEREO_SWAP)) {
380             left = dp; /* write stream */
381             left_len = wlen;
382             right = fp; /* read stream */
383             right_len = rlen;
384         } else {
385             left = fp; /* read stream */
386             left_len = rlen;
387             right = dp; /* write stream */
388             right_len = wlen;
389         }
390         for (x = 0; x < blen; x++) {
391             if (x < left_len) {
392                 *(tp++) = *(left + x);
393             } else {
394                 *(tp++) = 0;
395             }
396             if (x < right_len) {
397                 *(tp++) = *(right + x);
398             } else {
399                 *(tp++) = 0;
400             }
401         }
402         memcpy(frame->data, bug->tmp, bytes * 2);

```

如果不是立体声的，则需要混音。将两个声道的数据混音，参见第 3 章。

```

403     } else {
404         for (x = 0; x < blen; x++) {
405             int32_t w = 0, r = 0, z = 0;
406
407             if (x < rlen) {
408                 r = (int32_t) * (fp + x);
409             }
410
411             if (x < wlen) {

```

```

412         w = (int32_t) * (dp + x);
413     }
414
415     z = w + r;
416
417     if (z > SWITCH_SMAX || z < SWITCH_SMIN) {
418         if (r) z += (r/2);
419         if (w) z += (w/2);
420     }
421
422     switch_normalize_to_16bit(z);
423
424     *(fp + x) = (int16_t) z;
425 }
426 }

```

在上面将数据处理完毕后，返回相应的帧。

```

428     frame->datalen = (uint32_t)bytes;
429     frame->samples = (uint32_t)(bytes / sizeof(int16_t) / read_impl.number_of_channels);
430     frame->rate = read_impl.actual_samples_per_second;
431     frame->codec = NULL;
432
433     if (switch_test_flag(bug, SMBF_STEREO)) {
434         frame->datalen *= 2;
435         frame->channels = 2;
436     } else {
437         frame->channels = read_impl.number_of_channels;
438     }
439
440     return SWITCH_STATUS_SUCCESS;
441 }

```

解析 SPY（偷听/偷看）相关的参数，视频相关的位置。

```

443 SWITCH_DECLARE(switch_vid_spy_fmt_t) switch_media_bug_parse_spy_fmt(const char *name)
444 {
445     if (zstr(name)) goto end;
446
447     if (!strcasecmp(name, "dual-crop")) {
448         return SPY_DUAL_CROP;
449     }
450
451     if (!strcasecmp(name, "lower-right-large")) {

```

```
452     return SPY_LOWER_RIGHT_LARGE;
453 }
454
455 end:
456
457     return SPY_LOWER_RIGHT_SMALL;
458 }
```

设置 SPY 相关的参数。

```
460 SWITCH_DECLARE(void) switch_media_bug_set_spy_fmt(switch_media_bug_t *bug, switch_vid_spy_fmt_t spy_fmt)
461 {
462     bug->spy_fmt = spy_fmt;
463 }
```

修改 SPY 数据，需要建立一个队列（L475）保存视频的帧。

```
465 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_patch_spy_frame(switch_media_bug_t *bug,
↪ switch_image_t *img, switch_rw_t rw)
466 {
467     switch_queue_t *spy_q = NULL;
468     int w = 0, h = 0;
469     switch_status_t status;
470     void *pop;
471     int i;
472
473     for (i = 0; i < 2; i++) {
474         if (!bug->spy_video_queue[i]) {
475             switch_queue_create(&bug->spy_video_queue[i], SWITCH_CORE_QUEUE_LEN,
↪ switch_core_session_get_pool(bug->session));
476         }
477     }
478
479     spy_q = bug->spy_video_queue[rw];
```

`switch_rw_t` 结构如下。

```
typedef enum {
    SWITCH_RW_READ,
    SWITCH_RW_WRITE
} switch_rw_t;
```

循环，找到队列中最后一帧，放到 `bug->spy_img[rw]` 中。

```

481 while(switch_queue_size(spy_q) > 0) {
482     if ((status = switch_queue_peek(spy_q, &pop)) == SWITCH_STATUS_SUCCESS) {
483         switch_img_free(&bug->spy_img[rw]);
484         if (!(bug->spy_img[rw] = (switch_image_t *) pop)) {
485             break;
486         }
487     }
488 }

```

如果需要剪裁（L496），则计算需要剪裁的大小并剪裁（L507，L514），缩放（L523 ~ L524），找到相应的位置（L529），并将图片贴到原来的图片上（L530），其它位置（L541 ~ L542）类似。

```

493 if (bug->spy_img[rw]) {
494
495     switch (bug->spy_fmt) {
496     case SPY_DUAL_CROP:
497         {
498             switch_image_t *spy_tmp = NULL;
499             switch_image_t *img_tmp = NULL;
500             switch_image_t *img_dup = NULL;
501             int x = 0, y = 0;
502             float aspect169 = (float)1920 / 1080;
503             switch_rgb_color_t bgcolor = { 0 };
504
505             if ((float)w/h == aspect169) {
506                 if ((float)bug->spy_img[rw]->d_w / bug->spy_img[rw]->d_h == aspect169) {
507                     spy_tmp = switch_img_copy_rect(bug->spy_img[rw], bug->spy_img[rw]->d_w / 4, 0,
508 ↪ bug->spy_img[rw]->d_w / 2, bug->spy_img[rw]->d_h);
509                 } else {
510                     switch_img_copy(bug->spy_img[rw], &spy_tmp);
511                 }
512             } else {
513                 if ((float)bug->spy_img[rw]->d_w / bug->spy_img[rw]->d_h == aspect169) {
514                     spy_tmp = switch_img_copy_rect(bug->spy_img[rw], bug->spy_img[rw]->d_w / 6, 0,
515 ↪ bug->spy_img[rw]->d_w / 4, bug->spy_img[rw]->d_h);
516                 } else {
517                     spy_tmp = switch_img_copy_rect(bug->spy_img[rw], bug->spy_img[rw]->d_w / 4, 0,
518 ↪ bug->spy_img[rw]->d_w / 2, bug->spy_img[rw]->d_h);
519                 }
520             }
521
522             switch_img_copy(img, &img_dup);

```

```

521         img_tmp = switch_img_copy_rect(img_dup, w / 4, 0, w / 2, h);
522
523         switch_img_fit(&spy_tmp, w / 2, h, SWITCH_FIT_SIZE);
524         switch_img_fit(&img_tmp, w / 2, h, SWITCH_FIT_SIZE);
525
526         switch_color_set_rgb(&bgcolor, "#000000");
527         switch_img_fill(img, 0, 0, img->d_w, img->d_h, &bgcolor);
528
529         switch_img_find_position(POS_CENTER_MID, w / 2, h, img_tmp->d_w, img_tmp->d_h, &x, &y);
530         switch_img_patch(img, img_tmp, x, y);
531
532         switch_img_find_position(POS_CENTER_MID, w / 2, h, spy_tmp->d_w, spy_tmp->d_h, &x, &y);
533         switch_img_patch(img, spy_tmp, x + w / 2, y);
534
535
536         switch_img_free(&img_tmp);
537         switch_img_free(&img_dup);
538         switch_img_free(&spy_tmp);
539     }
540     break;
541 case SPY_LOWER_RIGHT_SMALL:
542 case SPY_LOWER_RIGHT_LARGE:
543 default:

```

清空视频队列。

```

574 static int flush_video_queue(switch_queue_t *q, int min)
575 {
576     void *pop;
577
578     if (switch_queue_size(q) > min) {
579         while (switch_queue_trypop(q, &pop) == SWITCH_STATUS_SUCCESS) {
580             switch_image_t *img = (switch_image_t *) pop;
581             switch_img_free(&img);
582             if (min && switch_queue_size(q) <= min) {
583                 break;
584             }
585         }
586     }
587
588     return switch_queue_size(q);
589 }

```

视频 Bug 的线程，在一个独立的线程里执行。L637 初始化一个时钟。

如果 $\text{fps} = 25$ ，则 $1000/\text{fps} = 40$ ，即每 40 毫秒跳一次表。但是，由于视频的时间戳是 90000Hz，即将每秒钟分成 90000 份，所以每跳一次为 $40 * 90000 / 1000 = 3600$ ，即时间戳增量为 3600。

然后循环读取视频帧（L639）。

```

591 static void *SWITCH_THREAD_FUNC video_bug_thread(switch_thread_t *thread, void *obj)
592 {
...
637     switch_core_timer_init(&timer, "soft", 1000 / fps, (90000 / (1000 / fps)), NULL);
638
639     while (bug->ready) {

```

取得通话两侧（两个视频队列中）的图像（L663, L679），创建一个背景图像（L699）并填充背景色（L702），找到相应的位置（L712, L719）并将图像贴到背景上（L714, L721）。

```

656     if ((status = switch_queue_trypop(main_q, &pop)) == SWITCH_STATUS_SUCCESS) {
...
663         img = (switch_image_t *) pop;
672     }
673
674     if (other_q) {
677         if ((status = switch_queue_trypop(other_q, &other_pop)) == SWITCH_STATUS_SUCCESS) {
679             other_img = (switch_image_t *) other_pop;
688         }
689
699         if (!IMG) {
700             IMG = switch_img_alloc(NULL, SWITCH_IMG_FMT_I420, vw, vh, 1);
701             new_canvas = 1;
702             switch_img_fill(IMG, 0, 0, IMG->d_w, IMG->d_h, &color);
703         }
704     }
706
707     if (IMG) {
710         if (img && (new_canvas || new_main)) {
711             int x = 0, y = 0;
712             switch_img_find_position(POS_CENTER_MID, w, h, img->d_w, img->d_h, &x, &y);
713
714             switch_img_patch(IMG, img, x, y);
715         }
716
717         if (other_img && (new_canvas || new_other)) {
718             int x = 0, y = 0;
719             switch_img_find_position(POS_CENTER_MID, w, h, other_img->d_w, other_img->d_h, &x, &y);
720
721             switch_img_patch(IMG, other_img, w + x, y);

```

```

722     }
723 }

```

回调 Media Bug 函数 (L732)

```

725     if (IMG || img) {
...
731         if (bug->callback) {
732             if (bug->callback(bug, bug->user_data, SWITCH_ABC_TYPE_STREAM_VIDEO_PING) ==
↪ SWITCH_FALSE
733                 || (bug->stop_time && bug->stop_time <= switch_epoch_time_now(NULL))) {
734                 ok = SWITCH_FALSE;
735             }
736         }
769 }

```

向视频队列中 Push 视频帧，事先需要复制一份 (L780)。

```

771 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_push_spy_frame(switch_media_bug_t *bug, switch_frame_t *frame, switch_rw_
772 {
780     switch_img_copy(frame->img, &img);
781
782     if (img) {
783         switch_queue_push(bug->spy_video_queue[rw], img);
784         return SWITCH_STATUS_SUCCESS;
785     }
789 }

```

向 Session 上增加一个 Media Bug，增加成功会调用 [SWITCH_ABC_TYPE_INIT](#) 回调 (942)。

```

791 #define MAX_BUG_BUFFER 1024 * 512
792 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_add(switch_core_session_t *session,
793     const char *function,
794     const char *target,
795     switch_media_bug_callback_t callback,
796     void *user_data, time_t stop_time,
797     switch_media_bug_flag_t flags,
798     switch_media_bug_t **new_bug)
799 {
...
941     if (bug->callback) {

```

```

942     switch_bool_t result = bug->callback(bug, bug->user_data, SWITCH_ABC_TYPE_INIT);
949 }

```

Flush。

```

1007 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_flush_all(switch_core_session_t *session)

```

在发生转移的情况下，可以将 Media Bug 转移到别的 Channel 上。

```

1024 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_transfer_callback(switch_core_session_t
↪ *orig_session, switch_core_session_t *new_session,
1067 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_pop(switch_core_session_t *orig_session, const
↪ char *function, switch_media_bug_t **pop)

```

返回 Media Bug 的数量。

```

1093 SWITCH_DECLARE(uint32_t) switch_core_media_bug_count(switch_core_session_t *orig_session, const char
↪ *function)

```

在 Media Bug 上处理视频。

```

1111 SWITCH_DECLARE(uint32_t) switch_core_media_bug_patch_video(switch_core_session_t *orig_session,
↪ switch_frame_t *frame)
1112 {
1113     switch_media_bug_t *bp;
1114     uint32_t x = 0, ok = SWITCH_TRUE, prune = 0;
1115
1116     if (orig_session->bugs) {
1117         switch_thread_rwlock_rdlock(orig_session->bug_rwlock);
1118         for (bp = orig_session->bugs; bp; bp = bp->next) {
1119             if (!switch_test_flag(bp, SMBF_PRUNE) && !switch_test_flag(bp, SMBF_LOCK) && !strcmp(bp-
↪ >function, "patch:video")) {
1120                 if (bp->ready && frame->img && switch_test_flag(bp, SMBF_VIDEO_PATCH)) {
1121                     bp->video_ping_frame = frame;
1122                     if (bp->callback) {
1123                         if (bp->callback(bp, bp->user_data, SWITCH_ABC_TYPE_VIDEO_PATCH) ==
↪ SWITCH_FALSE
1124                             || (bp->stop_time && bp->stop_time <= switch_epoch_time_now(NULL))) {

```



```

1125             ok = SWITCH_FALSE;
1126         }
1127     }
1128     bp->video_ping_frame = NULL;
1129 }
...
1135     }
1136 }
...
1141 }
...
1144 }
```

在当前 Session 所有的 Media Bug 上执行回调函数 `cb`。

```

1146 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_exec_all(switch_core_session_t *orig_session,
1147                                                                 const char *function,
↪ switch_media_bug_exec_cb_t cb, void *user_data)
1148 {
...
1154     if (orig_session->bugs) {
1155         switch_thread_rwlock_wrlock(orig_session->bug_rwlock);
1156         for (bp = orig_session->bugs; bp; bp = bp->next) {
1157             if (!switch_test_flag(bp, SMBF_PRUNE) && !switch_test_flag(bp, SMBF_LOCK) && !strcmp(bp-
↪ >function, function)) {
1158                 cb(bp, user_data);
1159                 x++;
1160             }
1161         }
1162         switch_thread_rwlock_unlock(orig_session->bug_rwlock);
1163     }
1164
1165     return x ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
1166 }
```

遍历 Media Bug。

```

1168 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_enumerate(switch_core_session_t *session,
↪ switch_stream_handle_t *stream)
1169 {
1170     switch_media_bug_t *bp;
1171
1172     stream->write_function(stream, "<media-bugs>\n");
1173 }
```

```

1174     if (session->bugs) {
1175         switch_thread_rwlock_rdlock(session->bug_rwlock);
1176         for (bp = session->bugs; bp; bp = bp->next) {
1177             int thread_locked = (bp->thread_id && bp->thread_id == switch_thread_self());
1178             stream->write_function(stream,
1179                                   " <media-bug>\n"
1180                                   "  <function>%s</function>\n"
1181                                   "  <target>%s</target>\n"
1182                                   "  <thread-locked>%d</thread-locked>\n"
1183                                   " </media-bug>\n",
1184                                   bp->function, bp->target, thread_locked);
1185         }
1186         switch_thread_rwlock_unlock(session->bug_rwlock);
1187     }
1188
1189     stream->write_function(stream, "</media-bugs>\n");
1190
1191     return SWITCH_STATUS_SUCCESS;
1192 }

```

销毁、删除、释放资源等。

```

1195 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_remove_all_function(switch_core_session_t
↪ *session, const char *function)
1248 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_close(switch_media_bug_t **bug, switch_bool_t
↪ destroy)
1289 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_remove(switch_core_session_t *session,
↪ switch_media_bug_t **bug)
1343 SWITCH_DECLARE(uint32_t) switch_core_media_bug_prune(switch_core_session_t *session)
1385 SWITCH_DECLARE(switch_status_t) switch_core_media_bug_remove_callback(switch_core_session_t
↪ *session, switch_media_bug_callback_t callback)

```

4.21 switch_core_video.c

本章基于 Commit Hash [1681db4](#)。

关于图像格式可以参考第[2.1.14](#)节。

在 FreeSWITCH 代码中，一般用 **w** 与 **h** 分别表示图像的宽度和高度。

FreeSWITCH 中的视频图像格式基于 **libvpx** 库中的定义，FreeSWITCH 只是做了一个包装，使用了 **switch_** 命名空间。相关的定义在 **switch_vpx.h** 中，如下：

```

/*! \file switch_vpx.h
    \brief vpx resources

    The things powered by libvpx are renamed into the switch_ namespace to provide a cleaner
    look to things and helps me to document what parts of video I am using I'd like to take this
    opportunity to thank libvpx for all the awesome stuff it does and for making my life much easier.

*/

#ifdef SWITCH_VPX_H
#define SWITCH_VPX_H

#include <switch.h>
#include <switch_image.h>

SWITCH_BEGIN_EXTERN_C

#define SWITCH_IMG_FMT_PLANAR    VPX_IMG_FMT_PLANAR
#define SWITCH_IMG_FMT_UV_FLIP   VPX_IMG_FMT_UV_FLIP
#define SWITCH_IMG_FMT_HAS_ALPHA VPX_IMG_FMT_HAS_ALPHA

#define SWITCH_PLANE_PACKED VPX_PLANE_PACKED
#define SWITCH_PLANE_Y      VPX_PLANE_Y
#define SWITCH_PLANE_U      VPX_PLANE_U
#define SWITCH_PLANE_V      VPX_PLANE_V
#define SWITCH_PLANE_ALPHA  VPX_PLANE_ALPHA

#ifdef VPX_IMG_FMT_HIGH           /* not available in libvpx 1.3.0 (see commit hash e97aea28) */
#define VPX_IMG_FMT_HIGH         0x800 /**< Image uses 16bit framebuffer */
#endif

#define SWITCH_IMG_FMT_NONE      VPX_IMG_FMT_NONE
#define SWITCH_IMG_FMT_RGB24     VPX_IMG_FMT_RGB24
#define SWITCH_IMG_FMT_RGB32     VPX_IMG_FMT_RGB32
#define SWITCH_IMG_FMT_RGB565    VPX_IMG_FMT_RGB565
#define SWITCH_IMG_FMT_RGB555    VPX_IMG_FMT_RGB555
#define SWITCH_IMG_FMT_UYVY      VPX_IMG_FMT_UYVY
#define SWITCH_IMG_FMT_YUY2      VPX_IMG_FMT_YUY2
#define SWITCH_IMG_FMT_YVYU      VPX_IMG_FMT_YVYU
#define SWITCH_IMG_FMT_BGR24     VPX_IMG_FMT_BGR24
#define SWITCH_IMG_FMT_RGB32_LE  VPX_IMG_FMT_RGB32_LE
#define SWITCH_IMG_FMT_ARGB      VPX_IMG_FMT_ARGB
#define SWITCH_IMG_FMT_ARGB_LE   VPX_IMG_FMT_ARGB_LE
#define SWITCH_IMG_FMT_RGB565_LE VPX_IMG_FMT_RGB565_LE
#define SWITCH_IMG_FMT_RGB555_LE VPX_IMG_FMT_RGB555_LE
#define SWITCH_IMG_FMT_YV12      VPX_IMG_FMT_YV12
#define SWITCH_IMG_FMT_I420      VPX_IMG_FMT_I420

```

```
#define SWITCH_IMG_FMT_VPXV12    VPX_IMG_FMT_VPXV12
#define SWITCH_IMG_FMT_VPXI420  VPX_IMG_FMT_VPXI420
#define SWITCH_IMG_FMT_I422     VPX_IMG_FMT_I422
#define SWITCH_IMG_FMT_I444     VPX_IMG_FMT_I444
#define SWITCH_IMG_FMT_I440     VPX_IMG_FMT_I440
#define SWITCH_IMG_FMT_444A     VPX_IMG_FMT_444A
#define SWITCH_IMG_FMT_I42016   VPX_IMG_FMT_I42016
#define SWITCH_IMG_FMT_I42216   VPX_IMG_FMT_I42216
#define SWITCH_IMG_FMT_I44416   VPX_IMG_FMT_I44416
#define SWITCH_IMG_FMT_I44016   VPX_IMG_FMT_I44016
/* experimental */
#define SWITCH_IMG_FMT_GD       VPX_IMG_FMT_NONE

typedef vpx_img_fmt_t switch_img_fmt_t;

typedef vpx_image_t switch_image_t;

SWITCH_END_EXTERN_C
#endif
```

从一帧图像上获取 yuv 和 rgb 格式的像素。

```
52 static inline void switch_img_get_yuv_pixel(switch_image_t *img, switch_yuv_color_t *yuv, int x, int y);
55 static inline void switch_img_get_rgb_pixel(switch_image_t *img, switch_rgb_color_t *rgb, int x, int y);
```

rgb 和 yuv 像素互转。

```
64 static inline void switch_color_rgb2yuv(switch_rgb_color_t *rgb, switch_yuv_color_t *yuv);
73 static inline void switch_color_yuv2rgb(switch_yuv_color_t *yuv, switch_rgb_color_t *rgb);
```

位置，用于在图像上放置 Logo 等。

```
102 struct pos_el {
103     switch_img_position_t pos;
104     const char *name;
105 };
106
107
108 static struct pos_el POS_TABLE[] = {
109     {POS_LEFT_TOP, "left-top"},
110     {POS_LEFT_MID, "left-mid"},
```

```
111     {POS_LEFT_BOT, "left-bot"},
112     {POS_CENTER_TOP, "center-top"},
113     {POS_CENTER_MID, "center-mid"},
114     {POS_CENTER_BOT, "center-bot"},
115     {POS_RIGHT_TOP, "right-top"},
116     {POS_RIGHT_MID, "right-mid"},
117     {POS_RIGHT_BOT, "right-bot"},
118     {POS_NONE, "none"},
119     {POS_NONE, NULL}
120 };
```

解析字符串，返回对应的位置。

```
123 SWITCH_DECLARE(switch_img_position_t) parse_img_position(const char *name)
124 {
125     switch_img_position_t r = POS_NONE;
126     int i;
127
128     switch_assert(name);
129
130     for(i = 0; POS_TABLE[i].name; i++) {
131         if (!strcasecmp(POS_TABLE[i].name, name)) {
132             r = POS_TABLE[i].pos;
133             break;
134         }
135     }
136
137     return r;
138 }
```

图像大小适配。

```
141 struct fit_el {
142     switch_img_fit_t fit;
143     const char *name;
144 };
145
146
147 static struct fit_el IMG_FIT_TABLE[] = {
148     {SWITCH_FIT_SIZE, "fit-size"},
149     {SWITCH_FIT_SCALE, "fit-scale"},
150     {SWITCH_FIT_SIZE_AND_SCALE, "fit-size-and-scale"},
151     {SWITCH_FIT_NECESSARY, "fit-necessary"},
152     {SWITCH_FIT_NONE, NULL}
```

```
153 };

156 SWITCH_DECLARE(switch_img_fit_t) parse_img_fit(const char *name)
157 {
158     switch_img_fit_t r = SWITCH_FIT_SIZE;
159     int i;
160
161     switch_assert(name);
162
163     for(i = 0; IMG_FIT_TABLE[i].name; i++) {
164         if (!strcasecmp(IMG_FIT_TABLE[i].name, name)) {
165             r = IMG_FIT_TABLE[i].fit;
166             break;
167         }
168     }
169
170     return r;
171 }
```

FreeSWITCH 是否编译了视频支持。

```
173 SWITCH_DECLARE(switch_bool_t) switch_core_has_video(void)
174 {
175     #ifdef SWITCH_HAVE_VPX
176     #ifdef SWITCH_HAVE_YUV
177         return SWITCH_TRUE;
178     #else
179         return SWITCH_FALSE;
180     #endif
181     #else
182         return SWITCH_FALSE;
183     #endif
184 }
```

从内存中申请一个图像，返回图像指针，实际上就是调用了 `vpm_img_alloc`。其中，`align` 用于对齐，如果值为 0 或 1，则不使用 `pitch`，否则，可以使用 16 或 32 字节的 `pitch`。

如果在申请时传入一个旧图像的指针，则可以利用旧图像的缓冲区，如果新图像与旧图像大小不匹配，则可能会销毁掉原来的图像并返回新指针。

```
186 SWITCH_DECLARE(switch_image_t *)switch_img_alloc(switch_image_t *img,
187                                                    switch_img_fmt_t fmt,
188                                                    unsigned int d_w,
```

```
189             unsigned int d_h,  
190             unsigned int align)  
191 {  
192 #ifdef SWITCH_HAVE_VPX  
193     switch_image_t *r = NULL;  
218     r = (switch_image_t *)vpx_img_alloc((vpx_image_t *)img, (vpx_img_fmt_t)fmt, d_w, d_h, align);  
219     switch_assert(r);  
220     switch_assert(r->d_w == d_w);  
221     switch_assert(r->d_h == d_h);  
222  
223     return r;  
227 }
```

将一个已存在的图像数据缓冲区包括成一个图像。用完后缓冲区与图像需要独立地释放。

```
229 SWITCH_DECLARE(switch_image_t *)switch_img_wrap(switch_image_t *img,  
230             switch_img_fmt_t fmt,  
231             unsigned int d_w,  
232             unsigned int d_h,  
233             unsigned int align,  
234             unsigned char *img_data)  
235 {  
237     return (switch_image_t *)vpx_img_wrap((vpx_image_t *)img, (vpx_img_fmt_t)fmt, d_w, d_h, align,  
↪ img_data);  
241 }
```

在图像上划定一个矩形区域，区域外的图像将“不可见”。

```
243 SWITCH_DECLARE(int) switch_img_set_rect(switch_image_t *img,  
244             unsigned int x,  
245             unsigned int y,  
246             unsigned int w,  
247             unsigned int h)  
248 {  
250     return vpx_img_set_rect((vpx_image_t *)img, x, y, w, h);  
254 }
```

旋转。直接调用了 [libyuv](#) 库中的函数（L274）。

```
256 SWITCH_DECLARE(void) switch_img_rotate(switch_image_t **img, switch_image_rotation_mode_t mode)  
257 {
```

```

259     switch_image_t *tmp_img;
260
261     switch_assert(img);
262
263
264     if ((*img)->fmt != SWITCH_IMG_FMT_I420) return;
265
266     if (mode == SRM_90 || mode == SRM_270) {
267         tmp_img = switch_img_alloc(NULL, (*img)->fmt, (*img)->d_h, (*img)->d_w, 1);
268     } else {
269         tmp_img = switch_img_alloc(NULL, (*img)->fmt, (*img)->d_w, (*img)->d_h, 1);
270     }
271
272     switch_assert(tmp_img);
273
274     I420Rotate((*img)->planes[SWITCH_PLANE_Y], (*img)->stride[SWITCH_PLANE_Y],
275               (*img)->planes[SWITCH_PLANE_U], (*img)->stride[SWITCH_PLANE_U],
276               (*img)->planes[SWITCH_PLANE_V], (*img)->stride[SWITCH_PLANE_V],
277               tmp_img->planes[SWITCH_PLANE_Y], tmp_img->stride[SWITCH_PLANE_Y],
278               tmp_img->planes[SWITCH_PLANE_U], tmp_img->stride[SWITCH_PLANE_U],
279               tmp_img->planes[SWITCH_PLANE_V], tmp_img->stride[SWITCH_PLANE_V],
280               (*img)->d_w, (*img)->d_h, (int)mode);
281
282
283     switch_img_free(img);
284     *img = tmp_img;
285 }

```

释放图像。

```

289 SWITCH_DECLARE(void) switch_img_free(switch_image_t **img)
290 {
291     if (img && *img) {
292         if ((*img)->fmt == SWITCH_IMG_FMT_GD) {
293             #ifdef HAVE_LIBGD
294                 gdImageDestroy((gdImagePtr)(*img)->user_priv);
295             #endif
296         } else {
297             if ((int)(intptr_t)(*img)->user_priv != 1) {
298                 switch_safe_free((*img)->user_priv);
299             }
300         }
301     }
302     switch_assert((*img)->fmt <= SWITCH_IMG_FMT_I44016);
303     switch_assert((*img)->d_w <= 7860 && (*img)->d_w > 0);
304     switch_assert((*img)->d_h <= 4320 && (*img)->d_h > 0);
305     vpx_img_free((vpx_image_t *)*img);
306     *img = NULL;

```



```

307     }
308 #endif
309 }

```

将一个小的图像（img）“贴”到大图像（IMG）上，不处理 Alpha Channel。当然如果 img 比 IMG 大的话，会完全覆盖后面的图像，超出部分将剪裁。

如果两个图像均为 ARGB 格式（L324），则逐行（L331）复制每一个像素。

```

320 static void switch_img_patch_rgb_noalpha(switch_image_t *IMG, switch_image_t *img, int x, int y)
321 {
322     int i;
323
324     if (img->fmt == SWITCH_IMG_FMT_ARGB && IMG->fmt == SWITCH_IMG_FMT_ARGB) {
325         int max_w = MIN(img->d_w, IMG->d_w - abs(x));
326         int max_h = MIN(img->d_h, IMG->d_h - abs(y));
327         int j;
328         uint8_t alpha, alphadiff;
329         switch_rgb_color_t *rgb, *RGB;
330
331         for (i = 0; i < max_h; i++) {
332             for (j = 0; j < max_w; j++) {
333                 rgb = (switch_rgb_color_t *) (img->planes[SWITCH_PLANE_PACKED] + i * img->
334 ↪ stride[SWITCH_PLANE_PACKED] + j * 4);
335                 RGB = (switch_rgb_color_t *) (IMG->planes[SWITCH_PLANE_PACKED] + (y + i) * IMG->
336 ↪ stride[SWITCH_PLANE_PACKED] + (x + j) * 4);
337
338                 alpha = rgb->a;
339
340                 if (RGB->a != 0) {
341                     continue;
342                 }
343
344                 if (alpha == 255) {
345                     *RGB = *rgb;
346                 } else if (alpha != 0) {
347                     alphadiff = 255 - alpha;
348                     RGB->a = 255;
349                     RGB->r = ((RGB->r * alphadiff) + (rgb->r * alpha)) >> 8;
350                     RGB->g = ((RGB->g * alphadiff) + (rgb->g * alpha)) >> 8;
351                     RGB->b = ((RGB->b * alphadiff) + (rgb->b * alpha)) >> 8;
352                 }
353             }
354         }
355     }
356 }

```

将图像按 Alpha Channel 衰减。

```

357 SWITCH_DECLARE(void) switch_img_attenuate(switch_image_t *img)
358 {
360     if (img->fmt != SWITCH_IMG_FMT_ARGB) {
361         return;
362     }
363
364     if (img->user_priv) return;
365
366     img->user_priv = (void *) (intptr_t) 1;
367
368     ARGBAttenuate(img->planes[SWITCH_PLANE_PACKED], img->stride[SWITCH_PLANE_PACKED],
369                 img->planes[SWITCH_PLANE_PACKED], img->stride[SWITCH_PLANE_PACKED], img->d_w, img->d_h);
374 }

```

将 `img` 贴到 `IMG` 上，两个图像必须都是 ARGB 格式，并且要处理 Alpha Channel。在处理前先将 `img` 根据 Alpha Channel 处理一遍，然后再使用 `libyuv` 的 `ARGBBlendRow` 对每一行进行处理，确保能正确计算 Alpha Channel。

```






376 SWITCH_DECLARE(void) switch_img_patch_rgb(switch_image_t *IMG, switch_image_t *img, int x, int y,
    ↪ switch_bool_t noalpha)
377 {
378     #ifdef SWITCH_HAVE_YUV
379         int i;
380
381         if (noalpha) {
382             switch_img_patch_rgb_noalpha(IMG, img, x, y);
383             return;
384         }
385
386         if (img->fmt == SWITCH_IMG_FMT_ARGB && IMG->fmt == SWITCH_IMG_FMT_ARGB) {
387             uint8* src_argb0 = img->planes[SWITCH_PLANE_PACKED];
388             int src_stride_argb0 = img->stride[SWITCH_PLANE_PACKED];
389             uint8* src_argb1 = IMG->planes[SWITCH_PLANE_PACKED];
390             int src_stride_argb1 = IMG->stride[SWITCH_PLANE_PACKED];
391             uint8* dst_argb = IMG->planes[SWITCH_PLANE_PACKED];
392             int dst_stride_argb = IMG->stride[SWITCH_PLANE_PACKED];
393             int width = MIN(img->d_w, IMG->d_w - abs(x));
394             int height = MIN(img->d_h, IMG->d_h - abs(y));
395             void (*ARGBBlendRow)(const uint8* src_argb, const uint8* src_argb1, uint8* dst_argb, int width)
    ↪ = GetARGBBlend();
396
397             switch_img_attenuate(img);

```

```

398
399     // Coalesce rows. we have same size images, treat as a single row
400     if (src_stride_argb0 == width * 4 &&
401         src_stride_argb1 == width * 4 &&
402         x == 0 && y == 0) {
403         width *= height;
404         height = 1;
405         src_stride_argb0 = src_stride_argb1 = dst_stride_argb = 0;
406     }
407
408     if (y) {
409         src_argb1 += (y * IMG->d_w * 4);
410         dst_argb += (y * IMG->d_w * 4);
411     }
412     if (x) {
413         src_argb1 += (x * 4);
414         dst_argb += (x * 4);
415     }
416
417     for (i = 0; i < height; ++i) {
418         ARGBBlendRow(src_argb0, src_argb1, dst_argb, width);
419         src_argb0 += src_stride_argb0;
420         src_argb1 += src_stride_argb1;
421         dst_argb += dst_stride_argb;
422     }
423 }
424 #endif
425 }

```

将贴到上。必须为 I420 格式，如果为 ARGB 格式，则对每行（L466）每列（L467）获取每个像素（L468），然后将像素画到上（L452）。如果有 Alpha Channel，则先计算透明度（L457 ~ L460），然后再画像素（L462）。

```

427 SWITCH_DECLARE(void) switch_img_patch(switch_image_t *IMG, switch_image_t *img, int x, int y)
428 {
429     int i, len, max_h;
430     int xoff = 0, yoff = 0;
431
432     if (img->fmt == SWITCH_IMG_FMT_ARGB && IMG->fmt == SWITCH_IMG_FMT_ARGB) {
433         switch_img_patch_rgb(IMG, img, x, y, SWITCH_FALSE);
434         return;
435     }
436
437     switch_assert(IMG->fmt == SWITCH_IMG_FMT_I420);
438
439     if (img->fmt == SWITCH_IMG_FMT_ARGB) {

```

```

440     int max_w = MIN(img->d_w, IMG->d_w - abs(x));
441     int max_h = MIN(img->d_h, IMG->d_h - abs(y));
442     int j;
443     uint8_t alpha;
444     switch_rgb_color_t *rgb;
445
446     for (i = 0; i < max_h; i++) {
447         for (j = 0; j < max_w; j++) {
448             rgb = (switch_rgb_color_t *) (img->planes[SWITCH_PLANE_PACKED] + i * img-
↳ >stride[SWITCH_PLANE_PACKED] + j * 4);
449             alpha = rgb->a;
450
451             if (alpha == 255) {
452                 switch_img_draw_pixel(IMG, x + j, y + i, rgb);
453             } else if (alpha != 0) {
454                 switch_rgb_color_t RGB = { 0 };
455
456                 switch_img_get_rgb_pixel(IMG, &RGB, x + j, y + i);
457                 RGB.a = 255;
458                 RGB.r = ((RGB.r * (255 - alpha)) >> 8) + ((rgb->r * alpha) >> 8);
459                 RGB.g = ((RGB.g * (255 - alpha)) >> 8) + ((rgb->g * alpha) >> 8);
460                 RGB.b = ((RGB.b * (255 - alpha)) >> 8) + ((rgb->b * alpha) >> 8);
461
462                 switch_img_draw_pixel(IMG, x + j, y + i, &RGB);
463             }
464         }
465     }
466
467     return;
...
498 }

```

如果 `img` 为 I420 格式，则首先保证 `img` 大小和位置处于 `IMG` 内并且边界为偶数（L500 ~ L516）。然后复制 Y 平面的每一行（L518 ~ L520），然后复制 U 和 V 平面的每一行（L526 ~ 529）。

其中，`img->planes[SWITCH_PLANE_Y]`（或 U / V），表示每个平台的起始位置，`img->stride[SWITCH_PLANE_Y]`（或 U/V）表示每一行占用的内存长度，而每一行有效数据的长度，Y 平台为 `img->d_w`，U 和 V 平台为 `img->d_w / 2`。

```

499
500     if (x < 0) {
501         xoff = -x;
502         x = 0;
503     }
504
505     if (y < 0) {

```

```

506     yoff = -y;
507     y = 0;
508 }
509
510 max_h = MIN(y + img->d_h - yoff, IMG->d_h);
511 len = MIN(img->d_w - xoff, IMG->d_w - x);
512
513
514 if (x & 0x1) { x++; len--; }
515 if (y & 0x1) y++;
516 if (len <= 0) return;
517
518 for (i = y; i < max_h; i++) {
519     memcpy(IMG->planes[SWITCH_PLANE_Y] + IMG->stride[SWITCH_PLANE_Y] * i + x, img-
↪ >planes[SWITCH_PLANE_Y] + img->stride[SWITCH_PLANE_Y] * (i - y + yoff) + xoff, len);
520 }
521
522 if ((len & 1) && (x + len) < img->d_w - 1) len++;
523
524 len /= 2;
525
526 for (i = y; i < max_h; i += 2) {
527     memcpy(IMG->planes[SWITCH_PLANE_U] + IMG->stride[SWITCH_PLANE_U] * (i / 2) + x / 2, img-
↪ >planes[SWITCH_PLANE_U] + img->stride[SWITCH_PLANE_U] * ((i - y + yoff) / 2) + xoff / 2, len);
528     memcpy(IMG->planes[SWITCH_PLANE_V] + IMG->stride[SWITCH_PLANE_V] * (i / 2) + x / 2, img-
↪ >planes[SWITCH_PLANE_V] + img->stride[SWITCH_PLANE_V] * ((i - y + yoff) / 2) + xoff / 2, len);
529 }
530 }

```

只贴图像某个矩形区域以内。

```

532 SWITCH_DECLARE(void) switch_img_patch_rect(switch_image_t *IMG, int X, int Y, switch_image_t *img,
↪ uint32_t x, uint32_t y, uint32_t w, uint32_t h)

```

复制图像，从 `img` 复制到 `new_img`。仅支持两种图像格式（L578）。如果 `new_img` 已存在，则使用它的存储空间，但如果格式或大小不匹配（L581 ~ L582），则销毁 `new_img` 后面会重新创建一个新的。如果 `new_img` 不存在（L588），则申请一个新的（L589）。最后调用 `libyuv` 库里的图像复制函数复制图像。

```

570 SWITCH_DECLARE(void) switch_img_copy(switch_image_t *img, switch_image_t **new_img)
571 {
573     switch_img_fmt_t new_fmt = img->fmt;
574

```

```
575     switch_assert(img);
576     switch_assert(new_img);
577
578     if (img->fmt != SWITCH_IMG_FMT_I420 && img->fmt != SWITCH_IMG_FMT_ARGB) return;
579
580     if (*new_img) {
581         if ((*new_img)->fmt != SWITCH_IMG_FMT_I420 && (*new_img)->fmt != SWITCH_IMG_FMT_ARGB) return;
582         if (img->d_w != (*new_img)->d_w || img->d_h != (*new_img)->d_h ) {
583             new_fmt = (*new_img)->fmt;
584             switch_img_free(new_img);
585         }
586     }
587
588     if (*new_img == NULL) {
589         *new_img = switch_img_alloc(NULL, new_fmt, img->d_w, img->d_h, 1);
590     }
591
592     switch_assert(*new_img);
593
594     if (img->fmt == SWITCH_IMG_FMT_I420) {
595         if (new_fmt == SWITCH_IMG_FMT_I420) {
596             I420Copy(img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
597                     img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
598                     img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
599                     (*new_img)->planes[SWITCH_PLANE_Y], (*new_img)->stride[SWITCH_PLANE_Y],
600                     (*new_img)->planes[SWITCH_PLANE_U], (*new_img)->stride[SWITCH_PLANE_U],
601                     (*new_img)->planes[SWITCH_PLANE_V], (*new_img)->stride[SWITCH_PLANE_V],
602                     img->d_w, img->d_h);
603         } else if (new_fmt == SWITCH_IMG_FMT_ARGB) {
604             I420ToARGB(img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
605                       img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
606                       img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
607                       (*new_img)->planes[SWITCH_PLANE_PACKED], (*new_img)->stride[SWITCH_PLANE_PACKED],
608                       img->d_w, img->d_h);
609         }
610     } else if (img->fmt == SWITCH_IMG_FMT_ARGB) {
611         if (new_fmt == SWITCH_IMG_FMT_ARGB) {
612             ARGBCopy(img->planes[SWITCH_PLANE_PACKED], img->stride[SWITCH_PLANE_PACKED],
613                     (*new_img)->planes[SWITCH_PLANE_PACKED], (*new_img)->stride[SWITCH_PLANE_PACKED],
614                     img->d_w, img->d_h);
615         } else if (new_fmt == SWITCH_IMG_FMT_I420) {
616             ARGBToI420(img->planes[SWITCH_PLANE_PACKED], img->stride[SWITCH_PLANE_PACKED],
617                       (*new_img)->planes[SWITCH_PLANE_Y], (*new_img)->stride[SWITCH_PLANE_Y],
618                       (*new_img)->planes[SWITCH_PLANE_U], (*new_img)->stride[SWITCH_PLANE_U],
619                       (*new_img)->planes[SWITCH_PLANE_V], (*new_img)->stride[SWITCH_PLANE_V],
620                       img->d_w, img->d_h);
621         }
622     }
626 }
```

旋转并复制。

```
629 SWITCH_DECLARE(void) switch_img_rotate_copy(switch_image_t *img, switch_image_t **new_img,
↪ switch_image_rotation_mode_t mode)
630 {
631     switch_assert(img);
632     switch_assert(new_img);
633
634 #ifdef SWITCH_HAVE_YUV
635     if (img->fmt != SWITCH_IMG_FMT_I420) abort();
636
637     if (*new_img != NULL) {
638         if (img->fmt != (*new_img)->fmt || img->d_w != (*new_img)->d_w || img->d_h != (*new_img)->d_h) {
639             switch_img_free(new_img);
640         }
641     }
642
643     if (*new_img == NULL) {
644         if (mode == SRM_90 || mode == SRM_270) {
645             *new_img = switch_img_alloc(NULL, img->fmt, img->d_h, img->d_w, 1);
646         } else {
647             *new_img = switch_img_alloc(NULL, img->fmt, img->d_w, img->d_h, 1);
648         }
649     }
650
651     switch_assert(*new_img);
652
653
654     I420Rotate(img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
655               img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
656               img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
657               (*new_img)->planes[SWITCH_PLANE_Y], (*new_img)->stride[SWITCH_PLANE_Y],
658               (*new_img)->planes[SWITCH_PLANE_U], (*new_img)->stride[SWITCH_PLANE_U],
659               (*new_img)->planes[SWITCH_PLANE_V], (*new_img)->stride[SWITCH_PLANE_V],
660               img->d_w, img->d_h, (int)mode);
661 #else
662     return;
663 #endif
664 }
```

仅复制图像的一块矩形区域。先画好矩形（L688），再复制（L689）。

```

666 SWITCH_DECLARE(switch_image_t *) switch_img_copy_rect(switch_image_t *img, uint32_t x, uint32_t y,
↳   uint32_t w, uint32_t h)
667 {
...
688     if (!switch_img_set_rect(tmp, x, y, w, h)) {
689         switch_img_copy(tmp, &new_img);
690     }
698 }

```

计算两个像素之间的差异。用于“抠图”的情况，比较两个像素的相似程度。基本上是套公式计算。

```

910 static inline int switch_color_distance(switch_rgb_color_t *c1, switch_rgb_color_t *c2)
911 {
912     int cr, cg, cb;
913     int cr2, cg2, cb2;
914     double a, b;
915     int aa, bb, r;
916
917
918     cr = c1->r - c2->r;
919     cg = c1->g - c2->g;
920     cb = c1->b - c2->b;
921
922     if (!cr && !cg && !cb) return 0;
923
924     cr2 = c1->r/2 - c2->r/2;
925     cg2 = c1->g/2 - c2->g/2;
926     cb2 = c1->b/2 - c2->b/2;
927
928     a = ((2*cr*cr) + (4*cg*cg) + (3*cb*cb));
929     b = ((2*cr2*cr2) + (4*cg2*cg2) + (3*cb2*cb2));
930
931     aa = (int)a;
932     bb = (int)b*25;
933
934     r = (((bb*4)+(aa))/9)/100;
935
936     return r;
937
938 }

```

计算多个像素间的差异。

```

940 static inline int switch_color_distance_multi(switch_rgb_color_t *c1, switch_rgb_color_t *clist, int
↳ count, uint32_t *thresholds)
941 {
942     int x = 0, hits = 0;
943
944     for (x = 0; x < count; x++) {
945         int distance = switch_color_distance(c1, &clist[x]);
946
947         if (distance <= thresholds[x]) {
948             hits++;
949             break;
950         }
951     }
952
953     return hits;
954 }

```

Chromakey 为在“抠图”时计算相关的色度。

```

957 struct switch_chromakey_s {
958     switch_image_t *cache_img;
959     switch_rgb_color_t mask[CHROMAKEY_MAX_MASK];
960     uint32_t thresholds[CHROMAKEY_MAX_MASK];
961     int mask_len;
962     switch_shade_t autocolour;
963     uint32_t dft_thresh;
964     uint32_t dft_thresh_squared;
965
966     uint32_t rr;
967     uint32_t gg;
968     uint32_t bb;
969     uint32_t color_count;
970
971     switch_rgb_color_t auto_color;
972     int no_cache;
973     int frames_read;
974 };

```

将字符串转换成 Shade 值。

```

976 SWITCH_DECLARE(switch_shade_t) switch_chromakey_str2shade(switch_chromakey_t *ck, const char
↳ *shade_name)
977 {
978     switch_shade_t shade = SWITCH_SHADE_NONE;

```

```
979
980     if (!strcasecmp(shade_name, "red")) {
981         shade = SWITCH_SHADE_RED;
982     } else if (!strcasecmp(shade_name, "green")) {
983         shade = SWITCH_SHADE_GREEN;
984     } else if (!strcasecmp(shade_name, "blue")) {
985         shade = SWITCH_SHADE_BLUE;
986     } else if (!strcasecmp(shade_name, "auto")) {
987         shade = SWITCH_SHADE_AUTO;
988     }
989
990     return shade;
991 }
```

设置默认阈值。

```
993 SWITCH_DECLARE(void) switch_chromakey_set_default_threshold(switch_chromakey_t *ck, uint32_t threshold)
994 {
995     int i;
996
997     ck->dft_thresh = threshold;
998     ck->dft_thresh_squared = threshold * threshold;
999
1000     for (i = 0; i < ck->mask_len; i++) {
1001         if (!ck->thresholds[i]) ck->thresholds[i] = ck->dft_thresh_squared;
1002     }
1003 }
```

清除颜色。

```
1004
1005 SWITCH_DECLARE(switch_status_t) switch_chromakey_clear_colors(switch_chromakey_t *ck)
1006 {
1007     switch_assert(ck);
1008
1009     ck->autocolor = SWITCH_SHADE_NONE;
1010     ck->mask_len = 0;
1011     memset(ck->mask, 0, sizeof(ck->mask[0]) * CHROMAKEY_MAX_MASK);
1012     memset(ck->thresholds, 0, sizeof(ck->thresholds[0]) * CHROMAKEY_MAX_MASK);
1013     ck->no_cache = 1;
1014
1015     return SWITCH_STATUS_SUCCESS;
1016 }
```

自动计算颜色。

```

1018 SWITCH_DECLARE(switch_status_t) switch_chromakey_autocolor(switch_chromakey_t *ck, switch_shade_t
↪ autocolor, uint32_t threshold)
1019 {
1020     switch_assert(ck);
1021
1022     switch_chromakey_clear_colors(ck);
1023     ck->autocolor = autocolor;
1024     ck->dft_thresh = threshold;
1025     ck->dft_thresh_squared = threshold * threshold;
1026     switch_img_free(&ck->cache_img);
1027     ck->no_cache = 90;
1028     memset(&ck->auto_color, 0, sizeof(ck->auto_color));
1029
1030     return SWITCH_STATUS_SUCCESS;
1031 }

```

增加颜色，可以在 Chromakey 中支持多种颜色去除。

```

1033 SWITCH_DECLARE(switch_status_t) switch_chromakey_add_color(switch_chromakey_t *ck,
↪ switch_rgb_color_t *color, uint32_t threshold)
1034 {
1035     switch_assert(ck);
1036
1037     if (ck->mask_len == CHROMAKEY_MAX_MASK) {
1038         return SWITCH_STATUS_FALSE;
1039     }
1040
1041     ck->mask[ck->mask_len] = *color;
1042     ck->thresholds[ck->mask_len] = threshold * threshold;
1043     ck->mask_len++;
1044
1045     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Adding color %d:%d:%d #%.2x%.2x%.2x\n",
1046 ck->auto_color.r, ck->auto_color.g, ck->auto_color.b, ck->auto_color.r, ck-
↪ >auto_color.g, ck->auto_color.b);
1047
1048     return SWITCH_STATUS_SUCCESS;
1049 }

```

销毁、创建 Chromakey。

```
1051 SWITCH_DECLARE(switch_status_t) switch_chromakey_destroy(switch_chromakey_t **ckP)
1068 SWITCH_DECLARE(switch_status_t) switch_chromakey_create(switch_chromakey_t **ckP)
```

缓存。

```
1082 SWITCH_DECLARE(switch_image_t *) switch_chromakey_cache_image(switch_chromakey_t *ck)
1083 {
1084     switch_assert(ck);
1085
1086     return ck->cache_img;
1087 }
```

获取颜色最大值。

```
1089 static inline int get_max(switch_rgb_color_t *c1)
1090 {
1091     if (c1->r > c1->g && c1->r > c1->b) {
1092         return 1;
1093     }
1094
1095     if (c1->g > c1->r && c1->g > c1->b) {
1096         return 2;
1097     }
1098
1099     if (c1->b > c1->r && c1->b > c1->g) {
1100         return 3;
1101     }
1102
1103     return 0;
1104 }
```

比较。

```
1106 static inline int switch_color_dom_cmp(switch_rgb_color_t *c1, switch_rgb_color_t *c2)
1107 {
1108
1109     int c1_max = get_max(c1);
1110     int c2_max = get_max(c2);
1111
1112     if (c1_max && c1_max == c2_max) return 1;
1113 }
```

```
1114     return 0;
1115
1116 }
```

比较颜色的距离。

```
1118 static inline int switch_color_distance_literal(switch_rgb_color_t *c1, switch_rgb_color_t *c2, int
↪ distance)
1119 {
1120     int r = abs(c1->r - c2->r);
1121     int g = abs(c1->g - c2->g);
1122     int b = abs(c1->b - c2->b);
1123
1124     if (r < distance && g < distance && b < distance) return 1;
1125
1126     return 0;
1127 }
```

简单粗暴的方法计算颜色距离（节省 CPU）。

```
1129 static inline int switch_color_distance_cheap(switch_rgb_color_t *c1, switch_rgb_color_t *c2)
1130 {
1131     int r = abs(c1->r - c2->r);
1132     int g = abs(c1->g - c2->g);
1133     int b = abs(c1->b - c2->b);
1134
1135     if (r < 5 && g < 5 && b < 5) return 0;
1136
1137     return (3*r) + (4*g) + (3*b);
1138 }
```

```
1140 static inline void get_dom(switch_shade_t autocolor, switch_rgb_color_t *color, int *domP, int *aP,
↪ int *bP)
1141 {
1142     int dom, a, b;
1143
1144     switch(autocolor) {
1145     case SWITCH_SHADE_RED:
1146         dom = color->r;
1147         a = color->g;
1148         b = color->b;
```

