

# FreeSWITCH 源代碼分析

杜金房 著

INVITE sip:+861234567890@example.com SIP/2.0  
Via: SIP/2.0/UDP 10.20.30.40:5060;rport;branch=z9hG4bKj9Nm7v3047Ha  
Max-Forwards: 70                   FreeSWITCH-CN 出品  
From: "Seven Du" <sip:+861234567890@example.com>;tag=B0U9ZQZQ124SK  
To: <sip:+861234567890@example.com>  
Call-ID: eb9f724e-9fed-1233-88ac-525400272cd0  
CSeq: 77777777 INVITE

# FreeSWITCH 源代码分析

杜金房

版权所有，侵权必究

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

开    本 1890 毫米 × 2360 毫米

印    张 7.5

印    数 7

版    数 2016 年 7 月第 1 版     2016 年 7 月第 1 次发布

电子邮箱 [freeswitch@dujinfang.com](mailto: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的事情，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》，仅有小小的改动。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 1.10 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。

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

|                                   |           |
|-----------------------------------|-----------|
| <b>前 言</b>                        | <b>1</b>  |
| <b>1 源代码导读及编译指南</b>               | <b>7</b>  |
| 1.1 准备 FreeSWITCH 源代码环境 . . . . . | 7         |
| 1.2 FreeSWITCH 源代码目录结构 . . . . .  | 8         |
| 1.3 FreeSWITCH 源代码的编译 . . . . .   | 8         |
| 1.3.1 首次编译 . . . . .              | 9         |
| 1.3.2 增量编译 . . . . .              | 11        |
| 1.3.3 常见问题及最佳实践 . . . . .         | 12        |
| 1.4 小结 . . . . .                  | 12        |
| <b>2 FreeSWITCH 源代码导读</b>         | <b>13</b> |
| 2.1 核心架构 . . . . .                | 13        |
| 2.1.1 APR . . . . .               | 13        |
| 2.1.2 SWITCH APR . . . . .        | 14        |
| 2.1.3 main 函数 . . . . .           | 16        |
| 2.1.4 可加载模块 . . . . .             | 18        |
| 2.1.5 模块的结构 . . . . .             | 26        |
| 2.1.6 Session 和 Channel . . . . . | 28        |
| 2.1.7 SWITCH IVR . . . . .        | 33        |
| 2.1.8 Core IO . . . . .           | 34        |
| 2.1.9 Core Media . . . . .        | 38        |

|   |           |
|---|-----------|
| 2.1.10 Core RTP . . . . .                     | 39        |
| 2.1.11 SWITCH XML . . . . .                   | 42        |
| 2.1.12 SWITCH Event . . . . .                 | 43        |
| 2.1.13 Core Codec 和 Core File . . . . .       | 47        |
| 2.1.14 Core Video . . . . .                   | 49        |
| 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 . . . . .                            | 77        |
| <b>3 FreeSWITCH 二次开发</b> . . . . .            | <b>79</b> |
| 3.1 给 FreeSWITCH 汇报 Bug 和打补丁 . . . . .        | 79        |
| 3.1.1 汇报 Bug 时注意的问题 . . . . .                 | 79        |
| 3.1.2 修复内存泄露问题 . . . . .                      | 80        |
| 3.1.3 给中文模块打补丁 . . . . .                      | 81        |
| 3.1.4 给 FreeSWITCH 核心打补丁 . . . . .            | 82        |
| 3.1.5 高手也会犯错误 . . . . .                       | 84        |
| 3.1.6 汇报严重的问题 . . . . .                       | 85        |
| 3.1.7 给 Sofia-SIP 打补丁 . . . . .               | 86        |
| 3.1.8 给现有 App 增加新功能 . . . . .                 | 88        |
| 3.1.9 给 FreeSWITCH 增加一个新的 Interface . . . . . | 91        |
| 3.2 写一个新的 FreeSWITCH 编解码模块 . . . . .          | 93        |
| 3.3 从头开始写一个模块 . . . . .                       | 95        |
| 3.3.1 初始准备工作 . . . . .                        | 95        |
| 3.3.2 写一个简单的 Dialplan . . . . .               | 96        |
| 3.3.3 增加一个 App . . . . .                      | 99        |
| 3.3.4 写一个 API . . . . .                       | 100       |
| 3.3.5 小结 . . . . .                            | 101       |

|   |     |
|---|-----|
| 3.4 使用 libfreeswitch . . . . .          | 101 |
| 3.4.1 自己写一个软交换机 . . . . .               | 102 |
| 3.4.2 使用 libfreeswitch 提供的库函数 . . . . . | 103 |
| 3.4.3 其它 . . . . .                      | 108 |
| 3.5 调试跟踪 . . . . .                      | 109 |
| 3.6 小结 . . . . .                        | 110 |
| <br>                                    |     |
| 4 核心代码详解 . . . . .                      | 112 |
| 4.1 g711.c . . . . .                    | 112 |
| 4.2 inet_pton.c . . . . .               | 116 |
| 4.3 switch.c . . . . .                  | 116 |
| 4.4 switch_apr.c . . . . .              | 127 |
| 4.5 switch_buffer.c . . . . .           | 127 |
| 4.6 switch_caller.c . . . . .           | 128 |
| 4.7 switch_channel.c . . . . .          | 132 |
| 4.8 switch_config.c . . . . .           | 177 |
| 4.9 switch_console.c . . . . .          | 178 |
| 4.10 switch_core.c . . . . .            | 190 |
| 4.11 switch_core_asr.c . . . . .        | 209 |
| 4.12 switch_core_cert.c . . . . .       | 214 |
| 4.13 switch_core_codec.c . . . . .      | 216 |
| 4.14 switch_core_db.c . . . . .         | 223 |
| 4.15 switch_core_directory.c . . . . .  | 227 |
| 4.16 switch_core_event_hook.c . . . . . | 227 |
| 4.17 switch_core_file.c . . . . .       | 228 |
| <br>                                    |     |
| 5 代码修炼之道 . . . . .                      | 235 |
| 5.1 虚拟演播室 . . . . .                     | 235 |
| 5.1.1 Chroma Key . . . . .              | 235 |
| 5.1.2 mod_video_filter . . . . .        | 238 |

|   |         |
|---|---------|
| 5.2 编译相关  | 247     |
| 5.3 精彩继续  | 248     |
| 5.4 永无止境  | 249     |
| 5.5 小结  | 249     |
| <br>写在最后  | <br>251 |
| <br>作者简介  | <br>252 |
| <br>版权声明  | <br>253 |
| <br>广告  | <br>254 |
| 关于广告的广告   | 254     |
| FreeSWITCH 第五届开发者沙龙于 2016 年 8 月 27 日在京举行胜利闭幕            | 254     |
| FreeSWITCH 培训 2016 夏季班（北京站）于 2016 年 8 月 28-30 日在京举行胜利闭幕 | 254     |
| 烟台小樱桃网络科技有限公司提供商业 FreeSWITCH 及 OpenSIPS 技术支持            | 254     |
| 烟台小樱桃网络科技有限公司是潮流网络（GrandStream）山东总代理                    | 255     |
| FreeSWITCH 相关图书   | 255     |
|   | 258     |

# 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，由浅入深地一步一步进入 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们需要注意的问题。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<sup>1</sup>。APR 出身于 Apache<sup>2</sup>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。

---

<sup>1</sup>参见<http://apr.apache.org/>。

<sup>2</sup>参见。

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 {
    ...
}
```

---

上面的程序片断是错误的，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<sup>3</sup>，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<sup>4</sup>。

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。

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。

---

<sup>3</sup>Since 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-archive.apache.org/mod\\_mbox/apr-dev/200502.mbox/%3C1f1d9820502241330123f955f@mail.gmail.com%3E](http://mail-archive.apache.org/mod_mbox/apr-dev/200502.mbox/%3C1f1d9820502241330123f955f@mail.gmail.com%3E)

<sup>4</sup>REMARK: 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>

---

```

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后命令历史（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）是区分是大小写的，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<sup>5</sup>。

---

```
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，这两个模块是直接在核心代码中实现的，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。

---

```
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) ==
```

---

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）。

---

<sup>5</sup>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。

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(
1520         path, file, global, &new_module)) == SWITCH_STATUS_SUCCESS) {
1521         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，产生相关的事情，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(
1523            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，描述了一个最精简的模块的结构。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，我们对空行做了行号）。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（实际上一个个双重指针）——module\_interface，我们需要被初始化这个指针（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 its own thread when the module-load completes
64     If it returns anything but SWITCH_STATUS_TERM it will be called again automatically
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，如果说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;
1558     ...
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:
485             ...
486             case CS_DESTROY:
487                 goto done;
488             case CS_REPORTING: /* Call Detail */
489             ...
490             case CS_HANGUP: /* Deactivate and end the thread */
491             ...
492             case CS_INIT: /* Basic setup tasks */
493             {
494             ...
495                 STATE_MACRO(init, "INIT");
496             ...
497             case CS_ROUTING:/* Look for a dialplan and find something to do */
498                 STATE_MACRO(routing, "ROUTING");
499                 break;
500             case CS_RESET: /* Reset */
501                 STATE_MACRO(reset, "RESET");
502                 break;
503             }
504         }

```

---

STATE\_MACRO宏是在第 356 行定义的，该宏比较复杂，其实展开以后，大致就想相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 }
7 } 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<sup>6</sup>。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）。

<sup>6</sup>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>。

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调。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。

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，我们就看一看它是怎么实现的。

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         }
649
650         switch_ivr_parse_all_events(session);
651         ...
652
653         if (switch_channel_has_dtmf(channel)) {
654             ...

```

---

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 毫秒由到一帧音频数据，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，该函数就回返回一个静音包（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，得看底层的驱却是怎么实现的。

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,
567                                         &session->raw_read_frame.datalen,
568                                         &session->raw_read_frame.rate,
569                                         &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,
793                                     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,
841         &session->enc_read_frame.datalen,
842         &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，它的主要作用就是，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(
1026     switch_core_session_t *session, switch_frame_t *frame,
1027     switch_io_flag_t flags,
1028     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, ...
1174     ...
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进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(
1014             session, frame, flags, stream_id)) == SWITCH_STATUS_SUCCESS) {
1015             for (ptr = session->event_hooks.write_frame; ptr; ptr = ptr->next) {
1016                 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(
1396         switch_core_session_t *session, switch_frame_t **frame,
1397         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_
...
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(
5503     switch_rtp_t *rtp_session, switch_frame_t *frame,
5504     switch_io_flag_t io_flags)
5505 {
5506 ...
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__;
5516     frame->timestamp = ntohl(rtp_session->recv_msg.header.ts);
5517     frame->seq = (uint16_t) ntohs((uint16_t) rtp_session->recv_msg.header.seq);
5518     frame->ssrc = ntohl(rtp_session->recv_msg.header.ssrc);
5519     frame->m = rtp_session->recv_msg.header.m ? SWITCH_TRUE : SWITCH_FALSE;
5520 ...
5521     frame->datalen = bytes;
5522     return SWITCH_STATUS_SUCCESS;
5523 }
5524 }
```

---

`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,
4675     switch_payload_t *payload_type,
4676     payload_map_t **pmapP, switch_frame_flag_t *flags,
4677     switch_io_flag_t io_flags)
4678 ...
4679     status = read_rtp_packet(rtp_session, &bytes, flags, SWITCH_FALSE);
4680 ...
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,
4138     switch_size_t *bytes, switch_frame_flag_t *flags,
4139     switch_bool_t return_jb_packet)
4140 {
4141 ...
4142     status = switch_socket_recvfrom(rtp_session->from_addr,
4143         rtp_session->sock_input, 0,
4144         (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<sup>7</sup>。

