



FreeSWITCH

VoIP 实战

杜金房 著

FreeSWITCH: VoIP实战

杜金房 著

FreeSWITCH-CN 内部培训教材

图书不在版编目 (NCIP) 数据

FreeSWITCH: VoIP实战 / 杜金房 著 / 2012.6

FreeSWITCH: VoIP实战

作 者	杜金房
封面设计	布肯尼[美]
校 对	高超
排 版	杜金房
开 本	1890毫米 × 2360 毫米
印 张	7.5
印 数	100
版 数	2012年6月第1版 2013年4月第8次印刷
电子邮箱	book@freeswitch.org.cn

版权所有，侵权必究



图 1: FreeSWITCH-CN第一届开发者沙龙, 北京中关村, 车库咖啡, 2012年6月



图 2: ClueCon2012现场, 美国芝加哥, 2012年8月

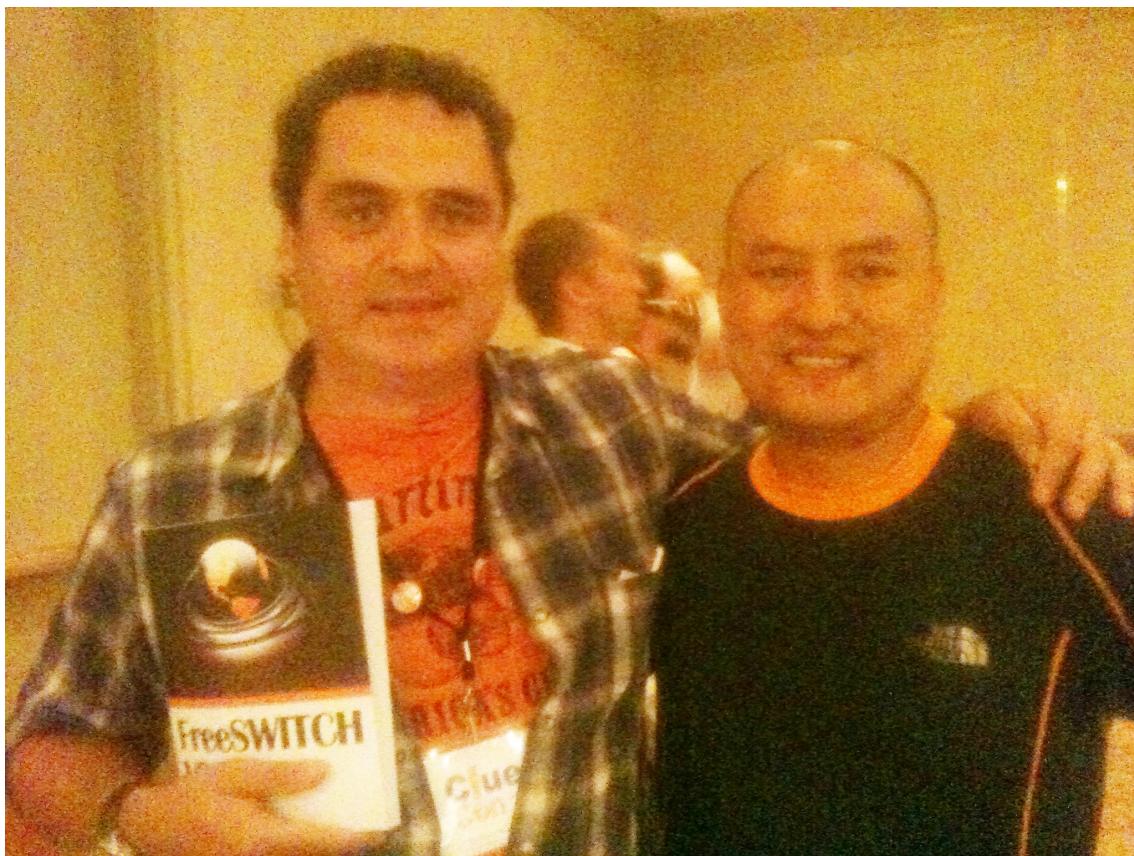


图 3: 本书作者 (右) 与FreeSWITCH作者Anthony Minessale (左) 合影, ClueCon 2012

前　　言

我们已经步入了一个新的时代。当前，VoIP已开始成为语音通信的主导并将在全世界范围内引领一场革命，而SIP（Session Initiation Protocol，会话初始协议）必将是这场革命的核心。

试想一下我们常用的电子邮件，它用于文字通信，经过三十多年的发展，到现在几乎是人人都有一个Email地址了。而在不久的将来，每个人也将拥有一个用于语音及视频通信的SIP地址。随着互联网的高速发展，数据流量的成本会越来越低。而且，随着3G、4G及WiMax无线网络的发展，网络将无处不在，各种新型的SIP电话及可以运行在各种移动设备上的SIP客户端可以让你以极低的成本与世界上任何一个角落的人通信。

为什么写一本书？

“大多数关于操作系统的图书均重理论而轻实践，而本书则在这两者之间进行了较好的折中。” — Andrew S. Tanenbaum

从第一次读Tanenbaum的《操作系统设计与实现》到现在已经好多年了，可这句写在前言里的话到现在我还记忆犹新。在学校里，也曾学习过《程控交换网》、《移动通信》之类的知识，但那时只有肤浅的认识，理解不深。毕业后，我应聘到电信局(这个名字也许太老了。中国电信业在短短的几年内经过了数次重组改制，我离开时叫网通，电信局是我刚参加工作时的名字)工作，负责程控交换机的维护。在工作中我学到了PSTN网络交换的各种技术，掌握了七号信令系统(SS7)，算是做到了理论与实践相结合吧。那时候，VoIP还是很新的东西，由于网络条件的限制，国内也少有人用。2007年底，我开始接触Asterisk。阅读了《Asterisk，电话未来之路》，并买了一个单口的语音卡，实现了VoiceMail，PSTN网关，SIP中继等各种功能。能在自己电脑上实现这些有趣的东西，令我感到非常兴奋。后来，我加入Idapted Inc.，做一种一对一的网络教学平台。最初的后台语音系统也使用Asterisk，但不久后便转到FreeSWITCH。虽然当时FreeSWITCH还是不到1.0的Beta版，但已经显出了比Asterisk高几倍的性能，并且相当稳定。

FreeSWITCH的主要作者Anthony Minessal曾有多年的Asterisk开发经验，后来由于他提的一些设想未得到社区管理者的支持，便独立开发了FreeSWITCH，并以开源软件发布。FreeSWITCH主要使用C、C++开发。为避免“重复发明轮子”，它使用了大量的成熟的第三方软件库。FreeSWITCH功能丰富，可伸缩性强，并可以使用Lua、Javascript、Perl等多种嵌入式语言控制呼叫流程。除此之外，它还提供Event Socket接口，可以使用任何语言进行二次开发或

与其它系统进行集成。最重要的是，它有一个非常友好、活跃的社区支持。我不止一次遇到这样的情况—某天想到一项新功能，可能过几天就被实现了；发现一个Bug，提交给开发者，一觉醒来就修好了（我睡觉的时候正好美国是白天）。FreeSWITCH是极少数的git master分支代码比最新的发行版更稳定的项目之一。而与此相对的很多商业系统如果发现问题却常常需要很长的修复周期。

当然，我们在使用过程中也遇到不少问题，除了向开发者报告漏洞（BUG）外，我们也提交一些补丁（Patch），这不仅能在一定程度上能让FreeSWITCH按我们期望的方式工作，而且，也可以为开源事业做一点点贡献，从而也可以获得一些成就感。而这也正是我们最喜欢开源软件的原因。

FreeSWITCH的文档¹非常丰富，它采用wiki系统，内容都是来自众多FreeSWITCH爱好者和实践者的奉献。不过，对于初学者来说，查阅起来还不是很方便。因此freeswitch-users邮件列表中也多次有人提到希望能有一本能系统地介绍FreeSWITCH的书。FreeSWITCH官方后来响应了这个写书的计划，但是进展不快。

FreeSWITCH在美国及欧洲其它国家已有很多的应用，但国内的用户还很少。2009年下半年，我创办了FreeSWITCH-CN²，希望能跟更多说中文的朋友一起学习和交流。我曾经设想能找一些志同道合者把所有wiki资料都翻译成中文的。但由于种种原因一直未能实现。随着中文社区的日益发展壮大，越来越多的人向我提问问题，而我也没有太多的时间一一作答。与此同时，我在学习和使用的过程中积累了好多经验，因此，便有了自己写一本书的计划。当时想，如果能在官方的书出版之前写出一些东西，该有多酷啊。

我相信我这么做不是在重复发明轮子。因为我发现，好多人问问题时，并不是因为不懂FreeSWITCH，而是对一些基本理论、概念或逻辑理解不是很清楚。当然，我不会像教科书上那样照本宣科的讲理论，事实上，我也讲不了。我只是希望能结合多年的工作经验，用一些比较通俗的语言讲一些解决实际问题的思路，让与我遇到同样问题的朋友少走弯路。

后来，FreeSWITCH官方曾在2010年和2012年分别出版了《FreeSWITCH 1.0.6》以及《FreeSWITCH Cookbook》，而我的进度却一直迟迟不前。我的书不仅没赶在官方第一本书前面，也没赶在第二本前面。这都是后话。

章节与内容安排

以什么风格来写呢？曾听人说过，“写作的难处不是考虑该写些什么，而是需要决定什么不应该写进书里。”我对此深有感触。FreeSWITCH官方Wiki上有几百页的资料，该从何写起呢？如果只是盲目照抄的话，只不过是相当于做了些翻译工作，也没什么意思；如果只是将一些功能及参数机械地罗列出来，那也不过相当于一个中文版的Wiki。所以，我最后决定写成一个由浅入深步步推进的教程，并搭配一些我在工作中用到的实际的例子。

在最初几章，我介绍了一些基本概念及背景知识，这主要是给没有电信背景的人看的，另外，对从电路交换转到VoIP来的读者也会很有帮助。这些内容是不能舍弃的。

¹<http://wiki.freeswitch.org>

²<http://www.freeswitch.org.cn>

接下来应该是安装和配置。笔者看到有不少的图书，在讲一个软件时，将整个的安装过程都会用图一步一步的列出来。比如有些讲Linux的书，甚至从如何安装Linux起，所有的步骤都抓了图。窃以为那真是太没必要，事实上，这几年在FreeSWITCH邮件列表中看到大家问得比较多的问题是“我装上了 FreeSWITCH，该怎么用啊？”，而不是“谁能告诉我怎么安装FreeSWITCH啊？”所以，如何取舍就显而易见了。本书仅在第二章中提到了如何安装，或许以后如果觉得不够，可以加一个附录，但绝对不会把如何安装FreeSWITCH单独作为一章。

实战部分，则以实际的例子讲配置，穿插讲解或复习一些基本概念。如果有需要罗列的命令参考，则在附录中给出。

附录也很重要。除重要的参考资料，背景知识等，还收集了一些我所知道的奇闻轶事。

另外，FreeSWITCH一直处于很活跃的开发中，所以，某些章节可能刚写完就过时了，最新、最权威的参考还是官方的Wiki。但是，无论如何，本书所阐述的基本架构、理念，尤其是历史永远不会过时。

鉴于本书的内容安排，本书适合顺序阅读（唯一例外的就是—你可以先读后记部分）。

谁适合阅读本书？

- 学生：我看过去一些学校的教材，大部分只是讲VoIP原理及SIP协议等，很枯燥；
- 教师：显而易见，老师总是教育学生“要理论与实践相结合”；
- FreeSWITCH初学者：本书肯定对你有帮助；
- FreeSWITCH高级用户、开发人员：如果你喜欢FreeSWITCH，也一定会喜欢这本书；
- VoIP爱好者、开发人员：他山之石，可以攻玉。即使你不使用FreeSWITCH，本书也会对你有帮助；
- 电信企业的维护人员、销售人员、决策人员：相信本书能使你了解新技术，更了解客户需求，以及如何才能为客户提供更好的服务；
- 其它企业管理人员：如果你知道电信业务其实还可以提供许多你所不知道的功能和业务，你肯定能很好地加以利用，带来的是效率、节省的是成本；
- 其它人员：开卷有益，而且，你会对你天天离不开的电话、手机，以及新兴的网络电话、即时通讯工具等有一个更好的了解，进而增加工作效率。

排版约定

- 本书使用Latex³排版。

³<http://zh.wikipedia.org/zh/LaTeX>

- 部分插图由yed、XMind、keynote等生成，呼叫流程图就直接使用了纯文本。
- 程序代码、系统的输入、输出等均使用等宽字体，大部分有行号。强调部分使用粗体字，某些英文专有名词、特殊符号等则使用斜体字。
- 有些程序代码行较长，为适应版面，进行了人工换行，一般都会标有“人工换行”标记。有些无关紧要的代码则任其超出版面。
- 本书中所称Wiki一般指 wiki.freeswitch.org，除非有特别说明。

为方便读者，书中术语首次出现时尽量给出英文及中文全称，如：SIP（Session Initiation Protocol，会话初始协议）。

由于水平所限，存在错误和疏漏在所难免，欢迎广大读者朋友批评指正。

本书现在只是一个草稿，会不定期做改动，即使大的改动也不一定发布通知。

版权声明

本书版权归作者杜金房⁴所有，保留所有权利。

致谢

FreeSWITCH™是<http://www.freeswitch.org>的注册商标。感谢Anthony Minessal及他的团队给我们提供了如此优秀的软件；同时感谢FreeSWITCH社区所有成员的热心帮助。本书的一些资料和例子来自FreeSWITCHWiki及[邮件列表](#)，不能一一查证原作者，在此一并致谢。

感谢我的妻子吕佳婷，她不仅帮我修订文字错误，还时常在我埋头写作时在我案头放一个削好的苹果。感谢Jonathan Palley先生，是他指导我走上了FreeSWITCH之路。感谢崔钢先生跟我一起创业，他管理公司的方方面面，使我有更多的时间专注于FreeSWITCH。感谢高超，他校对了本书前面一些章节。感谢Kenny Bloom，他帮我设计了本书封面。感谢程祝波，他维护着FreeSWITCH-CN QQ群，给大家提供一个实时交流的平台。感谢Tim Yang先生，他提供了本书写作以来的第一笔捐赠。感谢网友flyingnn及其它热心读者，他们对本书给予了积极的反馈并帮助我刊误。没有他们，便没有此书。

杜金房
2012年6月于北京和园国际青年旅舍

⁴<http://www.dujinfang.com/>

目 录

1 PSTN与VoIP	19
1.1 PSTN起源	19
1.2 模拟与数字信号	21
1.3 PCM	22
1.4 我国电话网结构	22
1.5 时分复用与局间中继	23
1.5.1 时分复用	23
1.5.2 局间中继	23
1.6 信令	23
1.6.1 信令分类	23
1.6.2 用户线信令	24
1.6.3 局间信令	25
1.6.4 七号信令	25
1.7 电路交换与分组交换	27
1.7.1 电路交换	27
1.7.2 分组交换	28
1.8 VoIP	29
2 初识FreeSWITCH	31
2.1 什么是 FreeSWITCH ?	31
2.2 快速体验	31

2.3 安装FreeSWITCH基本系统	32
2.3.1 最快安装（推荐）	33
2.3.2 从 Git 仓库安装:	33
2.3.3 解压缩源码包安装:	33
2.4 安装声音文件	33
2.4.1 连接SIP软电话	36
2.5 配置简介	38
2.6 添加一个新的SIP用户	39
2.7 FreeSWITCH用作软电话	40
2.8 配置SIP网关拨打外部电话	42
2.8.1 从某一分机上呼出	42
2.8.2 呼入电话处理。	43
2.9 小结	43
3 PSTN与PBX 业务	45
3.1 PSTN 业务	45
3.1.1 POTS	45
3.1.2 商务业务	46
3.1.3 其它增值业务	47
3.2 PBX 业务	47
4 SIP协议	51
4.1 SIP的概念和相关元素	51
4.2 SIP 注册	53
4.3 SIP 呼叫流程	57
4.3.1 UA 间直接呼叫	57
4.3.2 通过 B2BUA 呼叫	60
4.4 再论 SIP URI	70

5 FreeSWITCH架构	73
5.1 总体结构	73
5.2 核心	73
5.3 数据库	73
5.4 模块	75
5.4.1 终点	75
5.4.2 拨号计划	75
5.4.3 聊天计划	75
5.4.4 应用程序	76
5.4.5 应用程序接口 (FSAPI)	76
5.4.6 XML 接口	76
5.4.7 编解码器	76
5.4.8 语音识别及语音合成	76
5.4.9 文件格式	76
5.4.10 日志	76
5.4.11 嵌入式语言	77
5.4.12 事件套接字	77
5.5 目录结构	77
5.6 配置文件	77
5.6.1 目录结构	78
5.6.2 freeswitch.xml	79
5.6.3 vars.xml	80
5.6.4 autoload_configs 目录	80
5.6.5 dialplan 目录	80
5.6.6 directory 目录	80
5.6.7 sip_profiles	80
5.7 XML 用户目录	80
5.8 呼叫流程及相关概念	82
5.8.1 Session 与 Channel	84
5.8.2 回铃音与 Early Media	84

5.8.3 Channel Variable	84
5.9 小结	86
6 运行FreeSWITCH	87
6.1 命令行参数	87
6.2 系统启动脚本	88
6.3 如何判断 FreeSWITCH 已经运行?	89
6.4 控制台与命令客户端	89
6.5 发起呼叫	91
6.5.1 呼叫字符串	92
6.5.2 API 与 App	92
6.6 命令行帮助	94
6.7 小结	95
7 SIP 模块 — mod_sofia	97
7.1 配置文件	97
7.1.1 internal.xml	98
7.1.2 external.xml	109
7.1.3 gateway	110
8 认识拨号计划	113
8.1 XML Dialplan	113
8.1.1 配置文件的结构	113
8.1.2 默认的配置文件结构	117
8.1.3 正则表达式	117
8.1.4 信道变量 - Channel Variables	118
8.1.5 测试条件 - Conditions	121
8.1.6 动作与反动作 - Action & Anti-Action	125
8.1.7 工作机制深入剖析	125
8.1.8 内连执行 - inline 参数	128
8.1.9 实例解析	129
8.2 内连拨号计划 - Inline Dialplan	133

9 嵌入式脚本	137
9.1 什么是嵌入式脚本?	137
9.2 应用场景	138
9.3 Lua	138
9.3.1 语法简介	138
9.3.2 将电话路由到 Lua 脚本	139
9.3.3 Session 相关函数	139
9.3.4 非 Session 函数	140
9.3.5 独立的 Lua 脚本	140
9.3.6 数据库	140
9.4 Javascript	141
9.5 其它脚本语言	141
10 Event Socket	143
10.1 架构	143
10.1.1 外连模式	143
10.1.2 内连模式	144
10.1.3 模式小结	145
10.2 Event Socket 协议	145
10.2.1 外连	145
10.2.2 内连	147
10.3 Event Socket 库	148
10.3.1 Event Socket 的例子	148
11 FreeSWITCH实战	155
11.1 用FreeSWITCH实现IVR	155
11.1.1 最简单的菜单	155
11.1.2 默认菜单简介	157
11.1.3 调试	157
11.2 在FreeSWITCH中执行长期运行的嵌入式脚本—Lua语言例子	158
11.3 一个在FreeSWITCH中外呼的脚本	159

11.4 也谈 FreeSWITCH 中语音识别	162
11.4.1 样本	162
11.4.2 第一种方案, 关键词	163
11.4.3 第二种方案, 连续识别	163
11.4.4 第三种方案, 只录音, 采用外部程序识别	163
11.4.5 第四种方案, Google Voice	164
11.4.6 结论	165
11.5 在 FreeSWITCH 中使用 google translate 进行文本语音转换	165
11.6 FreeSWITCH 与 H323	166
11.7 视频会议	167
11.7.1 普通视频会议	167
11.7.2 多画面融屏	167
11.8 多台 FreeSWITCH 服务器级联	169
11.8.1 双机级联	169
11.8.2 汇接模式	170
11.8.3 安全性	170
11.9 万能 FreeSWITCH directory 脚本	171
11.9.1 脚本	171
11.9.2 调试	173
11.10 使用 Erlang 控制呼叫流程	173
11.10.1 概述	173
11.10.2 安装 Erlang 模块	174
11.10.3 把电话控制权转给 Erlang	174
11.10.4 把电话发给 Erlang	175
11.10.5 呼叫控制	177
11.10.6 使用状态机实现	178
11.11 呼叫是怎样工作的?	182
11.12 在呼叫失败的情况下向主叫用户播放语音提示	184
11.13 在同一台服务器上启动多个 FreeSWITCH 实例	186
11.13.1 背景故事	186

11.13.2 练习	187
11.13.3 进阶	188
11.13.4 让两个实例相互通信?	188
11.13.5 小结	188
12 附录	189
12.1 FreeSWITCH背后的故事(译)	189
12.2 FreeSWITCH 与 Asterisk 比较	191
12.3 FreeSWITCH中文FAQ	196
12.3.1 基本问题	196
12.3.2 获取帮助	199
12.3.3 调试与排错	199
12.3.4 后台运行	201
12.3.5 硬件兼容性	202
12.3.6 编译	203
12.3.7 SIP	204
12.3.8 IAX2	205
12.3.9 程序	205
12.3.10 呼叫路由	206
12.3.11 配置 FreeSWITCH	207
12.4 Idapted的FreeSWITCH实践	207
13 后记	211

第一章 PSTN与VoIP

说起VoIP，也许大家对“网络电话”这个词更熟悉一些。其英文原意是Voice Over IP，即承载于IP网上的语音通信。大家熟悉家庭用来上网的ADSL吧，也许有些人还记得前些年用过的吱吱叫的老“猫”。技术日新月异，前面的技术都是用电话线上网，现在，VoIP技术使我们可以在网上打电话，生活就是这样。

所谓温故而知新，在学习任何新东西以前，我们都最好了解一下其历史，以做到心中有数。在了解VoIP之前，我们需要先看一下PSTN，那么，在PSTN之前呢？

1.1 PSTN起源

PSTN（Public Switched Telephone Network）的全称是公共交换电话网，就是我们现在打电话所使用的电话网络。

第一次语音传输是苏格兰人亚历山大·贝尔(Alexander Granham Bell)在1876年用振铃电路实现的。在那之前，普遍认为烽火台是最早的远程通信方式。其实烽火台不仅具备通信的完整要素(通信双方，通信线路及中继器)，而且还属于无线通信呢。当时没有电话号码，相互通话的用户之间必须有物理线路连接；并且，在同一时间只能有一个用户可以讲话(半双工)。发话方通过话音的振动激励电炭精麦克风而转换成电信号，电信号传到远端后通过振动对方的扬声器发声，从而传到对方的耳朵里。

由于每对通话的个体之间都需要单独的物理线路，如果整个电话网上有10个人，而你想要与另外9个人通话，你家就需要铺设9对电话线。同时整个电话网上就需要 $10 \times (10-1)/2 = 45$ 对电话线。