```

1149         break;
1150     case SWITCH_SHADE_GREEN:
1151         dom = color->g;
1152         a = color->r;
1153         b = color->b;
1154         break;
1155     case SWITCH_SHADE_BLUE:
1156         dom = color->b;
1157         a = color->r;
1158         b = color->g;
1159         break;
1160     default:
1161         dom = 0;
1162         a = 0;
1163         b = 0;
1164         break;
1165     }
1166
1167     *domP = dom;
1168     *aP = a;
1169     *bP = b;
1170
1171 }

```

处理 Chromakey，抠图。如果像素颜色大于一定阈值（L1380），则抠除掉（将 Alpha Channel 设为透明，L1387）。

```

1173 SWITCH_DECLARE(void) switch_chromakey_process(switch_chromakey_t *ck, switch_image_t *img)
1174 SWITCH_DECLARE(void) switch_img_chromakey(switch_image_t *img, switch_rgb_color_t *mask, int
↳ threshold)
1175 {
1176     switch_rgb_color_t *pixel, *last_pixel = NULL;
1177     int last_threshold = 0;
1178     switch_assert(img);
1179
1180     if (img->fmt != SWITCH_IMG_FMT_ARGB) return;
1181
1182     pixel = (switch_rgb_color_t *)img->planes[SWITCH_PLANE_PACKED];
1183
1184     for (; pixel < ((switch_rgb_color_t *)img->planes[SWITCH_PLANE_PACKED] + img->d_w * img->d_h);
↳ pixel++) {
1185         int threshold = 0;
1186
1187         if (last_pixel && (*(uint32_t *)pixel & 0xFFFFFF) == (*(uint32_t *)last_pixel & 0xFFFFFF)) {
1188             threshold = last_threshold;
1189         } else {

```

```

1380         threshold = switch_color_distance(pixel, mask);
1381     }
1382
1383     last_threshold = threshold;
1384     last_pixel = pixel;
1385
1386     if (threshold) {
1387         pixel->a = 0;
1388     }
1389 }
1390
1391 return;
1392 }
```

往 YUV 图像上画像素。

```

1394 static inline void switch_img_draw_pixel(switch_image_t *img, int x, int y, switch_rgb_color_t
↪ *color)
1395 {
1396     switch_yuv_color_t yuv = {0};
1397
1398     if (x < 0 || y < 0 || x >= img->d_w || y >= img->d_h) return;
1399
1400     if (img->fmt == SWITCH_IMG_FMT_I420) {
1401         switch_color_rgb2yuv(color, &yuv);
1402
1403         img->planes[SWITCH_PLANE_Y][y * img->stride[SWITCH_PLANE_Y] + x] = yuv.y;
1404
1405         if (((x & 0x1) == 0) && ((y & 0x1) == 0)) { // only draw on even position
1406             img->planes[SWITCH_PLANE_U][y / 2 * img->stride[SWITCH_PLANE_U] + x / 2] = yuv.u;
1407             img->planes[SWITCH_PLANE_V][y / 2 * img->stride[SWITCH_PLANE_V] + x / 2] = yuv.v;
1408         }
1409     } else if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1410         *((switch_rgb_color_t *)img->planes[SWITCH_PLANE_PACKED] + img->d_w * y + x) = *color;
1411     }
1412 }
1413 }
```

用颜色填充图像。

```

1416 SWITCH_DECLARE(void) switch_img_fill_noalpha(switch_image_t *img, int x, int y, int w, int h,
↪ switch_rgb_color_t *color)
1417 {
1418     int i;
1419
1420     for (i = 0; i < w; i++)
```

```

1421     if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1422         int max_w = img->d_w;
1423         int max_h = img->d_h;
1424         int j;
1425         switch_rgb_color_t *rgb;
1426
1427         for (i = 0; i < max_h; i++) {
1428             for (j = 0; j < max_w; j++) {
1429                 rgb = (switch_rgb_color_t *) (img->planes[SWITCH_PLANE_PACKED] + i * img-
↵ >stride[SWITCH_PLANE_PACKED] + j * 4);
1430
1431                 if (rgb->a != 0) {
1432                     continue;
1433                 }
1434
1435                 *rgb = *color;
1436             }
1437         }
1438     }
1441 }

```

将图像像素的值控制在一定范围内。实际上每字节范围达不到 0 ~ 255。

```

1443 SWITCH_DECLARE(void) switch_img_8bit(switch_image_t *img)
1444 {
1445     #ifdef SWITCH_HAVE_YUV
1446         int i;
1447
1448         if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1449             int max_w = img->d_w;
1450             int max_h = img->d_h;
1451             int j;
1452             switch_rgb_color_t *rgb;
1453             uint32_t *bytes;
1454
1455             for (i = 0; i < max_h; i++) {
1456                 for (j = 0; j < max_w; j++) {
1457                     rgb = (switch_rgb_color_t *) (img->planes[SWITCH_PLANE_PACKED] + i * img-
↵ >stride[SWITCH_PLANE_PACKED] + j * 4);
1458                     //if (rgb);
1459
1460
1461                     if (!rgb->a) continue;;
1462
1463                     //rgb->r = rgb->r & 0xE0, rgb->g = rgb->g & 0xE0, rgb->b = rgb->b & 0xC0;
1464                     bytes = (uint32_t *) rgb;

```



```

1465
1466     #if SWITCH_BYTE_ORDER == __BIG_ENDIAN
1467         *bytes = *bytes & 0xE0E0C0FF;
1468     #else
1469         *bytes = *bytes & 0xFFC0E0E0;
1470     #endif
1471
1472     }
1473 }
1474 } else if (img->fmt == SWITCH_IMG_FMT_I420) {
1475     switch_image_t *tmp_img = switch_img_alloc(NULL, SWITCH_IMG_FMT_ARGB, img->d_w, img->d_h,
↪ 1);
1476
1477     I420ToARGB(img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
1478               img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
1479               img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
1480               tmp_img->planes[SWITCH_PLANE_PACKED], tmp_img->stride[SWITCH_PLANE_PACKED],
1481               img->d_w, img->d_h);
1482
1483     switch_img_8bit(tmp_img);
1484
1485     ARGBToI420(tmp_img->planes[SWITCH_PLANE_PACKED], tmp_img->stride[SWITCH_PLANE_PACKED],
1486               img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
1487               img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
1488               img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
1489               tmp_img->d_w, tmp_img->d_h);
1490
1491     switch_img_free(&tmp_img);
1492 }
1493 #endif
1494 }

```

将图像处理成棕褐色。如果图像为 ARGB 格式，则直接调用 `libyuv` 提供的函数（L1502），如果图像为 YUV 格式，则只需要处理 U 和 V 色彩平面，无须处理 Y 平面。

```

1496 SWITCH_DECLARE(void) switch_img_sepia(switch_image_t *img, int x, int y, int w, int h)
1497 {
1498     if (x < 0 || y < 0 || x >= img->d_w || y >= img->d_h) return;
1499
1500
1501     if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1502         ARGBSepia(img->planes[SWITCH_PLANE_PACKED], img->stride[SWITCH_PLANE_PACKED], x, y, w, h);
1503     } else if (img->fmt == SWITCH_IMG_FMT_I420) {
1504         ...
1505
1506         for (i = y; i < max_h; i += 2) {
1507             memset(img->planes[SWITCH_PLANE_U] + img->stride[SWITCH_PLANE_U] * (i / 2) + x / 2, 108,
↪ len);

```

```

1519         memset(img->planes[SWITCH_PLANE_V] + img->stride[SWITCH_PLANE_V] * (i / 2) + x / 2, 137,
↪ len);
1520     }
1521 }
1523 }
```

灰度处理，与上面的函数类似。

```

1525 SWITCH_DECLARE(void) switch_img_gray(switch_image_t *img, int x, int y, int w, int h)
1526 {
1527     #ifdef SWITCH_HAVE_YUV
1528
1529         if (x < 0 || y < 0 || x >= img->d_w || y >= img->d_h) return;
1530
1531         if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1532             ARGBGray(img->planes[SWITCH_PLANE_PACKED], img->stride[SWITCH_PLANE_PACKED], x, y, w, h);
1533         } else if (img->fmt == SWITCH_IMG_FMT_I420) {
1534             ...
1547             for (i = y; i < max_h; i += 2) {
1548                 memset(img->planes[SWITCH_PLANE_U] + img->stride[SWITCH_PLANE_U] * (i / 2) + x / 2, 128,
↪ len);
1549                 memset(img->planes[SWITCH_PLANE_V] + img->stride[SWITCH_PLANE_V] * (i / 2) + x / 2, 128,
↪ len);
1550             }
1551         }
1552     #endif
1553 }
```

用颜色填充图像。

```

1555 SWITCH_DECLARE(void) switch_img_fill(switch_image_t *img, int x, int y, int w, int h,
↪ switch_rgb_color_t *color)
1556 {
1557     #ifdef SWITCH_HAVE_YUV
1558         int len, i, max_h;
1559         switch_yuv_color_t yuv_color;
1560
1561         if (x < 0 || y < 0 || x >= img->d_w || y >= img->d_h) return;
1562
1563         if (img->fmt == SWITCH_IMG_FMT_I420) {
1564             switch_color_rgb2yuv(color, &yuv_color);
1565
1566             max_h = MIN(y + h, img->d_h);
1567             len = MIN(w, img->d_w - x);
```

```

1568
1569     if (x & 1) { x++; len--; }
1570     if (y & 1) y++;
1571     if (len <= 0) return;
1572
1573     for (i = y; i < max_h; i++) {
1574         memset(img->planes[SWITCH_PLANE_Y] + img->stride[SWITCH_PLANE_Y] * i + x, yuv_color.y,
↪ len);
1575     }
1576
1577     if ((len & 1) && (x + len) < img->d_w - 1) len++;
1578
1579     len /= 2;
1580
1581     for (i = y; i < max_h; i += 2) {
1582         memset(img->planes[SWITCH_PLANE_U] + img->stride[SWITCH_PLANE_U] * (i / 2) + x / 2,
↪ yuv_color.u, len);
1583         memset(img->planes[SWITCH_PLANE_V] + img->stride[SWITCH_PLANE_V] * (i / 2) + x / 2,
↪ yuv_color.v, len);
1584     }
1585     } else if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1586         for (i = 0; i < img->d_w; i++) {
1587             *((switch_rgb_color_t *)img->planes[SWITCH_PLANE_PACKED] + i) = *color;
1588         }
1589
1590         for (i = 1; i < img->d_h; i++) {
1591             memcpy( img->planes[SWITCH_PLANE_PACKED] + i * img->d_w * 4,
1592                   img->planes[SWITCH_PLANE_PACKED], img->d_w * 4);
1593         }
1594     }
1595 #endif
1596 }

```

套公式，获取 YUV 和 RGB 像素。

```

1599 static inline void switch_img_get_yuv_pixel(switch_image_t *img, switch_yuv_color_t *yuv, int x, int
↪ y)
1600 {
1601     // switch_assert(img->fmt == SWITCH_IMG_FMT_I420);
1602     if (x < 0 || y < 0 || x >= img->d_w || y >= img->d_h) return;
1603
1604     yuv->y = *(img->planes[SWITCH_PLANE_Y] + img->stride[SWITCH_PLANE_Y] * y + x);
1605     yuv->u = *(img->planes[SWITCH_PLANE_U] + img->stride[SWITCH_PLANE_U] * (y / 2) + x / 2);
1606     yuv->v = *(img->planes[SWITCH_PLANE_V] + img->stride[SWITCH_PLANE_V] * (y / 2) + x / 2);
1607 }

```

```

1610 static inline void switch_img_get_rgb_pixel(switch_image_t *img, switch_rgb_color_t *rgb, int x, int
↪ y)
1611 {
1613     if (x < 0 || y < 0 || x >= img->d_w || y >= img->d_h) return;
1614
1615     if (img->fmt == SWITCH_IMG_FMT_I420) {
1616         switch_yuv_color_t yuv;
1617
1618         switch_img_get_yuv_pixel(img, &yuv, x, y);
1619         switch_color_yuv2rgb(&yuv, rgb);
1620     } else if (img->fmt == SWITCH_IMG_FMT_ARGB) {
1621         *rgb = *((switch_rgb_color_t *)img->planes[SWITCH_PLANE_PACKED] + img->d_w * y + x);
1622     }
1624 }

```

将 **img** 覆盖到 **IMG** 上。根据百分比 (**percent**) 处理透明度。

```

1626 SWITCH_DECLARE(void) switch_img_overlay(switch_image_t *IMG, switch_image_t *img, int x, int y,
↪ uint8_t percent)
1627 {
1628     int i, j, len, max_h;
1629     switch_rgb_color_t RGB = {0}, rgb = {0}, c = {0};
1630     int xoff = 0, yoff = 0;
1631     uint8_t alpha = (int8_t)((255 * percent) / 100);
1632
1633
1634     switch_assert(IMG->fmt == SWITCH_IMG_FMT_I420);
1635
1636     if (x < 0) {
1637         xoff = -x;
1638         x = 0;
1639     }
1640
1641     if (y < 0) {
1642         yoff = -y;
1643         y = 0;
1644     }
1645
1646     max_h = MIN(y + img->d_h - yoff, IMG->d_h);
1647     len = MIN(img->d_w - xoff, IMG->d_w - x);
1648
1649     if (x & 1) { x++; len--; }
1650     if (y & 1) y++;
1651     if (len <= 0) return;
1652
1653     for (i = y; i < max_h; i++) {

```

```

1654     for (j = 0; j < len; j++) {
1655         switch_img_get_rgb_pixel(IMG, &RGB, x + j, i);
1656         switch_img_get_rgb_pixel(img, &rgb, j + xoff, i - y + yoff);
1657
1658         if (rgb.a > 0) {
1659             c.r = ((RGB.r * (255 - alpha)) >> 8) + ((rgb.r * alpha) >> 8);
1660             c.g = ((RGB.g * (255 - alpha)) >> 8) + ((rgb.g * alpha) >> 8);
1661             c.b = ((RGB.b * (255 - alpha)) >> 8) + ((rgb.b * alpha) >> 8);
1662         } else {
1663             c.r = RGB.r;
1664             c.g = RGB.g;
1665             c.b = RGB.b;
1666         }
1667
1668         switch_img_draw_pixel(IMG, x + j, i, &c);
1669     }
1670 }
1671 }

```

在没有字体支持的情况下，用点阵方式画简单的英文字母和数字、符号等。

```

1690 static void scv_tag(void *buffer, int w, int x, int y, uint8_t n)
1691 {
1692     int i = 0, j=0;
1693     uint8_t *p = buffer;
1694
1695     if (n > 13) return;
1696
1697     for(i=0; i<8; i++) {
1698         for (j=0; j<16; j++) {
1699             *(p + (y + j) * w + (x + i)) = (scv_art[n][j] & 0x80 >> i) ? 0xFF : 0x00;
1700         }
1701     }
1702 }
1703
1704 SWITCH_DECLARE(void) switch_img_add_text(void *buffer, int w, int x, int y, char *s)
1705 {
1706     while (*s) {
1707         int index;
1708
1709         if (x > w - 8) break;
1710
1711         switch (*s) {
1712             case '.': index = 10; break;
1713             case ':': index = 11; break;
1714             case '-': index = 12; break;

```

```
1715         case ' ': index = 13; break;
1716         default:
1717             index = *s - 0x30;
1718     }
1719
1720     scv_tag(buffer, w, x, y, index);
1721     x += 8;
1722     s++;
1723 }
1724 }
```

将颜色字符串转换成 RGB 值。

```
1726 SWITCH_DECLARE(void) switch_color_set_rgb(switch_rgb_color_t *color, const char *str)
1727 {
1728     if (zstr(str)) return;
1729
1730     if ((*str) == '#' && strlen(str) == 7) {
1731         unsigned int r, g, b;
1732         sscanf(str, "%02x%02x%02x", &r, &g, &b);
1733         color->r = r;
1734         color->g = g;
1735         color->b = b;
1736     } else {
1737         if (!strcmp(str, "red")) {
1738             color->r = 255;
1739             color->g = 0;
1740             color->b = 0;
1741         } else if (!strcmp(str, "green")) {
1742             color->r = 0;
1743             color->g = 255;
1744             color->b = 0;
1745         } else if (!strcmp(str, "blue")) {
1746             color->r = 0;
1747             color->g = 0;
1748             color->b = 255;
1749         }
1750     }
1751
1752     color->a = 255;
1753 }
```

颜色转换公式。

```

1756 static inline void switch_color_rgb2yuv(switch_rgb_color_t *rgb, switch_yuv_color_t *yuv)
1757 {
1758
1759     yuv->y = ( ( 66 * rgb->r + 129 * rgb->g + 25 * rgb->b + 128) >> 8) + 16;
1760     yuv->u = ( ( -38 * rgb->r - 74 * rgb->g + 112 * rgb->b + 128) >> 8) + 128;
1761     yuv->v = ( ( 112 * rgb->r - 94 * rgb->g - 18 * rgb->b + 128) >> 8) + 128;
1762 }
1763 #define CLAMP(val) MAX(0, MIN(val, 255))
1764 static inline void switch_color_yuv2rgb(switch_yuv_color_t *yuv, switch_rgb_color_t *rgb)
1765 {
1766 #if 0
1767     int C = yuv->y - 16;
1768     int D = yuv->u - 128;
1769     int E = yuv->v - 128;
1770
1771     rgb->r = CLAMP((298 * C + 409 * E + 128) >> 8);
1772     rgb->g = CLAMP((298 * C - 100 * D - 208 * E + 128) >> 8);
1773     rgb->b = CLAMP((298 * C + 516 * D + 128) >> 8);
1774 #endif
1775
1776     rgb->a = 255;
1777     rgb->r = CLAMP( yuv->y + ((22457 * (yuv->v-128)) >> 14));
1778     rgb->g = CLAMP((yuv->y - ((715 * (yuv->v-128)) >> 10) - ((5532 * (yuv->u-128)) >> 14)));
1779     rgb->b = CLAMP((yuv->y + ((28384 * (yuv->u-128)) >> 14)));
1780 }

```

设置 YUV 颜色。

```

1791 SWITCH_DECLARE(void) switch_color_set_yuv(switch_yuv_color_t *color, const char *str)
1792 {
1793     switch_rgb_color_t rgb = { 0 };
1794
1795     switch_color_set_rgb(&rgb, str);
1796     switch_color_rgb2yuv(&rgb, color);
1797 }

```

FreeSWITCH 字库相关处理，需要 [libfreetype](#) 库。

```

1801 #if SWITCH_HAVE_FREETYPE
1802 #include <ft2build.h>
1803 #include FT_FREETYPE_H
1804 #include FT_GLYPH_H
1805 #endif

```

```
1806
1807     #define MAX_GRADIENT 8
```

定义文本 Handle。

```
1809     struct switch_img_txt_handle_s {
1810     #if SWITCH_HAVE_FREETYPE
1811         FT_Library library;
1812         FT_Face face;
1813     #endif
1814         char *font_family;
1815         double angle;
1816         uint16_t font_size;
1817         switch_rgb_color_t color;
1818         switch_rgb_color_t bgcolor;
1819         switch_image_t *img;
1820         switch_memory_pool_t *pool;
1821         int free_pool;
1822         switch_rgb_color_t gradient_table[MAX_GRADIENT];
1823         switch_bool_t use_bgcolor;
1824     };
```

颜色梯度表。

```
1826     static void init_gradient_table(switch_img_txt_handle_t *handle)
1827     {
1828         int i;
1829         switch_rgb_color_t *color;
1830
1831         switch_rgb_color_t *c1 = &handle->bgcolor;
1832         switch_rgb_color_t *c2 = &handle->color;
1833
1834         for (i = 0; i < MAX_GRADIENT; i++) {
1835             color = &handle->gradient_table[i];
1836             color->r = c1->r + (c2->r - c1->r) * i / MAX_GRADIENT;
1837             color->g = c1->g + (c2->g - c1->g) * i / MAX_GRADIENT;
1838             color->b = c1->b + (c2->b - c1->b) * i / MAX_GRADIENT;
1839             color->a = 255;
1840         }
1841     }
```

创建文本 Handle，需要字体文件路径、前景色、背景色、字体大小等参数。如果传入内存池

指针为空，则会自己创建一个（L1851），然后在内存池上申请一个 **handle** 指针（L1854）。初始化 **FreeType**（L1857）及颜色梯度表（L1892）。

```

1843 SWITCH_DECLARE(switch_status_t) switch_img_txt_handle_create(switch_img_txt_handle_t **handleP,
↪  const char *font_family,
1844     const char *font_color, const char *bgcolor, uint16_t font_size, double angle,
↪  switch_memory_pool_t *pool)
1845 {
1846     int free_pool = 0;
1847     switch_img_txt_handle_t *new_handle;
1848
1849     if (!pool) {
1850         free_pool = 1;
1851         switch_core_new_memory_pool(&pool);
1852     }
1853
1854     new_handle = switch_core_alloc(pool, sizeof(*new_handle));
1855
1856     if (FT_Init_FreeType(&new_handle->library)) {
1857         return SWITCH_STATUS_FALSE;
1858     }
1859
1860     new_handle->pool = pool;
1861     new_handle->free_pool = free_pool;
1862     new_handle->font_size = font_size;
1863     new_handle->angle = angle;
1864
1865     switch_color_set_rgb(&new_handle->color, font_color);
1866     switch_color_set_rgb(&new_handle->bgcolor, bgcolor);
1867
1868     init_gradient_table(new_handle);
1869
1870     *handleP = new_handle;
1871
1872     return SWITCH_STATUS_SUCCESS;
1873 }

```

销毁文本 Handle。

```

1900 SWITCH_DECLARE(void) switch_img_txt_handle_destroy(switch_img_txt_handle_t **handleP)

```

往 **img** 上写字，位置坐标为 (**x**, **y**)。根据字体类型，如果是 **MONO** 格式，则可以直接根据字体描述画像素（L1941 ~ L1957）。如果是灰度模式（L1938），则循环进行灰度处理并写像素（L1966 ~ L1990）。

```

1928     #if SWITCH_HAVE_FREETYPE
1929     static void draw_bitmap(switch_img_txt_handle_t *handle, switch_image_t *img, FT_Bitmap* bitmap,
↪   FT_Int x, FT_Int y)
1930     {
1931         FT_Int i, j, p, q;
1932         FT_Int x_max = x + bitmap->width;
1933         FT_Int y_max = y + bitmap->rows;
1934
1935         if (bitmap->width == 0) return;
1936
1937         switch (bitmap->pixel_mode) {
1938             case FT_PIXEL_MODE_GRAY: // it should always be GRAY since we use FT_LOAD_RENDER?
1939                 break;
1940             case FT_PIXEL_MODE_NONE:
1941             case FT_PIXEL_MODE_MONO:
1942                 {
1943                     for ( j = y, q = 0; j < y_max; j++, q++ ) {
1944                         for ( i = x, p = 0; i < x_max; i++, p++ ) {
1945                             uint8_t byte;
1946                             int linesize = ((bitmap->width - 1) / 8 + 1) * 8;
1947
1948                             if ( i < 0 || j < 0 || i >= img->d_w || j >= img->d_h) continue;
1949
1950                             byte = bitmap->buffer[(q * linesize + p) / 8];
1951                             if ((byte >> (7 - (p % 8))) & 0x1) {
1952                                 switch_img_draw_pixel(img, i, j, &handle->color);
1953                             }
1954                         }
1955                     }
1956                     return;
1957                 }
1958             case FT_PIXEL_MODE_GRAY2:
1959             case FT_PIXEL_MODE_GRAY4:
1960             case FT_PIXEL_MODE_LCD:
1961             case FT_PIXEL_MODE_LCD_V:
1962                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "unsupported pixel mode %d\n",
↪   bitmap->pixel_mode);
1963                 return;
1964         }
1965
1966         for ( i = x, p = 0; i < x_max; i++, p++ ) {
1967             for ( j = y, q = 0; j < y_max; j++, q++ ) {
1968                 int gradient = bitmap->buffer[q * bitmap->width + p];
1969                 if ( i < 0 || j < 0 || i >= img->d_w || j >= img->d_h) continue;
1970
1971                 if (handle->use_bgcolor) {
1972                     switch_img_draw_pixel(img, i, j, &handle->gradient_table[gradient * MAX_GRADIENT /
↪   256]);

```

```

1973         } else {
1974             switch_rgb_color_t rgb_color = {0};
1975             switch_rgb_color_t c;
1976             switch_img_get_rgb_pixel(img, &rgb_color, i, j);
1977
1978             if (rgb_color.a > 0) {
1979                 c.a = rgb_color.a * gradient / 255;
1980                 c.r = ((rgb_color.r * (255 - gradient)) >> 8) + ((handle->color.r * gradient) >>
↵ 8);
1981                 c.g = ((rgb_color.g * (255 - gradient)) >> 8) + ((handle->color.g * gradient) >>
↵ 8);
1982                 c.b = ((rgb_color.b * (255 - gradient)) >> 8) + ((handle->color.b * gradient) >>
↵ 8);
1983             } else {
1984                 c.a = gradient;
1985                 c.r = handle->color.r;
1986                 c.g = handle->color.g;
1987                 c.b = handle->color.b;
1988             }
1989
1990             switch_img_draw_pixel(img, i, j, &c);
1991         }
1992     }
1993 }
1994 }

```

往图像上写字。创建字体 Face (L2051)，设置字体大小，初始化梯度表 (L2063)。设置字体矩阵 (L2067) 以及画笔的初始位置 (L2072 ~ L2073)。

```

1998 SWITCH_DECLARE(uint32_t) switch_img_txt_handle_render(switch_img_txt_handle_t *handle,
↵ switch_image_t *img,
1999     int x, int y, const char *text,
2000     const char *font_family, const char *font_color,
2001     const char *bgcolor, uint16_t font_size, double angle)
2002 {
2051     error = FT_New_Face(handle->library, font_family, 0, &face); /* create face object */
2057     error = FT_Set_Char_Size(face, 64 * font_size, 0, 96, 0); /* set character size */
2059
2060     slot = face->glyph;
2061
2062     if (handle->use_bgcolor && slot->bitmap.pixel_mode != FT_PIXEL_MODE_MONO) {
2063         init_gradient_table(handle);
2064     }
2065
2066     /* set up matrix */
2067     matrix.xx = (FT_Fixed)( cos( angle ) * 0x10000L );

```

```
2068     matrix.xy = (FT_Fixed)(-sin( angle ) * 0x10000L );
2069     matrix.yx = (FT_Fixed)( sin( angle ) * 0x10000L );
2070     matrix.yy = (FT_Fixed)( cos( angle ) * 0x10000L );
2071
2072     pen.x = x;
2073     pen.y = y;
```

循环，对于每一个 UTF-8 字符（L2076），如果遇到换行符（L2078），则将画笔移到下一行。从字库中获取字形（L2088），移动画笔（L2092），将字形画到 `img` 上，然后移动画笔（L2110 ~ L2111）并继续处理下一字符。

```
2075     while(*(text + index)) {
2076         ch = switch_u8_get_char((char *)text, &index);
2077
2078         if (ch == '\n') {
2079             pen.x = x;
2080             pen.y += (font_size + font_size / 4);
2081             continue;
2082         }
2083
2084         /* set transformation */
2085         FT_Set_Transform(face, &matrix, &pen);
2086
2087         /* load glyph image into the slot (erase previous one) */
2088         error = FT_Load_Char(face, ch, FT_LOAD_RENDER);
2089
2090         if (error) continue;
2091
2092         this_x = pen.x + slot->bitmap_left;
2093
2094         if (img) {
2095             /* now, draw to our target surface (convert position) */
2096             draw_bitmap(handle, img, &slot->bitmap, this_x, pen.y - slot->bitmap_top + font_size);
2097         }
2098
2099         if (last_x) {
2100             space = this_x - last_x;
2101         } else {
2102             space = 0;
2103         }
2104
2105         last_x = this_x;
2106
2107         width += space;
2108
2109         /* increment pen position */
```

```

2110         pen.x += slot->advance.x >> 6;
2111         pen.y += slot->advance.y >> 6;
2112     }
2113
2114     ret = width + slot->bitmap.width * 5;
2115
2116     FT_Done_Face(face);
2117
2118     return ret;
2122 }
```

根据文字内容生成并返回一张图片。创建一个文本 Handle (L2185)，并调用上面讲的函数 (L2187)，由于传入的图像为 NULL (2188)，不会实际产生图像，但会返回文本渲染后的宽度。然后根据宽度生成一张图像 (L2210)，在图像上再写一遍 (L2210) 相应的文字内容。

```

2124 SWITCH_DECLARE(switch_image_t *) switch_img_write_text_img(int w, int h, switch_bool_t full, const
↳ char *text)
2125 {
...
2185     switch_img_txt_handle_create(&txthandle, font_face, fg, bg, font_size, 0, NULL);
2186
2187     pre_width = switch_img_txt_handle_render(txthandle,
2188                                             NULL,
2189                                             font_size / 2, font_size / 2,
2190                                             txt, NULL, fg, bg, 0, 0);
2191
...
2210     txtimg = switch_img_alloc(NULL, SWITCH_IMG_FMT_ARGB, width, height, 1);
...
2228     switch_img_txt_handle_render(txthandle,
2229                                 txtimg,
2230                                 x, y,
2231                                 txt, NULL, fg, bg, 0, 0);
...
2239     return txtimg;
2240 }
```

往 IMG 上贴 img，但空下一个矩形区域不贴（如露出 Logo）。

```

+-----+-----+
| IMG . . . |img..| rect |
| . . . . . |.....+-----+
| . . . . . |.....|
```

```
| . . . . . |.....|
| . . . . . +-----|
| . . . . . . . . . |
| . . . . . . . . . |
+-----+
```

```
2247 SWITCH_DECLARE(void) switch_img_patch_hole(switch_image_t *IMG, switch_image_t *img, int x, int y,
↪ switch_image_rect_t *rect)
```

打开 PNG 图片，需要 `libpng >= 1.6`。设置 PNG 打开的格式为 `PNG_FORMAT_ARGB` (L2324)，与 FreeSWITCH 的 `ARGB` 格式对应。申请缓冲区存放图像 (L2326)，并读取 PNG (L2329)。

其中，`switch_png_t` 的定义如下 (opaque 为实际 `libpng` 打开的格式)：

```
typedef struct switch_png_s {
    switch_png_opaque_t *pvt;
    int w;
    int h;
} switch_png_t;
```

```
2310 SWITCH_DECLARE(switch_status_t) switch_png_open(switch_png_t **pngP, const char *file_name)
2311 {
2324     use_png->pvt->png.format = PNG_FORMAT_ARGB;
2325
2326     use_png->pvt->buffer = malloc(PNG_IMAGE_SIZE(use_png->pvt->png));
2328
2329     if (!png_image_finish_read(&use_png->pvt->png, NULL/*background*/, use_png->pvt->buffer, 0,
↪ NULL)) {
2330         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Error read PNG %s\n", file_name);
2331         switch_goto_status(SWITCH_STATUS_FALSE, end);
2332     }
2333
2334
2335     use_png->w = use_png->pvt->png.width;
2336     use_png->h = use_png->pvt->png.height;
2337
2338     end:
2339
2340     if (status == SWITCH_STATUS_SUCCESS) {
2341         *pngP = use_png;
2342     } else {
2343         switch_png_free(&use_png);
```

```
2344         *pngP = NULL;
2345     }
2346
2347     return status;
2348 }
```

销毁 PNG。

```
2350 SWITCH_DECLARE(void) switch_png_free(switch_png_t **pngP)
```

往 FreeSWITCH 格式的图像 `img` 上贴 PNG。循环取得每个像素 (L2375)，然后画到 `img` 上 (2378)。

```
2365 SWITCH_DECLARE(switch_status_t) switch_png_patch_img(switch_png_t *use_png, switch_image_t *img, int
↪ x, int y)
2366 {
2367     switch_status_t status = SWITCH_STATUS_SUCCESS;
2368     switch_rgb_color_t *rgb_color;
2369     int i, j;
2370
2371     switch_assert(use_png);
2372
2373     for (i = 0; i < use_png->pvt->png.height; i++) {
2374         for (j = 0; j < use_png->pvt->png.width; j++) {
2375             rgb_color = (switch_rgb_color_t *)use_png->pvt->buffer + i * use_png->pvt->png.width +
↪ j;
2376
2377             if (rgb_color->a) { // todo, mux alpha with the underlying pixel
2378                 switch_img_draw_pixel(img, x + j, y + i, rgb_color);
2379             }
2380         }
2381     }
2382
2383     return status;
2384 }
```

直接读 PNG，并产生一个 FreeSWITCH 格式的 `img`。首先打开并读出 `png_image` 格式 (L2418)，然后根据 `img` 的格式确定 PNG 在内存中的格式 (L2423 ~ L2430)，申请一个 `buffer` (L2436)，并将图像读到 `buffer` 指向的内存位置 (L2439)。

```
2408  #ifdef PNG_SIMPLIFIED_READ_SUPPORTED /* available from libpng 1.6.0 */
2409
2410  SWITCH_DECLARE(switch_image_t *) switch_img_read_png(const char* file_name, switch_img_fmt_t
↳ img_fmt)
2411  {
2412      png_image png = { 0 };
2413      png_bytep buffer = NULL;
2414      switch_image_t *img = NULL;
2415
2416      png.version = PNG_IMAGE_VERSION;
2417
2418      if (!png_image_begin_read_from_file(&png, file_name)) {
2419          switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Error open png: %s\n", file_name);
2420          goto err;
2421      }
2422
2423      if (img_fmt == SWITCH_IMG_FMT_I420) {
2424          png.format = PNG_FORMAT_RGB;
2425      } else if (img_fmt == SWITCH_IMG_FMT_ARGB) {
2426          #if SWITCH_BYTE_ORDER == __BIG_ENDIAN
2427              png.format = PNG_FORMAT_ARGB;
2428          #else
2429              png.format = PNG_FORMAT_BGRA;
2430          #endif
2431      } else {
2432          switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Unsupported image format: %x\n",
↳ img_fmt);
2433          goto err;
2434      }
2435
2436      buffer = malloc(PNG_IMAGE_SIZE(png));
2437      switch_assert(buffer);
2438
2439      if (!png_image_finish_read(&png, NULL/*background*/, buffer, 0, NULL)) {
2440          switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Error read png: %s\n", file_name);
2441          goto err;
2442      }
2443
2444      if (png.width > SWITCH_IMG_MAX_WIDTH || png.height > SWITCH_IMG_MAX_HEIGHT) {
2445          switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "PNG is too large! %dx%d\n",
↳ png.width, png.height);
2446          goto err;
2447      }
```

初始化与 PNG 相同大小的一个 **img**，然后使用 **libyuv** 提供的函数复制图像数据（L2453，L2459）。


```

2449     img = switch_img_alloc(NULL, img_fmt, png.width, png.height, 1);
2450     switch_assert(img);
2451
2452     if (img_fmt == SWITCH_IMG_FMT_I420) {
2453         RAWToI420(buffer, png.width * 3,
2454             img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
2455             img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
2456             img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
2457             png.width, png.height);
2458     } else if (img_fmt == SWITCH_IMG_FMT_ARGB){
2459         ARGBCopy(buffer, png.width * 4,
2460             img->planes[SWITCH_PLANE_PACKED], png.width * 4,
2461             png.width, png.height);
2462     }

```

在 `libpng < 1.6.0` 的情况下，需要更多的代码读 PNG，具体请参阅相关链接，就不多解释了。

```

    #else /* libpng < 1.6.0 */
2471
2472     // ref: most are out-dated, man libpng :)
2473     // http://zarb.org/~gc/html/libpng.html
2474     // http://www.libpng.org/pub/png/book/toc.html
2475     // http://www.vias.org/pngguide/chapter01_03_02.html
2476     // http://www.libpng.org/pub/png/libpng-1.2.5-manual.html
2477     // ftp://ftp.oreilly.com/examples/9781565920583/CDROM/SOFTWARE/SOURCE/LIBPNG/EXAMPLE.C
2478
2479     SWITCH_DECLARE(switch_image_t *) switch_img_read_png(const char* file_name, switch_img_fmt_t
↪ img_fmt)

```

将 `img` 写入 PNG 文件。需要先图像的数据转换成 RGB 格式（L2748），如果 `img` 已经是 `ARGB` 格式就不用转换了，只是需要注意字节序（L2754 ~ L2758）。将 `buffer` 写入文件（L2766）。

```

2735     #ifdef PNG_SIMPLIFIED_WRITE_SUPPORTED /* available from libpng 1.6.0 */
2736
2737     SWITCH_DECLARE(switch_status_t) switch_img_write_png(switch_image_t *img, char* file_name)
2738     {
2739         png_image png = { 0 };
2740         png_bytep buffer = NULL;
2741         switch_status_t status = SWITCH_STATUS_SUCCESS;
2742
2743         if (img->fmt == SWITCH_IMG_FMT_I420) {
2744             png.format = PNG_FORMAT_RGB;
2745             buffer = malloc(img->d_w * img->d_h * 3);

```

```

2746         switch_assert(buffer);
2747
2748         I420ToRAW(img->planes[SWITCH_PLANE_Y], img->stride[SWITCH_PLANE_Y],
2749                 img->planes[SWITCH_PLANE_U], img->stride[SWITCH_PLANE_U],
2750                 img->planes[SWITCH_PLANE_V], img->stride[SWITCH_PLANE_V],
2751                 buffer, img->d_w * 3,
2752                 img->d_w, img->d_h);
2753     } else if (img->fmt == SWITCH_IMG_FMT_ARGB) {
2754 #if SWITCH_BYTE_ORDER == __BIG_ENDIAN
2755         png.format = PNG_FORMAT_ARGB;
2756 #else
2757         png.format = PNG_FORMAT_BGRA;
2758 #endif
2759         buffer = img->planes[SWITCH_PLANE_PACKED];
2760     }
2761
2762     png.version = PNG_IMAGE_VERSION;
2763     png.width = img->d_w;
2764     png.height = img->d_h;
2765
2766     if (!png_image_write_to_file(&png, file_name, 0, buffer, 0, NULL)) {

```

在 `libpng < 1.6.0` 的情况下，需要更多代码，不赘述。

```

2778     #else
2779
2780     SWITCH_DECLARE(switch_status_t) switch_img_write_png(switch_image_t *img, char* file_name)

```

缩放当前图像为 `width x height` 并产生新图像，如果比例不匹配，则以背景色（`color`）填充。

```

2909     SWITCH_DECLARE(switch_status_t) switch_img_letterbox(switch_image_t *img, switch_image_t **imgP, int
↵ width, int height, const char *color)
2910     {
2911         int img_w = 0, img_h = 0;
2912         double screen_aspect = 0, img_aspect = 0;
2913         int x_pos = 0;
2914         int y_pos = 0;
2915         switch_image_t *IMG = NULL, *scale_img = NULL;
2916         switch_rgb_color_t bgcolor = { 0 };
2917
2918         switch_assert(imgP);
2919         *imgP = NULL;
2920
2921         if (img->d_w == width && img->d_h == height) {

```

```

2922         switch_img_copy(img, imgP);
2923         return SWITCH_STATUS_SUCCESS;
2924     }
2925
2926     IMG = switch_img_alloc(NULL, SWITCH_IMG_FMT_I420, width, height, 1);
2927     switch_color_set_rgb(&bgcolor, color);
2928     switch_img_fill(IMG, 0, 0, IMG->d_w, IMG->d_h, &bgcolor);
2929
2930     img_w = IMG->d_w;
2931     img_h = IMG->d_h;
2932
2933     screen_aspect = (double) IMG->d_w / IMG->d_h;
2934     img_aspect = (double) img->d_w / img->d_h;
2935
2936
2937     if (screen_aspect > img_aspect) {
2938         img_w = img_aspect * IMG->d_h;
2939         x_pos = (IMG->d_w - img_w) / 2;
2940     } else if (screen_aspect < img_aspect) {
2941         img_h = IMG->d_h / img_aspect;
2942         y_pos = (IMG->d_h - img_h) / 2;
2943     }
2944
2945     switch_img_scale(img, &scale_img, img_w, img_h);
2946     switch_img_patch(IMG, scale_img, x_pos, y_pos);
2947     switch_img_free(&scale_img);
2948
2949     *imgP = IMG;
2950
2951     return SWITCH_STATUS_SUCCESS;
2952 }

```

缩放图像适应width x height决定的大小。

```

2954 SWITCH_DECLARE(switch_status_t) switch_img_fit(switch_image_t **srcP, int width, int height,
↪ switch_img_fit_t fit)
2955 {
2956     switch_image_t *src, *tmp = NULL;
2957     int new_w = 0, new_h = 0;
2958
2959     switch_assert(srcP);
2960     switch_assert(width && height);
2961
2962     src = *srcP;
2963
2964     if (!src || (src->d_w == width && src->d_h == height)) {

```

```
2965         return SWITCH_STATUS_SUCCESS;
2966     }
2967
2968     if (fit == SWITCH_FIT_NECESSARY && src->d_w <= width && src->d_h < height) {
2969         return SWITCH_STATUS_SUCCESS;
2970     }
2971
2972     if (fit == SWITCH_FIT_SCALE) {
2973         switch_img_scale(src, &tmp, width, height);
2974         switch_img_free(&src);
2975         *srcP = tmp;
2976         return SWITCH_STATUS_SUCCESS;
2977     }
2978
2979     new_w = src->d_w;
2980     new_h = src->d_h;
2981
2982     if (src->d_w < width && src->d_h < height) {
2983         float rw = (float)new_w / width;
2984         float rh = (float)new_h / height;
2985
2986         if (rw > rh) {
2987             new_h = (int)((float)new_h / rw);
2988             new_w = width;
2989         } else {
2990             new_w = (int)((float)new_w / rh);
2991             new_h = height;
2992         }
2993     } else {
2994         while(new_w > width || new_h > height) {
2995             if (new_w > width) {
2996                 double m = (double) width / new_w;
2997                 new_w = width;
2998                 new_h = (int) (new_h * m);
2999             } else {
3000                 double m = (double) height / new_h;
3001                 new_h = height;
3002                 new_w = (int) (new_w * m);
3003             }
3004         }
3005     }
3006
3007     if (new_w && new_h) {
3008         if (switch_img_scale(src, &tmp, new_w, new_h) == SWITCH_STATUS_SUCCESS) {
3009             switch_img_free(&src);
3010             *srcP = tmp;
3011
3012             if (fit == SWITCH_FIT_SIZE_AND_SCALE) {
3013                 src = *srcP;
```

```

3014         tmp = NULL;
3015         switch_img_scale(src, &tmp, width, height);
3016         switch_img_free(&src);
3017         *srcP = tmp;
3018     }
3019
3020     return SWITCH_STATUS_SUCCESS;
3021 }
3022 }
3023
3024 return SWITCH_STATUS_FALSE;
3025 }

```

FreeSWITCH 支持的 FOURCC⁶。FOURCC 是 Four Character Code 的缩写，即用 4 个字母表示图像的格式。下面代码为 FreeSWITCH 内部格式（即 libvpx 提供的格式）与 FOURCC 的对应关系，FreeSWITCH 仅支持一部分 FOURCC。

```

3028 static inline uint32_t switch_img_fmt2fourcc(switch_img_fmt_t fmt)
3029 {
3030     uint32_t fourcc;
3031
3032     switch(fmt) {
3033         case SWITCH_IMG_FMT_NONE:         fourcc = (uint32_t)FOURCC_ANY ; break;
3034         case SWITCH_IMG_FMT_RGB24:        fourcc = (uint32_t)FOURCC_24BG; break;
3035         case SWITCH_IMG_FMT_RGB32:        fourcc = (uint32_t)FOURCC_ANY ; break;
3036         case SWITCH_IMG_FMT_RGB565:       fourcc = (uint32_t)FOURCC_ANY ; break;
3037         case SWITCH_IMG_FMT_RGB555:       fourcc = (uint32_t)FOURCC_ANY ; break;
3038         case SWITCH_IMG_FMT_UYVY:         fourcc = (uint32_t)FOURCC_ANY ; break;
3039         case SWITCH_IMG_FMT_YUY2:         fourcc = (uint32_t)FOURCC_YUY2; break;
3040         case SWITCH_IMG_FMT_YVYU:         fourcc = (uint32_t)FOURCC_ANY ; break;
3041         case SWITCH_IMG_FMT_BGR24:        fourcc = (uint32_t)FOURCC_RAW ; break;
3042         case SWITCH_IMG_FMT_RGB32_LE:     fourcc = (uint32_t)FOURCC_ANY ; break;
3043         case SWITCH_IMG_FMT_ARGB:         fourcc = (uint32_t)FOURCC_ARGB; break;
3044         case SWITCH_IMG_FMT_ARGB_LE:     fourcc = (uint32_t)FOURCC_ANY ; break;
3045         case SWITCH_IMG_FMT_RGB565_LE:    fourcc = (uint32_t)FOURCC_ANY ; break;
3046         case SWITCH_IMG_FMT_RGB555_LE:    fourcc = (uint32_t)FOURCC_ANY ; break;
3047         case SWITCH_IMG_FMT_YV12:         fourcc = (uint32_t)FOURCC_ANY ; break;
3048         case SWITCH_IMG_FMT_I420:         fourcc = (uint32_t)FOURCC_I420; break;
3049         case SWITCH_IMG_FMT_VPXV12:       fourcc = (uint32_t)FOURCC_ANY ; break;
3050         case SWITCH_IMG_FMT_VPXI420:      fourcc = (uint32_t)FOURCC_ANY ; break;
3051         case SWITCH_IMG_FMT_I422:         fourcc = (uint32_t)FOURCC_ANY ; break;
3052         case SWITCH_IMG_FMT_I444:         fourcc = (uint32_t)FOURCC_ANY ; break;
3053         case SWITCH_IMG_FMT_I440:         fourcc = (uint32_t)FOURCC_ANY ; break;
3054         case SWITCH_IMG_FMT_444A:         fourcc = (uint32_t)FOURCC_ANY ; break;

```

⁶参见<http://www.fourcc.org/fourcc.php>及<http://www.fourcc.org/codecs.php>。

```

3055         case SWITCH_IMG_FMT_I42016:    fourcc = (uint32_t)FOURCC_ANY ; break;
3056         case SWITCH_IMG_FMT_I42216:    fourcc = (uint32_t)FOURCC_ANY ; break;
3057         case SWITCH_IMG_FMT_I44416:    fourcc = (uint32_t)FOURCC_ANY ; break;
3058         case SWITCH_IMG_FMT_I44016:    fourcc = (uint32_t)FOURCC_ANY ; break;
3059         default: fourcc = (uint32_t)FOURCC_ANY;
3060     }
3061
3062     return fourcc;
3063 }

```

将 FreeSWITCH 图像转换为 RAW（原始 RGB）格式（在内存中的表示为 **BGRBGR**）。

```

3066 SWITCH_DECLARE(switch_status_t) switch_img_to_raw(switch_image_t *src, void *dest, int stride,
↪ switch_img_fmt_t fmt)
3067 {
3068     #ifdef SWITCH_HAVE_YUV
3069         uint32_t fourcc;
3070         int ret;
3071
3072         switch_assert(dest);
3073
3074         fourcc = switch_img_fmt2fourcc(fmt);
3075
3076         if (fourcc == FOURCC_ANY) {
3077             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "unsupported format: %d\n", fmt);
3078             return SWITCH_STATUS_FALSE;
3079         }
3080
3081         if (src->fmt == SWITCH_IMG_FMT_I420) {
3082             ret = ConvertFromI420(src->planes[0], src->stride[0],
3083                                   src->planes[1], src->stride[1],
3084                                   src->planes[2], src->stride[2],
3085                                   dest, stride,
3086                                   src->d_w, src->d_h,
3087                                   fourcc);
3088         } else if (src->fmt == SWITCH_IMG_FMT_ARGB && fmt == src->fmt) {
3089             ret = ARGBCopy(src->planes[SWITCH_PLANE_PACKED], src->stride[SWITCH_PLANE_PACKED],
3090                           dest, stride,
3091                           src->d_w, src->d_h);
3092         } else {
3093             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Conversion not supported %d ->
↪ %d\n", src->fmt, fmt);
3094             return SWITCH_STATUS_FALSE;
3095         }
3096
3097         return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3101     }

```

从 RAW 格式转换为 FreeSWITCH 的图像格式 (I420 或 ARGB)。

```
3103 SWITCH_DECLARE(switch_status_t) switch_img_from_raw(switch_image_t *dest, void *src,
↪ switch_img_fmt_t fmt, int width, int height)
3104 {
3105     #ifdef SWITCH_HAVE_YUV
3106         uint32_t fourcc;
3107         int ret = -1;
3108
3109         fourcc = switch_img_fmt2fourcc(fmt);
3110
3111         if (fourcc == FOURCC_ANY) {
3112             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "unsupported format: %d\n", fmt);
3113             return SWITCH_STATUS_FALSE;
3114         }
3115
3116         if (!dest && width > 0 && height > 0) dest = switch_img_alloc(NULL, SWITCH_IMG_FMT_I420, width,
↪ height, 1);
3117         if (!dest) return SWITCH_STATUS_FALSE;
3118
3119         if (width == 0 || height == 0) {
3120             width = dest->d_w;
3121             height = dest->d_h;
3122         }
3123         ...
3138         if (dest->fmt == SWITCH_IMG_FMT_I420) {
3139             ret = ConvertToI420(src, 0,
3140                               dest->planes[0], dest->stride[0],
3141                               dest->planes[1], dest->stride[1],
3142                               dest->planes[2], dest->stride[2],
3143                               0, 0,
3144                               width, height,
3145                               width, height,
3146                               0, fourcc);
3147         } else if (dest->fmt == SWITCH_IMG_FMT_ARGB) {
3148             ConvertToARGB(src, 0,
3149                           dest->planes[0], width * 4,
3150                           0, 0,
3151                           width, height,
3152                           width, height,
3153                           0, fourcc);
3154         }
3155
3156         return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3160     }
```

缩放。

```

3162 SWITCH_DECLARE(switch_status_t) switch_img_scale(switch_image_t *src, switch_image_t **destP, int
↪ width, int height)
3163 {
...
3177     if (!dest) dest = switch_img_alloc(NULL, src->fmt, width, height, 1);
3178
3179     if (src->fmt == SWITCH_IMG_FMT_I420) {
3180         ret = I420Scale(src->planes[0], src->stride[0],
3181             src->planes[1], src->stride[1],
3182             src->planes[2], src->stride[2],
3183             src->d_w, src->d_h,
3184             dest->planes[0], dest->stride[0],
3185             dest->planes[1], dest->stride[1],
3186             dest->planes[2], dest->stride[2],
3187             width, height,
3188             kFilterBox);
3189     } else if (src->fmt == SWITCH_IMG_FMT_ARGB) {
3190         ret = ARGBScale(src->planes[SWITCH_PLANE_PACKED], src->d_w * 4,
3191             src->d_w, src->d_h,
3192             dest->planes[SWITCH_PLANE_PACKED], width * 4,
3193             width, height,
3194             kFilterBox);
3195     }
...
3210 }

```

镜像（照镜子，左右翻转）。

```

3213 SWITCH_DECLARE(switch_status_t) switch_img_mirror(switch_image_t *src, switch_image_t **destP)
3214 {
...
3225     if (!dest) dest = switch_img_alloc(NULL, src->fmt, src->d_w, src->d_h, 1);
3226
3227     if (src->fmt == SWITCH_IMG_FMT_I420) {
3228         ret = I420Mirror(src->planes[0], src->stride[0],
3229             src->planes[1], src->stride[1],
3230             src->planes[2], src->stride[2],
3231             dest->planes[0], dest->stride[0],
3232             dest->planes[1], dest->stride[1],
3233             dest->planes[2], dest->stride[2],
3234             src->d_w, src->d_h);
3235     } else if (src->fmt == SWITCH_IMG_FMT_ARGB) {
3236         ret = ARGBMirror(src->planes[SWITCH_PLANE_PACKED], src->d_w * 4,

```