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 {
6112 ...
6113     if (switch_socket_sendto(rtp_session->sock_output,
6114         rtp_session->remote_addr, 0, frame->packet, &bytes)
6115 ...
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],
3148     char *timer_name, const char **err, switch_memory_pool_t *pool)

```

---

<sup>7</sup>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。

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) == S
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偶合关系，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,
1758         ...
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_
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(
264     switch_thread_data_t *thread, void *obj)
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,
718     uint32_t *encoded_rate, unsigned int *flag)
718 {
...
736     status = codec->implementation->encode(codec, other_codec,
737         decoded_data, decoded_data_len,
738         decoded_rate, encoded_data, encoded_data_len, encoded_rate, flag);
737
...
}
```

---

```

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,
750     uint32_t *decoded_rate, unsigned int *flag)
751 {
752 ...
753
754     status = codec->implementation->decode(codec, other_codec,
755         encoded_data, encoded_data_len, encoded_rate,
756         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(
254     switch_file_handle_t *fh, void *data, switch_size_t *len)
255 {
256 ...
257     if ((status = fh->file_interface->file_read(fh, fh->pre_buffer_data, &rlen)) == SWITCH_STATUS_BREAK) {
258 ...
259
260     if ((status = fh->file_interface->file_read(fh, data, len)) == SWITCH_STATUS_BREAK) {
261 ...
262
263 ...
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287 SWITCH_DECLARE(switch_status_t) switch_core_file_write(
288     switch_file_handle_t *fh, void *data, switch_size_t *len)
289 {
290 ...
291     if ((status = fh->file_interface->file_write(fh,
292         fh->pre_buffer_data, &rlen)) != SWITCH_STATUS_SUCCESS) {
293 ...
294     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

TODO…

## 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，我们再结合实际的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，我们在上一章已通找到一个突破口——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 (;;) {
371         app_int = (switch_application_interface_t *) \
372             switch_loadable_module_create_interface(*module_interface, \
373             SWITCH_APPLICATION_INTERFACE); \
374         app_int->interface_name = int_name; \
375         app_int->application_function = funcptr; \
376         app_int->short_desc = short_descript; \
377         app_int->long_desc = long_descript; \
378         app_int->syntax = syntax_string; \
379         app_int->flags = app_flags; \
380         break; \
381     }

```

---

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`，然后给它赋于合适的值。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_CONFERENCE);
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**stristr**, 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\_CONFERENCE**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上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上调用**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 };
3696     ...
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`宏安装到核心中去的。

### 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），则在第 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(
1401     switch_channel_t *channel,
1402     const char *varname, const char *value,
1403     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,
...

```

---

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_originate`函数发起一个新的呼叫。

---

```

3194     if ((status =
3195         switch_ivr_originate(session, &peer_session, &cause,
3196         data, 0, NULL, NULL, NULL, NULL, NULL, SOF_NONE, NULL))
3197             != SWITCH_STATUS_SUCCESS) {
3198     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<sup>8</sup>。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 {
3216     ...
3236         switch_ivr_multi_threaded_bridge(session, peer_session,

```

---

<sup>8</sup>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。

---

```
    func, a_key, b_key);
3237 }
```

---

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(
1271     switch_core_session_t *session,
1272     switch_core_session_t *peer_session,
1273     switch_input_callback_function_t input_callback, void *session_data,
1274     void *peer_session_data)
1275 {
1276     switch_ivr_bridge_data_t *a_leg =
1277         switch_core_session_alloc(session, sizeof(*a_leg));
1278     switch_ivr_bridge_data_t *b_leg =
1279         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 =
698         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<sup>9</sup>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);

```

---

<sup>9</sup>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_ST

```

---

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_rwunlock(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" />

```

---

这里面的`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, '@')) != NULL) {
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) {
3944
3945         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 上进行处理的。我们将在第2.2.1节再讲`mod_soifia`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;
...
6393     *module_interface = switch_loadable_module_create_module_interface(pool, modname);
6394
...
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进行声明。该声明也是一个在switch\_types.h:2016行定义的一个宏，该宏定义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_rwunlock(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 通话都是从它开始和终止的，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。为了从根本上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     ...
1798     switch(event) {
1799         case nua_i_terminated:
1800             ...
1801         case nua_i_invite:
1802         case nua_i_register:
1803         case nua_i_options:
1804         case nua_i_notify:
1805         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队列以提高并发量。

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 {
1751     ...
1752
1753     if (event == nua_i_invite && !sofia_private) {
1754         ...
1755         session = switch_core_session_request(sofia_endpoint_interface, SWITCH_CALL_DIRECTION_INBOUND, SOF_NONE, NU

```

---

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，并推到队列中等候处理。

**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(
1672     switch_thread_t *thread, void *obj)
1673 {
1674     ...
1675     for(;;) {
1676         if (switch_queue_pop(q, &pop) != SWITCH_STATUS_SUCCESS) {
1677             ...
1678             sofia_dispatch_event_t *de = (sofia_dispatch_event_t *) pop;
1679             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 {
1645     ...
1646     our_sofia_event_callback(de->data->e_event, de->data->e_status,
1647         de->data->e_phrase, de->nua, de->profile,
1648         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[])
1030 {
1031     ...
1032     switch (event) {
1033         case nua_r_get_params:
1034         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，如果需要对来话进行认证，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,
1267             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到到正确的入口。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(
4058         sofia_endpoint_interface, SWITCH_CALL_DIRECTION_OUTBOUND,
4059         flags, pool, switch_event_get_header(var_event,
4060             "origination_uuid")))) {
4061
4062     tech_pvt = sofia_glue_new_pvt(nsession);
4063
4064     data = switch_core_session_strdup(nsession,
4065         outbound_profile->destination_number);
4066
4067     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。它在第 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。退一万步讲，这些代码是用了将近十年的时间写成的，不要止望一天就精通。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。

# 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 Bug<sup>1</sup>，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，并附加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?? Win7? 2012?)

---

<sup>1</sup>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，只是你看不见而已。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。

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: xxxx
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（有时候问题描述并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 f255f65a82e2cfe1aed1f44aa2459e5faa150e
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，因此我们把它们全部贴在后面。

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,
275     switch_size_t samples, uint32_t channels)
276 {
277     int16_t *buf;
278     switch_size_t len = samples * sizeof(int16_t);
279     switch_size_t i = 0;
280     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`<sup>2</sup> 将 32 位的整

<sup>2</sup>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。

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）。

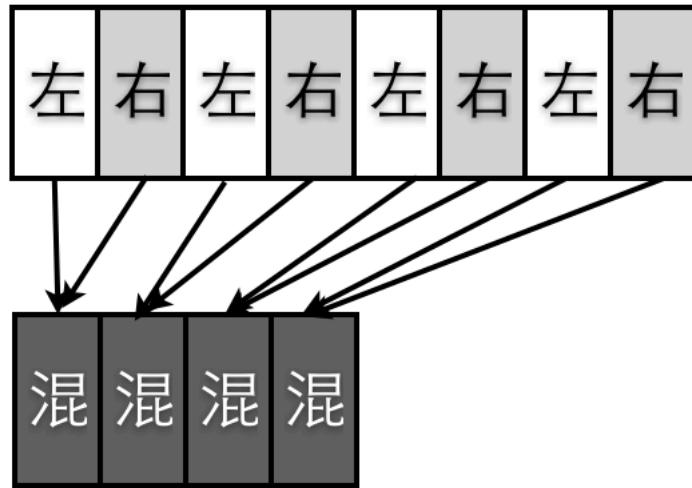


图 3.1: 图 22-1 双声道混音

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,
275     switch_size_t samples, uint32_t channels)
276 {
277     switch_size_t i = 0;
278     uint32_t j = 0;
279
280     for (i = 0; i < samples; i++) {
281         int32_t z = 0;
282         for (j = 0; j < channels; j++) {
283             z += data[i * channels + j];
284             switch_normalize_to_16bit(z);
285             data[i] = (int16_t) z;
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 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地多”。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

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_rwunlock(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 MSRP<sup>3</sup>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。下面我们来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<sup>4</sup>。

---

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

---

<sup>3</sup>MSRP (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>。

<sup>4</sup>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。

```
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_msrp: 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_msrp = 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，它想与现有的 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 *
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,
90             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 }
```

---

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，我们就来再写一个。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，重温这段美好的回快。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
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 *fmtcp = "";
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[i];

```

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, ftmp, 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
117     if (status != SWITCH_STATUS_SUCCESS &&
118         status != SWITCH_STATUS_BREAK) {
119         goto end;
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，即然，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 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 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。

## 3.6 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。

PCM<sup>1</sup>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  $8000\text{Hz}$ ，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 (Linear)，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 a 律 (*alaw*) 和  $\mu$  律 (*ulaw*，希腊字母  $\mu$  简单起见写成 *u*)。它们都是在 ITU G.711<sup>2</sup> 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：

---

<sup>1</sup> 参见 <https://zh.wikipedia.org/wiki/脈衝編碼調變>。

<sup>2</sup> 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<sup>3</sup>。

---

```
45 #ifdef __cplusplus
46 extern "C" {
47 #endif
```

<sup>3</sup> 参见 <https://zh.wikipedia.org/wiki/>。

---

```

...
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 & 0xFOFOFOFO) {
114             bits &= 0xFOFOFOFO;
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

该文件只导取了一个函数 `switch_inet_ntop`，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 Presentation format (which usually means ASCII printable) to Network 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 方式在后台运行时会用到, 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 {
...
198     hStatus = RegisterServiceCtrlHandler(service_name, &ServiceCtrlHandler);
...

```

---

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 类系统上启动后台进程的标准方法，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 会启动到后台模式，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，操作系统上就会有两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 session<sup>4</sup>。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();

```

---

<sup>4</sup>The 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.

该函数的实现再 `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 完全启动完毕后才能启动。

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);
303
304         if (system_ready < 0) {
305             printf("FreeSWITCH[%d] Error starting system! pid:%d\n", (int) getpid(), (int) pid);
306             kill(pid, 9);
307             exit(EXIT_FAILURE);
308         }
309
310         printf("FreeSWITCH[%d] System Ready pid:%d\n", (int) getpid(), (int) pid);
311     }
312
313     exit(EXIT_SUCCESS);
314 }

```

---

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，会给予进程发送`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，并且在 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())==0) { /* 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_CL...
```

---

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());
```

---

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) {
...
1123 #else
...
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) {
...
1218
1219     if (destroy_status == SWITCH_STATUS_RESTART) {
1220         if (!argv || execv(argv[0], argv) == -1) {
1221             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。理论上来讲，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 *tocopy)
```

---

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 *tocopy)
179 {
180     switch_memory_pool_t *pool;
182     pool = switch_core_session_get_pool(session);
184     return switch_caller_profile_dup(pool, tocopy);
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     }
192     ...
193     if (!strcasecmp(name, "caller_id_number")) {
194         return caller_profile->caller_id_number;
195     }
196     ...
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");
320 ...
321
322     switch_snprintf(header_name, sizeof(header_name), "%s-Privacy-Hide-Number", prefix);
323     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, header_name,
324                                   switch_test_flag(caller_profile, SWITCH_CPF_HIDE_NUMBER) ? "true" : "false");
325 }
```

---

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,
                                                               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,
498                                         const char *extension_name,
500                                         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",
290                                         switch_channel_callstate2str(o_callstate));
291         switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Channel-Call-State-Number", "%d", callstate);
292         switch_channel_event_set_data(channel, event);
293         switch_event_fire(&event);
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,
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,
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",
424             switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_OUTBOUND ? "outbound" : "inbound");
425     }
426 }

```

---

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);
452     (*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);
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);
556     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_trypop(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
...
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