当电话用户数量增加的时候，为每对通话的家庭之间铺设电话线是不可能的。因此一种称为交换机（Switch）的设备诞生了。它位于整个电话网的中心，用于连接每个用户。用户想打电话时，先拿起电话连接到管理交换机的接线员，由接线员负责接通到对方的线路。这便是最早的电话交换网。

随着技术的进步，电子交换机替代了人工交换机，便出现了现代意义的PSTN。随着通信网络的进一步扩大，便出现了许许多多的交换机。交换机间通过中继线（Trunk）相连。有时一个用户与另一个用户通话需要穿越多台交换机。

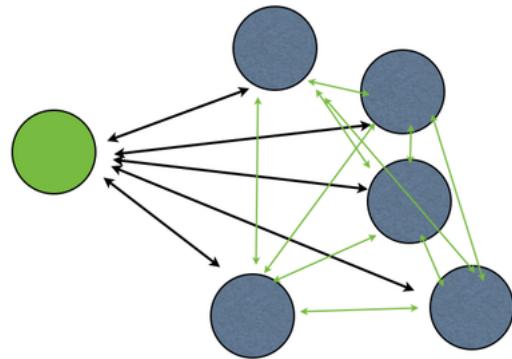


图 1.1: 随着用户的增多，需要的电话线越来越多

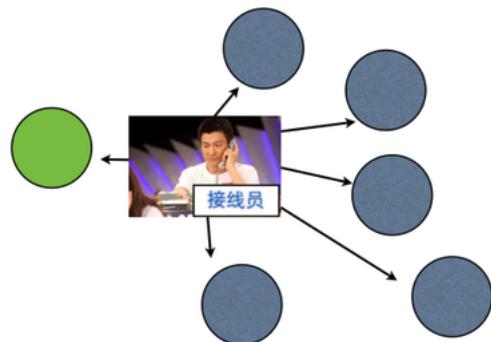


图 1.2: 电话交换网、接线员

后来出现了移动电话（当移动电话小到可以拿在手里的时候就开始叫“手机”），专门用于对移动电话进行交换的通信网络称移动网，而原来的程控交换网则叫固定电话网，简称固网。简单来说，移动网就是在普通固网的基础上增加了许多基站（Base Station，可以简单理解为天线），并增加了归属位置寄存器（HLR，Home Location Register）和拜访位置寄存器（VLR，Visitor Location Register），以记录用户的位置（在哪个天线的覆盖范围内）、支持异地漫游等。移动交换中心称之为MSC（Mobile Switch Center）。

随着下一代网络（NGN）技术的成熟及下一代移动通信时代的到来，大众对多媒体业务的需求越来越强烈。因而运营商又部署了IMS系统（IP Multimedia Subsystem）。它运行于标准的IP网络上，使用一种基于第三方伙伴计划（3GPP）的SIP标准的VoIP实现方式。IMS的目标不仅是在现网基础上提供新的业务，而且它还要能提供现在以及未来因特网上能够承载的所有的业务。现代最新的无线通信标准是LTE（Long Term Evolution，长期演进），为现代的手机及其它移动设备提供高速的数据通信手段。

1.2 模拟与数字信号

现实生活中的一切都是模拟的。模拟量（Analog）是连续的变化的，如温度、声音等。早期的电话网是基于模拟交换的。模拟信号对于人类交流来讲非常理想，但它很容易引入噪声。如果通话双方距离很远的话，由于信号的衰减，需要对信号进行放大。问题是信号中经常混入线路的噪音，放大信号的同时也放大了噪音，导致信噪比（信号量与噪声的比例）下降，严重时会难以分辨。

数字（Digital）信号是不连续的（离散的）。它是按一定的时间间隔（单位时间内抽样的次数称为频率）对模拟信号进行抽样得出的一些离散值。然后通过量化和编码过程将这些离散值变成数字信号。根据抽样定理¹，当抽样频率是最高模拟信号频率的两倍时，就能够完全还原原来的模拟信号。

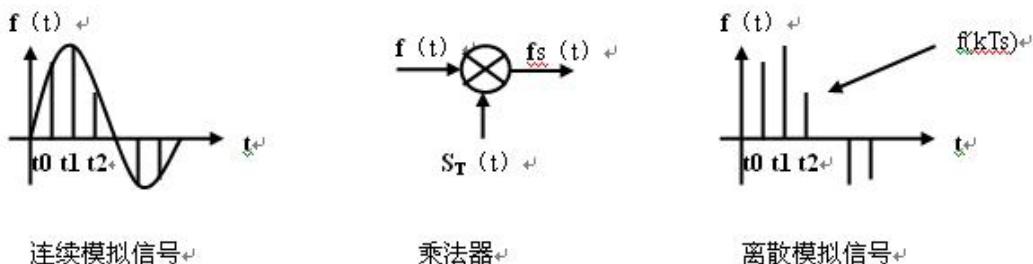


图 1.3: 抽样

¹又称采样定理或奈奎斯特·香农定理，见 <http://zh.wikipedia.org/wiki/采样定理>。

1.3 PCM

PCM (Pulse Code Modulation) 的全称是脉冲编码调制。它是一种通用的将模拟信号转换成以0和1表示的数字信号的方法。

一般来说，人的声音频率范围在 $300Hz \sim 3400Hz$ 之间，通过滤波器对超过 $4000Hz$ 的频率过滤出去，便得到 $4000Hz$ 内的模拟信号。然后根据抽样定理，使用 $8000Hz$ 进行抽样，便得到离散的数字信号。使用PCM方法得到的数字信号就称为PCM信号，一般一次抽样会得到16bit的信息。

为了更有效地在线路上进行传输，通常对PCM信号进行一定的压缩。通过使用压缩算法（实际为压扩法，因为有的部分压缩有的是扩张的。目的是给小信号更多的比特位数以提高语音质量），可以将每一个抽样值压缩到8个比特。这样就得到 $8 \times 8000 = 64000bit$ 的信号。通常我们就简称为 $64kbit/s$ （注意，通常来说，对于二进制数， $1kbit=1024bit$ ，但此处的 $k=1000$ ）。

PCM通常有两种压缩方式：A律和 μ 律。其中北美使用 μ 律，我国和欧洲使用A律。这两种压缩方法很相似，都采用8bit的编码获得12bit到13bit的语音质量。但在低信噪比的情况下， μ 律比A律略好。A律也用于国际通信，因此，凡是涉及到A律和 μ 律转换的情况，都由使用 μ 律的国家负责。

1.4 我国电话网结构

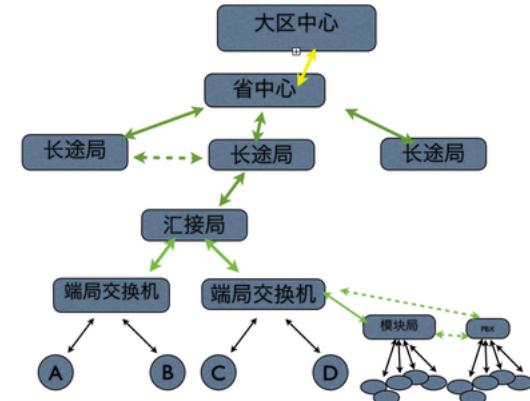


图 1.4: 我国电话网结构

图中主体部分为一地市级电话网的结构。通常，话机（如c）通过一对电话线连接到距离最近的交换机上，该交换机称为端局交换机（一般以区或县为单位）。端局交换机通过局间中继线连接到汇接局。为了保证安全，汇接局通常会成对出现，平常实行负荷分担，一台汇接局出现故障时与之配对的汇接局承担所有话务。长途电话需要通过长途局与其它长途局相连。但根据话务量

要求，汇接局也可以直接与其它长途局开通高速直达中继。为节省用户线，在一些人口比较集中的地方（如学校、小区），端局下会再设模块局或接入网，用户则就近接入到模块局上。

智能网一般用于实现电话卡、预付费或400/800类业务，前几年新布署的NGN（Next Generation Network，下一代网络，一般指软交换）则支持更灵活、更复杂的业务。

但是技术发展很快，响应国家“光进铜退”²的号召，直接光纤到户，就全网都能IP化了。新技术代替旧技术，给用户带来了五彩斑斓的业务，但老旧的技术永远有让人值得怀念的地方，那就是原来的电话线是带电的，家里停电不影响打电话。但光纤是不能馈电的，因而没有了铜线，停电以后用户就不能打电话了（除非加UPS）。

1.5 时分复用与局间中继

1.5.1 时分复用

通过将多个信道以时分复用的方式合并到一条电路上，可以减少局间中继线的数量。通过将32个64k的信道利用时分复用合并到一条2M（ $64k \times 32 = 2.048M$ ，通俗来说就直接叫一个2M）电路上，称为一个E1（在北美和日本，是24个64k复用，称为T1，速率是1.544M）。在E1中，每一个信道称作一个时隙。其中，除0时隙固定传同步时钟，其它31个时隙最多可以同时支持31路电话（有时候会使用第16时隙传送信令，这时最多支持30路电话）。

1.5.2 局间中继

这些连接交换机(局)的2M电路就称为局间中继。随着话务量的增加，交换机之间的电路越开越多，目前通常的做法是将63个2M合并到一个155M（ $2 \times 63 + P = 155$ ，其中P是电路复用的开销）的光路（光纤）上，在SDH（Synchronous Digital Hierarchy，同步数字传输体系）技术中称为STM-1（Synchronous Transfer Module，同步传输模块）。

1.6 信令

用户设备（如话机）与端局交换机之间，以及交换机与交换机之间需要进行通信。这些通信所包含的信息包括（但不限于）用户、中继线状态，主、被叫号码，中继路由的选择等。我们把这些消息称为信令（Signaling）。

1.6.1 信令分类

信令主要有以下几种分类方式：

² 用户家里的电话线使用铜双绞线。

按信令的功能分

- 线路信令：具有监视功能，用来监视主被叫的摘、挂机状态及设备忙闲。
- 路由信令：具有选择功能，指主叫所拨的被叫号码，用来选择路由。
- 管理信令：具有操作功能，用于电话网的管理和维护。

按信令的工作区域分

- 用户线信令：是用户终端与交换机之间的信令。它包括用户状态（摘、挂机）信号及用户拨号（脉冲、DTMF）所产生的数字信号，以及交换机向用户终端发送的信号（铃流、信号音）。
- 局间信令：是交换机和交换机之间的信令，在局间中继线上传送，用来控制呼叫接续和拆线。

用户线信令少而简单，中继线信令多而复杂。

按信令的信道分

- 随路信令：信令和话音在同一条话路中传送的信令方式。
- 公共信道信令：是以时分方式在一条高速数据链路上传送一群话路的信令的信令方式。

随路信令信令传送速度慢，信息容量有限（传递与呼叫无关的信令能力有限）。而公共信令传送速度快、容量大、具有改变或增加信令的灵活性，便于开放新业务。

其它分类

其它的分类方式有带内信令与带外信令、模拟信令和数字信令、前向信令和后向信令、线路信令和记发器信令等，我们在这里就不多解释了。有兴趣的读者可以自行搜索相关的关键词进一步学习。

1.6.2 用户线信令

用户线信令是从用户终端（通常是话机）到端局交换机之间传送的信令。对于普通的话机，线路上传送的是模拟信号，信令只能在电话线路上传送，这种信令称为带内信令。话机通过电压变化来传递摘、挂机信号；通过DTMF（Dual Tone Multi Frequency，双音多频。话机上每个数字或字母都可以发送一个低频和一个高频信号相结合的正弦波，交换机经过解码即可知道对应的话机按键）传送要拨打的电话号码。另外，也可以通过移频键控（FSK，Frequency Shift-keying）

技术来支持“主叫号码显示”，俗称“来电显示”（Caller ID或CLIP，Caller Line Identification Presentation，主叫线路识别提示）。

与普通电话不同，ISDN（Integrated Service Digital Network，综合业务数字网）在用户线上传送的是数字信号。它的基本速率接口（BRI，Base Rate Interface）使用144k的2B+D信道—两个64k的B信道及一个16k的D信道。其中B信道一般用来传输语音、数据和图像，D信道用来传输信令或分组信息。

2B+D的ISDN最初为了解决用户线上的语音与数据同步传输问题，野心勃勃。但事实上，2B+D的ISDN并不像传说中的那么美，而且需要专门的NT1终端设备，在我国并没有发挥出它应有的作用，后来很快被ADSL技术取代了。

1.6.3 局间信令

局间信令主要在局间中继上传送。一般一条信令链路通常只占用一个64k的时隙。一条信令消息通常只有几十或上百个字节，一条64k的电路足以容纳成千上万路电话所需要的信令。但随着技术的进步，话务量的上涨以及更多增值业务的出现，完成一次通话需要更多的信令消息，因此出现了2M速率的信令链路，即整个E1链路上全部传送信令。

目前常见的局间信令有ISDN PRI(Primary Rate Interface，基群速率接口)信令和七号信令。PRI是在跟话路同一个E1上传送的，通常使用第16时隙，而0时隙传同步信号，其它30个时隙可传输通话信息，因此又称为30B+D（与上面的2B+D相对）。

与PRI信令不同，七号信令除可以与话路在同一个E1上传送外，还可以在专门的用于传送信令链路的E1中继上传送的，因而它组网更加灵活，支持更大的话务量。我国的电话网络中有专门的七号信令网。

但支持七号信令的每个通信设备都需要有一个全局唯一信令点编码，而信令点编码资源是比较有限的。因而，七号信令主要在运营商的设备上使用，而运营商与用户设备（如PBX）一般使用PRI信令对接。

1.6.4 七号信令

七号信令（SS7，Signaling System No. 7）是目前我国使用的最主要的信令方式，用于局间通信。

```

1 用户a          A交换机          B交换机          用户b
2 |             |             |             |
3 | 摘机          |             |             |
4 |----->|     |             |             |
5 | 拨号音        |             |             |
6 |-----<----|     |             |             |
7 | 拨号          IAM           |             |
8 |----->|----->|     |             |
9 | 回铃音        ACM          振铃          |
10 |-----<----|-----<----|----->|
11 | 通话          ANC          接听          |
12 |-----<----|-----<----|----->|
13 | <===== 通    话 =====> |             |
14
15 -----
16
17 | 主叫挂机      CLF          送催挂音
18 |----->|----->|----->|
19 |          RLG          |
20 |-----<----|             |
21
22 -----
23
24 | <===== 通    话 =====> |
25 | 送催挂音      CBK          被叫挂机
26 |-----<----|-----<----|----->|
27 |          CLF          |
28 |----->|             |
29 |          RLG          |
30 |-----<----|             |

```

我们来看一次简单的固定电话的通话流程。如上图。用户a摘机，与其相连的A交换机根据电压、电流的变化检测到A摘机后，即送拨号音，同时启动收号程序。a开始拨号，待A交换机号码收齐后，即查找路由，发送IAM（Initial Address Message，初始地址消息）给B交换机。B向A发ACM（Adress Complete Message，地址全消息）并通知话机b振铃，A向a送回铃音。这时如果b接听电话，则B向A发送ANC（Answer Charge，应答计费消息），a与b开始通话，同时A对a计费³。

通话完毕，如果主叫挂机，则本端交换机A向对端B发送CLF（前向释放消息），B向A回RLG（Release Gard，释放监护消息），并向b送催挂音（嘟嘟嘟...）。

如果被叫挂机，则B向A发送CBK（Clear Backword，后向释放消息），A回送CLF，最后B回RLG。

上面在交换机A与B之间传递的为七号信令中的TUP（Telephone User Part，电话用户部分）部分。目前，由于ISUP（ISDN User Part，ISDN用户部分）能与ISDN互联并提供比TUP更多的

³B也可以对A计费，称为中继计费，主要用于局间结算（话费）。如果A和B分属于不同的运营商（如移动和电信），则称为网间结算。

能力和服务，已基本取代TUP而成为我国七号信令网上主要的信令方式。ISUP信令与TUP互通时的对应关系如下：

```

1   端局A          汇接局TM          端局B
2   | ISUP           | TUP           |
3   |               |               |
4   | IAM(IAI)       | IAM           |
5   |----->|----->|
6   | ACM           | ACM           |
7   |<-----|<-----|
8   | ANM           | ANC           |
9   |<-----|<-----|
10  | <===== 通    话 =====> |
11
12 -----
13
14  | REL            | CLF           | <-- 主叫挂机
15  |----->|<-----|
16  | RLC            | RLG           |
17  |<-----|-----|
18
19 -----
20
21  | <===== 通    话 =====> |
22  | REL            CBK           | <-- 被叫挂机
23  |<-----|<-----|
24  | RLC            CLF           |
25  |----->|----->|
26  |           | RLG           |
27  |           |<-----|

```

以上为端局A与端局B经过汇接局TM汇接通信中ISUP与TUP信令转换的例子。ISUP信令的初始地址消息有IAM和IAI (IAM With Additional Information, 带附加信息的IAM) 两种，后者能提供更多的信息（如主叫号码等）。另外ISUP信令的拆线信号不分前向后向，只有REL (Release, 翻译) 和RLC (Release Complete, 释放完成)。

1.7 电路交换与分组交换

1.7.1 电路交换

传统的电话都是基于电路交换。由于电路交换在通信之前要在通信双方之间建立一条被双方独占的物理通路（由通信双方之间的交换设备和链路逐段连接而成），因而有以下优缺点。

优点：

- 由于通信线路为通信双方用户专用，数据直达，所以传输数据的时延非常小。
- 通信双方之间的物理通路一旦建立，双方可以随时通信，实时性强。
- 双方通信时按发送顺序传送数据，不存在失序问题。
- 电路交换既适用于传输模拟信号，也适用于传输数字信号。
- 电路交换的交换的交换设备（交换机等）及控制均较简单。

缺点：

- 电路交换的平均连接建立时间对计算机通信来说较长。
- 电路交换连接建立后，物理通路被通信双方独占，即使通信线路空闲，也不能供其他用户使用，因而信道利用率低。
- 电路交换时，数据直达，不同类型、不同规格、不同速率的终端很难相互进行通信，也难以在通信过程中进行差错控制。

1.7.2 分组交换

我们熟悉的IP交换就采用分组交换的方式。它仍采用存储转发传输方式，但将一个长报文先分割为若干个较短的分组，然后把这些分组（携带源、目的地址和编号信息）逐个地发送出去，因此分组交换除了具有报文的优点外，与报文交换相比有以下优缺点：

优点：

- 加速了数据在网络中的传输。因为分组是逐个传输，可以使后一个分组的存储操作与前一个分组的转发操作并行，这种流水线式传输方式减少了报文的传输时间。此外，传输一个分组所需的缓冲区比传输一份报文所需的缓冲区小得多，这样因缓冲区不足而等待发送的机率及等待的时间也必然少得多。
- 简化了存储管理。因为分组的长度固定，相应的缓冲区的大小也固定，在交换结点中存储器的管理通常被简化为对缓冲区的管理，相对比较容易。
- 减少了出错机率和重发数据量。因为分组较短，其出错机率必然减少，每次重发的数据量也就大大减少，这样不仅提高了可靠性，也减少了传输时延。
- 由于分组短小，更适用于采用优先级策略，便于及时传送一些紧急数据，因此对于计算机之间的突发式的数据通信，分组交换显然更为合适些。

缺点：

- 尽管分组交换比报文交换的传输时延少，但仍存在存储转发时延，而且其结点交换机必须具有更强的处理能力。

- 分组交换与报文交换一样，每个分组都要加上源、目的地址和分组编号等信息，使传送的信息量大约增大5%~10%，一定程度上降低了通信效率，增加了处理的时间，使控制复杂，时延增加。
- 当分组交换采用数据报服务时，可能出现失序、丢失或重复分组，分组到达目的结点时，要对分组按编号进行排序等工作，增加了麻烦。若采用虚电路服务，虽无失序问题，但有呼叫建立、数据传输和虚电路释放三个过程。

总之，若要传送的数据量很大，且其传送时间远大于呼叫时间，则采用电路交换较为合适；当端到端的通路有很多段的链路组成时，采用分组交换传送数据较为合适。从提高整个网络的信道利用率上看，报文交换和分组交换优于电路交换，其中分组交换比报文交换的时延小，尤其适合于计算机之间的突发式的数据通信。

1.8 VoIP

维基百科上是这样说的：

IP电话（简称VoIP，源自英语Voice over Internet Protocol；又名宽带电话或网络电话）是一种透过互联网或其他使用IP技术的网络，来实现新型的电话通讯。过去IP电话主要应用在大型公司的内联网内，技术人员可以复用同一个网络提供数据及语音服务，除了简化管理，更可提高生产力。随着互联网日渐普及，以及跨境通讯数量大幅飙升，IP电话亦被应用在长途电话业务上。由于世界各主要大城市的通信公司竞争日剧，以及各国电信相关法令松绑，IP电话也开始应用于固网通信，其低通话成本、低建设成本、易扩充性及日渐优良化的通话质量等主要特点，被目前国际电信企业看成是传统电信业务的有力竞争者。详细内容参见维基百科上的[IP电话⁴](#)。

目前，VoIP呼叫控制协议主要有SIP、H323，以及MGCP与H.248/MEGACO等。H323是由ITU-T（国际电信联盟）定义的多媒体信息如何在分组交换网络上承载的建议书。它是一个相当复杂的协议，使用起来很不灵活。而SIP则是IETF（互联网工程任务组）开发的（RFC3261），它是一种类似HTTP的基于文本的协议，很容易实现和扩展，被普遍认为是VoIP信令的未来。

⁴<http://zh.wikipedia.org/wiki/IP电话>

第二章 初识FreeSWITCH

2.1 什么是 FreeSWITCH ?