```
3237             dest->planes[SWITCH_PLANE_PACKED], src->d_w * 4,  
3238             src->d_w, src->d_h);  
3239  
3240     }  
...  
3255 }
```

寻找合适的位置坐标。

```
3257 SWITCH_DECLARE(void) switch_img_find_position(switch_img_position_t pos, int sw, int sh, int iw, int  
↪ ih, int *xP, int *yP)  
3258 {  
3259     switch(pos) {  
3260     case POS_NONE:  
3261     case POS_LEFT_TOP:  
3262         *xP = 0;  
3263         *yP = 0;  
3264         break;  
3265     case POS_LEFT_MID:  
3266         *xP = 0;  
3267         *yP = (sh - ih) / 2;  
3268         break;  
3269     case POS_LEFT_BOT:  
3270         *xP = 0;  
3271         *yP = (sh - ih);  
3272         break;  
3273     case POS_CENTER_TOP:  
3274         *xP = (sw - iw) / 2;  
3275         *yP = 0;  
3276         break;  
3277     case POS_CENTER_MID:  
3278         *xP = (sw - iw) / 2;  
3279         *yP = (sh - ih) / 2;  
3280         break;  
3281     case POS_CENTER_BOT:  
3282         *xP = (sw - iw) / 2;  
3283         *yP = (sh - ih);  
3284         break;  
3285     case POS_RIGHT_TOP:  
3286         *xP = (sw - iw);  
3287         *yP = 0;  
3288         break;  
3289     case POS_RIGHT_MID:  
3290         *xP = (sw - iw);  
3291         *yP = (sh - ih) / 2;  
3292         break;
```

```

3293     case POS_RIGHT_BOT:
3294         *xP = (sw - iw);
3295         *yP = (sh - ih);
3296         break;
3297     };
3298 }

```

使用 `libgd` 读文件（实验性的函数）。

```

3302 SWITCH_DECLARE(switch_image_t *) switch_img_read_file(const char* file_name)
3303 {
3304     switch_image_t *img = switch_img_alloc(NULL, SWITCH_IMG_FMT_ARGB, 1, 1, 1);
3305     gdImagePtr gd = NULL;
3306     char *ext;
3307     FILE *fp;
3308
3309     if (!img) return NULL;
3310
3311     // gd = gdImageCreateFromFile(file_name); // only available in 2.1.1
3312
3313     ext = strrchr(file_name, '.');
3314     if (!ext) goto err;
3315
3316     fp = fopen(file_name, "rb");
3317     if (!fp) goto err;
3318
3319     if (!strcmp(ext, ".png")) {
3320         gd = gdImageCreateFromPng(fp);
3321     } else if (!strcmp(ext, ".gif")) {
3322         gd = gdImageCreateFromGif(fp);
3323     } else if (!strcmp(ext, ".jpg") || !strcmp(ext, ".jpeg")) {
3324         gd = gdImageCreateFromJpeg(fp);
3325     } else {
3326         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Not supported file type: %s\n",
↪ ext);
3327     }
3328
3329     fclose(fp);
3330     if (!gd) goto err;
3331
3332     img->fmt = SWITCH_IMG_FMT_GD;
3333     img->d_w = gd->sx;
3334     img->d_h = gd->sy;
3335     img->user_priv = gd;
3336     return img;
3337

```

```

3338     err:
3339         switch_img_free(&img);
3340         return NULL;
3341     }

```

封装 libyuv 的同名函数。

```

3349     SWITCH_DECLARE(switch_status_t) switch_I420_copy(const uint8_t *src_y, int src_stride_y,
3350         const uint8_t *src_u, int src_stride_u,
3351         const uint8_t *src_v, int src_stride_v,
3352         uint8_t *dst_y, int dst_stride_y,
3353         uint8_t *dst_u, int dst_stride_u,
3354         uint8_t *dst_v, int dst_stride_v,
3355         int width, int height)
3356     {
3357         int ret = I420Copy(src_y, src_stride_y, src_u, src_stride_u, src_v, src_stride_v,
3358             dst_y, dst_stride_y, dst_u, dst_stride_u, dst_v, dst_stride_v,
3359             width, height);
3360         return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3361     }

```

简化版的封装，支持传入平面数组，而不是每个平面的指针。

```

3367     SWITCH_DECLARE(switch_status_t) switch_I420_copy2(uint8_t *src_planes[], int src_stride[],
3368         uint8_t *dst_planes[], int dst_stride[],
3369         int width, int height)
3370     {
3371         int ret = I420Copy(src_planes[SWITCH_PLANE_Y], src_stride[SWITCH_PLANE_Y],
3372             src_planes[SWITCH_PLANE_U], src_stride[SWITCH_PLANE_U],
3373             src_planes[SWITCH_PLANE_V], src_stride[SWITCH_PLANE_V],
3374             dst_planes[SWITCH_PLANE_Y], dst_stride[SWITCH_PLANE_Y],
3375             dst_planes[SWITCH_PLANE_U], dst_stride[SWITCH_PLANE_U],
3376             dst_planes[SWITCH_PLANE_V], dst_stride[SWITCH_PLANE_V],
3377             width, height);
3378         return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3379     }

```

libyuv 同名函数封装。

```

3385     SWITCH_DECLARE(switch_status_t) switch_I420ToARGB(const uint8_t *src_y, int src_stride_y,
3386         const uint8_t *src_u, int src_stride_u,

```

```
3387     const uint8_t *src_v, int src_stride_v,
3388     uint8_t *dst_argb, int dst_stride_argb,
3389     int width, int height)
3390 {
3391
3393     int ret = I420ToARGB(src_y, src_stride_y, src_u, src_stride_u, src_v, src_stride_v,
3394                         dst_argb, dst_stride_argb, width, height);
3395
3396     return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3400 }
```

libyuv 同名函数封装。

```
3403 SWITCH_DECLARE(switch_status_t) switch_RGBAToARGB(const uint8_t* src_frame, int src_stride_frame,
3404     uint8_t* dst_argb, int dst_stride_argb,
3405     int width, int height)
3406 {
3408     int ret = RGBAToARGB(src_frame, src_stride_frame, dst_argb, dst_stride_argb, width, height);
3409
3410     return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3414 }
```

libyuv 同名函数封装。

```
3417 SWITCH_DECLARE(switch_status_t) switch_ABGRToARGB(const uint8_t* src_frame, int src_stride_frame,
3418     uint8_t* dst_argb, int dst_stride_argb,
3419     int width, int height)
3420 {
3422     int ret = ABGRToARGB(src_frame, src_stride_frame, dst_argb, dst_stride_argb, width, height);
3423
3424     return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3428 }
```

libyuv 同名函数封装。

```
3430 SWITCH_DECLARE(switch_status_t) switch_ARGBToARGB(const uint8_t* src_frame, int src_stride_frame,
3431     uint8_t* dst_argb, int dst_stride_argb,
3432     int width, int height)
3433 {
3435     int ret = ARGBToARGB(src_frame, src_stride_frame, dst_argb, dst_stride_argb, width, height);
3436 }
```

```
3437     return ret == 0 ? SWITCH_STATUS_SUCCESS : SWITCH_STATUS_FALSE;
3441 }
```

解析字符串。

```
3444 SWITCH_DECLARE(void) switch_core_video_parse_filter_string(switch_core_video_filter_t *filters,
↳ const char *filter_str)
3445 {
3450     if (switch_stristr("fg-gray", filter_str)) *filters |= SCV_FILTER_GRAY_FG;
3454     if (switch_stristr("bg-gray", filter_str)) *filters |= SCV_FILTER_GRAY_BG;
3458     if (switch_stristr("fg-sepia", filter_str)) *filters |= SCV_FILTER_SEPIA_FG;
3462     if (switch_stristr("bg-sepia", filter_str)) *filters |= SCV_FILTER_SEPIA_BG;
3466     if (switch_stristr("fg-8bit", filter_str)) *filters |= SCV_FILTER_8BIT_FG;
3469 }
```

4.22 switch_scheduler.c

本节基于 Commit Hash [1d68ab18](#)。

`switch_scheduler` 实现了定时器任务调度功能。

下面代码定义了任务调度器的数据结构。

```
35 struct switch_scheduler_task_container {
36     switch_scheduler_task_t task;
37     int64_t executed;
38     int in_thread;
39     int destroyed;
40     int running;
41     switch_scheduler_func_t func;
42     switch_memory_pool_t *pool;
43     uint32_t flags;
44     char *desc;
45     struct switch_scheduler_task_container *next;
46 };
47 typedef struct switch_scheduler_task_container switch_scheduler_task_container_t;
```

调度器相关的全局变量。

```

49 static struct {
50     switch_scheduler_task_container_t *task_list;
51     switch_mutex_t *task_mutex;
52     uint32_t task_id;
53     int task_thread_running;
54     switch_queue_t *event_queue;
55     switch_memory_pool_t *memory_pool;
56 } globals;

```

L63, 执行调度器的任务。如果该任务是可重复的 (L65), 则设置下一次执行的时间 (L66)。若需要继续执行, 则触发一个 **RE_SCHEDULE** 事件 (L54), 广播下一次要执行的时间。否则, 设置已销毁标志 (L80)。

```

58 static void switch_scheduler_execute(switch_scheduler_task_container_t *tp)
59 {
60     switch_event_t *event;
61     //switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Executing task %u %s (%s)\n", tp-
↪ >task.task_id, tp->desc, switch_str_nil(tp->task.group));
62
63     tp->func(&tp->task);
64
65     if (tp->task.repeat) {
66         tp->task.runtime = switch_epoch_time_now(NULL) + tp->task.repeat;
67     }
68
69     if (tp->task.runtime > tp->executed) {
70         tp->executed = 0;
71         if (switch_event_create(&event, SWITCH_EVENT_RE_SCHEDULE) == SWITCH_STATUS_SUCCESS) {
72             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Task-ID", "%u", tp->task.task_id);
73             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Task-Desc", tp->desc);
74             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Task-Group", switch_str_nil(tp-
↪ >task.group));
75             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Task-Runtime", "%" SWITCH_INT64_T_FMT,
↪ tp->task.runtime);
76             switch_queue_push(globals.event_queue, event);
77             event = NULL;
78         }
79     } else {
80         tp->destroyed = 1;
81     }
82 }

```

L84 函数会在一个独立的线程中执行, 不会阻塞其它任务。它会在 L92 调用上面 L58 实现的函数执行任务。

```

84 static void *SWITCH_THREAD_FUNC task_own_thread(switch_thread_t *thread, void *obj)
85 {
86     switch_scheduler_task_container_t *tp = (switch_scheduler_task_container_t *) obj;
87     switch_memory_pool_t *pool;
88
89     pool = tp->pool;
90     tp->pool = NULL;
91
92     switch_scheduler_execute(tp);
93     switch_core_destroy_memory_pool(&pool);
94     tp->in_thread = 0;
95
96     return NULL;
97 }

```

L99 中的函数在一个线程中执行，它会循环执行所有到期的任务。

首先锁定临界区（L104），循环（L106）遍历任务链表，L107 是退出条件，否则（L109）会继续执行。L110 获取当前时间，如果当前时间超过了约定的执行时间且任务没有在单独的线程中执行（L111），则开始执行任务。

```

99 static int task_thread_loop(int done)
100 {
101     switch_scheduler_task_container_t *tofree, *tp, *last = NULL;
102
103
104     switch_mutex_lock(globals.task_mutex);
105
106     for (tp = globals.task_list; tp; tp = tp->next) {
107         if (done) {
108             tp->destroyed = 1;
109         } else if (!tp->destroyed) {
110             int64_t now = switch_epoch_time_now(NULL);
111             if (now >= tp->task.runtime && !tp->in_thread) {

```

如果当前时间比约定的时间慢了一秒（L113），则打印一条警告信息（L114）。这在时钟不准的服务器上会看到，或者，在笔记本合上后来又打开的时候会看到这样的警告。

```

112         int32_t diff = (int32_t) (now - tp->task.runtime);
113         if (diff > 1) {
114             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Task was executed late by
↪ %d seconds %u %s (%s)\n",
115                             diff, tp->task.task_id, tp->desc, switch_str_nil(tp->task.group));
116         }

```

设置任务执行时间（L117），如果任务需要在独立的线程中执行（L118），则启动一个线程调用 L84 中描述的函数去执行。

```

117         tp->executed = now;
118         if (switch_test_flag(tp, SSHF_OWN_THREAD)) {
119             switch_thread_t *thread;
120             switch_threadattr_t *thd_attr;
121             switch_core_new_memory_pool(&tp->pool);
122             switch_threadattr_create(&thd_attr, tp->pool);
123             switch_threadattr_detach_set(thd_attr, 1);
124             tp->in_thread = 1;
125             switch_thread_create(&thread, thd_attr, task_own_thread, tp, tp->pool);

```

否则，直接调用 `switch_scheduler_execute()` 执行任务（L129）。注意由于执行任务并不占用临界区，所以 L128 会让出临界区，以避免占用过长。

```

126         } else {
127             tp->running = 1;
128             switch_mutex_unlock(globals.task_mutex);
129             switch_scheduler_execute(tp);
130             switch_mutex_lock(globals.task_mutex);
131             tp->running = 0;
132         }
133     }
134 }
135 }

```

重新遍历任务链表（L138），如果任务已完成（L139），则从链表中删除该任务（L143），并触发相应事件（L148）。

```

136     switch_mutex_unlock(globals.task_mutex);
137     switch_mutex_lock(globals.task_mutex);
138     for (tp = globals.task_list; tp;) {
139         if (tp->destroyed && !tp->in_thread) {
140             switch_event_t *event;
141
142             tofree = tp;
143             tp = tp->next;
144             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Deleting task %u %s (%s)\n",

```

```

145         tofree->task.task_id, tofree->desc, switch_str_nil(tofree->task.group));
146
147
148         if (switch_event_create(&event, SWITCH_EVENT_DEL_SCHEDULE) == SWITCH_STATUS_SUCCESS) {
149             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Task-ID", "%u", tofree-
↳ >task.task_id);
150             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Task-Desc", tofree->desc);
151             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Task-Group",
↳ switch_str_nil(tofree->task.group));
152             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Task-Runtime", "%"
↳ SWITCH_INT64_T_FMT, tofree->task.runtime);
153             switch_queue_push(globals.event_queue, event);
154             event = NULL;
155         }

```

后面做一个环境清理，如果任务有 `SSH_F_FREE_ARG` 标志，则清楚任务的参数（L164）所占用的内存。

```

157         if (last) {
158             last->next = tofree->next;
159         } else {
160             globals.task_list = tofree->next;
161         }
162         switch_safe_free(tofree->task.group);
163         if (tofree->task.cmd_arg && switch_test_flag(tofree, SSH_F_FREE_ARG)) {
164             free(tofree->task.cmd_arg);
165         }
166         switch_safe_free(tofree->desc);
167         free(tofree);
168     } else {
169         last = tp;
170         tp = tp->next;
171     }
172 }
173 switch_mutex_unlock(globals.task_mutex);
174
175 return done;
176 }

```

L178 是任务调度的主线程。启动后，进入无限循环（L184），在 L185 调用上面 L99 中提到的函数不断地执行任务。如果在任务执行过程中产生了事件，则会在此触发（L188 ~ L190）。

```

178 static void *SWITCH_THREAD_FUNC switch_scheduler_task_thread(switch_thread_t *thread, void *obj)
179 {

```

```

180     void *pop;
181     globals.task_thread_running = 1;
182
183     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Starting task thread\n");
184     while (globals.task_thread_running == 1) {
185         if (task_thread_loop(0)) {
186             break;
187         }
188         if (switch_queue_pop_timeout(globals.event_queue, &pop, 500000) == SWITCH_STATUS_SUCCESS) {
189             switch_event_t *event = (switch_event_t *) pop;
190             switch_event_fire(&event);
191         }
192     }

```

如果需要停止调度器，则 L184 中的条件终止，在 L194 会再执行一次 `task_thread_loop` 参数为 1 清理相关任务。

```

194     task_thread_loop(1);
195
196     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Task thread ending\n");
197
198     while(switch_queue_etrypop(globals.event_queue, &pop) == SWITCH_STATUS_SUCCESS) {
199         switch_event_t *event = (switch_event_t *) pop;
200         switch_event_destroy(&event);
201     }
202
203     globals.task_thread_running = 0;
204
205     return NULL;
206 }

```

向调度器添加一个任务。设置任务的执行时间 `task_runtime`，回调函数 `func`，描述信息 `desc` 和任务组 `group`（后续可以删除整个任务组）。回调函数的参数由 `cmd_arg` 传进来。

```

208 SWITCH_DECLARE(uint32_t) switch_scheduler_add_task(time_t task_runtime,
209                                                     switch_scheduler_func_t func,
210                                                     const char *desc, const char *group, uint32_t cmd_id,
↪ void *cmd_arg, switch_scheduler_flag_t flags)

```

其中，`flags` 在 `switch_types.h` 中定义：

```

typedef enum {
    SSHF_NONE = 0,
    SSHF_OWN_THREAD = (1 << 0), //任务在独立的线程中执行
    SSHF_FREE_ARG = (1 << 1),   //任务结束后释放 cmd_arg 内存
    SSHF_NO_DEL = (1 << 2)      //任务不可删除
} switch_scheduler_flag_enum_t;
typedef uint32_t switch_scheduler_flag_t;

```

书接上文。L218 初始化任务容器，并初始化相应的值（L226 ~ L234），插入到任务链表尾部（L236 ~ L242）。

```

212     switch_scheduler_task_container_t *container, *tp;
213     switch_event_t *event;
214     switch_time_t now = switch_epoch_time_now(NULL);
215     switch_ssize_t hlen = -1;
216
217     switch_mutex_lock(globals.task_mutex);
218     switch_zmalloc(container, sizeof(*container));
219     switch_assert(func);
220
221     if (task_runtime < now) {
222         container->task.repeat = (uint32_t)task_runtime;
223         task_runtime += now;
224     }
225
226     container->func = func;
227     container->task.created = now;
228     container->task.runtime = task_runtime;
229     container->task.group = strdup(group ? group : "none");
230     container->task.cmd_id = cmd_id;
231     container->task.cmd_arg = cmd_arg;
232     container->flags = flags;
233     container->desc = strdup(desc ? desc : "none");
234     container->task.hash = switch_ci_hashfunc_default(container->task.group, &hlen);
235
236     for (tp = globals.task_list; tp && tp->next; tp = tp->next);
237
238     if (tp) {
239         tp->next = container;
240     } else {
241         globals.task_list = container;
242     }

```

后面就是设置 `task_id`，并触发一个任务添加的事件。

```

244     for (container->task.task_id = 0; !container->task.task_id; container->task.task_id = +
↳ +globals.task_id);
245
246     switch_mutex_unlock(globals.task_mutex);
247
248     tp = container;
249     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Added task %u %s (%s) to run at %"
↳ SWITCH_INT64_T_FMT "\n",
250                     tp->task.task_id, tp->desc, switch_str_nil(tp->task.group), tp->task.runtime);
251
252     if (switch_event_create(&event, SWITCH_EVENT_ADD_SCHEDULE) == SWITCH_STATUS_SUCCESS) {
253         switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Task-ID", "%u", tp->task.task_id);
254         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Task-Desc", tp->desc);
255         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "Task-Group", switch_str_nil(tp-
↳ >task.group));
256         switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Task-Runtime", "%" SWITCH_INT64_T_FMT, tp-
↳ >task.runtime);
257         switch_queue_push(globals.event_queue, event);
258         event = NULL;
259     }
260     return container->task.task_id;
261 }

```

删除任务。遍历链表（L269），如果找到相应的 `task_id`（L170），且满足相应的条件（可删除并且不在执行，L271，L277），则删除（L283）。该函数返回成功删除的任务数量（L290）。

```

263 SWITCH_DECLARE(uint32_t) switch_scheduler_del_task_id(uint32_t task_id)
264 {
265     switch_scheduler_task_container_t *tp;
266     uint32_t delcnt = 0;
267
268     switch_mutex_lock(globals.task_mutex);
269     for (tp = globals.task_list; tp; tp = tp->next) {
270         if (tp->task.task_id == task_id) {
271             if (switch_test_flag(tp, SSHF_NO_DEL)) {
272                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Attempt made to delete
↳ undeletable task #%u (group %s)\n",
273                                 tp->task.task_id, tp->task.group);
274                 break;
275             }
276
277             if (tp->running) {
278                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Attempt made to delete
↳ running task #%u (group %s)\n",
279                                 tp->task.task_id, tp->task.group);

```

```

280         break;
281     }
282
283     tp->destroyed++;
284     delcnt++;
285     break;
286 }
287 }
288 switch_mutex_unlock(globals.task_mutex);
289
290 return delcnt;
291 }

```

删掉整个任务组。

```

293 SWITCH_DECLARE(uint32_t) switch_scheduler_del_task_group(const char *group)
294 {
295     switch_scheduler_task_container_t *tp;
296     uint32_t delcnt = 0;
297     switch_ssize_t hlen = -1;
298     unsigned long hash = 0;
299
300     if (zstr(group)) {
301         return 0;
302     }
303
304     hash = switch_ci_hashfunc_default(group, &hlen);
305
306     switch_mutex_lock(globals.task_mutex);
307     for (tp = globals.task_list; tp; tp = tp->next) {
308         if (tp->destroyed) {
309             continue;
310         }
311         if (hash == tp->task.hash && !strcmp(tp->task.group, group)) {
312             if (switch_test_flag(tp, SSHF_NO_DEL)) {
313                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Attempt made to delete
↳ undeletable task #%u (group %s)\n",
314                                 tp->task.task_id, group);
315             }
316             continue;
317         }
318         tp->destroyed++;
319         delcnt++;
320     }
321     switch_mutex_unlock(globals.task_mutex);
322

```

```
323     return delcnt;
324 }
```

启动任务调度线程。

```

326 switch_thread_t *task_thread_p = NULL;
327
328 SWITCH_DECLARE(void) switch_scheduler_task_thread_start(void)
329 {
330
331     switch_threadattr_t *thd_attr;
332
333     switch_core_new_memory_pool(&globals.memory_pool);
334     switch_threadattr_create(&thd_attr, globals.memory_pool);
335     switch_mutex_init(&globals.task_mutex, SWITCH_MUTEX_NESTED, globals.memory_pool);
336     switch_queue_create(&globals.event_queue, 250000, globals.memory_pool);
337
338     switch_thread_create(&task_thread_p, thd_attr, switch_scheduler_task_thread, NULL,
↪     globals.memory_pool);
339 }

```

停止任务调度线程。

```

341 SWITCH_DECLARE(void) switch_scheduler_task_thread_stop(void)
342 {
343     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Stopping Task Thread\n");
344     if (globals.task_thread_running == 1) {
345         int sanity = 0;
346         switch_status_t st;
347
348         globals.task_thread_running = -1;
349
350         switch_thread_join(&st, task_thread_p);
351
352         while (globals.task_thread_running) {
353             switch_yield(100000);
354             if (++sanity > 10) {
355                 break;
356             }
357         }
358     }
359
360     switch_core_destroy_memory_pool(&globals.memory_pool);
361 }

```

注意，任务回调函数应该尽可能少占用运行时间，如果占用时间比较多，则应该在独立的线程中运行（加 `SSH_F_OWN_THREAD` 标志），以免阻塞其它任务。

4.23 switch_core_memory.c

张洪

本章基于 Commit Hash [1681db4](#)。

本文件实现了内存池相关的操作函数，FreeSWITCH 的内存池是基于 APR 的内存池技术。

获取 session 的内存池（L63），从 session 的内存池中申请内存（L72），从核心内存池中申请内存。

```
63 SWITCH_DECLARE(switch_memory_pool_t *) switch_core_session_get_pool(switch_core_session_t *session)

72 SWITCH_DECLARE(void *) switch_core_perform_session_alloc(switch_core_session_t *session, switch_size_t
↪ memory, const char *file, const char *func, int line)

108 SWITCH_DECLARE(void *) switch_core_perform_permanent_alloc(switch_size_t memory, const char *file,
↪ const char *func, int line)
```

使用核心内存池拷贝字符串（L138），使用 session 的内存池生成格式化字符串（L175），使用指定内存池生成格式化字符串（L216），使用 session 的内存池拷贝字符串（L227），使用指定内存池拷贝字符串（L271）。

```
138 SWITCH_DECLARE(char *) switch_core_perform_permanent_strdup(const char *todup, const char *file, const
↪ char *func, int line)

175 SWITCH_DECLARE(char *) switch_core_session_sprintf(switch_core_session_t *session, const char *fmt,
↪ ...)

216 SWITCH_DECLARE(char *) switch_core_sprintf(switch_memory_pool_t *pool, const char *fmt, ...)

227 SWITCH_DECLARE(char *) switch_core_perform_session_strdup(switch_core_session_t *session, const char
↪ *todup, const char *file, const char *func, int line)

271 SWITCH_DECLARE(char *) switch_core_perform_strdup(switch_memory_pool_t *pool, const char *todup, const
↪ char *file, const char *func, int line)
```

给内存池绑定相关联的用户数据（L310）和获取该绑定数据（L315）。

```
310 SWITCH_DECLARE(void) switch_core_memory_pool_set_data(switch_memory_pool_t *pool, const char *key, void  
↳ *data)
```

```
315 SWITCH_DECLARE(void *) switch_core_memory_pool_get_data(switch_memory_pool_t *pool, const char *key)
```

给内存池打标签。

```
324 SWITCH_DECLARE(void) switch_core_memory_pool_tag(switch_memory_pool_t *pool, const char *tag)
```

清理内存池中的内存。

```
329 SWITCH_DECLARE(void) switch_pool_clear(switch_memory_pool_t *p)
```

创建内存池, 如果不需要立即创建, 则会从可重复使用的内存池队列 `memory_manager.pool_recycle_queue` 中取出, 取出失败才创建新内存池。

```
352 SWITCH_DECLARE(switch_status_t) switch_core_perform_new_memory_pool(switch_memory_pool_t **pool, const  
↳ char *file, const char *func, int line)  
353 {  
...  
355 #ifdef INSTANTLY_DESTROY_POOLS  
356     apr_pool_create(pool, NULL);  
357     switch_assert(*pool != NULL);  
358 #else  
...  
373     if (switch_queue_trypop(memory_manager.pool_recycle_queue, &pop) == SWITCH_STATUS_SUCCESS && pop) {  
374         *pool = (switch_memory_pool_t *) pop;  
375     } else {  
...  
397         apr_pool_create(pool, NULL);  
...  
416 }
```

销毁指定内存池, 如果不需要立即销毁则将内存池 push 到销毁队列, 在销毁线程中销毁。

```
418 SWITCH_DECLARE(switch_status_t) switch_core_perform_destroy_memory_pool(switch_memory_pool_t **pool,  
↳ const char *file, const char *func, int line)  
419 {
```



```
...
426 #ifdef INSTANTLY_DESTROY_POOLS
...
434 #else
435     if ((memory_manager.pool_thread_running != 1) || (switch_queue_push(memory_manager.pool_queue,
↪ *pool) != SWITCH_STATUS_SUCCESS)) {
...
449 }
```

从指定内存池中申请内存。

```
451 SWITCH_DECLARE(void *) switch_core_perform_alloc(switch_memory_pool_t *pool, switch_size_t memory,
↪ const char *file, const char *func, int line)
```

回收内存池循环队列 `memory_manager.pool_recycle_queue` 中的内存池。

```
483 SWITCH_DECLARE(void) switch_core_memory_reclaim(void)
```

内存池销毁线程，销毁内存池队列 `memory_manager.pool_queue` 中的内存池。

```
508 static void *SWITCH_THREAD_FUNC pool_thread(switch_thread_t *thread, void *obj)
```

停止内存池销毁线程，销毁内存池队列中的剩余项。

```
600 void switch_core_memory_stop(void)
```

初始化核心内存池，返回的值将赋值给核心全局变量 `runtime.memory_pool`。

```
618 switch_memory_pool_t *switch_core_memory_init(void)
```

4.24 switch_core_timer.c

张洪

本章基于 Commit Hash [1681db4](#)。

本文件实现了核心的 timer（定时器）接口，FreeSWITCH 中所有 timer 的使用都需要调用本文档中的接口，这些接口会找到对应 timer 实际的实现模块，并调用对用的方法。

初始化 timer（L38），根据 timer 的名称找到对应的实现（L44），设置好初始化参数并调用实现模块的初始化方法完成初始化。销毁 timer（L115），并调用 timer 实现模块的销毁方法最终销毁 timer。

```

38 SWITCH_DECLARE(switch_status_t) switch_core_timer_init(switch_timer_t *timer, const char *timer_name,
↳ int interval, int samples,
39                                     switch_memory_pool_t *pool)
40 {
...
44     if ((timer_interface = switch_loadable_module_get_timer_interface(timer_name)) == 0 || !
↳ timer_interface->timer_init) {
45         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "invalid timer %s!\n", timer_name);
46         return SWITCH_STATUS_GENERR;
47     }
48
49     timer->interval = interval;
50     timer->samples = samples;
51     timer->samplecount = samples;
52     timer->timer_interface = timer_interface;
...
64     return timer->timer_interface->timer_init(timer);
65 }

115 SWITCH_DECLARE(switch_status_t) switch_core_timer_destroy(switch_timer_t *timer)
116 {
...
122     timer->timer_interface->timer_destroy(timer);
...
132 }

```

阻塞等待定时器下一次到期（L67），步进定时器（L82），同步定时器（L93）。检查定时器（L103）是否到期，如果到期步进值 step 为 true 则步进定时器。

```

67 SWITCH_DECLARE(switch_status_t) switch_core_timer_next(switch_timer_t *timer)
82 SWITCH_DECLARE(switch_status_t) switch_core_timer_step(switch_timer_t *timer)

```

```

93 SWITCH_DECLARE(switch_status_t) switch_core_timer_sync(switch_timer_t *timer)
103 SWITCH_DECLARE(switch_status_t) switch_core_timer_check(switch_timer_t *timer, switch_bool_t step)

```

4.25 switch_core_port_allocator.c

李洋

本节基于 Commit Hash [1681db4](#)。

本文件实现了端口分配及释放接口，目前 FreeSWITCH 中 RTP 端口管理使用了该接口。

初始化端口分配器（L51），根据奇偶数分配要求，调整端口范围（L76 ~ L96），并计算出指定端口范围内的可用端口总数（L98 ~ L102）。

```

51 SWITCH_DECLARE(switch_status_t) switch_core_port_allocator_new(const char *ip, switch_port_t start,
52                                                                switch_port_t end, switch_port_flag_t
↪ flags, switch_core_port_allocator_t **new_allocator)
...
76     if (!(even && odd)) {
77         if (even) {
78             if ((start % 2) != 0) {
79                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Rounding odd start port %d to
↪ %d\n", start, start + 1);
80                 start++;
81             }
82             if ((end % 2) != 0) {
83                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Rounding odd end port %d to
↪ %d\n", end, end - 1);
84                 end--;
85             }
86         } else if (odd) {
87             if ((start % 2) == 0) {
88                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Rounding even start port %d
↪ to %d\n", start, start + 1);
89                 start++;
90             }
91             if ((end % 2) == 0) {
92                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Rounding even end port %d to
↪ %d\n", end, end - 1);
93                 end--;
94             }
95         }
96     }
97

```

```

98     alloc->track_len = (end - start) + 2;
99
100     if (!(even && odd)) {
101         alloc->track_len /= 2;
102     }

```

端口分配函数 (L143)。随机生成索引 (L152)，从该索引开始查找空闲的端口 (L161 ~ L169)。根据索引及奇偶数分配要求，查找出对应的端口 (L174 ~ L179)，同时检查该端口是否被占用 (L181 ~ L189)，检查通过则结束 (L191 ~ L195)。

```

143 SWITCH_DECLARE(switch_status_t) switch_core_port_allocator_request_port(switch_core_port_allocator_t
↪ *alloc, switch_port_t *port_ptr)

...
152     index = rand() % alloc->track_len;
...
161     while (alloc->track[index] && tries < alloc->track_len) {
162         tries++;
163         if (alloc->track[index] < 0) {
164             alloc->track[index]++;
165         }
166         if (++index >= alloc->track_len) {
167             index = 0;
168         }
169     }
170
171     if (tries < alloc->track_len) {
172         switch_bool_t r = SWITCH_TRUE;
173
174         if ((even && odd)) {
175             port = (switch_port_t) (index + alloc->start);
176         } else {
177             port = (switch_port_t) (index + (alloc->start / 2));
178             port *= 2;
179         }
180
181         if ((alloc->flags & SPF_ROBUST_UDP)) {
182             r = test_port(alloc, AF_INET, SOCK_DGRAM, port);
183             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "UDP port robustness check for
↪ port %d %s\n", port, r ? "pass" : "fail");
184         }

186         if ((alloc->flags & SPF_ROBUST_TCP)) {
187             r = test_port(alloc, AF_INET, SOCK_STREAM, port);
188             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "TCP port robustness check for
↪ port %d %s\n", port, r ? "pass" : "fail");

```

```
189         }
190
191         if (r) {
192             alloc->track[index] = 1;
193             alloc->track_used++;
194             status = SWITCH_STATUS_SUCCESS;
195             goto end;
196         } else {
```

端口释放函数（L218）。找到对应端口的索引（L229 ~ L233），并释放（L236 ~ L240）。

```
218 SWITCH_DECLARE(switch_status_t) switch_core_port_allocator_free_port(switch_core_port_allocator_t
↳ *alloc, switch_port_t port)
...
229     index = port - alloc->start;
230
231     if (!(even && odd)) {
232         index /= 2;
233     }
...
236     if (alloc->track[index] > 0) {
237         alloc->track[index] = -4;
238         alloc->track_used--;
239         status = SWITCH_STATUS_SUCCESS;
240     }
```

端口分配器销毁函数（L246）。

```
246 SWITCH_DECLARE(void) switch_core_port_allocator_destroy(switch_core_port_allocator_t **alloc)
```

4.26 switch_core_sqldb.c

殷鑫博

本章基于 Commit Hash 5fca55a0a。

在 FreeSWITCH 启动时，switch_core.c 文件里会进行数据库的启动操作，调用 switch_core_sqldb_start 函数，代码如下：

```

SWITCH_DECLARE(switch_status_t) switch_core_init(switch_core_flag_t flags, switch_bool_t console, const char
↳ **err)
...
    if (switch_core_sqldb_start(runtime.memory_pool, switch_test_flag(&runtime), SCF_USE_SQL) ? SWITCH_TRUE
↳ : SWITCH_FALSE) != SWITCH_STATUS_SUCCESS) {
        *err = "Error activating database";
        return SWITCH_STATUS_FALSE;
    }
...

```

数据库启动函数 `switch_core_sqldb_start` 定义如下：

```

switch_status_t switch_core_sqldb_start(switch_memory_pool_t *pool, switch_bool_t manage)

```

其中 L3369-L3371 初始化三个互斥锁，分别为 `dbh`io`ctl`，L3373 是对否需要启动数据库进行判断。

```

3369    switch_mutex_init(&sql_manager.dbh_mutex, SWITCH_MUTEX_NESTED, sql_manager.memory_pool);
3370    switch_mutex_init(&sql_manager.io_mutex, SWITCH_MUTEX_NESTED, sql_manager.memory_pool);
3371    switch_mutex_init(&sql_manager.ctl_mutex, SWITCH_MUTEX_NESTED, sql_manager.memory_pool);
3372
3373    if (!sql_manager.manage) goto skip;

```

L3378-L3390 表示启动系统默认数据库，若失败则输出相应错误日志，并清除 `SCF_USE_SQL` flag (L3388)，其中关键函数为 `switch_core_db_handle`，我们接下来就要讲解这个函数，GO ON!

```

3378    if (switch_core_db_handle(&sql_manager.dbh) != SWITCH_STATUS_SUCCESS) {
3379        switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Error Opening DB!\n");
3380
3381        if (switch_test_flag(&runtime), SCF_CORE_NON_SQLITE_DB_REQ)) {
3382            switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Failure! ODBC IS REQUIRED!\n");
3383            return SWITCH_STATUS_FALSE;
3384        }
3385
3386        switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "CORE DATABASE INITIALIZATION FAILURE!
↳ CHECK `core-db-dsn`!\n");
3387
3388        switch_clear_flag(&runtime, SCF_USE_SQL);
3389        return SWITCH_STATUS_FALSE;
3390    }

```

会优先选取 `odbc_dsn` 或 `dbname` 作为数据库的 `dsn`，若两者均不存在，则 `dsn` 默认设置为 `core`。

```

184 SWITCH_DECLARE(switch_status_t) _switch_core_db_handle(switch_cache_db_handle_t **dbh, const char *file,
↳ const char *func, int line)
...
193     if (!zstr(runtime.odbc_dsn)) {
194         dsn = runtime.odbc_dsn;
195     } else if (!zstr(runtime.dbname)) {
196         dsn = runtime.dbname;
197     } else {
198         dsn = "core";
199     }
...

```

在确定完 `dsn` 的取值后，则会根据 `dsn` 来确定后面连接数据库所需的 `type`、`dsn/db_path`、`user`、`pass`，这些都存在于 `_switch_cache_db_get_db_handle_dsn()` 函数内，是在此处进行调用：

```

201     if ((r = _switch_cache_db_get_db_handle_dsn(dbh, dsn, file, func, line)) != SWITCH_STATUS_SUCCESS) {
202         *dbh = NULL;
203     }

```

FreeSWITCH 目前可选数据库类型有 `pgsql/postgresql`、`sqlite/core`、`odbc` 三种，当然也可通过修改配置文件来实现支持其他种类数据库，这里就不做过多的阐述。

L345-L377 是根据不同的数据库类型设置相应的 `type` 和 `dsn/db_path`，当数据库类型为 `odbc` 时，需要设置 `user` 和 `pass` 用于后面数据库的连接操作。

当参数设置完成后，会调用 `_switch_cache_db_get_db_handle()` 函数进行连接数据库操作，我们接着往下看。

```

335 SWITCH_DECLARE(switch_status_t) _switch_cache_db_get_db_handle_dsn(switch_cache_db_handle_t **dbh, const
↳ char *dsn, const char *file, const char *func, int line)
{
345     if (!strncasecmp(dsn, "pgsql://", 8)) {
346         type = SCDB_TYPE_PGSQL;
347         connection_options.pgsql_options.dsn = (char *) (dsn + 8);
348     } else if (!strncasecmp(dsn, "postgresql://", 13)) {
349         type = SCDB_TYPE_PGSQL;
350         connection_options.pgsql_options.dsn = (char *) (dsn);
351     } else if (!strncasecmp(dsn, "sqlite://", 9)) {
352         type = SCDB_TYPE_CORE_DB;

```

```

353     connection_options.core_db_options.db_path = (char *)(dsn + 9);
354 } else if ((!(i = strncasecmp(dsn, "odbc://", 7))) || strchr(dsn+2, ':')) {
355     type = SCDB_TYPE_ODBC;
356
357     if (i) {
358         switch_set_string(tmp, dsn);
359     } else {
360         switch_set_string(tmp, dsn+7);
361     }
362
363     connection_options.odbc_options.dsn = tmp;
364
365     if ((p = strchr(tmp, ':')) {
366         *p++ = '\\0';
367         connection_options.odbc_options.user = p;
368
369         if ((p = strchr(connection_options.odbc_options.user, ':')) {
370             *p++ = '\\0';
371             connection_options.odbc_options.pass = p;
372         }
373     }
374 } else {
375     type = SCDB_TYPE_CORE_DB;
376     connection_options.core_db_options.db_path = (char *)dsn;
377 }
}
...
379 status = _switch_cache_db_get_db_handle(dbh, type, &connection_options, file, func, line);
...

```

当数据库使用句柄数达到最大值（`runtime.max_db_handles` 默认值为 50）时，会进行等待操作，等待时长为 `runtime.db_handle_timeout`，默认时长为 5 秒，若超过 5 秒，则判定连接数据库失败，那么数据库的启动操作就到此以失败告终。

```

387 SWITCH_DECLARE(switch_status_t) _switch_cache_db_get_db_handle(switch_cache_db_handle_t **dbh,
388                                                                switch_cache_db_handle_type_t type,
389                                                                switch_cache_db_connection_options_t
↪ *connection_options,
390                                                                const char *file, const char *func, int
↪ line)
...
405 while(runtime.max_db_handles && sql_manager.total_handles >= runtime.max_db_handles &&
↪ sql_manager.total_used_handles >= sql_manager.total_handles) {
406     if (!waiting++) {
407         switch_log_printf(SWITCH_CHANNEL_ID_LOG, file, func, line, NULL, SWITCH_LOG_WARNING, "Max
↪ handles %u exceeded, blocking...\n",

```



```
408             runtime.max_db_handles);
409     }
410
411     switch_yield(yield_len);
412     total_yield += yield_len;
413
414     if (runtime.db_handle_timeout && total_yield > runtime.db_handle_timeout) {
415         switch_log_printf(SWITCH_CHANNEL_ID_LOG, file, func, line, NULL, SWITCH_LOG_ERROR, "Error
↳ connecting\n");
416         *dbh = NULL;
417         return SWITCH_STATUS_FALSE;
418     }
419 }
...
```

若连接成功，则根据不同的数据库类型进行设置 `db_name`、`odbc_user`、`odbc_pass`、`db_type` 等参数。

```
421     switch (type) {
422     case SCDB_TYPE_PGSQL:
423     {
424         db_name = connection_options->pgsql_options.dsn;
425         odbc_user = NULL;
426         odbc_pass = NULL;
427         db_type = "pgsql";
428     }
429     case SCDB_TYPE_ODBC:
430     {
431         db_name = connection_options->odbc_options.dsn;
432         odbc_user = connection_options->odbc_options.user;
433         odbc_pass = connection_options->odbc_options.pass;
434         db_type = "odbc";
435     }
436     break;
437     case SCDB_TYPE_CORE_DB:
438     {
439         db_name = connection_options->core_db_options.db_path;
440         odbc_user = NULL;
441         odbc_pass = NULL;
442         db_type = "core_db";
443     }
444     break;
445 }
```

FreeSWITCH 会优先复用内存池 `sql_manager.handle_pool` 中已经存在的 `handle`，用来连接数

据库，实现函数为 `get_handle`，L459；若没有则会根据 `type` 新建 `dbh`，L462-L529。

```

459     if ((new_dbh = get_handle(db_str, db_callsite_str, thread_str))) {
460         switch_log_printf(SWITCH_CHANNEL_ID_LOG, file, func, line, NULL, SWITCH_LOG_DEBUG10,
461             "Reuse Unused Cached DB handle %s [%s]\n", new_dbh->name,
↪ switch_cache_db_type_name(new_dbh->type));
462     } else {
463         switch_core_db_t *db = NULL;
464         switch_odbc_handle_t *odbc_dbh = NULL;
465         switch_psql_handle_t *psql_dbh = NULL;
466
467         switch (type) {
468             case SCDB_TYPE_PGSQL:
469             ...
482             case SCDB_TYPE_ODBC:
469             ...
500             case SCDB_TYPE_CORE_DB:
469             ...
506             default:
469             ...
508         }
469             ...
510         if (!db && !odbc_dbh && !psql_dbh) {
511             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Failure to connect to %s %s!\n",
↪ switch_cache_db_type_name(type), db_name);
512             goto end;
513         }
514
515         new_dbh = create_handle(type);
516
517         switch_log_printf(SWITCH_CHANNEL_ID_LOG, file, func, line, NULL, SWITCH_LOG_DEBUG10,
518             "Create Cached DB handle %s [%s] %s:%d\n", new_dbh->name,
↪ switch_cache_db_type_name(type), file, line);
519
520         if (db) {
521             new_dbh->native_handle.core_db_dbh = db;
522         } else if (odbc_dbh) {
523             new_dbh->native_handle.odbc_dbh = odbc_dbh;
524         } else {
525             new_dbh->native_handle.psql_dbh = psql_dbh;
526         }
527
528         add_handle(new_dbh, db_str, db_callsite_str, thread_str);

```

依函数调用顺序返回，最终 `switch_core_db_handle(&sql_manager.dbh) = SWITCH_STATUS_SUCCESS`，至此，启动数据库操作已经完成。接下来我们可以愉快的对数据库进行操作了。

接下来就是对数据库进行初始化操作，L3395-L3425 为删除数据库内原有的表或视图，当数据库类型为默认 DB 时，也会使用 **PRAGMA** 设置一些环境变量和状态标识（L3418-L3422）。

```

3395     switch (sql_manager.dbh->type) {
3396     case SCDB_TYPE_PGSQL:
3397     case SCDB_TYPE_ODBC:
3398         if (switch_test_flag(&runtime), SCF_CLEAR_SQL) {
3399             char sql[512] = "";
3400             char *tables[] = { "channels", "calls", "tasks", NULL };
3401             int i;
3402             const char *hostname = switch_core_get_switchname();
3403
3404             for (i = 0; tables[i]; i++) {
3405                 switch_snprintfv(sql, sizeof(sql), "delete from %q where hostname='%q'", tables[i],
↵ hostname);
3406                 switch_cache_db_execute_sql(sql_manager.dbh, sql, NULL);
3407             }
3408         }
3409         break;
3410     case SCDB_TYPE_CORE_DB:
3411     {
3412         switch_cache_db_execute_sql(sql_manager.dbh, "drop table channels", NULL);
3413         switch_cache_db_execute_sql(sql_manager.dbh, "drop table calls", NULL);
3414         switch_cache_db_execute_sql(sql_manager.dbh, "drop view detailed_calls", NULL);
3415         switch_cache_db_execute_sql(sql_manager.dbh, "drop view basic_calls", NULL);
3416         switch_cache_db_execute_sql(sql_manager.dbh, "drop table interfaces", NULL);
3417         switch_cache_db_execute_sql(sql_manager.dbh, "drop table tasks", NULL);
3418         switch_cache_db_execute_sql(sql_manager.dbh, "PRAGMA synchronous=OFF;", NULL);
3419         switch_cache_db_execute_sql(sql_manager.dbh, "PRAGMA count_changes=OFF;", NULL);
3420         switch_cache_db_execute_sql(sql_manager.dbh, "PRAGMA default_cache_size=8000", NULL);
3421         switch_cache_db_execute_sql(sql_manager.dbh, "PRAGMA temp_store=MEMORY;", NULL);
3422         switch_cache_db_execute_sql(sql_manager.dbh, "PRAGMA journal_mode=OFF;", NULL);
3423     }
3424     break;
3425 }

```

其中 `switch_cache_db_execute_sql()` 为 FreeSWITCH 执行 `sql` 语句关键函数，查看函数调用关系可以清楚的看出 FreeSWITCH 执行 `sql` 的详细步骤和处理方式，由于篇幅限制，这里就不过多讲解，感兴趣的话可自行研究。

接下来会对 `aliases`、`complete`、`nat`、`registrations`、`recovery` 等表进行相应的检查(L3427-L3436)。若第一条 `sql` 执行失败后，说明表里没有想要操作的字段，则会执行后续的第二条和第三条 `sql` 语句；反之，若第一条 `sql` 执行成功后，则大家相安无事，继续往下走。我们后面也会解释 `switch_cache_db_test_reactive()` 函数的作用。

L3437-L3440 则是对 `recovery` 表的不同列进行创建独立的索引

```

3427     switch_cache_db_test_reactive(sql_manager.dbh, "select hostname from aliases", "DROP TABLE
↪ aliases", create_alias_sql);
3428     switch_cache_db_test_reactive(sql_manager.dbh, "select hostname from complete", "DROP TABLE
↪ complete", create_complete_sql);
3429     switch_cache_db_test_reactive(sql_manager.dbh, "select hostname from nat", "DROP TABLE nat",
↪ create_nat_sql);
3430     switch_cache_db_test_reactive(sql_manager.dbh, "delete from registrations where reg_user=''",
3431                                   "DROP TABLE registrations", create_registrations_sql);
3432
3433     switch_cache_db_test_reactive(sql_manager.dbh, "select metadata from registrations", NULL, "ALTER
↪ TABLE registrations ADD COLUMN metadata VARCHAR(256)");
3434
3435
3436     switch_cache_db_test_reactive(sql_manager.dbh, "select hostname from recovery", "DROP TABLE
↪ recovery", recovery_sql);
3437     switch_cache_db_create_schema(sql_manager.dbh, "create index recovery1 on recovery(technology)",
↪ NULL);
3438     switch_cache_db_create_schema(sql_manager.dbh, "create index recovery2 on recovery(profile_name)",
↪ NULL);
3439     switch_cache_db_create_schema(sql_manager.dbh, "create index recovery3 on recovery(uuid)", NULL);
3440     switch_cache_db_create_schema(sql_manager.dbh, "create index recovery3 on recovery(runtime_uuid)",
↪ NULL);

```

`switch_cache_db_test_reactive()` 主要是对数据库内的表进行检查操作，达到预期的目的。在执行 `test_sql` 之前会先对 `runtime` 的 `flag` 进行判断，若为 `SCF_CLEAR_SQL` 则直接返回 `SWITCH_TRUE` (L1311-L1313)；若为 `SCF_AUTO_SCHEMAS`，则不管 `test_sql` 成功与否，都不会执行后面的 `drop_sql` 和 `reactive_sql` (L1315-L1319)。若两个 `flag` 都没有，则会进行下面的操作(L1323-L1378)：执行 `test_sql`，如果失败则会执行后面的 `drop_sql` 和 `reactive_sql`，如果成功则直接返回 `SWITCH_TRUE`。

```

SWITCH_DECLARE(switch_bool_t) switch_cache_db_test_reactive(switch_cache_db_handle_t *dbh,
                                                            const char *test_sql, const char *drop_sql,
                                                            ↪ const char *reactive_sql)
...
1311     if (!switch_test_flag((&runtime), SCF_CLEAR_SQL)) {
1312         return SWITCH_TRUE;
1313     }
...
1315     if (!switch_test_flag((&runtime), SCF_AUTO_SCHEMAS)) {
1316         switch_status_t status = switch_cache_db_execute_sql(dbh, (char *)test_sql, NULL);
1317
1318         return (status == SWITCH_STATUS_SUCCESS) ? SWITCH_TRUE : SWITCH_FALSE;
1319     }
...

```

```

1321     if (io_mutex) switch_mutex_lock(io_mutex);
...
1323     switch (dbh->type){
...
1324         case SCDB_TYPE_PGSQL:
...
1334         case SCDB_TYPE_ODBC:
...
1344         case SCDB_TYPE_CORE_DB:
...
    }
1376     if (io_mutex) switch_mutex_unlock(io_mutex);

```

前面我们说到如果 `switch_cache_db_test_reactive()` 在遇到 `flag` 为 `SCF_CLEAR_SQL` 时直接返回 `SWITCH_TRUE`。是因为当 `flag` 为 `SCF_CLEAR_SQL` 时这里会独自为 `complete`、`aliases`、`nat` 表进行操作。

```

3560     if (switch_test_flag((&runtime), SCF_CLEAR_SQL)) {
3561         char sql[512] = "";
3562         char *tables[] = { "complete", "aliases", "nat", NULL };
3563         int i;
3564         const char *hostname = switch_core_get_hostname();
3565
3566         for (i = 0; tables[i]; i++) {
3567             switch_snprintfv(sql, sizeof(sql), "delete from %q where sticky=0 and hostname='%q'",
↪ tables[i], hostname);
3568             switch_cache_db_execute_sql(sql_manager.dbh, sql, NULL);
3569         }
3570
3571         switch_snprintfv(sql, sizeof(sql), "delete from interfaces where hostname='%q'", hostname);
3572         switch_cache_db_execute_sql(sql_manager.dbh, sql, NULL);
3573     }

```

L3601-L3638 会绑定一些事件，并回调 `core_event_handler()` 函数，而 `core_event_handler()` 函数会根据回调时事件类型进行相应的数据库操作。由此可见，**FreeSWITCH 是通过事件 EVENT 来对数据库进行操作的。**