```

---

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，Presence。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,
759                                         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");
777     ...
785     if (switch_event_create(&event, type) == SWITCH_STATUS_SUCCESS) {
786         ...
787     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 {
843     ...
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_rwunlock(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     }
890
891     return var;
892 }
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);
973             ...
974             if (!cp || !(v = switch_caller_get_field_by_name(cp, varname))) {
975                 if ((vdup = switch_core_get_variable_pdup(varname, switch_core_session_get_pool(channel->session)))) {
976                     v = vdup;
977                 }
978             }
979         }
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)
1089 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 {
1217     const char *export_vars = switch_channel_get_variable(channel, export_varname);
1218     char *cptmp = switch_core_session_strdup(channel->session, export_vars);
1219     int argc;
1220     char *argv[256];
1221     if (zstr(export_vars)) return;
1222     if (var_event) {
1223         switch_event_del_header(var_event, export_varname);
1224         switch_event_add_header_string(var_event, SWITCH_STACK_BOTTOM, export_varname, export_vars);
1225     }
1226
1227     if (peer_channel) {
1228         switch_channel_set_variable(peer_channel, export_varname, export_vars);
1229     }
1230
1231     if ((argc = switch_separate_string(cptmp, ',', argv, (sizeof(argv) / sizeof(argv[0]))))) {
1232         int x;
1233
1234         for (x = 0; x < argc; x++) {
1235             const char *vval;
1236             if ((vval = switch_channel_get_variable(channel, argv[x]))) {
1237                 char *vvar = argv[x];
1238                 if (!strcasecmp(vvar, "nolocal:", 8)) { /* remove this later ? */
1239                     vvar += 8;
1240                 } else if (!strcasecmp(vvar, "_nolocal_", 9)) {
1241                     vvar += 9;
1242                 }
1243             }
1244         }
1245     }
1246 }
```

```

1246
...
1256     if (peer_channel) {
...
1262         switch_channel_set_variable(peer_channel, vvar, vval);

```

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 va

```

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;
1429             if (var_check) {
1430                 ok = !switch_string_var_check_const(value);
1431             }
1432             if (ok) {
1433                 switch_event_add_header_string(channel->variables, SWITCH_STACK_BOTTOM, varname, value);
1434             } else {
1435                 switch_log_printf(SWITCH_CHANNEL_CHANNEL_LOG(channel), SWITCH_LOG_CRIT, "Invalid data ${%s} contains %s\n", varname, value);
1436             }
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,
1448                         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,
1481                         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 *name);

```

---

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,
1542                         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_
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,
1645           switch_channel_state_t want_state, uint32_t timeout)
1646 {
```

---

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,
1666           switch_channel_flag_t want_flag,
1667           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 };
1751     SWITCH_STANDARD_STREAM(stream);
1752
1753     switch_mutex_lock(channel->flag_mutex);
1754     for (i = 0; i < CF_FLAG_MAX; i++) {
1755         if (channel->flags[i]) {
1756             stream.write_function(&stream, "%d=%d;", i, channel->flags[i]);
1757         }
1758     }
1759     switch_mutex_unlock(channel->flag_mutex);
1760     r = (char *) stream.data;
1761     if (end_of(r) == ';') end_of(r) = '\0';
1762     return r;
1763 }
1764 }
```

---

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 {
1802     ...
1803     if (channel->flags[flag] != value) {
1804         just_set = 1;
1805         channel->flags[flag] = value;
1806     }
1807     ...
1808     if (flag == CF_VIDEO_READY && just_set) {
1809         switch_core_session_request_video_refresh(channel->session);
1810     }
1811     ...
1812     if (flag == CF_VIDEO_ECHO || flag == CF_VIDEO_BLANK || flag == CF_VIDEO_DECODED_READ || flag == CF_VIDEO_PASSIVE)
1813         switch_core_session_start_video_thread(channel->session);
1814 }
```

---

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`为真则继续检查信令。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_RINGING)
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,
2191                                 switch_channel_state_t val) {
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         }
2212 }
```

---

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,
2217                                         switch_channel_state_t state,
2218                                         const char *file, const char *func, int line)
2219 {
2220 ...
2221     if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_STATE) == SWITCH_STATUS_SUCCESS) {
2222         switch_channel_event_set_data(channel, event);
2223         switch_event_fire(&event);
2224     }

```

---

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。之后返回当前 Channel 的状态 (L2488)。

---

```

2269 SWITCH_DECLARE(switch_channel_state_t) switch_channel_perform_set_state(switch_channel_t *channel,
2270                                         const char *file, const char *func, int line, switch_channel_state_t state)
2271 {
2272 ...
2273     last_state = channel->state;
2274
2275     if (last_state == state) goto done;
2276
2277     switch (last_state) {
2278         case CS_NEW:
2279         case CS_RESET:
2280             switch (state) {
2281                 default:
2282                     ok++; break;
2283             }
2284             break;
2285         case CS_INIT:
2286             switch (state) {
2287                 case CS_EXCHANGE_MEDIA:
2288                 case CS_SOFT_EXECUTE:
2289                 case CS_ROUTING:
2290                 case CS_EXECUTE:
2291                 case CS_PARK:
2292                 case CS_CONSUME_MEDIA:
2293                 case CS_HIBERNATE:
2294                 case CS_RESET:
2295                     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;
2458         default:
2459             break;
2460         }
2461     }
2462 }
2463 if (ok) {
...
2466     careful_set(channel, &channel->state, state);
...
2473     return channel->state;
2474 }
```

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,
2522                                     "Channel-State", switch_channel_state_name(channel->running_state));
2523     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
2524                                     "Channel-Call-State", switch_channel_callstate2str(channel->callstate));
2525     switch_event_add_header_string(event, SWITCH_STACK_BOTTOM,
2526                                     "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,
2570                                         "Hangup-Cause", switch_channel_cause2str(channel->hangup_cause));
2571     }

```

与上面类似，设置扩展数据，大部分是一些通道变量。大家在编码开发中可能遇到，有一些 Channel 相关的事件中是所有的通道变量的，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 ||
2620         event->event_id == SWITCH_EVENT_CHANNEL_ANSWER ||
2621         event->event_id == SWITCH_EVENT_CHANNEL_BRIDGE ||
2622         event->event_id == SWITCH_EVENT_CHANNEL_HANGUP ||
2623         event->event_id == SWITCH_EVENT_CHANNEL_HANGUP_COMPLETE ||
...
2645         event->event_id == SWITCH_EVENT_CUSTOM) {
2646         if (channel->scope_variables) {
2647             switch_event_t *ep;
2648             for (ep = channel->scope_variables; ep; ep = ep->next) {
2649                 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
2700     if (channel->variables) {
2701         for (hi = channel->variables->headers; hi; hi = hi->next) {
...
2708             switch_event_add_header_string(event, SWITCH_STACK_BOTTOM, buf, vval);
2709         }
2710     }
2711 }
```

---

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 {
2699     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_pro
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 *
```

---

---

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
```

---

设置呼叫 Caller Profile。

---

```
2807 SWITCH_DECLARE(void) switch_channel_set_origination_caller_profile(switch_channel_t *channel, switch_caller_profile_t
```

---

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 *
```

```
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 *st
```

```
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 );
2932 SWITCH_DECLARE(void) switch_channel_clear_state_handler(switch_channel_t *channel, const switch_state_handler_table_t )

```

---

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,
2986                                         switch_channel_t *new_channel, uint32_t offset)
2987 {
2988     ...
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,
3010                                              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,
3020                                         ap->application_name, ap->application_data);
3020         }
3021
3022         caller_profile->destination_number = switch_core_strdup(caller_profile->pool,
3023                                                               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) {
3027             switch_channel_set_variable(new_channel, hi->name, hi->value);
3028         }
3029     }

```

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>
```

---

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 &&
3141         switch_channel_test_flag(channel, CF_BLEG)) {
3142         switch_channel_flip_cid(channel);
3143         switch_channel_clear_flag(channel, CF_BLEG);
3144     } else if (switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_OUTBOUND &&
3145                !switch_channel_test_flag(channel, CF_DIALPLAN)) {
3146         switch_channel_set_flag(channel, CF_DIALPLAN);
3147         switch_channel_flip_cid(channel);
3148     }
3149 }
```

---

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,
3162                                         switch_caller_extension_t *caller_extension)
3163 {
3164     switch_mutex_lock(channel->profile_mutex);
3165     channel->queued_extension = caller_extension;
3166     switch_mutex_unlock(channel->profile_mutex);
3167
3168     switch_channel_set_flag(channel, CF_TRANSFER);
3169     switch_channel_set_state(channel, CS_ROUTING);
3170 }
```

---

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)
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->brridged = 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,
3219                         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,
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,
3307                                         switch_ring_ready_t rv,
3308                                         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->u
...
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->
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_rwunlock(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_rwunlock(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,
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
3465     if ((uuid = switch_channel_get_variable(channel, SWITCH_ORIGINATOR_VARIABLE))
3466         && (other_session = switch_core_session_locate(uuid))) {
3467         switch_core_session_kill_channel(other_session, SWITCH_SIG_BREAK);
3468         switch_core_session_rwunlock(other_session);
3469     }
3470 }
```

---

```

3471     switch_channel_set_callstate(channel, CCS_EARLY);
3473     send_ind(channel, SWITCH_MESSAGE_PROGRESS_EVENT, file, func, line);
3475     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     ...
3486     if (switch_channel_direction(channel) == SWITCH_CALL_DIRECTION_INBOUND) {
3487         msg.message_id = SWITCH_MESSAGE_INDICATE_PROGRESS;
3488         msg.from = channel->name;
3489         status = switch_core_session_perform_receive_message(channel->session, &msg, file, func, line);
3490     }
3491
3492     if (status == SWITCH_STATUS_SUCCESS) {
3493         switch_channel_perform_mark_pre_answered(channel, file, func, line);
3494         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,
3519                                         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 {
3557     switch_stream_handle_t stream = { 0 };
3558
3559     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 (!strcasecmp(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 {
...
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 zrtp（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 *method)
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);
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
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_chann
```

---

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 {
4676     if (!(uuid = switch_channel_get_variable(channel, SWITCH_SIGNAL_BOND_VARIABLE))) {
4677         uuid = switch_channel_get_variable(channel, SWITCH_ORIGINATE_SIGNAL_BOND_VARIABLE);
```

---

---

```

4678     }
4680     return uuid;
4681 }
```

---

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 {
...
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];
...
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++) {
4750                     if (...) {
4753                         return;
4754                     }
4755                 }
4756             }
4757         }
...
4765     if (transfer_on_fail || failure_causes) {
...
4792         switch_ivr_session_transfer(session, tof_array[1], tof_array[2], tof_array[3]);
...
4814     }
4815 }
```

---

```
4819     switch_channel_get_state(channel) != CS_ROUTING) {  
4820         switch_channel_hangup(channel, cause);  
4821     }  
4822 }
```

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
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_SD_P_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_SD_P_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
95     ...
96
97     if (!(xml = switch_xml_open_cfg(cf, &cfg, NULL))) {
98         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Open of %s failed\n", cf);
99         return SWITCH_STATUS_TERM;
100    }
101
102    if ((settings = switch_xml_child(cfg, "cli-keybindings"))){
103        for (param = switch_xml_child(settings, "key"); param; param = param->next) {
104            char *var = (char *) switch_xml_attr_soft(param, "name");
105            char *val = (char *) switch_xml_attr_soft(param, "value");
106            i = atoi(var);
107
108            console_fnkeys[i - 1] = switch_core_permanent_strdup(val);
109        }
110    }
111 }
```

---

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,
127                                uint8_t *data, switch_size_t datalen)
128 {
129     switch_size_t need = handle->data_len + datalen;
130
131     if (need >= handle->data_size) {
132         void *new_data;
133         need += handle->alloc_chunk;
134
135         if (!(new_data = realloc(handle->data, need))) {
136             return SWITCH_STATUS_MEMERR;
137         }
138
139         handle->data = new_data;
140         handle->data_size = need;
141     }
142
143     memcpy((uint8_t *) (handle->data) + handle->data_len, data, datalen);
144     handle->data_len += datalen;
145     handle->end = (uint8_t *) (handle->data) + handle->data_len;
146     *(uint8_t *) handle->end = '\0';
147
148     return SWITCH_STATUS_SUCCESS;
149 }
```