FreeSWITCH 是一个开源的电话交换平台。它具有很强的可伸缩性—从一个简单的软电话客户端到运营商级的软交换设备几乎无所不能。它能原生地运行于Windows、Max OS X、Linux、BSD 及 Solaris 等诸多32/64位平台。可以用作一个简单的交换引擎、一个PBX，一个媒体网关或媒体支持IVR的服务器，或在运营商的IMS网络中担当Application Server等。

它支持SIP、H323、Skype、Google Talk等协议，并能很容易地与各种开源的PBX系统如sipXecs、Call Weaver、Bayonne、YATE及Asterisk等通信。FreeSWITCH 遵循RFC并支持很多高级的SIP特性，如 presence、BLF、SLA以及TCP、TLS和sRTP等。它也可以用作一个SBC进行透明的SIP代理（proxy）以支持其它媒体如T.38等。

FreeSWITCH 支持宽带及窄带语音编码，电话会议桥可同时支持8、12、16、24、32及48kHz的语音。

至于 FreeSWITCH 能做什么，通俗一点，我们可能以这样说：

- 任何像华为、中兴等局端交换机能做的事情它都能做
- 任何像Avaya、Cisco、NEC等企业级交换机能做的事情它也能做
- 甚至上述两类交换机不能做的事情它也能做

说的专业一点，它实际上就是一个 B2BUA。

2.2 快速体验

FreeSWITCH 的功能确实非常丰富和强大，在进一步学习之前我们先来一次完整的体验。

FreeSWITCH 默认的配置是一个SOHO PBX(家用电话小交换机)，那么我们本章的目标就是从零安装，实现分机互拨电话，测试各种功能，并通过添加一个SIP-PSTN网关拨打PSTN电话。

这样，即使你没有任何使用经验，你也应该能顺利走完本章，从而建立一个直接的认识。在体验过程中，你会遇到一点稍微复杂的配置，如果不能完全理解，也不用担心，我们在后面会详细的介绍。当然，如果你是一个很有经验的 FreeSWITCH 用户，那么大可跳过本章。

2.3 安装FreeSWITCH基本系统

在本文写作时，最新的版本1.2.0-RC2。FreeSWITCH支持32位及64位的Linux、Mac OS X、BSD、Solaris、Windows等众多平台。某些平台上有关于安装包，但本人强烈建议从源代码进行安装，因为 FreeSWITCH 更新非常快，而已编译好的版本通常都比较旧。你可以下载源码包，也可以直接从git仓库中取得最新的代码。与其它项目不同的是，其git的主（master）分支代码通常比稳定的发布版更稳定。而且，当你需要技术支持时，开发人员也通常建议你先升级到git中最新的代码，再看是不是仍有问题。

Windows用户可以直接下载安装文件 <http://files.freeswitch.org/windows/installer/>（再提醒一下，版本比较旧，如果从源代码安装的话，需要Visual Studio 2008或2010）。安装完成执行 `c:\freeswitch\freeswitch.exe` 便可启动，其配置文件都在 `c:\freeswitch\conf\` 目录下。

以下假定你使用 Linux 平台，并假定你有 Linux 的基本知识。如何从头安装 Linux 超出了本书的范围，而且，你也可以很容易的从网上找到这样的资料。一般来说，任何发行套件都是可以的，但是，有些发行套件的内核、文件系统、编译环境，LibC 版本会有一些问题。所以，如果你在遇到问题后想获得社区支持，最好选择一种大家都熟悉的发行套件。FreeSWITCH 开发者使用的平台是 CentOS 5.2/5.3（使用CentOS 5.8 也没有问题，有人也成功地在CentOS 6 上成功安装部署，但版本并不总是越新越好），社区中也有许多人在使用 Ubuntu 和 Debian，如果你想用于生产环境，建议使用 LTS (Long Term Support) 的版本，即 Ubuntu8.04/10.04/12.04 或 Debian Stable。在安装之前，我们需要先准备一些环境—FreeSWITCH 可以以普通用户权限运行，但为了简单起见，以下所有操作均用 root 执行(这不是一个好习惯，但在此，让我们专注于FreeSWITCH而不是Linux）：

CentOS:

```
1 yum install -y autoconf automake libtool gcc-c++
2 yum install -y ncurses-devel make zlib-devel libjpeg-devel
```

Ubuntu:

```
1 apt-get -y install build-essential automake autoconf git-core wget libtool
2 apt-get -y install libncurses5-dev libtiff-dev libjpeg-dev zlib1g-dev
```

以下三种安装方式任选其一，默认安装位置在`/usr/local/freeswitch`。安装过程中会下载源代码目录，请保留，以便以后升级及安装配置其它组件。值得一提的是，CentOS默认的软件仓库中可能没有git，如下你需要使用git安装，则可以先安装 rpm-forge (<http://pkgs.repoforge.org/rpmforge-release/>)，然后再安装 git。

2.3.1 最快安装（推荐）

```
1 wget http://www.freeswitch.org/eg/Makefile && make install
```

以上命令会下载一个 Makefile，然后使用 make 执行安装过程。安装过程中它会从 Git 仓库中获取代码，实际上执行的操作跟下一种安装方式相同。

2.3.2 从 Git 仓库安装：

从代码库安装能让你永远使用最新的版本：

```
1 git clone git://git.freeswitch.org/freeswitch.git
2 cd freeswitch
3 ./bootstrap.sh
4 ./configure
5 make install
```

这是在在 Linux 上从源代码安装软件的标准过程。首先第1行使用git工具从软件仓库中下载最新的源代码，第3行执行bootstrap.sh初始化一些编译环境，第4行配置编译环境，第5行编译安装。

2.3.3 解压缩源码包安装：

```
1 wget http://files.freeswitch.org/freeswitch-1.2.rc2.tar.bz2
2 tar xvjf http://files.freeswitch.org/freeswitch-1.2.rc2.tar.bz2
3 cd freeswitch-1.2
4 ./configure
5 make install
```

与上一种方法不同的是，它不需要执行过bootstrap.sh（打包前已经执行过了，因而不需要automake和autoconf工具），便可以直接配置安装。

2.4 安装声音文件

在以下例子中我们需要一些声音文件，而安装这些声音文件也异常简单。你只需在源代码目录中执行：

```
1 make sounds-install
2 make moh-install
```

以下高质量的声音文件可选择安装。FreeSWITCH支持8、16、32及48kHz的语音，很少有其它电话系统支持如此多的抽样频率（普通电话是8K，更高频率意味着更好的通话质量）。

```
1 make cd-sounds-install  
2 make cd-moh-install
```

安装完成后，会显示一个有用的帮助，

```
1 +----- FreeSWITCH install Complete -----+
2 + FreeSWITCH has been successfully installed. +
3 +
4 +     Install sounds: +
5 +         (uhd-sounds includes hd-sounds, sounds) +
6 +         (hd-sounds includes sounds) +
7 + -----
8 +             make cd-sounds-install +
9 +             make cd-moh-install +
10 +
11 +             make uhd-sounds-install +
12 +             make uhd-moh-install +
13 +
14 +             make hd-sounds-install +
15 +             make hd-moh-install +
16 +
17 +             make sounds-install +
18 +             make moh-install +
19 +
20 +     Install non english sounds: +
21 +     replace XX with language +
22 +     (ru : Russian) +
23 + -----
24 +         make cd-sounds-XX-install +
25 +         make uhd-sounds-XX-install +
26 +         make hd-sounds-XX-install +
27 +         make sounds-XX-install +
28 +
29 +     Upgrade to latest: +
30 + -----
31 +         make current +
32 +
33 +     Rebuild all: +
34 + -----
35 +         make sure +
36 +
37 +     Install/Re-install default config: +
38 + -----
39 +         make samples +
40 +
41 +     Additional resources: +
42 + -----
43 +         http://www.freeswitch.org +
44 +         http://wiki.freeswitch.org +
45 +         http://jira.freeswitch.org +
46 +         http://lists.freeswitch.org +
47 +
48 +         irc.freenode.net / #freeswitch +
49 +-----+
```

至此，已经安装完了。在Unix类操作系统上，其默认的安装位置是/usr/local/freeswitch，下文所述的路径全部相对于该路径。两个常用的命令是 bin/freeswitch 和 bin/fs_cli，为了便于使用，建议将这两个命令做符号链接放到你的搜索路径中，如：

```
1 ln -sf /usr/local/freeswitch/bin/freeswitch /usr/bin/
2 ln -sf /usr/local/freeswitch/bin/fs_cli /usr/bin/
```

接下来就应该可以启动了，通过在终端中执行freeswitch命令(如果你已做符号链接的话，否则要执行/usr/local/freeswitch/bin/freeswitch)可以将其启动到前台，启动过程中会有许多log输出，第一次启动时会有一些错误和警告，可以不用理会。启动完成后会进入到系统控制台，并显示类似的提示符“freeswitch@localhost>”(以下简作“FS>”)。通过在控制台中输入shutdown命令可以关闭FreeSWITCH。

如果您想将FreeSWITCH启动到后台(daemon，服务模式)，可以使用freeswitch -nc (No console)。后台模式没有控制台，如果这时想控制FreeSWITCH，可以使用客户端软件fs_cli连接。当然，也可以直接在Linux 提示符下通过 freeswitch -stop 命令关闭。如果不想退出 FreeSWITCH 服务，只退出fs_cli客户端，则需要输入 /exit，或Ctrl + D，或者，直接关掉终端窗口。

2.4.1 连接SIP软电话

FreeSWITCH最典型的应用是作为一个服务器(它实际上是一个背靠背的用户代理，B2BUA)，并用电话客户端软件(一般叫软电话)连接到它。虽然FreeSWITCH支持IAX、H323、Skype、Gtalk等众多通信协议，但其最主要的协议还是SIP。支持SIP的软电话有很多，我最常用的是X-Lite和Zoiper。这两款软电话都支持Linux、Mac OS X和Windows平台，免费使用但是不开源。在Linux上你还可以使用ekiga软电话。

强烈建议在同一局域网上的其它机器上安装软电话，并确保麦克风和耳机可以正常工作。当然，如果你没有多余的机器做这个实验，那么你也可以在同一台机器上安装。只是需要注意，软电话不要占用UDP 5060 端口，因为FreeSWITCH 默认要使用该端口，这是新手常会遇到的一个问题。你可以通过先启动FreeSWITCH再启动软电话来避免该问题(如果它们发现5060端口已被占用，会选择其它端口)，另外有些软电话允许你修改本地监听端口¹。

通过输入以下命令可以知道FreeSWITCH监听在哪个IP地址上，记住这个IP地址(:5060以前的部分)，下面要用到：

```
1 netstat -an | grep 5060
2
3 udp      0      0 192.168.0.9:5060          0.0.0.0:*
```

FreeSWITCH默认配置了1000~1019共20个用户，你可以随便选择一个用户进行配置：

在X-Lite上点右键，选Sip Account Settings...，点Add添加一个账号，填入以下参数(Zoiper可参照配置)：

¹特别注意，如果你是在Linux上，并且在同一台机器上使用Ekiga的话，肯定会遇到这个问题。你需要手工使用gconf_editor来更改ekiga使用的端口，当然，也可以改FreeSWITCH的端口，如果你会的话。

```
1 Display Name: 1000
2 User name: 1000
3 Password: 1234
4 Authorization user name: 1000
5 Domain: 你的IP地址, 就是刚才你记住的那个
```

其它都使用默认设置，点 OK 就可以了。然后点 Close 关闭 Sip Account 设置窗口。这时 X-Lite 将自动向 FreeSWITCH 注册。注册成功后会显示“Ready. Your username is 1000”，另外，左侧的“拨打电话”（Dial）按钮会变成绿色的。如下图。



图 2.1: XLite软电话

激动人心的时刻就要来了。输入“9664”按回车（或按绿色拨打电话按钮），就应该能听到保持音乐(MOH, Music on Hold)。如果听不到也不要气馁，看一下控制台上有没有提示什么错误。如果有“File Not Found”之类的提示，多半是声音文件没有安装，重新查看 make moh-install 是否有错误。接下来，可以依次试试拨打以下号码：

号码	说明
9664	保持音乐
9196	echo, 回音测试
9195	echo, 回音测试, 延迟5秒
9197	milliwatt extension, 铃音生成
9198	TGML 铃音生成示例
5000	示例IVR
4000	听取语音信箱
33xx	电话会议, 48K(其中xx可为00-99, 下同)
32xx	电话会议, 32K
31xx	电话会议, 16K
30xx	电话会议, 8K
2000-2002	呼叫组
1000-1019	默认分机号

表一： 默认号码及说明

详情见 http://wiki.freeswitch.org/wiki/Default_Dialplan_QRF。

另外，也许你想尝试注册另外一个SIP用户并在两者间通话。最好是在同一个局域网中的另外一台机器上启动另一个 X-Lite，并使用 1001 注册，注册完毕后就可以在 1000 上呼叫 1001，或在 1001 上呼叫 1000。当然，你仍然可以在同一台机器上做这件事（比方说用Zoiper注册为1001），需要注意的是，由于你机器上只有一个声卡，两者可能会争用声音设备。特别是在Linux上，有些软件会独占声音设备。如果同时也有一个USB接口的耳机，那就可以设置不同的软件使用不同的声音设备。

2.5 配置简介

FreeSWITCH配置文件默认放在 conf/，它由一系列XML配置文件组成。最顶层的文件是freeswitch.xml，系统启动时它依次装入其它一些XML文件并最终组成一个大的XML文件。

文件	说明
vars.xml	一些常用变量
dialplan/default.xml	缺省的拨号计划
directory/default/*.xml	SIP用户，每用户一个文件
sip_profiles/internal.xml	一个SIP profile，或称作一个SIP-UA，监听在本地IP及端口5060，一般供内网用户使用
sip_profiles/externa.xml	另一个SIP-UA，用作外部连接，端口5080
autoload_configs/modules.conf.xml	配置当FreeSWITCH启动时自动装载哪些模块

2.6 添加一个新的SIP用户

FreeSWITCH默认设置了20个用户(1000-1019)，如果你需要更多的用户，或者想通过添加一个用户来学习FreeSWITCH配置，只需要简单执行以下三步：

- 在 conf/directory/default/ 增加一个用户配置文件
- 修改拨号计划(Dialplan)使其它用户可以呼叫到它
- 重新加载配置使其生效

如果想添加用户Jack，分机号是1234。只需要到 conf/directory/default 目录下，将 1000.xml 拷贝到 1234.xml。打开1234.xml，将所有1000都改为1234。并把 effective_caller_id_name 的值改为 Jack，然后存盘退出。如：

```
1 <variable name="effective_caller_id_name" value="Jack"/>
```

接下来，打开 conf/dialplan/default.xml，找到下面一行

```
1 <condition field="destination_number" expression="^(10[01][0-9])$">
```

改为

```
1 <condition field="destination_number" expression="^(10[01][0-9]|1234)$">
```

熟悉正则表达式的人应该知道，“^(10[01][0-9])\$”匹配被叫号码1000-1019。因此我们修改之后的表达式就多匹配了一个1234。FreeSWITCH使用Perl兼容的正则表达式(PCRE)。

现在，回到控制台，或启动fs_cli，执行reloadxml命令或按快捷F6键，使新的配置生效。

找到刚才注册为1001的那个软电话(或启动一个新的，如果你有足够的机器的话)，把1001都改为1234然后重新注册，则可以与1000相互进行拨打测试了。如果没有多台机器，在同一台机器上运行多个软电话可能有冲突，这时，也可以直接进在FreeSWITCH控制台上使用命令进行测试：

```
1 FS> sofia status profile internal reg      (显示多少用户已注册)
2 FS> originate sofia/internal/1000 &echo   (拨打1000并执行echo程序)
3 FS> originate user/1000 &echo             (同上)
4 FS> originate user/1000 9999              (相当于在软电话1000上拨打9999)
5 FS> originate user/1000 9999 XML default (同上)
```

其中，echo() 程序一个很简单的程序，它只是将你说话的内容原样再放给你听，在测试时很有用，在本书中，我们会经常用它来测试。

2.7 FreeSWITCH用作软电话

FreeSWITCH也可以简单的用作一个软电话，如X-Lite。虽然相比而言比配置X-Lite略微麻烦一些，但你会从中得到更多好处：FreeSWITCH是开源的，更强大、灵活。关键是它是目前我所知道的唯一支持CELT高清通话的软电话。

FreeSWITCH使用mod_portaudio支持你本地的声音设备。该模块默认是不编译的。到你的源代码树下，执行：

```
1 make mod_portaudio  
2 make mod_portaudio-install
```

其它的模块也可以依照上面的方式进行重新编译和安装。然后到控制台中，执行：

```
1 FS> load mod_portaudio
```

如果得到“Cannot find an input device”之类的错误可能是你的声卡驱动有问题。如果是提示“+OK”就是成功了，接着执行：

```
1 FS> pa devlist  
2  
3 API CALL [pa(devlist)] output:  
4 0;Built-in Microphone;2;o;  
5 1;Built-in Speaker;0;2;r  
6 2;Built-in Headphone;0;2;  
7 3;Logitech USB Headset;0;2;o  
8 4;Logitech USB Headset;1;0;i
```

以上是在我笔记本上的输出，它列出了所有的声音设备。其中，3和4最后的“o”和“i”分别代表声音输出(out)和输入(in)设备。在你的电脑上可能不一样，如果你想选择其它设备，可以使用命令：

```
1 FS> pa indev #0  
2 FS> pa outdev #2
```

以上命令会选择我电脑上内置的麦克风和耳机。

接下来你就可以有一个可以用命令行控制的软电话了，酷吧？

```
1 FS> pa looptest      (回路测试, echo)  
2 FS> pa call 9999  
3 FS> pa call 1000  
4 FS> pa hangup
```

如上所示，你可以呼叫刚才试过的所有号码。现在假设想从SIP分机1000呼叫到你，那需要修改拨号计划(Dialplan)。用你喜欢的编辑器编辑以下文件放到conf/dialplan/default/portaudio.xml

```

1 <include>
2   <extension name="call me">
3     <condition field="destination_number" expression="^(me|12345678)$">
4       <action application="bridge" data="portaudio"/>
5     </condition>
6   </extension>
7 </include>
```

然后，在控制台中按“F6”或输入以下命令使之生效：

```
1 FS> reloadxml
```

在分机1000上呼叫“me”或“12345678”(你肯定想为自己选择一个更酷的号码)，然后在控制台上应该能看到类似“[DEBUG] mod_portaudio.c:268 BRRRRING! BRRRRING! call 1”的输出（如果看不到的话按“F8”能得到详细的Log），这说明你的软电话在振铃。多打几个回车，然后输入“pa answer”就可以接听电话了。“pa hangup”可以挂断电话。

当然，你肯定希望在振铃时能听到真正的振铃音而不是看什么BRRRRRING。好办，选择一个好听一声音文件(.wav格式)，编辑conf/autoload_configs/portaudio.conf.xml，修改下面一行：

```
1 <param name="ring-file" value="/home/your_name/your_ring_file.wav"/>
```

然后重新加载模块：

```

1 FS> reloadxml
2 FS> reload mod_portaudio
```

再打打试试，看是否能听到振铃音了？

如果你用不惯字符界面，可以看一下FreeSWITCH-Air(<http://www.freeswitch.org.cn/download>)，它为FreeSWITCH提供一个简洁的软电话的图形界面。另外，如果你需要高清通话，除需要设置相关的语音编解码器(codec)外，你还需要有一幅好的耳机才能达到最好的效果。本人使用的是一款USB耳机。

另外两款基于FreeSWITCH的软电话是 [FSComm](#)² (QT实现) 和 [FSClient](#)³ (C#实现) 。

²<http://wiki.freeswitch.org/wiki/FSComm>

³<http://wiki.freeswitch.org/wiki/FSClient>

2.8 配置SIP网关拨打外部电话

如果你在某个运营商拥有SIP账号，你就可以配置上拨打外部电话了。该SIP账号（或提供该账号的设备）在 FreeSWITCH 中称为SIP网关（Gateway）。添加一个网关只需要在 conf/sip_profiles/external/ 创建一个XML文件，名字可以随便起，如gw1.xml。

```

1 <gateway name="gw1">
2   <param name="realm" value="SIP服务器地址, 可以是IP或IP:端口号"/>
3   <param name="username" value="SIP用户名"/>
4   <param name="password" value="密码"/>
5 </gateway>
```

如果你的SIP网关还需要其它参数，可以参阅同目录下的 example.xml，但一般来说上述参数就够了。你可以重启 FreeSWITCH，或者执行以下命令使用之生效。

```
1 FS> sofia profile external rescan
```

显示一下状态：

```
1 FS> sofia status
```

如果显示 gateway gw1 的状态是 REGED，则表明正确的注册到了网关上。你可以先用命令试一下网关是否工作正常：

```
1 FS> originate sofia/gateway/gw1/xxxxxx &echo()
```

以上命令会通过网关 gw1 呼叫号码 xxxxxx（可能是你的手机号），被叫号码接听电话后，FreeSWITCH 会执行 echo() 程序，你就应该能听到自己的回音。

2.8.1 从某一分机上呼出

如果网关测试正常，你就可以配置从你的SIP软电话或portaudio呼出了。由于我们是把 FreeSWITCH 当作 PBX 用，我们需要选一个出局字冠。常见的 PBX 一般是内部拨小号，打外部电话就需要加拨 0 或先拨 9。当然，这是你自己的交换机，你可以用任何你喜欢的数字（甚至是字母）。继续修改拨号计划，创建新XML文件：conf/dialplan/default/call_out.xml：

```

1 <include>
2   <extension name="call_out">
3     <condition field="destination_number" expression="^0(\d+)$">
4       <action application="bridge" data="sofia/gateway/gw1/$1"/>
5     </condition>
6   </extension>
7 </include>
```

其中, (\d+)为正则表达式, 匹配 0 后面的所有数字并存到变量 \$1 中。然后通过 bridge 程序通过网关 gw1 打出该号码。当然, 建立该XML后需要在控制台中执行 reloadxml 使用之生效。

2.8.2 呼入电话处理。

如果你的 SIP 网关支持呼入, 那么你需要知道呼入的 DID。DID的全称是 Direct Inbound Dial, 即直接呼入。一般来说, 呼入的 DID 就是你的 SIP 号码, 如果你不知道, 也没关系, 后面你会学会如何得到 (如果你等不及, 可以先翻到第八章看看)。编辑以下XML文件放到 conf/dialplan/public/my_did.xml

```
1 <include>
2   <extension name="public_did">
3     <condition field="destination_number" expression="^(\d+)$">
4       <action application="transfer" data="1000 XML default"/>
5     </condition>
6   </extension>
7 </include>
```

reloadxml 使之生效。上述配置会将来话直接转接到分机 1000 上。在后面的章节你会学到如何更灵活的处理呼入电话, 如转接到语音菜单或语音信箱等。

2.9 小结

其实本章涵盖了从安装、配置到调试、使用的相当多的内容, 如果你能顺利走到这儿, 你肯定对 FreeSWITCH 已经爱不释手了。如果你卡在了某处, 或某些功能未能实现, 也不是你的错, 主要是因为 FreeSWITCH 博大精深, 我不能在短短的一章内把所有的方面解释清楚。在后面的章节中, 你会学到更多的基本概念、更加深入地了解 FreeSWITCH 的哲学, 学到更多的调试技术和技巧, 解决任何问题都会是小菜一碟了。

第三章 PSTN与PBX 业务

在继续学习 FreeSWITCH 之前，我们有必要了解一下传统的电话网所能提供的服务。这些服务有的是你已经熟悉的，有的也可能没听说过。有一些业务在 VoIP 中实现起来就异常简单，而有一些业务已经不需要了。

3.1 PSTN 业务

3.1.1 POTS

除为用户提供基本的话音通话外，PSTN 还能提供一些附加的业务，这些业务在国外称为普通老式电话业务（POTS，Plain Old Telephone Service），而在国内，我们称之为新业务，当然，这还是沿用数年前的叫法。这些业务有的是收费的，有的是不收费的，而这些新业务号码通常以 * 开头。古老的话机是转盘式的（就是电影上蒋介石用的那种话机），使用脉冲方式拨号，只能拨0~9的号码，现在在民间已极少再使用。现代的话机多为按键式，使双音频方式拨号，有0~9及*和#字键。其中，*字键通常读作“星”，但有些运营商的话务员读作“米”，我觉得无所适从。另外有的话机上有A、B、C、D键，很少用到。在某些新业务中（如三方通话），会用到话机上的叉簧，快速拍一下以给交换机传递信号，某些话机上有 Flash 键或 R 键或闪断键能实现相同的功能。下面仅列举几种典型的业务：