```

3601     if (sql_manager.manage) {
3602 #ifdef SWITCH_SQL_BIND_EVERY_EVENT
3603         switch_event_bind("core_db", SWITCH_EVENT_ALL, SWITCH_EVENT_SUBCLASS_ANY, core_event_handler,
↪ NULL);
3604 #else

```

```
3605      switch_event_bind("core_db", SWITCH_EVENT_ADD_SCHEDULE, SWITCH_EVENT_SUBCLASS_ANY,  
↪      core_event_handler, NULL);  
...
```

随后会起一个名为 **CORE** 的 **sql** 队列，FreeSWITCH 成功启动后的关于数据库的操作，均是基于该队列进行。

```
static void switch_core_sqldb_start_thread(void){  
...  
3691      switch_sql_queue_manager_init_name("CORE",  
3692                                         &sql_manager.qm,  
3693                                         4,  
3694                                         dbname,  
3695                                         SWITCH_MAX_TRANS,  
3696                                         runtime.core_db_pre_trans_execute,  
3697                                         runtime.core_db_post_trans_execute,  
3698                                         runtime.core_db_inner_pre_trans_execute,  
3699                                         runtime.core_db_inner_post_trans_execute);  
3700  
3701  }  
3702  switch_sql_queue_manager_start(sql_manager.qm);  
...  
}
```

随后会回调 **switch_core_sql_db_thread()** 函数在 **sql_manager.memory_pool** 内申请启动一个线程。

```
3636      switch_thread_create(&sql_manager.db_thread, thd_attr, switch_core_sql_db_thread, NULL,  
↪      sql_manager.memory_pool);
```

至此，数据库启动、初始化的操作已经完成，释放 **sql_manager.dbh**，后续的数据库操作均会在 **sql_queue** 里面进行。

除了 **start**，还有 **pause**、**resume**、**stop** 等操作，分别对应以下函数：

```
3645 SWITCH_DECLARE(void) switch_core_sqldb_pause(void)  
  
3653 SWITCH_DECLARE(void) switch_core_sqldb_resume(void)  
  
3709 void switch_core_sqldb_stop(void)
```

下面这个函数是为了提供有关数据库连接池状态的一些反馈。比如你在 FreeSWITCH 前台输入 `db_cache status` 就是调用的该函数。

```
3726 SWITCH_DECLARE(void) switch_cache_db_status(switch_stream_handle_t *stream)
```

`db_cache status` 的返回如下所示：

```
db="core",type="core_db"
  Type: CORE_DB
  Last used: 0
  Total used: 8
  Flags: Unlocked, Detached(0)
  Creator: src/switch_console.c:725
  Last User: src/switch_console.c:255
db="json",type="core_db"
  Type: CORE_DB
  Last used: 16
  Total used: 1
  Flags: Unlocked, Detached(0)
  Creator: mod_verto.c:5633
  Last User:
db="sofia_reg_internal",type="core_db"
  Type: CORE_DB
  Last used: 0
  Total used: 41
  Flags: Unlocked, Detached(0)
  Creator: sofia_glue.c:2500
  Last User: sofia_glue.c:2536
db="sofia_reg_external",type="core_db"
  Type: CORE_DB
  Last used: 0
  Total used: 36
  Flags: Unlocked, Detached(0)
  Creator: sofia_glue.c:2500
  Last User: sofia_glue.c:2536
db="sofia_reg_internal-ipv6",type="core_db"
  Type: CORE_DB
  Last used: 0
  Total used: 41
  Flags: Unlocked, Detached(0)
  Creator: sofia_glue.c:2500
  Last User: sofia_glue.c:2536
...
```

以下为执行 `sql` 语句相关的函数，函数调用关系由上至下，可以看出在执行 `sql` 语句时的操作顺序。

```

SWITCH_DECLARE(switch_status_t) switch_cache_db_execute_sql(switch_cache_db_handle_t *dbh, char *sql, char
↳ **err)

static switch_status_t switch_cache_db_execute_sql_chunked(switch_cache_db_handle_t *dbh, char *sql,
↳ uint32_t chunk_size, char **err)

static switch_status_t switch_cache_db_execute_sql_real(switch_cache_db_handle_t *dbh, const char *sql, char
↳ **err) {
...
switch_pgsql_handle_exec(dbh->native_handle.pgsql_dbh, sql, &errmsg) {

switch_pgsql_handle_exec_detailed(__FILE__, (char *)__SWITCH_FUNC__, __LINE__, handle, sql, err)

}
...
switch_odbc_handle_exec(dbh->native_handle.odbc_dbh, sql, NULL, &errmsg);
...
switch_core_db_exec(dbh->native_handle.core_db_dbh, sql, NULL, NULL, &errmsg);
}

```

4.27 switch_ivr.c

王浩星

本章主要介绍了一些 FreeSWITCH ivr 功能函数。

`switch_ivr_sound_test`，该函数用于测试通道音频播放情况。`switch_core_session_get_read_impl` 用于获取 session 编解码器回调表，编解码器初始化，channel 准备好之后读取帧内容，若读到的帧为空包或者是 SFF_CNG 结束该帧，读取下一帧，打印相关的音频包信息（L83）。

```

41 SWITCH_DECLARE(switch_status_t) switch_ivr_sound_test(switch_core_session_t *session)
42 {
...
58 switch_core_session_get_read_impl(session, &imp);
60 period_len = imp.actual_samples_per_second / imp.samples_per_packet;
62 if (switch_core_codec_init(&codec,
63                             "L16",

```



```

64         NULL,
65         NULL,
66         imp.samples_per_second,
67         imp.microseconds_per_packet / 1000,
68         imp.number_of_channels,
69         SWITCH_CODEC_FLAG_ENCODE | SWITCH_CODEC_FLAG_DECODE, NULL,
70         switch_core_session_get_pool(session)) != SWITCH_STATUS_SUCCESS) {
71     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Codec Error L16@%uhz %u
↪ channels %dms\n",
72         imp.samples_per_second, imp.number_of_channels, imp.microseconds_per_packet /
↪ 1000);
73     return SWITCH_STATUS_FALSE;
74 }
75 while (switch_channel_ready(channel)) {
76     status = switch_core_session_read_frame(session, &read_frame, SWITCH_IO_FLAG_NONE, 0);
77     if (switch_test_flag(read_frame, SFF_CNG) || !read_frame->samples) {
78         continue;
79     }
80     ...

```

函数测试用例如下:

```

<extension name="test" continue="false">
  <condition field="destination_number" expression="^88888$">
    <action application="answer"/>
    <action application="sound_test"/>
  </condition>
</extension>

```

函数 `switch_ivr_sleep`，该函数用于 sleep（休眠）一段时间。设置频道标识符为 `CF_VIDEO_BLANK`，获取 session 编解码器回调表,媒体状态判定,频道标识符为 `CF_BREAK`,跳到代码 `end`，及结束本次 sleep，具体的 sleep 时间计算在 for（L224）循环中实现。

```

127 SWITCH_DECLARE(switch_status_t) switch_ivr_sleep(switch_core_session_t *session, uint32_t ms,
↪ switch_bool_t sync, switch_input_args_t *args)
128 {
129     ...
130     switch_core_session_get_read_impl(session, &imp);
131     for (;;) {
132         now = switch_micro_time_now();
133         elapsed = (int32_t) ((now - start) / 1000);
134         left = ms - elapsed;

```

```

228
229         if (!switch_channel_ready(channel)) {
230             status = SWITCH_STATUS_FALSE;
231             break;
232         }
233
234         if (switch_channel_test_flag(channel, CF_BREAK)) {
235             switch_channel_clear_flag(channel, CF_BREAK);
236             status = SWITCH_STATUS_BREAK;
237             break;
238         }
239
240         if (now > done || left <= 0) {
241             break;
242         }
243
244
245         switch_ivr_parse_all_events(session);
...

```

函数测试用例如下:

```

<extension name="test" continue="false">
  <condition field="destination_number" expression="^88888$">
    <action application="answer"/>
    <action application="sleep" data="800000"/>
    <action application="echo" />
  </condition>
</extension>

```

单线程通过调用内核函数 (L346) `switch_core_session_write_frame` 对 session 循环写入帧操作, 接收到网络套接字数据或相应 flag 异常, 则跳出循环。

```

330 static void *SWITCH_THREAD_FUNC unicast_thread_run(switch_thread_t *thread, void *obj)
331 {
...
339     while (switch_test_flag(conninfo, SUF_READY) && switch_test_flag(conninfo, SUF_THREAD_RUNNING))
↪ {
340         len = conninfo->write_frame.buflen;
341         if (switch_socket_recv(conninfo->socket, conninfo->write_frame.data, &len) !=
↪ SWITCH_STATUS_SUCCESS || len == 0) {
342             break;
343         }

```

```

344         conninfo->write_frame.datalen = (uint32_t) len;
345         conninfo->write_frame.samples = conninfo->write_frame.datalen / 2;
346         switch_core_session_write_frame(conninfo->session, &conninfo->write_frame,
↪ SWITCH_IO_FLAG_NONE, conninfo->stream_id);
347     }
    ...

```

线程池创建，堆栈大小设置，池内线程的创建。具体功能的实现在线程函数 (L330) `unicast_thread_run` 中。该线程池的调度是在函数 (L934) `switch_ivr_park` 中，稍后会谈及该函数。

```

355 static void unicast_thread_launch(switch_unicast_conninfo_t *conninfo)
356 {
357     switch_threadattr_t *thd_attr = NULL;
358
359     switch_threadattr_create(&thd_attr, switch_core_session_get_pool(conninfo->session));
360     switch_threadattr_stacksize_set(thd_attr, SWITCH_THREAD_STACKSIZE);
361     switch_set_flag_locked(conninfo, SUF_THREAD_RUNNING);
362     switch_thread_create(&conninfo->thread, thd_attr, unicast_thread_run, conninfo,
↪ switch_core_session_get_pool(conninfo->session));
363 }

```

IVR 功能函数，用于关闭单播功能。通过查询通道中的私有数据查询判定，清除标志锁，停止 socket 传输数据，线程的销毁，若线程还在跑，10ms 之后，再销毁读取编码，关闭 socket，清空 flag 等操作。

```

365 SWITCH_DECLARE(switch_status_t) switch_ivr_deactivate_unicast(switch_core_session_t *session);
    {
    ...
375     if ((conninfo = switch_channel_get_private(channel, "unicast"))) {
376         switch_status_t st;
377
378         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Shutting down
↪ unicast connection\n");
379         switch_clear_flag_locked(conninfo, SUF_READY);
380         switch_socket_shutdown(conninfo->socket, SWITCH_SHUTDOWN_READWRITE);
381         switch_thread_join(&st, conninfo->thread);
382
383         while (switch_test_flag(conninfo, SUF_THREAD_RUNNING)) {
384             switch_yield(10000);
385             if (++sanity >= 10000) {
386                 break;

```

```

387         }
388     }
389     if (switch_core_codec_ready(&conninfo->read_codec)) {
390         switch_core_codec_destroy(&conninfo->read_codec);
391     }
392     switch_socket_close(conninfo->socket);
393 }
394 switch_channel_clear_flag(channel, CF_UNICAST);
395 return SWITCH_STATUS_SUCCESS;
396 }

```

IVR 功能函数，用于激活单播功能。获取本地、远端的 ip、端口，初始化编解码 L16(L438)；获取 local、远端网络套接字，创建 socket，绑定地址，设置 channel 私有数据 `unicast`，设置 flag。

```

398 SWITCH_DECLARE(switch_status_t) switch_ivr_activate_unicast(switch_core_session_t *session,
399                                                             char *lo
↳ cal_ip,
400
↳ switch_port_t local_port,
401
↳ char *remote_ip, switch_port_t remote_port, char *transport, char *flags)
402 {
433     switch_mutex_init(&conninfo->flag_mutex, SWITCH_MUTEX_NESTED,
↳ switch_core_session_get_pool(session));
434
435     read_codec = switch_core_session_get_read_codec(session);
436
437     if (!switch_test_flag(conninfo, SUF_NATIVE)) {
438         if (switch_core_codec_init(&conninfo->read_codec,
439                                     "L16",
440                                     NULL,
441                                     NULL,
442                                     read_codec->implementation-
↳ >actual_samples_per_second,
443                                     read_codec->implementation-
↳ >microseconds_per_packet / 1000,
444                                     1, SWITCH_CODEC_FLAG_ENCODE |
↳ SWITCH_CODEC_FLAG_DECODE,
445                                     NULL,
↳ switch_core_session_get_pool(session)) == SWITCH_STATUS_SUCCESS) {
446         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG,
447                           "Raw Codec Activation Success L16@%uhz 1
↳ channel %dms\n",
448                           read_codec->implementation-
↳ >actual_samples_per_second, read_codec->implementation->microseconds_per_packet / 1000);
449     } else {

```

```

450         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "Raw
↳ Codec Activation Failed L1      6@%uhz 1 channel %dms\n",
451                               read_codec->implementation-
↳ >actual_samples_per_second, read_codec->imp      lementation->microseconds_per_packet / 1000);
452         goto fail;
453     }
454 }
455
456     conninfo->write_frame.data = conninfo->write_frame_data;
457     conninfo->write_frame buflen = sizeof(conninfo->write_frame_data);
458     conninfo->write_frame.codec = switch_test_flag(conninfo, SUF_NATIVE) ? read_codec : &conninfo-
↳ >read_codec;
    ...

```

IVR 功能函数，用于各事件具体解析过程。获取事件信息,根据哈希表进行相应的事件消息深层获取。**event-lock**、**event-lock-pri** 设置相应的递归标志 flag；**lead-frames** 判断频道媒体准备好并且关键帧存在，调取内核函数 **switch_core_session_read_frame** 读取帧内容；

```

497 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_event(switch_core_session_t *session, switch_event_t
↳ *event){
    ...
534     if (lead_frames && switch_channel_media_ready(channel)) {
535         switch_frame_t *read_frame;
536         int frame_count = atoi(lead_frames);
537         int max_frames = frame_count * 2;
538
539         while (frame_count > 0 && --max_frames > 0) {
540             status = switch_core_session_read_frame(session, &read_frame,
↳ SWITCH_IO_FLAG_NONE, 0);
541             if (!SWITCH_READ_ACCEPTABLE(status)) {
542                 goto done;
543             }
544             if (!switch_test_flag(read_frame, SFF_CNG)) {
545                 frame_count--;
546             }
547         }
548     }
    ...

```

哈希为 **CMD_EXECUTE**, 获取待执行 APP 名、事件的 uuid、**hold-bleg**，对 b 腿播放等待音设置通道变量，执行 API；哈希为 **CMD_UNICAST**, 调用函数(L723) **switch_ivr_activate_unicast** 激活单播功能；哈希为 **CMD_HANGUP**, 调用函数(L763) **switch_channel_hangup** 挂断通话；哈希为 **CMD_NOMEDIA** 无媒体，调用函数(L764) **switch_ivr_nomedia**, 实现对 message 的接收。

```

550     if (cmd_hash == CMD_EXECUTE) {
551         char *app_name = switch_event_get_header(event, "execute-app-name");
552         char *event_uuid = switch_event_get_header(event, "event-uuid");
553         char *app_arg = switch_event_get_header(event, "execute-app-arg");
554         char *content_type = switch_event_get_header(event, "content-type");
555         char *loop_h = switch_event_get_header(event, "loops");
556         char *hold_bleg = switch_event_get_header(event, "hold-bleg");
557         ...
622         for (x = 0; x < loops || loops < 0; x++) {
636             if (switch_core_session_execute_application(session, app_name, app_arg)
↳ != SWITCH_STATUS_SUCCESS) {
637                 if (!inner || switch_channel_test_flag(channel,
↳ CF_STOP_BROADCAST)) switch_channel_clear_flag(channel, CF_BROADCAST);
638                 break;
639             }
640
641             aftr = switch_micro_time_now();
642             if (!switch_channel_ready(channel) || switch_channel_test_flag(channel,
↳ CF_STOP_BROADCAST) || aftr - b4 < 500000) {
643                 break;
644             }
645         }
646         ...
699     } else if (cmd_hash == CMD_UNICAST) {
700         ...
723         switch_ivr_activate_unicast(session, local_ip, (switch_port_t) atoi(local_port),
↳ remote_ip, (switch_port_t) atoi(remote_port), transport, flags);
725     } else if (cmd_hash == CMD_XFEREXT) {

```

解析单个事件 (L497、L786)、解析 message (L832、L803)、解析信号数据 (L854、L890)。

```

497 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_event(switch_core_session_t *session, switch_event_t
↳ *event)
786 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_next_event(switch_core_session_t *session)
803 SWITCH_DECLARE(switch_status_t) switch_ivr_process_indications(switch_core_session_t *session,
↳ switch_core_session_message_t *message)
832 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_all_messages(switch_core_session_t *session)
854 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_signal_data(switch_core_session_t *session,
↳ switch_bool_t all, switch_bool_t only_session_thread)
890 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_all_signal_data(switch_core_session_t *session)

```

以上函数均为 `switch_ivr_parse_all_events` 函数的功能点函数，主要用于解析所有的事件。

```
898 SWITCH_DECLARE(switch_status_t) switch_ivr_parse_all_events(switch_core_session_t *session)
```

函数 `switch_ivr_park`，该函数用于挂起功能。在某些特殊场景下，会使用该功能。

设置频道变量 `CF_PARK`，频道挂起事件的创建（L992），初始化 L16 编解码器（L1007），内核读取帧信息（L1039） `SWITCH_IO_FLAG_NONE`，对会话中的私有队列数据进行获取解析（L1045-L1047），内核调用写入帧（1058） `SWITCH_IO_FLAG_NONE`，调用单线程函数（L1079） `unicast_thread_launch` 创建线程池，创建单线程循环写入帧，当接收到网络套接字跳出循环，解锁。（L330）。解析所有的事件（L1138），对不同的事件进行处理。结束时对频道标记的清理，单通道线程的关闭（L1190-L1212）。

```
934 SWITCH_DECLARE(switch_status_t) switch_ivr_park(switch_core_session_t *session, switch_input_args_t
↪ *args)
935 {
    ...
992     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_PARK) == SWITCH_STATUS_SUCCESS) {
993         switch_channel_event_set_data(channel, event);
994         switch_event_fire(&event);
995     }
996
997     while (switch_channel_ready(channel) && switch_channel_test_flag(channel, CF_CONTROLLED) &&
↪ switch_channel_test_flag(channel, CF_PARK)) {
1073         if (!conninfo) {
1074             if (!(conninfo = switch_channel_get_private(channel, "unicast"))) {
1075                 switch_channel_clear_flag(channel, CF_UNICAST);
1076             }
1077
1078             if (conninfo) {
1079                 unicast_thread_launch(conninfo);
1080             }
1081         }
1082
1083         if (conninfo) {
1104             switch_codec_t *read_codec =
↪ switch_core_session_get_read_codec(session);
            ... }
1138         switch_ivr_parse_all_events(session);
1190 end:
    ...
}
```

函数功能测试，可通过 Dialplan 配置如下：

```
<action application="park"/>
```

以下函数用于 dtmf 按键回调、按键统计。

```

1215 SWITCH_DECLARE(switch_status_t) switch_ivr_collect_digits_callback(switch_core_session_t *session,
↪ switch_input_args_t *args, uint32_t digit_timeout,
1216
↪ uint32_t abs_timeout)
1329 SWITCH_DECLARE(switch_status_t) switch_ivr_collect_digits_count(switch_core_session_t *session,
1330     char *buf, switch_size_t buflen,
        switch_size_t maxdigits,
        const char *terminators, char *terminator,
        uint32_t first_timeout, uint32_t digit_timeout,
        uint32_t abs_timeout)

```

统计 DTMF 按键过程，abs、dtmf、eff 等超时设置，超时判定，解析所有的事件，DTMF 循环出队列，进行统计。

```

1449 for (y = 0; y <= maxdigits; y++) {
1450     if (switch_channel_dequeue_dtmf(channel, &dtmf) != SWITCH_STATUS_SUCCESS) {
1451         break;
1452     }
1453
1454     if (!zstr(terminators) && strchr(terminators, dtmf.digit) && terminator != NULL) {
1455         *terminator = dtmf.digit;
1456         switch_safe_free(abuf);
1457         return SWITCH_STATUS_SUCCESS;
1458     }
1459
1460     buf[x++] = dtmf.digit;
1461     buf[x] = '\0';
1462
1463     if (x >= buflen || x >= maxdigits) {
1464         switch_safe_free(abuf);
1465         return SWITCH_STATUS_SUCCESS;
1466     }
1467 }
1468 }

```

函数 `switch_ivr_hold`，该函数用于等待功能，等待的同时播放等待音。设置频道标识 `CF_HOLD`，接收会话消息，获取等待音，获取会话的 uuid 放音，创建频道 hold 事件。

```

1496 SWITCH_DECLARE(switch_status_t) switch_ivr_hold(switch_core_session_t *session, const char *message,
↪ switch_bool_t moh)
{

```

```

1513     if (moh && (stream = switch_channel_get_hold_music(channel))) {
1514         if ((other_uuid = switch_channel_get_partner_uuid(channel))) {
1515             switch_ivr_broadcast(other_uuid, stream, SMF_ECHO_ALEG | SMF_LOOP);
1516         }
1517     }
1519     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_HOLD) == SWITCH_STATUS_SUCCESS) {
1520         switch_channel_event_set_data(channel, event);
1521         switch_event_fire(&event);
1522     }
1523 }

```

函数 `switch_ivr_hold_uuid`，该函数用于根据 uuid，向指定 session，发送 hold 消息。主要调用 `switch_ivr_hold` 函数实现功能。

```

1528 SWITCH_DECLARE(switch_status_t) switch_ivr_hold_uuid(const char *uuid, const char *message,
↪ switch_bool_t moh)
1529 {
1530     switch_core_session_t *session;
1531
1532     if ((session = switch_core_session_locate(uuid))) {
1533         switch_ivr_hold(session, message, moh);
1534         switch_core_session_rwlock(session);
1535     }
1536
1537     return SWITCH_STATUS_SUCCESS;
1538 }

```

函数 `switch_ivr_hold_toggle_uuid`，该函数主要 hold 的开关函数。通过获取频道状态，根据返回状态的，设置等待 or 取消等待。

```

1540 SWITCH_DECLARE(switch_status_t) switch_ivr_hold_toggle_uuid(const char *uuid, const char *message,
↪ switch_bool_t moh)
1541 {
1542     if ((session = switch_core_session_locate(uuid))) {
1543         if ((channel = switch_core_session_get_channel(session))) {
1544             callstate = switch_channel_get_callstate(channel);
1545
1546             if (callstate == CCS_ACTIVE) {
1547                 switch_ivr_hold(session, message, moh);
1548             } else if (callstate == CCS_HELD) {
1549                 switch_ivr_unhold(session);
1550             }
1551         }
1552     }
1553 }

```

函数 `switch_ivr_unhold`，用于设置 unhold。清除频道标识 `CF_HOLD`，创建频道 unhold 事件。

```
1562 SWITCH_DECLARE(switch_status_t) switch_ivr_unhold(switch_core_session_t *session)
{
1576     switch_core_session_receive_message(session, &msg);
1587     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_UNHOLD) == SWITCH_STATUS_SUCCESS) {
1588         switch_channel_event_set_data(channel, event);
1589         switch_event_fire(&event);
1590     }
}
```

函数 `switch_ivr_unhold_uuid`，该函数用于根据 uuid，向指定 session，发送 unhold 消息。

```
1595 SWITCH_DECLARE(switch_status_t) switch_ivr_unhold_uuid(const char *uuid)
1596 {
1597     switch_core_session_t *session;
1598
1599     if ((session = switch_core_session_locate(uuid))) {
1600         switch_ivr_unhold(session);
1601         switch_core_session_rwlock(session);
1602     }
1603
1604     return SWITCH_STATUS_SUCCESS;
1605 }
```

函数 `switch_ivr_3p_media`、`switch_ivr_media`，用于向会话发起信号，要求对其远端进行直接媒体访问。flag (`SMF_REBRIDGE` 用于在媒体模式下重新桥接调用)

```
1611 SWITCH_DECLARE(switch_status_t) switch_ivr_3p_media(const char *uuid, switch_media_flag_t flags)
1721 SWITCH_DECLARE(switch_status_t) switch_ivr_media(const char *uuid, switch_media_flag_t flags)
```

函数 `switch_ivr_3p_nomedia`、`switch_ivr_nomedia`，用于向会话发起请求间接媒体访问的信号，允许它与另一个设备直接交换媒体。flag (`SMF_REBRIDGE` 在 no_media 模式下重新桥接调用)。

```
1821 SWITCH_DECLARE(switch_status_t) switch_ivr_3p_nomedia(const char *uuid, switch_media_flag_t flags)
1927 SWITCH_DECLARE(switch_status_t) switch_ivr_nomedia(const char *uuid, switch_media_flag_t flags)
```

函数 `switch_ivr_bg_media`，用于后媒体协商或者单腿 hold 状态，再次建立连接（rebridge）。该功能通过内部线程 `media_thread_run` 对媒体任务状态的判定，调用以上 4 个函数实现向会话发送信号，重新桥接。

```
2061 SWITCH_DECLARE(void) switch_ivr_bg_media(const char *uuid, switch_media_flag_t flags, switch_bool_t on,
↳ switch_bool_t is3p, uint32_t delay)
2035 static void *SWITCH_THREAD_FUNC media_thread_run(switch_thread_t *thread, void *obj)
```

函数 `switch_ivr_session_transfer`，transfer 的功能函数，实现呼叫转移。可通过 APP `transfer`、API `uuid_transfer` 进行该函数功能的测试。

```
2084 SWITCH_DECLARE(switch_status_t) switch_ivr_session_transfer(switch_core_session_t *session, const char
↳ *extension, const char *dialplan,
2085
↳ const char *context)
```

函数 `switch_ivr_transfer_variable`，通过内核注册，添加 APP `transfer_vars`，用于传递变量。获取一路通话中频道 a 的变量，在该路通话频道 b 中设置。

```
2229 SWITCH_DECLARE(switch_status_t) switch_ivr_transfer_variable(switch_core_session_t *sessa,
↳ switch_core_session_t *sessb, char *var)
```

创建（L2284）、销毁（L2324）一个数字流解析器对象。

```
2284 SWITCH_DECLARE(switch_status_t) switch_ivr_digit_stream_parser_new(switch_memory_pool_t *pool,
↳ switch_ivr_digit_stream_parser_t ** parser)
2324 SWITCH_DECLARE(switch_status_t) switch_ivr_digit_stream_parser_destroy(switch_ivr_digit_stream_parser_t
↳ *parser)
```

创建（L2342）、销毁（L2358）一个数字流对象。

```
2342 SWITCH_DECLARE(switch_status_t) switch_ivr_digit_stream_new(switch_ivr_digit_stream_parser_t *parser,
↳ switch_ivr_digit_stream_t ** stream)
2358 SWITCH_DECLARE(switch_status_t) switch_ivr_digit_stream_destroy(switch_ivr_digit_stream_t ** stream)
```

设置 (L2372) 一个字符串做行为映射。(L2414) 删除一个字符串的行为映射。

```
2372 SWITCH_DECLARE(switch_status_t)
↳ switch_ivr_digit_stream_parser_set_event(switch_ivr_digit_stream_parser_t *parser, char *digits, void
↳ *data)
2414 SWITCH_DECLARE(switch_status_t)
↳ switch_ivr_digit_stream_parser_del_event(switch_ivr_digit_stream_parser_t *parser, char *digits)
```

函数 `switch_ivr_digit_stream_parser_feed`，从数字流中收集数字，事件匹配测试。

```
2429 SWITCH_DECLARE(void *) switch_ivr_digit_stream_parser_feed(switch_ivr_digit_stream_parser_t *parser,
↳ switch_ivr_digit_stream_t *stream, char digit)
```

函数 `switch_ivr_digit_stream_reset`，将收集到的数字流重置为 0.

```
2482 SWITCH_DECLARE(switch_status_t) switch_ivr_digit_stream_reset(switch_ivr_digit_stream_t *stream)
```

函数 `switch_ivr_digit_stream_parser_set_terminator`，设置一个数字字符串终端。当数字流解析器终端被设置后，重置解析器中的最大最小长度。

```
2495 SWITCH_DECLARE(switch_status_t)
↳ switch_ivr_digit_stream_parser_set_terminator(switch_ivr_digit_stream_parser_t *parser, char digit)
```

函数 `switch_ivr_set_xml_profile_data`，设置配置文件 (XML) 中的参数。

```
2510 SWITCH_DECLARE(int) switch_ivr_set_xml_profile_data(switch_xml_t xml, switch_caller_profile_t
↳ *caller_profile, int off)
```

函数 `switch_ivr_set_xml_call_stats`，设置 XML 呼叫调度，添加调度数据。

```
2626 SWITCH_DECLARE(int) switch_ivr_set_xml_call_stats(switch_xml_t xml, switch_core_session_t *session, int
↳ off, switch_media_type_t type)
```

函数 (L2744) `switch_ivr_set_xml_chan_vars`, 通过循环调用函数 `switch_ivr_set_xml_chan_var` 实现 XML 文件中变量的设置。

```
2722 static int switch_ivr_set_xml_chan_var(switch_xml_t xml, const char *var, const char *val, int off)
2744 SWITCH_DECLARE(int) switch_ivr_set_xml_chan_vars(switch_xml_t xml, switch_channel_t *channel, int off)
```

函数 `switch_ivr_generate_xml_cdr`, 生成 XML 格式的数据。

```
2768 SWITCH_DECLARE(switch_status_t) switch_ivr_generate_xml_cdr(switch_core_session_t *session,
↳ switch_xml_t *xml_cdr)
```

函数 `switch_ivr_set_json_profile_data`, 通过调用配置文件, 设置 json 格式的配置文件数据。

```
3117 static void switch_ivr_set_json_profile_data(cJSON *json, switch_caller_profile_t *caller_profile)
```

函数 `switch_ivr_set_json_call_stats`, 创建 cJSON 数组, 向对象中添加 a、b 腿相关元素, 实现相应的数据调用。

```
3138 SWITCH_DECLARE(void) switch_ivr_set_json_call_stats(cJSON *json, switch_core_session_t *session,
↳ switch_media_type_t type)
```

函数 `switch_ivr_set_json_chan_vars`, 循环获取频道变量的 name 和 value, 将其添加到 json 格式的对象中。

```
3210 static void switch_ivr_set_json_chan_vars(cJSON *json, switch_channel_t *channel, switch_bool_t
↳ urlencode)
```

函数 `switch_ivr_generate_json_cdr`, 生成 json 格式对象, 该 json 对象根据传入会话的 uuid, 包含以下相关数据: switchname、呼叫状态、变量、app_log、调度配置文件, 呼叫过程时间戳等。

```
3241 SWITCH_DECLARE(switch_status_t) switch_ivr_generate_json_cdr(switch_core_session_t *session, cJSON
↳ **json_cdr, switch_bool_t urlencode)
```

函数 `switch_ivr_park_session`，用于会话中 FreeSWITCH 的通道挂起。

```
3477 SWITCH_DECLARE(void) switch_ivr_park_session(switch_core_session_t *session)
```

函数 `switch_ivr_delay_echo`，该函数用于延迟一小段时间，读取数据帧，获取网络包，将收到的数据帧内容写入会话中，及实现延迟后的 echo。

```
3485 SWITCH_DECLARE(void) switch_ivr_delay_echo(switch_core_session_t *session, uint32_t delay_ms)
```

函数 `switch_ivr_say`，根据预先录制的声音，播放时间、IP 地址、数字等。设置 language,创建并设置通道事件，通过 `mod_name` 检索 say 接口的注册名称（L3661），调用函数 `say_function` (L3669)向 say 模块传入数据。

```
3573 SWITCH_DECLARE(switch_status_t) switch_ivr_say(switch_core_session_t *session,  
3574 const char *tosay, const char *module_name,  
    const char *say_type, const char *say_method,  
    const char *say_gender, switch_input_args_t *args)
```

函数 `switch_ivr_say`，根据预先录制的声音，播放时间、IP 地址、数字等。设置 language,创建并设置频道事件，通过 `mod_name` 检索 say 接口的注册名称（L3770），调用函数 `say_string_function` (L3778)向 say 模块传入数据。

```
3695 SWITCH_DECLARE(switch_status_t) switch_ivr_say_string(switch_core_session_t *session,  
    const char *lang, const char *ext, const char *tosay,  
    const char *module_name, const char *say_type,  
    const char *say_method, const char *say_gender,  
    char **rstr)
```

函数 `get_prefixed_str`，用于获取配置文件字符，实现内存 cp 到 buffer 中。

```
3802 static const char *get_prefixed_str(char *buffer, size_t buffer_size, const char *prefix, size_t  
↪ prefix_size, const char *str)
```

函数 `switch_ivr_set_user_xml`，设置用户 XML。根据会话获取频道，设置频道中的成员别名、成员名、域名以及相关变量。

```
3827 SWITCH_DECLARE(switch_status_t) switch_ivr_set_user_xml(switch_core_session_t *session, const char
↳ *prefix,
3828 const char *user, const char *domain, switch_xml_t x_user
```

函数（L3887）`switch_ivr_set_user_extended`，设置用户扩展 `user@domain`，返回设置状态。

函数 `switch_ivr_set_user` 返回以上函数（L3887）的设置状态。

```
3882 SWITCH_DECLARE(switch_status_t) switch_ivr_set_user(switch_core_session_t *session, const char *data)
3887 SWITCH_DECLARE(switch_status_t) switch_ivr_set_user_extended(switch_core_session_t *session, const char
↳ *data, switch_event_t *params)
```

函数 `switch_ivr_uuid_exists`、`switch_ivr_uuid_force_exists`，根据 `uuid`，强制退出会话。根据传入的 `uuid` 获取会话，会话解锁，`exit` 置 1。返回强制状态。

```
3932 SWITCH_DECLARE(switch_bool_t) switch_ivr_uuid_exists(const char *uuid)
3945 SWITCH_DECLARE(switch_bool_t) switch_ivr_uuid_force_exists(const char *uuid)
```

函数 `switch_ivr_process_fh`，用于文件句柄的解析。

```
3958 SWITCH_DECLARE(switch_status_t) switch_ivr_process_fh(switch_core_session_t *session, const char *cmd,
↳ switch_file_handle_t *fhp)
```

函数 `switch_ivr_insert_file`，在任意采样点，将一个文件插入另一个文件。

```
4079 SWITCH_DECLARE(switch_status_t) switch_ivr_insert_file(switch_core_session_t *session, const char
↳ *file, const char *insert_file, switch_size_t sample_point)
```

函数 `switch_ivr_create_message_reply`，用于创建消息回应事件。

```
4233 SWITCH_DECLARE(switch_status_t) switch_ivr_create_message_reply(switch_event_t **reply, switch_event_t
↳ *message, const char *new_proto)
```

函数 `switch_ivr_check_presence_mapping`，用于检查可能存在的映射关系。

```
4246 SWITCH_DECLARE(char *) switch_ivr_check_presence_mapping(const char *exten_name, const char
↳ *domain_name)
```

函数 `switch_ivr_kill_uuid`，用于 kill 掉 uuid。通过 uuid 获取会话和频道号，调用函数 `switch_channel_hangup` 向频道中发送挂断原因，解锁返回函数状态。

```
4310 SWITCH_DECLARE(switch_status_t) switch_ivr_kill_uuid(const char *uuid, switch_call_cause_t cause)
```

函数 `switch_ivr_blind_transfer_ack`，用于盲转的响应。获取通道变量 `blind_transfer_ack`，设置消息盲转响应，接收会话消息。

```
4324 SWITCH_DECLARE(switch_status_t) switch_ivr_blind_transfer_ack(switch_core_session_t *session,
↳ switch_bool_t success)
```

4.28 switch_loadable_module.c

贵永东

本章基于 Commit Hash `1681db4`

本文件主要用来处理模块的加载，卸载以及聊天等功能。

L72~L93 定义了一个全局模块表结构，里面存储了所有模块以及将所有模块根据不同的类型进行划分存储。例如一个 `endpoint` 模块，它即会存储在 `module_hash` 表中，也会存储在 `endpoint_hash` 表中。

```
72 struct switch_loadable_module_container {
73     switch_hash_t *module_hash;
74     switch_hash_t *endpoint_hash;
75     switch_hash_t *codec_hash;
76     switch_hash_t *dialplan_hash;
77     switch_hash_t *timer_hash;
78     switch_hash_t *application_hash;
79     switch_hash_t *chat_application_hash;
80     switch_hash_t *api_hash;
```



```

81     switch_hash_t *json_api_hash;
82     switch_hash_t *file_hash;
83     switch_hash_t *speech_hash;
84     switch_hash_t *asr_hash;
85     switch_hash_t *directory_hash;
86     switch_hash_t *chat_hash;
87     switch_hash_t *say_hash;
88     switch_hash_t *management_hash;
89     switch_hash_t *limit_hash;
90     switch_hash_t *secondary_recover_hash;
91     switch_mutex_t *mutex;
92     switch_memory_pool_t *pool;
93 };

```

启动 FreeSWITCH 时, 会先调用 `switch_loadable_module_init` 函数加载配置文件 `modules.conf` 和 `post_load_modules.conf` 的模块。

L1856~L1873, 对全局模块存储哈希表进行初始化。

L1887~L1918, 读取配置文件 `modules.conf`, 并加载配置文件中的模块。

L1920~1946, 读取配置文件 `post_load_modules.conf`, 并加载配置文件中的模块。

L1948~L1981, 判断 `modules.conf` 和 `post_load_modules.conf` 是否全部加载失败, 如果加载失败, 则加载 `mod_dir` 文件夹中的所有的 `.so` 文件, Windows 平台为 `.dll`, Mac 平台为 `.dylib`, 通常 `mod_dir` 文件夹为 `/usr/local/freeswitch/mod`。

L1983, 假如模块定义了 `runtime` 函数, 则启动一个新线程执行模块的 `runtime` 函数。

L1990, 启动聊天线程。

```

1823 SWITCH_DECLARE(switch_status_t) switch_loadable_module_init(switch_bool_t autoload)
1824 {
    ...

1856     switch_core_hash_init(&loadable_modules.module_hash);
1857     switch_core_hash_init_nocase(&loadable_modules.endpoint_hash);
1858     switch_core_hash_init_nocase(&loadable_modules.codec_hash);
1859     switch_core_hash_init_nocase(&loadable_modules.timer_hash);
1860     switch_core_hash_init_nocase(&loadable_modules.application_hash);
1861     switch_core_hash_init_nocase(&loadable_modules.chat_application_hash);
1862     switch_core_hash_init_nocase(&loadable_modules.api_hash);
1863     switch_core_hash_init_nocase(&loadable_modules.json_api_hash);
1864     switch_core_hash_init(&loadable_modules.file_hash);
1865     switch_core_hash_init_nocase(&loadable_modules.speech_hash);
1866     switch_core_hash_init_nocase(&loadable_modules.asr_hash);

```

```

1867     switch_core_hash_init_nocase(&loadable_modules.directory_hash);
1868     switch_core_hash_init_nocase(&loadable_modules.chat_hash);
1869     switch_core_hash_init_nocase(&loadable_modules.say_hash);
1870     switch_core_hash_init_nocase(&loadable_modules.management_hash);
1871     switch_core_hash_init_nocase(&loadable_modules.limit_hash);
1872     switch_core_hash_init_nocase(&loadable_modules.dialplan_hash);
1873     switch_core_hash_init(&loadable_modules.secondary_recover_hash);

    ...

1887     if ((xml = switch_xml_open_cfg(cf, &cfg, NULL))) {
1888         switch_xml_t mods, ld;
1889         if ((mods = switch_xml_child(cfg, "modules"))) {
1890             for (ld = switch_xml_child(mods, "load"); ld; ld = ld->next) {
1891                 switch_bool_t global = SWITCH_FALSE;
1892                 const char *val = switch_xml_attr_soft(ld, "module");
1893                 const char *path = switch_xml_attr_soft(ld, "path");
1894                 const char *critical = switch_xml_attr_soft(ld, "critical");
1895                 const char *sglobal = switch_xml_attr_soft(ld, "global");
1896                 if (zstr(val) || (strchr(val, '.') && !strstr(val, ext) && !strstr(val,
↵ EXT))) {
1897                     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE,
↵ "Invalid extension for %s\n", val);
1898                     continue;
1899                 }
1900                 global = switch_true(sglobal);
1901
1902                 if (path && zstr(path)) {
1903                     path = SWITCH_GLOBAL_dirs.mod_dir;
1904                 }
1905                 if (switch_loadable_module_load_module_ex(path, val, SWITCH_FALSE,
↵ global, &err) == SWITCH_STATUS_GENERR) {
1906                     if (critical && switch_true(critical)) {
1907                         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT,
↵ "Failed to load critical module '%s', abort()\n", val);
1908                         abort();
1909                     }
1910                 }
1911                 count++;
1912             }
1913         }
1914         switch_xml_free(xml);
1915
1916     } else {
1917         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "open of %s failed\n", cf);
1918     }
1919
1920     if ((xml = switch_xml_open_cfg(pcf, &cfg, NULL))) {
1921         switch_xml_t mods, ld;

```

```

1922
1923         if ((mods = switch_xml_child(cfg, "modules"))) {
1924             for (ld = switch_xml_child(mods, "load"); ld; ld = ld->next) {
1925                 switch_bool_t global = SWITCH_FALSE;
1926                 const char *val = switch_xml_attr_soft(ld, "module");
1927                 const char *path = switch_xml_attr_soft(ld, "path");
1928                 const char *sglobal = switch_xml_attr_soft(ld, "global");
1929                 if (zstr(val) || (strchr(val, '.') && !strstr(val, ext) && !strstr(val,
↳ EXT))) {
1930                     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE,
↳ "Invalid extension for %s\n", val);
1931                     continue;
1932                 }
1933                 global = switch_true(sglobal);
1934
1935                 if (path && zstr(path)) {
1936                     path = SWITCH_GLOBAL_dirs.mod_dir;
1937                 }
1938                 switch_loadable_module_load_module_ex(path, val, SWITCH_FALSE, global,
↳ &err);
1939                 count++;
1940             }
1941         }
1942         switch_xml_free(xml);
1943
1944     } else {
1945         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "open of %s failed\n", pcf);
1946     }
1947
1948     if (!count) {
1949         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "No modules loaded, assuming
↳ 'load all'\n");
1950         all = 1;
1951     }
1952
1953     if (all) {
1954         if (apr_dir_open(&module_dir_handle, SWITCH_GLOBAL_dirs.mod_dir, loadable_modules.pool)
↳ != APR_SUCCESS) {
1955             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Can't open
↳ directory: %s\n", SWITCH_GLOBAL_dirs.mod_dir);
1956             return SWITCH_STATUS_GENERR;
1957         }
1958
1959         while (apr_dir_read(&finfo, finfo_flags, module_dir_handle) == APR_SUCCESS) {
1960             const char *fname = finfo.fname;
1961
1962             if (finfo.filetype != APR_REG) {
1963                 continue;
1964             }

```

```

1965
1966         if (!fname) {
1967             fname = finfo.name;
1968         }
1969
1970         if (!fname) {
1971             continue;
1972         }
1973
1974         if (zstr(fname) || (!strstr(fname, ext) && !strstr(fname, EXT))) {
1975             continue;
1976         }
1977
1978         switch_loadable_module_load_module(SWITCH_GLOBAL_dirs.mod_dir, fname,
↪ SWITCH_FALSE, &err);
1979     }
1980     apr_dir_close(module_dir_handle);
1981 }
1982
1983 switch_loadable_module_runtime();
1984
1985 memset(&chat_globals, 0, sizeof(chat_globals));
1986 chat_globals.running = 1;
1987 chat_globals.pool = loadable_modules.pool;
1988 switch_mutex_init(&chat_globals.mutex, SWITCH_MUTEX_NESTED, chat_globals.pool);
1989
1990 chat_thread_start(1);
1991
1992 return SWITCH_STATUS_SUCCESS;
1993 }

```

上述 L1905、L1938、L1978 都是直接或间接调用 `switch_loadable_module_load_module_ex` 函数加载模块。

L1591, 判断全局模块表中是否已经加载此模块, 如果已经加载, 则进行相应的提示日志输出。

L1595, 调用 `switch_loadable_module_load_file` 读取模块文件。

L1596, 调用 `switch_loadable_module_process` 函数解析模块。

L1597~L1599, 判断 `runtime` 是否是 `True`, 并且模块定义了 `runtime` 函数, 则启动新的线程执行模块的 `runtime` 函数。

```

1552 static switch_status_t switch_loadable_module_load_module_ex(const char *dir, const char *fname,
↪ switch_bool_t runtime, switch_bool_t global, const char **err)
1553 {

```

```

...

1591     if (switch_core_hash_find_locked(loadable_modules.module_hash, file, loadable_modules.mutex)) {
1592         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Module %s Already
↳ Loaded!\n", file);
1593         *err = "Module already loaded";
1594         status = SWITCH_STATUS_FALSE;
1595     } else if ((status = switch_loadable_module_load_file(path, file, global, &new_module)) ==
↳ SWITCH_STATUS_SUCCESS) {
1596         if ((status = switch_loadable_module_process(file, new_module)) ==
↳ SWITCH_STATUS_SUCCESS && runtime) {
1597             if (new_module->switch_module_runtime) {
1598                 new_module->thread =
↳ switch_core_launch_thread(switch_loadable_module_exec, new_module, new_module->pool)    ;
1599             }
1600         } else if (status != SWITCH_STATUS_SUCCESS) {
1601             *err = "module load routine returned an error";
1602         }
1603     } else {
1604         *err = "module load file routine returned an error";
1605     }
1606
1607
1608     return status;
1609
1610 }

```

L1454~L1456, 加载模块动态库, 获取到模块的 **load**、**shutdown**、**runtime** 函数。

L1477~L1493, 执行模块的 **load** 函数, 如果执行失败, 则模块加载失败。

L1527~L1539, 对 **module** 进行赋值。

```

1403 static switch_status_t switch_loadable_module_load_file(char *path, char *filename, switch_bool_t
↳ global, switch_loadable_module_t **new_    module)
1404 {

1448     while (loading) {
        ...

1453
1454         if (!interface_struct_handle) {
1455             interface_struct_handle = switch_dso_data_sym(dso, struct_name, &derr);
1456         }
1457
        ...

1476

```

```

1477         if (interface_struct_handle) {
1478             mod_interface_functions = interface_struct_handle;
1479             load_func_ptr = mod_interface_functions->load;
1480         }
1481
1482         if (load_func_ptr == NULL) {
1483             err = "Cannot locate symbol 'switch_module_load' please make sure this is a
↵ valid module.";
1484             break;
1485         }
1486
1487         status = load_func_ptr(&module_interface, pool);
1488
1489         if (status != SWITCH_STATUS_SUCCESS && status != SWITCH_STATUS_NOUNLOAD) {
1490             err = "Module load routine returned an error";
1491             module_interface = NULL;
1492             break;
1493         }
1494
1495         ...
1510     }
1511     ...
1526
1527     module->pool = pool;
1528     module->filename = switch_core_strdup(module->pool, path);
1529     module->module_interface = module_interface;
1530     module->switch_module_load = load_func_ptr;
1531
1532     if (mod_interface_functions) {
1533         module->switch_module_shutdown = mod_interface_functions->shutdown;
1534         module->switch_module_runtime = mod_interface_functions->runtime;
1535     }
1536
1537     module->lib = dso;
1538
1539     *new_module = module;
1540     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Successfully Loaded [%s]\n",
↵ module_interface->module_name);
1541
1542     switch_core_set_signal_handlers();
1543
1544     return SWITCH_STATUS_SUCCESS;
1545
1546 }

```

L57~L70 定义了 `switch_loadable_module` 结构类型，包含模块的文件名、模块状态、模块的 `runtime`，`shutdown` 函数等。

```
57 struct switch_loadable_module {
58     char *key;
59     char *filename;
60     int perm;
61     switch_loadable_module_interface_t *module_interface;
62     switch_dso_lib_t lib;
63     switch_module_load_t switch_module_load;
64     switch_module_runtime_t switch_module_runtime;
65     switch_module_shutdown_t switch_module_shutdown;
66     switch_memory_pool_t *pool;
67     switch_status_t status;
68     switch_thread_t *thread;
69     switch_bool_t shutting_down;
70 };
```

在 `switch_loadable_module.h` 头文件的 L55~L93 定义了一个模块的结构类型。包含模块名称以及各功能类型链表，如 `application_interface` 链表。因为一个模块，即可以实现一个 APP，也可以实现一个 API 或者 timer。

```
55 struct switch_loadable_module_interface {
56     /*! the name of the module */
57     const char *module_name;
58     /*! the table of endpoints the module has implemented */
59     switch_endpoint_interface_t *endpoint_interface;
60     /*! the table of timers the module has implemented */
61     switch_timer_interface_t *timer_interface;
62     /*! the table of dialplans the module has implemented */
63     switch_dialplan_interface_t *dialplan_interface;
64     /*! the table of codecs the module has implemented */
65     switch_codec_interface_t *codec_interface;
66     /*! the table of applications the module has implemented */
67     switch_application_interface_t *application_interface;
68     /*! the table of chat applications the module has implemented */
69     switch_chat_application_interface_t *chat_application_interface;
70     /*! the table of api functions the module has implemented */
71     switch_api_interface_t *api_interface;
72     /*! the table of json api functions the module has implemented */
73     switch_json_api_interface_t *json_api_interface;
74     /*! the table of file formats the module has implemented */
75     switch_file_interface_t *file_interface;
76     /*! the table of speech interfaces the module has implemented */
77     switch_speech_interface_t *speech_interface;
78     /*! the table of directory interfaces the module has implemented */
79     switch_directory_interface_t *directory_interface;
80     /*! the table of chat interfaces the module has implemented */
```

```

81     switch_chat_interface_t *chat_interface;
82     /*! the table of say interfaces the module has implemented */
83     switch_say_interface_t *say_interface;
84     /*! the table of asr interfaces the module has implemented */
85     switch_asr_interface_t *asr_interface;
86     /*! the table of management interfaces the module has implemented */
87     switch_management_interface_t *management_interface;
88     /*! the table of limit interfaces the module has implemented */
89     switch_limit_interface_t *limit_interface;
90     switch_thread_rwlock_t *rwlock;
91     int refs;
92     switch_memory_pool_t *pool;
93 };

```

L146~L587 对模块进行了解析，总的来说，查看当前模块是否实现了一个 **endpoint** 或者一个 **application** 等，如果实现了，则将这个 **endpoint** 或者 **application** 插入到全局对应的哈希表中，并产生相应的事件。

```

146 static switch_status_t switch_loadable_module_process(char *key, switch_loadable_module_t *new_module)
147 {
148     switch_event_t *event;
149     int added = 0;
150
151     new_module->key = switch_core_strdup(new_module->pool, key);
152
153     switch_mutex_lock(loadable_modules.mutex);
154     switch_core_hash_insert(loadable_modules.module_hash, key, new_module);
155
156     if (new_module->module_interface->endpoint_interface) {
157         const switch_endpoint_interface_t *ptr;
158         for (ptr = new_module->module_interface->endpoint_interface; ptr; ptr = ptr->next) {
159             if (!ptr->interface_name) {
160                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Failed to load
↪ endpoint interface from %s due to no interface name.\n", key);
161             } else {
162                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Adding
↪ Endpoint '%s'\n", ptr->interface_name);
163                 switch_core_hash_insert(loadable_modules.endpoint_hash, ptr->
↪ interface_name, (const void *) ptr);
164                 if (switch_event_create(&event, SWITCH_EVENT_MODULE_LOAD) ==
↪ SWITCH_STATUS_SUCCESS) {
165                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "type", "endpoint");
166                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "name", ptr->interface_name);

```



```

167             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "key", new_module->key);
168             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "filename", new_module->filename);
169             switch_event_fire(&event);
170             added++;
171         }
172     }
173 }
174 }
175
...