---

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  {
152     va_start(ap, fmt);
```

---

```

164     if (!(data = switch_vmpprintf(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")) != NULL) {
...
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 sipt sofia global siptrace on
freeswitch> alias add sipf sofia global siptrace off
```

---

以后，只需要使用sipt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 {
317     FILE *handle = switch_core_get_console();
319
320     SWITCH_STANDARD_STREAM(stream);
322
323     status = switch_console_execute(xcmd, 0, &stream);
324
325     if (status == SWITCH_STATUS_SUCCESS) {
326         if (handle) {
327             fprintf(handle, "\n%s\n", (char *) stream.data);
328             fflush(handle);
329     }

```

---

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
406 {
431     switch_time_exp_lt(&tm, switch_micro_time_now());
432     switch_strerror_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, filep, 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);
443     switch_event_fire(&event);
444 }

```

---

命令补全和回调，暂不多说。

---

```

470 static int comp_callback(void *pArg, int argc, char **argv, char **columnNames)
471 {
472     return 0;
473 }
474
475 static int modulename_callback(void *pArg, const char *module_name)
476 {
477     return 0;
478 }
479
480 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_available_modules(const char *line,
481                         const char *cursor, switch_console_callback_match_t **matches)
482 {
483     return 0;
484 }
485
486 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_loaded_modules(const char *line,
487                         const char *cursor, switch_console_callback_match_t **matches)
488 {
489     return 0;
490 }
491
492 SWITCH_DECLARE_NONSTD(switch_status_t) switch_console_list_interfaces(const char *line, const char *cursor,
493                         switch_console_callback_match_t **matches)
494 {
495     return 0;
496 }

```

---

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_c
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_sw
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     }
1043     return prompt_str;
1044 }

```

---

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 {
1049     while (running) {
1050         if (getppid() == 1) break;
1051         switch_core_session_ctl(SCSC_CHECK_RUNNING, &arg);
1052         if (!arg) break;
1053
1054         line = el_gets(el, &count);
1055         if (count > 1) {
1056             if (!zstr(line)) {
1057                 char *cmd = strdup(line);
1058                 history(myhistory, &ev, H_ENTER, line);
1059                 running = switch_console_process(cmd);
1060             }
1061         }
1062         switch_cond_next();
1063     }
1064 }

```

---

命令补全函数。

---

```

1093 static unsigned char complete(EditLine * el, int ch)
1094 {
1095     const LineInfo *lf = el_line(el);
1096     return switch_console_complete(lf->buffer, lf->cursor, switch_core_get_console(), NULL, NULL);
1097 }

```

---

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 rfd, efd;
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_swit
1592         }
1593 #ifdef _MSC_VER
1594         if (cmd[0]) {
1595             running = switch_console_process(cmd);
1596         }
1597         Sleep(20);
1598 #else
1599         FD_ZERO(&rfd);
1600         if ((activity = select(fileno(stdin) + 1, &rfd, NULL, &efd, &tv
1601         if (cmd[0]) {
1602             running = switch_console_process(cmd);
1603         }
1604     }
1605 #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",
1672                                     (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
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,
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 {
75     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));
...
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");
176             switch_event_fire(&event);
177     }
178 }
```

---

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();
208     /* reschedule this task */
209     task->runtimetime = 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();
216     task->runtimetime = switch_epoch_time_now(NULL) + 60;
217 }
```

---

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)
1998
1999     switch_scheduler_add_task(switch_epoch_time_now(NULL), check_ip_callback, "check_ip", "core", 0, NULL, SSHF_NONE)
```

---

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) {
225
226     SWITCH_DECLARE(FILE *) switch_core_get_console(void)
227     {
228         return runtime.console;
229     }
```

---

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     }
278     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，在相应状态改变时会发生回调。

---

```

283 SWITCH_DECLARE(void) switch_core_remove_state_handler(const switch_state_table_t *state_handler)
284
308 SWITCH_DECLARE(int) switch_core_add_state_handler(const switch_state_table_t *state_handler)
309
325 SWITCH_DECLARE(const switch_state_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 (!zstr(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
496     if (switch_core_session_read_lock(session) != SWITCH_STATUS_SUCCESS) {
497         return NULL;
498     }
499
500     switch_mutex_lock(session->frame_read_mutex);
501     channel = switch_core_session_get_channel(session);
502
503     switch_channel_set_flag(channel, CF_SERVICE);
504     while (switch_channel_test_flag(channel, CF_SERVICE)) {
505         if (switch_channel_test_flag(channel, CF_SERVICE_AUDIO)) {
```

```

511     switch (switch_core_session_read_frame(session, &read_frame, SWITCH_IO_FLAG_NONE, 0)) {
512         case SWITCH_STATUS_SUCCESS:
513         case SWITCH_STATUS_TIMEOUT:
514         case SWITCH_STATUS_BREAK:
515             break;
516         default:
517             switch_channel_clear_flag(channel, CF_SERVICE);
518             break;
519     }
520 }
521
522 if (switch_channel_test_flag(channel, CF_SERVICE_VIDEO) && switch_channel_test_flag(channel, CF_VIDEO)) {
523 }
524
525 switch_mutex_unlock(session->frame_read_mutex);
526 switch_channel_clear_flag(channel, CF_SERVICE_AUDIO);
527 switch_channel_clear_flag(channel, CF_SERVICE_VIDEO);
528 switch_core_session_rwunlock(session);
529 return NULL;
530 }

```

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 audio, switch_bool_t
563 {
570     if (audio) switch_channel_set_flag(channel, CF_SERVICE_AUDIO);
571     if (video) switch_channel_set_flag(channel, CF_SERVICE_VIDEO);
572
573     switch_core_session_launch_thread(session, (void *(*)(switch_thread_t *,void *))switch_core_service_thread, session);
574 }

```

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 {
554     switch_channel_clear_flag(channel, CF_SERVICE);
555     switch_channel_clear_flag(channel, CF_SERVICE_AUDIO);

```

---

```

556     switch_channel_clear_flag(channel, CF_SERVICE_VIDEO);
558     switch_core_session_kill_channel(session, SWITCH_SIG_BREAK);
560 }

```

---

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

```

---

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 {
...
876     switch_assert(SWITCH_GLOBAL_dirs.base_dir);
877     switch_assert(SWITCH_GLOBAL_dirs.mod_dir);
878     switch_assert(SWITCH_GLOBAL_dirs.lib_dir);
879     switch_assert(SWITCH_GLOBAL_dirs.conf_dir);
880     switch_assert(SWITCH_GLOBAL_dirs.log_dir);
881     switch_assert(SWITCH_GLOBAL_dirs.run_dir);
882     switch_assert(SWITCH_GLOBAL_dirs.db_dir);
883     switch_assert(SWITCH_GLOBAL_dirs.script_dir);
884     switch_assert(SWITCH_GLOBAL_dirs.htdocs_dir);
885     switch_assert(SWITCH_GLOBAL_dirs.grammar_dir);
886     switch_assert(SWITCH_GLOBAL_dirs.fonts_dir);
887     switch_assert(SWITCH_GLOBAL_dirs.images_dir);
888     switch_assert(SWITCH_GLOBAL_dirs.recordings_dir);
889     switch_assert(SWITCH_GLOBAL_dirs.sounds_dir);
890     switch_assert(SWITCH_GLOBAL_dirs.certs_dir);
891     switch_assert(SWITCH_GLOBAL_dirs.temp_dir);
892     switch_assert(SWITCH_GLOBAL_dirs.data_dir);
893     switch_assert(SWITCH_GLOBAL_dirs.localstate_dir);
894     switch_assert(SWITCH_GLOBAL_filenames.conf_name);
895
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
930 #endif

```

---

---

```
931     return 0;
932 }
```

---

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 {
1064     if (!runtime.cpu_count) runtime.cpu_count = 1;
1065
1066     return set_realtime_priority();
1070 }
```

---

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)
```

---

MIME<sup>5</sup>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。

<sup>5</sup>参见：<https://zh.wikipedia.org/wiki/多用途互聯網郵件擴展>

---

```

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 }
```

---

L1200 和 L1208，在扩展名（如.html）MIME 类型（如text/html）间转换。实际上就是查哈希表。该哈希表是使用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哈希表（L1298）。

---

```

1200 SWITCH_DECLARE(const char *) switch_core_mime_ext2type(const char *ext)
1201 {
1205     return (const char *) switch_core_hash_find(runtime.mime_types, ext);
1206 }
1207
1208 SWITCH_DECLARE(const char *) switch_core_mime_type2ext(const char *mime)
1209 {
1213     return (const char *) switch_core_hash_find(runtime.mime_type_exts, mime);
1214 }
```

---

L1216，返回 MIME 类型的 Hash Index，以便遍历。

---

```

1216 SWITCH_DECLARE(switch_hash_index_t *) switch_core_mime_index(void)
1217 {
1218     return switch_core_hash_first(runtime.mime_types);
1219 }
```

---

L1221，前 MIME 类型插入哈希表。

---

```
1221 SWITCH_DECLARE(switch_status_t) switch_core_mime_add_type(const char *type, const char *ext)
```

---

L1261 加载 MIME 表。就是从一个配置文件中依次读出第一行 (L1278)，然后解析并调用 L1221 中的函数插入一个哈希表，备用。

---

```
1261 static void load_mime_types(void)
1262 {
1263     char *cf = "mime.types";
1264
1265     mime_path = switch_mprintf("%s/%s", SWITCH_GLOBAL_dirs.conf_dir, cf);
1266     while ((switch_fp_read_dline(fd, &line_buf, &llen))) {
1267         switch_core_mime_add_type(type, p);
1268     }
1269 }
```

---

L1315，设置系统资源限制，如堆栈大小 (L1330)，最大打开文件数 (L1337) 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。

---

```
1315 SWITCH_DECLARE(void) switch_core_setrlimits(void)
1316 {
1317     rlp.rlim_cur = SWITCH_THREAD_STACKSIZE;
1318     rlp.rlim_max = SWITCH_SYSTEM_THREAD_STACKSIZE;
1319     setrlimit(RLIMIT_NOFILE, &rlp);
```

---

L1360 ~ L1365，定义一个 IP 地址列表数据结构和变量，内部用哈希存储。

```
1360 typedef struct { 1361 switch_memory_pool_t pool; 1362 switch_hash_t hash; 1363 }
switch_ip_list_t; 1364 1365 static switch_ip_list_t IP_LIST = { 0 };
```

L1367 ~ 1438，检查 IP 地址 (`ip_str`) 是否在对应的 ACL 列表中，如果是，则返回 True，并且，返回一个 `token`，该 `token` 通常是一个标记。

---

```
1367 SWITCH_DECLARE(switch_bool_t) switch_check_network_list_ip_token(const char *ip_str, const char *list_name, const char *
```

---

L1441 ~ 1667，装入 ACL 列表。其中，L1451 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`。

---

```

1441 SWITCH_DECLARE(void) switch_load_network_lists(switch_bool_t reload)
1442 {
1451     switch_find_local_ip(guess_ip, sizeof(guess_ip), &mask, AF_INET);

```

---

接下来构建各种列表。

---

```

1470     tmp_name = "rfc6598.auto";
1473     switch_network_list_add_cidr(rfc_list, "100.64.0.0/10", SWITCH_TRUE);
1475
1476     tmp_name = "rfc1918.auto";
1479     switch_network_list_add_cidr(rfc_list, "10.0.0.0/8", SWITCH_TRUE);
1480     switch_network_list_add_cidr(rfc_list, "172.16.0.0/12", SWITCH_TRUE);
1481     switch_network_list_add_cidr(rfc_list, "192.168.0.0/16", SWITCH_TRUE);
1485     tmp_name = "wan.auto";
1496     tmp_name = "wan_v6.auto";
1504     tmp_name = "wan_v4.auto";
1516     tmp_name = "any_v6.auto";
1523     tmp_name = "any_v4.auto";
1530     tmp_name = "nat.auto";
1542     tmp_name = "loopback.auto";
1549     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。