- 缩位拨号：通过事先登记的代码拨打长号码，如 **1 则可以拨打指定的 12345678，该功能比较实用。
- 呼叫转移：基本的有三种：无条件转移，即任何来电转移至事先登记的号码；遇忙转移，若被叫忙，则转移；无应答转移，若指定时间内无应答，则转移。其中无条件转移的登记方式为 *57*电话号码#，取消方式为 #57#。登记成功后，所有到该话机的来电会转到所登记的电话号码。如在话机A上操作 *57*B#，则所有对A的呼叫都会转移到B上。适用于将家里或办公室电话转移到手机上的情况。运营商也经常使用该功能做一些特殊的业务，如改号通知—通过后台操作将某一号码转移至特定的语音平台，实现类似“您拨打的电话已改号，新的号码是XXX”的功能。

- 立即热线：拿起电话不用拨号即自动拨打某号码，我在北京某银行网点用过该项业务，拿起电话直接连接到他们的自助语音服务。还有一类似的功能叫延迟热线，即延迟一段特定的时间（如5秒）再自动拨号。
- 延迟热线：与立即热线差不多，区别是摘机后会延迟一段时间自动拨号，在特殊场所有用。
- 呼叫等待。被叫忙时，主叫仍听正常的回铃音（或个性化的语音提示：请不要挂机，您拨打的电话正在通话中...），而交换机会通过特殊的提示音提示有新电话呼入，被叫可选择是否接听，或在两者间切换。
- 三方通话：通过比较复杂的操作实现三方通话，某些交换机支持最多5方的多方通话（会议电话），更多方的电话会议系统需要专门的平台。
- 来电显示：就是在被叫话机上显示主叫方的电话号码。
- 呼出限制：在电话费还是很贵的年代比较有用，可以使用密码限制话机能否打长途等，也可以限制小孩乱打电话。
- 免打扰服务：登记该业务后，如果有来电，交换机会提示主叫用户被叫用户不想被打扰。不过，实际使用起来，直接拨掉电话线比登记这个容易多了。
- 叫醒服务：登记后在相应的时间电话会振铃，不过在这个手机异常普及的年代相信一般人不会用这个功能了。
- 遇忙回叫：如果被叫忙，则主叫可以按一个特殊的号码登录该业务，待被叫空闲后双方话机会自动振铃，接听后双方进行通话，省了好多重复拨叫的操作。不过，该业务一般限制在主、被叫用户都在同一交换机上的情况，因为实际使用意义不大。

3.1.2 商务业务

运营商的大部分收入还是来自于商务业务。

模拟中继线

模拟中继线又称为用户小交换机，它主要提供号码连选功能。典型应用是提供一个总机号（又称引示号）及若干条中继线（实际上就是普通的电话线）。当有人拨号总机号时，交换机会根据指定的策略选择一条空闲的中继线呼入。而用户端通常会接PBX设备，下设分机。当用户呼出时，通过用户端的PBX设备选择一条空闲的线路，用户可选择是否显示总机号。

数字中继线

如果用户需要的中线继数量较多时，数字中继线能提供更稳定的服务，设备通常支持2M的一号信令或30B+D的ISDN信令。

虚拟网

虚拟网又称商务组（BCG, Business Call Group）或汇线通（Centrex）业务。虚拟网主要提供在无需用户端PBX设备的情况下，实现网内（组）电话互拨小号，通常小号间的通话是免费的，但要比普通电话多收月租费。

虚拟网与模拟中继线的区别是它的每路电话都是直线，可以直接呼入呼出，但需要占用更多的PSTN号码资源。它与普通电话的区别是网内可以互拨小号。

立即计费

传统的PSTN需要通过额外的系统来计算通话费用，通常需要有一段时间的滞后。而立即计费主要用于酒店等需要立即计费的场合，通常使用ISDN信令配合用户端的话务台软件实现。

VPN

VPN(Virtual Private Network)的全称是虚拟专用网，有别于Internet上的VPN。它主要是用在连接大型企业在不同城市的分支机构，实现公司内部互拨小号。

3.1.3 其它增值业务

传统的语音业务所带来的收入比例越来越低，因此，各大运营商都纷纷推出基于数据库和计算机系统的各种增值业务以扩大收入。这些业务包括预付费业务（电话卡类业务等），800、400业务以及彩铃、电话秘书台（语音信箱）等。

3.2 PBX 业务

PBX (Private Branch eXchange) 的全称是专用小交换机。企业使用 PBX 的好处是可以自己控制内部呼叫，而且内部通话免费。它通常可以提供呼叫保持、自动选线、呼叫转移、呼叫转接等基本功能，比较高级的小交换机还可以提供自动总机、三方通话、语音信箱等。

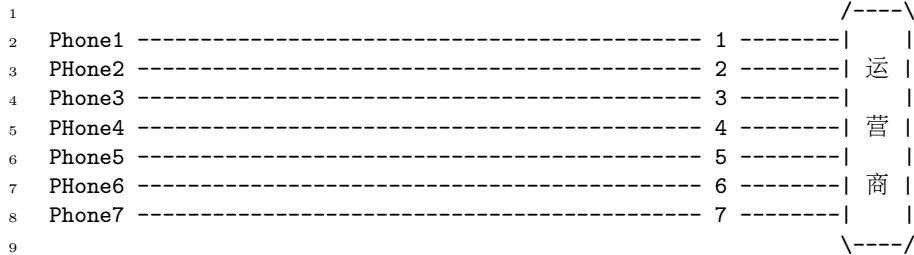
PBX 的上端通过模拟或数字中继线连接到PSTN，而下端则直接接话机。

中继线

由于 FreeSWITCH 缺省配置可以用作一个PBX，深入理解PBX是非常必要的。下面，我们以模拟中继线为例，通过一则故事进行说明。

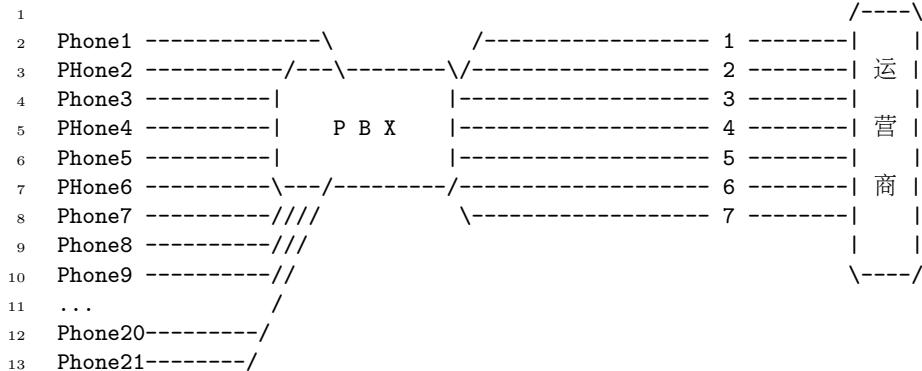
我们刚开了一个公司，需要 7 部电话，于是申请了 7 条模拟中继线。上文已经指出，实际上就是 7 条普通的电话线，只是在 PSTN 交换机端有特殊的数据设置，将其逻辑上分为一个组，并

为该组设一个总机号。我们有幸选到了一个很酷的号码 – 88888888（它可以是一个虚拟的号码，或者是其中某一条中继线的号码）。而其它的中继线号则可能是，44440001 ~ 44440007。现在，我们把这 7 条线都接上话机。如果有人呼叫 88888888，则 PSTN 交换机会从 7 条线中选择一条空闲的线呼入，因此某个电话会振铃。一般来说，交换机有两种选线策略——顺序选线和循环选线。所谓顺序选，就是每次都从 44440001 开始，寻找一条空闲的线路进行呼入；而循环选则是每次都从上一次呼叫的下一个开始选起，使用这种选线方式，每个话机接到的电话数会比较平均。



为维护企业形象，当有人呼出时，不管是从哪个分机呼出，都显示总机号 88888888。当然，也可以显示单线的号码（如44440004），这个要在 PSTN 交换机端设置，一旦设置后，用户端不能动态更改。

一个月后，公司发展到 21 个人，因此，需要 21 部电话。但由于不会所有人同时打电话，因此我们买了一个小交换机。把原来的 7 条中线线接到小交换机上，而每个人都有一个分机号，从 601 到 621。当客户打总机号时，PSTN 仍然会选择一条线进入小交换机，这时候，选线方式已经不像以前那样重要，因为现在是小交换机在接电话，对它来说，7 条线哪条都一样。就这样，小交换机接了电话，并播放“您好，欢迎致电XX公司，请直拨分机号，查号请拨 0”，如果客户按某一分机号，则对应分机振铃，电话接通。



有了小交换机，内部通话就免费了。但增加了另外一个问题，就是如果拨打外线，则需要先拨一个特殊的数字，一般是 0 或 9。有的小交换机会送二次拨号音，即你拿起电话，听小交换机的拨号音，拨了 0 之后，则听到外部 PSTN 交换机的拨号音，表明你是可以拨打外线了。总之，小交换机会选择一条空闲的中继线对外呼叫。

其中，21 : 7 称为集线比。即 3 比 1。集线比是由话务量决定的，如果同时通话的人数比较多，那我们可能考虑把中继线增加到12条，集线比就降为 2 比 1。

即使增加了线路，也经常会遇到这样的情况，由于打进来的电话太多，占用了太多的线路，经常一个电话都打不出，因此，我们联系电信部门，将中继线分为两组，其中4条只进不出，4条只出不进，4条能出能进。在电信术语中，分别叫做单出，单入和双向，而北京则称发专、受专和双向，需要指出，对电信部门来说，出和入跟我们是相反的，因为我们（小交换机）的出，对应他们（PSTN 交换机）的入。当然，这种分配方式降低了总体线路的使用率，为此，我们把每个组都增加1条线，现在总共中继线达到15条。

又过了几天，有客户反映这样的情况，打电话经常无人接听，需要打好几遍；而同时，内部也有人反映往外打电话时有时拨 0 没反应，再试一次就好了。我们没有这方面的经验，只好请教PBX专家，专家说可能是某条外线断了，因为，如果有一条线断了，交换机仍会向对方送回铃音，跟没接话机是一样的。但到底是哪条线断了，却不好查。由于双方都是自动选线。我们只好将每条线都从小交换机上拔下来，接上话机试一试。某些小交换机也支持指定端口拨打功能，即在拨打真正号码前先拨几个特殊的数字可以选择从指定的线路出去。实际上还有一种方法。一般来说，每条单线的号码也都是可以呼入的，只是外人不知道而已。只需要依次拨打所有的单线号码就可以知道哪条线是断了。当然，这依赖于PSTN交换机端的设置，有些城市（甚至同一城市不同的交换机都有不同的设置）默认设置单线是不允许呼入的。

几天后，老板又很幸运地搞到了一个新号码 66666666，该号码并未加入中继线组，而是直接扯了根线拉到老板办公桌上。为了能拨打内线，他不得不把办公桌上放两部电话，另一部专门打内线。后来技术人员小张仔细阅读了说明书，发现该小交换机功能还比较强，就进行了以下设置：将 66666666 这个号码接到小交换机上，仍给老板一个内线电话，同时在小交换机上进行设置，只有老板打出时才走66666666这个端口；而对于打入的电话，也不播放“欢迎致电XX公司...”，而是直接向老板电话振铃。这种拨入方式叫做 DID，即对内直接呼叫（Direct Inbound Dial）。

接下来，随着公司的发展，加入的中继线条数越来越多，而维护起来更加复杂，比如，其中有一条线断了，在很长的一段时间内你根本不知道，即使知道了，要找到是哪条线也非常麻烦。后来，当公司发展到 100 人的时候，公司终于购买了新设备，并将中继线换成了两条 E1 数字线路，可同时支持 60 路电话。

公司发展一帆风顺，电话量也越来越多，公司有了很多分支机构，也有了更多客户，需要更复杂的语音菜单及更智能的电话分配策略，而更换专门的电话系统不仅价格昂贵，而且跟现有业务系统的集成难度很大。在综合考虑了多种解决方案以后，技术人员开始学习 FreeSWITCH

.....

第四章 SIP协议

SIP协议是FreeSWITCH的核心协议。讲清楚SIP需要很大篇幅，甚至是整本书。本书是关于FreeSWITCH的，重点不在SIP。因此，我将仅就理解FreeSWITCH必需的一些概念加以通俗的解释，更严肃一些的资料请参阅其它资料或相关RFC（Request For Comments，如 RFC3261）。

4.1 SIP的概念和相关元素

会话初始协议（Session Initiation Protocol）是一个控制发起、修改和终结交互式多媒体会话的信令协议。它是由 IETF（Internet Engineering Task Force, Internet工程任务组）在 RFC 2543 中定义的。最早发布于 1999 年 3 月，后来在 2002 年 6 月又发布了一个新的标准 RFC 3261。

SIP 是一个基于文本的协议，在这一点上与 HTTP 和 SMTP 相似。我们来对比一个简单的 SIP 请求与 HTTP 请求：

```
1 GET /index.html HTTP/1.1
2
3 INVITE sip:seven@freeswitch.org.cn SIP/2.0
```

请求由三部分组成。在 HTTP 中（第1行），GET 指明一个获取资源（文件）的动作，而 /index.html 则是资源的地址，最后是协议版本号。而在 SIP 中（第3行），INVITE 表示发起一次请求，seven@freeswitch.org.cn 为请求的地址，称为 SIP URI，第3部分也是版本号。其中，SIP URI很类似一个电子邮件地址，其格式为“协议:名称@主机”。与 http 和 https 相对应，有 *sip* 和 *sips*，后者是加密的；名称可以是一串数字的电话号码，也可以是字母表示的名称；而主机可以是一个域名，也可以是一个IP地址。

SIP 是一个对等的协议，类似 P2P。不像传统电话那样必须有一个中心的交换机，它可以在不需要服务器的情况下进行通信，只要通信双方都彼此知道对方地址（或者，只有一方知道另一方地址），如下图，bob 给 alice 发送一个 INVITE 请求，说“Hi, 一起吃饭吧...”，alice 说“好的，OK”，电话就通了。

在 SIP 网络中，alice 和 bob 都叫做用户代理（UA, User Agent）。UA 是在 SIP 网络中发起或响应 SIP 处理的逻辑功能。UA是有状态的，也就是说，它维护会话（或称对话）的状态。UA

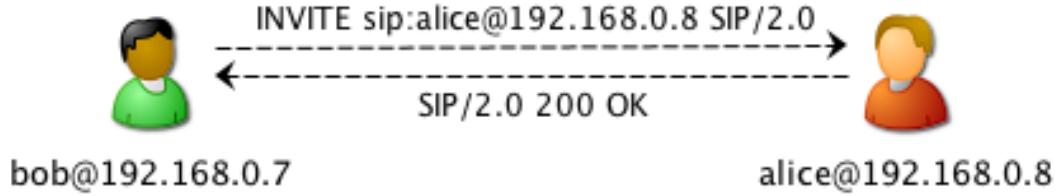


图 4.1: SIP 点对点通信

有两种功能：一种是 UAC (UA Client 用户代理客户端)，它是发起 SIP 请求的一方，如上图的 bob。另一种是 UAS (UA Server)，它是接受请求并发送响应的一方，如上图中的 alice。由于 SIP 是对等的，如果 alice 呼叫 bob 时（有时候 alice 也主动叫 bob 一起吃饭），alice 就称为 UAC，而 bob 则执行 UAS 的功能。一般来说，UA 都会实现上述两种功能。

设想 bob 和 alice 是经人介绍认识的，而他们还不熟悉，bob 想请 alice 吃饭就需要一个中间人 (M) 传话，而这个中间人就叫代理服务器 (Proxy Server)。还有另一种中间人叫做重定向服务器 (Redirect Server)，它类似于这样的方式工作——中间人 M 告诉 bob，我也不知道 alice 在哪里，但我老婆知道，要不然我告诉你我老婆的电话，你直接问她吧，我老婆叫 W。这样，M 就成了一个重定向服务器，而他老婆 W 则是真正的代理服务器。这两种服务器都是 UAS，它们主要是提供一对欲通话的 UA 之间的路由选择功能。

还有一种 UAS 叫做注册服务器。试想这样一种情况，alice 还是个学生，没有自己的手机，但它又希望 bob 能随时找到她，于是当她在学校时就告诉中间人 M 说她在学校，如果有事打她可以打宿舍的电话；而当她回家时也通知 M 说有事打家里电话。只要 alice 换一个新的位置，它就要向 M 重新“注册”新位置的电话，以让 M 能随时找到她，这时候 M 就是一个注册服务器。

最后一种叫做背靠背用户代理 (B2BUA, Back-to-Back UA)。需要指出，其实 RFC 3261 并没有定义 B2BUA 的功能，它只是一对 UAS 和 UAC 的串联。FreeSWITCH 就是一个典型的 B2BUA，事实上，B2BUA 的概念会贯穿本书始终，所以，在此我们需要多花一点笔墨来解释。

我们来看上述故事的另一个版本：M 和 W 是一对恩爱夫妻。M 认识 bob 而 W 认识 alice。M 和 W 有意撮合两个年轻人，但见面时由于两人太腼腆而互相没留电话号码。事后 bob 相知道 alice 对他感觉如何，于是打电话问 M，M 不认识 alice，就转身问老婆 W（注意这次 M 没有直接把 W 电话给 bob），W 接着打电话给 alice，alice 说印象还不错，W 就把这句话告诉 M，M 又转过身告诉 bob。M 和 W 一个面向 bob，一个对着 alice，他们两个合在一起，称作 B2BUA。在这里，bob 是 UAC，因为他发起请求；M 是 UAS，因为他接受 bob 的请求并为他服务；我们把 M 和 W 看做一个整体，他们背靠着背（站着坐着躺着都行），W 是 UAC，因为她又向 alice 发起了请求，最后 alice 是 UAS。其实这里 UAC 和 UAS 的概念也不是那么重要，重要的是要理解这个**背靠背的用户代理**。因为事情还没有完，bob 一听说 alice 对他印象还不错，心花怒放，便想请 alice 吃饭，他告诉 M，M 告诉 W，W 又告诉 alice，alice 问去哪吃，W 又只好问 M，M 再问 bob……在这对年轻人挂断电话之前，M 和 W 只能“背对背”不停地工作。

从上图可以看出，四个人其实全是 UA。从上面故事可以看出，虽然 FreeSWITCH 是

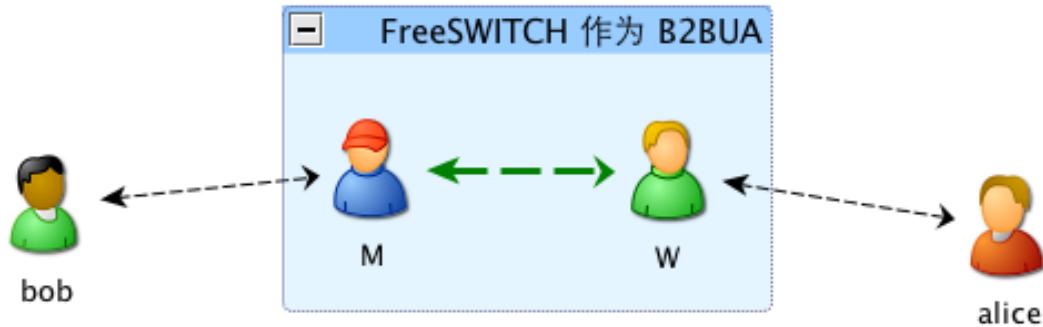


图 4.2: B2BUA

B2BUA，但也可以经过特殊的配置，实现一些代理服务器和重定向服务器的功能，甚至也可以从中间劈开，两边分别作为一个普通的 UA 来工作。这没有什么奇怪的，在 SIP 世界中，所有 UA 都是平等的。具体到实物，则 M 和 W 就组成了实现软交换功能的交换机，它们对外说的语言是 SIP，而在内部，它们则使用自己家的语言沟通。bob 和 alice 就分别成了我们常见的软电话，或者硬件的 SIP 话机。

这里还有一个概念，叫做边界会话控制器（SBC，Session Border Controller）。它主要位于一堆服务器的边界，用于隐藏内部服务器的拓扑结构，抵御外来攻击等。SBC可能是一个代理服务器，也可能是一个B2BUA。

4.2 SIP 注册

不像普通的固定电话网中，电话的地址都是固定的。因特网是开放的，alice 的 UA 可能在家也可能在学校，或者，在世界是任何角落，只要能上网，它就能与世界通信。为了让我们的 FreeSWITCH 服务器能找到它，它必须向服务器进行注册。通常的注册流程是：

```

1      Alice
2      |
3      |      REGISTER
4      |----->|
5      |      SIP/2.0 401 Unauthorized
6      |-----<|
7      |      REGISTER
8      |----->|
9      |      SIP/2.0 200 OK
10     |

```

我们用真正的注册流程进行说明。下面的 SIP 消息是在真正的 FreeSWITCH 中 trace 出来的。其中 FreeSWITCH 服务器的 IP 地址是 192.168.4.4，使用默认的端口号 5060，在这里，我们使用的 SIP 承载方式是 UDP。alice 使用的 UAC 是 Zoiper，端口号是 5090（在我写作时它与 FreeSWITCH 在同一台机器上，所以不能再使用端口 5060）。其中每个消息短横线之间的内容都是 FreeSWITCH 中输出的调试信息，不是 SIP 的一部分。

```

1 -----
2 recv 584 bytes from udp/[192.168.4.4]:5090 at 12:30:57.916812:
3 -----
4 REGISTER sip:192.168.4.4;transport=UDP SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-d9ed3bbae47e568b-1--d8754z-rport
6 Max-Forwards: 70
7 Contact: <sip:alice@192.168.4.4:5090;rinstance=d42207a765c0626b;transport=UDP>
8 To: <sip:alice@192.168.4.4;transport=UDP>
9 From: <sip:alice@192.168.4.4;transport=UDP>;tag=9c709222
10 Call-ID: NmFjNzA3MwY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY20GE.
11 CSeq: 1 REGISTER
12 Expires: 3600
13 Allow: INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO, SUBSCRIBE
14 User-Agent: Zoiper rev.5415
15 Allow-Events: presence
16 Content-Length: 0

```

recv 表明 FreeSWITCH 收到来自 alice 的消息。我们前面已经说过，SIP 是纯文本的协议，类似 HTTP，所以很容易阅读。

- 第一行的 REGISTER 表示是一条注册消息。
- Via 是 SIP 的消息路由，如果 SIP 经过好多代理服务器转发，则会有多条 Via 记录。
- Max-forwards 指出消息最多可以经过多少次转发，主要是为了防止产生死循环。
- Contact 是 alice 家的地址，本例中，FreeSWITCH 应该能在 192.168.4.4 这台机器上的 5090 端口找到她。
- To 和 From 先不深究。
- Call-ID 是本次 SIP 会话（Session）的标志。
- CSeq 是一个序号，由于 UDP 是不可靠的协议，在不可靠的网络上可能丢包，所以有些包需要重发，该序号则可以防止重发引起的消息重复。
- Expires 是说明本次注册的有效期，单位是秒。在本例中，alice 的注册信息会在一小时后失效，它应该在半小时内再次向 FreeSWITCH 注册，以防止 FreeSWITCH 忘掉她。实际上，大部分 UA 的实现都会在几十秒内就重新发一次注册请求，这在 NAT 的网络中有助于保持连接。