284
285     if (new_module->module_interface->application_interface) {
286         const switch_application_interface_t *ptr;
287
288         for (ptr = new_module->module_interface->application_interface; ptr; ptr = ptr->next) {
289             if (!ptr->interface_name) {
290                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Failed to load
↪ application interface from %s due to no interface name.\n", key);
291             } else {
292                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Adding
↪ Application '%s'\n", ptr->interface_name);
293                 if (switch_event_create(&event, SWITCH_EVENT_MODULE_LOAD) ==
↪ SWITCH_STATUS_SUCCESS) {
294                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "type", "application");
295                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "name", ptr->interface_name);
296                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "description", switch_str_nil(ptr->short_desc));
297                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "syntax", switch_str_nil(ptr->syntax));
298                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "key", new_module->key);
299                     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↪ "filename", new_module->filename);
300                     switch_event_fire(&event);
301                     added++;
302                 }
303                 switch_core_hash_insert(loadable_modules.application_hash, ptr-
↪ >interface_name, (const void *) ptr);
304             }
305         }
306     }
...

572
573     if (!added) {
574         if (switch_event_create(&event, SWITCH_EVENT_MODULE_LOAD) == SWITCH_STATUS_SUCCESS) {

```

```

575         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "type", "generic");
576         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "name", new_module-
↳ >key);
577         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "key", new_module-
↳ >key);
578         switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, "filename",
↳ new_module->filename);
579         switch_event_fire(&event);
580         added++;
581     }
582 }
583
584     switch_mutex_unlock(loadable_modules.mutex);
585     return SWITCH_STATUS_SUCCESS;
586
587 }

```

模块加载解析完毕以后，`switch_loadable_module_init` 函数中会调用 `switch_loadable_module_runtime` 函数，去执行模块的 `runtime` 函数。

如果模块定义了 `runtime` 函数，则启动一个线程去执行这个 `runtime` 函数。

```

127 static void switch_loadable_module_runtime(void)
128 {
129     switch_hash_index_t *hi;
130     void *val;
131     switch_loadable_module_t *module;
132
133     switch_mutex_lock(loadable_modules.mutex);
134     for (hi = switch_core_hash_first(loadable_modules.module_hash); hi; hi =
↳ switch_core_hash_next(&hi)) {
135         switch_core_hash_this(hi, NULL, NULL, &val);
136         module = (switch_loadable_module_t *) val;
137
138         if (module->switch_module_runtime) {
139             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Starting runtime
↳ thread for %s\n", module->module_intel_rface->module_name);
140             module->thread = switch_core_launch_thread(switch_loadable_module_exec, module,
↳ loadable_modules.pool);
141         }
142     }
143     switch_mutex_unlock(loadable_modules.mutex);
144 }

```

在新线程中执行模块的 `runtime` 函数。

```

100 static void *SWITCH_THREAD_FUNC switch_loadable_module_exec(switch_thread_t *thread, void *obj)
101 {
    ...
111
112     for (restarts = 0; status != SWITCH_STATUS_TERM && !module->shutting_down; restarts++) {
113         status = module->switch_module_runtime();
114     }
    ...
123     return NULL;
124 }

```

当执行 `load` 这个 API 加载模块时，核心会调用 `switch_loadable_module_load_module` 函数加载模块

```

1547 SWITCH_DECLARE(switch_status_t) switch_loadable_module_load_module(const char *dir, const char *fname,
↪ switch_bool_t runtime, const char **err)
1548 {
1549     return switch_loadable_module_load_module_ex(dir, fname, runtime, SWITCH_FALSE, err);
1550 }

```

卸载模块，从全局模块哈希表中删除模块，并调用 `do_shutdown` 函数执行模块的 `shutdown` 函数。

```

1631 SWITCH_DECLARE(switch_status_t) switch_loadable_module_unload_module(char *dir, char *fname,
↪ switch_bool_t force, const char **err)
1632 {
    ...
1641     if ((module = switch_core_hash_find(loadable_modules.module_hash, fname))) {
1642         if (module->perm) {
1643             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Module is not
↪ unloadable.\n");
1644             *err = "Module is not unloadable";
1645             status = SWITCH_STATUS_NOUNLOAD;
1646             goto unlock;
1647         } else {
1648             /* Prevent anything from using the module while it's shutting down */
1649             switch_core_hash_delete(loadable_modules.module_hash, fname);
1650             switch_mutex_unlock(loadable_modules.mutex);
1651             if ((status = do_shutdown(module, SWITCH_TRUE, SWITCH_TRUE, !force, err)) !=
↪ SWITCH_STATUS_SUCCESS) {
1652                 /* Something went wrong in the module's shutdown function, add it again
↪ */
1653                 switch_core_hash_insert_locked(loadable_modules.module_hash, fname,
↪ module, loadable_modules.mutex);

```

```

1654         }
1655         goto end;
1656     }
1657 } else {
1658     *err = "No such module!";
1659     status = SWITCH_STATUS_FALSE;
1660 }
1661 ...
1668
1669 return status;
1670
1671 }

```

L2001~L2007, 判断模块是否正在使用, 如果正在使用, 模块则无法卸载。

L2011~L2019, 查看模块是否有 `shutdown` 函数, 如果有, 则执行模块的 `shutdown` 函数。并执行 `switch_loadable_module_unprocess` 函数, 删除模块中的 APP、API 等。

L2029~L2032, 如果模块有 `runtime` 函数, 则一直阻塞到模块的 `runtime` 函数执行完。

L2038, 释放模块的内存池。

```

1995 static switch_status_t do_shutdown(switch_loadable_module_t *module, switch_bool_t shutdown,
↪ switch_bool_t unload, switch_bool_t fail_if_busy,
1996                                     const char **err)
1997 {
1998     int32_t flags = switch_core_flags();
1999     switch_assert(module != NULL);
2000
2001     if (fail_if_busy && module->module_interface->rwlock && switch_thread_rwlock_trywrlock(module-
↪ >module_interface->rwlock) != SWITCH_STATUS_SUCCESS) {
2002         if (err) {
2003             *err = "Module in use.";
2004         }
2005         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Module %s is in use, cannot
↪ unload.\n", module->module_interface->module_name);
2006         return SWITCH_STATUS_FALSE;
2007     }
2008
2009     module->shutting_down = SWITCH_TRUE;
2010
2011     if (shutdown) {
2012         switch_loadable_module_unprocess(module);
2013         if (module->switch_module_shutdown) {
2014             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Stopping: %s\n",
↪ module->module_interface->module_name);
2015             module->status = module->switch_module_shutdown();

```

```

2016         } else {
2017             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "%s has no shutdown
↳ routine\n", module->module_interfac    e->module_name);
2018         }
2019     }
2020
2021     if (fail_if_busy && module->module_interface->rwlock) {
2022         switch_thread_rwlock_unlock(module->module_interface->rwlock);
2023     }
2024
2025     if (unload && module->status != SWITCH_STATUS_NUNLOAD && !(flags & SCF_VG)) {
2026         switch_memory_pool_t *pool;
2027         switch_status_t st;
2028
2029         if (module->thread) {
2030             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "%s stopping runtime
↳ thread.\n", module->module_interfa    ce->module_name);
2031             switch_thread_join(&st, module->thread);
2032         }
2033
2034         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "%s unloaded.\n", module-
↳ >module_interface->module_name);
2035         switch_dso_destroy(&module->lib);
2036         if ((pool = module->pool)) {
2037             module = NULL;
2038             switch_core_destroy_memory_pool(&pool);
2039         }
2040     }
2041
2042     return SWITCH_STATUS_SUCCESS;
2043
2044 }

```

此函数为从全局的 `endpoint_hash`、`codec_hash` 等哈希表中删除此模块。

```

928 static switch_status_t switch_loadable_module_unprocess(switch_loadable_module_t *old_module)
929 {
930     switch_event_t *event;
931     int removed = 0;
932
933     switch_mutex_lock(loadable_modules.mutex);
934
935     if (old_module->module_interface->endpoint_interface) {
936         const switch_endpoint_interface_t *ptr;
937
938         for (ptr = old_module->module_interface->endpoint_interface; ptr; ptr = ptr->next) {

```

```

939         if (ptr->interface_name) {
940
941             switch_core_session_hupall_endpoint(ptr, SWITCH_CAUSE_MANAGER_REQUEST);
942             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Write lock
↳ interface '%s' to wait for existing r   eferences.\n",
943                             ptr->interface_name);
944             if (switch_thread_rwlock_trywrlock_timeout(ptr->rwlock, 10) ==
↳ SWITCH_STATUS_SUCCESS) {
945                 switch_thread_rwlock_unlock(ptr->rwlock);
946             } else {
947                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Giving
↳ up on '%s' waiting for existing r   eferences.\n", ptr->interface_name);
948             }
949
950             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Deleting
↳ Endpoint '%s'\n", ptr->interface_name) ;
951             if (switch_event_create(&event, SWITCH_EVENT_MODULE_UNLOAD) ==
↳ SWITCH_STATUS_SUCCESS) {
952                 switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↳ "type", "endpoint");
953                 switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
↳ "name", ptr->interface_name);
954                 switch_event_fire(&event);
955                 removed++;
956             }
957             switch_core_hash_delete(loadable_modules.endpoint_hash, ptr-
↳ >interface_name);
958         }
959     }
960     }
    ...
1400 }

```

查看模块是否已经加载。

```

1612 SWITCH_DECLARE(switch_status_t) switch_loadable_module_exists(const char *mod)
1613 {
1614     switch_status_t status;
1615
1616     if (zstr(mod)) {
1617         return SWITCH_STATUS_FALSE;
1618     }
1619
1620     switch_mutex_lock(loadable_modules.mutex);
1621     if (switch_core_hash_find(loadable_modules.module_hash, mod)) {
1622         status = SWITCH_STATUS_SUCCESS;

```

```
1623     } else {
1624         status = SWITCH_STATUS_FALSE;
1625     }
1626     switch_mutex_unlock(loadable_modules.mutex);
1627
1628     return status;
1629 }
```

根据 `endpoint` 模块名字，从全局 `endpoint_hash` 哈希表中读取一个 `endpoint_interface`。

```
2108 SWITCH_DECLARE(switch_endpoint_interface_t *) switch_loadable_module_get_endpoint_interface(const char
↳ *name)
2109 {
2110     switch_endpoint_interface_t *ptr;
2111
2112     switch_mutex_lock(loadable_modules.mutex);
2113     ptr = switch_core_hash_find(loadable_modules.endpoint_hash, name);
2114     if (ptr) {
2115         PROTECT_INTERFACE(ptr);
2116     }
2117     switch_mutex_unlock(loadable_modules.mutex);
2118
2119
2120     return ptr;
2121 }
```

根据 `file` 模块名字，从全局 `file_hash` 哈希表中读取一个 `file_interface`。

```
2123 SWITCH_DECLARE(switch_file_interface_t *) switch_loadable_module_get_file_interface(const char *name,
↳ const char *modname)
2124 {
2125     switch_file_interface_t *i = NULL;
2126     switch_file_node_t *node, *head;
2127
2128     switch_mutex_lock(loadable_modules.mutex);
2129
2130     if ((head = switch_core_hash_find(loadable_modules.file_hash, name))) {
2131         if (modname) {
2132             for (node = head; node; node = node->next) {
2133                 if (!strcasecmp(node->interface_name, modname)) {
2134                     i = (switch_file_interface_t *) node->ptr;
2135                     break;
2136                 }
2137             }
2138         }
2139     }
```

```

2138         } else {
2139             i = (switch_file_interface_t *) head->ptr;
2140         }
2141     }
2142
2143     switch_mutex_unlock(loadable_modules.mutex);
2144
2145     if (i) PROTECT_INTERFACE(i);
2146
2147     return i;
2148 }

```

根据 `codec` 模块名字，从全局 `codec_hash` 哈希表中读取一个 `codec_interface`。

```

2150 SWITCH_DECLARE(switch_codec_interface_t *) switch_loadable_module_get_codec_interface(const char *name,
↪ const char *modname)
2151 {
2152     switch_codec_interface_t *codec = NULL;
2153     switch_codec_node_t *node, *head;
2154
2155     switch_mutex_lock(loadable_modules.mutex);
2156
2157     if ((head = switch_core_hash_find(loadable_modules.codec_hash, name))) {
2158         if (modname) {
2159             for (node = head; node; node = node->next) {
2160                 if (!strcasecmp(node->interface_name, modname)) {
2161                     codec = (switch_codec_interface_t *) node->ptr;
2162                     break;
2163                 }
2164             }
2165         } else {
2166             codec = (switch_codec_interface_t *) head->ptr;
2167         }
2168     }
2169
2170     switch_mutex_unlock(loadable_modules.mutex);
2171
2172     if (codec) {
2173         PROTECT_INTERFACE(codec);
2174     }
2175
2176     return codec;
2177 }

```

根据 `say` 模块名字，从全局 `say_hash` 哈希表中读取一个 `say_interface`。

```

2201 SWITCH_DECLARE(switch_say_interface_t *) switch_loadable_module_get_say_interface(const char *name)
2202 {
2203     return switch_core_hash_find_locked(loadable_modules.say_hash, name, loadable_modules.mutex);
2204 }

```

L2179~L2186, 定义了根据名字, 从全局各个类型中的哈希表中取出对应的类型。例如 L2190, 表示定义了这样的函数:

```

SWITCH_DECLARE(switch_application_interface_t *) switch_loadable_module_get_application_interface(const char
↳ *name)
{
    switch_application_interface_t *i = NULL;
    if (loadable_modules.application_hash && (i =
↳ switch_core_hash_find_locked(loadable_modules.application_hash, name, loadable_modules.mutex))) {
        PROTECT_INTERFACE(i);
    }
    return i;
}

```

```

2179 #define HASH_FUNC(_kind_) SWITCH_DECLARE(switch_##_kind_##_interface_t *)
↳ switch_loadable_module_get_##_kind_##_interface(const char *name) \
2180 {
↳ \
2181     switch_##_kind_##_interface_t *i = NULL;
↳ \
2182     if (loadable_modules._kind_##_hash && (i =
↳ switch_core_hash_find_locked(loadable_modules._kind_##_hash, name, loadable_modules.mutex))) {
↳ \
2183         PROTECT_INTERFACE(i);
↳ \
2184     }
↳ \
2185     return i;
↳ \
2186 }
2187
2188 HASH_FUNC(dialplan)
2189 HASH_FUNC(timer)
2190 HASH_FUNC(application)
2191 HASH_FUNC(chat_application)
2192 HASH_FUNC(api)
2193 HASH_FUNC(json_api)
2194 HASH_FUNC(speech)

```

```

2195 HASH_FUNC(asr)
2196 HASH_FUNC(directory)
2197 HASH_FUNC(chat)
2198 HASH_FUNC(limit)

```

通常我们加载模块都是通过动态库来加载，FreeSWITCH也提供了通过代码动态增加模块。

L1729~L1733 定义了动态创建模块的函数及函数实现。

```

1729 SWITCH_DECLARE(switch_status_t) switch_loadable_module_build_dynamic(char *filename,
1730
↪ switch_module_load_t switch_module _load,
1731
↪ switch_module_runtime_t switch_mod ule_runtime,
1732
↪ switch_module_shutdown_t switch_mo dule_shutdown, switch_bool_t runtime)
1733 {
    ...
1805 }

```

获取所有的 codec，并返回 codec 的数量。

```

2298 SWITCH_DECLARE(int) switch_loadable_module_get_codecs(const switch_codec_implementation_t **array, int
↪ arraylen)
2299 {
    ...
2343 }

```

根据 api 的名字，从全局 api_hash 表中查询对应的 api_interface，并执行。

```

2532 SWITCH_DECLARE(switch_status_t) switch_api_execute(const char *cmd, const char *arg,
↪ switch_core_session_t *session, switch_stream_handle_t *stream)
2533 {
    ...
2589 }

```

解析 json 数据，并根据数据中的 command 值，从全局 json_api_hash 表中查询对应的 json_api_interface，并执行。

```

2591 SWITCH_DECLARE(switch_status_t) switch_json_api_execute(cJSON *json, switch_core_session_t *session,
↳ cJSON **retval)
2592 {
    ...
2629 }

```

对 `codec` 模块根据 `ptime` 大小进行排序。

```

2232 static void switch_loadable_module_sort_codecs(const switch_codec_implementation_t **array, int
↳ arraylen)
2233 {
    ...
2295 }

```

根据 `codec` 名字, 解析 `bit`、`rate` 等信息, 例如: 如果在 `vars.conf` 中的 `global_codec_prefs` 中定了 `speex@8000h@20i` 编码名, 则此函数会解析出 `8kHz` 以及 `20ms` `ptime` 的 `speex` 编解码类型。

```

2345 SWITCH_DECLARE(char *) switch_parse_codec_buf(char *buf, uint32_t *interval, uint32_t *rate, uint32_t
↳ *bit, uint32_t *channels, char **modname, char **fmt)
2346 {
    ...
2393 }

```

创建一个 `switch_loadable_module_interface_t` 结构类型。

```

2632 SWITCH_DECLARE(switch_loadable_module_interface_t *)
↳ switch_loadable_module_create_module_interface(switch_memory_pool_t *pool, const char *name)
2633 {
    ...
2644 }

```

根据不同的模块类型, 调用不同的宏创建不同的模块, `switch_module_interface_name_t` 为一个枚举类型, 定义了 `SWITCH_ENDPOINT_INTERFACE`、`SWITCH_APPLICATION_INTERFACE` 等类型。

```

2662 SWITCH_DECLARE(void *) switch_loadable_module_create_interface(switch_loadable_module_interface_t *mod,
↳ switch_module_interface_name_t iname)
2663 {

```

```
...
2718 }
```

初始化全局 `msg_queue`，并启动线程去处理聊天消息队列。

```
750 static void chat_thread_start(int idx)
751 {
...
781 }
```

在线程中循环读取全局 `msg_queue`，然后处理消息。

```
730 void *SWITCH_THREAD_FUNC chat_thread_run(switch_thread_t *thread, void *obj)
731 {
...
747 }
```

如果 `event` 事件中未定义 `proto`，则去全局 `chat_hash` 中查询哪些 `chat` 模块名包含 `GLOBAL_`，如果包含，则使用调用此 `chat_application_interface` 的 `chat_send` 进行发送。根据 `event` 事件中的 `dest_proto` 值，去 `chat_hash` 中查询对应的 `chat_application_interface`，并调用 `chat_send` 进行发送。

```
604 static switch_status_t do_chat_send(switch_event_t *message_event)
605
606 {
...
711 }
```

通过 `chat` 模块的名字，查询到 `chat` 模块，并调用这个模块的回调函数，将消息发送出去。

```
820 SWITCH_DECLARE(switch_status_t) switch_core_execute_chat_app(switch_event_t *message, const char *app,
↳ const char *data)
821 {
...
854 }
```

此函数也是将消息发送出去，但是此函数可以定义更多消息参数。

```
858 SWITCH_DECLARE(switch_status_t) switch_core_chat_send_args(const char *dest_proto, const char *proto,
↳ const char *from, const char *to,
859
↳ const char *subject, const char *body, const char *type, const char *h      int, switch_bool_t blocking)
860 {
    ...
897 }
```

根据 `dest_proto`，即 `chat` 模块名，将一个 `event` 消息发送出去。

```
900 SWITCH_DECLARE(switch_status_t) switch_core_chat_send(const char *dest_proto, switch_event_t
↳ *message_event)
901 {
    ...
912 }
```

和 `switch_core_chat_send` 唯一不同的是，前者会 `dup` 出一个 `event`。

```
915 SWITCH_DECLARE(switch_status_t) switch_core_chat_deliver(const char *dest_proto, switch_event_t
↳ **message_event)
916 {
    ...
925 }
```

调用 FreeSWITCH 的 `shutdown` 时，执行此函数。

L2059~L2066。等待聊天线程结束。

L2069~L2085。卸载所有的模块。

L2087~L2105。释放资源。

```
2046 SWITCH_DECLARE(void) switch_loadable_module_shutdown(void)
2047 {
2048     switch_hash_index_t *hi;
2049     void *val;
2050     switch_loadable_module_t *module;
2051     int i;
2052
2053     if (!loadable_modules.module_hash) {
2054         return;
```

```
2055     }
2056
2057     chat_globals.running = 0;
2058
2059     for (i = 0; i < chat_globals.msg_queue_len; i++) {
2060         switch_queue_push(chat_globals.msg_queue[i], NULL);
2061     }
2062
2063     for (i = 0; i < chat_globals.msg_queue_len; i++) {
2064         switch_status_t st;
2065         switch_thread_join(&st, chat_globals.msg_queue_thread[i]);
2066     }
2067
2068
2069     for (hi = switch_core_hash_first(loadable_modules.module_hash); hi; hi =
↵ switch_core_hash_next(&hi)) {
2070         switch_core_hash_this(hi, NULL, NULL, &val);
2071         module = (switch_loadable_module_t *) val;
2072         if (!module->perm) {
2073             do_shutdown(module, SWITCH_TRUE, SWITCH_FALSE, SWITCH_FALSE, NULL);
2074         }
2075     }
2076
2077     switch_yield(1000000);
2078
2079     for (hi = switch_core_hash_first(loadable_modules.module_hash); hi; hi =
↵ switch_core_hash_next(&hi)) {
2080         switch_core_hash_this(hi, NULL, NULL, &val);
2081         module = (switch_loadable_module_t *) val;
2082         if (!module->perm) {
2083             do_shutdown(module, SWITCH_FALSE, SWITCH_TRUE, SWITCH_FALSE, NULL);
2084         }
2085     }
2086
2087     switch_core_hash_destroy(&loadable_modules.module_hash);
2088     switch_core_hash_destroy(&loadable_modules.endpoint_hash);
2089     switch_core_hash_destroy(&loadable_modules.codec_hash);
2090     switch_core_hash_destroy(&loadable_modules.timer_hash);
2091     switch_core_hash_destroy(&loadable_modules.application_hash);
2092     switch_core_hash_destroy(&loadable_modules.chat_application_hash);
2093     switch_core_hash_destroy(&loadable_modules.api_hash);
2094     switch_core_hash_destroy(&loadable_modules.json_api_hash);
2095     switch_core_hash_destroy(&loadable_modules.file_hash);
2096     switch_core_hash_destroy(&loadable_modules.speech_hash);
2097     switch_core_hash_destroy(&loadable_modules.asr_hash);
2098     switch_core_hash_destroy(&loadable_modules.directory_hash);
2099     switch_core_hash_destroy(&loadable_modules.chat_hash);
2100     switch_core_hash_destroy(&loadable_modules.say_hash);
2101     switch_core_hash_destroy(&loadable_modules.management_hash);
```

```
2102     switch_core_hash_destroy(&loadable_modules.limit_hash);
2103     switch_core_hash_destroy(&loadable_modules.dialplan_hash);
2104
2105     switch_core_destroy_memory_pool(&loadable_modules.pool);
2106 }
```

4.29 switch_utils.c

殷鑫博

本章基于 Commit Hash [1681db4](#)。

本文件主要实现了一些兼容性函数和功能性函数。

最开始定义一些参数和结构体，备用。

```
35 #include <switch.h>
36 #ifndef WIN32
37 #include <arpa/inet.h>
38 #if defined(HAVE_SYS_TIME_H) && defined(HAVE_SYS_RESOURCE_H)
39 #include <sys/time.h>
40 #include <sys/resource.h>
41 #endif
42 #endif
43 #include "private/switch_core_pvt.h"
44 #define ESCAPE_META '\\'
45 #ifdef SWITCH_HAVE_GUMBO
46 #include "gumbo.h"
47 #endif
48
49 struct switch_network_node {
50     ip_t ip;
51     ip_t mask;
52     uint32_t bits;
53     int family;
54     switch_bool_t ok;
55     char *token;
56     char *str;
57     struct switch_network_node *next;
58 };
59 typedef struct switch_network_node switch_network_node_t;
60
61 struct switch_network_list {
62     struct switch_network_node *node_head;
63     switch_bool_t default_type;
```

```

64     switch_memory_pool_t *pool;
65     char *name;
66 };

```

如果在 WIN32 系统下会预处理 `switch_inet_pton` 函数，实现将 IPv4 IP 地址由「点分十进制」转换为「二进制整数」。

```

68 #ifndef WIN32
69 SWITCH_DECLARE(int) switch_inet_pton(int af, const char *src, void *dst)
70 {
71     return inet_pton(af, src, dst);
72 }
73 #endif

```

L90 申请一块连续内存，L92 将新申请内存的 flag 设置为 `SFF_DYNAMIC`，意为动态且待释放，L92-L95 规定新内存 `buff` 长度和数据大小，并对数据进行断言。

`switch_zmalloc` 函数使用 `calloc` 进行内存申请，优点是会将所分配的内存空间中的每一位都初始化为零。

```

86 SWITCH_DECLARE(switch_status_t) switch_frame_alloc(switch_frame_t **frame, switch_size_t size)
87 {
88     switch_frame_t *new_frame;
89
90     switch_zmalloc(new_frame, sizeof(*new_frame));
91
92     switch_set_flag(new_frame, SFF_DYNAMIC);
93     new_frame->buflen = (uint32_t)size;
94     new_frame->data = malloc(size);
95     switch_assert(new_frame->data);
96
97     *frame = new_frame;
98
99     return SWITCH_STATUS_SUCCESS;
100 }

```

L120 首先定义一个 `switch_frame_node_t` 的结构体，L123 和 L192 对要操作的线程分别进行加锁和解锁。L125~L144 遍历 `fd` 链表找出一个可用的节点 `np`，并保证链表 `fd` 的连续性。

```

118 static switch_frame_t *find_free_frame(switch_frame_buffer_t *fb, switch_frame_t *orig)
119 {

```

```

120     switch_frame_node_t *np;
121     int x = 0;
122
123     switch_mutex_lock(fb->mutex);
124
125     for (np = fb->head; np; np = np->next) {
126         x++;
127
128         if (!np->inuse && ((orig->packet && np->frame->packet) || (!orig->packet && !np->frame-
↵ >packet))) {
129
130             if (np == fb->head) {
131                 fb->head = np->next;
132             } else if (np->prev) {
133                 np->prev->next = np->next;
134             }
135
136             if (np->next) {
137                 np->next->prev = np->prev;
138             }
139
140             fb->total--;
141             np->prev = np->next = NULL;
142             break;
143         }
144     }

```

L177~L187 将源 `packet` 的数据拷贝到 `np` 内，事先会对 `packet` 进行判断是否为空。

```

177     if (orig->packet) {
178         memcpy(np->frame->packet, orig->packet, orig->packetlen);
179         np->frame->packetlen = orig->packetlen;
180         np->frame->data = ((unsigned char *)np->frame->packet) + 12;
181         np->frame->datalen = orig->datalen;
182     } else {
183         np->frame->packet = NULL;
184         np->frame->packetlen = 0;
185         memcpy(np->frame->data, orig->data, orig->datalen);
186         np->frame->datalen = orig->datalen;
187     }

```

至此已完成寻找一个空的 `np` 节点，并将源 `packet` 内的数据包拷贝至 `np` 节点内。

L209, L210 表示将 `frame->extra_data` 置为闲置状态，并释放掉 `frame->img`，L212 表示将 `fd` 链表总长度加一，L214~L220 表示将释放的 `node` 节点放在 `fd` 链表的最前面。

- 之所以将 **node** 放在 **fd** 链表的最前面，是因为在 **find_free_frame** 函数内，寻找空节点时，是从头遍历 **fd** 链表。这样可以保证在有可用空节点的情况下，能以最小的消耗、最快的速度找到。达到性能提升的效果。

```
209     node->inuse = 0;
210     switch_img_free(&node->frame->img);
    ...
212     fb->total++;
    ...
214     if (fb->head) {
215         fb->head->prev = node;
216     }
217
218     node->next = fb->head;
219     node->prev = NULL;
220     fb->head = node;
```

L272 表示将传来的 **buffer** 结构体置空，L273-L274 表示释放 **buffer** 指向的内存池。

```
272     *fbP = NULL;
273     pool = fb->pool;
274     switch_core_destroy_memory_pool(&pool);
```

L284 表示 **frame buffer** 队列长度默认为 500，但优先选择自定义长度。L286~L290 依次申请内存池，为句柄在内存池内申请一块内存空间，队列、线程互斥锁初始化等操作。

```
286     switch_core_new_memory_pool(&pool);
287     fb = switch_core_alloc(pool, sizeof(*fb));
288     fb->pool = pool;
289     switch_queue_create(&fb->queue, qlen, fb->pool);
290     switch_mutex_init(&fb->mutex, SWITCH_MUTEX_NESTED, pool);
```

L297^{L336} 为根据 **orig** 复制一个新的 **frame** 到 **clone**，其中 L313^{L322} 表示复制数据包与帧数据，L325~L332 表示复制编码格式、**payload map**、原 **img** 等。

```
313     if (orig->packet) {
314         new_frame->packet = malloc(SWITCH_RTP_MAX_BUF_LEN);
315         memcpy(new_frame->packet, orig->packet, orig->packetlen);
```

```

316     new_frame->data = ((unsigned char *)new_frame->packet) + 12;
317 } else {
318     new_frame->packet = NULL;
319     new_frame->data = malloc(new_frame->buflen);
320     switch_assert(new_frame->data);
321     memcpy(new_frame->data, orig->data, orig->datalen);
322 }
323
324
325 new_frame->codec = orig->codec;
326 new_frame->pmap = orig->pmap;
327 new_frame->img = NULL;
328
329
330 if (orig->img && !switch_test_flag(orig, SFF_ENCODED)) {
331     switch_img_copy(orig->img, &new_frame->img);
332 }

```

L390 L432 为查找 `param` 的值。假如第一个参数为 `method=INVITE`，L399 此时 `e` 指向的就是 `=`，L400 L411 存在多个键值对时循环查找，分隔符为 `;`。L405 此时 `ptr` 指向的即 `param` 的值，也就是例子中的 `INVITE`，L415~L419 则为 `param` 的值申请内存空间。

```

...
395 ptr = (char *) str;
396
397 while (ptr) {
398     len = strlen(param);
399     e = ptr+len;
400     next = strchr(ptr, ';');
401
402     if (!strncasecmp(ptr, param, len) && *e == '=') {
403         size_t mlen;
404
405         ptr = ++e;
406
407         if (next) {
408             e = next;
409         } else {
410             e = ptr + strlen(ptr);
411         }
412
413         mlen = (e - ptr) + 1;
414
415         if (pool) {
416             r = switch_core_alloc(pool, mlen);
417         } else {

```

```

418         r = malloc(mlen);
419     }
...

```

验证 IPv4 网络链表的可用性。

```

switch_network_list_validate_ip_token(switch_network_list_t *list, uint32_t ip, const char **token)
switch_network_list_validate_ip_port_token(switch_network_list_t *list, uint32_t ip, int port, const char
↪ **token)
is_port_in_node(int port, switch_network_node_t *node)

```

处理 IPv4 子网掩码。

```

switch_network_list_add_host_mask(switch_network_list_t *list, const char *host, const char *mask_str,
↪ switch_bool_t ok)
switch_network_list_add_host_port_mask(switch_network_list_t *list, const char *host, const char *mask_str,
↪ switch_bool_t ok, switch_network_port_range_p port)

```

处理 IPv4/IPv6 的 CIDR。

```

switch_network_list_add_cidr_token(switch_network_list_t *list, const char *cidr_str, switch_bool_t ok,
↪ const char *token)

switch_network_list_add_cidr_port_token(switch_network_list_t *list, const char *cidr_str, switch_bool_t ok,
↪ const char *token, switch_network_port_range_p port)

switch_network_list_perform_add_cidr_token(switch_network_list_t *list, const char *cidr_str, switch_bool_t
↪ ok, const char *token, switch_network_port_range_p port)

switch_network_ipv4_mapped_ipv6_addr(const char* ip_str)

switch_network_port_range_to_string(switch_network_port_range_p port)

switch_parse_cidr(const char *string, ip_t *ip, ip_t *mask, uint32_t *bitp)

```

截取 open end 之间的字符串，L777~L779 为去空格，而 depth 用作是否继续查找的标识符，L795 e && *e == close 成立则返回 *e（即已截取字符串的值），若不成立则返回 NULL。

```
switch_find_end_paren(const char *s, char open, char close)
...
777     while (s && *s && *s == ' ') {
778         s++;
779     }
...
795     return (e && *e == close) ? (char *) e : NULL;
```

读取一行数据，通常为一个命令，当遇到回车符或换行符时停止读取，将读取内容存到 `buf` 中，返回读取的字符长度。

```
switch_fd_read_line(int fd, char *buf, switch_size_t len)
switch_fd_read_dline(int fd, char **buf, switch_size_t *len)
```

读取文件内容，将读取内容存到 `buf` 中，返回读取的字符长度。

```
switch_fp_read_dline(FILE *fd, char **buf, switch_size_t *len)
```

使用 XML 的转义符进行编码，其中 “`\' & < >`” 依次对应的转义字符分别如下：`"`、`'`、`&`、`<`、`>`。

```
switch_amp_encode(char *s, char *buf, switch_size_t len)
```

使用 `base64` 进行编码，其中 `L999` 为初始化一个长度为 64 个字符的字符集。

```
999 static const char switch_b64_table[65] =
    ↪ "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
switch_b64_encode(unsigned char *in, switch_size_t ilen, unsigned char *out, switch_size_t olen)
```

使用 `base64` 进行解码。

```
switch_b64_decode(const char *in, char *out, switch_size_t olen)
```

基于 `MIME` 的邮件处理函数。

```
switch_simple_email(const char *to,const char *from,const char *headers, const char *body, const char *file,  
↳ const char *convert_cmd, const char *convert_ext)
```

判断是否为局域网地址，L1331~L1332 对入参 `ip` 进行判空。后续若匹配到其中一项，则返回 `SWITCH_TRUE`。

```
switch_is_lan_addr(const char *ip)  
1331     if (zstr(ip))  
1332         return SWITCH_FALSE;
```

字符替换函数，在 `str` 内查找 `from`，替换为 `to`。L1404~L1408 进行查找替换操作。

```
switch_replace_char(char *str, char from, char to, switch_bool_t dup)  
1404     for (; p && *p; p++) {  
1405         if (*p == from) {  
1406             *p = to;  
1407         }  
1408     }
```

根据 ASCII 码，去空格和其他特殊字符函数，其中 `13 10 9 32 11` 分别对应的是归位键、换行键、水平定位符号、空格、垂直定位符号。

```
switch_pool_strip_whitespace(switch_memory_pool_t *pool, const char *str)  
switch_strip_whitespace(const char *str)
```

根据字符匹配去除空格。

```
switch_strip_spaces(char *str, switch_bool_t dup)
```

剥离数字 0-9 和逗号 `,`。

```
switch_strip_commas(char *in, char *out, switch_size_t len)
```

只取数字 0-9、句号 .、减号 -、加号 +，其他全部丢弃。

```
switch_strip_nonnumerics(char *in, char *out, switch_size_t len)
```

去括号。

```
switch_separate_paren_args(char *str)
```

判断字符串是否为纯数字（包括小数）。

```
switch_is_number(const char *str)
switch_is_leading_number(const char *str)
```

分别就 HAVE_GETIFADDRS、linux、WIN32 三种情况下获取子网掩码。

```
#ifdef HAVE_GETIFADDRS
get_netmask(struct sockaddr_in *me, int *mask)
...
#elif defined(__linux__)
get_netmask(struct sockaddr_in *me, int *mask)
...
#elif defined(WIN32)
get_netmask(struct sockaddr_in *me, int *mask)
```

获取本地 IP，L1831_{L1850} 判断网络类型，若是 IPv4 或 IPv6，且在 force_local_ip_v4/force_local_ip_v6 非空情况下，则直接将 IP 地址复制到 buf 内，并根据 IP 地址创建 socket 并连接，最终根据 getnameinfo 函数将 socket 地址转换为 IP 地址存入 buf 内。

```
switch_find_local_ip(char *buf, int len, int *mask, int family)
...
1831     switch (family) {
1832     case AF_INET:
1833         if (force_local_ip_v4) {
1834             switch_copy_string(buf, force_local_ip_v4, len);
1835             switch_safe_free(force_local_ip_v4);
1836             switch_safe_free(force_local_ip_v6);
```

```

1837         return SWITCH_STATUS_SUCCESS;
1838     }
1839     case AF_INET6:
1840         if (force_local_ip_v6) {
1841             switch_copy_string(buf, force_local_ip_v6, len);
1842             switch_safe_free(force_local_ip_v4);
1843             switch_safe_free(force_local_ip_v6);
1844             return SWITCH_STATUS_SUCCESS;
1845         }
1846     default:
1847         switch_safe_free(force_local_ip_v4);
1848         switch_safe_free(force_local_ip_v6);
1849         break;
1850     }
...
1932         switch_copy_string(buf, get_addr(abuf, sizeof(abuf), (struct sockaddr *) &iface_out,
↳ sizeof(struct sockaddr_storage)), len);

```

查找接口的 IP, L2002 将获取到的 IP &((struct sockaddr_in*)(addr->ifa_addr))->sin_addr) 由数值格式转化为点分十进制的 ip 地址格式。

```

switch_find_interface_ip(char *buf, int len, int *mask, const char *ifname, int family)
2002     inet_ntop(AF_INET, &((struct sockaddr_in*)(addr->ifa_addr))->sin_addr), buf, len - 1);

```

将时间转换为 GMT（世界标准时间）格式。L2041 匹配的时间格式为 年-月-日 时:分:秒，L2043 匹配的时间格式为 时:分:秒，L2110~L2115 调用自封装函数进行时间格式转换。

```

switch_str_time(const char *in)
2041     char *pattern = "^((\\d+)-(\\d+)-(\\d+)\\s*(\\d*):{0,1}(\\d*):{0,1}(\\d*))";
2042     char *pattern2 = "^((\\d{4})(\\d{2})(\\d{2})(\\d{2})(\\d{2})(\\d{2}))";
2043     char *pattern3 = "^((\\d*):{0,1}(\\d*):{0,1}(\\d*)$";
...
2110     switch_time_exp_get(&local_time, &tm);
2111     switch_time_exp_lt(&local_tm, local_time);
2115     switch_time_exp_gmt_get(&ret, &tm);

```

根据入参选择优先级，返回值为字符串，分为 NORMAL LOW HIGH INVALID 四种情况。

```
switch_priority_name(switch_priority_t priority)
```


网络字节序转换为主机字节序。

```
get_addr_int(switch_sockaddr_t *sa)
```

对比地址信息。分别就地址类型（IPv4 or IPv6）、IP 地址、端口三个方面进行对比分析。

```
switch_cmp_addr(switch_sockaddr_t *sa1, switch_sockaddr_t *sa2)
```

复制 IP 地址信息，复制信息分别地址类型（IPv4 or IPv6）、IP 地址、端口。

```
switch_cp_addr(switch_sockaddr_t *sa1, switch_sockaddr_t *sa2)
```

根据入参构造 URI。

```
switch_build_uri(char *uri, switch_size_t size, const char *scheme, const char *user, const  
↪ switch_sockaddr_t *sa, int flags)
```

RFC2833 与字符串之间相互转换。

```
switch_rfc2833_to_char(int event)  
switch_char_to_rfc2833(char key)
```

通过在带有转义字符的字符列表前添加前缀来转义字符串。

```
switch_escape_char(switch_memory_pool_t *pool, char *in, const char *delim, char esc)
```

将字符串分离为字符时使用的辅助函数。支持的字符有 \n 换行符，\r 回车，\s 空格，\t Tab。

```
unescape_char(char escaped)
```

对字符串内特殊字符进行转移, 支持的字符有 `\n` 换行符, `\r` 回车, `' '` 空格, `\t` Tab, `$`, 返回值为 `out` 字符串。

```
switch_escape_string(const char *in, char *out, switch_size_t outlen)
```

对字符串内特殊字符进行转移, 支持的字符有 `\n` 换行符, `\r` 回车, `' '` 空格, `\t` Tab, `$`, 返回值为内存池内的 `buf`。

```
switch_escape_string_pool(const char *in, switch_memory_pool_t *pool)
```

辅助函数, 用于分隔字符串时删除引号、删除首位空格以及转换转移字符。

```
static char *cleanup_separated_string(char *str, char delim)
```

根据分隔符 `delim` 分隔字符串到数组内。

```
switch_separate_string_string(char *buf, char *delim, char **array, unsigned int arraylen)
```

使用不是空格的分隔符分隔字符串。

```
separate_string_char_delim(char *buf, char delim, char **array, unsigned int arraylen)
```

使用空格作为分隔符分隔字符串。

```
separate_string_blank_delim(char *buf, char **array, unsigned int arraylen)
```

拆分字符串, 根据分隔符是否为空格选择不同的函数。

```
switch_separate_string(char *buf, char delim, char **array, unsigned int arraylen)
return (delim == ' ' ? separate_string_blank_delim(buf, array, arraylen) : separate_string_char_delim(buf,
↪ delim, array, arraylen));
```

根据给出的文件路径创建指向该文件的指针。如文件路径为 `/usr/local/678.txt`，则返回值为指向字符 `6` 的指针。

```
switch_cut_path(const char *in)
```

字符串比对函数。

```
switch_string_match(const char *string, size_t string_len, const char *search, size_t search_len)
```

字符串替换函数。

```
switch_string_replace(const char *string, const char *search, const char *replace)
```

引用 shell 参数。

```
switch_util_quote_shell_arg(const char *string)
```

引用 shell 参数，若提供内存池，则返回值存放在内存池中；若没有提供内存池，则返回值存放在新申请的内存中，此时在使用完毕后，需释放该内存。其中还涉及到一些特殊字符在不同场景下的转换。

```
switch_util_quote_shell_arg_pool(const char *string, switch_memory_pool_t *pool)
```

对 URL 进行编码，由于 URL 只能使用固定的字符，其他字符会被编码转义为可用的格式。L3455 若 `*p` 为 `%`，则 L3446 检查 `*p+1` `*p+2` 是否存在于 `hex` 内，若是，则说明该字符已经被转义过或无需转移，若不是，L3455~L3457 对该字符进行转义，第一位为 `%`，第二位第三位为该字符对应十六进制数，如空格会被转移为 `%20`，结果存入 `buf` 内。

```
switch_url_encode_opt(const char *url, char *buf, size_t len, switch_bool_t double_encode)
3445     if (!double_encode && *p == '%' && e-p > 1) {
3446         if (strchr(hex, *(p+1)) && strchr(hex, *(p+2))) {
3447             ok = 1;
```

```

3448         }
3449     }
3450
3451     if (!ok && (*p < ' ' || *p > '~' || strchr(SWITCH_URL_UNSAFE, *p))) {
3452         if ((x + 3) > len) {
3453             break;
3454         }
3455         buf[x++] = '%';
3456         buf[x++] = hex[(*p >> 4) & 0x0f];
3457         buf[x++] = hex[*p & 0x0f];
3458     } else {
3459         buf[x++] = *p;
3460     }
3461 }

```

对 URL 进行解码，其中 `sscanf(s + 1, "%2x", &tmp)` 的意思是从 `s + 1` 开始读取两位十六进制数，放入 `tmp` 内。

```
switch_url_decode(char *s)
```

分离时间，时：分：秒。

```
switch_split_time(const char *exp, int *hour, int *min, int *sec)
```

分离日期，年-月-日。

```
switch_split_date(const char *exp, int *year, int *month, int *day)
```

比较两个时间点（如 2019-08-01 11:13:34~2019-09-01 12:32:22），后者是否晚于前者，且 `ts` 位于该时间段内。

```
switch_fulldate_cmp(const char *exp, switch_time_t *ts)
```

`int` 转十六进制函数，其中 `_switch_C_tolower_[1 + SWITCH_CTYPE_NUM_CHARS]` 和 `_switch_C_toupper_[1 + SWITCH_CTYPE_NUM_CHARS]` 为对照表，前者为小写，后者为大写。

```
old_switch_toupper(int c)
old_switch_tolower(int c)
```

L3796~L3849 为 Linux 下匹配模式的具体实现，转换方法则参照 `_switch_C_ctype_[1 + SWITCH_CTYPE_NUM_CHARS]` 数组，若是，则返回该值；若不是，则进行转换并返回对应值。函数名称可参考下表。

特殊符号	说明
alnum	代表英文大小写字符及数字，即 0-9, A-Z, a-z
alpha	代表任何英文大小写字符，即 A-Z, a-z
cntrl	代表键盘上面的控制按键，即包括 CR, LF, Tab, Del...等
digit	代表数字，即 0-9
graph	除了空白字符（空白按键与[Tab]按键）外的其他所有按键
lower	代表小写字符，即 a-z
print	代表任何可以被打印出来的字符
punct	代表标点符号，即” ’ ? ! ; : # \$...
space	任何会产生空白的字符，包括空白键, [Tab], CR 等
upper	代表大写字符，即 A-Z
xdigit	代表 16 进制的数字类型，包括： 0-9, A-F, a-f 的数字与字符

```
switch_isalnum(int c)
switch_isalpha(int c)
switch_iscntrl(int c)
switch_isdigit(int c)
switch_isgraph(int c)
switch_islower(int c)
switch_isprint(int c)
switch_ispunct(int c)
switch_isspace(int c)
switch_isupper(int c)
switch_isxdigit(int c)
```

1-7 数字转换为字符串形式的周一至周日。L3862 对 val 进行取余，余数对应的下标，即 int 型数字对应的星期几。

```
switch_dow_int2str(int val)
L3862 val = val % switch_arraylen(DOW);
```

字符串形式的星期几转换为数字 1-7。

```
switch_dow_str2int(const char *exp)
```

分离 `user` 和 `domain`，L4045 将 `user` 和 `domain` 分隔开（SIP 头域内，两者以 `user@domain` 形式存在）。

```
switch_split_user_domain(char *in, char **user, char **domain)
L4045 if ((h = in, p = strchr(h, '@')) *p = '\\0', u = in, h = p+1;
```

获取 UUID，L4066 生成 UUID，L4067 按照标准格式将 UUID 格式化为字符串。

```
switch_uuid_str(char *buf, switch_size_t len)
L4066 switch_uuid_get(&uuid);
L4067 switch_uuid_format(buf, &uuid);
```

生成纯数字的函数。

```
switch_format_number(const char *num)
```

字符串转无符号整型、字符串转无符号长整型函数。

```
switch_atoui(const char *nptr)
switch_atoul(const char *nptr)
```

解析 HTTP 头，L4242 若数据长度小于 16 字节，为 `GET / HTTP/1.1\r\n` 请求。L4250~L4258 解析出 HTTP 请求内的 `body` 字段。

```
switch_http_parse_header(char *buffer, uint32_t datalen, switch_http_request_t *request)
L4242 if (datalen < 16) return status;
...
4250 if ((body = strstr(buffer, "\\r\\n\\r\\n")) {
4251     *body = '\\0';
4252     body += 4;
```

```
4253     } else if (( body = strstr(buffer, "\n\n"))) {
4254         *body = '\0';
4255         body += 2;
4256     } else {
4257         return status;
4258     }
...

```

L4260~L4270 取出 HTTP 请求方法，目前只支持 GET POST PUT DELETE OPTIONS PATCH HEAD。

```
4260     request->_buffer = strdup(buffer);
4261     request->method = request->_buffer;
4262     request->bytes_buffered = datalen;
4263     if (body) {
4264         request->bytes_header = body - buffer;
4265         request->bytes_read = body - buffer;
4266     }
4267
4268     p = strchr(request->method, ' ');
4269
4270     if (!p) goto err;

```

L4272~L4293 获取 HTTP 请求资源字符串。

```
4272     *p++ = '\0';
4273
4274     if (*p != '/') goto err; /* must start from '/' */
4275
4276     request->uri = p;
4277     p = strchr(request->uri, ' ');
4278
4279     if (!p) goto err;
4280
4281     *p++ = '\0';
4282     http = p;
4283
4284     p = strchr(request->uri, '?');
4285
4286     if (p) {
4287         *p++ = '\0';
4288         request->qs = p;
4289     }
4290
4291     if (clean_uri((char *)request->uri) != SWITCH_STATUS_SUCCESS) {

```

```
4292     goto err;
4293 }
```

L4295~L4299 获取 HTTP 版本号, 当前只支持 HTTP/1.1。

```
4295     if (!strcmp(http, "HTTP/1.1", 8)) {
4296         request->keepalive = SWITCH_TRUE;
4297     } else if (strcmp(http, "HTTP/1.0", 8)) {
4298         goto err;
4299     }
```

L4316 获取 HTTP 头的数量, L4320~L4357 遍历所有 HTTP 头, L4337 将键值对存入对应的 EVENT headers 内。

```
4308     p = strchr(http, '\n');
4309
4310     if (p) {
4311         *p++ = '\0'; // now the first header
4312     } else {
4313         goto noheader;
4314     }
4315
4316     header_count = switch_separate_string(p, '\n', headers, sizeof(headers)/ sizeof(headers[0]));
4317
4318     if (header_count < 1) goto err;
4319
4320     for (i = 0; i < header_count; i++) {
4321         char *header, *value;
4322         int len;
4323
4324         argc = switch_separate_string(headers[i], ':', argv, 2);
4325
4326         if (argc != 2) goto err;
4327
4328         header = argv[0];
4329         value = argv[1];
4330
4331         if (*value == ' ') value++;
4332
4333         len = strlen(value);
4334
4335         if (len && *(value + len - 1) == '\r') *(value + len - 1) = '\0';
4336     }
```



```
4337     switch_event_add_header_string(request->headers, SWITCH_STACK_BOTTOM, header, value);
4338
4339     if (!strncasecmp(header, "User-Agent", 10)) {
4340         request->user_agent = value;
4341     } else if (!strncasecmp(header, "Host", 4)) {
4342         request->host = value;
4343         p = strchr(value, ':');
4344
4345         if (p) {
4346             *p++ = '\0';
4347
4348             if (*p) request->port = (switch_port_t)atoi(p);
4349         }
4350     } else if (!strncasecmp(header, "Content-Type", 12)) {
4351         request->content_type = value;
4352     } else if (!strncasecmp(header, "Content-Length", 14)) {
4353         request->content_length = atoi(value);
4354     } else if (!strncasecmp(header, "Referer", 7)) {
4355         request->referer = value;
4356     }
4357 }
```

打印出 HTTP 请求的相关字段。

```
switch_http_dump_request(switch_http_request_t *request)
```

4.30 switch_vad.c

本节基于 Commit Hash [f3fdb5a](#)。

VAD 是在音频处理，特别是语音识别中用到的一项基本技术。VAD 技术的好坏会在很大程度上影响语音识别的质量。

那么什么是 VAD 呢？

VAD 的全称是 Voice Activity Detection，即语音状态检测，也就是说检测什么时候开始讲话，什么时候结束讲话。注意，这里，V 是指语音，而不是声音。比如你在马路上打电话，刮风的声音很大，或者从你身边忽然跑过去一辆车，那些噪声都不是语音，好的 VAD 应该能区分人的语音和噪音。但是，比如你在打电话的时候旁边也有人讲话，这种区分就更难一些。不过，好在好的麦克风应该能根据音源的远近过滤声音，那就是硬件层面的事情了。

好的 VAD 需要 DSP（数字信号处理），这些处理一般都需要通过付立叶变换到频域中去做。

FreeSWITCH 里有几个简单的基于能量检测的算法，并没有用到 DSP。笔者也尝试加入了 `libfvad`⁷，它是从 WebRTC 的代码里提取出来的一个 VAD 库，FreeSWITCH 在编译时会自动检测是否有 `libfvad`。

闲言少叙，看代码。

L36，如果有 `libfvad`，会加载相应的头文件。

```
33 #include <switch.h>
34
35 #ifdef SWITCH_HAVE_FVAD
36 #include <fvad.h>
37 #endif
```

代码中加了注释。