---

```

1559     if ((xml = switch_xml_open_cfg("acl.conf", &cfg, NULL))) {
1560         if ((x_lists = switch_xml_child(cfg, "network-lists"))) {
1561             for (x_list = switch_xml_child(x_lists, "list"); x_list; x_list = x_list->next) {

```

---

如果列表的某一个 Node 中有`domain`属性，则，进一步检查用户目录（User Directory），看该`domain`中配置了哪些用户（L1609）。遍历所有用户，如果某一用户有`cidr`属性（L1622 ~ L1625），则也为该`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 Chanllenge 验证（即仅验证来源 IP）。

---

```

1597             domain = switch_xml_attr(x_node, "domain");
1598
1599             if (domain) {
1609                 if (switch_xml_locate_domain(domain, my_params, &xml_root, &x_domain) != SWITCH_STATUS_SUCCESS)
1620
1621                 for (ut = switch_xml_child(x_domain, "user"); ut; ut = ut->next) {

```

---

---

```

1622         const char *user_cidr = switch_xml_attr(ut, "cidr");
1623         const char *id = switch_xml_attr(ut, "id");
1624
1625         if (id && user_cidr) {
1626             char *token = switch_mprintf("%s@%s", id, domain);
1627             switch_assert(token);
1628             switch_network_list_add_cidr_token(list, user_cidr, ok, token);

```

---

设置并返回最大 (L1669) 或最小 (L1709) DTMF 间隔。

---

```

1669 SWITCH_DECLARE(uint32_t) switch_core_max_dtmf_duration(uint32_t duration)
1670 {
1671     if (duration) {
1672         if (duration > SWITCH_MAX_DTMF_DURATION) {
1673             duration = SWITCH_MAX_DTMF_DURATION;
1674         }
1675         if (duration < SWITCH_MIN_DTMF_DURATION) {
1676             duration = SWITCH_MIN_DTMF_DURATION;
1677         }
1678         runtime.max_dtmf_duration = duration;
1679         if (duration < runtime.min_dtmf_duration) {
1680             runtime.min_dtmf_duration = duration;
1681         }
1682     }
1683     return runtime.max_dtmf_duration;
1684 }
...
1709 SWITCH_DECLARE(uint32_t) switch_core_min_dtmf_duration(uint32_t duration)

```

---

返回默认的 DTMF 间隔。

---

```
1686 SWITCH_DECLARE(uint32_t) switch_core_default_dtmf_duration(uint32_t duration)
```

---

设置 CPU 亲缘性，通过将某些线程绑定到特定的 CPU 上，避免过多的上下文切换开销。

---

```
1728 SWITCH_DECLARE(switch_status_t) switch_core_thread_set_cpu_affinity(int cpu)
```

---

如果启用了 ZRTP，L1757，设置 FreeSWITCH 的序列号。FreeSWITCH 首次启动将产生一个随机的序列号 (L1766)，并设置全局变量 `switch_serial`。

---

```

1757 #ifdef ENABLE_ZRTP
1758 static void switch_core_set_serial(void)
1759 {
1760     switch_snprintf(path, sizeof(path), "%s%sfreeswitch.serial", SWITCH_GLOBAL_dirs.conf_dir, SWITCH_PATH_SEPARATOR);
1761     switch_core_set_variable("switch_serial", buf);
1762 }
1763 #endif

```

---

检测核心标志。

---

```

1805 SWITCH_DECLARE(int) switch_core_test_flag(int flag)
1806 {
1807     return switch_test_flag(&runtime), flag);
1808 }

```

---

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。

---

```

1811 SWITCH_DECLARE(switch_status_t) switch_core_init(switch_core_flag_t flags, switch_bool_t console, const char **err)
1812 {

```

---

SIGBUS 回调函数，其实没什么用。

---

```

2009 #ifdef TRAP_BUS
2010 static void handle_SIGBUS(int sig)
2015 #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。

---

```

2017 static void handle_SIGHUP(int sig)
2018 {
2019     if (sig) {
2020         switch_event_t *event;
2021
2022         if (switch_event_create(&event, SWITCH_EVENT_TRAP) == SWITCH_STATUS_SUCCESS) {
2023             switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Trapped-Signal", "HUP");
2024             switch_event_fire(&event);

```

---

设置默认的打包间隔。

---

```
2031 SWITCH_DECLARE(uint32_t) switch_default_ptime(const char *name, uint32_t number)
2032 {
2033     uint32_t *p;
2035     if ((p = switch_core_hash_find(runtime.ptimes, name))) {
2036         return *p;
2037     }
2039     return 20;
2040 }
```

---

设置默认的采样率。

---

```
2042 SWITCH_DECLARE(uint32_t) switch_default_rate(const char *name, uint32_t number)
2043 {
2045     if (!strcasecmp(name, "opus")) {
2046         return 48000;
2047     } else if (!strcasecmp(name, "h26", 3)) { // h26x
2048         return 90000;
2049     } else if (!strcasecmp(name, "vp", 2)) { // vp8, vp9
2050         return 90000;
2051     }
2053     return 8000;
2054 }
```

---

L2056，有些编码如 iLBC、iSAC、G723 等默认的打包间隔是30ms。

L2058，加载核心的设置，参见switch.conf及post\_load\_switch.conf。

---

```
2056 static uint32_t d_30 = 30;
2057
2058 static void switch_load_core_config(const char *file)
```

---

Banner。

---

```
2347 SWITCH_DECLARE(const char *) switch_core_banner(void)
```

---

L2370，初始化并加载模块。L2379 初始化核心参数及数据结构。L2398 加载模块。模块加载完成后，L2406 再加载post\_load\_switch.conf。L2410 产生一个事件。L2444，如果设置了api\_on\_startup，则执行一个 API。

---

```

2370 SWITCH_DECLARE(switch_status_t) switch_core_init_and_modload(switch_core_flag_t flags, switch_bool_t console, const char *modlist)
2371 {
2372     if (switch_core_init(flags, console, err) != SWITCH_STATUS_SUCCESS) {
2373         return SWITCH_STATUS_GENERR;
2374     }
2375     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Loading Modules.\n");
2376     if (switch_loadable_module_init(SWITCH_TRUE) != SWITCH_STATUS_SUCCESS) {
2377     }
2378     switch_load_network_lists(SWITCH_FALSE);
2379     switch_load_core_config("post_load_switch.conf");
2380     if (switch_event_create(&event, SWITCH_EVENT_STARTUP) == SWITCH_STATUS_SUCCESS) {
2381         switch_event_add_header(event, SWITCH_STACK_BOTTOM, "Event-Info", "System Ready");
2382         switch_event_fire(&event);
2383     }
2384     if ((cmd = switch_core_get_variable_dup("api_on_startup"))) {
2385         switch_stream_handle_t stream = { 0 };
2386         SWITCH_STANDARD_STREAM(stream);
2387         switch_console_execute(cmd, 0, &stream);
2388         switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CONSOLE, "Startup command [%s] executed. Output:\n%s\n", cmd->name, stream.data);
2389         free(stream.data);
2390         free(cmd);

```

---

计算某一时刻对应的年月日时分秒等。

---

```

2457 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。

---

```

2480 #ifdef _MSC_VER
2481 static void win_shutdown(void)
2482#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。

---

```

2501 SWITCH_DECLARE(void) switch_core_set_signal_handlers(void)
2502 {
2503     /* set signal handlers */
2504     signal(SIGINT, SIG_IGN);
2505 #ifdef SIGUSR1
2506     signal(SIGUSR1, handle_SIGHUP);
2507 #endif
2508     signal(SIGHUP, handle_SIGHUP);
2509 }
```

---

返回debug\_level。

---

```

2529 SWITCH_DECLARE(uint32_t) switch_core_debug_level(void)
2530 {
2531     return runtime.debug_level;
2532 }
```

---

L2535 ~ L2857 是核心控制函数，主要完成fsctl相关的命令。如 L2789 修改调试级别、L2806 修改最大并发数、L2836 修改每秒最大启动的 Session 数量等、L2884 回收内存等。

---

```

2535 SWITCH_DECLARE(int32_t) switch_core_session_ctl(switch_session_ctl_t cmd, void *val)
2536 {
2537     switch (cmd) {
2538         case SCSC_RECOVER:
2539         case SCSC_DEBUG_SQL:
2540         case SCSC_LOGLEVEL:
2541         case SCSC_DEBUG_LEVEL:
2542         case SCSC_MAX_SESSIONS:
2543             newintval = switch_core_session_limit(oldintval);
2544             break;
2545         case SCSC_SPS:
2546             switch_mutex_lock(runtime.throttle_mutex);
2547             if (oldintval > 0) {
2548                 runtime.sps_total = oldintval;
2549             }
2550             newintval = runtime.sps_total;
2551             switch_mutex_unlock(runtime.throttle_mutex);
2552             break;
2553         case SCSC_RECLAIM:
```

---

---

```

2885     switch_core_memory_reclaim_all();
2886     return 0;
2887 }
```

---

下面一些函数只是检查相应的核心变量并返回当前状态的布尔值。

---

```

2859 SWITCH_DECLARE(switch_core_flag_t) switch_core_flags(void)
2860 SWITCH_DECLARE(switch_bool_t) switch_core_running(void)
2861 SWITCH_DECLARE(switch_bool_t) switch_core_ready(void)
2862 SWITCH_DECLARE(switch_bool_t) switch_core_ready_inbound(void)
2863 SWITCH_DECLARE(switch_bool_t) switch_core_ready_outbound(void)
```

---

L2884 ~ L2988 在 FreeSWITCH 关闭时释放相关内存，关闭 Socket、清理现场等。如 L2900 卸载所有模块、L2909 ~ L2910 释放 RTP 和 MSRP。

```

2884 SWITCH_DECLARE(switch_status_t) switch_core_destroy(void) 2885 {
2885 switch_loadable_module_shutdown(); 2900 2909 switch_rtp_shutdown(); 2910 switch_msrp_destroy();
2886 } 2976 } ``
```

L2978 管理接口相关函数。

---

```

2978 SWITCH_DECLARE(switch_status_t) switch_core_management_exec(char *relative_oid, switch_management_action_t action, char
2979 )
```

---

L2990 ~ L2995，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 L2884 被调用。

---

```

2990 SWITCH_DECLARE(void) switch_core_memory_reclaim_all(void)
2991 {
2992     switch_core_memory_reclaim_logger();
2993     switch_core_memory_reclaim_events();
2994     switch_core_memory_reclaim();
2995 }
```

---

L2998 定义了一个系统线程句柄结构。

---

```

2998 struct system_thread_handle {
2999     const char *cmd;
```

---

---

```

3000     switch_thread_cond_t *cond;
3001     switch_mutex_t *mutex;
3002     switch_memory_pool_t *pool;
3003     int ret;
3004     int *fds;
3005 };

```

---

L3007 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 L3009 行被强制转换为`struct system_thread_handle`结构。它将执行 L3031 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，L3078 获取一个互斥锁，然后给`sth->cond`发一个信号（L3079），通知其它线程该函数执行完毕了。

---

```

3007 static void *SWITCH_THREAD_FUNC system_thread(switch_thread_t *thread, void *obj)
3008 {
3009     struct system_thread_handle *sth = (struct system_thread_handle *) obj;
3010     sth->ret = system(sth->cmd);
3011     switch_mutex_lock(sth->mutex);
3012     switch_thread_cond_signal(sth->cond);
3013     switch_mutex_unlock(sth->mutex);
3046 }

```

---

下面是真正启动线程执行命令的函数。L3057 初始化一个内存池。L3062 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（L3067~L3072）。注意 L3070 行初始化了一个`cond`，它是一个条件变量。一切准备好后，在 L3077 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。

然后，如果需要等待（L3079），它将一直等待`sth->cond`（L3079），直到回调函数执行完毕并给它发一个信号（L3079），表示`system`线程执行完了。

---