- Allow 是说明 alice 的 UA 所能支持的功能，某些 UA 功能丰富，而某些 UA 仅有有限的功能。
- User-Agent 是 UA 的型号。
- Allow-Events 则是说明她允许哪些事件通知。
- Content-Length 是消息体（Body）的长度，在这里，只有消息头（Header），没有消息体，因此长度为 0。

FreeSWITCH 需要验证 alice 的身份才允许她注册。在 SIP 中，没有发明新的认证方式，而是使用已有的 HTTP 摘要（Digest）方式来认证。这里它给alice发送401消息。

```

1 -----
2 send 664 bytes to udp/[192.168.4.4]:5090 at 12:30:57.919364:
3 -----
4 SIP/2.0 401 Unauthorized
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-d9ed3bbae47e568b-1---d8754z-rport=5090
6 From: <sip:alice@192.168.4.4;transport=UDP>;tag=9c709222
7 To: <sip:alice@192.168.4.4;transport=UDP>;tag=QFXyg6gcByvUH
8 Call-ID: NmFjNzA3MWY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY20GE.
9 CSeq: 1 REGISTER
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
12     NOTIFY, PUBLISH, SUBSCRIBE
13 Supported: timer, precondition, path, replaces
14 WWW-Authenticate: Digest realm="192.168.4.4",
15     nonce="62fb812c-71d2-4a36-93d6-e0008e6a63ee", algorithm=MD5, qop="auth"
16 Content-Length: 0

```

401 消息表示未认证，它是FreeSWITCH对alice请求的响应。同时，它在本端生成一个认证摘要（WWW-Authenticate），一齐发送给 alice。

```

1 -----
2 recv 846 bytes from udp/[192.168.4.4]:5090 at 12:30:57.921011:
3 -----
4 REGISTER sip:192.168.4.4;transport=UDP SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-dae1693be9f8c10d-1---d8754z-rport
6 Max-Forwards: 70
7 Contact: <sip:alice@192.168.4.4:5090;rinstance=d42207a765c0626b;transport=UDP>
8 To: <sip:alice@192.168.4.4;transport=UDP>
9 From: <sip:alice@192.168.4.4;transport=UDP>;tag=9c709222
10 Call-ID: NmFjNzA3MwY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY20GE.
11 CSeq: 2 REGISTER
12 Expires: 3600
13 Allow: INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO, SUBSCRIBE
14 User-Agent: Zoiper rev.5415
15 Authorization: Digest username="alice",realm="192.168.4.4",
16     nonce="62fb812c-71d2-4a36-93d6-e0008e6a63ee",
17     uri="sip:192.168.4.4;transport=UDP",response="32b5ddaea8647a3becd25cb84346b1c3",
18     cnonce="b4c6ac7e57fc76b85df9440994e2ede8",nc=00000001,qop=auth,algorithm=MD5
19 Allow-Events: presence
20 Content-Length: 0

```

alice 收到带有摘要的 401 后，重新发起注册请求，这一次，加上了根据收到的摘要和它自己的密码生成的认证信息（Authorization）。并且，你可能已经注意到，CSeq 序号变成了 2。

```

1 -----
2 send 665 bytes to udp/[192.168.4.4]:5090 at 12:30:57.936940:
3 -----
4 SIP/2.0 200 OK
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-dae1693be9f8c10d-1---d8754z-rport=5090
6 From: <sip:alice@192.168.4.4;transport=UDP>;tag=9c709222
7 To: <sip:alice@192.168.4.4;transport=UDP>;tag=rrpQj11F86jeD
8 Call-ID: NmFjNzA3MwY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY20GE.
9 CSeq: 2 REGISTER
10 Contact: <sip:alice@192.168.4.4:5090;rinstance=d42207a765c0626b;transport=UDP>;expires=3600
11 Date: Tue, 27 Apr 2010 12:30:57 GMT
12 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
13 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
14     NOTIFY, PUBLISH, SUBSCRIBE
15 Supported: timer, precondition, path, replaces
16 Content-Length: 0

```

FreeSWITCH 收到带有认证的注册消息后，核实 alice 身份，认证通过，回应 200 OK。如果失败，则回应 403 Forbidden 或其它失败消息，如下。

```

1 -----
2 send 542 bytes to udp/[192.168.4.4]:5090 at 13:22:49.195554:
3 -----
4 SIP/2.0 403 Forbidden
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-d447f43b66912a1b-1--d8754z;rport=5090
6 From: <sip:alice@192.168.4.4;transport=UDP>;tag=c097e17f
7 To: <sip:alice@192.168.4.4;transport=UDP>;tag=yeecX364pvryj
8 Call-ID: ZjkxMGJmMjE4Y2ZiNjU5MzM5NDZkMTE5NzMzM0Mjc.
9 CSeq: 2 REGISTER
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
12 NOTIFY, PUBLISH, SUBSCRIBE
13 Supported: timer, precondition, path, replaces
14 Content-Length: 0

```

你可以看到，alice 的密码是不会直接在 SIP 中传送的，因而一定程度上保证了安全（当然还是会有中间人，重放之类的攻击，我们会留到后面讨论）。

4.3 SIP 呼叫流程

4.3.1 UA 间直接呼叫

上面我们说过，SIP 的 UA 是平等的，如果一方知道另一方的地址，就可以通信。我们先来做一个实验。在笔者的机器上，我启动了两个软电话（UA），一个是 bob 的 X-Lite（左），另一个是 alice 是 Zoiper。它们的 IP 地址都是 192.168.4.4，而端口号分别是 26000 和 5090，当 bob 呼叫 alice 时，它只需直接呼叫 alice 的 SIP 地址：sip:alice@192.168.4.4:5090。如图，alice 的电话正在振铃：

详细的呼叫流程图为：



图 4.3: SIP UA间直接呼叫

```

1      bob           alice
2      |           |
3      |   INVITE alice@example.com
4      |----->|
5      |   100 Trying    |
6      |<-----|
7      |   180 Ringing   |
8      |<-----|
9      |   200 OK         |
10     |<-----|
11     |   ACK           |
12     |----->|
13     |
14     |<=====RTP=====>|
15     |
16     |   BYE           |
17     |<-----|
18     |   200 OK         |
19     |----->|
20

```

首先 bob 向 alice 发送 INVITE 请求建立 SIP 连接，alice 的 UA 回 100 Trying 说我收到你的请求了，先等会，接着 alice 的电话开始振铃，并给对方回消息 180 Ringing 说我这边已经振铃了，alice 一会就过来接电话，bob 的 UA 收到该消息后可以播放回铃音。接着 alice 接了电话，她发送 200 OK 消息给 bob，该消息是对 INVITE 消息的最终响应，而先前的 100 和 180 消息都是临时状态，只是表明呼叫进展的情况。bob 收到 200 后向 alice 回 ACK 证实消息。INVITE - 200 - ACK 完成三次握手，它们合在一起称作一个对话（Dialogue）。这时候 bob 已经在跟 alice 能通话了，他们通话的内容（语音数据）是在SIP之外的 RTP 包中传递的，我们后面再详细讨论。

最后，alice 挂断电话，向 bob 送 BYE 消息，bob 收到 BYE 后回送 200 OK，通话完毕。其中 BYE 和 200 OK 也是一个对话，而上面的所有消息，称作一个会话（Session）。

反过来也一样，alice 可以直接呼叫 bob 的地址：sip:bob@192.168.4.4:26000。

上面描述了一个最简单的 SIP 呼叫流程。实际上，SIP 还有其它一些消息，它们大致可分为请求和响应两类。请求由 UAC 发出，到达 UAS 后，UAS 回送响应消息。某些响应消息需要证实（ACK），以完成三次握手。其中请求消息包括 INVITE、ACK、OPTIONS、BYE、CANCEL、REGISTER 以及一些扩展 re-INVITE、PRACK、SUBSCRIBE、NOTIFY、UPDATE、MESSAGE、REFER 等。而响应消息则都包含一个状态码。与 HTTP 响应类似，状态码有三位数字组成。其中，1xx 组的响应为临时状态，表明呼叫进展的情况；2xx 表明请求已成功处理；3xx 表明 SIP 请求需要转向到另一个 UAS 处理；4xx 表明请求失败，这种失败一般是由客户端或网络引起的，如密码错误等；5xx 为服务器内部错误；6xx 为更严重的错误。

4.3.2 通过 B2BUA 呼叫

在真实世界中，bob 和 alice 肯定要经常改变位置，那么它们的 SIP 地址也会相应改变，并且，如果他们之中有一个或两个处于 NAT 的网络中时，直接通信就更困难了。所以，他们通常会借助于一个服务器来相互通信。通过注册到服务器上，他们都可以获得一个服务器上的 SIP 地址。注册服务器的地址一般是不变的，因此他们的 SIP 地址就不会发生变化，因而，他们总是能够进行通信。

我们让他们两个都注册到 FreeSWITCH 上。上面已经说过，FreeSWITCH 监听的端口是 SIP 默认的端口 5060。bob 和 alice 注册后，他们分别获得了一个服务器的地址（SIP URI）：sip:bob@192.168.4.4 和 sip:alice@192.168.4.4（默认的端口号 5060 可以省略）。

下面是 bob 呼叫 alice 的流程。需要指出，如果 bob 只是发起呼叫而不接收呼叫，他并不需要向 FreeSWITCH 注册（有些软交换服务器需要先注册才能发起呼叫，但 SIP 是不强制这么做的）。

```

1 -----
2 recv 1118 bytes from udp/[192.168.4.4]:26000 at 13:31:39.938891:
3 -----
4 INVITE sip:alice@192.168.4.4 SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---d8754z;rport
6 Max-Forwards: 70
7 Contact: <sip:bob@192.168.4.4:26000>
8 To: "alice"<sip:alice@192.168.4.4>
9 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
10 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
11 CSeq: 1 INVITE
12 Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
13 Content-Type: application/sdp
14 User-Agent: X-Lite release 1014k stamp 47051
15 Content-Length: 594

```

上面的消息中省略了 SDP 的内容，我们将留到以后再探讨。bob 的 UAC 通过 INVITE 消息向 FreeSWITCH 发起请求。bob 的 UAC 用的是 X-Lite (User-Agent)，它运行在端口 26000 上（由 Contact 字段给出。实际上，它默认在端口也是 5060，但由于在我的实验环境下它也是跟 FreeSWITCH 运行在一台机器上，已被占用，因此它需要选择另一个端口）。其中，From 为主叫用户的地址，To 为被叫用户的地址。此时 FreeSWITCH 作为一个 UAS 接受请求并进行响应。它得知 bob 要呼叫 alice，需要在自己的数据库中查找 alice 是否已在服务器上注册，好知道应该怎么找到 alice。但在此之前，它先通知 bob 它已经收到了他的请求。

```

1 -----
2 send 345 bytes to udp/[192.168.4.4]:26000 at 13:31:39.940278:
3 -----
4 SIP/2.0 100 Trying
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---d8754z-;rport=26000
6 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
7 To: "alice"<sip:alice@192.168.4.4>
8 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
9 CSeq: 1 INVITE
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Content-Length: 0

```

FreeSWITCH 通过 100 Trying 消息告诉 bob “我已经收到你的消息了，别着急，我正在联系 alice 呢...”该消息称为呼叫进展消息。

```

1 -----
2 send 826 bytes to udp/[192.168.4.4]:26000 at 13:31:39.943392:
3 -----
4 SIP/2.0 407 Proxy Authentication Required
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---d8754z-;rport=26000
6 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
7 To: "alice" <sip:alice@192.168.4.4>;tag=B4pem31jHgtHS
8 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
9 CSeq: 1 INVITE
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Accept: application/sdp
12 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
13 NOTIFY, PUBLISH, SUBSCRIBE
14 Supported: timer, precondition, path, replaces
15 Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
16 include-session-description, presence.winfo, message-summary, refer
17 Proxy-Authenticate: Digest realm="192.168.4.4",
18 nonce="31c5c3e0-cc6e-46c8-a661-599b0c7f87d8", algorithm=MD5, qop="auth"
19 Content-Length: 0

```

但就在此时，FreeSWITCH 发现 bob 并不是授权用户，因而它需要确认 bob 的身份。它通过发送带有 Digest 验证信息的 407 消息来通知 bob（注意，这里与注册流程中的 401 不同）。

```

1 -----
2 recv 319 bytes from udp/[192.168.4.4]:26000 at 13:31:39.945314:
3 -----
4 ACK sip:alice@192.168.4.4 SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---d8754z-;rport
6 To: "alice" <sip:alice@192.168.4.4>;tag=B4pem31jHgtHS
7 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
8 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
9 CSeq: 1 ACK
10 Content-Length: 0

```

bob 回送 ACK 证实消息向 FreeSWITCH 证实已收到认证要求。并重新发送 INVITE，这次，附带了验证信息。

```

1 -----
2 recv 1376 bytes from udp/[192.168.4.4]:26000 at 13:31:39.945526:
3 -----
4 INVITE sip:alice@192.168.4.4 SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---d8754z;rport
6 Max-Forwards: 70
7 Contact: <sip:bob@192.168.4.4:26000>
8 To: "alice"<sip:alice@192.168.4.4>
9 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
10 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
11 CSeq: 2 INVITE
12 Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
13 Content-Type: application/sdp
14 Proxy-Authorization: Digest username="bob",realm="192.168.4.4",
15     nonce="31c5c3e0-cc6e-46c8-a661-599b0c7f87d8",
16     uri="sip:alice@192.168.4.4",response="327887635344405bcd545da06763c466",
17     cnonce="c164b74f625ff2161bd8d47dba3a0ee2",nc=00000001,qop=auth,
18     algorithm=MD5
19 User-Agent: X-Lite release 1014k stamp 47051
20 Content-Length: 594

```

这里也省略了 SDP 消息体。

```

1 -----
2 send 345 bytes to udp/[192.168.4.4]:26000 at 13:31:39.946349:
3 -----
4 SIP/2.0 100 Trying
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---d8754z;rport=26000
6 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
7 To: "alice"<sip:alice@192.168.4.4>
8 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
9 CSeq: 2 INVITE
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Content-Length: 0

```

FreeSWITCH 重新回 100 Trying，告诉 bob 呼叫进展情况。

至此，bob 与 FreeSWITCH 之间的通信已经初步建立，这种通信的通道称作一个信道（channel）。该信道是由 bob 的 UA 和 FreeSWITCH 的一个 UA 构成的，我们称它为 FreeSWITCH 的一条腿，叫做 a-leg。

接下来 FreeSWITCH 要建立另一条腿，称为 b-leg。它通过查找本地数据库，得到了 alice 的位置，接着启动一个 UA（用作 UAC），向 alice 发送 INVITE 消息。如下：

```

1 -----
2 send 1340 bytes to udp/[192.168.4.4]:5090 at 13:31:40.028988:
3 -----
4 INVITE sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4;rport;branch=z9hG4bKey90QUyHZQXNN
6 Route: <sip:alice@192.168.4.4:5090>;rinstance=e7d5364c81f2b879;transport=UDP
7 Max-Forwards: 69
8 From: "Bob" <sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
9 To: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
10 Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
11 CSeq: 130069214 INVITE
12 Contact: <sip:mod_sofia@192.168.4.4:5060>
13 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
14 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
15 NOTIFY, PUBLISH, SUBSCRIBE
16 Supported: timer, precondition, path, replaces
17 Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
18 include-session-description, presence.winfo, message-summary, refer
19 Content-Type: application/sdp
20 Content-Disposition: session
21 Content-Length: 313
22 X-FS-Support: update_display
23 Remote-Party-ID: "Bob" <sip:bob@192.168.4.4>;party=calling;screen=yes;privacy=off

```

你可以看到，该INVITE 的 Call-ID 与前面的不同，说明这是另一个 SIP 会话（Session）。另外，它还多了一个 Remote-Party-ID，它主要是用来支持来电显示。因为，在 alice 的话机上，希望显示 bob 的号码，显示呼叫它的那个 UA（负责 b-leg的那个 UA）没什么意义。与普通的 POTS 电话不同，在 SIP 电话中，不仅能显示电话号码（这里是 bob），还能显示一个可选的名字（“Bob”）。这也说明了 FreeSWITCH 这个 B2BUA 本身是一个整体，它虽然是以一个单独的 UA 呼叫 alice，但还是跟负责 bob 的那个 UA 有联系—就是这种背靠背的串联。

```

1 -----
2 recv 309 bytes from udp/[192.168.4.4]:5090 at 13:31:40.193634:
3 -----
4 SIP/2.0 100 Trying
5 Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKey90QUyHZQXNN
6 To: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
7 From: "Bob" <sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
8 Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
9 CSeq: 130069214 INVITE
10 Content-Length: 0

```

跟上面的流程差不多，alice 回的呼叫进展。此时，alice 的 UA 开始振铃。

```

1 -----
2 recv 431 bytes from udp/[192.168.4.4]:5090 at 13:31:40.193816:
3 -----
4 SIP/2.0 180 Ringing
5 Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKey90QUyHZQXNN
6 Contact: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
7 To: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
8 From: "Bob"<sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
9 Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
10 CSeq: 130069214 INVITE
11 User-Agent: Zoiper rev.5415
12 Content-Length: 0

```

180也是呼叫进展消息，它说明，我这边已经准备好了，alice 的电话已经响了，她听到了一会就会接听。

```

1 send 1125 bytes to udp/[192.168.4.4]:26000 at 13:31:40.270533:
2 -----
3 SIP/2.0 183 Session Progress
4 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---d8754z;rport=26000
5 From: "Bob"<sip:bob@192.168.4.4>;tag=15c8325a
6 To: "alice" <sip:alice@192.168.4.4>;tag=cDg7NyjpeSg4m
7 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
8 CSeq: 2 INVITE
9 Contact: <sip:alice@192.168.4.4:5060;transport=udp>
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Accept: application/sdp
12 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
13 NOTIFY, PUBLISH, SUBSCRIBE
14 Supported: timer, precondition, path, replaces
15 Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
16 include-session-description, presence.winfo, message-summary, refer
17 Content-Type: application/sdp
18 Content-Disposition: session
19 Content-Length: 267
20 Remote-Party-ID: "alice" <sip:alice@192.168.4.4>

```

FreeSWITCH 在收到 alice 的 180 Ringing 消息后，便告诉 bob 呼叫进展情况，183 与 180 不同的是，它包含 SDP，即接下来它会向 bob 发送 RTP 的媒体流，就是回铃音。

```

1 -----
2 recv 768 bytes from udp/[192.168.4.4]:5090 at 13:31:43.251980:
3 -----
4 SIP/2.0 200 OK
5 Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKey90QUyHZQXNN
6 Contact: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
7 To: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
8 From: "Bob" <sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
9 Call-ID: Od74ac35-cca4-122d-81a2-2990e5b2bd3e
10 CSeq: 130069214 INVITE
11 Allow: INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO, SUBSCRIBE
12 Content-Type: application/sdp
13 User-Agent: Zoiper rev.5415
14 Content-Length: 226

```

alice 接听电话以后，其 UA 向 FreeSWITCH 送 200 OK，即应答消息。

```

1 -----
2 send 436 bytes to udp/[192.168.4.4]:5090 at 13:31:43.256692:
3 -----
4 ACK sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4;rport;branch=z9hG4bKF72SSpFNv0K8g
6 Max-Forwards: 70
7 From: "Bob" <sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
8 To: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
9 Call-ID: Od74ac35-cca4-122d-81a2-2990e5b2bd3e
10 CSeq: 130069214 ACK
11 Contact: <sip:mod_sofia@192.168.4.4:5060>
12 Content-Length: 0

```

FreeSWITCH 向 alice 回送证实消息，证实已经知道了。至此，b-leg 已经完全建立完毕，多半这时 alice 已经开始说话了：“Hi, bob, 你好.....”

```

1 -----
2 send 1135 bytes to udp/[192.168.4.4]:26000 at 13:31:43.293311:
3 -----
4 SIP/2.0 200 OK
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---d8754z-;rport=26000
6 From: "Bob" <sip:bob@192.168.4.4>;tag=15c8325a
7 To: "alice" <sip:alice@192.168.4.4>;tag=cDg7NyjpeSg4m
8 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
9 CSeq: 2 INVITE
10 Contact: <sip:alice@192.168.4.4:5060;transport=udp>
11 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
12 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
13 NOTIFY, PUBLISH, SUBSCRIBE
14 Supported: timer, precondition, path, replaces
15 Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
16 include-session-description, presence.winfo, message-summary, refer
17 Session-Expires: 120;refresher=uas
18 Min-SE: 120
19 Content-Type: application/sdp
20 Content-Disposition: session
21 Content-Length: 267
22 Remote-Party-ID: "alice" <sip:alice@192.168.4.4>
```

与此同时，它也给 bob 送应答消息，告诉他电话已经接通了，可以跟 alice 说话了。在需要计费的情况下，应该从此时开始对 bob 的电话计费。bob 的 UA 收到该消息后启动麦克风让 bob 讲话。

```

1 -----
2 recv 697 bytes from udp/[192.168.4.4]:26000 at 13:31:43.413025:
3 -----
4 ACK sip:alice@192.168.4.4:5060;transport=udp SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-ef53864320037c04-1---d8754z-;rport
6 Max-Forwards: 70
7 Contact: <sip:bob@192.168.4.4:26000>
8 To: "alice" <sip:alice@192.168.4.4>;tag=cDg7NyjpeSg4m
9 From: "Bob" <sip:bob@192.168.4.4>;tag=15c8325a
10 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
11 CSeq: 2 ACK
12 Proxy-Authorization: Digest username="bob",realm="192.168.4.4",
13 nonce="31c5c3e0-cc6e-46c8-a661-599b0c7f87d8",
14 uri="sip:alice@192.168.4.4",response="327887635344405bcd545da06763c466",
15 cnonce="c164b74f625ff2161bd8d47dba3a0ee2",nc=00000001,qop=auth,
16 algorithm=MD5
17 User-Agent: X-Lite release 1014k stamp 47051
18 Content-Length: 0
```

bob 在收到应答消息后也需要回送证实消息。至此 a-leg 也建立完毕。双方正常通话。

████████ 此处省略 5000 字 ... █████

```

1 -----
2 recv 484 bytes from udp/[192.168.4.4]:5090 at 13:31:49.949240:
3 -----
4 BYE sip:mod_sofia@192.168.4.4:5060 SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-2146ae0ddd113efe-1---d8754z-;rport
6 Max-Forwards: 70
7 Contact: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
8 To: "Bob"<sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
9 From: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
10 Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
11 CSeq: 2 BYE
12 User-Agent: Zoiper rev.5415
13 Content-Length: 0

```

终于聊完了， alice 挂断电话，发送 BYE 消息。

```

1 -----
2 send 543 bytes to udp/[192.168.4.4]:5090 at 13:31:49.950425:
3 -----
4 SIP/2.0 200 OK
5 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-2146ae0ddd113efe-1---d8754z-;rport=5090
6 From: <sip:alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
7 To: "Bob"<sip:bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
8 Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
9 CSeq: 2 BYE
10 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
11 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
12 NOTIFY, PUBLISH, SUBSCRIBE
13 Supported: timer, precondition, path, replaces
14 Content-Length: 0