```
39 struct switch_vad_s {
40     int talking;        // 是否正在讲话
41     int talked;         // 是否检测到讲话
42     int talk_hits;      // 检测到几次讲话
43     int listen_hits;
44     int hangover;       // 检测到停止讲话后，当前状态持续的次数
45     int hangover_len;   // 检测到停止讲话后，等待多长时间
46     int divisor;        // 除数，初始化为：采样率/8000
47     int thresh;         // 能量阈值
48     int channels;       // 声道数
49     int sample_rate;    // 采样率
50     int debug;
51     int _hangover_len;
52     int _thresh;
53     int _listen_hits;
54     switch_vad_state_t vad_state;
55 #ifdef SWITCH_HAVE_FVAD
56     Fvad *fvad;         // Libfvad 指针
57 #endif
58 };
```

VAD 状态机。

```
60 SWITCH_DECLARE(const char *) switch_vad_state2str(switch_vad_state_t state)
61 {
62     switch(state) {
```

⁷<https://github.com/dpirc/libfvad>

```
63     case SWITCH_VAD_STATE_NONE:
64         return "none";
65     case SWITCH_VAD_STATE_START_TALKING:
66         return "start_talking";
67     case SWITCH_VAD_STATE_TALKING:
68         return "talking";
69     case SWITCH_VAD_STATE_STOP_TALKING:
70         return "stop_talking";
71     default:
72         return "error";
73 }
74 }
```

初始化, 需要传入音频的采样率 (`sample`) 以及声道数 (`channels`), 成功返回一个 `switch_vad_t` 指针。

```
76 SWITCH_DECLARE(switch_vad_t *) switch_vad_init(int sample_rate, int channels)
77 {
78     switch_vad_t *vad = malloc(sizeof(switch_vad_t));
79
80     if (!vad) return NULL;
81
82     memset(vad, 0, sizeof(*vad));
83     vad->sample_rate = sample_rate ? sample_rate : 8000;
84     vad->channels = channels;
85     vad->_hangover_len = 25;
86     vad->_thresh = 100;
87     vad->_listen_hits = 10;
88     switch_vad_reset(vad);
89
90     return vad;
91 }
```

设置 VAD 的模式, 默认为 `-1`, 其它值都是 `libfvd` 提供的, 如果 `libfvd` 不存在, 则跟所有值都跟 `-1` 是一样的。

- `-1`: 使用 FreeSWITCH 原生的算法, 不使用 `libfvd`
- `0`: (“quality”), `1` (“low bitrate”), `2` (“aggressive”), and `3` * (“very aggressive”)

L95, 只有在有 `libfvd` 的情况下, 该函数才有意义。

L108, L116, L117, 初始化 `libfvd`。

```
93 SWITCH_DECLARE(int) switch_vad_set_mode(switch_vad_t *vad, int mode)
94 {
95     #ifdef SWITCH_HAVE_FVAD
96         int ret = 0;
97
98         if (mode < 0) {
99             if (vad->fvad) fvad_free(vad->fvad);
100
101             vad->fvad = NULL;
102             return ret;
103         } else if (mode > 3) {
104             mode = 3;
105         }
106
107         if (!vad->fvad) {
108             vad->fvad = fvad_new();
109
110             if (!vad->fvad) {
111                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "libfvad init error\n");
112             }
113         }
114
115         if (vad->fvad) {
116             ret = fvad_set_mode(vad->fvad, mode);
117             fvad_set_sample_rate(vad->fvad, vad->sample_rate);
118         }
119
120         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "libfvad started, mode = %d\n", mode);
121         return ret;
122     #else
123         return 0;
124     #endif
125 }
```

设置 VAD 相关的参数。`hangover_len` 为检测到停止讲话后，等待的时间（在这段时间内没有再次讲话）；`thresh` 为能量阈值，超过这个值才认为是在讲话；`listen_hits` 是『听』到几次讲话后才认为是开始讲话。

```
127 SWITCH_DECLARE(void) switch_vad_set_param(switch_vad_t *vad, const char *key, int val)
128 {
129     if (!key) return;
130
131     if (!strcmp(key, "hangover_len")) {
132         vad->hangover_len = val;
133     } else if (!strcmp(key, "thresh")) {
```

```

134     vad->thresh = vad->_thresh = val;
135 } else if (!strcmp(key, "debug")) {
136     vad->debug = val;
137 } else if (!strcmp(key, "listen_hits")) {
138     vad->listen_hits = vad->_listen_hits = val;
139 }
140 }

```

重置 VAD 状态（以便开启下一次检测）。

```

142 SWITCH_DECLARE(void) switch_vad_reset(switch_vad_t *vad)
143 {
144 #ifdef SWITCH_HAVE_FVAD
145     if (vad->fvad) {
146         fvad_reset(vad->fvad);
147         return;
148     }
149 #endif
150
151     vad->talking = 0;
152     vad->talked = 0;
153     vad->talk_hits = 0;
154     vad->hangover = 0;
155     vad->listen_hits = vad->_listen_hits;
156     vad->hangover_len = vad->_hangover_len;
157     vad->divisor = vad->sample_rate / 8000;
158     vad->thresh = vad->_thresh;
159     vad->vad_state = SWITCH_VAD_STATE_NONE;
160 }

```

下面是 VAD 检测的主要函数。输入参数为当前的 VAD 实例指针 `vad`、数据包中的数据 `data`（每个采样点为 2 字节，16 位整数），以及数据的长度 `samples`（采样的数量）。

一般来说，在 `ptime = 20` 的情况下，即 20 毫秒的数据包，8000Hz 采样率的情况下，每个数据包里包含 $8000 / (1000 / 20) = 160$ 个采样，每个数据占 2 个字节，一共点 320 字节。

```

162 SWITCH_DECLARE(switch_vad_state_t) switch_vad_process(switch_vad_t *vad, int16_t *data, unsigned int
↪ samples)
163 {
164     int energy = 0, j = 0, count = 0;
165     int score = 0;

```

L167, 如果上一个状态为 **TALKING** (停止讲话) 状态, 那就将状态设为 **NONE**。L171, 如果上一个状态为 **START_TALKING** (开始讲话), 则将当前状态设为 **TALKING** (正在讲话)。

```

167     if (vad->vad_state == SWITCH_VAD_STATE_STOP_TALKING) {
168         vad->vad_state = SWITCH_VAD_STATE_NONE;
169     }
170
171     if (vad->vad_state == SWITCH_VAD_STATE_START_TALKING) {
172         vad->vad_state = SWITCH_VAD_STATE_TALKING;
173     }

```

L175, 如果有 FVAD, 则在 L177 调用 **fvad_process** 将数据送给 FVAD 处理。FVAD 会返回 **1** 代表检测到语音活动, 0 为未检测到, -1 为错误。为了适配 FreeSWITCH 自身的算法, L181 将返回值与 **thresh** 相加并减去 **1**, 以便用与 FreeSWITCH 自身算法同样的方法比较是否超过 **thresh**。

```

175 #ifdef SWITCH_HAVE_FVAD
176     if (vad->fvad) {
177         int ret = fvad_process(vad->fvad, data, samples);
178
179         // printf("%d ", ret); fflush(stdout);
180
181         score = vad->thresh + ret - 1;
182     } else {
183 #endif

```

FVAD 是个黑盒子, 但下面就是 FreeSWITCH 原生的算法了。L185 循环对每一个采样值进行计算, L186 将所有能量绝对值相加 (有正数也有负数, 绝对值越大声音越大)。从 L187 可以看到, 如果声音有多个声道, 其实只计算了其中一个声道的能量, 另一个声道直接跳过了。

注: 在 FreeSWITCH 中, 多个声道的数据是交错排列的, 即 **左右左右左右左右...**。

L190 会计算出一个得数 **score**。在调试时可以打印出这些得数看一看 (L196 ~ L197)。

```

185     for (energy = 0, j = 0, count = 0; count < samples; count++) {
186         energy += abs(data[j]);
187         j += vad->channels;
188     }
189
190     score = (uint32_t) (energy / (samples / vad->divisor));
191

```

```
192 #ifdef SWITCH_HAVE_FVAD
193     }
194 #endif
195
196     //printf("%d ", score); fflush(stdout);
197     //printf("yay %d %d %d\n", score, vad->hangover, vad->talking);
```

L199 如果当前正在讲话，而得数值小于设定的阈值，那说明是停止讲话了。但这时还不能确认后面会不会是不是极短暂的停顿（比如讲话中喘口气），所以要继续判断 `hangover` (L200)，如果 `hangover` 大于 0，则减 1 (L201) 并继续检测，否则，说明检测到停止讲话有一段时间了 (L202)，则重置计数器。

```
199     if (vad->talking && score < vad->thresh) {
200         if (vad->hangover > 0) {
201             vad->hangover--;
202         } else { // if (hangover <= 0) {
203             vad->talking = 0;
204             vad->talk_hits = 0;
205             vad->hangover = 0;
206         }
```

其它状态（正在讲话或没有讲话），如果声音超过了阈值 (L208)，则会根据当前的状态计算出是第一次检测到讲话（`START_TALKING`）还是一直在讲话（`TALKING`）。

```
207     } else {
208         if (score >= vad->thresh) {
209             vad->vad_state = vad->talking ? SWITCH_VAD_STATE_TALKING : SWITCH_VAD_STATE_START_TALKING;
210             vad->talking = 1;
211             vad->hangover = vad->hangover_len;
212         }
213     }
```

如果认为当前正在讲话 (L217)，则将检测到的讲话的次数（`talk_hits`）加 1 (L218) 并与预设的需要『听』到几次以后才认为是开始说话（`listen_hits`）进行比较。也就是说，只有检测到一定数量的正能量后，才认为是开始说话。

```
217     if (vad->talking) {
218         vad->talk_hits++;
220         if (vad->talk_hits > vad->listen_hits) {
```

```

221         vad->talked = 1;
222         vad->vad_state = SWITCH_VAD_STATE_TALKING;
223     }
224 } else {
225     vad->talk_hits = 0;
226 }

```

如果以前讲话，现在又不讲了，则将状态设成 `STOP_TALKING`。

```

228     if ((vad->talked && !vad->talking)) {
229         // printf("NOT TALKING ANYMORE\n");
230         vad->talked = 0;
231         vad->vad_state = SWITCH_VAD_STATE_STOP_TALKING;
232     }
233
234     if (vad->debug > 0) {
235         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "VAD DEBUG energy: %d state %s\n", score,
236             ↪ switch_vad_state2str(vad->vad_state));
237     }
238     return vad->vad_state;
239 }

```

有点晕是吧。记住一件事情就可以了：**检测是有误差和延迟的，所以要多检测几个数据包。**看图，然后再回去看几遍代码。

注：关于参数的值，我们以 `hangover_len` 为例讲一下。它的默认值是 `25`，也就是说需要检测到 `25` 个能量小于预设阈值的数据包才能确认是停止说话了，如果每个包是 `20ms`，那么就是 `20 * 25 = 500ms` 后 FreeSWITCH 才会触发 `STOP_TALKING` 状态。



下面的代码是检测完成后清理现场。

```
241 SWITCH_DECLARE(void) switch_vad_destroy(switch_vad_t **vad)
242 {
243     if (*vad) {
244
245 #ifdef SWITCH_HAVE_FVAD
246         if ((*vad)->fvad) fvad_free ((*vad)->fvad);
247 #endif
248
249         free(*vad);
250         *vad = NULL;
251     }
252 }
```

在测试中，我们并没有感觉 `libfvad` 带来多大的好处，感兴趣的同学可以多测一测。

在 `mod_dptools` 中有一个 App 可以测试 VAD 的用法。只需要在 Dialplan 里加个 `<action application="vad_test"/>`，打电话测试看日志就可以了。代码如下。

首先可以在参数中指定检测模式（L6194），如果在没有 FVAD 的情况下模式无效。

```
6183 SWITCH_STANDARD_APP(vad_test_function)
6184 {
6185     switch_channel_t *channel = switch_core_session_get_channel(session);
6186     switch_codec_implementation_t imp = { 0 };
6187     switch_vad_t *vad;
6188     switch_frame_t *frame = { 0 };
6189     switch_vad_state_t vad_state;
6190     int mode = -1;
6191     const char *var = NULL;
6192     int tmp;
6193
6194     if (!lstrcmp(data)) {
6195         mode = atoi(data);
6196
6197         if (mode > 3) mode = 3;
6198     }
```

L6200 设置转码，读到的数据将解码为 RAW 格式的 PCM 数据（每个 Sample 占两个字节的 `short` 型整数）。L6201 读到当前 `session` 的参数到 `imp` 结构体中。接着 L6203 用读到的参数初始化 VAD。

```
6200     switch_core_session_raw_read(session);
6201     switch_core_session_get_read_impl(session, &imp);
```

```
6202
6203     vad = switch_vad_init(imp.samples_per_second, imp.number_of_channels);
6204     switch_assert(vad);
6205     switch_vad_set_mode(vad, mode);
```

L6207 ~ L6232 检测通道变量，可以设置不同的参数。参数的意义我们上面已经讲过了，默认值也可以在源代码中找到。

```
6207     if ((var = switch_channel_get_variable(channel, "vad_hangover_len"))) {
6208         tmp = atoi(var);
6209
6210         if (tmp > 0) switch_vad_set_param(vad, "hangover_len", tmp);
6211     }
6212
6213     if ((var = switch_channel_get_variable(channel, "vad_thresh"))) {
6214         tmp = atoi(var);
6215
6216         if (tmp > 0) switch_vad_set_param(vad, "thresh", tmp);
6217     }
6218
6219     if ((var = switch_channel_get_variable(channel, "vad_listen_hits"))) {
6220         tmp = atoi(var);
6221
6222         if (tmp > 0) switch_vad_set_param(vad, "listen_hits", tmp);
6223     }
6224
6225     if ((var = switch_channel_get_variable(channel, "vad_debug"))) {
6226         tmp = atoi(var);
6227
6228         if (tmp < 0) tmp = 0;
6229         if (tmp > 1) tmp = 1;
6230
6231         switch_vad_set_param(vad, "debug", tmp);
6232     }
```

L6234 无限循环，L6235 读取一个数据包得到一个 **frame**（如果是 8000Hz，20ms 的包，将包含 160 个 **sample**，占 320 字节），然后在 L6245 对该数据包进行检测。

```
6234     while (switch_channel_ready(channel)) {
6235         switch_status_t status = switch_core_session_read_frame(session, &frame,
↳ SWITCH_IO_FLAG_NONE, 0);
6236
6237         if (!SWITCH_READ_ACCEPTABLE(status)) {
```

```
6238         break;
6239     }
6240
6241     if (switch_test_flag(frame, SFF_CNG)) {
6242         continue;
6243     }
6244
6245     vad_state = switch_vad_process(vad, frame->data, frame->datalen / 2);
```

从上面的代码可以看到，VAD 里面有个状态机，它会根据收到的所有的数据包计算当前的状态。我们只需要将这些状态打印出来即可。

```
6247     if (vad_state == SWITCH_VAD_STATE_START_TALKING) {
6248         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "START TALKING\n");
6249         switch_core_session_write_frame(session, frame, SWITCH_IO_FLAG_NONE, 0);
6250     } else if (vad_state == SWITCH_VAD_STATE_STOP_TALKING) {
6251         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "STOP TALKING\n");
6252     } else if (vad_state == SWITCH_VAD_STATE_TALKING) {
6253         switch_core_session_write_frame(session, frame, SWITCH_IO_FLAG_NONE, 0);
6254     } else {
6255         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "vad_state: %s\n",
↪ switch_vad_state2str(vad_state));
6256     }
6257 }
```

完事记得清理现场。

```
6259     switch_vad_destroy(&vad);
6260     switch_core_session_reset(session, SWITCH_TRUE, SWITCH_TRUE);
6261 }
```

FreeSWITCH 的 VAD 算法是比较简单的，但通过这些代码我们可以看到整个 VAD 的过程。如果你有更高级的算法也欢迎贡献。

第五章 模块代码选析

本章我们开始分析模块代码。FreeSWITCH 的模块非常多，我们也不能穷尽所有模块，仅选一些经典模块简单分析一下。

5.1 mod_conference.c

王凯

本章基于 Commit Hash [cc02a4abfc4](#)

会议模块相关

L3884 模块加载 [mod_conference_load](#)

首先在 3895 行向核心注册该模块，接着增加命令补全函数。

```
3895  *module_interface = switch_loadable_module_create_module_interface(pool, modname);
3897  switch_console_add_complete_func("::conference::conference_list_conferences",
↪ conference_list_conferences);
```

(L3900 ~ L3903) 绑定会议相关事件处理函数，这些处理函数最终会调用 [switch_event_channel_broadcast](#) 分发对应的事件。

在 (L3919 ~ L3923) 初始化模块相关的 hash 以及 mutex，在 (L3926 ~ 3936) 通过 [switch_event_bind](#) 向核心订阅部分事件。

然后在 L3938 ~ L3941 向核心注册该模块实现的 API、APP 以及 CHAT 回调。

```
3938  SWITCH_ADD_API(api_interface, "conference", "Conference module commands", conference_api_main, p);
3939  SWITCH_ADD_APP(app_interface, mod_conference_app_name, mod_conference_app_name, NULL,
↪ conference_function, NULL, SAF_SUPPORT_TEXT_ONLY);
```

```

3940 SWITCH_ADD_APP(app_interface, "conference_set_auto_outcall", "conference_set_auto_outcall", NULL,
↪ conference_auto_function, NULL, SAF_NONE);
3941 SWITCH_ADD_CHAT(chat_interface, CONF_CHAT_PROTO, chat_send);

```

最后产生并发送一个 **PRESENCE_IN** 事件，同时设置模块运行标记 `conference_globals.running = 1`。

接下来我们看一下与会议相关的 API 及 APP。

L55 为会议 API 回调函数 `conference_api_main`，该回调直接调用 `conference_api_main_real`，在 L218 ~ L255 对命令行参数进行解析，在指定会议名称并且会议存在时，调用 `conference_api_dispatch` 进行 API 匹配处理。

会议支持的 API 子命令储存在全局的结构体数组 `conference_api_sub_commands` 中。大部分 API 的使用说明可以参考 [权威指南 12.5](#) 相关章节。

我们主要看一下 `conference` APP，配置好 Dialplan 之后，就可以将电话路由到会议，便会回调 `conference_function`，其中参数 `data` 为会议名称，举例 `3000-192.168.3.233@default`，其中 `default` 为该 3000 会议对应的 `profile`。接着便会在 L1991~L1998 打开配置文件 `conference.conf.xml` 并查找 `profile`，`profile` 中的 `param` 会在后面进行说明。

```

1997 xml_cfg.profile = switch_xml_find_child(profiles, "profile", "name", profile_name);

```

这里只关注非 `bidge` 模式会议流程，`bridge conference` 参考 L2008 ~ L2050。

首先会在全局 `hash` 中查找该会议是否已经存在，参见 `conference_find`，如果不存在则执行 L2068 ~ L2186。首先检查是否携带 `flags{}` 参数，并执行相应的操作。接着执行 `conference_new` 创建会议。

`conference_new` 函数定义在 2643 行。

```

2643 conference_obj_t *conference_new(char *name, conference_xml_cfg_t cfg, switch_core_session_t
↪ *session, switch_memory_pool_t *pool)

```

该函数比较长，其中参数 `cfg` 为上面提到的 `profile tree`。在 L2806 ~ L3114 会解析其中的参数并保存至局部变量中，接着会初始化内存池并在内存池中申请会议资源 `conference_obj_t`。

```

3139 if (switch_core_new_memory_pool(&pool) != SWITCH_STATUS_SUCCESS)
...
3147 if (!(conference = switch_core_alloc(pool, sizeof(*conference))))

```

随后会对 `conference` 结构进行初始化，主要涉及参数赋值，`mutex` 初始化，更新全局 `hash` 等。其中我们关心 L3693 的判断，如果视频模式为 `mux`，则会触发触屏模式的视频会议（[权威指南 12.5.2](#) 已经介绍过该模式）。在 L3707 初始化画布信息以及在 L3709 创建触屏线程，显而易见我们支持多画布，并且每个画布拥有自己的处理线程。

```

3703     for (j = 0; j < video_canvas_count; j++) {
3704         mcu_canvas_t *canvas = NULL;
3705
3706         switch_mutex_lock(conference->canvas_mutex);
3707         conference_video_init_canvas(conference, vlayout, &canvas);
3708         conference_video_attach_canvas(conference, canvas, 0);
3709         conference_video_launch_muxing_thread(conference, canvas, 0);
3710         switch_mutex_unlock(conference->canvas_mutex);
3711     }

```

该函数的最后发布会议创建的事件 `conference-create`。

会议创建成功之后会为该会议室创建线程 `conference_launch_thread`，我们看一下线程处理函数 `conference_thread_run`。TODO

在 `conference_function` 最后会调用 `conference_member_add`，将主叫添加至会议中。并且会为该会议成员 `session` 设置 `video_read_callback`

```

2415     /* Add the caller to the conference */
2416     if (conference_member_add(conference, &member) != SWITCH_STATUS_SUCCESS) {
2417         switch_core_codec_destroy(&member.read_codec);
2418         goto done;
2419     }
2420     ...
2437     /* Chime in the core video thread */
2438     switch_core_session_set_video_read_callback(session, conference_video_thread_callback, (void
↳ *)&member);
2439     switch_core_session_set_text_read_callback(session, conference_text_thread_callback, (void
↳ *)&member);

```

函数 `switch_core_session_set_video_read_callback` 定义在 `switch_core_media.c` L14809。该函数除了对 `session->video_read_callback` 进行赋值外，还调用了 `switch_core_session_start_video_thread`，启动视频线程，对视频数据进行收发，主要是调用 `switch_core_session_read_video_frame` 和 `switch_core_session_write_video_frame`。

在函数 `switch_core_session_read_video_frame` 最后 L14802 行，调用 `switch_core_session_video_read_callback`，而该函数会调用 `session->video_read_callback`。至此该会议成员的视频数据将由 `mod_conference` 接管，即 `conference_video_thread_callback`，该回调函数位于 `conference_video.c` L4954。

```

14836 SWITCH_DECLARE(switch_status_t) switch_core_session_video_read_callback(switch_core_session_t
↳ *session, switch_frame_t *frame)
14837 {
...
14845 switch_mutex_lock(smh->control_mutex);
14846
14847 if (session->video_read_callback) {
14848     status = session->video_read_callback(session, frame, session->video_read_user_data);
14849 }
14850
14851 switch_mutex_unlock(smh->control_mutex);
...
}

```

函数 `conference_video_thread_callback` 检查 `CFLAG_VIDEO_MUXING` 并会将视频 `image` push 至 `video_queue` 中。

```

if (switch_queue_trypush(member->video_queue, img_copy) != SWITCH_STATUS_SUCCESS)

```

在 `conference_video_muxing_thread_run` 线程中会调用 `conference_video_pop_next_image` 从队列中取出 `image` 并进行处理。

5.2 mod_hiredis.c

Mariah

本章基于 Commit Hash [6f8d65c3487](#)

`mod_hiredis` 提供了一些基本的 redis 命令操作接口，同时弃用了 `mod_redis` 模块，提供了相应的 `limit` 接口。

L422 模块加载 `mod_hiredis_load`

首先在 429 行向核心注册该模块。

```

429 *module_interface = switch_loadable_module_create_module_interface(pool, modname);

```

(L430) 初始化全局变量 `pool`，指向模块的内存池，模块的内存池会在核心里申请。

在（L410~L433）初始化模块全局的 `hash` 及 `mutex`，它们在初始化的时候会使用全局的内存池。

（L435）加载解析 xml 配置文件，如果解析失败，则模块不会成功加载。

```
435 if ( mod_hiredis_do_config() != SWITCH_STATUS_SUCCESS ) {
    return SWITCH_STATUS_GENERR;
};
```

在（L439~L440）向核心注册该模块实现的 `limit`。

```
439 SWITCH_ADD_LIMIT(limit_interface, "hiredis", hiredis_limit_incr, hiredis_limit_release,
↳ hiredis_limit_usage,
440 hiredis_limit_reset, hiredis_limit_status, hiredis_limit_interval_reset);
```

其中 `hiredis_limit_incr` 向核心提供了使用 redis 作为 limit 后台时资源被占用后计数加一的回调函数，`hiredis_limit_release` 提供了相应的释放资源后使资源计数减一的回调函数。`hiredis_limit_usage` 则提供了资源查询的回调函数。`hiredis_limit_reset` 提供了重置资源计数的回调函数，目前不支持全局的重置，可以通过 `hiredis_raw set <resource_name> 0` 来代替，`hiredis_limit_status` 提供了获取 limit 资源当前状态的回调函数，该模块目前还不支持。`hiredis_limit_interval_reset` 提供了重置间隔计数器的回调函数，该模块目前还不支持。

在 L441 ~ L442 向核心注册该模块实现的 APP、API。

```
441 SWITCH_ADD_APP(app_interface, "hiredis_raw", "hiredis_raw", "hiredis_raw", raw_app, "",
↳ SAF_SUPPORT_NOMEDIA | SAF_ROUTING_EXEC | SAF_ZOMBIE_EXEC);
442 SWITCH_ADD_API(api_interface, "hiredis_raw", "hiredis_raw", raw_api, "");
```

L447 模块卸载 `mod_hiredis_shutdown` 模块卸载的时候需要释放申请的资源（L453~L456），停止运行的线程等等。

```
453 while ((hi = switch_core_hash_first(mod_hiredis_globals.profiles))) {
454     switch_core_hash_this(hi, NULL, NULL, (void **)&profile);
455     hiredis_profile_destroy(&profile);
456     switch_safe_free(hi);
457 }

switch_core_hash_destroy(&(mod_hiredis_globals.profiles));
```

首先我们看一下该模块实现的 limit 回调函数。

L185 为 `hiredis_limit_incr` 的函数实现。`interval` 表示最大资源并发使用的间隔 (L251)。接着调用函数 `hiredis_profile_execute_pipeline_printf` 使资源数计数加一 (L221)。如果参数 `interval` 的值大于 0，则在第一次计数器加一后，设置了 redis 键的超时时间 (L235_{L236})，这样经过 `interval` 秒后，又可以重新使用资源。如果在计数器加一后返回的值超过了 `max` 的值，则会设置通道变量 `hiredis_limit_exceeded` 的值为 `true`

```

216     limit_key = switch_core_session_sprintf(session, "%s_%d", resource, now / interval);
217 } else {
218     limit_key = switch_core_session_sprintf(session, "%s", resource);
219 }
220
221 if ( (status = hiredis_profile_execute_pipeline_printf(profile, session, &response, "incr %s",
↳ limit_key) ) != SWITCH_STATUS_SUCCESS ) {
222     if ( status == SWITCH_STATUS_SOCKERR && profile->ignore_connect_fail ) {
223         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_INFO, "hiredis: ignoring
↳ profile[%s] connection error incrementing [%s]\n", realm, limit_key);
224         switch_goto_status(SWITCH_STATUS_SUCCESS, done);
225     } else if ( profile->ignore_error ) {
226         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_INFO, "hiredis: ignoring
↳ profile[%s] general error incrementing [%s]\n", realm, limit_key);
227         switch_goto_status(SWITCH_STATUS_SUCCESS, done);
228     }
229     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "hiredis: profile[%s] error
↳ incrementing [%s] because [%s]\n", realm, limit_key, response ? response : "");
230     switch_channel_set_variable(channel, "hiredis_raw_response", response ? response : "");
231     switch_goto_status(SWITCH_STATUS_GENERR, done);
232 }
233
234 /* set expiration for interval on first increment */
235 if ( interval && !strcmp("1", response ? response : "") ) {
236     hiredis_profile_execute_pipeline_printf(profile, session, NULL, "expire %s %d", limit_key,
↳ interval);
237 }
238
239 switch_channel_set_variable(channel, "hiredis_raw_response", response ? response : "");
240
241 count = atoll(response ? response : "");
242
243 if ( switch_is_number(response ? response : "") && count <= 0 ) {
244     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_WARNING, "limit not positive after
↳ increment, resource = %s, val = %s\n", limit_key, response ? response : "");
245 } else {
246     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_DEBUG, "resource = %s, response =
↳ %s\n", limit_key, response ? response : "");
247 }
248

```

```

249 if ( !switch_is_number(response ? response : "") && !profile->ignore_error ) {
250     /* got response error */
251     switch_goto_status(SWITCH_STATUS_GENERR, done);
252 } else if ( max > 0 && count > 0 && count > max ) {
253     switch_channel_set_variable(channel, "hiredis_limit_exceeded", "true");
254     if ( !interval ) { /* don't need to decrement intervals if limit exceeded since the interval keys
↳ are named w/ timestamp */
255         if ( profile->delete_when_zero ) {
256             hiredis_profile_eval_pipeline(profile, session, NULL, DECR_DEL_SCRIPT, 1, limit_key);
257         } else {
258             hiredis_profile_execute_pipeline_printf(profile, session, NULL, "decr %s", limit_key);
259         }
260     }
261     switch_goto_status(SWITCH_STATUS_GENERR, done);
262 }

```

L286 为 `hiredis_limit_release` 的函数实现。该回调函数首先判断 `realm` 和 `resource` 的值，如果都为空（L300），则会去遍历模块的 limit 资源列表（L303），使所有的资源计数器减一，如果 profile 里的成员 `delete_when_zero` 值为真，那么当该资源执行减一操作后，资源值是小于等于零，则会从 redis 里删除该资源键（L309~L310）。如果执行失败，则只打印一行错误日志（L315），然后继续遍历列表。

```

303 for ( cur = limit_pvt->first; cur; cur = cur->next ) {
304     /* Rate limited resources are not auto-decremented, they will expire. */
305     if ( !cur->interval && cur->inc ) {
306         switch_status_t result;
307         cur->inc = 0; /* mark as released */
308         profile = switch_core_hash_find(mod_hiredis_globals.profiles, cur->realm);
309         if ( profile->delete_when_zero ) {
310             result = hiredis_profile_eval_pipeline(profile, session, &response, DECR_DEL_SCRIPT, 1, cur-
↳ >limit_key);
311         } else {
312             result = hiredis_profile_execute_pipeline_printf(profile, session, &response, "decr %s",
↳ cur->limit_key);
313         }
314         if ( result != SWITCH_STATUS_SUCCESS ) {
315             switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "hiredis:
↳ profile[%s] error decrementing [%s] because [%s]\n",
316                             cur->realm, cur->limit_key, response ? response : "");
317         }
318         switch_safe_free(response);
319         response = NULL;
320     }
321 }

```

如果 `resource` 有值，则只对 `resource` 指定的资源进行减一操作，过程同上，最后会将执行的结果设置到通道变量 `hiredis_raw_response` 中（L349）。

L365 为 `hiredis_limit_usage` 的函数实现。首先会根据 `realm` 的值去模块全局 hash 查询对应的 profile 结构（L367），如果查询失败，则返回错误（L376~L379）。然后调用 `hiredis_profile_execute_pipeline_printf` 函数把把查询命令放在管道里，并且等待执行结果（L381），最后将获取的资源数转成整数返回给核心（L386）。

```

367 hiredis_profile_t *profile = switch_core_hash_find(mod_hiredis_globals.profiles, realm);
368 int64_t count = 0; /* Redis defines the incr action as to be performed on a 64 bit signed integer */
369 char *response = NULL;
370
371 if ( zstr(realm) ) {
372     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "hiredis: realm must be defined\n");
373     goto err;
374 }
375
376 if ( !profile ) {
377     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "hiredis: Unable to locate profile[%s]\n",
↪ realm);
378     goto err;
379 }
380
381 if ( hiredis_profile_execute_pipeline_printf(profile, NULL, &response, "get %s", resource) !=
↪ SWITCH_STATUS_SUCCESS ) {
382     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "hiredis: profile[%s] error querying [%s]
↪ because [%s]\n", realm, resource, response ? response : "");
383     goto err;
384 }
385
386 count = atoll(response ? response : "");
387
388 switch_safe_free(response);

```

接下来我们看一下模块相关的 API 及 APP。

L102 为模块 APP 回调函数 `raw_app`，该回调首先解析传入的参数（L108_{L121}），参数之间以空格隔开。第一个参数是配置文件里的 profile 的名称，然后会调用 `hiredis_profile_execute_sync` 函数去同步执行 redis 命令（L130）。最后将执行结果设置在通道变量 `hiredis_raw_response` 中（L134）。

```

104 switch_channel_t *channel = switch_core_session_get_channel(session);
105 char *response = NULL, *profile_name = NULL, *cmd = NULL;
106 hiredis_profile_t *profile = NULL;
107

```

```

108 if ( !zstr(data) ) {
109     profile_name = strdup(data);
110 } else {
111     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "hiredis: invalid data! Use
↪ the format 'default set keyname value' \n");
112     goto done;
113 }
114
115 if ( (cmd = strchr(profile_name, ' ')) ) {
116     *cmd = '\0';
117     cmd++;
118 } else {
119     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "hiredis: invalid data! Use
↪ the format 'default set keyname value' \n");
120     goto done;
121 }
122
123 profile = switch_core_hash_find(mod_hiredis_globals.profiles, profile_name);
124
125 if ( !profile ) {
126     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "hiredis: Unable to locate
↪ profile[%s]\n", profile_name);
127     return;
128 }
129
130 if ( hiredis_profile_execute_sync(profile, session, &response, cmd) != SWITCH_STATUS_SUCCESS ) {
131     switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "hiredis: profile[%s] error
↪ executing [%s] because [%s]\n", profile_name, cmd, response ? response : "");
132 }
133
134 switch_channel_set_variable(channel, "hiredis_raw_response", response ? response : "");

```

L142 为模块 API 回调函数 `raw_api`，该 API 的实现函数基本和 APP 相同。

5.3 mod_fifo.c

本节基于 Commit Hash `1681db4`。

`mod_fifo` 模块实现了一些简单的 ACD（自动电话分配）功能，是一个呼叫中心应用程序，它允许您使用指定的优先级创建自定义呼叫队列。

首先是 `load` 和 `shutdown` 模块的函数。

```

35 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_fifo_shutdown);
36 SWITCH_MODULE_LOAD_FUNCTION(mod_fifo_load);
37 SWITCH_MODULE_DEFINITION(mod_fifo, mod_fifo_load, mod_fifo_shutdown, NULL);

```

mod_fifo.c 返回 fifo 信息的 API。

```

4886 SWITCH_ADD_API(commands_api_interface, "fifo", "Return data about a fifo", fifo_api_function,
↳ FIFO_API_SYNTAX);

```

其 API 语法为：

```

4163 #define FIFO_API_SYNTAX "list|list_verbose|count|debug|status|has_outbound|importance [<fifo name>]|
↳ reparse [del_all]"

```

mod_fifo.c 向 fifo 中添加或删除成员的 API。

```

4887 SWITCH_ADD_API(commands_api_interface, "fifo_member", "Add members to a fifo",
↳ fifo_member_api_function, FIFO_MEMBER_API_SYNTAX);

```

其 API 语法为：

```

4755 #define FIFO_MEMBER_API_SYNTAX "[add <fifo_name> <originate_string> [<simo_count>] [<timeout>]
↳ [<lag>] [<expires>] [<taking_calls>] | del <fifo_name> <originate_string>]"

```

其中 `simo_count`、`timeout`、`lag`、`expires`、`taking_calls` 是 `fifo_member_add` 特有的参数，会在判断出是 `add` 后再做出详细的解析，若这些参数没有设置，则会采取默认值。

```

86 if (action && !strcasecmp(action, "add")) {
87 if (argc > 3) {
...
92 while (--sane > 0) {
93     ret = sqlite3_exec(db, sql, callback, data, &err);
94     if (ret == SQLITE_BUSY || ret == SQLITE_LOCKED) {
95         if (sane > 1) {
96             switch_core_db_free(err);

```

```

97         switch_yield(100000);
98         continue;
99     }
100 } else {
101     break;
102 }
103 }
```

`mod_fifo.c` 将出站成员添加到 fifo 的 API。

```

4888 SWITCH_ADD_API(commands_api_interface, "fifo_add_outbound", "Add outbound members to a fifo",
↳ fifo_add_outbound_function, "<node> <url> [<priority>]");
```

`mod_fifo.c` 检查 UUID 是否在网桥中的 API。

```

4889 SWITCH_ADD_API(commands_api_interface, "fifo_check_bridge", "check if uuid is in a bridge",
↳ fifo_check_bridge_function, "<uuid>|<outbound_id>");
```

在拨打电话时，`fifo_function` 这个 APP 启动，将电话停在一个泊位上，并可以设置电话停在泊位上时播放的音乐。

```

4883 SWITCH_ADD_APP(app_interface, "fifo", "Park with FIFO", FIFO_DESC, fifo_function, FIFO_USAGE,
↳ SAF_NONE);
```

从队列中删除匹配时间。

```

275 static switch_status_t fifo_queue_pop_nameval(fifo_queue_t *queue, const char *name, const char *val,
↳ switch_event_t **pop, int remove)
```

销毁指定 `uuid` 的时间并从队列中删除

```

334 static switch_status_t fifo_queue_popfly(fifo_queue_t *queue, const char *uuid)
```

创建新的 `fifo` 节点

```
1006     static fifo_node_t *create_node(const char *name, uint32_t importance, switch_mutex_t *mutex)
```

启动节点线程

```
2194     static void start_node_thread(switch_memory_pool_t *pool);
```

停止节点线程

```
2204     static int stop_node_thread(void)
```

跟踪以及取消跟踪呼叫的会话消息

```
1100     static void do_unbridge(switch_core_session_t *consumer_session, switch_core_session_t  
↳ *caller_session)  
1187     static switch_status_t messagehook (switch_core_session_t *session, switch_core_session_message_t  
↳ *msg)
```

向出站成员发起呼叫

```
1389     static void *SWITCH_THREAD_FUNC outbound_ringall_thread_run(switch_thread_t *thread, void *obj)  
1782     static void *SWITCH_THREAD_FUNC outbound_enterprise_thread_run(switch_thread_t *thread, void *obj)
```

累计出站成员结果

```
1935     static int place_call_ringall_callback(void *pArg, int argc, char **argv, char **columnNames)
```

提取出站成员结果

```
1962     static int place_call_enterprise_callback(void *pArg, int argc, char **argv, char **columnNames)
```

查找要调用给定 FIFO 节点的出站成员

```
2016 static int find_consumers(fifo_node_t *node)
```

不断尝试将呼叫传递给出站成员

```
2097 static void *SWITCH_THREAD_FUNC node_thread_run(switch_thread_t *thread, void *obj)
```

5.4 mod_amqp

韩小仿

本章基于 Commit Hash [1681db4](#)

先看头文件，mod_amqp.h

```
179 typedef struct mod_amqp_globals_s {  
180     switch_memory_pool_t *pool;  
181  
182     switch_hash_t *producer_hash;  
183     switch_hash_t *command_hash;  
184     switch_hash_t *logging_hash;  
185 } mod_amqp_globals_t;
```

顾名思义，这是个全局变量，

其中 pool 是内存池。

producer_hash 生产者的哈希表, mod_amqp 收到 FreeSWITCH 事件(订阅哪些事件是可配置的)之后处理成 amqp 消息，发送到 RabbitMQ 服务器。

command_hash 命令哈希表, 是 amqp 消费者，在收到 amqp 消息之后执行 FreeSWITCH API，并且可以根据需要反馈执行的结果。

logging_hash 日志哈希表, 作用是收到 FreeSWITCH 日志之后发送 amqp 消息。

```
87 typedef struct {  
88     char *name;
```



```
89
90  char *exchange;
91  char *exchange_type;
92  int exchange_durable;
93  int exchange_auto_delete;
94  int delivery_mode;
95  int delivery_timestamp;
96  char *content_type;
97  mod_amqp_keypart_t format_fields[MAX_ROUTING_KEY_FORMAT_FIELDS+1];
98
99
100 /* Array to store the possible event subscriptions */
101 int event_subscriptions;
102 switch_event_node_t *event_nodes[SWITCH_EVENT_ALL];
103 switch_event_types_t event_ids[SWITCH_EVENT_ALL];
104 switch_event_node_t *eventNode;
105
106
107 /* Because only the 'running' thread will be reading or writing to the two connection pointers
108  * this does not 'yet' need a read/write lock. Before these structures can be destroyed,
109  * the running thread must be joined first.
110  */
111 mod_amqp_connection_t *conn_root;
112 mod_amqp_connection_t *conn_active;
113
114 /* Rabbit connections are not thread safe so one connection per thread.
115  * Communicate with sender thread using a queue */
116 switch_thread_t *producer_thread;
117 switch_queue_t *send_queue;
118 unsigned int send_queue_size;
119
120 int reconnect_interval_ms;
121 int circuit_breaker_ms;
122 switch_time_t circuit_breaker_reset_time;
123 switch_bool_t enable_fallback_format_fields;
124
125 switch_bool_t running;
126 switch_memory_pool_t *pool;
127 char *custom_attr;
128 } mod_amqp_producer_profile_t;
```

看下 mod_amqp_producer_profile_t 这个结构里面的变量

- **name**: profile 的名字。
- **exchange**: amqp 交换机的名字，在配置文件里面配置。
- **exchange_type**: amqp 交换机的名字，在配置文件里面配置。

- **exchange_durable**: 是否持久化, 在配置文件里面配置。
- **exchange_auto_delete**: 是否自动删除, 在配置文件里面配置。
- **delivery_mode**: 消息派发模式, 在配置文件里面配置。
- **delivery_timestamp**: 消息派发时是否打时间戳, 在配置文件里面配置。
- **content_type**: 消息派发时 **Content_Header** 的值, 在配置文件里面配置, 默认是 **text/json**。
- **format_fields**: 在配置文件里面配置, 程序会据此计算出 **routing_key**。
- **event_subscriptions**: 配置文件配置 **event_filter**, 定义要订阅哪些 FreeSWITCH 事件, **event_subscriptions** 是事件个数。
- **event_ids**: 数组, 分析配置文件里面的配置项 **event_filter**, 按分割符分开后保存在 **event_ids** 数组。
- **event_nodes**: 数组, 调用 **switch_event_bind_removable** 函数订阅事件, 绑定的句柄保存在 **event_nodes**。
- **eventNode**: 这个没有任何作用, 完全可以删除。
- **conn_root**: amqp 根连接。
- **conn_active**: amqp 已经激活的连接。通过 **conn_root** 和 **conn_active** 这两个元素可以把所有的 **producer profile** 串到一起。这有两个作用, 一是退出时删除所有的连接, 二是如果连接断开可以尝试重连。
- **producer_thread**: 线程, 不多讲。
- **send_queue**: 发送队列。
- **send_queue_size**: 发送队列的大小。
- **reconnect_interval_ms**: 重连的时间间隔。
- **circuit_breaker_ms**:
- **circuit_breaker_reset_time**: 这个参数跟上面的参数构成一个断路器, 下面再讲。

exchange_type 比较常用的就是 **topic**, **Fanout** 和 **Direct** 比较少用。

topic 方式 **routing_key** 特别重要, 否则, 尽管 **exchange** 名字相同, 但有可能收不到消息。

笔者是这样做的, **format_fields** 配置为 **#FreeSWITCH,FreeSWITCH- Hostname,Event-Name,Event-Subclass,Unique-ID**,

其中, 第一个项目以 **#** 开头, 表示这是一个常数, 另外四个项目都是变量,

接收消息的一方, channel 绑定的 **routing_key** 是 ***.*.*.***, 正好匹配。

`mod_amqp_command_profile_t` 和 `mod_amqp_logging_profile_t` 的定义与此类似，不再赘述。

接着来看 `mod_amqp.c`

```

57 SWITCH_MODULE_LOAD_FUNCTION(mod_amqp_load)
58 {
59     switch_api_interface_t *api_interface;
60
61     memset(&mod_amqp_globals, 0, sizeof(mod_amqp_globals_t));
62     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
63
64     mod_amqp_globals.pool = pool;
65     switch_core_hash_init(&(mod_amqp_globals.producer_hash));
66     switch_core_hash_init(&(mod_amqp_globals.command_hash));
67     switch_core_hash_init(&(mod_amqp_globals.logging_hash));
68
69     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "mod_apqp loading: Version %s\n",
↪ switch_version_full());
70
71     /* Create producer profiles */
72     if ( mod_amqp_do_config(SWITCH_FALSE) != SWITCH_STATUS_SUCCESS ){
73         return SWITCH_STATUS_GENERR;
74     }
75
76     SWITCH_ADD_API(api_interface, "amqp", "amqp API", amqp_reload, "syntax");
77
78     switch_log_bind_logger(mod_amqp_logging_recv, SWITCH_LOG_DEBUG, SWITCH_FALSE);
79
80     return SWITCH_STATUS_SUCCESS;
81 }

```

加载 `mod_amqp` 模块时会自动调用 `mod_amqp_load` 函数。

L65-L67 初始化三个哈希表。

L72 读配置文件。

L76 初始化 `API`，其作用是重新读配置文件，并且重启所有的 profile。

L78 绑定 `logger`。

现在看 `mod_amqp_producer.c`

```

163 switch_status_t mod_amqp_producer_create(char *name, switch_xml_t cfg)
164 {
165     mod_amqp_producer_profile_t *profile = NULL;
166     int arg = 0, i = 0;

```

```

167     char *argv[SWITCH_EVENT_ALL];
168     switch_xml_t params, param, connections, connection;
169     switch_threadattr_t *thd_attr = NULL;
170     char *exchange = NULL, *exchange_type = NULL, *content_type = NULL;
171     int exchange_durable = 1; /* durable */
172     int delivery_mode = -1;
173     int delivery_timestamp = 1;
174     switch_memory_pool_t *pool;
175     char *format_fields[MAX_ROUTING_KEY_FORMAT_FIELDS+1];
176     int format_fields_size = 0;
177
178     memset(format_fields, 0, MAX_ROUTING_KEY_FORMAT_FIELDS + 1);
179
180     if (switch_core_new_memory_pool(&pool) != SWITCH_STATUS_SUCCESS) {
181         goto err;
182     }
183
184     profile = switch_core_alloc(pool, sizeof(mod_amqp_producer_profile_t));
185     profile->pool = pool;
186     profile->name = switch_core_strdup(profile->pool, name);
187     profile->running = 1;
188     memset(profile->format_fields, 0, (MAX_ROUTING_KEY_FORMAT_FIELDS + 1) * sizeof(mod_amqp_keypart_t));
189     profile->event_ids[0] = SWITCH_EVENT_ALL;
190     profile->event_subscriptions = 1;
191     profile->conn_root = NULL;
192     profile->conn_active = NULL;
193     /* Set reasonable defaults which may change if more reasonable defaults are found */
194     /* Handle defaults of non string types */
195     profile->circuit_breaker_ms = 10000;
196     profile->reconnect_interval_ms = 1000;
197     profile->send_queue_size = 5000;
198
199     if ((params = switch_xml_child(cfg, "params")) != NULL) {
200         for (param = switch_xml_child(params, "param"); param; param = param->next) {
201             char *var = (char *) switch_xml_attr_soft(param, "name");
202             char *val = (char *) switch_xml_attr_soft(param, "value");
203
204             if (!var) {
205                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Profile[%s] param missing 'name'
↵ attribute\n", profile->name);
206                 continue;
207             }
208
209             if (!val) {
210                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Profile[%s] param[%s] missing
↵ 'value' attribute\n", profile->name, var);
211                 continue;
212             }
213

```

```

214         if (!strcmp(var, "reconnect_interval_ms", 21)) {
215             int interval = atoi(val);
216             if ( interval && interval > 0 ) {
217                 profile->reconnect_interval_ms = interval;
218             }
219         } else if (!strcmp(var, "circuit_breaker_ms", 18)) {
220             int interval = atoi(val);
221             if ( interval && interval > 0 ) {
222                 profile->circuit_breaker_ms = interval;
223             }
224         } else if (!strcmp(var, "send_queue_size", 15)) {
225             int interval = atoi(val);
226             if ( interval && interval > 0 ) {
227                 profile->send_queue_size = interval;
228             }
229         } else if (!strcmp(var, "enable_fallback_format_fields", 29)) {
230             int interval = atoi(val);
231             if ( interval && interval > 0 ) {
232                 profile->enable_fallback_format_fields = 1;
233                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "amqp fallback format fields
↳ enabled\n");
234             }
235         } else if (!strcmp(var, "exchange-type", 13)) {
236             exchange_type = switch_core_strdup(profile->pool, val);
237         } else if (!strcmp(var, "exchange-name", 13)) {
238             exchange = switch_core_strdup(profile->pool, val);
239         } else if (!strcmp(var, "exchange-durable", 16)) {
240             exchange_durable = switch_true(val);
241         } else if (!strcmp(var, "delivery-mode", 13)) {
242             delivery_mode = atoi(val);
243         } else if (!strcmp(var, "delivery-timestamp", 18)) {
244             delivery_timestamp = switch_true(val);
245         } else if (!strcmp(var, "exchange_type", 13)) {
246             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Found exchange_type parameter.
↳ please change to exchange-type\n");
247         } else if (!strcmp(var, "exchange", 8)) {
248             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Found exchange parameter.
↳ please change to exchange-name\n");
249         } else if (!strcmp(var, "content-type", 12)) {
250             content_type = switch_core_strdup(profile->pool, val);
251         } else if (!strcmp(var, "format_fields", 13)) {
252             char *tmp = switch_core_strdup(profile->pool, val);
253             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "amqp format fields : %s\n",
↳ tmp);
254             if ((format_fields_size = mod_amqp_count_chars(tmp, ',') >=
↳ MAX_ROUTING_KEY_FORMAT_FIELDS) {
255                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "You can have only %d routing
↳ fields in the routing key.\n",
256                                 MAX_ROUTING_KEY_FORMAT_FIELDS);

```

```

257         goto err;
258     }
259
260     /* increment size because the count returned the number of separators, not number of
↳ fields */
261     format_fields_size++;
262     switch_separate_string(tmp, ',', format_fields, MAX_ROUTING_KEY_FORMAT_FIELDS);
263     format_fields[format_fields_size] = NULL;
264 } else if (!strcmp(var, "event_filter", 12)) {
265     char *tmp = switch_core_strdup(profile->pool, val);
266     /* Parse new events */
267     profile->event_subscriptions = switch_separate_string(tmp, ',', argv, (sizeof(argv) /
↳ sizeof(argv[0])));
268
269     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "Found %d subscriptions\n",
↳ profile->event_subscriptions);
270
271     for (arg = 0; arg < profile->event_subscriptions; arg++) {
272         if (switch_name_event(argv[arg], &(profile->event_ids[arg])) !=
↳ SWITCH_STATUS_SUCCESS) {
273             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "The switch event %s was
↳ not recognised.\n", argv[arg]);
274         }
275     }
276
277 }
278 } /* params for loop */
279 }
280
281 /* Handle defaults of string types */
282 profile->exchange = exchange ? exchange : switch_core_strdup(profile->pool, "TAP.Events");
283 profile->exchange_type = exchange_type ? exchange_type : switch_core_strdup(profile->pool, "topic");
284 profile->exchange_durable = exchange_durable;
285 profile->delivery_mode = delivery_mode;
286 profile->delivery_timestamp = delivery_timestamp;
287 profile->content_type = content_type ? content_type : switch_core_strdup(profile->pool,
↳ MOD_AMQP_DEFAULT_CONTENT_TYPE);
288
289
290 for(i = 0; i < format_fields_size; i++) {
291     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "amqp routing key %d : %s\n", i,
↳ format_fields[i]);
292     if(profile->enable_fallback_format_fields) {
293         profile->format_fields[i].size = switch_separate_string(format_fields[i], '|',
↳ profile->format_fields[i].name,
↳ MAX_ROUTING_KEY_FORMAT_FALLBACK_FIELDS);
294     }
295     if(profile->format_fields[i].size > 1) {
296         for(arg = 0; arg < profile->format_fields[i].size; arg++) {
297             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO,

```