```

3049 static int switch_system_thread(const char *cmd, switch_bool_t wait)
3050 {
3051     if (switch_core_new_memory_pool(&pool) != SWITCH_STATUS_SUCCESS) {
3060     }
3061
3062     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);
3075     switch_threadattr_stacksize_set(thd_attr, SWITCH_SYSTEM_THREAD_STACKSIZE);
3076     switch_threadattr_detach_set(thd_attr, 1);
3077     switch_thread_create(&thread, thd_attr, system_thread, sth, sth->pool);
3078     if (wait) {
3079         switch_thread_cond_wait(sth->cond, sth->mutex);
3080         ret = sth->ret;
3081     }
3082     switch_mutex_unlock(sth->mutex);
3083
3084     return ret;
3085
3086 }
```

---

L3088，返回进程能打开的最大的文件描述符数量。

---

```
3088 SWITCH_DECLARE(int) switch_max_file_desc(void)
```

---

关闭不再使用的文件描述符。该函数在 L3240 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。

---

```
3104 SWITCH_DECLARE(void) switch_close_extra_files(int *keep, int keep_ttl)
```

---

L3128 ~ L3290 都是执行`system`相关的函数，略。

获取 RTP 起始和结束端口号。

---

```
3292 SWITCH_DECLARE(uint16_t) switch_core_get_rtp_port_range_start_port()
3293 SWITCH_DECLARE(uint16_t) switch_core_get_rtp_port_range_end_port()
```

---

## 4.11 switch\_core\_asr.c

以下代码基于 Git Commit c9aa3522。

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 {
45     switch_status_t status;
46     char buf[256] = "";
47     char *param = NULL;
48
49     if (strchr(module_name, ':')) {
50         switch_set_string(buf, module_name);
51         if ((param = strchr(buf, ':'))) {
52             *param++ = '\0';
53             module_name = buf;
54         }
55     }

```

---

L58, 获取 ASR 模块的入口函数。设置 ASR 句柄 (ah) 相关的参数后 (L63, L66, L75 ~ L81), 调用 ASR 模块的`asr_open`回调函数 (83) 以便打开 ASR 接口。

---

```

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     }
62
63     ah->flags = *flags;
64
65     if (pool) {
66         ah->memory_pool = pool;
67     } else {
68         if ((status = switch_core_new_memory_pool(&ah->memory_pool)) != SWITCH_STATUS_SUCCESS) {
69             UNPROTECT_INTERFACE(ah->asr_interface);
70             return status;
71         }
72         switch_set_flag(ah, SWITCH_ASR_FLAG_FREE_POOL);
73     }
74
75     if (param) {
76         ah->param = switch_core_strdup(ah->memory_pool, param);
77     }
78     ah->rate = rate;
79     ah->name = switch_core_strdup(ah->memory_pool, module_name);
80     ah->samplerate = rate;
81     ah->native_rate = rate;
82

```

---

---

```

83     status = ah->asr_interface->asr_open(ah, codec, rate, dest, flags);
84
85     if (status != SWITCH_STATUS_SUCCESS) {
86         UNPROTECT_INTERFACE(ah->asr_interface);
87     }
88
89     return status;
91 }
```

---

L93 ~ L154，加载语法文件，并最终调用相关模块的`asr_load_grammar`函数加载语法文件。

---

```

93 SWITCH_DECLARE(switch_status_t) switch_core_asr_load_grammar(switch_asr_handle_t *ah, const char *grammar, const char *name)
94 {
148     status = ah->asr_interface->asr_load_grammar(ah, data, name);
154 }
```

---

卸载语法文件。

---

```

156 SWITCH_DECLARE(switch_status_t) switch_core_asr_unload_grammar(switch_asr_handle_t *ah, const char *name)
157 {
161     status = ah->asr_interface->asr_unload_grammar(ah, name);
164 }
```

---

启用和禁用语法文件。

---

```

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)
```

---

暂停语音识别。

---

```

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 }
```

---

继续语音识别。

---

```
212 SWITCH_DECLARE(switch_status_t) switch_core_asr_resume(switch_asr_handle_t *ah)
213 {
214     return ah->asr_interface->asr_resume(ah);
215 }
```

---

关闭语音识别接口，并释放相关资源。

---

```
219 SWITCH_DECLARE(switch_status_t) switch_core_asr_close(switch_asr_handle_t *ah, switch_asr_flag_t *flags)
```

---

向 ASR 中“喂”数据。这个函数没什么特别的。只是如果在采样率与 ASR 所支持的采样率不匹配的情况下（L244），会启动一个采样转换器（resampler，L246），以便转换成匹配的采样率。

其中，数据的长用len表示，但实际的数据是用2个字节整数表示的，所以，可以看到类似/2及\*2的转换（L253、L257），这是因为有的函数（如L253）是以整数来计算的，而有的函数（如L257）是以字节数来计算长度的。

---

```
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 {
241     switch_size_t orig_len = len;
242     switch_assert(ah != NULL);
243
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,
248                                         SWITCH_RESAMPLE_QUALITY, 1) != SWITCH_STATUS_SUCCESS) {
249                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Unable to create resampler!\n");
250                 return SWITCH_STATUS_GENERR;
251             }
252
253             switch_resample_process(ah->resampler, data, len / 2);
254             if (ah->resampler->to_len > orig_len) {
255                 if (!ah->dbuf) {
256                     void *mem;
257                     ah->dbuflen = ah->resampler->to_len * 2;
258                     mem = realloc(ah->dbuf, ah->dbuflen);
259                     switch_assert(mem);
260                     ah->dbuf = mem;
261                 }
262             }
263         }
264     }
265 }
```

---

```

262     switch_assert(ah->resampler->to_len * 2 <= ah->dbuflen);
263     memcpy(ah->dbuf, ah->resampler->to, ah->resampler->to_len * 2);
264     data = ah->dbuf;
265 } else {
266     memcpy(data, ah->resampler->to, ah->resampler->to_len * 2);
267 }
268
269     len = ah->resampler->to_len;
270 }
271
272     return ah->asr_interface->asr_feed(ah, data, len, flags);
273 }

```

---

“喂”DTMF。

```
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     return ah->asr_interface->asr_check_results(ah, flags);
290 }
```

---

获取识别结果。

```

295 SWITCH_DECLARE(switch_status_t) switch_core_asr_get_results(switch_asr_handle_t *ah, char **xmlstr, switch_asr_flag_t *flags);
296 {
297     return ah->asr_interface->asr_get_results(ah, xmlstr, flags);
298 }
```

---

获取识别结果的头域。

```
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);
```

---

开启输入计时器。

```
314 SWITCH_DECLARE(switch_status_t) switch_core_asr_start_input_timers(switch_asr_handle_t *ah);
```

---

向 ASR 传送不同类型的参数。

---

```
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)
```

---

本文件实现的函数只是 ASR 接口的一些抽象，绝大部分函数都会调用实际的实现语音识别功能的模块中定义的相关函数。

## 4.12 switch\_core\_cert.c

本文件实现证书相关功能。

L39 定义一个回调函数，当 SSL 需要获取锁时便执行该回调，进行加锁（L42）或解锁（L45）。

---

```
35 static switch_mutex_t **ssl_mutexes;
36 static switch_memory_pool_t *ssl_pool = NULL;
37 static int ssl_count = 0;
38
39 static inline void switch_ssl_ssl_lock_callback(int mode, int type, char *file, int line)
40 {
41     if (mode & CRYPTO_LOCK) {
42         switch_mutex_lock(ssl_mutexes[type]);
43     }
44     else {
45         switch_mutex_unlock(ssl_mutexes[type]);
46     }
47 }
```

---

获取 SSL 线程 ID。

---

```
49 static inline unsigned long switch_ssl_ssl_thread_id(void)
50 {
51     return (unsigned long) switch_thread_self();
52 }
```

---

初始化及销毁 SSL 相关的锁。

```
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)
```

---

检查证书指纹。

```
133 SWITCH_DECLARE(int) switch_core_cert_verify(dtls_fingerprint_t *fp)
```

---

将指纹字符串扩展内存中的格式。

```
152 SWITCH_DECLARE(int) switch_core_cert_expand_fingerprint(dtls_fingerprint_t *fp, const char *str)
```

---

从 X509 证书中提取指纹。

```
168 SWITCH_DECLARE(int) switch_core_cert_extract_fingerprint(X509* x509, 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。

```
240 SWITCH_DECLARE(int) switch_core_gen_certs(const char *prefix)
```

---

产生 X509 证书。

```
336 static int mkcert(X509 **x509p, EVP_PKEY **pkeyp, int bits, int serial, int days)
```

---

本文其实没有什么太多需要解释的，主要是调用 OpenSSL 生成和检查证书。

## 4.13 switch\_core\_codec.c

核心编解码接口实现。

L39 ~ L41，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 }
```

---

将当前 Session 的读写 Codec 清除。

---

```

46 SWITCH_DECLARE(void) switch_core_session_unset_read_codec(switch_core_session_t *session)
47 {
48     switch_mutex_t *mutex = NULL;
49
50     switch_mutex_lock(session->codec_read_mutex);
51     if (session->read_codec) mutex = session->read_codec->mutex;
52     if (mutex) switch_mutex_lock(mutex);
53     session->real_read_codec = session->read_codec = NULL;
54     session->raw_read_frame.codec = session->read_codec;
55     session->raw_write_frame.codec = session->read_codec;
56     session->enc_read_frame.codec = session->read_codec;
57     session->enc_write_frame.codec = session->read_codec;
58     if (mutex) switch_mutex_unlock(mutex);
59     switch_mutex_unlock(session->codec_read_mutex);
60 }
```

```

82 SWITCH_DECLARE(void) switch_core_session_unset_write_codec(switch_core_session_t *session)
83 {
84     switch_mutex_t *mutex = NULL;
85
86     switch_mutex_lock(session->codec_write_mutex);
87     if (session->write_codec) mutex = session->write_codec->mutex;
88     if (mutex) switch_mutex_lock(mutex);
89     session->real_write_codec = session->write_codec = NULL;
90     if (mutex) switch_mutex_unlock(mutex);
91     switch_mutex_unlock(session->codec_write_mutex);
92 }
```

---

在访问 Session 上的 Codec 时，通常需要加锁和解锁。

---

```

62 SWITCH_DECLARE(void) switch_core_session_lock_codec_write(switch_core_session_t *session)
63 {
64     switch_mutex_lock(session->codec_write_mutex);
65 }
66
67 SWITCH_DECLARE(void) switch_core_session_unlock_codec_write(switch_core_session_t *session)
68 {
69     switch_mutex_unlock(session->codec_write_mutex);
70 }
71
72 SWITCH_DECLARE(void) switch_core_session_lock_codec_read(switch_core_session_t *session)
73 {
74     switch_mutex_lock(session->codec_read_mutex);
75 }
76
77 SWITCH_DECLARE(void) switch_core_session_unlock_codec_read(switch_core_session_t *session)
78 {
79     switch_mutex_unlock(session->codec_read_mutex);
80 }

```

---

读取 Session 上 Read Codec 和 Write Codec，以及相关的实现（Implementation）的相关函数。  
根据情况，可以分别在读写方向上使用不同的 Codec，具体的 Codec 实现在其它模块中实现。