```

FreeSWITCH 返回 OK， b-leg 释放完毕。

```

1 -----
2 send 630 bytes to udp/[192.168.4.4]:26000 at 13:31:50.003165:
3 -----
4 BYE sip:bob@192.168.4.4:26000 SIP/2.0
5 Via: SIP/2.0/UDP 192.168.4.4;rport;branch=z9hG4bKggvjuH0rS99tc
6 Max-Forwards: 70
7 From: "alice" <sip:alice@192.168.4.4>;tag=cDg7NyjpeSg4m
8 To: "Bob" <sip:bob@192.168.4.4>;tag=15c8325a
9 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
10 CSeq: 130069219 BYE
11 Contact: <sip:alice@192.168.4.4:5060;transport=udp>
12 User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
13 Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
14 NOTIFY, PUBLISH, SUBSCRIBE
15 Supported: timer, precondition, path, replaces
16 Reason: Q.850;cause=16;text="NORMAL_CLEARING"
17 Content-Length: 0

```

接下来 FreeSWITCH 给 bob 发送 BYE，通知要拆线了。出于对 bob 负责，它包含了挂机原因（Reason: Hangup Cause），此处 NORMAL_CLEARING 表示正常释放。

```

1 -----
2 recv 367 bytes from udp/[192.168.4.4]:26000 at 13:31:50.111765:
3 -----
4 SIP/2.0 200 OK
5 Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKggvjuH0rS99tc
6 Contact: <sip:bob@192.168.4.4:26000>
7 To: "Bob" <sip:bob@192.168.4.4>;tag=15c8325a
8 From: "alice" <sip:alice@192.168.4.4>;tag=cDg7NyjpeSg4m
9 Call-ID: YWEwYjN1ZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
10 CSeq: 130069219 BYE
11 User-Agent: X-Lite release 1014k stamp 47051
12 Content-Length: 0

```

bob 回送 OK，a-leg 释放完毕，通话结束。从下图可以很形象地看出 FreeSWITCH 的两条“腿”— a-leg 和 b-leg。

整个呼叫流程图示如下：

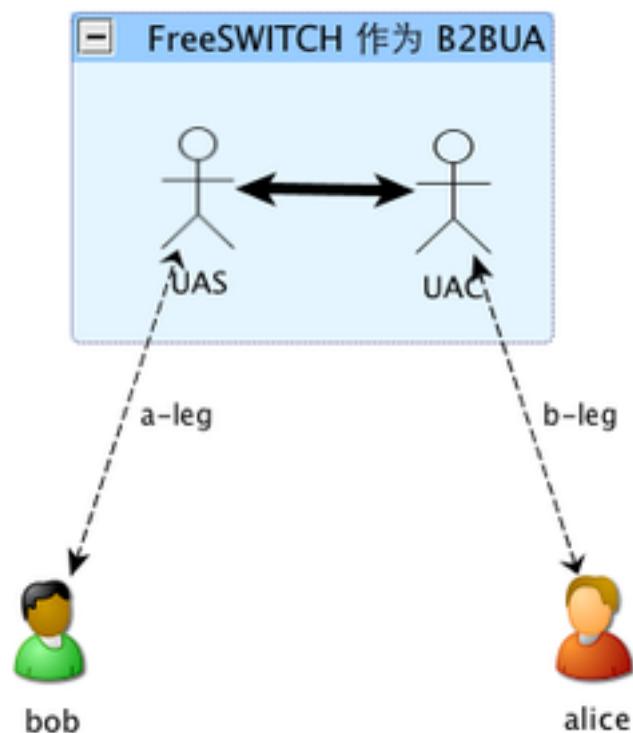


图 4.4: B2BUA的两条腿

```

1      bob (UAC)          [ UAS-UAC ]          (UAS) alice
2      |                   |   |                   |
3      |   INVITE          |   |                   |
4      |----->|   |   |
5      |   100 Trying       |   |                   |
6      |<-----|   |   |
7      |   407 Authentication Required   |
8      |<-----|   |   |
9      |   ACK              |   |                   |
10     |----->|   |   |
11     |   INVITE          |   |                   |
12     |----->|   |   |
13     |   100 Trying       |   |   INVITE
14     |<-----<   >----->|
15     |           |   |   100 Trying
16     |           |   |<-----|
17     |   183 Progress      |   |   180 Ringing
18     |<-----<   >----->|
19     |           |   |   200 OK
20     |           |   |<-----|
21     |   200 OK            |   |   ACK
22     |<-----<   >----->|
23     |   ACK              |   |                   |
24     |----->|   |                   |
25
26     |           Call Connected, talking ...
27
28     |           |   |   BYE
29     |           |   |<-----|
30     |   BYE              |   |   200 OK
31     |<-----<   >----->|
32     |   200 OK            |   |                   |
33
34     |----->|   |                   |

```

从流程图可以看出，右半部分跟上一节“UA间直接呼叫”一样，而左半部分也类似。这就更好的说明了实际上有 4 个 UA（两对）参与到了通信中，并且，有两个 Session。

4.4 再论 SIP URI

上面我们介绍了一些 FreeSWITCH 的基本概念，并通过一个真正的呼叫流程讲解了一下 SIP。由于实验中所有 UA 都运行在一台机器上，这可能会引起迷惑，如果我们有三台机器，其中 192.168.0.1 是 FreeSWITCH 服务器，而 bob 和 alice 分别在另外两台机器上，那么情况可能是：

```
1      /-----\
2      |   FreeSWITCH   |
3      |   192.168.0.1   |
4      \-----/
5  sip:bob@192.168.0.1   /           \    sip:alice@192.168.0.1
6          /           \
7          /           \
8      /-----\           /-----\
9      |   bob        |           |   alice       |
10     |   192.168.0.100 |           |   192.168.0.200 |
11     \-----/           \-----/
12
13  sip:bob@192.168.0.100           sip:alice@192.168.0.200
```

alice 注册到 FreeSWITCH, bob 呼叫她时, 使用她的服务器地址, 即 `sip:alice@192.168.0.1`, FreeSWITCH 接到请求后, 查找本地数据库, 发现 alice 的实际地址 (Contact) 是 `sip:alice@192.168.0.200`, 便可以建立呼叫。

SIP URI 除使用 IP 地址外, 也可以使用域名, 如 `sip:alice@freeswitch.org.cn`。更高级及更复杂的配置则可能需要 DNS 的 SRV 记录, 在此就不做讨论了。

第五章 FreeSWITCH架构

从本章开始，我们正式开始我们的 FreeSWITCH 之旅。今后我们不再用单独的章节来讲述VoIP中的其它要素和概念，而是在用到时穿插于各个章节之中。

5.1 总体结构

FreeSWITCH 由一个稳定的核心及外围模块组成，如图1：

FreeSWITCH 使用线程模型来处理并发请求，每个连接都在单独的线程中进行处理。这不仅能提供最大强度的并发，更重要的是，即使某路电话发生问题，也只影响到它所在的线程，而不会影响到其它电话。FreeSWITCH 的核心非常短小精悍，这也是保持稳定的关键。所有其它功能都在外围的模块中。模块是可以动态加载（以及卸载）的，在实际应用中可以只加载用到的模块。外围模块通过核心提供的 Public API 与核心进行通信，而核心则通过回调机制执行外围模块中的代码。

5.2 核心

FS Core 是 FreeSWITCH 的核心，它包含了关键的数据结构和复杂的代码，但这些代码只出现在核心中，并保持了最大限度的重用。外围模块只能通过 API 调用核心的功能，因而核心运行在一个受保护的环境中，核心代码都经过精心的编码和严格的测试，最大限度地保持了系统整体的稳定。

核心代码保持了最高度的抽象，因而它可以调用不同功能，不同协议的模块。同时，良好的 API 也使得编写不同的外围模块非常容易。

5.3 数据库

FreeSWITCH 的核心除了使用内部的队列、哈希表存储数据外，也使用外部的 SQL 数据库存储数据。当前，系统的核心数据库使用 SQLite，默认的存储位置是 db/core.db 。 使用外部数

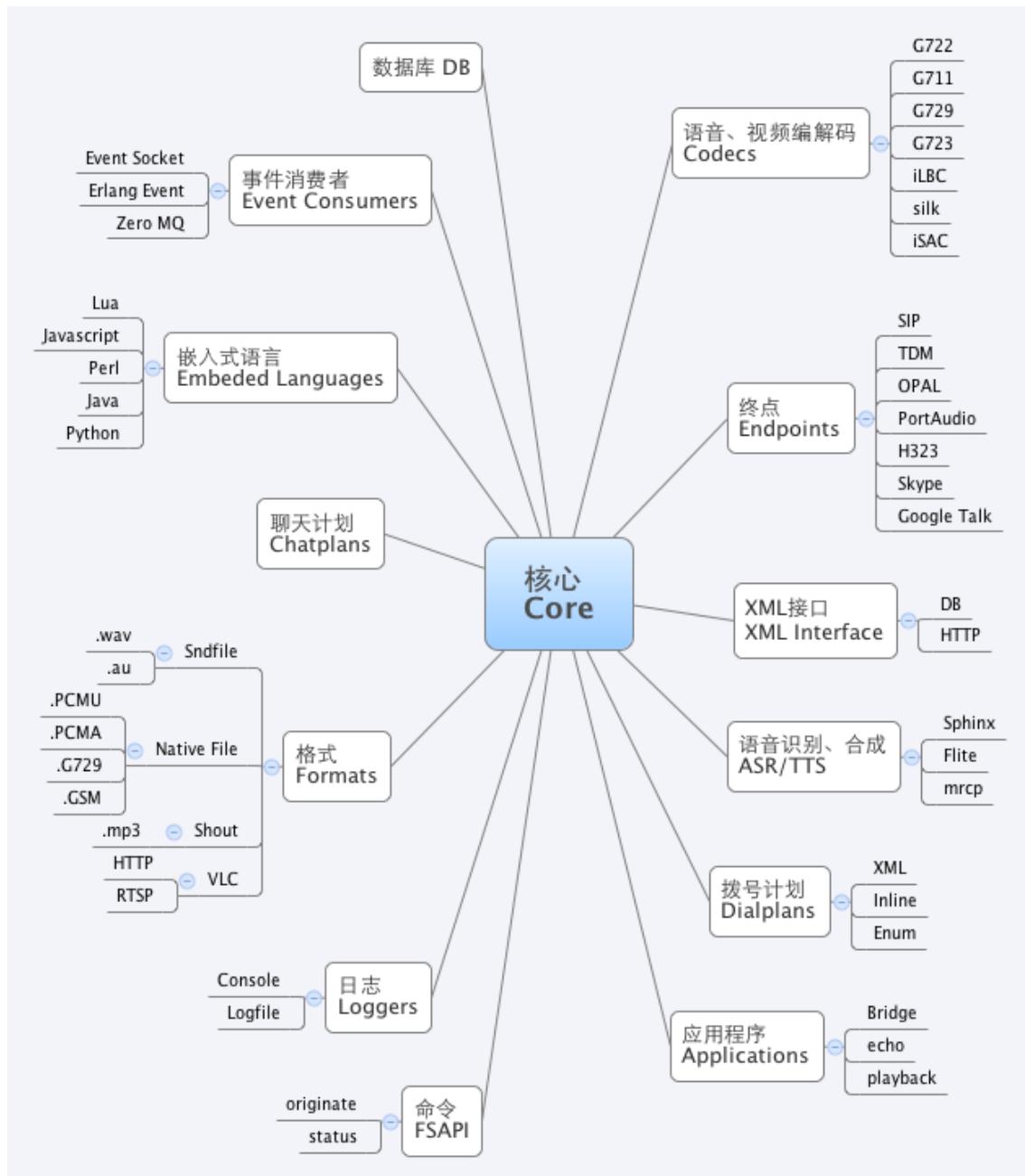


图 5.1: FreeSWITCH架构

数据库的好处是—查询数据不用锁定内存数据结构，这不仅能提高性能，而且降低了死锁的风险，保证了系统稳定。命令 show calls、show channels 等都是直接从数据库中读取内容并显示的。由于 SQLite 会进行读锁定，因此不建议直接读取核心数据库。

系统对数据库操作做了优化，在高并发状态时，核心会尽量将几百条 SQL 一齐执行，这大大提高了性能。但在低并发的状态下执行显得稍微有点慢，如一个 channel 已经建立了，但还不能在 show channels 中显示；或者，一个 channel 已经 destroy 了，还显示在 show channels 中。但由于这些数据只用于查询，而不用于决策，所以一般没什么问题。

除核心数据库外，系统也支持使用 ODBC 方式连接其它数据库，如 PostgreSQL、MySQL 等。某些模块，如 mod_sofia、mod_fifo 等都有自己的数据库（表）。如果在 *nix 类系统上使用 ODBC，需要安装 UnixODBC，并进行正确的配置，如果编译安装的话还需要开发包 unixodbc-devel（CentOS）或 unixodbc-dev（Debian/Ubuntu）。由于 PostgreSQL、MySQL 等都是 Client-Server 的结构，因此，外部程序可以直接查询数据（但需要清楚数据的准确性，可能比 FreeSWITCH 核心中的数据有所滞后）。

5.4 模块

FreeSWITCH 主要分为以下几个部分：

5.4.1 终点

Endpoint 是终结 FreeSWITCH 的地方，也就是说再往外走就超出 FreeSWITCH 的控制了。它主要包含了不同呼叫控制协议的接口，如 SIP、TDM 硬件、H323 以及 Google Talk 等。这使得 FreeSWITCH 可以与众多不同的电话系统进行通信。如，可以使用 mod_skypopen 与 Skype 网络进行通信。另外，前面也讲过，它还可以通过 portaudio 驱动本地声卡，用作一个软电话。

5.4.2 拨号计划

Dialplan 主要是为了查找电话路由，主要的是 XML 描述的，但它也支持 Asterisk 格式的配置文件。另外它也支持 ENUM 查询。

5.4.3 聊天计划

类似于 Dialplan，Chatplan 主要是对 Message 进行路由，如 SIP SIMPLE、Skype Message、XMPP Message 等。

5.4.4 应用程序

FreeSWITCH 提供了许多应用程序使用复杂的任务变得异常简单，如 mod_voicemail 模块可以很简单的实现语音留言；而 mod_onference 模块则可以实现高质量的多方会议。

5.4.5 应用程序接口（FSAPI）

FSAPI 是一种命令接口，它的原理非常简单——输入一个简单的字符串（以空格分隔），该字符串由模块的内部函数处理，然后得到一个输出。输出可以是一个简单的字符串，一大串文本，以及 XML 及 JSON 文本。通过使用 FSAPI，一个模块可以调用另一个模块的功能，而不用链接其它模块的函数（或代码）。它实际上是一种高度抽象的输入-输出机制，不仅在模块内部，在 FreeSWITCH 外部也可能通过 Event Socket（我们第二章中说的 fs_cli 就是用这种方式）或 XML-RPC 等进行调用。

5.4.6 XML 接口

XML Interface 支持多种获取 XML 配置的方式，它可以是本地的配置文件，或从数据库中读取，甚至是一个能动态返回 XML 的远程 HTTP 服务器。

5.4.7 编解码器

FreeSWITCH 支持最广泛的 Codec，除了大多数 VoIP 系统支持的 G711、G722、G729、GSM 外，它还支持 iLBC，BV16/32、SILK、iSAC，CELT 等。它可以同时桥接不同采样频率的电话，以及电话会议等。

5.4.8 语音识别及语音合成

支持语音自动识别（ASR）及文本-语音转换（TTS）。

5.4.9 文件格式

支持不同的声音文件格式，如 wav，mp3 等。

5.4.10 日志

日志可以写到控制台、日志文件、系统日志（syslog）以及远程的日志服务器。

5.4.11 嵌入式语言

通过 swig 包装支持多种脚本语本语言控制呼叫流程，如 Lua、Javascript、Perl 等。

5.4.12 事件套接字

使用 Event Socket 可以使用任何其它语言通过 Socket 方式控制呼叫流程、扩展 FreeSWITCH 功能。

5.5 目录结构

在 *nix 类系统上，FreeSWITCH 默认的安装位置是 /usr/local/freeswitch，在 Windows 上可能是 C:\freeswitch，目录结构大致相同。

1	bin	可执行程序
2	db	系统数据库 (sqlite)，将呼叫信息存放到数据库里以便在查询时无需对核心数据结构加锁
3	htdocs	HTTP Web server 根目录
4	lib	库文件
5	mod	可加载模块
6	run	运行目录，存放 PID
7	sounds	声音文件，使用 <code>playback()</code> 时默认的寻找路径
8	grammar	语法
9	include	头文件
10	log	日志，CDR 等
11	recordings	录音，使用 <code>record()</code> 时默认的存放路径
12	scripts	嵌入式语言写的脚本，如使用 <code>lua()</code> 、 <code>luarun()</code> 、 <code>jsrun</code> 等默认寻找的路径
13	storage	语言留言 (Voicemail) 的录音
14	conf	配置文件，详见下节

5.6 配置文件

配置文件由许多 XML 文件组成。在系统装载时，XML 解析器会将所有 XML 文件组织在一起，并读入内存，称为 XML 注册表。这种设计的好处在于其非常高的可扩展性。由于 XML 文档本身非常适合描述复杂的数据结构，在 FreeSWITCH 中就可以非常灵活的使用这些数据。并且，外部应用程序也可以很简单地生成 XML，FreeSWITCH 在需要时可以动态的装载这些 XML。另外，系统还允许在某些 XML 节点上安装回调程序(函数)，当这些节点的数据变化时，系统便自动调用这些回调程序。

使用 XML 唯一的不足就是手工编辑这些 XML 比较困难，但正如[其作者所言¹](#)，他绝对不是 XML 的粉丝，但这一缺点与它所带来的好处相比是微不足道的。而且，将来也许会有图形化的配置工具，到时候只有高级用户会去看这些XML了。

5.6.1 目录结构

配置文件的目录结构如下（其中结尾有“/”的为目录）：

```
1 autoload_configs/
2 chatplan/
3 dialplan/
4 directory/
5 extensions.conf
6 freeswitch.xml
7 fur_elise.ttm
8 jingle_profiles/
9 lang/
10 mime.types
11 notify-voicemail.tpl
12 sip_profiles/
13 tetris.ttm
14 vars.xml
15 voicemail.tpl
16 web-vm.tpl
```

其中最重要的是 freeswitch.xml，就是它将所有配置文件“粘”到一起。只要有一点 XML 知识，这些配置是很容易看懂的。其中，X-PRE-PROCESS标签称为预处理命令，它用来设置一些变量和装入其它配置文件。在 XML 加载阶段，FreeSWITCH 的 XML 解析器会先将预处理命令进行展开，生成一个大的 XML 文件 log/freeswitch.xml.fsxml。该文件是一个内存镜像，用户不应该手工编辑它。但它对调试非常有用，假设你不慎弄错了某个标签，又不知道它在哪个地方，FreeSWITCH 在加载时就报 XML 的某一行出错，在该文件中就行容易找到。

整个XML文件分为几个重要的部分：configuration（配置）、dialplan（拨号计划）、directory（用户目录）及phrase（分词）。每一部分又分别装入不同的 XML。

¹ 小知识：XML
2 XML由标签(Tag)和属性构成。`<tag>` 和 `</tag>`组成一对标签，如果该标签有相关属性，则以
3 `<tag attr="value"></tag>` 形式指定。有些标签无须配对，则必须以 `">"`关闭该标签定义，
4 如`<other_tag attr="value"/>`。

¹<http://www.freeswitch.org/node/123>

5.6.2 freeswitch.xml

```

1  <?xml version="1.0"?>
2  <document type="freeswitch/xml">
3      <!-- #comment 这是一个配置文件, 本行是注释 -->
4
5      <X-PRE-PROCESS cmd="include" data="vars.xml"/>
6
7      <section name="configuration" description="Various Configuration">
8          <X-PRE-PROCESS cmd="include" data="autoload_configs/*.xml"/>
9      </section>
10 </document>

```

上面是一个精减了的 freeswitch.xml。它的根是 *document*, 在 *document* 中, 有许多 *section*, 每个 *section* 都对应一部分功能。其中有两个 X-PRE-PROCESS 预处理指令, 它们的作用是将 *data* 参数指定的文件包含 (*include*) 到本文件中来。由于它是一个预处理指令, FreeSWITCH 在加载阶段只对其进行简单替换, 并不进行语法分析, 因此, 对它进行注释是没有效果的, 这是一个新手常犯的错误。假设 vars.xml 的内容如下, 它是一个合法的 XML:

```

1  <!-- this is vars.xml -->
2  <var>xxxxx</var>

```

若你在调试阶段想把一条 X-PRE-PROCESS 指令注释掉:

```

1  <!-- <X-PRE-PROCESS cmd="include" data="vars.xml"/> -->

```

当 FreeSWITCH 预处理时, 还没有到达 XML 解析阶段, 也就是说它还不认识 XML 注释语法, 而仅会机械地将预处理指令替换为 vars.xml 里的内容:

```

1  <!-- <!-- this is vars.xml -->
2  <var>xxxxx</var> -->

```

由于 XML 的注释不能嵌套, 因此便产生错误的 XML。解决办法是破坏掉 X-PRE-PROCESS 的定义, 如我常用下面两种方法:

```

1  <xX-PRE-PROCESS cmd="include" data="vars.xml"/>
2  <XPRE-PROCESS cmd="include" data="vars.xml"/>

```

由于 FreeSWITCH 不认识 xX-PRE-PROCESS 及 XPRE-PROCESS, 因此它会忽略掉该行, 相当于注释掉了。

5.6.3 vars.xml

vars.xml 主要通过 X-PRE-PROCESS 指令定义了一些全局变量。全局变量以 \$\$\{var\} 表示，临时变量以 \${var} 表示。有些变量是系统在运行时自动获取的，如默认情况下 \$\$\{base_dir\}=/usr/local/freeswitch, \$\$\{local_ip_v4\}=你机器的IP地址等。

5.6.4 autoload_configs 目录

autoload_configs 目录下面的各种配置文件会在系统启动时装入。一般来说都是模块级的配置文件，每个模块对应一个。文件名一般以“模块名.conf.xml”方式命名。其中 modules.conf.xml 决定了 FreeSWITCH 启动时自动加载哪些模块。

5.6.5 dialplan 目录

定义 XML 拨号计划，我们会有专门的章节讲解拨号计划。

5.6.6 directory 目录

它里面的配置文本决定了 FreeSWITCH 作为注册服务器时哪些用户可以注册上来。FreeSWITCH 支持多个域（Domain），每个域可以写到一个 XML 文件里。默认的配置包括一个 default.xml，里面定义了 1000 ~ 1019 一共 20 个用户。

5.6.7 sip_profiles

它定义了 SIP 配置文件，实际上它是由 mod_sofia 模块在 autoload_configs/sofia.conf.xml 中加载的。但由于它本身比较复杂又是核心的功能，因此单列了一个目录。我们将会在后面加以详细解释。