```

298                                     "amqp routing key %d : sub key %d : %s\n", i, arg, profile-
↳ >format_fields[i].name[arg]);
299     }
300 }
301 } else {
302     profile->format_fields[i].name[0] = format_fields[i];
303     profile->format_fields[i].size = 1;
304 }
305 }
306
307
308 if ((connections = switch_xml_child(cfg, "connections")) != NULL) {
309     for (connection = switch_xml_child(connections, "connection"); connection; connection =
↳ connection->next) {
310         if (! profile->conn_root ) { /* Handle first root node */
311             if (mod_amqp_connection_create(&(profile->conn_root), connection, profile->pool) !=
↳ SWITCH_STATUS_SUCCESS) {
312                 /* Handle connection create failure */
313                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Profile[%s] failed to
↳ create connection\n", profile->name);
314                 continue;
315             }
316             profile->conn_active = profile->conn_root;
317         } else {
318             if (mod_amqp_connection_create(&(profile->conn_active->next), connection, profile->pool)
↳ != SWITCH_STATUS_SUCCESS) {
319                 /* Handle connection create failure */
320                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Profile[%s] failed to
↳ create connection\n", profile->name);
321                 continue;
322             }
323             profile->conn_active = profile->conn_active->next;
324         }
325     }
326 }
327 profile->conn_active = NULL;
328
329 if ( mod_amqp_connection_open(profile->conn_root, &(profile->conn_active), profile->name, profile-
↳ >custom_attr) != SWITCH_STATUS_SUCCESS) {
330     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Profile[%s] was unable to connect to
↳ any connection\n", profile->name);
331     goto err;
332 }
333 #if AMQP_VERSION_MAJOR == 0 && AMQP_VERSION_MINOR >= 6
334     amqp_exchange_declare(profile->conn_active->state, 1,
335                           amqp_cstring_bytes(profile->exchange),
336                           amqp_cstring_bytes(profile->exchange_type),
337                           0, /* passive */
338                           profile->exchange_durable,

```

```

339         profile->exchange_auto_delete,
340         0,
341         amqp_empty_table);
342 #else
343     amqp_exchange_declare(profile->conn_active->state, 1,
344         amqp_cstring_bytes(profile->exchange),
345         amqp_cstring_bytes(profile->exchange_type),
346         0, /* passive */
347         profile->exchange_durable,
348         amqp_empty_table);
349 #endif
350
351     if (mod_amqp_log_if_amqp_error(amqp_get_rpc_reply(profile->conn_active->state), "Declaring
↪ exchange\n")) {
352         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Profile[%s] failed to create
↪ exchange\n", profile->name);
353         goto err;
354     }
355
356     /* Create a bounded FIFO queue for sending messages */
357     if (switch_queue_create(&(profile->send_queue), profile->send_queue_size, profile->pool) !=
↪ SWITCH_STATUS_SUCCESS) {
358         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Cannot create send queue of size
↪ %d!\n", profile->send_queue_size);
359         goto err;
360     }
361
362
363     /* Start the event send thread. This will set up the initial connection */
364     switch_threadattr_create(&thd_attr, profile->pool);
365     switch_threadattr_stacksize_set(thd_attr, SWITCH_THREAD_STACKSIZE);
366     if (switch_thread_create(&profile->producer_thread, thd_attr, mod_amqp_producer_thread, profile,
↪ profile->pool)) {
367         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Cannot create 'amqp event sender'
↪ thread!\n");
368         goto err;
369     }
370
371     /* Subscribe events */
372     for (i = 0; i < profile->event_subscriptions; i++) {
373         if (switch_event_bind_removable("AMQP",
374             profile->event_ids[i],
375             SWITCH_EVENT_SUBCLASS_ANY,
376             mod_amqp_producer_event_handler,
377             profile,
378             &(profile->event_nodes[i])) != SWITCH_STATUS_SUCCESS) {
379             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Cannot bind to event handler
↪ %d!\n", (int)profile->event_ids[i]);
380             goto err;

```



```

381     }
382 }
383
384 if ( switch_core_hash_insert(mod_amqp_globals.producer_hash, name, (void *) profile) !=
↪ SWITCH_STATUS_SUCCESS) {
385     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Failed to insert new profile [%s] into
↪ mod_amqp profile hash\n", name);
386     goto err;
387 }
388
389 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "Profile[%s] Successfully started\n",
↪ profile->name);
390 return SWITCH_STATUS_SUCCESS;
391
392 err:
393 /* Cleanup */
394 mod_amqp_producer_destroy(&profile);
395 return SWITCH_STATUS_GENERR;
396
397 }

```

这个函数的作用是根据配置文件创建一个 **amqp 生产者**

读配置的部分跳过，没什么好说的

L329 创建 **amqp_connection**，其步骤是：

- **amqp_new_connection()**
- **amqp_tcp_socket_new()**
- **amqp_socket_open()**
- **amqp_channel_open**

L333 比较 **amqp lib 库** 的版本，老版本的 **amqp_exchange_declare** 支持的参数较少。

L334 声明（创建）一个交换机。

L351 检查 **amqp 服务器**的返回值以及出错后的处理。

L357 创建一个先进先出的队列。

L365 创建一个线程。

L371-L382 订阅 FreeSWITCH 事件

L384 把创建好的 **profile** 加到 **producer_hash** 这个哈希表里面。

```

41 switch_status_t mod_amqp_producer_routing_key(mod_amqp_producer_profile_t *profile, char
↳ routingKey[MAX_AMQP_ROUTING_KEY_LENGTH],
42                                     switch_event_t* evt, mod_amqp_keypart_t
↳ routingKeyEventHeaderNames[])
43 {
44     int i = 0, idx = 0, x = 0;
45     char keybuffer[MAX_AMQP_ROUTING_KEY_LENGTH];
46
47     for (i = 0; i < MAX_ROUTING_KEY_FORMAT_FIELDS && idx < MAX_AMQP_ROUTING_KEY_LENGTH; i++) {
48         if (routingKeyEventHeaderNames[i].size) {
49             if (idx) {
50                 routingKey[idx++] = '.';
51             }
52             for (x = 0; x < routingKeyEventHeaderNames[i].size; x++) {
53                 if (routingKeyEventHeaderNames[i].name[x][0] == '#') {
54                     strncpy(routingKey + idx, routingKeyEventHeaderNames[i].name[x] + 1,
↳ MAX_AMQP_ROUTING_KEY_LENGTH - idx);
55                     break;
56                 } else {
57                     char *value = switch_event_get_header(evt, routingKeyEventHeaderNames[i].name[x]);
58                     if (value) {
59                         amqp_util_encode(value, keybuffer);
60                         strncpy(routingKey + idx, keybuffer, MAX_AMQP_ROUTING_KEY_LENGTH - idx);
61                         break;
62                     }
63                 }
64             }
65             idx += strlen(routingKey + idx);
66         }
67     }
68     return SWITCH_STATUS_SUCCESS;
69 }

```

这个函数的作用是产生 `routing_key`

L53 比较是不是 # 开头, 如果是, 就是一个常数。

L57 不是 # 开头, 从 `event` 里面取出值。

把所有的项目用 . 连起来, 形成 `routing_key`。

```

400 switch_status_t mod_amqp_producer_send(mod_amqp_producer_profile_t *profile, mod_amqp_message_t *msg)
401 {
402     amqp_table_entry_t messageTableEntries[2];
403     amqp_basic_properties_t props;
404     int status;

```

```

405     uint64_t timestamp;
406
407     if (! profile->conn_active) {
408         /* No connection, so we can not send the message. */
409         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Profile[%s] not active\n", profile-
↵ >name);
410         return SWITCH_STATUS_NOT_INITIALIZED;
411     }
412     memset(&props, 0, sizeof(amqp_basic_properties_t));
413
414     props._flags = AMQP_BASIC_CONTENT_TYPE_FLAG;
415     props.content_type = amqp_cstring_bytes(profile->content_type);
416
417     if(profile->delivery_mode > 0) {
418         props._flags |= AMQP_BASIC_DELIVERY_MODE_FLAG;
419         props.delivery_mode = profile->delivery_mode;
420     }
421
422     if(profile->delivery_timestamp) {
423         props._flags |= AMQP_BASIC_TIMESTAMP_FLAG | AMQP_BASIC_HEADERS_FLAG;
424         props.timestamp = (uint64_t)time(NULL);
425         props.headers.num_entries = 1;
426         props.headers.entries = messageTableEntries;
427         timestamp = (uint64_t)switch_micro_time_now();
428         messageTableEntries[0].key = amqp_cstring_bytes("x_Liquid_MessageSentTimeStamp");
429         messageTableEntries[0].value.kind = AMQP_FIELD_KIND_TIMESTAMP;
430         messageTableEntries[0].value.value.u64 = (uint64_t)(timestamp / 1000000);
431         messageTableEntries[1].key = amqp_cstring_bytes("x_Liquid_MessageSentTimeStampMicro");
432         messageTableEntries[1].value.kind = AMQP_FIELD_KIND_U64;
433         messageTableEntries[1].value.value.u64 = timestamp;
434     }
435
436     status = amqp_basic_publish(
437         profile->conn_active->state,
438         1,
439         amqp_cstring_bytes(profile->exchange),
440         amqp_cstring_bytes(msg->routing_key),
441         0,
442         0,
443         &props,
444         amqp_cstring_bytes(msg->pjson));
445
446     if (status < 0) {
447         const char *errstr = amqp_error_string2(-status);
448         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Profile[%s] failed to send event on
↵ connection[%s]: %s\n",
449             profile->name, profile->conn_active->name, errstr);
450
451         /* This is bad, we couldn't send the message. Clear up any connection */

```

```

452     mod_amqp_connection_close(profile->conn_active);
453     profile->conn_active = NULL;
454     return SWITCH_STATUS_SOCKERR;
455 }
456
457 return SWITCH_STATUS_SUCCESS;
458 }

```

```

71 void mod_amqp_producer_event_handler(switch_event_t* evt)
72 {
73     mod_amqp_message_t *amqp_message;
74     mod_amqp_producer_profile_t *profile = (mod_amqp_producer_profile_t *)evt->bind_user_data;
75     switch_time_t now = switch_time_now();
76     switch_time_t reset_time;
77
78     if (!profile) {
79         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Event without a profile %p %p\n", (void
↵ *)evt, (void *)evt->event_user_data);
80         return;
81     }
82
83     /* If the mod is disabled ignore the event */
84     if (!profile->running) {
85         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Profile[%s] not running\n", profile-
↵ >name);
86         return;
87     }
88
89     /* If the circuit breaker is active, ignore the event */
90     reset_time = profile->circuit_breaker_reset_time;
91     if (now < reset_time) {
92         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Profile[%s] circuit breaker hit[%d]
↵ (%d)\n", profile->name, (int) now, (int) reset_time);
93         return;
94     }
95
96     switch_malloc(amqp_message, sizeof(mod_amqp_message_t));
97
98     switch_event_serialize_json(evt, &amqp_message->pjson);
99     mod_amqp_producer_routing_key(profile, amqp_message->routing_key, evt, profile->format_fields);
100
101     /* Queue the message to be sent by the worker thread, errors are reported only once per circuit
↵ breaker interval */
102     if (switch_queue_trypush(profile->send_queue, amqp_message) != SWITCH_STATUS_SUCCESS) {
103         unsigned int queue_size = switch_queue_size(profile->send_queue);
104

```

```
105     /* Trip the circuit breaker for a short period to stop recurring error messages (time is
↳ measured in uS) */
106     profile->circuit_breaker_reset_time = now + profile->circuit_breaker_ms * 1000;
107
108     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "AMQP message queue full. Messages will
↳ be dropped for %.1fs! (Queue capacity %d)",
109                     profile->circuit_breaker_ms / 1000.0, queue_size);
110
111     mod_amqp_util_msg_destroy(&amqp_message);
112 }
113 }
```

```
460 void * SWITCH_THREAD_FUNC mod_amqp_producer_thread(switch_thread_t *thread, void *data)
461 {
462     mod_amqp_message_t *msg = NULL;
463     switch_status_t status = SWITCH_STATUS_SUCCESS;
464     mod_amqp_producer_profile_t *profile = (mod_amqp_producer_profile_t *)data;
465     amqp_boolean_t passive = 0;
466     amqp_boolean_t durable = 1;
467
468     while (profile->running) {
469
470         if (!profile->conn_active) {
471             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Amqp no connection-
↳ reconnecting...\n");
472
473             status = mod_amqp_connection_open(profile->conn_root, &(profile->conn_active), profile->name,
↳ profile->custom_attr);
474             if ( status == SWITCH_STATUS_SUCCESS ) {
475                 // Ensure that the exchange exists, and is of the correct type
476 #if AMQP_VERSION_MAJOR == 0 && AMQP_VERSION_MINOR >= 6
477                 amqp_exchange_declare(profile->conn_active->state, 1,
478                                     amqp_cstring_bytes(profile->exchange),
479                                     amqp_cstring_bytes(profile->exchange_type),
480                                     passive,
481                                     durable,
482                                     profile->exchange_auto_delete,
483                                     0,
484                                     amqp_empty_table);
485 #else
486                 amqp_exchange_declare(profile->conn_active->state, 1,
487                                     amqp_cstring_bytes(profile->exchange),
488                                     amqp_cstring_bytes(profile->exchange_type),
489                                     passive,
490                                     durable,
491                                     amqp_empty_table);
492 #endif
```

```

493         if (!mod_amqp_log_if_amqp_error(amqp_get_rpc_reply(profile->conn_active->state),
↪ "Declaring exchange")) {
494             switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "Amqp reconnect successful-
↪ connected\n");
495             continue;
496         }
497     }
498
499     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_WARNING, "Profile[%s] failed to connect
↪ with code(%d), sleeping for %dms\n",
500                     profile->name, status, profile->reconnect_interval_ms);
501     switch_sleep(profile->reconnect_interval_ms * 1000);
502     continue;
503 }
504
505     if (!msg && switch_queue_pop_timeout(profile->send_queue, (void**)&msg, 1000000) !=
↪ SWITCH_STATUS_SUCCESS) {
506         continue;
507     }
508
509     if (msg) {
510 #ifdef MOD_AMQP_DEBUG_TIMING
511         long times[TIME_STATS_TO_AGGREGATE];
512         static unsigned int thistime = 0;
513         switch_time_t start = switch_time_now();
514 #endif
515         switch (mod_amqp_producer_send(profile, msg)) {
516             case SWITCH_STATUS_SUCCESS:
517                 /* Success: prepare for next message */
518                 mod_amqp_util_msg_destroy(&msg);
519                 break;
520
521             case SWITCH_STATUS_NOT_INITIALIZED:
522                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Send failed with 'not
↪ initialised'\n");
523                 break;
524
525             case SWITCH_STATUS_SOCKERR:
526                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Send failed with 'socket
↪ error'\n");
527                 break;
528
529             default:
530                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Send failed with a generic
↪ error\n");
531
532                 /* Send failed and closed the connection; reconnect will happen at the beginning of the
↪ loop
533                 * NB: do we need a delay here to prevent a fast reconnect-send-fail loop? */

```

```

534         break;
535     }
536
537 #ifdef MOD_AMQP_DEBUG_TIMING
538     times[thistime++] = switch_time_now() - start;
539     if (thistime >= TIME_STATS_TO_AGGREGATE) {
540         int i;
541         long min_time, max_time, avg_time;
542
543         /* Calculate aggregate times */
544         min_time = max_time = avg_time = times[0];
545         for (i = 1; i < TIME_STATS_TO_AGGREGATE; ++i) {
546
547             avg_time += times[i];
548             if (times[i] < min_time) min_time = times[i];
549             if (times[i] > max_time) max_time = times[i];
550         }
551
552         avg_time /= TIME_STATS_TO_AGGREGATE;
553         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_DEBUG, "Microseconds to send last %d
↪ messages: Min %ld Max %ld Avg %ld\n",
554                         TIME_STATS_TO_AGGREGATE, min_time, max_time, avg_time);
555         thistime = 0;
556     }
557 #endif
558 }
559 }
560
561 /* Abort the current message */
562 mod_amqp_util_msg_destroy(&msg);
563
564 // Terminate the thread
565 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_INFO, "Event sender thread stopped\n");
566 switch_thread_exit(thread, SWITCH_STATUS_SUCCESS);
567 return NULL;
568 }

```

附：测试用到的资料

- 安装 RabbitMQ-server 3.8.0

```
sudo apt-get update -y
```

```
## Install prerequisites
```

```
sudo apt-get install curl gnupg -y
```

```
## Install RabbitMQ signing key
curl -fsSL https://github.com/rabbitmq/signing-keys/releases/download/2.0/rabbitmq-release-signing-key.asc |
↳ sudo apt-key add -

## Install apt HTTPS transport
sudo apt-get install apt-transport-https

## Add Bintray repositories that provision latest RabbitMQ and Erlang 21.x releases
sudo tee /etc/apt/sources.list.d/bintray.rabbitmq.list <<EOF
## Installs the latest Erlang 21.x release.
## Change component to "erlang" to install the latest version (22.x or later).
deb https://dl.bintray.com/rabbitmq-erlang/debian stretch erlang-21.x
deb https://dl.bintray.com/rabbitmq/debian stretch main
EOF

## Update package indices
sudo apt-get update -y

## Install rabbitmq-server and its dependencies
sudo apt-get install rabbitmq-server -y --fix-missing
```

- 编译 mod_amqp
 - apt-get install librabbitmq-dev
 - cd FreeSWITCH 源码目录, 重新执行./configure
 - 编译 mod_amqp
 - pika 安装
 - apt-get install python3-pip
 - pip3 install pika # 目前是 1.1.0 版本
 - mod_amqp 配置文件
-

```
<configuration name="amqp.conf" description="mod_amqp">
  <producers>
    <profile name="default">
      <connections>
        <connection name="primary">
          <param name="hostname" value="localhost"/>
          <param name="virtualhost" value="/" />
          <param name="username" value="guest"/>
          <param name="password" value="guest"/>
          <param name="port" value="5672"/>
          <param name="heartbeat" value="0"/>
        </connection>
      </connections>
    </profile>
  </producers>
</configuration>
```



```

</connection>
<!-- <connection name="secondary"> -->
    <!-- <param name="hostname" value="localhost"/> -->
    <!-- <param name="virtualhost" value="/" /> -->
    <!-- <param name="username" value="guest"/> -->
    <!-- <param name="password" value="guest"/> -->
    <!-- <param name="port" value="5672"/> -->
    <!-- <param name="heartbeat" value="0"/> -->
<!-- </connection> -->
</connections>
<params>
    <param name="exchange-name" value="TAP.Events"/>
    <param name="exchange-type" value="topic"/>
    <param name="circuit_breaker_ms" value="10000"/>
    <param name="reconnect_interval_ms" value="1000"/>
    <param name="send_queue_size" value="5000"/>
    <param name="enable_fallback_format_fields" value="1"/>
    <param name="format_fields" value="#FreeSWITCH,FreeSWITCH-Hostname,Event-Name,Event-
↳ Subclass,Unique-ID"/>
    <param name="event_filter"
↳ value="SWITCH_EVENT_CHANNEL_CREATE,SWITCH_EVENT_CHANNEL_DESTROY,SWITCH_EVENT_HEARTBEAT,SWITCH_EVENT_DTMF,SWITCH
↳ >
</params>
</profile>
</producers>

<commands>
    <profile name="default">
        <connections>
            <connection name="primary">
                <param name="hostname" value="localhost"/>
                <param name="virtualhost" value="/" />
                <param name="username" value="guest"/>
                <param name="password" value="guest"/>
                <param name="port" value="5672"/>
                <param name="heartbeat" value="0"/>
            </connection>
        </connections>
        <params>
            <param name="exchange-name" value="TAP.Commands"/>
            <param name="binding_key" value="*.*/>
            <param name="reconnect_interval_ms" value="1000"/>
        </params>
    </profile>
</commands>

<logging>
    <profile name="default">
        <connections>

```

```

    <connection name="primary">
        <param name="hostname" value="localhost"/>
        <param name="virtualhost" value="/" />
        <param name="username" value="guest"/>
        <param name="password" value="guest"/>
        <param name="port" value="5672"/>
        <param name="heartbeat" value="0"/>
    </connection>
</connections>
<params>
    <param name="exchange-name" value="TAP.Logging"/>
    <param name="send_queue_size" value="5000"/>
    <param name="reconnect_interval_ms" value="1000"/>
    <param name="log-levels" value="debug,info,notice,warning,err,crit,alert"/>
</params>
</profile>
</logging>
</configuration>

```

- amqp_logging.py, amqp 消费者, 接收 mod_amqp 发过来的 FreeSWITCH 日志

```
# python3 amqp_logging.py
```

```
import json
import pika
```

```

credentials = pika.PlainCredentials("guest", "guest")
parameters = pika.ConnectionParameters(host="guest", virtual_host="/", credentials=credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()
exchange = "TAP.Logging"
channel.exchange_declare(exchange=exchange, exchange_type="topic", durable=True)
result = channel.queue_declare("", exclusive=True)
queue_name = result.method.queue
print("queue_name: " + queue_name)

```

```

channel.queue_bind(exchange=exchange, queue=queue_name, routing_key="*.*.*")
print(" [*] Waiting for FreeSWITCH logs. To exit press CTRL+C")

```

```

def callback(ch, method, properties, body):
    j = json.loads(str(body, encoding="utf-8"))
    print(j)

```

```

channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)
channel.start_consuming()

```

运行界面:

```
{'level': 'DEBUG', 'timestamp_epoch': 1571293000.0, 'content': ' sofia/internal/1001@192.168.0.128 Standard  
↪ DESTROY\n', 'timestamp': '2019-10-17 14:15:35.782782', 'function': 'switch_core_standard_on_destroy',  
↪ 'line': 181, 'file': 'switch_core_state_machine.c'}
```

- amqp_client.py, amqp 消费者, 接收 mod_amqp 发过来的 FreeSWITCH 事件

```
# python3 amqp_client.py
```

```
import json  
import pika
```

```
credentials = pika.PlainCredentials("guest", "guest")  
parameters = pika.ConnectionParameters(host="localhost", virtual_host="/", credentials=credentials)  
connection = pika.BlockingConnection(parameters)  
channel = connection.channel()  
exchange = "TAP.Events"  
channel.exchange_declare(exchange=exchange, exchange_type="topic", durable=True)  
result = channel.queue_declare("", exclusive=True)  
queue_name = result.method.queue  
print("queue_name: " + queue_name)
```

```
# 注意 routing_key 的值, 跟 amqp.conf.xml 里面生产者的这个配置项目 `format_fields` 对应起来  
channel.queue_bind(exchange=exchange, queue=queue_name, routing_key="*.*.*.*")  
print(" [*] Waiting for FreeSWITCH event. To exit press CTRL+C")
```

```
def callback(ch, method, properties, body):  
    j = json.loads(str(body, encoding="utf-8"))  
    print(j)
```

```
channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)  
channel.start_consuming()
```

运行界面:

```
{'Core-UUID': 'd39aaef6-ef20-11e9-b5ad-ad3c7a803873', 'FreeSWITCH-Switchname': 'fs187', 'Event-Calling-Function': 'send_heartbeat', 'Event-Date-GMT': 'Thu, 17 Oct 2019 08:16:07 GMT', 'Session-Per-Sec-FiveMin': '0', 'Event-Calling-Line-Number': '80', 'Up-Time': '0 years, 2 days, 0 hours, 22 minutes, 59 seconds, 412 milliseconds, 592 microseconds', 'Event-Calling-File': 'switch_core.c', 'Idle-CPU': '89.366667', 'Session-Per-Sec-Max': '1', 'Event-Sequence': '24214', 'FreeSWITCH-IPv4': '192.168.0.128', 'Event-Date-Timestamp': '1571300167362820', 'Session-Peak-FiveMin': '0', 'Event-Date-Local': '2019-10-17 16:16:07', 'Session-Count': '0', 'Session-Peak-Max': '1', 'Session-Since-Startup': '21', 'Max-Sessions': '1000', 'Session-Per-Sec': '30', 'Event-Name': 'HEARTBEAT', 'Session-Per-Sec-Last': '0', 'FreeSWITCH-IPv6': '::1', 'FreeSWITCH-Version': '1.8.7~64bit', 'FreeSWITCH-Hostname': 'fs187', 'Uptime-msec': '174179412', 'Event-Info': 'System Ready'}
```

- amqp_command.py, amqp 生产者, 发送 amqp 消息, mod_amqp 收到之后执行 FreeSWITCH API

```
# python3 amqp_command.py

import pika

credentials = pika.PlainCredentials("guest", "guest")
parameters = pika.ConnectionParameters(host="localhost", virtual_host="/", credentials=credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()
channel.exchange_declare(exchange="TAP.Commands", exchange_type="topic", durable=True)
result = channel.queue_declare("", exclusive=True)

exchange = "TAP.Commands"
routing_key = "freeswitch.api"
message = "originate user/1001 &echo"

# 注意这里的 headers, 如果这样写就可以收到 reply
channel.basic_publish(exchange=exchange, routing_key=routing_key,
                    properties=pika.BasicProperties(headers={"x-fs-api-resp-exchange": "fsapi.exchange",
                    "x-fs-api-resp-key": "fsapi.reply"}),
                    body=message)

print(" [x] Sent %r:%r" % (routing_key, message))

connection.close()
```

- amqp_reply.py, amqp 消费者, 获取 FreeSWITCH API 的执行结果

```
# python3 amqp_reply.py
```

```
import json
import pika

credentials = pika.PlainCredentials("guest", "guest")
parameters = pika.ConnectionParameters(host="localhost", virtual_host="/", credentials=credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()
exchange="fsapi.exchange"
channel.exchange_declare(exchange=exchange, exchange_type="topic", durable=True)
result = channel.queue_declare("", exclusive=True)
queue_name = result.method.queue
print("queue_name: " + queue_name)

channel.queue_bind(exchange=exchange, queue=queue_name, routing_key="*.")
print(" [*] Waiting for reply. To exit press CTRL+C")

def callback(ch, method, properties, body):
    j = json.loads(str(body, encoding="utf-8"))
    print(j)

channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)
channel.start_consuming()
```

运行界面:

```
{'command': 'originate user/1001 &echo', 'output': '+OK d7d8a816-f0be-11e9-b6bc-ad3c7a803873\n', 'status':  
↩ 0}
```

第六章 代码修炼之道

恋爱的最终结果是结婚，但婚后又往往怀念——当初谈恋爱的过程才是最美丽、最令人难忘的。

同样，程序员写代码最辛苦然而也最令人兴奋的往往并不是程序最终运行的结果，而是，不断地调试挥汗如雨的过程。

我们现在看到的 FreeSWITCH 代码是十年间不断修改、迭代而成的，即使你有兴趣顺着 Git 代码库的提交历史仔细研究每一个提交，那些已经提交的代码也是反复测试修改后又提交的，而开发过程中的种种崩溃，不身临其境可能永远也体会不到。

本章，我们就来看一看 FreeSWITCH 成长过程中出现的那些代码。希望读者能从另外一个视角理解 FreeSWITCH 代码。

6.1 虚拟演播室

虚拟演播室是一种很好玩的技术，可以通过图像处理技术把当前图像的背景去掉，换上另外的背景，比如高山或大海，或者是央视的舞台，给观众的感觉就像你真在那里一样。

前几天，Anthony 跟我聊，说他找到一个开源的库，可以做这个功能，问我是否可以把这个功能在 FreeSWITCH 里做出来。我看了看，这个库叫 OpenShot¹，跨平台，看起来很不错，但有两个主要问题：1) 它依赖于很多很多其它的库；2) 没有单独的发行版的开发库，要从源代码编译安装，而前面已经说过了，它依赖太多，编译起来太麻烦。

后来，经过研究该库的源代码，发现技术实现其实比较简单，只照着它写了几个函数。不多说，上代码。

6.1.1 Chroma Key

这些代码已经提交到 FreeSWITCH 代码库里了，读者可以用如下命令查看：

```
git show f31393d
```

¹<http://openshot.org/>

为节省篇幅，我们仅列出关键内容。L16 ~ L20 在 `switch_core_video.h` 中加了一个函数声明，该函数可以对一幅图像进行处理。该技术叫 ChromaKey，中文叫色键抠像，就是常说的抠图。

```

7 diff --git a/src/include/switch_core_video.h b/src/include/switch_core_video.h
16 +
17 +/*! \brief chromakey an img, img must be RGBA and return modified img */
18 +
19 +SWITCH_DECLARE(void) switch_img_chromakey(switch_image_t *img, switch_rgb_color_t *mask, int
↪ threshold);

```

在 `switch_core_video.c` 中，增加了一个内联函数声明。该函数用于比较两个颜色间的差异（距离）。如果值越大，说明颜色差异越大。

```

24 diff --git a/src/switch_core_video.c b/src/switch_core_video.c
32 +/*! \brief compute distance between two colors
33 +*
34 +* \param[in]  c1      RGB color1
35 +* \param[in]  c2      RGB color2
36 +*/
37 +static inline int switch_color_distance(switch_rgb_color_t *c1, switch_rgb_color_t *c2);
38 +

```

下面，就是 `chromakey` 函数的具体实现。L46，函数传入参数是一帧图像，一个画布的颜色（`mask`，用 RGB 色彩空间），以及一个阈值 `threshold`，即允许的色彩范围。

L51，检查图像必须使用 ARGB 色彩空间，即，图像的一个像素用 4 个字节表示，其中 A 为 Alpha 通道，即透明度（0 ~ 255，值越小越透明），RGB 分别表示红绿蓝，在内存同的表示也是 ARGB 字节的顺序。

L53，让 `pixel` 指针，指向图像的数据起始区。ARGB 图像使用连续的内存区，总长度为 `width x height x 4`，即“长 x 宽 x 4”，因为一个像素点 4 个字节。

```

46 +SWITCH_DECLARE(void) switch_img_chromakey(switch_image_t *img, switch_rgb_color_t *mask, int threshold)
47 +{
48 +  uint8_t *pixel;
49 +  switch_assert(img);
50 +
51 +  if (img->fmt != SWITCH_IMG_FMT_ARGB) return;
52 +
53 +  pixel = img->planes[SWITCH_PLANE_PACKED];

```

L55, 遍历所有像素。L56 通过 `color` 变量指向当前像素, L57 计算该像素与画布像素 (`mask`) 的差异, 返回值越小说明颜色越相近, 如果它们的相似度小于某一阈值 (`threshold`), 则将该像素的 Alpha 通道置为 0, 即完全透明。

```

55 + for (; pixel < (img->planes[SWITCH_PLANE_PACKED] + img->d_w * img->d_h * 4); pixel += 4) {
56 +     switch_rgb_color_t *color = (switch_rgb_color_t *)pixel;
57 +     int distance = switch_color_distance(color, mask);
58 +
59 +     if (distance <= threshold) {
60 +         *pixel = 0;
61 +     }
62 + }
63 +
64 + return;
65 +}

```

上述函数实际上完成了把背景做成透明的处理。在实际应用中, 背景色应该使用单色背景 (绿色效果最好) 而场景中的人物则不能穿与背景色相近的衣服。

L74 是实际的颜色对比函数, 它被实现成 `inline` 的以保证效率。

```

74 +static inline int switch_color_distance(switch_rgb_color_t *c1, switch_rgb_color_t *c2)
75 +{
76 +     int rmean = ( c1->r + c2->r ) / 2;
77 +     int r = c1->r - c2->r;
78 +     int g = c1->g - c2->g;
79 +     int b = c1->b - c2->b;
80 +
81 +     return sqrt((((512+rmean)*r*r)>>8) + 4*g*g + (((767-rmean)*b*b)>>8));
82 +}

```

L91 被替换成了 L92, 它维护了一个 FreeSWITCH 内部图像格式与 FOURCC² 的一个对应关系。用于图像转换。

```

91 -     case SWITCH_IMG_FMT_ARGB:      fourcc = (uint32_t)FOURCC_ANY ; break;
92 +     case SWITCH_IMG_FMT_ARGB:      fourcc = (uint32_t)FOURCC_BGRA; break;

```

²<http://www.fourcc.org/yuv.php>。

6.1.2 mod_video_filter

为了在 FreeSWITCH 内部对图像进行处理，我们实现了一个 `mod_video_filter` 模块。这部分代码可以用 `git show a0a7b41` 命令查看。

该模块的主要文件是 `mod_video_filter.c`，为了阅读方便，我们不以 `diff` 形式显示，而以原始文件的方式列出。

L37 ~ L39，定义了模块的加载和卸载函数。

```

33 #include <switch.h>
34
35 switch_loadable_module_interface_t *MODULE_INTERFACE;
36
37 SWITCH_MODULE_LOAD_FUNCTION(mod_video_filter_load);
38 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_video_filter_shutdown);
39 SWITCH_MODULE_DEFINITION(mod_video_filter, mod_video_filter_load, mod_video_filter_shutdown, NULL);

```

定义一个结构体，用于描述相关的环境（`context`）。其中，`bgimg` 是一个背景图片（如央视演播室或者烟台的海滩）；当然，如果不提供图片也可以提供一个背景色（`bgcolor`）；`mask` 为画布的颜色，而 `session` 为当前通话的 `session`。

```

41 typedef struct chromakey_context_s {
42     int threshold;
43     switch_image_t *bgimg;
44     switch_rgb_color_t bgcolor;
45     switch_rgb_color_t mask;
46     switch_core_session_t *session;
47 } chromakey_context_t;

```

L49 ~ L54，初始化当前的 `context`。

```

49 static void init_context(chromakey_context_t *context)
50 {
51     switch_color_set_rgb(&context->bgcolor, "#000000");
52     switch_color_set_rgb(&context->mask, "#FFFFFF");
53     context->threshold = 300;
54 }

```

L56 ~ L59，释放资源。其中 `switch_img_free` 会检查空指针，即如果传入一个空指针也不会出错。

```
56 static void uninit_context(chromakey_context_t *context)
57 {
58     switch_img_free(&context->bging);
59 }
```

解析命令行参数，设置 `context` 相关的值。

```
61 static void parse_params(chromakey_context_t *context, int start, int argc, char **argv)
62 {
63     int n = argc - start;
64     int i = start;
65
66     if (n > 0 && argv[i]) { // color
67         switch_color_set_rgb(&context->mask, argv[i]);
68     }
69
70     i++;
71
72     if (n > 1 && argv[i]) { // thresh
73         int thresh = atoi(argv[i]);
74
75         if (thresh > 0) context->threshold = thresh;
76     }
77
78     i++;
79
80     if (n > 2 && argv[i]) {
81         if (argv[i][0] == '#') { // bgcolor
82             switch_color_set_rgb(&context->bgcolor, argv[i]);
83         } else {
84             if (!context->bging) {
85                 context->bging = switch_img_read_png(argv[i], SWITCH_IMG_FMT_ARGB);
86             }
87         }
88     }
89 }
```

接下来是一个回调函数。该函数对收到的每一帧图像都进行处理，并返回处理后的结果。

```
91 static switch_status_t video_thread_callback(switch_core_session_t *session, switch_frame_t *frame, void
↪ *user_data)
92 {
93     chromakey_context_t *context = (chromakey_context_t *)user_data;
```

```
94     switch_channel_t *channel = switch_core_session_get_channel(session);
95     switch_image_t *img = NULL;
96     void *data = NULL;
97
98     if (!switch_channel_ready(channel)) {
99         return SWITCH_STATUS_FALSE;
100    }
101
102    if (!frame->img) {
103        return SWITCH_STATUS_SUCCESS;
104    }
```

L106, 申请内存用于存放 ARGB 图像。

L109, 将 FreeSWITCH 收到的当然图像转换为 ARGB 色彩空间, 数据存放到 `data` 里。FreeSWITCH 中的图像都是 YUV I420 格式的, 因而需要一个转换。(还记得上一节 `diff` 中的 L92 吗?)

L110, 把内存中的图像数据包装成一个新图像 `img`, 相当于产生了一个临时图像, 该图像是 ARGB 格式的。

L112, 对图像进行处理, 把背景色变成透明的。

```
106     data = malloc(frame->img->d_w * frame->img->d_h * 4);
107     switch_assert(data);
108
109     switch_img_to_raw(frame->img, data, frame->img->d_w * 4, SWITCH_IMG_FMT_ARGB);
110     img = switch_img_wrap(NULL, SWITCH_IMG_FMT_ARGB, frame->img->d_w, frame->img->d_h, 1, data);
111     switch_assert(img);
112     switch_img_chromakey(img, &context->mask, context->threshold);
```

如果设置了背景图像, 则把背景图像先叠加到原来的图像上 (L115, 即覆盖原来的图像)。注意, 这里, 应该保证背景图你足够大, 否则, 可能不足以盖住原始图像。当然, 这里可以多加一些代码根据情况对图你进行缩放等, 在此没有实现。

如果没有背景图像, 则把原图像变成单色的 (L117)。

```
114     if (context->bgimg) {
115         switch_img_patch(frame->img, context->bgimg, 0, 0);
116     } else {
117         switch_img_fill(frame->img, 0, 0, img->d_w, img->d_h, &context->bgcolor);
118     }
```

准备好背景后, 将临时的处理过的透明的 `img` 贴到背景图像上 (L120), 释放临时图像 (L121), 并释放临时存储区 (L122)。

```
120     switch_img_patch(frame->img, img, 0, 0);
121     switch_img_free(&img);
122     free(data);
123
124     return SWITCH_STATUS_SUCCESS;
125 }
```

下面也是个回调函数，它是 Media Bug 的回调。Media Bug 是 FreeSWITCH 中用于在中途截获媒体流的一种方式。如果在一个 Channel 上装了 Media Bug，则每一帧音频或视频数据都会回调一个回调函数，如 L127 的回调函数。

L134 是 Media Bug 刚刚安装上时执行的每一个回调，它会在 L136 设置一个参数，强制对该 Channel 的视频进行解码。VIDEO_DECODE_READ 表示对读（收）到的视频进行解码。

当然，在解除该 Media Bug 时（L139）会释放相应的锁（L141）并解除解码标志（L142），释放相应资源（L143）。

```
127 static switch_bool_t chromakey_bug_callback(switch_media_bug_t *bug, void *user_data, switch_abc_type_t
↪ type)
128 {
129     chromakey_context_t *context = (chromakey_context_t *)user_data;
130
131     switch_channel_t *channel = switch_core_session_get_channel(context->session);
132
133     switch (type) {
134     case SWITCH_ABC_TYPE_INIT:
135     {
136         switch_channel_set_flag_recursive(channel, CF_VIDEO_DECODED_READ);
137     }
138     break;
139     case SWITCH_ABC_TYPE_CLOSE:
140     {
141         switch_thread_rwlock_unlock(MODULE_INTERFACE->rwlock);
142         switch_channel_clear_flag_recursive(channel, CF_VIDEO_DECODED_READ);
143         uninit_context(context);
144     }
145     break;
```

对于解码后的每一帧视频，都会以 SWITCH_ABC_TYPE_READ_VIDEO_PING 参数回调（L146），此时，可以取到这一帧（frame，L149），然后对这一帧执行上面讲到的 L91 定义的回调函数 video_thread_callback 对视频图像进行处理并替换。

```

146     case SWITCH_ABC_TYPE_READ_VIDEO_PING:
147     case SWITCH_ABC_TYPE_VIDEO_PATCH:
148         {
149             switch_frame_t *frame = switch_core_media_bug_get_video_ping_frame(bug);
150             video_thread_callback(context->session, frame, context);
151         }
152     break;
153 default:
154     break;
155 }
156
157 return SWITCH_TRUE;
158 }

```

L160 是一个宏，定义了一个参数语法格式。

L161 实现了一个 APP，用于往一个 Channel 上安装 Media Bug。

```

160 #define CHROMAKEY_APP_SYNTAX "<#mask_color> [threshold] [#bg_color|path/to/image.png]"
161 SWITCH_STANDARD_APP(chromakey_start_function)
162 {
163     switch_media_bug_t *bug;
164     switch_status_t status;
165     switch_channel_t *channel = switch_core_session_get_channel(session);
166     char *argv[4] = { 0 };
167     int argc;
168     char *lbuf;
169     switch_media_bug_flag_t flags = SMBF_READ_VIDEO_PING | SMBF_READ_VIDEO_PATCH;
170     const char *function = "chromakey";
171     chromakey_context_t *context;

```

首先检查该 Channel 上是否已经有了一个 Bug (L173)，该 Bug 以 `_chromakey_bug` 作为唯一标志。如果调用参数为 `stop` (L174)，则清除 Media Bug (L175 ~ L176)，否则，打印错误日志 (L178) 并退出 (L179)。

```

173     if ((bug = (switch_media_bug_t *) switch_channel_get_private(channel, "_chromakey_bug"))) {
174         if (!zstr(data) && !strcasecmp(data, "stop")) {
175             switch_channel_set_private(channel, "_chromakey_bug", NULL);
176             switch_core_media_bug_remove(session, &bug);
177         } else {
178             switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_WARNING, "Cannot run 2
↳ chromakey at once on the same channel!\n");
179         }

```

```

180     return;
181 }

```

L183 等待视频就绪。

L185 初始化一个 `context` 用于描述当前的场景数据，并进行一些适当的初始化（L187 ~ 189）。

L191 ~ L193 解析 APP 的参数，并设置到 `context` 中。

```

183     switch_channel_wait_for_flag(channel, CF_VIDEO_READY, SWITCH_TRUE, 10000, NULL);
184
185     context = (chromakey_context_t *) switch_core_session_alloc(session, sizeof(*context));
186     switch_assert(context != NULL);
187     memset(context, 0, sizeof(*context));
188     init_context(context);
189     context->session = session;
190
191     if (data && (lbuf = switch_core_session_strdup(session, data))
192         && (argc = switch_separate_string(lbuf, ' ', argv, (sizeof(argv) / sizeof(argv[0])))) {
193         parse_params(context, 1, argc, argv);
194     }

```

L196，锁住当前的 INTERFACE，以避免该模块被卸载。

L198，安装 Media Bug 回调函数，并以当前的 `context` 作为参数传入。安装成功后，核心就会按部就班的在适当的时候回该回调函数。

L204，记住这个 Media Bug，以便能在回调函数中取到。

```

196     switch_thread_rwlock_rdlock(MODULE_INTERFACE->rwlock);
197
198     if ((status = switch_core_media_bug_add(session, function, NULL, chromakey_bug_callback, context,
199 ↪ 0, flags, &bug)) != SWITCH_STATUS_SUCCESS) {
200         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Failure!\n");
201         switch_thread_rwlock_unlock(MODULE_INTERFACE->rwlock);
202         return;
203     }
204     switch_channel_set_private(channel, "_chromakey_bug_", bug);
205 }

```

APP 的使用方法是在 Dialplan 中，如：

```
<action application="chromakey" data="#00FF00 60 /tmp/background.png"/>
```

这样便能安装一个 Media Bug，对收到的图像去掉绿幕（#00FF00 为绿色），容差为 60，并贴到背景图像 .png 上。

下面，实现了一个 API，用于动态的添加和删除 Media Bug，如

```
freeswitch> chromakey <uuid> start #00FF00 60 /tmp/background.png  
freeswitch> chromakey <uuid> stop
```

L209，是函数定义。L221，定义了 Media Bug 的类型，我们只关心视频相关的 Media Bug。

```
207 /* API Interface Function */  
208 #define CHROMAKEY_API_SYNTAX "<uuid> [start|stop] " CHROMAKEY_APP_SYNTAX  
209 SWITCH_STANDARD_API(chromakey_api_function)  
210 {  
211     switch_core_session_t *rsession = NULL;  
212     switch_channel_t *channel = NULL;  
213     switch_media_bug_t *bug;  
214     switch_status_t status;  
215     chromakey_context_t *context;  
216     char *mycmd = NULL;  
217     int argc = 0;  
218     char *argv[25] = { 0 };  
219     char *uuid = NULL;  
220     char *action = NULL;  
221     switch_media_bug_flag_t flags = SMBF_READ_VIDEO_PING | SMBF_READ_VIDEO_PATCH;  
222     const char *function = "chromakey";  
223  
224     if (zstr(cmd)) {  
225         goto usage;  
226     }  
227  
228     if (!(mycmd = strdup(cmd))) {  
229         goto usage;  
230     }  
231  
232     if ((argc = switch_separate_string(mycmd, ' ', argv, (sizeof(argv) / sizeof(argv[0]))) < 2) {  
233         goto usage;  
234     }  
235  
236     uuid = argv[0];  
237     action = argv[1];
```

L239, 通过 `uuid` 取得 Session, 进而取得 Channel (L244)。

```
239     if (!(rsession = switch_core_session_locate(uuid))) {
240         stream->write_function(stream, "-ERR Cannot locate session!\n");
241         goto done;
242     }
243
244     channel = switch_core_session_get_channel(rsession);
```

类似于 APP, 如果 Bug 已存在 (L246), 则或者停止 (L248), 或者更新参数 (L252 ~ L255)。

```
246     if ((bug = (switch_media_bug_t *) switch_channel_get_private(channel, "_chromakey_bug_"))) {
247         if (!zstr(action)) {
248             if (!strcasecmp(action, "stop")) {
249                 switch_channel_set_private(channel, "_chromakey_bug_", NULL);
250                 switch_core_media_bug_remove(rsession, &bug);
251                 stream->write_function(stream, "+OK Success\n");
252             } else if (!strcasecmp(action, "start")) {
253                 context = (chromakey_context_t *) switch_core_media_bug_get_user_data(bug);
254                 switch_assert(context);
255                 parse_params(context, 2, argc, argv);
256                 stream->write_function(stream, "+OK Success\n");
257             }
258         } else {
259             stream->write_function(stream, "-ERR Invalid action\n");
260         }
261         goto done;
262     }
```

如果当前 Channel 上, Media Bug 不存在, 则初始化一个 `context` (L268), 并安装一个 (L277)。

凡是通过 `switch_core_session_locate` 获取到的 Session 都自动获得一个读锁, 因而用完后要释放相关的锁 (L293)。

```
264     if (!zstr(action) && strcmp(action, "start")) {
265         goto usage;
266     }
267
268     context = (chromakey_context_t *) switch_core_session_alloc(rsession, sizeof(*context));
269     switch_assert(context != NULL);
```



```

270     context->session = rsession;
271
272     init_context(context);
273     parse_params(context, 2, argc, argv);
274
275     switch_thread_rwlock_rdlock(MODULE_INTERFACE->rwlock);
276
277     if ((status = switch_core_media_bug_add(rsession, function, NULL,
278                                             chromakey_bug_callback, context, 0, flags, &bug)) !=
↪ SWITCH_STATUS_SUCCESS) {
279         stream->write_function(stream, "-ERR Failure!\n");
280         switch_thread_rwlock_unlock(MODULE_INTERFACE->rwlock);
281         goto done;
282     } else {
283         switch_channel_set_private(channel, "_chromakey_bug_", bug);
284         stream->write_function(stream, "+OK Success\n");
285         goto done;
286     }
287
288 usage:
289     stream->write_function(stream, "-USAGE: %s\n", CHROMAKEY_API_SYNTAX);
290
291 done:
292     if (rsession) {
293         switch_core_session_rwlock_unlock(rsession);
294     }
295
296     switch_safe_free(mycmd);
297     return SWITCH_STATUS_SUCCESS;
298 }

```

下面是模块卸载 (L301) 和加载 (L306) 时的回调函数。函数加载时, 通过 [SWITCH_ADD_APP](#) (L315) 和 [SWITCH_ADD_API](#) (L318) 向核心中注册 APP 和 API, 并设置命令行自动补全规则 (L320)。

```

301 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_video_filter_shutdown)
302 {
303     return SWITCH_STATUS_SUCCESS;
304 }
305
306 SWITCH_MODULE_LOAD_FUNCTION(mod_video_filter_load)
307 {
308     switch_application_interface_t *app_interface;
309     switch_api_interface_t *api_interface;
310
311     /* connect my internal structure to the blank pointer passed to me */
312     *module_interface = switch_loadable_module_create_module_interface(pool, modname);

```

```

313     MODULE_INTERFACE = *module_interface;
314
315     SWITCH_ADD_APP(app_interface, "chromakey", "chromakey", "chromakey bug",
316                     chromakey_start_function, CHROMAKEY_APP_SYNTAX, SAF_NONE);
317
318     SWITCH_ADD_API(api_interface, "chromakey", "chromakey", chromakey_api_function,
319     ↪ CHROMAKEY_API_SYNTAX);
319
320     switch_console_set_complete("add chromakey ::console::list_uuid ::[start:stop]");
321
322     return SWITCH_STATUS_SUCCESS;
323 }

```

这是一个典型的使用 Media Bug 在 FreeSWITCH 中进行视频处理的例子。其中也用到了一些视频处理的函数，这些函数都比较有代表性。

6.1.3 编译相关

当然，有了代码，还需要让这些代码能顺利地编译。

需要在 `build/modules.conf.in` 文件中增加一行：

“`bash #applications/mod_video_filter`

``configure.ac`` 中增加一行：

```

```bash
src/mod/applications/mod_video_filter/Makefile

```

以及增加一个 `Makefile.am` 文件：

```

include $(top_srcdir)/build/modmake.rulesam
MODNAME=mod_video_filter

mod_LTLIBRARIES = mod_video_filter.la
mod_video_filter_la_SOURCES = mod_video_filter.c
mod_video_filter_la_CFLAGS = $(AM_CFLAGS)
mod_video_filter_la_LIBADD = $(switch_builddir)/libfreeswitch.la
mod_video_filter_la_LDFLAGS = -avoid-version -module -no-undefined -shared -lm -lz

```

`bootstrap.sh` 中增加：

---

```
freeswitch-mod-video_filter (= \${binary:Version}),
```

---

另外, 为了让它能正常的进入 FreeSWITCH 的发布包, 还需要在 `debian/control-modules` 以及 `freeswitch.spec` 中增加相关的设置, 详见 `git show c9aa3522`。

### 6.1.4 精彩继续

接下来, Anthony 又提交了一个补丁 `96e823b`, 支持背景图片的缩放。

L15, 增加一个 `bgimg_scaled` 用于存放缩放后的图像。L23, 记得最后要销毁图像。

---

```
12 typedef struct chromakey_context_s {
13 int threshold;
14 switch_image_t *bgimg;
15 + switch_image_t *bgimg_scaled;
16 switch_rgb_color_t bgcolor;
17 switch_rgb_color_t mask;
18 switch_core_session_t *session;
19 @@ -56,6 +57,7 @@ static void init_context(chromakey_context_t *context)
20 static void uninit_context(chromakey_context_t *context)
21 {
22 switch_img_free(&context->bgimg);
23 + switch_img_free(&context->bgimg_scaled);
24 }
```

---

L31 ~ L36, 增加相应处理, 在重新解析命令行参数时, 安全释放图像内存。这样, L41 的判断就是没必要了, 直接用 L44 代替。注意, 之有的版本, 解析时是无法替换图像的, 通过这次修改, 就可以通过指定不同的图像路径替换图像了。

---

```
26 static void parse_params(chromakey_context_t *context, int start, int argc, char **argv, const char
↪ **function, switch_media_bug_flag_t *flags)
27 @@ -78,12 +80,17 @@ static void parse_params(chromakey_context_t *context, int start, int argc, char
28 i++;
29
30 if (n > 2 && argv[i]) {
31 + if (context->bgimg) {
32 + switch_img_free(&context->bgimg);
33 + }
34 + if (context->bgimg_scaled) {
35 + switch_img_free(&context->bgimg_scaled);
```

---

```

36 + }
37 +
38 if (argv[i][0] == '#') { // bgcolor
39 switch_color_set_rgb(&context->bgcolor, argv[i]);
40 } else {
41 - if (!context->bgimg) {
42 - context->bgimg = switch_img_read_png(argv[i], SWITCH_IMG_FMT_ARGB);
43 - }
44 + context->bgimg = switch_img_read_png(argv[i], SWITCH_IMG_FMT_I420);
45 }
46 }