---

```

94 SWITCH_DECLARE(switch_status_t) switch_core_session_set_real_read_codec(switch_core_session_t *session, switch_codec_t *)
197 SWITCH_DECLARE(switch_status_t) switch_core_session_set_read_codec(switch_core_session_t *session, switch_codec_t *codec)
305 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_effective_read_codec(switch_core_session_t *session)
312 SWITCH_DECLARE(switch_codec_t *) switch_core_session_get_read_codec(switch_core_session_t *session)
319 SWITCH_DECLARE(switch_status_t) switch_core_session_get_read_impl(switch_core_session_t *session, switch_codec_implementer_t *)
331 SWITCH_DECLARE(switch_status_t) switch_core_session_get_real_read_impl(switch_core_session_t *session, switch_codec_implementer_t *)
341 SWITCH_DECLARE(switch_status_t) switch_core_session_get_write_impl(switch_core_session_t *session, switch_codec_implementer_t *)
353 SWITCH_DECLARE(switch_status_t) switch_core_session_get_video_read_impl(switch_core_session_t *session, switch_codec_implementer_t *)
365 SWITCH_DECLARE(switch_status_t) switch_core_session_get_video_write_impl(switch_core_session_t *session, switch_codec_implementer_t *)
378 SWITCH_DECLARE(switch_status_t) switch_core_session_set_read_impl(switch_core_session_t *session, const switch_codec_implementer_t *)
384 SWITCH_DECLARE(switch_status_t) switch_core_session_set_write_impl(switch_core_session_t *session, const switch_codec_implementer_t *)

```

---

```

390 SWITCH_DECLARE(switch_status_t) switch_core_session_set_video_read_impl(switch_core_session_t *session, const switch_codec_t *codec);
396 SWITCH_DECLARE(switch_status_t) switch_core_session_set_video_write_impl(switch_core_session_t *session, const switch_codec_t *codec);
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fmtp, fmtp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_fmtp(const char *codec_name, const char *fmtp, uint32_t rate, s
581 {
591     if ((codec_interface = switch_loadable_module_get_codec_interface(codec_name, NULL))) {
592         if (codec_interface->parse_fmtp) {
593             codec_fmtp->actual_samples_per_second = rate;
594             status = codec_interface->parse_fmtp(fmtp, codec_fmtp);
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, switch_codec_t *new_codec,
615                                         const switch_codec_settings_t *codec_settings, switch_memory_pool_t *pool)
```

---

初始化一个 Codec。 `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。

`fmtp` 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 20ms。

`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 64 kbps，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,
634                                         const char *modname, const char *fmtp,
635                                         uint32_t rate, int ms, int channels, uint32_t bitrate, uint32_t flags,
636                                         const switch_codec_settings_t *codec_settings, switch_memory_pool_t *pool)
637 }
```

---

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     }
658
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     }
```

---

L677, 查找一个合适的编码实现 (Implementation)。

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。

---

```
676     if (!ms) {
677         for (iptr = codec_interface->implementations; iptr; iptr = iptr->next) {
678             uint32_t crate = !strcasecmp(codec_name, "g722") ? iptr->samples_per_second : iptr->actual_samples_per_second;
679             if ((!rate || rate == crate) && (!bitrate || bitrate == (uint32_t)iptr->bits_per_second) &&
680                 (20 == (iptr->microseconds_per_packet / 1000)) && (!channels || channels == iptr->number_of_channels)) {
681                 implementation = iptr;
682                 goto found;
683             }
684         }
685     }
```

---

如果没找到, 变着法继续找 (L688)。

---

```
687     /* Either looking for a specific interval or there was no interval specified and there wasn't one @20ms available */
688     for (iptr = codec_interface->implementations; iptr; iptr = iptr->next) {
689         uint32_t crate = !strcasecmp(codec_name, "g722") ? iptr->samples_per_second : iptr->actual_samples_per_second;
690         if ((!rate || rate == crate) && (!bitrate || bitrate == (uint32_t)iptr->bits_per_second) &&
691             (!ms || ms == (iptr->microseconds_per_packet / 1000)) && (!channels || channels == iptr->number_of_channels)) {
692             implementation = iptr;
693             break;
694         }
695     }
696
697     found:
698
699     if (implementation) {
```

---

如果找到了具体的实现模块, 则调用该模块的init函数对 Codec 进行初始化 (L718), 否则打印错误。

---

```
718     implementation->init(codec, flags, codec_settings);
```

---

L733，对音频数据进行编码。

`decoded_data`为未编码的原始 PCM 数据，`decoded_data_len`为数据长度。`decoded_rate`为采样率。

`encodec_*`则为编码后的数据。

`flag`为编码标志，通常为0。

该函数会调用具体的实现模块中的`encode`函数进行编码，编码完成后即可以访问`encodec_*`中的编码后的数据。

---

```

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,
739                                         uint32_t *encoded_rate, unsigned int *flag)
740 {
755     status = codec->implementation->encode(codec, other_codec, decoded_data, decoded_data_len,
756                                             encoded_rate, encoded_data, encoded_data_len, encoded_rate, flag);
763 }
```

---

解码，参数与编码类似。

---

```

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 {
781     status = codec->implementation->decode(codec, other_codec, encoded_data, encoded_data_len, encoded_rate,
782                                             decoded_data, decoded_data_len, decoded_rate, flag);
786 }
```

---

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 *frame)
809 {
826     if (codec->implementation->encode_video) {
827         status = codec->implementation->encode_video(codec, frame);
828
829         if (status == SWITCH_STATUS_MORE_DATA) {
830             frame->flags |= SFF_SAME_IMAGE;
831         } else {
832             frame->flags &= ~SFF_SAME_IMAGE;
833         }
834
835         frame->packetlen = frame->datalen + 12;
836     }
837
838     if (codec->mutex) switch_mutex_unlock(codec->mutex);
839
840     return status;
841 }
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 {
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 {
881     if (codec->implementation->codec_control) {
882         status = codec->implementation->codec_control(codec, cmd, ctype, cmd_data, atype, cmd_arg, rtype, ret_data);
883     }
884 }

```

---

L907, 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

核心数据库接口。核心数据库默认使用 SQLite。

L38, 计算数据库路径。

---

```

36 #include <sqlite3.h>
37
38 static void db_pick_path(const char *dbname, char *buf, switch_size_t size)
39 {
40     memset(buf, 0, size);
41     if (switch_is_file_path(dbname)) {
42         strncpy(buf, dbname, size);
43     } else {
44         switch_snprintf(buf, size, "%s%s%s.db", SWITCH_GLOBAL_dirs.db_dir, SWITCH_PATH_SEPARATOR, dbname);
45     }
46 }

```

---

打开、关闭、查询、获取数据等，下面的函数都是对 SQLite 提供的函数的简单封装。其中 `switch_core_db_exec` (L86) 中，查询时如果遇到数据库忙等情况会多次重试 (L92)。

```
48 SWITCH_DECLARE(int) switch_core_db_open(const char *filename, switch_core_db_t **ppDb)
49 {
50     return sqlite3_open(filename, ppDb);
51 }
52
53 SWITCH_DECLARE(int) switch_core_db_close(switch_core_db_t *db)
54 {
55     return sqlite3_close(db);
56 }
57
58 SWITCH_DECLARE(const unsigned char *) switch_core_db_column_text(switch_core_db_stmt_t *stmt, int iCol)
59 {
60     const unsigned char *txt = sqlite3_column_text(stmt, iCol);
61
62     if (!strcasecmp((char *) stmt, "(null)")) {
63         memset(stmt, 0, 1);
64         txt = NULL;
65     }
66
67     return txt;
68 }
69
70
71 SWITCH_DECLARE(const char *) switch_core_db_column_name(switch_core_db_stmt_t *stmt, int N)
72 {
73     return sqlite3_column_name(stmt, N);
74 }
75
76 SWITCH_DECLARE(int) switch_core_db_column_count(switch_core_db_stmt_t *pStmt)
77 {
78     return sqlite3_column_count(pStmt);
79 }
80
81 SWITCH_DECLARE(const char *) switch_core_db_errmsg(switch_core_db_t *db)
82 {
83     return sqlite3_errmsg(db);
84 }
85
86 SWITCH_DECLARE(int) switch_core_db_exec(switch_core_db_t *db, const char *sql, switch_core_db_callback_func_t callback,
87 {
88     int ret = 0;
89     int sane = 300;
90     char *err = NULL;
91
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 }
104
105 if (errormsg) {
106     *errormsg = err;
107 } else if (err) {
108     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "SQL ERR [%s]\n", err);
109     switch_core_db_free(err);
110     err = NULL;
111 }
112
113 return ret;
114 }
115
116 SWITCH_DECLARE(int) switch_core_db_finalize(switch_core_db_stmt_t *pStmt)
117 {
118     return sqlite3_finalize(pStmt);
119 }
120
121 SWITCH_DECLARE(int) switch_core_db_prepare(switch_core_db_t *db, const char *zSql, int nBytes, switch_core_db_stmt_t **pStmt)
122 {
123     return sqlite3_prepare(db, zSql, nBytes, ppStmt, pzTail);
124 }
125
126 SWITCH_DECLARE(int) switch_core_db_step(switch_core_db_stmt_t *stmt)
127 {
128     return sqlite3_step(stmt);
129 }
130
131 SWITCH_DECLARE(int) switch_core_db_reset(switch_core_db_stmt_t *pStmt)
132 {
133     return sqlite3_reset(pStmt);
134 }
135
136 SWITCH_DECLARE(int) switch_core_db_bind_int(switch_core_db_stmt_t *pStmt, int i, int iValue)
137 {
138     return sqlite3_bind_int(pStmt, i, iValue);
139 }
140
141 SWITCH_DECLARE(int) switch_core_db_bind_int64(switch_core_db_stmt_t *pStmt, int i, int64_t iValue)
142 {
143     return sqlite3_bind_int64(pStmt, i, iValue);
144 }
145
```



## 4.15 switch\_core\_directory.c

实现了目录服务接口，该接口目前没有太多应用。

## 4.16 switch\_core\_event\_hook.c

本文件通过一个宏实现了如下的事件钩子（回调函数）。

---

```
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)
```

---

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

实现了文件接口。

L39，打开一个文件，入口参数有channels（期望的声道数）、rate（期望的速率）等。

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（L120）。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）。

---

```

36 #include <switch.h>
37 #include "private/switch_core_pvt.h"
38
39 SWITCH_DECLARE(switch_status_t) switch_core_perform_file_open(const char *file, const char *func, int line,
40                 switch_file_handle_t *fh,
41                 const char *file_path,
42                 uint32_t channels, uint32_t rate, unsigned int flags, switch_memory_pool_t *pool)
43 {
44     if (*file_path == '{') {
45         if ((modname = switch_event_get_header(fh->params, "modname"))) {
46             fh->modname = switch_core_strdup(fh->memory_pool, modname);

```

```

122      }
123
124      if ((val = switch_event_get_header(fh->params, "amplerate"))) {
125          if ((val = switch_event_get_header(fh->params, "channels"))) {
126              if ((val = switch_event_get_header(fh->params, "ab"))) {
127                  if ((val = switch_event_get_header(fh->params, "vb"))) {
128                      if ((val = switch_event_get_header(fh->params, "try_hardware_encoder"))) {
129
130      }

```

L263，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 L301 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。

```

263      if ((fh->file_interface = switch_loadable_module_get_file_interface(ext, fh->modname)) == 0) {
264          switch_goto_status(SWITCH_STATUS_GENERR, fail);
265      }
266
267
301      if ((status = fh->file_interface->file_open(fh, file_path)) != SWITCH_STATUS_SUCCESS) {
302          UNPROTECT_INTERFACE(fh->file_interface);
303          goto fail;
304      }
305
306
344      if (fh->pre_buffer_dataLEN) {
345          switch_buffer_create_dynamic(&fh->pre_buffer, fh->pre_buffer_dataLEN * fh->channels, fh->pre_buffer_dataLEN * fh->channels);
346          fh->pre_buffer_data = switch_core_alloc(fh->memory_pool, fh->pre_buffer_dataLEN * fh->channels);
347      }
348
349
373      return status;
374  }

```

L376，从文件中读取音频数据。

L396，首先尝试从缓存中读取数据。