5.7 XML 用户目录

XML 用户目录决定了哪些用户可以注册到 FreeSWITCH 上。当然，SIP 并不要求一定要注册才可以打电话，但是用户认证仍需要在用户目录中配置。

用户目录的默认配置文件在 conf/directory/，系统自带的配置文件为 default.xml（其中 dial-string 一行由于排版要求人工换行，实际上不应该有换行）：

```

1 <domain name="$$\{domain\}">
2   <params>
3     <param name="dial-string" value="{presence_id=$\{dialed_user\}@$\{dialed_domain\}}
4       $\{sofia_contact($\{dialed_user\}@$\{dialed_domain\})\}">
5   </params>
6
7   <variables>
8     <variable name="record_stereo" value="true"/>
9     <variable name="default_gateway" value="$$\{default_provider\}">
10    <variable name="default_areacode" value="$$\{default_areacode\}">
11    <variable name="transferFallback_extension" value="operator"/>
12  </variables>
13
14 </domain>

```

该配置文件决定了哪些用户能注册到 FreeSWITCH 中。一般来说，所有用户都应该属于同一个 domain（除非你想使用多 domain，后面我们会有例子）。这里的 \$\$\{domain\} 全局变量是在 vars.xml 中设置的，它默认是主机的 IP 地址，但也可以修改，使用一个域名。params 中定义了该 domain 中所有用户的公共参数。在这里只定义了一个 dial-string，这是一个至关重要的参数。当你在使用 user/user_name 或 sofia/internal/user_name 这样的呼叫字符串时，它会扩展成实际的 SIP 地址。其中 sofia_contact 是一个 API，它会根据用户的注册地址扩展成相应的呼叫字符串。

variables 则定义了一些公共变量，在用户做主叫或被叫时，这些变量会绑定到相应的 Channel 上形成 Channel Variable。

在 domain 中还定义了许多组（group），组里面包含很多用户（user）。

```

1 <groups>
2   <group name="default">
3     <users>
4       <X-PRE-PROCESS cmd="include" data="default/*.xml"/>
5     </users>
6   </group>
7 </groups>

```

在这里，组名 default 并没有什么特殊的意义，它只是随便起的，你可以修改成任何值。在用户标签里，又使用预处理指令装入了 default/ 目录中的所有 XML 文件。你可以看到，在 default/ 目录中，每个用户都对应一个文件。

你也可以定义其它的用户组，组中的用户并不需要是完整的 XML 节点，也可以是一个指向一个已存在用户的“指针”，如下图，使用 type="pointer" 可以定义指针。

```

1   <group name="sales">
2     <users>
3       <user id="1000" type="pointer"/>
4       <user id="1001" type="pointer"/>
5       <user id="1002" type="pointer"/>
6     </users>
7   </group>

```

虽然我们这里设置了组，但使用组并不是必需的。如果你不打算使用组，可以将用户节点(users)直接放到 domain 的下一级。但使用组可以支持像群呼、代接等业务。使用 group_call 可以同时或顺序的呼叫某个组的用户。

实际用户相关的设置也很直观，下面显示了 alice 这个用户的设置：

```

1 <user id="alice">
2   <params>
3     <param name="password" value="$$\{default_password\}" />
4     <param name="vm-password" value="alice" />
5   </params>
6   <variables>
7     <variable name="toll_allow" value="domestic,international,local" />
8     <variable name="accountcode" value="alice" />
9     <variable name="user_context" value="default" />
10    <variable name="effective_caller_id_name" value="Extension 1000" />
11    <variable name="effective_caller_id_number" value="1000" />
12    <variable name="outbound_caller_id_name" value="$$\{outbound_caller_name\}" />
13    <variable name="outbound_caller_id_number" value="$$\{outbound_caller_id\}" />
14    <variable name="callgroup" value="techsupport" />
15  </variables>
16 </user>

```

由上面可以看到，实际上 params 和 variables 可以出现在 user 节点中，也可以出现在 group 或 domain 中。当它们有重复时，优先级顺序为 user, group, domain。

当然，用户目录还有一些更复杂的设置，我们留待以后再做研究。

5.8 呼叫流程及相关概念

再复习一下，FreeSWITCH是一个B2BUA，我们还是以第四章中的图为例：

主要呼叫流程有以下两种：

- bob 向 FreeSWITCH 发起呼叫，FreeSWITCH 接着启动另一个 UA 呼叫 alice，两者通话；
- FreeSWITCH 同时呼叫 bob 和 alice，两者接电话后 FreeSWITCH 将 a-leg 和 b-leg 桥接(bridge)到一起，两者通话。

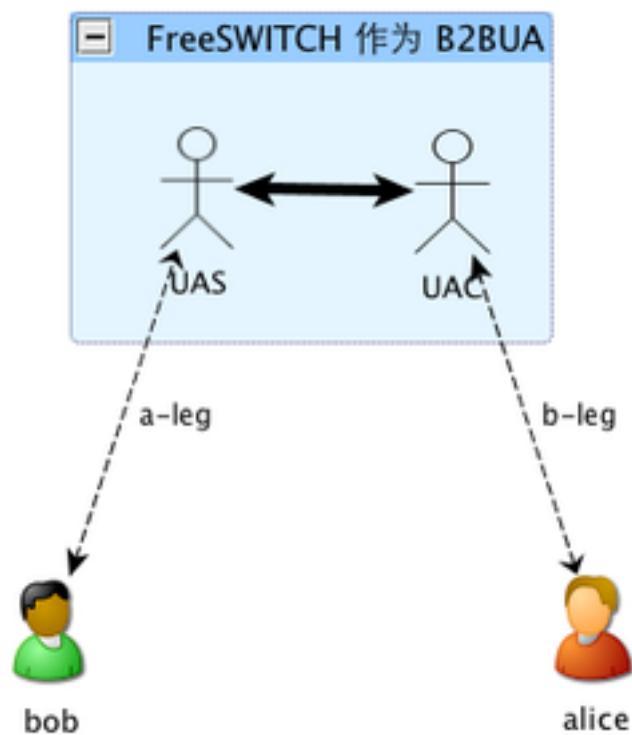


图 5.2: B2BUA

其中第二种又有一种变种。如，市场上有人利用上、下行通话的不对称性卖电话回拨卡获取利润：bob 呼叫 FreeSWITCH，FreeSWITCH 不应答，而是在获取 bob 的主叫号码后直接挂机；然后 FreeSWITCH 回拨 bob；bob 接听后 FreeSWITCH 启动一个 IVR 程序指示 bob 输入 alice 的号码；然后 FreeSWITCH 呼叫 Alice.....

在实际应用中，由于涉及回铃音、呼叫失败等，实际情况要复杂的多。

5.8.1 Session 与 Channel

对每一次呼叫，FreeSWITCH 都会启动一个 Session（会话，它包含SIP会话，SIP会在每对UAC-UAS之间生成一个 SIP Session），用于控制整个呼叫，它会一直持续到通话结束。其中，每个 Session 都控制着一个 Channel（信道），Channel 是一对 UA 间通信的实体，相当于 FreeSWITCH 的一条腿（leg），每个 Channel 都有一个唯一的 UUID。另外，Channel 上可以绑定一些呼叫参数，称为 Channel Variable（信道变量）。Channel 中可能包含媒体（音频或视频流），也可能不包含。通话时，FreeSWITCH 的作用是将两个 Channel（a-leg 和 b-leg，通常先创建的或占主动的叫 a-leg）桥接（bridge）到一起，使双方可以通话。

通话中，媒体（音频或视频）数据流在 RTP 包中传送（不同于 SIP，RTP 是另外的协议）。一般来说，Channel 是双向的，因此，媒体流会有发送（Send/Write）和接收（Receive/Read）两个方向。

5.8.2 回铃音与 Early Media

1 A ----- |a 交换机 | ---X--- | 交换机 b| ----- B

为了便于说明，我们假定A与B不在同一台服务器上（如在PSTN通话中可能不在同一座城市），中间需要经过多级服务器的中转。

假设上图是在 PSTN 网络中，A 呼叫 B，B 话机开始振铃，A 端听回铃音（Ring Back Tone）。在早期，B 端所在的交换机只给 A 端交换机送地址全（ACM）信号，证明呼叫是可以到达 B 的，A 端听到的回铃音铃流是由 A 端所在的交换机生成并发送的。但后来，为了在 A 端能听到 B 端特殊的回铃音（如“您拨打的电话正在通话中...”或“对方暂时不方便接听您的电话”尤其是现代交换机支持各种个性化的彩铃 - Ring Back Color Tone 等），回铃音就只能由 B 端交换机发送。在 B 接听电话前，回铃音和彩铃是不收费的（不收取本次通话费。彩铃费用一般是在 B 端以月租或套餐形式收取的）。这些回铃音就称为 Early Media(早期媒体)。它是由 SIP 的183(带有SDP)消息描述的。

理论上讲，B 接听电话后交换机 b 可以一直不向 a 交换机发送应答消息，而将真正的话音数据伪装成 Early Media，以实现“免费通话”。

5.8.3 Channel Variable

在每一个 Channel 上都有很多属性。这些属性就称为Channel Variable（信道变量）。FreeSWITCH 呼叫过程中，会根据这些变量控制 Channel 的行为。

\$\$\{var\} 与 \${var}

`${var}` 是在 dialplan、application 或 directory 中设置的变量，它会影响呼叫流程并且可以动态的改变。而 `$$\{var\}` 则是全局的变量，它仅在预处理阶段（系统启动时，或重新装载 - reloadxml 时）被求值。后者一般用于设置一些系统一旦启动就不会轻易改变的量，如 `$$\{domain\}` 或 `$$\{local_ip_v4\}` 等。所以，两者最大的区别是， `$$\{var\}` 只求值一次，而 `${var}` 则在每次执行时求值（如一个新电话进来时）。

\$variable_xxxx

你会发现，有些变量在显示时（可以使用 dp_tools 中的 info() 显示，后面会讲到）是以“variable_”开头的，但在实际引用时要去掉这开头的“variable_”。如“variable_user_name”，引用时要使用“ `${user_name}`”。 http://wiki.freeswitch.org/wiki/Channel_Variables#variable_xxxx 列举了一些常见的变量显示与引用时的对应关系。

给 Variable 赋值

在 dialplan 中，有两个程序可以给 Variable 赋值

```
1 <action application="set" data="my_var=my_value"/>
2 <action application="export" data="my_var=my_value"/>
```

以上两条命令都可以设置 my_var 变量的值为 my_value。不同的是 — set 程序仅会作用于“当前”的 Channel (a-leg)，而 export 程序则会将变量设置到两个 Channel (a-leg 和 b-leg) 上，如果当时 b-leg 还没有创建，则会在创建时设置。另外，export 还可以只将变量设置到 b-leg 上：

```
1 <action application="export" data="nolocal:my_var=my_value"/>
```

在实际应用中，如果 a-leg 上已经有一些变量的值（如 var1、var2、var3），而想同时把这些变量都复制到 b-leg 上，可以使用以下几种办法：

```
1 <action application="export" data="var1=$var1">
2 <action application="export" data="var2=$var2">
3 <action application="export" data="var3=$var3">
```

或者使用如下等价的方式：

```
1 <action application="set" data="export_vars=var1,var2,var3">
```

所以，其实 set 也具有能往 b-leg 上赋值的能力，其实，它和 export 一样，都是操作 export_vars 这个特殊的变量。

取消 Variable 定义

取消 Variable 定义只需对它赋一个特殊的值`_undef_`:

```
1 <action application="set" data="var1=_undef_">
```

截取 Variable 的一部分

可以使用特殊的语法取一个 Variable 的子串，格式是“ `${var:位置:长度}` ”。其中“位置”从 0 开始计数，若为负数则从字符串尾部开始计数；如果“长度”为 0 或小于 0，则会从当前“位置”一直取到字符串结尾（或开头，若“位置”为负的话）。例如 var 的值为 1234567890，那么：

```
1 ${var}      = 1234567890
2 ${var:0:1}  = 1
3 ${var:1}    = 234567890
4 ${var:-4}   = 7890
5 ${var:-4:2} = 78
6 ${var:4:2}  = 56
```

5.9 小结

FreeSWITCH 是由一个稳定的核心及外围的可加载模块组成的。核心代码经过精心设计及严格测试，保证了它的稳定性。外围的模块则保证了它的可扩展性。核心与模块间通过 Public API 通信，而模块间则通过事件系统（Event）进行通信，事件机制消除了模块间的耦合，进一步增加了其可伸缩性。

第六章 运行FreeSWITCH

读到本章，你应该对 FreeSWITCH 有了一个比较全面的了解，迫切地想实验它强大的功能了。让我们从最初的运行开始，一步一步进入 FreeSWITCH 的神秘世界。

6.1 命令行参数

一般来说，FreeSWITCH 不需要任何命令行参数就可以启动，但在某些情况下，你需要以一些特殊的参数启动。在此，仅作简单介绍。如果你知道是什么意思，那么你就可以使用，如果不知道，多半你用不到。

使用 `freeswitch -help` 或 `freeswitch --help` 会显示以下信息：

```

1  -nf          -- no forking
2  -u [user]    -- 启动后以非 root 用户 user 身份运行
3  -g [group]   -- 启动后以非 root 组 group 身份运行
4  -help        -- 显示本帮助信息
5  -version     -- 显示版本信息
6  -waste       -- 允许浪费内存, FreeSWITCH 仅需 240K 的栈空间
7           你可以使用 ulimit -s 240 限制栈空间使用, 或使用该选择忽略警告信息
8  -core        -- 出错时进行内核转储
9  -rp          -- 开启高优先级 (实时) 设置
10 -lp          -- 开启低优先级设置
11 -np          -- 普通优先级 (系统默认)
12 -vg          -- 在 valgrind 下运行, 调试内存泄露时使用
13 -nosql       -- 不使用 SQL, show channels 类的命令将不能显示结果
14 -heavy-timer -- 更精确的时钟。可能会更精确, 但对系统要求更高
15 -nonat       -- 如果路由器支持 uPnP 或 NAT-PMP, 则 FreeSWITCH
               可以自动解决 NAT 穿越问题。如果路由器不支持, 则该选项可以使启动更快
16
17 -nocal       -- 关闭时钟核准。FreeSWITCH 理想的运行环境是 1000 Hz 的内核时钟
               如果你的内核时钟小于 1000 Hz 或在虚拟机上, 可以尝试关闭该选项
18
19 -nort        -- 关闭 FreeSWITCH, 它会在 run 目录中查找 PID文件
20 -stop        -- 启动到后台模式, 没有控制台
21 -nc          -- 后台模式, 等待系统完全初始化完毕之后再退出父进程, 隐含 -nc 选项
22 -ncwait      -- 启动到控制台, 默认
23 -c           -- 指定其它的基准目录, 在配置文件中使用 $$ {base}
24 -base [confdir] -- 指定其它的配置文件所在目录, 须与 -log、-db 合用
25 -conf [confdir]
26 -log [logdir] -- 指定其它的日志目录
27 -run [rundir] -- 指定其它存放 PID 文件的运行目录
28 -db [dbdir]   -- 指定其它数据库目录
29 -mod [moddir] -- 指定其它模块目录
30 -htdocs [htdocsdir] -- 指定其它 HTTP 根目录
31 -scripts [scriptsdir] -- 指定其它脚本目录
32 -temp [directory] -- 指定其它临时文件目录
33 -grammar [directory] -- 指定其它语法目录
34 -recordings [directory] -- 指定其它录音目录
35 -storage [directory] -- 指定其它存储目录 (语音信箱等)
36 -sounds [directory] -- 指定其它声音文件目录

```

6.2 系统启动脚本

在学习调试阶段, 你可以启动到前台, 而系统真正运行时, 你可以使用 -nc 参数启动到后台, 然后通过查看 log/freeswitch.log 跟踪系统运行情况 (你可以用 tail -f 命令实时跟踪, 我一般使用 less)。

一般情况下, 启动到前台更容易调试, 但你又不想在每次关闭 Terminal 时停止 FreeSWITCH, 那么, 你可以借助 screen 来实现。

在真正的生产系统上，你需要它能跟系统一起启动。在 *nix 系统上，启动脚本一般放在 /etc/init.d/。你可以在系统源代码目录下找到不同系统启动脚本 debian/freeswitch.init 及 build/freeswitch.init.*，参考使用。在 Windows 上，你也可以注册为 Windows 服务，参见附录中的 FAQ。

6.3 如何判断 FreeSWITCH 已经运行？

某些情况下，我们需要知道 FreeSWITCH 是否已经运行，在 *nix 系统上可以用下面方法判断：

- 1) 看进程是否存在。如果进程已经启动，下列命令能列出所有与 freeswitch 相关的进程

```
ps aux | grep freeswitch
```

- 2) 看相关端口是否被占用。假设你使用默认的 5060 端口，则使用下述命令可以列出所有 5060 端口相关的网络连接

```
netstat -an | grep 5060
```

Windows 平台上也有 netstat 命令，但没有 grep。进程信息可以按 Ctrl+Alt+Del 在程序管理器中查看。

6.4 控制台与命令客户端

系统不带参数会启动到控制台，在控制台上你可以输入各种命令以控制或查询 FreeSWITCH 的状态。试试输入以下命令：

```
1 version          -- 显示当前版本  
2 status           -- 显示当前状态  
3 sofia status    -- 显示 sofia 状态  
4 help             -- 显示帮助
```

为了调试方便，FreeSWITCH 还在 conf/autoload_configs/switch.conf.xml 中定义了一些控制台快捷键。你可以通过 F1-F12 来使用它们（不过，在某些操作系统上，有些快捷键可能与操作系统的相冲突，那你就只直接输入这些命令或重新定义他们了）。

```

1  <cli-keybindings>
2    <key name="1" value="help"/>
3    <key name="2" value="status"/>
4    <key name="3" value="show channels"/>
5    <key name="4" value="show calls"/>
6    <key name="5" value="sofia status"/>
7    <key name="6" value="reloadxml"/>
8    <key name="7" value="console loglevel 0"/>
9    <key name="8" value="console loglevel 7"/>
10   <key name="9" value="sofia status profile internal"/>
11   <key name="10" value="sofia profile internal siptrace on"/>
12   <key name="11" value="sofia profile internal siptrace off"/>
13   <key name="12" value="version"/>
14 </cli-keybindings>

```

FreeSWITCH 是 Client-Server 结构，不管 FreeSWITCH 运行在前台还是后台，你都可以使用客户端软件 `fs_cli` 连接 FreeSWITCH.

`fs_cli` 是一个类似 Telnet 的客户端（也类似于 Asterisk 中的 `asterisk -r` 命令），它使用 FreeSWITCH 的 ESL (Event Socket Library) 库与 FreeSWITCH 通信。当然，需要加载模块 `mod_event_socket`。该模块是默认加载的。

正常情况下，直接输入 `bin/fs_cli` 即可连接上，并出现系统提示符。如果出现： [ERROR] libs/esl/fs_cli.c:652 main() Error Connecting [Socket Connection Error] 这样的错误，说明 FreeSWITCH 没有启动或 `mod_event_socket` 没有正确加载，请检查 TCP 端口 8021 端口是否处于监听状态或被其它进程占用。

`fs_cli` 也支持很多命令行参数，值得一提的是 `-x` 参数，它允许执行一条命令后退出，这在编写脚本程序时非常有用（如果它能支持管道会更有用，但是它不支持）：

```

1  bin/fs_cli -x "version"
2  bin/fs_cli -x "status"

```

其它的参数都可以通过配置文件来实现，在这里就不多说了。可以参见：http://wiki.freeswitch.org/wiki/Fs_cli

使用 `fs_cli`，不仅可以连接到本机的 FreeSWITCH，也可以连接到其它机器的 FreeSWITCH 上（或本机另外的 FreeSWITCH 实例上），通过在用户主目录下编辑配置文件 `.fs_cli.conf`（注意前面的点“.”），可以定义要连接的多个机器：

```

1 [server1]
2 host      => 192.168.1.10
3 port      => 8021
4 password => secret_password
5 debug     => 7
6
7 [server2]
8 host      => 192.168.1.11
9 port      => 8021
10 password => someother_password
11 debug     => 0

```

注意：如果要连接到其它机器，要确保 FreeSWITCH 的 Event Socket 是监听在真实网卡的 IP 地址上，而不是 127.0.0.1。另外，在 UNIX 中，以点开头的文件是隐藏文件，普通的 ls 命令是不能列出它的，可以使用 ls -a。

一旦配置好，就可以这样使用它：

```

1 bin/fs_cli server1
2 bin/fs_cli server2

```

在 fs_cli 中，有几个特殊的命令，它们是以 “/” 开头的，这些命令并不直接发送到 FreeSWITCH，而是先由 fs_cli 处理。/quit、/bye、/exit、Ctrl + D 都可以退出 fs_cli；/help 是帮助。

其它一些 “/” 开头的指令与 Event Socket 中相关的命令相同，如：

```

1 /event      -- 启用事件接收
2 /noevents   -- 关闭事件接收
3 /nixevent   -- 除了特定一种外，启用所有事件
4 /log        -- 设置 log 级别，如 /log info 或 /log debug 等
5 /nolog      -- 关闭 log
6 /filter     -- 过滤事件

```

除此之外，其它命令都与直接在 FreeSWITCH 控制台上执行是一样的。它也支持快捷键，最常用的快捷键是 F6（reloadxml）、F7（关闭 log 输出）、F8（开启 debug 级别的 log 输出）。

在 *nix 上，两者都通过 libeditline 支持[命令行编辑功能](#)。可以通过上、下箭头查看命令历史。

6.5 发起呼叫

可以在 FreeSWITCH 中使用 originate 命令发起一次呼叫，如果用户 1000 已经注册，那么：

```
1 originate user/alice &echo
```

上述命令在呼叫 1000 这个用户后，便执行 echo 这个程序。echo 是一个回音程序，即它会把任何它“听到”的声音（或视频）再返回（说）给对方。因此，如果这时候用户 1000 接了电话，无论说什么都能听到自己的声音。

6.5.1 呼叫字符串

上面的例子中，user/alice 称为呼叫字符串，或呼叫 URL。user 是一种特殊的呼叫字符串。我们先来复习一下第四章的场景。FreeSWITCH UA 的地址为 192.168.4.4:5050，alice UA 的地址为 192.168.4.4:5090，bob UA 的地址为 192.168.4.4:26000。若 alice 已向 FreeSWITCH 注册，在 FreeSWITCH 中就可以看到她的注册信息：

```
1 freeswitch@du-sevens-mac-pro.local> sofia status profile internal reg
2
3
4 Registrations:
5 =====
6 Call-ID: ZTRkYjdjYzY00WFhNDRh0GFkNDUxMTdhMWJhNjRmNmE.
7 User: alice@192.168.4.4
8 Contact: "Alice" <sip:alice@192.168.4.4:5090;ri=instance=a86a656037ccfaba;transport=UDP>
9 Agent: Zoiper rev.5415
10 Status: Registered(UDP)(unknown) EXP(2010-05-02 18:10:53)
11 Host: du-sevens-mac-pro.local
12 IP: 192.168.4.4
13 Port: 5090
14 Auth-User: alice
15 Auth-Realm: 192.168.4.4
16 MWI-Account: alice@192.168.4.4
17
18 =====
```