```

L53 ~ L59, 如果检测收到到图像分辨率有变化, 则也重新缩放背景图像。

```

48 @@ -121,7 +128,15 @@ static switch_status_t video_thread_callback(switch_core_session_t *session, swi
49 switch_img_chromakey(img, &context->mask, context->threshold);
50
51 if (context->bgimg) {
52 - switch_img_patch(frame->img, context->bgimg, 0, 0);
53 + if (context->bgimg_scaled && (context->bgimg_scaled->d_w != frame->img->d_w || context-
↳ >bgimg_scaled->d_h != frame->img->d_h)) {
54 + switch_img_free(&context->bgimg_scaled);
55 + }
56 +
57 + if (!context->bgimg_scaled) {
58 + switch_img_scale(context->bgimg, &context->bgimg_scaled, frame->img->d_w, frame->img->d_h);
59 + }
60 +
61 + switch_img_patch(frame->img, context->bgimg_scaled, 0, 0);
62 } else {
63 switch_img_fill(frame->img, 0, 0, img->d_w, img->d_h, &context->bgcolor);
64 }

```

罗马不是一日建成的, 功能也是这么一点一点的加上去的。

### 6.1.5 永无止境

上面功能虽然做得差不多了, 但是实际的效果不甚理想。后来, Anthony 又增加了同时去掉多种颜色的功能。当然, 带来的后果是计算量非常大。这次提交见: [c60ae0f](https://c60ae0f), 可以移步到这里查看:

<https://freeswitch.org/fisheye/changelog/freeswitch?cs=c60ae0f0e11e761dd43d75bf9979a47721ab1f64>

### 6.1.6 小结

至于这部分代码后续会变成什么样，我们是不可预测的。因为历史是不断向前发展的。得益于 Git 强大的功能，我们可以随时查看代码的历史（根据多维空间原理，这只有在五维空间里才做得得到）。通过本章，能给大家带来一些新的视角，从另一个角度和维度看代码，希望能给广大读者带来一些新的收获。

最后，让我们走进直播间，一起看一看 Ken Rice 大侠的海边演播室吧。



图 6.1: Ken Rice 的海边演播室

## 6.2 使用 Perf 定位性能问题

FreeSWITCH 官方开源了两个辅助库 `libks`<sup>3</sup>和 `signalwire-c`<sup>4</sup>。我们用这个库开发了一个模块，但在使用过程中发现模块在什么都不做的情况下空转就使用了大约 2% 的 CPU。只用肉眼看代码找问题比较困难，所以，我们找了一个 Perf<sup>5</sup>工具帮助定位问题。

Perf 的说明是「DTrace-like tools for Linux」，在 Debian 上直接可以通过以下命令安装：

---

```
apt-get install perf-tools-unstable
```

---

<sup>3</sup><https://github.com/signalwire/libks>

<sup>4</sup><https://github.com/signalwire/signalwire-c>

<sup>5</sup><https://dev.to/etcwilde/perf---perfect-profiling-of-cc-on-linux-of>

由于是在 Docker 中运行的，使用如下命令调整内核参数：

```
echo 0 > /proc/sys/kernel/kptr_restrict
```

运行程序以获取相关数据

```
perf record ./my_test
```

程序运行几秒后，退出。然后执行

```
perf report
```

得到以下数据：

Overhead	Command	Shared Object	Symbol
24.73%	swclt_hmgr_init	libks.so.1	[.] ks_atomic_increment_uint32
22.66%	swclt_hmgr_init	libks.so.1	[.] ks_atomic_decrement_uint32
15.96%	swclt_hmgr_init	libks.so.1	[.] ks_spinlock_release
6.29%	swclt_hmgr_init	libks.so.1	[.] ks_spinlock_try_acquire
5.96%	swclt_hmgr_init	libks.so.1	[.] ks_handle_enum_type
3.56%	swclt_hmgr_init	libks.so.1	[.] __unlock_slot
2.73%	swclt_hmgr_init	libks.so.1	[.] __try_lock_slot
1.49%	swclt_hmgr_init	libjemalloc.s	[.] malloc
0.91%	swclt_hmgr_init	[kernel]	[k] 0xfffffffffb2044882
0.74%	swclt_hmgr_init	[kernel]	[k] 0xfffffffffb20fda30
0.66%	swclt_hmgr_init	[kernel]	[k] 0xfffffffffb2115523
0.33%	infrastructure_	ld-2.24.so	[.] do_lookup_x

当然，如果加上 `-g` 参数执行的话，能得到更详细的数据，如：

```
perf record -g ./mytest
```

`perf report` 输出如下：

Samples: 161 of event 'cpu-clock', Event count (approx.): 40250000

Children	Self	Command	Shared Object	Symbol
+ 61.49%	0.00%	swclt_hmgr_init	libks.so.1	[.] thread_launch
+ 61.49%	0.00%	swclt_hmgr_init	libpthread-2.24.so	[.] start_thread
+ 59.63%	0.00%	swclt_hmgr_init	libsignalwire_client.so.1	[.] __service_handle_type
+ 59.63%	0.00%	swclt_hmgr_init	libsignalwire_client.so.1	[.] __service_handles
+ 59.63%	0.00%	swclt_hmgr_init	libsignalwire_client.so.1	[.] __manager_loop
+ 59.63%	0.00%	swclt_hmgr_init	libsignalwire_client.so.1	[.] __manager_thread_wrapper
+ 56.52%	1.86%	swclt_hmgr_init	libks.so.1	[.] ks_handle_enum_type
+ 33.54%	1.24%	swclt_hmgr_init	libks.so.1	[.] __try_lock_slot
+ 32.30%	3.11%	swclt_hmgr_init	libks.so.1	[.] ks_spinlock_try_acquire
+ 28.57%	18.01%	swclt_hmgr_init	libks.so.1	[.] ks_atomic_increment_uint32
+ 18.63%	1.24%	swclt_hmgr_init	libks.so.1	[.] __unlock_slot
+ 18.01%	5.59%	swclt_hmgr_init	libks.so.1	[.] ks_spinlock_release
+ 14.29%	0.62%	swclt_hmgr_init	[kernel.kallsyms]	[k] handle_mm_fault
+ 13.66%	13.04%	swclt_hmgr_init	libks.so.1	[.] ks_atomic_decrement_uint32
+ 13.04%	0.00%	swclt_hmgr_init	[kernel.kallsyms]	[k] __do_page_fault
+ 13.04%	0.00%	swclt_hmgr_init	[kernel.kallsyms]	[k] page_fault
+ 11.80%	0.00%	swclt_hmgr_init	[kernel.kallsyms]	[k] entry_SYSCALL_64_after_swapgs
+ 9.94%	0.00%	swclt_hmgr_init	[kernel.kallsyms]	[k] prepare_exit_to_usermode
+ 8.70%	0.00%	swclt_hmgr_init	[kernel.kallsyms]	[k] __switch_to_asm
+ 8.07%	0.00%	swclt_hmgr_init	[kernel.kallsyms]	[k] __perf_event_task_sched_in

从上面的数据可以看出，CPU 花了大量的时间在运行 `ks_atomic_increment_uint32` 等，然后再对照代码，顺着 `ks_handle_enum_type`，发现一段热点代码：

```
for (uint32_t slot_index = KS_HANDLE_SLOT_INDEX_FROM_HANDLE(*handle) + 1;
 slot_index < KS_HANDLE_MAX_SLOTS; slot_index++)
```

这段代码在我们的应用里每秒钟会执行好几次，而在这里面就会执行 `ks_atomic_increment_uint32`。`KS_HANDLE_MAX_SLOTS` 值为 65535，所以每秒钟会执行 `65535 * n` 次，怪不得这么热。

找到问题所在，修复就简单了。修复代码参见<https://github.com/signalwire/libks/pull/47/files>。

虽然这部分代码本身跟 FreeSWITCH 无关，但是我们是在 FreeSWITCH 模块中用的，另外，相信这两个库也会在不远的将来有更大的用处。当然，Perf 工具也可以用于 FreeSWITCH 性能检测。

### 6.3 测试最新版的 FFmpeg

FreeSWITCH 的 `mod_av` 模块使用了 FFmpeg 库，最近遇到一新问题，需要深入调试 FFmpeg 的代码才能找到原因。我的系统是 Debian 10 Buster 版，由于系统上已经安装了 FFmpeg，因此自己编译的库会与

系统库有冲突。以前也解决过这个问题，参见<https://mp.weixin.qq.com/s/5IZqXiGUQ22S4LXx1hs10w>。不过，这次我并不想重新运行 `configure` 脚本，找了个偷懒的解决方案。

首先 Clone 源代码：

---

```
git clone https://github.com/Ffmpeg/Ffmpeg.git
```

---

编译。仅仅选了我需要用到的模块。

---

```
./configure --enable-libx264 --enable-shared --enable-gpl --disable-stripping
```

---

好在最新的版本与系统版本是兼容的，因此我不需要再重新编译 `mod_av`，只需要使用如下命令重新启动 FreeSWITCH：

---

```
LD_LIBRARY_PATH="../ffmpeg/libavcodec;../ffmpeg/libavformat;../ffmpeg/libavutil;../ffmpeg/libaswsample;../
↪ ffmpeg/libswscale" /usr/local/freeswitch/bin/freeswitch -nonat
```

---

其中，将 FreeSWITCH 源代码放到与 FFmpeg 源代码平行的目录中，以便用相对路径就能找到。`LD_LIBRARY_PATH` 指定在 FreeSWITCH 启动时优先查找的库目录。

这样，修改一个 FFmpeg 的源代码，重启 FreeSWITCH，就可以看到我修改的内容的。

经过一番查找，终于找到了 Bug 的原因。

这次我要查找的问题是：FreeSWITCH 在播放 HLS 流有时会卡住，但一直找不到原因，经过一番查找，问题定位到在播放过程中如果连接失败，则有可能卡住。并成功重现了问题。

如何重现呢？在播放过程中，使用如下命令，禁止 FreeSWITCH 访问我要播放的 HLS 的端口，导致网络不通，就可以重现（简单起见仅限制了端口，如果是个通用的端口（如 80），最好加上 IP 限制）：

---

```
iptables -I OUTPUT -p tcp --dport 8880 -j DROP
```

---

当然，事后要记得清除那个 `iptables` 规则，否则根本就联不上了。

问题重现后，查找问题就方便了，由于对 FFmpeg 代码不熟悉，只好找关键的地方加 Log。最后定位到 `interrupt_callback` 没有正确回调。这里有个坑，因为，`mod_av` 的代码里是这样实现的：



---

```
avformat_open_input(&context->fc, filename, NULL, NULL));
context->fc->interrupt_callback.callback = interrupt_cb;
```

---

即，`context->fc` 初始化以后再设置回调，但 HLS 模块的回调却是在 `open_input` 的时候加上去的，这就导致了回调加不上，因此 FreeSWITCH 在网络阻塞的情况下无法中断，导致 Channel 阻塞住无法挂机。

更坑的是 FFmpeg 好像没有什么解决方案。只好查源代码，想办法绕过去了。

解决思路是先初始化 `context->fc`，设置好回调后再 `open`，即：

---

```
context->fc = avformat_alloc_context();
context->fc->interrupt_callback.callback = interrupt_cb;
avformat_open_input(&context->fc, filename, NULL, NULL));
```

---

这样就能正常设置中断回调，不会再卡死了。

算是个办法吧。

另外，得出一个结论，FreeSWITCH 1.10.3 的代码用最新版的 FFmpeg 是没问题的（我测的 `master` 版是 `353aecbb`，最新的发行版应该是 2019 年 12 月份发布的 `4.2.2`）。如果在 CentOS 或其它系统上有比较旧版本的 FFmpeg，可以升到最新的版本一试。

与诸君共勉。

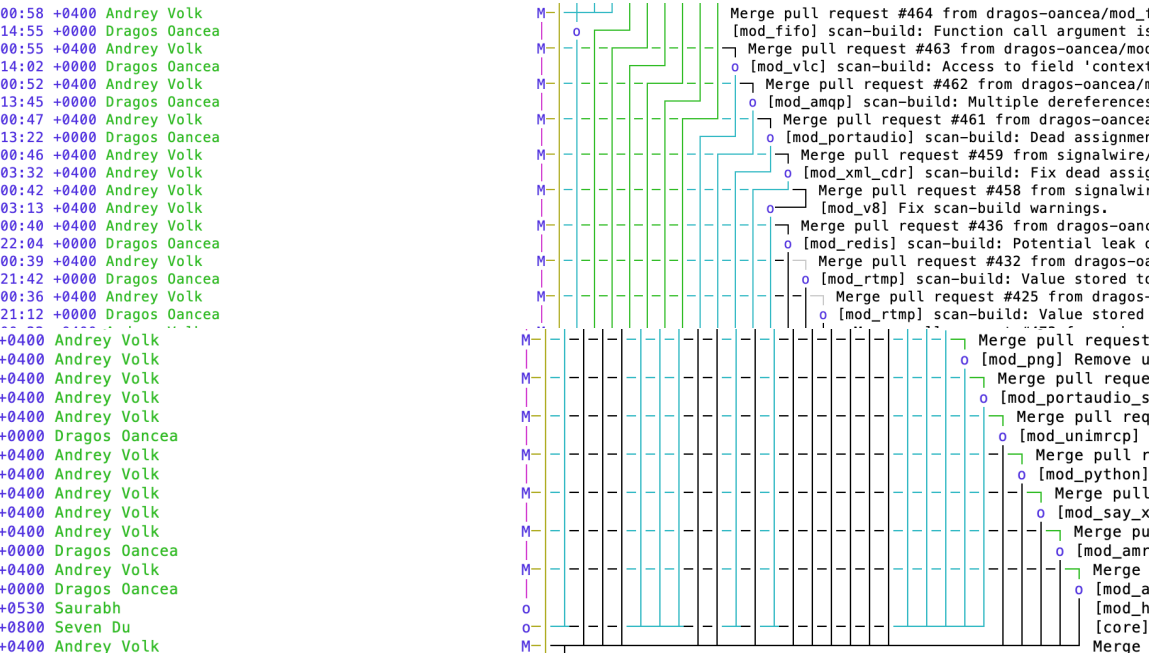
## 6.4 如何维护自己的 FreeSWITCH 分支

在开发中很多情况下需要维护自己的 FreeSWITCH 分支，但同时又要跟上游的 FreeSWITCH 代码同步，如果每次都使用 `merge` 来合并上游代码，势必造成本地的分支不清晰，日后再往上游合并的话就比较难了。当然，即使不考虑向上游合并，`merge` 也会打乱时间线，假以时日，你再比较你本地分支与上游分支的区别就很难了。

对比一下以下时间线，虽然看起来比较酷，但维护起来还是比较崩溃的：

```
11:45 +0800 Seven Du
11:43 +0800 Seven Du
16:23 +0400 Andrey Volk
17:44 +0800 paxc
14:58 -0400 CAUCA 9-1-1
21:54 +0400 Andrey Volk
20:08 +0400 Andrey Volk
16:46 +0400 Andrey Volk
09:50 +0000 Andrey Volk
03:33 +0400 Andrey Volk
02:56 +0400 Andrey Volk
01:55 +0400 Andrey Volk
01:43 +0800 Andrey Volk

o [fix-openh264] {seven/fix-openh264} [mod_openh264] fix conversion
o [mod_openh264] fix build warning of Dead store Dead assignment
o {xx/master} [Testing] Enable openh264 on drone
o [mod_openh264] add unit test and fix stap-a size issue
o [sofia-sip] Add urn: scheme support to sofia. (#445)
M Merge pull request #626 from signalwire/sofia-wss-keep-alive
o [sofia-sip] Timed out or not established wss should be destroyed
o version bump
o swigall
M {up/master} Merge pull request #619 from signalwire/amr_dead_ass:
o [mod_amr] scan-build: fix dead increment of ffmtmp_pos in switch
M Merge pull request #133 from jay98237438/master
```



那么，有没有更好的方法呢？有，我们一直用 **rebase** 方法维护自己的分支。

**rebase** 就是最好的方法吗？不是的。与 **merge** 相比，**rebase** 每次都需要解决冲突，而 **merge** 一般只需要解决一次。这也是为什么 **merge** 方法的时间线会比较混乱，而 **rebase** 就更清晰些。

当然，**rebase** 更大的问题是每次都会重写历史，向远端需要 **push -f**，在团队开发时可能会引起混乱（网传某团队成员因为经常 **push -f** 被队友枪杀，可见是高危动作）。

但对于对于与上游同步有极致追求的团队来说，**rebase** 只是唯一选择，也是可行的，下面说说我们的实践。

6.4.1 基础知识

首先，要熟悉 Git 的基本概念和基本操作。下在以 FreeSWITCH Github 上的仓库为例。

FreeSWITCH 的 Git 仓库地址为：<https://github.com/signalwire/freeswitch>，上述地址可以直接 Clone，Clone 后你就有了个本地目录：**freeswitch**，默认分支是 **master** 分支。

```
远端： | signalwire/freeswitch 分支： master|
本地： | 分支： master |
```

假设这时候仓库中只有 10 个 Commit，提交历史为：

```
commit-10
commit-9
```

```
commit-8
...
```

---

这时候你想打个补丁，因此启动了一个新分支：

```
git checkout -b fix-1
```

---

注意这时候 **fix-1** 分以与远程的 **master** 分支以及本地的 **master** 分支是一模一样的。

打一个补丁，进行了一次 **commit**。本地的 **fix-1** 分支变为：

```
commit-11 (新)
commit-10
commit-9
commit-8
...
```

---

OK，检查无误后，你向上游提交你的分支（「注：1」，详见后文）：

```
git push origin fix-1
```

---

其中，**origin** 就代表一个远端（**remote**）的仓库，是你在 Clone 的时候默认生成的。上述命令将你本的 **fix-1** 分支推到远端了。

```
远端： | signalwire/freeswitch 分支： master, fix-1 |
本地： | 分支： master, fix-1 | remote: origin
```

---

然后你就提一个 Pull Request（合并请求，简称 PR），请求上游的维护者把你的修改合并到上游的主分支里去（**master**）。

不过，现实情况不是这样，因为一般来说你没有上游仓库的 Push 权限，因此，「注：1」标注的地方无法实现。

OK，需要想别的办法了。

Github 支持远端 Fork，因此，你可以把远端的仓库复制一份到你自己的名下，比如 **rts/freeswitch**，其中 **rts** 是我们 RTS 社区的账号，你可以有你自己的名字。

Fork 后，你有了一个跟上游一模一样的仓库，但是为了能维护你的仓库，需要添加一个远端地址（**remote URL**）：

---

```
git remote add rts https://github.com/rts/freeswitch
git remote add rts git@github.com:rts/freeswitch
```

---

以上命令任选其一。这个上游的地址可以在 Github 上找到，前者是 [https](https://github.com/rts/freeswitch) 的，每次都需要输入密码，后者是 SSH，可以直接使用 Public Key 登录，不需要每次输入密码。

设置了上游后，就可以 Push 你的代码了：

---

```
git push rts fix-1
```

---

这时候，状态变成了如下的样子（「注：2」）：

---

```
远端： | signalwire/freeswitch 分支：master |
远端： | rts/freeswitch 分支：master, fix-1|
本地： | 分支：master, fix-1 | remote: origin, rts
```

---

你将本地的 **fix-1** 分支提交到你自己的远端仓库（**rts/freeswitch**）里，然后再向 [signalwire/freeswitch](https://github.com/rts/freeswitch) 发 PR。

这时候，上游维护人员就会 Review 代码，并向你提出修改建议，如代码格式，添加测试案例等。你根据建议，又提了两个提交，这时候本地 **fix-1** 分支变为：

---

```
commit-13 (新)
commit-12 (新)
commit-11 (新)
commit-10
commit-9
commit-8
...
```

---

然后你又 Push 到你自己的仓库：

---

```
git push rts fix-1
```

---

上游维护人员说这次可以了，不过这三个 Commit 没必要，因为改来改去比较乱，让你合并成一个 Commit。这就用到一个操作，叫 Squash。笔者一般使用 `git rebase -i` 操作，详细步骤请参阅 Git 相关文档。

---

```
git rebase -i commit-9
```

---

fix-1 分支又变成了如下的样子，不过现在的 `commit-11` 是原来三个 Commit 合并（Squash）后的结果：

---

```
commit-11 (新)
commit-10
commit-9
commit-8
...
```

---

如果这时候你再向远端 Push，会被拒绝，因为本地跟远端不一致了。这时候，可以用 `push -f`。

---

```
git push -f rts fix-1
```

---

注意，`push -f` 是比较危险的操作，但是你现在可以先忽略这个危险。如果你在一个团队中工作，问问有经验的同事你在什么情况下可以 `push -f`。

由于 `rebase` 会改变时间线，改变提交历史，因此，我们上面用 `push -f` 把远程分支更新的跟本地一样了。在此过程中，PR 还是与你的 `fix-1` 分支关联，**不需要** 关闭原 PR 重提一个新的（当然这要看上游的维护策略。新手往往会重提一个新的，可以那样做，但是没必要，另外，原来的 PR 上可能有沟通记录，所以另起一个 PR，会丢失沟通记录，即使可以将 PR 进行关联，也会显得比较乱）。

好的，你的 PR 被愉快地合并了，你可以在本地和远程删除这个 `fix-1` 分支了，然后再回到 `master`，并更同步上游的最新版本。

---

```
git checkout master
git pull
```

---

在「注：2」的地方，我们也可以删除本地的目录重新 Clone 你自己的仓库：

---

```
git clone git@github.com:rts/freeswitch
cd freeswitch
```

---

这时候添加上游的仓库：

---

```
git remote add upstream https://github.com/signalwire/freeswitch
```

---

---

```
远端: | signalwire/freeswitch 分支: master
远端: | rts/freeswitch 分支: master
本地: | 分支: master | remote: origin, upstream
```

---

不同的是，现在的 **origin** 是指向你自己的远端仓库，而 **upstream** 则指向上游。多试几遍就理解了，**以下假定你是使用这种模式** (生命在于折腾)。

### 6.4.2 冲突解决

你又发现一个 Bug，因此开了 **fix-2** 分支，修改并提交：

---

```
commit-12 (新)
commit-11
commit-10
...
```

---

但这时候，上游变了，有人比你手快，又提交了几个新提交，因此，上游远端的 master 变为：

---

```
commit-x-2
commit-x-1
commit-11
commit-10
...
```

---

如果你的修改与上游没有冲突，还可以按之前的方法提交 PR，只是合并后，可能会变成下面的样子，视你提交的时间不同：

---

```
merge
commit-x-2
commit-x-1
commit-12
commit-11
commit-10
...
```

---

或

---

```
merge
commit-x-2
commit-12
commit-x-1
commit-11
commit-10
...
```

---

合并后，会多一个 `merge` 提交，因为你的 `commit-12` 和别人的 `commit-x-1` 都是基于 `commit-11` 分出来的，最终还是要合并到主线上去。

很是很不幸，你的修改与别人的修改有冲突，所以你需要先在本地修改冲突，否则没法合并。

这时候你可以这样做，在 `fix-2` 分支中：

---

```
git fetch upstream
git rebase upstream/master
```

---

`rebase` 是变基，就是你原来是基于 `commit-11` 修改的，但是远端变了，你想做的是将 `fix-2` 分支更新到 `commit-x-2`，然后再进行修改。当然 `rebase` 过程中会停下来让你手工解决冲突（如果运气好可能可以自动解决），这是考验你 Git 实力的时候。

`rebase` 完成后，你有了一个全新的 `fix-2` 分支，相当于你从上游拉了最新的 `master` 代码，然后再建新分支进行修改。接下来的操作上面都已经学会了。

---

```
git push fix-2
提 PR ...
```

---

注意我们使用第二种模式，这时候 `remote` 参数省略，默认为 `origin`，即你自己的远端仓库（如 `rts/freeswitch`）。

### 6.4.3 维护自己的分支

理解了上述原理后，就可以维护自己的分支了。

维护一个自己的分支，如 `master`

经常同步上游代码：

---

```
git fetch upstream
git rebase upstrea/master
```

---

推到自己的分支

---

```
git push -f origin master
```

---

注意这里还是必须用 `push -f`。

但是，这里出乱子了，因为你的同事也在你的 `master` 分支上工作，而你穿越时空**修改了历史**。所以，在这个操作前，**一定要**通知你的同事，让他们在下次更新代码时，用 `rebase`，如：

---

```
git checkout master
git pull --rebase
```

---

或

---

```
git fetch origin
git rebase origin/master
```

---

一般来说，这个操作都能自动解决冲突，因为你之前已经整理好了。但是所有基于原来的 `master` 分支开出来的新分支都需要 `rebase`，如：

---

```
git checkout some-branch
git fetch origin
git rebase origin/master
```

---

#### 6.4.4 小结

- 用 `rebase` 方法维护自己的分支会使得提交历史清晰，但需要比较深的 Git 功力，以及整个团队都能理解这个流程。
- 在修改代码时多与上游讨论，最大程度地避免冲突。
- 经常将自己的代码通过 PR 的方式提交到上游仓库中，避免出现更多冲突。
- 大段的代码可以放到单独的 `.c` 文件中，`#include` 进来，最大程度避免冲突。
- 多练习，Git 的 `clone/commit/push` 可以很快掌握，`squash/rebase` 没有捷径。
- 本文仅仅关注维护的流程，具体的 Squash/Rebase 等操作还需要查看相关的 Git 文档。



## 6.5 解析 SIP 中携带的 ISUP 消息

ISUP 消息是在 7 号信令中定义的，7 号信令是传统的 PSTN 网络中使用的信令方式。在 7 号信令与 SIP 对接时，就需要一个信令转换的网关。通常来说网关只需要把 7 号信令转换成 SIP 消息即可，但是，在有些情况下，7 号信令希望通过 SIP 网络到达另一个 7 号信令网而不丢失消息，这就用到一个标准，称为 SIP/I 和 SIP/T。SIP/I 是 ITU 定义的，而 SIP/T 是 IETF 定义的，后者也是 SIP 的定义者。两个协议有所不同，但都可以传送 ISUP 消息。

在实际的传送中，由于 SIP 的 INVITE 消息中需要传 SDP，又要携带 ISUP 消息，这就需要用到 **multipart** 消息，即，在一个消息体（Body）中，传输多种类型的消息。

如下图，是在 WireShark 中显示的 INVITE 消息的 Body 部分，可以看到 **Content-Type** 是 **multipart/mixed**，后面的 **boundary** 是不同部分的分隔符。这里有两个部分，即标准的 SDP（**application/sdp**）和 ISUP 部分（**application/isup**）。

```
Content-Type: multipart/mixed;boundary=ssboundary
Content-Length: 383
r Message Body
 ▼ MIME Multipart Media Encapsulation, Type: multipart/mixed, Boundary: "ssboundary"
 [Type: multipart/mixed]
 First boundary: --ssboundary\r\n
 ▶ Encapsulated multipart part: (application/sdp)
 Boundary: \r\n--ssboundary\r\n
 ▶ Encapsulated multipart part: (application/isup)
 Last boundary: \r\n--ssboundary--
```

SDP 是可读的字符串，但 ISUP 部分直接就是二进制的，因而需要一些解析才能识别消息里的内容。

如下面的消息，我们想获取消息中的主被叫号码之类的信息。

```
0000 01 00 40 00 0a 00 02 08 06 01 10 78 56 34 12 0a
0010 08 81 13 31 86 67 45 23 01 28 08 81 10 31 16 32
0020 54 76 08 0b 08 81 10 31 16 32 54 76 08 13 02 00
0030 01 00
```

在 WireShark 中，展示如下：

```

ISDN User Part
 Message Type: Initial address (1)
 ▶ Nature of Connection Indicators : 0x0
 ▶ Forward Call Indicators : 0x4000
 ▶ Calling Party's category : 0xa (ordinary calling subscriber)
 ▶ Transmission medium requirement : 0 (speech)
 ▼ Called Party NumberCalled Party Number: 87654321
 Mandatory Parameter: Called party number (4)
 Pointer to Parameter: 2
 Parameter Length: 6
 0... = Odd/even indicator: even number of address signals
 .000 0001 = Nature of address indicator: subscriber number (national use) (1)
 0... = INN indicator: routing to internal network number allowed
 .001 = Numbering plan indicator: ISDN (Telephony) numbering plan (1)
 ▶ Called Party Number: 87654321
 Pointer to start of optional part: 8
 ▼ Parameter: (t=10, l=8) Calling party number: Calling party numberCalling Party Number: 13687654321
 Optional Parameter: Calling party number (10)
 Parameter Length: 8
 1... = Odd/even indicator: odd number of address signals
 .000 0001 = Nature of address indicator: subscriber number (national use) (1)
 0... = NI indicator: complete
 .001 = Numbering plan indicator: ISDN (Telephony) numbering plan (1)
 00.. = Address presentation restricted indicator: presentation allowed (0)
 11 = Screening indicator: network provided (3)
 ▶ Calling Party Number: 13687654321
 ▶ Parameter: (t=40, l=8) Original called number: Original called numberOriginal Called Number: 13612345678
 ▶ Parameter: (t=11, l=8) Redirecting number: Redirecting numberRedirecting Number: 13612345678
 ▶ Parameter: (t=19, l=2) Redirection information: Redirection information
 End of optional parameters (0)

```

从图中看出，01 代表这是一条初始地址消息（Initial Address），这是 ISUP 中的呼叫消息，里面有主被叫号码等信息。

后面 5 个字节的含义在图中很清楚，这是 ISUP 消息中的固定部分，即这些参数是必须存在的。

接下来，02 08 是两个指针，分别指向消息中的“固定可变”部分和“任选部分”。

在这里，固定可变部分实际上就是被叫号码。所以，02 指向它后面的第二个字节，即 06。可想而知，被叫号码的长度是可变的，所以，要有一个字节指示长度，这里，06 就是一个长度，表示这个参数占后面的 6 个字节。其中，前两个字节表示了号码的性质，后面 4 个字节是真正的电话号码。06 后面的 01，扩展成二进制是 0000 0001，最高位是 0，这是一个奇偶标志，代表后面的号码长度是偶数的。由于长度电话号码只剩下 4 个字节，又是偶数，因此，号码是 8 位（每个字节表示两位号码，如果这里的奇偶标志是 1，那么号长就是 7 位）。

78 56 34 12 是真正的被叫号码，这里是以 BCD 码（8421 码）保存的，也就是说每 4 位二进制数表示一个二进制数，且同一个字节内顺序是颠倒的。所以，读出电话号码 87654321。

继续往下看下一个字节是 0a，是可变部分的开始。如果从前面说的 08 那个指针数 8 个数，也会走到这个位置。

0a 对应的 10 进制数是 10（图中 t=10），代表这个参数是一个主叫号码。08 同样是长度，后面两个字节跟前面讲被号码时的含义相同，我们只看第一个字节，81 写成二进制是 1000 0001，最高位是 1 表示号码长度为奇数。还剩下 6 个字节就是实际的电话号码，一共可以表示 12 位，但由于号长为奇数，因此最后一位不用，始终为 0。所以，接着往下数，31 86 67 45 23 01，经过顺序颠倒后读到实际号码 13687654321。

继续，28 对应图中的 t=40，表示这个参数的类型是原被叫号码。可以看出这是一个转移呼

叫，原被叫号码转移到了新的被叫号码上。用同样的方法可以读出 31 16 32 54 76 08 对应的号码为 13612345678。

同理，0b（即 t=11）为发生转移的号码。13（即 t=19）说明这是一个转移呼叫，这个参数只有两个字节。最后的 00 代表参数终止。

好玩吧？

这就是 7 号信令里的呼叫消息，即 IAM。但最初的 IAM 消息里是不带主叫号码的（也就是说被叫侧的话机不能显示来电的号码），如果 IAM 消息带了主叫号码，就叫 IAI（Initial Address Message with Additional Information）。

好了，我们不必纠结 7 号信令的细节。我们收到一个需求，FreeSWITCH 跟其它系统对接，要能解析 ISUP 消息。然后对方发过一个 pcap 包，自己看着办。

由于没法实际测试，只能把抓包中的 INVITE 消息导出来，放到一个文件中。然后，先写一个 Lua 脚本解析消息。

---

```
-- 定义个 log 函数
```

```
function log(s)
 if not s then s = "nil" end
 if session then
 session:consoleLog("ERR", s)
 else
 print(s)
 end
end
```

```
-- 读号码。传入数据 `mpart`，`p` 为数据的偏移量，即指向实际号码的位置，`n` 为号码长度，`is_even` 是奇偶标志。
```

```
function read_number(mpart, p, n, is_even)
 local s = ""

 -- 逐位读

 while n > 0 do
 local byte = string.byte(mpart, p, p+1)
 -- 取低 4 位，转成整数，拼到结果字符串中
 s = s .. bit32.band(byte, 0x0f)

 -- 如果没读到最后一位，或者号长是偶数，则取高 4 位，并右移 4 位，拼到字符串中
 if (n > 1) or is_even then
 s = s .. bit32.arshift(byte, 4)
 end

 -- 移动偏移量指针循环读下一位
 p = p + 1
 end
end
```

```
 n = n - 1
 end
 return s
end

-- 读可选参数，包括奇偶位

function read_param_number(mpart, pos)
 pos = pos + 1
 len = string.byte(mpart, pos, pos + 1)
 pos = pos + 1
 odd_even = string.byte(mpart, pos, pos + 1)
 odd_even = bit32.arshift(odd_even, 7)
 is_even = (odd_even == 0)
 pos = pos + 2
 len = len - 2
 number = read_number(mpart, pos, len, is_even)
 pos = pos + len
 return number, pos
end

-- 读不关心的参数并丢弃

function read_param(mpart, pos)
 pos = pos + 1
 len = string.byte(mpart, pos, pos + 1)
 pos = pos + 1
 pos = pos + len
 return pos
end

local mpart

if session then -- session in freeswitch
 -- 如果在 FreeSWITCH 中，Body 可以通过通道变量获取
 mpart = session:getVariable("sip_multipart")
else -- 否则我们从文件里读，只是为了测试方便
 local file = io.open("isup.sip.bin", "r")
 mpart = file:read("*a")
 file:close()
end

log(mpart)

-- 查找 ISUP 消息的位置
local x = mpart:find("application/isup")
if not x then return end -- mpart has no isup
```

```
mpart = mpart:sub(x)

x = mpart:find("\r\n\r\n")
if not x then return end
-- 截取数据
mpart = mpart:sub(x + 4) -- Now we hit the ISUP data
-- 现在, mpart 指向 ISUP 消息的第一个字节, 解析开始

log(mpart)

if mpart:len() < 2 then return end

local mpart_len = mpart:len()
local pos = 1
byte = string.byte(mpart, pos, 1) -- Message Type

local code = tonumber(byte);
log(code)

-- 首位必须为 1, 我们仅解析初始地址消息
-- 1 = Initial Address
if code ~= 1 then return end

-- 跳过不关心的参数
pos = pos + 2 -- skip Nature of Connection Indicators
-- Now Forward Call Indicators

forward1, forward2 = string.byte(mpart, pos, pos + 2)

pos = pos + 2 -- Calling Parties Cat
pos = pos + 1 -- Transmission Medium Req
pos = pos + 1 -- pointer to variable params

-- 继续跳过, 现在, `pos` 指向固定可变参数起始位置, 赋值到 `pv`

pv = string.byte(mpart, pos, pos + 1)
log(pv)
pv = pv + pos
pos = pos + 1 -- pointer to optional params

-- 让 `po` 指现任选参数起始位置

po = string.byte(mpart, pos, pos + 1)
log(po)
po = po + pos

-- 开始解析被叫号码, 获取长度

len = string.byte(mpart, pv, pv + 1)
```

```
log(len)

-- 奇偶标志

pos = pv + 1 -- odd/even
odd_even = string.byte(mpart, pos, pos + 1)
odd_even = bit32.arshift(odd_even, 7)
is_even = (odd_even == 0)

pos = pos + 1 -- numbering plan
pos = pos + 1 -- now the real number

-- pos 现在指现被叫号码，然后通过函数读取到字符串

len = len - 2
called = read_number(mpart, pos, len, is_even)
log(called)

-- 解析可选参数，考虑到参数顺序不一定是固定的，循环读取

pos = po

while pos < mpart_len do
 type = string.byte(mpart, pos, pos + 1)
 log(type)

 -- 根据参数类型调用不同的函数读取实际的号码

 if type == 0x0a then -- 10
 caller, pos = read_param_number(mpart, pos)
 log(caller)
 elseif type == 0x28 then -- 40
 original_called, pos = read_param_number(mpart, pos)
 log(original_called)
 elseif type == 0x0b then
 redirecting_number, pos = read_param_number(mpart, pos)
 log(redirecting_number)
 elseif type == 0x00 then
 break
 else
 pos = read_param(mpart, pos) -- discard
 end
end

打印结果:

if caller then
 log("caller: " .. caller)
end
```

```
if called then log("called: " .. called) end
if original_called then log("original_called: " .. original_called) end
if redirecting_number then
 log("redirecting_number: " .. redirecting_number)

 if session then
 session:setVariable("redirecting_number", redirecting_number)
 end
end
end
```

---

最终结果如下：

---

```
caller: 13687654321
called: 87654321
original_called: 18612345678
redirecting_number: 18612345678
```

---

所以，可以看出，这是一个转移呼叫，13687654321 呼叫 18612345678，发生了呼叫转移，到了 87654321，然后到达 FreeSWITCH。

程序写好了，测试很完美，但到了现场运行就出问题了一一得不到想要的结果。

由于本地没有测试环境，只能自己想办法造一个。试过用 nc 直接发，用 sipsak 发，可能因为文件中有二进制的的原因，在 FreeSWITCH 中都得不到正确的消息。因此，只好又自己写了一个程序。代码很简单，就不多解释了。功能就是从文件中读取 SIP 消息（前面我们从 WireShark 中导出来的），然后发给 FreeSWITCH 的 UDP 端口。

---

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h> /* For <netinet/in.h> */
#include <netinet/in.h> /* For struct sockaddr_in */
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h> /* struct hostent */
#include <pwd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#ifdef INADDR_NONE
```

```
#define INADDR_NONE 0xffffffff /* should be in <netinet/in.h> */
#endif

static int size = 0;

static char* parse_file(const char *file) {
 char *buf;
 struct stat s;
 int fd = open(file, O_RDONLY);
 if (fd < 0) return NULL;
 fstat(fd, &s);
 buf = mmap(0, s.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
 close(fd);
 size = s.st_size;
 if (buf == MAP_FAILED) return NULL;
 // if (munmap(buf, s.st_size) < 0) abort();
 return buf;
}

int main(int argc, char *argv[]) {
 int i, sock, bytes;
 char buf[80];
 struct sockaddr_in srv_addr; /* server's Internet socket address */
 struct hostent *hp; /* from gethostbyname() */
 int port = 5060;

 if (argc != 3) {
 printf("Usage: %s <host[:port]> <file>\n", argv[0]);
 return 1;
 }

 char *host = argv[1];
 char *p = strchr(host, ':');

 if (p) {
 *p = '\0';
 p++;
 port = atoi(p);
 }

 if (port <= 0 || port > 65535) {
 port = 5060;
 }

 char *sip = parse_file(argv[2]);
 printf("%s\n", sip);

 bzero((char *) &srv_addr, sizeof(srv_addr));
 srv_addr.sin_family = AF_INET;
```



```

 srv_addr.sin_port = htons(port);
 if ((hp = gethostbyname(argv[1])) == NULL) {
 perror("host name error");
 return 1;
 }
 bcopy(hp->h_addr, (char *) &srv_addr.sin_addr, hp->h_length);
 if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
 perror("opening udp socket");
 return 1;
 }

 printf("connecting to %s:%d\n", host, port);

 if (connect(sock, (struct sockaddr *) &srv_addr, sizeof(srv_addr)) < 0) {
 perror("connect error");
 return 1;
 }

 printf("sending %d bytes\n", size);

 if (send(sock, sip, size, 0) != size) perror("send");

 close(sock);
 if (munmap(sip, size) < 0) abort();
}

```

这样就很方便地测试了。经过调试和代码分析，发现 FreeSWITCH，甚至底层的 Sofia 库，都不能很好地处理二进制类型的 Body，而是全当成了字符串处理。所以，我们的 Lua 脚本看起来是要废掉了。

不过，好在 FreeSWITCH 还是取到了 ISUP 数据的指针。所以，我们还是可以在 C 语言里进行处理。我们很快就写了一个补丁程序：

```

diff --git a/src/mod/endpoints/mod_sofia/sofia.c b/src/mod/endpoints/mod_sofia/sofia.c
index 0d3663bc79..d364272eeb 100644
--- a/src/mod/endpoints/mod_sofia/sofia.c
+++ b/src/mod/endpoints/mod_sofia/sofia.c
@@ -41,6 +41,7 @@
 */
#include "mod_sofia.h"

extern su_log_t tport_log[];
extern su_log_t iptsec_log[];
@@ -11045,6 +11046,10 @@ void sofia_handle_sip_i_invite(switch_core_session_t *session, nua_t *nua, sofia
 if (mp->mp_payload && mp->mp_payload->pl_data && mp->mp_content_type && mp->mp_content_type-
 ↪ >c_type) {

```

```
char *val = switch_core_session_sprintf(session, "%s:%s", mp->mp_content_type->c_type, mp-
↳ >mp_payload->pl_data);
switch_channel_add_variable_var_check(channel, "sip_multipart", val, SWITCH_FALSE,
↳ SWITCH_STACK_PUSH);
+
+ if (strstr(mp->mp_content_type->c_type, "application/isup")) {
+ parse_isup((uint8_t *)mp->mp_payload->pl_data, channel);
+ }
+ }
+ }
```

其中，`mp_payload->pl_data` 是 `multipart` 部分的数据指针，每个部分都可以取到，因此，我们在判断类型是 `application/isup` 的情况下，就自己写个函数解析一下，然后赋值到相应的通道变量上。具体解析函数就是照着上面的 Lua 再写个 C 的版本就好了，不再赘述。

值得说明的是，虽然取到了指针，但不知道数据的长度。好在我们要解析的数据最后有个 `00` 可以视为结束。但实际实现时要注意如果有人伪造非法的数据不要造成死循环，简单用个计数器限制一下解析的最大长度即可。如果要完美解析所有 SIP/I 或 SIP/T 消息，需要对 `libsofia` 打补丁，工作量就大了。

参考资料：

- <https://tools.ietf.org/html/rfc3372>
- <https://wiki.wireshark.org/Protocols/isup?action=show&redirect=ISUP>
- <https://www.differencebetween.com/difference-between-sip-i-and-sip-t/>

## 版本更新历史

V6 - V7: 20201220

- 新增：6.3 测试最新版的 FFmpeg
- 新增：6.4 如何维护自己的 FreeSWITCH 分支
- 新增：6.5 解析 SIP 中携带的 ISUP 消息

页数：手机版/1137 标准版/560 印刷版：不含第五章模块代码选析

V5 - V6: 20191112

- 新增：3.5.2 JSON
- 新增：3.5.3 JSON API
- 新增：4.23 switch\_core\_memory
- 新增：4.24 switch\_core\_timer
- 新增：4.25 switch\_port\_allocator
- 新增：4.28 switch\_loadable\_module
- 新增：4.29 switch\_utils
- 新增：4.30 switch\_vad
- 新增：第五章模块代码选析
- 新增：印刷版，不含第五章

页数：手机版/1115 标准版/553

V4 - V5: 20190723

- 完善：2.1.14 Core Video
- 新增：4.22 switch\_scheduler
- 新增：版本历史
- 优化：优化了移动版排版

页数：手机版/856 标准版/426

V3 - V4: 20190202

- 新增：2.3 Endpoint 接口
- 新增：3.5 主要数据结构和函数使用方法
- 新增：3.7 测试框架
- 更新：4.11 ~ 4.16
- 新增：4.17 switch\_core\_file
- 新增：4.18 switch\_core\_hash
- 新增：4.19 switch\_core\_io
- 新增：4.20 switch\_core\_media\_bug
- 新增：4.21 switch\_core\_video

页数：399

## 写在最后

本书将持续更新，这就是电子版的好处...

如果你对书中的内容和章节安排等有什么意见或建议，欢迎与我联系。如果你建议的内容适合放在本书里，我会考虑写进去；如果不适合放到本书中，我也会考虑写其它主题的书。

如果你的公司想在本书中植入广告或者赤裸裸地做广告，也欢迎与我们联系。

电子邮件：[info@x-y-t.cn](mailto:info@x-y-t.cn)。

## 作者简介

**杜金房**（网名：Seven Du）资深网络通信技术专家，在网络通信领域耕耘近 20 年，精通 VoIP、SIP 和 FreeSWITCH 等各种网络协议和技术，经验十分丰富。有超过 7 年的 FreeSWITCH 应用和开发经验，不仅为国内大型通信服务厂商提供技术支持和解决方案，而且客户还遍及美、欧、东南亚等海外国家。

FreeSWITCH-CN 中文社区创始人兼执行主席，被誉为国内 FreeSWITCH 领域的『第一人』；在 FreeSWITCH 开源社区非常活跃，不仅经常为开源社区提交补丁和新功能、新特性，而且还开发了很多外围模块和外围软件；此外，他经常在 FreeSWITCH 的 Wiki 上分享自己的使用心得和经验、在 FreeSWITCH IRC、QQ 及微信群中热心回答网友提问，并不定期在国内不同城市举行 FreeSWITCH 技术培训；自 2011 年起每年都应邀参加在美国芝加哥举办的 ClueCon 大会，并发表主题演讲。

此外，他还精通 C、Erlang、Ruby、Lua 等语言相关的技术。

著有[《FreeSWITCH 权威指南》](#)，2014 年出版。

创办了[北京信悦通科技有限公司](#)和[烟台小樱桃网络科技有限公司](#)，提供 FreeSWITCH 培训和商业技术支持服务。

## 版权声明

本书版权归作者所有，任何人未经书面授权均不得分发此书。

本书电子版仅在 FreeSWITCH VIP 知识星球和小樱桃科技微信商城上发布。如果您不小心从其它渠道获得本书，请删除您的版本并到以上渠道获取/购买正版。

## 第七章 广告

### 7.1 关于广告的广告

请允许我在本书中发布广告。广告合作联系邮箱：info@x-y-t.cn

### 7.2 XSWITCH 云——我自己的通信助手

精简版、专业版、旗舰版，三版私人订制，只为给你最优秀的体验

<http://xswitch.cn/>

### 7.3 RTS 中文社区

<http://rts.cn>

### 7.4 烟台小樱桃网络科技有限公司提供商业 FreeSWITCH、Kamailio 及 OpenSIPS 技术支持

网址：<http://x-y-t.cn> 邮箱：info@x-y-t.cn

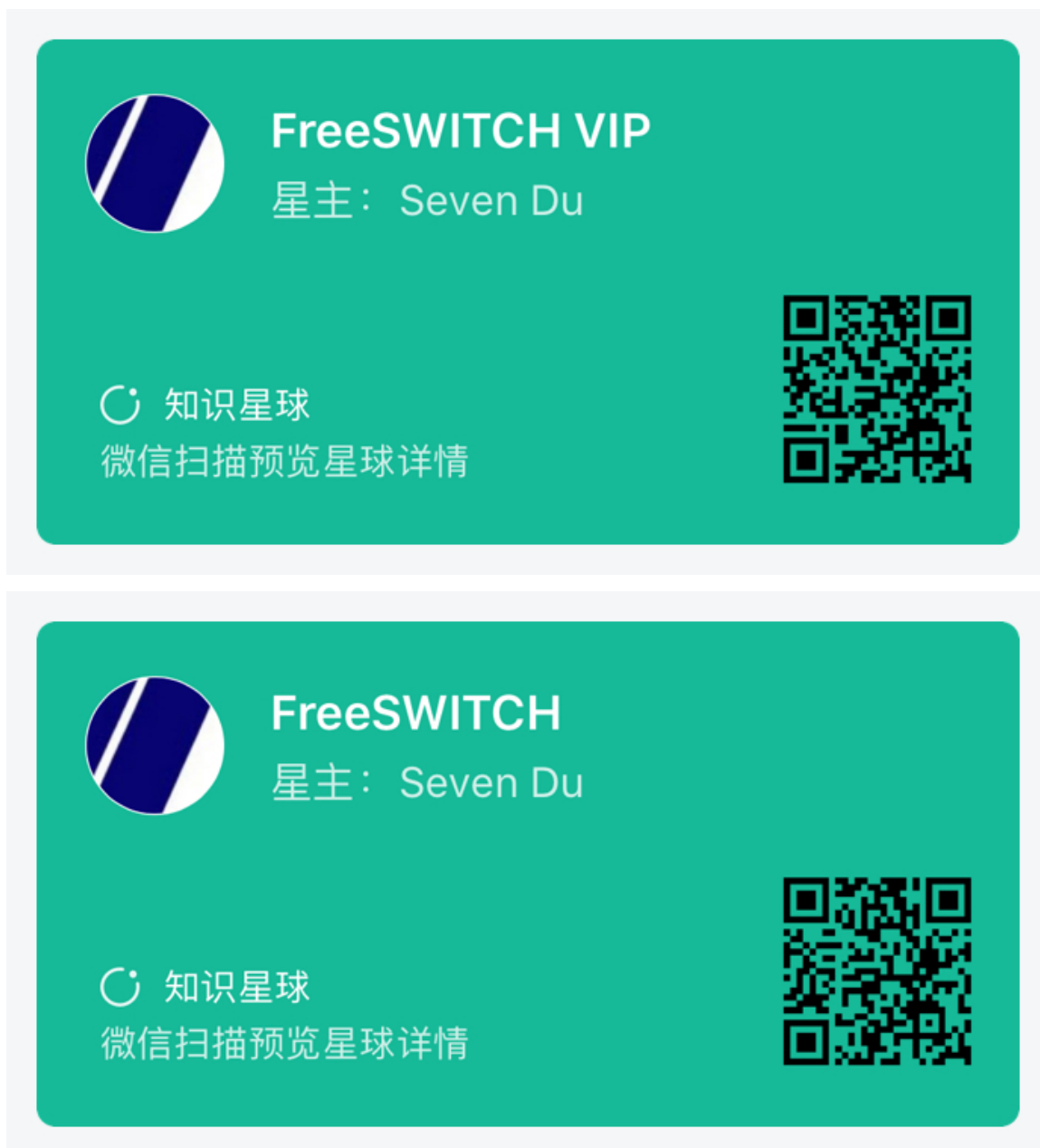
### 7.5 知识星球

为了给大家提供新的交流学习的平台，杜老师新开通了两个『知识星球』，一个免费版，一个收费版。可以使用如下链接或通过微信扫描二维码加入。

· FreeSWITCH：<https://t.zsxq.com/RBi6Ee2>



- FreeSWITCH VIP: <https://t.zsxq.com/2zb6qBE>



## 7.6 FreeSWITCH 相关图书推荐

- 《FreeSWITCH 文集》收集了一些 FreeSWITCH 文章，相比其它 FreeSWITCH 书来说，技术内容比较少，便于非技术人员快速了解 FreeSWITCH。
- 《FreeSWITCH 互联互通》主要收集了一些互联互通的例子，书中有些例子来自《FreeSWITCH 权威指南》。

- 《FreeSWITCH 实例解析》收集了一些如何使用 FreeSWITCH 的实际例子，方便读者参考。书中有些内容来自《FreeSWITCH 权威指南》。
- 《FreeSWITCH 实战》是《FreeSWITCH 权威指南》的前身，不再更新，但该书有其历史意义。
- 《FreeSWITCH WIRESHARK》是一本介绍如何使用 Wireshark 分析 SIP/RTP 数据包的书。
- 《FreeSWITCH 源代码分析》主要讲解源代码。
- 《FreeSWITCH 权威指南》是正式出版的纸质书和电子书。

以上所有图书均可以在<http://book.dujinfang.com>查看最新信息及购买。

我们除了有纸质版的书、电子书和线下培训，我们还有线上培训课程（点击课程报名，课程永久有效）。

<http://x-y-t.cn/#training>

微信扫一扫下方二维码，视频中有杜老师录制的 FreeSWITCH 系列课程。



微信号: FreeSWITCH-CN

THIS PAGE INTENTIONALLY LEFT BLANK.