```

376  SWITCH_DECLARE(switch_status_t) switch_core_file_read(switch_file_handle_t *fh, void *data, switch_size_t *len)
377  {
378
395      if (fh->buffer && switch_buffer_inuse(fh->buffer) >= *len * 2 * fh->channels) {
396          *len = switch_buffer_read(fh->buffer, data, orig_len * 2 * fh->channels) / 2 / fh->channels;
397          return *len == 0 ? SWITCH_STATUS_FALSE : SWITCH_STATUS_SUCCESS;
398      }

```

L411，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（L419，L431）。L413，`asis`代表该文件是否是一个 Native File。然后，再从`pre_buffer`中读取（436）。

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。

L429，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。

---

```

409     more:
410
411     if (fh->pre_buffer) {
412         switch_size_t rlen;
413         int asis = switch_test_flag(fh, SWITCH_FILE_NATIVE);
414
415         if (!switch_test_flag(fh, SWITCH_FILE_BUFFER_DONE)) {
416             rlen = asis ? fh->pre_buffer_datalen : fh->pre_buffer_datalen / 2 / fh->real_channels;
417
418             if (switch_buffer_inuse(fh->pre_buffer) < rlen * 2 * fh->channels) {
419                 if ((status = fh->file_interface->file_read(fh, fh->pre_buffer_data, &rlen)) == SWITCH_STATUS_BREAK) {
420                     return SWITCH_STATUS_BREAK;
421                 }
422
423                 if (status != SWITCH_STATUS_SUCCESS || !rlen) {
424                     switch_set_flag_locked(fh, SWITCH_FILE_BUFFER_DONE);
425                 } else {
426                     fh->samples_in += rlen;
427                     if (fh->real_channels != fh->channels && !switch_test_flag(fh, SWITCH_FILE_NOMUX)) {
428                         switch_mux_channels((int16_t *) fh->pre_buffer_data, rlen, fh->real_channels, fh->channels);
429                     }
430                     switch_buffer_write(fh->pre_buffer, fh->pre_buffer_data, asis ? rlen : rlen * 2 * fh->channels);
431                 }
432             }
433         }
434     }
435
436     rlen = switch_buffer_read(fh->pre_buffer, data, asis ? *len : *len * 2 * fh->channels);
437     *len = asis ? rlen : rlen / 2 / fh->channels;
438
439     if (*len == 0) {
440         switch_set_flag_locked(fh, SWITCH_FILE_DONE);
441         goto top;
442     } else {
443         status = SWITCH_STATUS_SUCCESS;
444     }

```

---

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（L448）。

如果不是 Native File 并且采样率不匹配 (L464) , 则尝试启动一个resampler (L466) 进行采样率转换 (L473) 。

---

```

446     } else {
447
448     if ((status = fh->file_interface->file_read(fh, data, len)) == SWITCH_STATUS_BREAK) {
449         return SWITCH_STATUS_BREAK;
450     }
451
452     if (!switch_test_flag(fh, SWITCH_FILE_NATIVE) && fh->native_rate != fh->samplerate) {
453         if (!fh->resampler) {
454             if (switch_resample_create(&fh->resampler,
455                 fh->native_rate, fh->samplerate, (uint32_t) orig_len, SWITCH_RESAMPLE_QUALITY, fh->channels)
456                 != SWITCH_STATUS_SUCCESS) {
457                 switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_CRIT, "Unable to create resampler!\n");
458                 return SWITCH_STATUS_GENERR;
459             }
460         }
461         switch_resample_process(fh->resampler, data, (uint32_t) *len);502
462     }
463 }
464 }
```

---

检测文件是否支持视频。

---

```

507
508 SWITCH_DECLARE(switch_bool_t) switch_core_file_has_video(switch_file_handle_t *fh, switch_bool_t check_open)
509 {
510     return ((!check_open || switch_test_flag(fh, SWITCH_FILE_OPEN)) &&
511             switch_test_flag(fh, SWITCH_FILE_FLAG_VIDEO)) ? SWITCH_TRUE : SWITCH_FALSE;
512 }
```

---

向文件中写入。也会根据情况决定是否启动resampler (L530) , 也支持pre\_buffer (预写, L563) , 或直接写入文件 (L587) 。

---

```

513 SWITCH_DECLARE(switch_status_t) switch_core_file_write(switch_file_handle_t *fh, void *data, switch_size_t *len)
514 {
515
516     if (!switch_test_flag(fh, SWITCH_FILE_NATIVE) && fh->native_rate != fh->samplerate) {
517         if (!fh->resampler) {
518             if (switch_resample_create(&fh->resampler, ...
519             )
520         }
521     }
522 }
```

---

---

```

539         switch_resample_process(fh->resampler, data, (uint32_t) * len);
557     }
562
563     if (fh->pre_buffer) {
585     } else {
586         switch_status_t status;
587         if ((status = fh->file_interface->file_write(fh, data, len)) == SWITCH_STATUS_SUCCESS) {
588             fh->samples_out += orig_len;
589         }
590         return status;
591     }
592 }
```

---

写视频数据，调用模块中实际的实现函数向文件中写入视频。

---

```

594 SWITCH_DECLARE(switch_status_t) switch_core_file_write_video(switch_file_handle_t *fh, switch_frame_t *frame)
595 {
607     return fh->file_interface->file_write_video(fh, frame);
609 }
```

---

读视频，返回一个frame，实际的图像存放在frame->img中。

---

```

611 SWITCH_DECLARE(switch_status_t) switch_core_file_read_video(switch_file_handle_t *fh, switch_frame_t *frame, switch_vide...
```

---

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）计算的。

---

```

635 SWITCH_DECLARE(switch_status_t) switch_core_file_seek(switch_file_handle_t *fh, unsigned int *cur_pos, int64_t samples)
636 {
675     status = fh->file_interface->file_seek(fh, cur_pos, samples, whence);
683     return status;
684 }
```

---

设置文件的元数据，如标题、专辑等。

---

```

686 SWITCH_DECLARE(switch_status_t) switch_core_file_set_string(switch_file_handle_t *fh, switch_audio_col_t col, const char *string)
687 {
699     return fh->file_interface->file_set_string(fh, col, string);
700 }

```

---

获取文件元数据。

---

```

702 SWITCH_DECLARE(switch_status_t) switch_core_file_get_string(switch_file_handle_t *fh, switch_audio_col_t col, const char *string)
703 {
715     return fh->file_interface->file_get_string(fh, col, string);
716 }

```

---

清空文件。

---

```

718 SWITCH_DECLARE(switch_status_t) switch_core_file_truncate(switch_file_handle_t *fh, int64_t offset)
719 {
733     if ((status = fh->file_interface->file_truncate(fh, offset)) == SWITCH_STATUS_SUCCESS) {
742     }
746 }

```

---

对打开的文件进行一些操作，如清空预读缓冲区（L760），暂停（SCFC\_PAUSE\_READ）等。

---

```

748 SWITCH_DECLARE(switch_status_t) switch_core_file_command(switch_file_handle_t *fh, switch_file_command_t command)
749 {
759     switch(command) {
760         case SCFC_FLUSH_AUDIO:
761             if (fh->pre_buffer) {
762                 switch_buffer_zero(fh->pre_buffer);
763             }
764             break;
765         default:
766             break;
767     }
768
769     if (fh->file_interface->file_command) {
770         switch_mutex_lock(fh->flag_mutex);
771         status = fh->file_interface->file_command(fh, command);
772         switch_mutex_unlock(fh->flag_mutex);
773     }
776 }

```

---

关闭打开的文件。

---

```
779 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去实现。

# 第五章 代码修炼之道

谈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。

## 5.1 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 OpenShot<sup>1</sup>，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。

### 5.1.1 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：

---

<sup>1</sup><http://openshot.org/>

---

```
git show f31393d
```

---

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，在内存中的表示也是 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 x4”，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 `Ahpha` 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 被替换了成了 L92，它维护了一个 FreeSWITCH 内部图像格式与 FOURCC<sup>2</sup>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;
```

---

<sup>2</sup><http://www.fourcc.org/yuv.php>。

### 5.1.2 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->bgimg);
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->bgimg) {
85                 context->bgimg = 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->bgiimg) {
115         switch_img_patch(frame->img, context->bgiimg, 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");
179         }
180     }
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, 0, flags, &bug)) != 0)
199         switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_ERROR, "Failure!\n");
200     switch_thread_rwlock_unlock(MODULE_INTERFACE->rwlock);
201     return;
202 }
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_")) != NULL) {
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) && strcasecmp(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->rlock);
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
289     usage:
290     stream->write_function(stream, "-USAGE: %s\n", CHROMAKEY_API_SYNTAX);
291
292     done:
293     if (rsession) {
294         switch_core_session_runlock(rsession);
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, 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。

## 5.2 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：

---

```
#applications/mod_video_filter
```

---

`configure.ac`中增加一行：

---

```
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_LT_LIBRARIES = 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`。

### 5.3 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_
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->bcolor);
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。

## 5.4 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](#)，可以移步到这里查看：

<https://freeswitch.org/fisheye/changelog/freeswitch?cs=c60ae0f0e11e761dd43d75bf9979a47721ab1f64>

## 5.5 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 5.1: Ken Rice 的海边演播室

# 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) 资深网络通信技术专家，在网络通信领域耕耘近 15 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，而且客户还遍及美国、印度等海外国家。

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 SelfStore (<https://selfstore.io/~dujinfang>，已关闭) 和小樱桃微信商城上发布。如果您不小心从其它渠道获得本书，请删除您的版本并到微信商城上购买正版。

# 广告

## 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。

**FreeSWITCH 第五届开发者沙龙于 2016 年 8 月 27 日在京举行胜利闭幕**

<http://www.freeswitch.org.cn/2016/08/27/FSCNDS-2016.html>

**FreeSWITCH 培训 2016 夏季班（北京站）于 2016 年 8 月 28-30 日在京举行胜利闭幕**

<http://www.freeswitch.org.cn/2016/08/28/freeswitch-training-2016-bj-happy-ending.html>

**烟台小樱桃网络科技有限公司提供商业 FreeSWITCH 及 OpenSIPS 技术支持**

网址：<http://x-y-t.cn> 邮箱：info@x-y-t.cn

## 烟台小樱桃网络科技有限公司是潮流网络（GrandStream）山东总代理

深圳市潮流网络技术有限公司（Grandstream）是全球知名的统一通讯和整体解决方案厂商和全球 TOP3 VoIP 终端供应商，是国内 IMS、统一通信、呼叫中心、调度市场、视频监控的主要 SIP 终端供货厂商，成功案例包括京东、网易、凡客诚品、平安、中集、东航、国家电网、中石化、中石油、外交部、中移动、中电信等等，公司是中国三大运营商 IMS 市场主要核心合作 SIP 终端厂商，入围中国电信集团 IMS SIP 电话短名单，持续配合运营商研究院 SIP 硬终端的核心业务及协议定制，参与及进入中电信、中移动多个省试点应用和产品供应厂商。在全球与渠道及增值合作伙伴成功服务将近千万终端用户，包括全球主流运营商数十万套终端以及美国共和党和美国民主党数万套 SIP 电话和网关等大型成功部署案例，连续 10 年保持近 50% 年复合增长率。

烟台小樱桃网络科技有限公司，是潮流全线产品在山东省的总代理。

联系方式：邮箱：info@x-y-t.cn 电话 0535-6753997

## FreeSWITCH 相关图书

《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》是一本介绍如果用 Wireshark 分析 SIP/RTP 数据包的书。

《FreeSWITCH 源代码分析》，讲源代码。

《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。



图 5.2: FreeSWITCH 相关图书



图 5.3: FreeSWITCH 权威指南



图 5.4: 小樱桃 (微信扫一扫)

THIS PAGE INTENTIONALLY LEFT BLANK.