FreeSWITCH 根据 Contact 字段知道 alice 的 SIP 地址 sip:alice@192.168.4.4:5090。当使用 originate 呼叫 user/alice 这个地址时，FreeSWITCH 便查找本地数据库，向 alice 的地址 sip:alice@192.168.4.4:5090 发送 INVITE 请求（实际的呼叫字符串是由用户目录中的 dial-string 参数决定的）。

6.5.2 API 与 App

在上面的例子中，originate 是一个命令（Command），它用于控制 FreeSWITCH 发起一个呼叫。FreeSWITCH 的命令不仅可以在控制台上使用，也可以在各种嵌入式脚本、Event Socket（fs_cli 就是使用了 ESL 库）或 HTTP RPC 上使用，所有命令都遵循一个抽象的接口，因而这些命令又称 API Commands。

echo() 则是一个应用程序 (Application, 简称 App) , 它的作用是控制一个 Channel 的一端。我们知道, 一个 Channel 有两端, 在上面的例子中, alice 是一端, 别一端就是 echo()。电话接通后相当于 alice 在跟 echo() 这个家伙在通话。如果你还记得第 4 章, 你会发现其实它们组成了 FreeSWITCH 的一条腿 (如, a-leg) , 称作“单腿通话 (one-legged connection) ”。

另一个常用的 App 是 park()

```
1 originate user/alice &park()
```

我们初始化了一个呼叫, 在 alice 接电话后对端必须有一个人在跟他讲话, 否则的话, 一个 Channel 只有一端, 那是不可思议的。而如果这时 FreeSWITCH 找不到一个合适的人跟 alice 通话, 那么它可以将该电话“挂起”, park()便是执行这个功能, 它相当于一个 Channel 特殊的一端。

park() 的用户体验不好, alice 不知道要等多长时间才有人接电话, 由于她听不到任何声音, 实际上她在奇怪电话到底有没有接通。相对而言, 另一个程序 hold()则比较友好, 它能在等待的同时播放保持音乐 (MOH, Music on Hold) 。

```
1 originate user/alice &hold()
```

当然, 你也可以直接播放一个特定的声音文件:

```
1 originate user/alice &playback(/root/welcome.wav)
```

或者, 直接录音:

```
1 originate user/alice &record(/tmp/voice_of_alice.wav)
```

以上的例子实际上都只是建立一个 Channel, 相当于 FreeSWITCH 作为一个 UA 跟 alice 通话。它是个一条腿 (one leg, 只有a-leg) 的通话。在大多数情况下, FreeSWITCH 都是做为一个 B2BUA 来桥接两个 UA 进行通话话的。在 alice 接听电话以后, bridge()程序可以再启动一个 UA 呼叫 bob:

```
1 originate user/alice &bridge(user/bob)
```

终于, alice 和 bob 可以通话了。我们也可以用另一个方式建立他们之音的通话:

```
1 originate user/alice &park()
2 originate user/bob &park()
3 show channels
4 uuid_bridge <alice_uuid> <bob_uuid>
```

在这里，我们分别呼叫 alice 和 bob，并把他们暂时 park 到一个地方。通过命令 show channels 我们可以知道每个 Channel 的 UUID，然后使用 uuid_bridge 命令将两个 Channel 桥接起来。与上一种方式不同，上一种方式实际上是先桥接，再呼叫 bob。

上面，我们一共学习了两条命令（API），originate 和 uuid_bridge。以及几个程序（App）-echo、park、bridge 等。细心的读者可以会发现，uuid_bridge API 和 bridge App 有些类似，我也知道他们一个是先呼叫后桥接，另一个是先桥接后呼叫，那么，它们到底有什么本质的区别呢？

简单来说，一个 App 是一个程序（Application），它作为一个 Channel 一端与另一端的 UA 进行通信，相当于它工作在 Channel 内部；而一个 API 则是独立于一个 Channel 之外的，它只能通过 Channel 的 UUID 来控制一个 Channel（如果需要的话）。

这就是 API 与 App 最本质的区别。通常，我们在控制台上输入的命令都是 API；而在 dialplan 中执行的程序都是 App（dialplan 中也能执行一些特殊的 API）。大部分公用的 API 都是在 mod_commands 模块中加载的；而 App 则在 mod_dptools 中，因而 App 又称为拨号计划工具（Dialplan Tools）。某些模块（如 mod_sofia）有自己的 API 和 App。

某些 App 有与其对应的 API，如上述的 bridge/uuid_bridge，还有 transfer/uuid_transfer、playback/uuid_playback 等。UUID 版本的 API 都是在一个 Channel 之外对 Channel 进行控制的，它们应用于不能参与到通话中却又想对正在通话的 Channel 做点什么的场景中。例如 alice 和 bob 正在畅聊，有个坏蛋使用 uuid_kill 将电话切断，或使用 uuid_broadcast 给他们广播恶作剧音频，或者使用 uuid_record 把他们谈话的内容录音等。

参考资料：

- mod_dptools: http://wiki.freeswitch.org/wiki/Mod_dptools
- mod_commands: http://wiki.freeswitch.org/wiki/Mod_commands

6.6 命令行帮助

在本章的最后，我们来学习一个如何使用 FreeSWITCH 的命令行帮助。

使用 help 命令可以列出所有命令的帮助信息。某些命令，也有自己的帮助信息，如 sofia：

```

1 freeswitch@du-sevens-mac-pro.local> sofia help
2
3 USAGE:
4 -----
5 sofia help
6 sofia profile <profile_name> [[start|stop|restart|rescan]
7     [reloadxml]|flush_inbound_reg [<call_id>] [reboot]|[register|unregister]
8 ....

```

其中，用尖括号 (<>) 括起来的表示要输入的参数，而用方括号 ([]) 括起来的则表示可选项，该参数可以有也可以没有。用竖线 (|) 分开的参数列表表示“或”的关系，即只能选其一。

FreeSWITCH 的命令参数没有统一的解析函数，而都是由命令本身的函数负责解析的，因而不是很规范，不同的命令可能有不同的风格。所以使用时，除使用帮助信息外，最好还是查阅一下 Wiki 上的帮助。本书的附录中也有相应的中文参考。

6.7 小结

本章介绍了如何启动与控制 FreeSWITCH，并提到了几个常用的命令。另外，本章还着重讲述了 App 与 API 的区别，搞清楚这些概念对后面的学习是很有帮助的。

第七章 SIP 模块 — mod_sofia

SIP 模块是 FreeSWITCH 的主要模块，所以，值得拿出专门一章来讲解。

在前几章时里，你肯定见过几次 `sofia` 这个词，只是或许还不知道是什么意思。是这样的，[Sofia-SIP](#) 是由诺基亚公司开发的 SIP 协议栈，它以开源的许可证 LGPL 发布，为了避免重复发明轮子，FreeSWITCH 便直接使用了它。

在 FreeSWITCH 中，实现一些互联协议接口的模块称为 `Endpoint`。FreeSWITCH 支持很多的 `Endpoint`，如 SIP、H232 等。那么实现 SIP 的模块为什么不支持叫 `mod_sip` 呢？这是由于 FreeSWITCH 的 `Endpoint` 是一个抽象的概念，你可以用任何的技术来实现。实际上 `mod_sofia` 只是对 Sofia-SIP 库的一个粘合和封装。除 Sofia-SIP 外，还有很多开源的 SIP 协议栈，如 `pjsip`、`osip` 等。最初选型的时候，FreeSWITCH 的开发团队也对比过许多不同的 SIP 协议栈，最终选用了 Sofia-SIP。FreeSWITCH 是一个高度模块化的结构，如果你不喜欢，可以自己实现 `mod_pjsip` 或 `mod_osip` 等，它们是互不影响的。这也正是 FreeSWITCH 架构设计的精巧之处。

Sofia-SIP 遵循 RFC3261 标准，因而 FreeSWITCH 也是。

7.1 配置文件

Sofia 的配置文件是 `conf/autoload_configs/sofia.conf.xml`，不过，你一般不用直接修改它，因为它实际上直接使用一条预处理指令装入了 `conf/sip_profiles/` 目录中的 XML 文件：

```
1 <X-PRE-PROCESS cmd="include" data="..sip_profiles/*.xml"/>
```

所以，从现在起，可以认为所有的 Sofia 配置文件都在 `conf/sip_profiles/` 中。

Sofia 支持多个 profile，而一个 profile 相当于一个 SIP UA，在启动后它会监听一个“IP地址：端口”对。读到这里细心的读者或许会发现我们前面的一个错误。我们在讲 B2BUA 的概念时，实际上只用到了一个 profile，也就是一个 UA，但我们还是说 FreeSWITCH 启动了两个 UA（一对背靠背的 UA）来为 alice 和 bob 服务。是的，从物理上来讲，它确实只是一个 UA，但由于它同时支持多个 Session，在逻辑上就是相当于两个 UA，为了不使读者太纠结于这种概念问题中，我在前面没有太多的分析。但到了本章，你应该非常清楚 UA 的含义了。

FreeSWITCH 默认的配置带了三个 profile (也就是三个 UA)，在这里，我们不讨论 IPv6，因此只剩下 internal 和 external 两个。internal 和 external 的区别就是一个运行在 5060 端口上，另一个是在 5080 端口上。当然，还有其它区别，我们慢慢讲。

7.1.1 internal.xml

internal.xml 定义了一个 profile，在本节，我们以系统默认的配置逐行来解释：

```
1 <profile name="internal">
```

profile 的名字就叫 internal，这个名字本身并没有特殊的意义，也不需要与文件名相同，你可以改成任何你喜欢的名字，只是需要记住它，因为很多地方要使用这个名字。

```
1 <aliases>
2     <!--<alias name="default"/>-->
3 </aliases>
```

如果你喜欢，可以为该 profile 起一个别名。注意默认是加了注释的，也就是说不起作用。

```
1 <gateways>
2     <!--<X-PRE-PROCESS cmd="include" data="internal/*.xml"/>
3 </gateways>
```

既然 profile 是一个 UA，它就可以注册到别的 SIP 服务器上去，它要注册的 SIP 服务器就称为 Gateway。我们一般不在 internal 这个 profile 上使用 Gateway。

```
1 <domains>
2     <!--<domain name="$$\{domain\}" parse="true"/>-->
3     <domain name="all" alias="true" parse="false"/>
4 </domains>
```

定义该 profile 所属的 domain。它可以是 IP 地址，或一个 DNS 域名。需要注意，直接在 hosts 文件中设置的 IP-域名可能不好用。

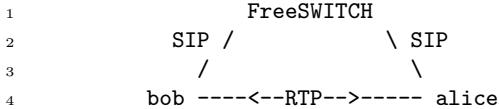
```
1 <settings>
```

settings 部分设置 profile 的参数。

```
1 <!--<param name="media-option" value="resume-media-on-hold"/>-->
```

如果 FreeSWITCH 是没有媒体 (no media) 的, 那么如果设置了该参数, 当你在话机上按下 hold 键时, FreeSWITCH 将会回到有媒体的状态。

那么什么叫有媒体无媒体呢? 如下图, bob 和 alice 通过 FreeSWITCH 使用 SIP 接通了电话, 他们谈话的语音 (或视频) 数据要通过 RTP 包传送的。RTP 可以像 SIP 一样经过 FreeSWITCH 转发, 但是, RTP 占用很大的带宽, 如果 FreeSWITCH 不需要“偷听”他们谈话的话, 为了节省带宽, 完全可以让 RTP 直接在两者间传送, 这种情况对 FreeSWITCH 来讲就是没有 media 的, 在 FreeSWITCH 中也称 bypass media (媒体绕过)。



```
1 <!--<param name="media-option" value="bypass-media-after-att-xfer"/><!--&gt;</pre>

```

Attended Transfer 称为出席转移, 它需要 media 才能完成工作。但如果在执行 att-xfer 之前没有媒体, 该参数能让 att-xfer 执行时有 media, 转移结束后再回到 bypass media 状态。

```
1 <!-- <param name="user-agent-string" value="FreeSWITCH Rocks!"/> -->
```

不用解释, 就是设置 SIP 消息中显示的 User-Agent 字段。

```
1 <param name="debug" value="0"/>
```

debug 级别。

```
1 <!-- <param name="shutdown-on-fail" value="true"/> -->
```

由于各种原因 (如端口被占用, IP地址错误等), 都可能造成 UA 在初始化时失败, 该参数在失败时会停止 FreeSWITCH。

```
1 <param name="sip-trace" value="no"/>
```

是否开启 SIP 消息跟踪。另外, 也可以在控制台上用以下命令开启和关闭 sip-trace:

```
1 sofia profile internal siptrace on
2 sofia profile internal siptrace off
```

```
1 <param name="log-auth-failures" value="true"/>
```

是否将认证错误写入日志。

```
1 <param name="context" value="public"/>
```

context 是 dialplan 中的环境。在此指定来话要落到 dialplan 的哪个 context 环境中。需要指出，如果用户注册到该 profile 上（或是经过认证的用户，即本地用户），则用户目录（directory）中设置的 context 优先级要比这里高。

```
1 <param name="rfc2833-pt" value="101"/>
```

设置 SDP 中 RFC2833 的值。RFC2833 是传递 DTMF 的标准。

```
1 <param name="sip-port" value="$$\{internal_sip_port\}"/>
```

监听的 SIP 端口号，变量 internal_sip_port 在 vars.xml 中定义，默认是 5060。

```
1 <param name="dialplan" value="XML"/>
```

设置对应默认的 dialplan。我们后面会专门讲 dialplan。

```
1 <param name="dtmf-duration" value="2000"/>
```

设置 DTMF 的时长。

```
1 <param name="inbound-codec-prefs" value="$$\{global_codec_prefs\}"/>
```

支持的来话语音编码，用于语音编码协商。global_codec_prefs 是在 vars.xml 中定义的。

```
1 <param name="outbound-codec-prefs" value="$$\{global_codec_prefs\}"/>
```

支持的去话语音编码。

```
1 <param name="rtp-timer-name" value="soft"/>
```

RTP 时钟名称

```
1 <param name="rtp-ip" value="$$\{local_ip_v4\}"/>
```

RTP 的 IP 地址，仅支持 IP 地址而不支持域名。虽然 RTP 标准说应该域名，但实际情况是域名解析有时不可靠。

```
1 <param name="sip-ip" value="$$\{local_ip_v4\}">
```

SIP 的 IP。不支持域名。

```
1 <param name="hold-music" value="$$\{hold_music\}">
```

UA 进行 hold 状态时默认播放的音乐。

```
1 <param name="apply-nat-acl" value="nat.auto"/>
```

使用哪个 NAT ACL。

```
1 <!-- <param name="extended-info-parsing" value="true" /> -->
```

扩展 INFO 解析支持。

```
1 <!--<param name="aggressive-nat-detection" value="true" />-->
```

NAT穿越，检测 SIP 消息中的 IP 地址与实际的 IP 地址是否相符，详见 NAT穿越。

```
1 <!--  
2     There are known issues (asserts and segfaults) when 100rel is enabled.  
3     It is not recommended to enable 100rel at this time.  
4 -->  
5 <!--<param name="enable-100rel" value="true" />-->
```

该功能暂时还不推荐使用。

```
1 <!--<param name="enable-compact-headers" value="true" />-->
```

支持压缩 SIP 头。

```
1 <!--<param name="enable-timer" value="false" />-->
```

开启、关闭 SIP 时钟。

```
1 <!--<param name="minimum-session-expires" value="120" />-->
```

SIP 会话超时值，在 SIP 消息中设置 Min-SE。

```
1 <param name="apply-inbound-acl" value="domains"/>
```

对来话采用哪个 ACL。详见 ACL。

```
1 <param name="local-network-acl" value="localnet.auto"/>
```

默认情况下，FreeSWITCH 会自动检测本地网络，并创建一条 localnet.auto ACL 规则。

```
1 <!--<param name="apply-register-acl" value="domains"/>-->
```

对注册请求采用哪个 ACL。

```
1 <!--<param name="dtmf-type" value="info"/>-->
```

DTMF 收号的类型。有三种方式，info、inband、rfc2833。

- info 方式是采用 SIP 的 INFO 消息传送 DTMF 按键信息的，由于 SIP 和 RTP 是分开走的，所以，可能会造成不同步。
- inband 是在 RTP 包中象普通语音数据那样进行带内传送，由于需要对所有包进行鉴别和提取，需要占用更多的资源。
- rfc2833 也是在带内传送，但它的 RTP 包有特殊的标记，因而比 inband 方式节省资源。它是在 RFC2833 中定义的。

如何发送请求消息。true 是每次都发送，而 first-only 只是首次注册时发送。

```
1 <!--<param name="caller-id-type" value="rpid|pid|none"/>-->
```

设置来电显示的类型，rpid 将会在 SIP 消息中设置 Remote-Party-ID，而 pid 则会设置 P-*-Identity，如果不需要这些，可以设置成 none。

```
1 <param name="record-path" value="$$[recordings_dir]"/>
```

录音文件的默认存放路径。

```
1 <param name="record-template"
2 value="${caller_id_number}.${target_domain}.${strftime(%Y-%m-%d-%H-%M-%S)}.wav"/>
```

录音文件名模板。

```
1 <param name="manage-presence" value="true"/>
```

是否支持列席。

```
1 <!--<param name="manage-shared-appearance" value="true"/>-->
```

是否支持 SLA - Shared Line Appearance。

```
1 <!--<param name="dbname" value="share_presence"/>-->
```

```
2 <!--<param name="presence-hosts" value="${domain}"/>-->
```

这两个参数用以在多个 profile 间共享列席信息。

```
1 <!-- This setting is for AAL2 bitpacking on G726 -->
```

```
2 <!--<param name="bitpacking" value="aal2"/> -->
```

```
3
```

```
4 <!--<param name="max-proceeding" value="1000"/>-->
```

最大的开放对话 (SIP Dialog) 数。

```
1 <!--session timers for all call to expire after the specified seconds -->
```

```
2 <!--<param name="session-timeout" value="120"/>-->
```

会话超时时间。

```
1 <!-- Can be 'true' or 'contact' -->
```

```
2 <!--<param name="multiple-registrations" value="contact"/>-->
```

是否支持多点注册，可以是 contact 或 true。开启多点注册后多个 UA 可以注册上来，有人呼叫这些 UA 时所有 UA 都会振铃。

```
1 <!--set to 'greedy' if you want your codec list to take precedence -->
```

```
2 <param name="inbound-codec-negotiation" value="generous"/>
```

SDP 中的语音编协商，如果设成 greedy，则自己提供的语音编码列表会有优先权。

```
1 <!-- if you want to send any special bind params of your own -->
```

```
2 <!--<param name="bind-params" value="transport=udp"/>-->
```

```
3
```

```
4 <!--<param name="unregister-on-options-fail" value="true"/>-->
```

为了 NAT 穿越或 keep alive, 如果 FreeSWITCH 向其它网关注册时, 可以周期性地发一些 OPTIONS 包, 相当于 ping 功能。该参数说明当 ping 失败时是否自动取消注册。

```
1 <param name="tls" value="$$\{internal_ssl_enable\}">
```

是否支持 TLS, 默认否。

```
1 <!-- additional bind parameters for TLS -->
2 <param name="tls-bind-params" value="transport=tls"/>
3 <!-- Port to listen on for TLS requests. (5061 will be used if unspecified) -->
4 <param name="tls-sip-port" value="$$\{internal_tls_port\}">
5 <!-- Location of the agent.pem and cafile.pem ssl certificates (needed for TLS server) -->
6 <param name="tls-cert-dir" value="$$\{internal_ssl_dir\}">
7 <!-- TLS version ("sslv23" (default), "tlsv1"). NOTE: Phones may not work with TLSv1 -->
8 <param name="tls-version" value="$$\{sip_tls_version\}">
```

下面都是与 TLS 有关的参数, 略。

```
1 <!--<param name="rtp-autoflush-during-bridge" value="false"/>-->
```

该选项默认为 true。即在桥接电话时是否自动 flush 媒体数据 (如果套接字上已有数据时, 它会忽略定时器睡眠, 能有效减少延迟)。

```
1 <!--<param name="rtp-rewrite-timestamps" value="true"/>-->
```

是否透传 RTP 时间戳。

```
1 <!--<param name="pass-rfc2833" value="true"/>-->
```

是否透传 RFC2833 DTMF 包。

```
1 <!--<param name="odbc-dsn" value="dsn:user:pass"/>-->
```

使用 ODBC 数据库代替默认的 SQLite。

```
1 <!--<param name="inbound-bypass-media" value="true"/>-->
```

将所有来电设置为媒体绕过。

```
1 <!--<param name="inbound-proxy-media" value="true"/>-->
```

将所有来电设置为媒体透传。

```
1 <!--Uncomment to let calls hit the dialplan *before* you decide if the codec is ok-->
2 <!--<param name="inbound-late-negotiation" value="true"/>-->
```

对所有来电来讲，晚协商有助于在协商媒体编码之前，先前电话送到 Dialplan，因而在 Dialplan 中可以进行个性化的媒体协商。

```
1 <!-- <param name="accept-blind-reg" value="true"/> -->
```

该选项允许任何电话注册，而不检查用户和密码及其它设置。

```
1 <!-- <param name="accept-blind-auth" value="true"/> -->
```

与上一条类似，该选项允许任何电话通过认证。

```
1 <!-- <param name="suppress-cng" value="true"/> -->
```

抑制 CNG。

```
1 <param name="nonce-ttl" value="60"/>
```

SIP 认证中 nonce 的生存时间。

```
1 <!--<param name="disable-transcoding" value="true"/>-->
```

禁止译码，如果该项为 true 则在 bridge 其它电话时，只提供与 a-leg 兼容或相同的语音编码列表进行协商，以避免译码。

```
1 <!--<param name="manual-redirect" value="true"/> -->
```

允许在 Dialplan 中进行人工转向。

```
1 <!--<param name="disable-transfer" value="true"/> -->
```

禁止转移。

```
1 <!--<param name="disable-register" value="true"/> -->
```

禁止注册。

```

1  <!-- Used for when phones respond to a challenged ACK with method INVITE in the hash -->
2  <!--<param name="NDLB-broken-auth-hash" value="true"/>-->
3  <!-- add a ;received=<ip>:<port>" to the contact when replying to register for nat handling -->
4  <!--<param name="NDLB-received-in-nat-reg-contact" value="true"/>-->
5
6  <param name="auth-calls" value="$$internal_auth_calls"/>
```

是否对电话进行认证。

```

1  <!-- Force the user and auth-user to match. -->
2
3  <param name="inbound-reg-force-matching-username" value="true"/>
```

强制用户与认证用户必须相同。

```
1  <param name="auth-all-packets" value="false"/>
```

在认证时，对所有 SIP 消息都进行认证，而不是仅针对 INVITE 消息。

```

1  <!-- external_sip_ip
2    Used as the public IP address for SDP.
3    Can be an one of:
4      ip address          - "12.34.56.78"
5      a stun server lookup - "stun:stun.server.com"
6      a DNS name         - "host:host.server.com"
7      auto                - Use guessed ip.
8      auto-nat            - Use ip learned from NAT-PMP or UPNP
9
10 <-->
11 <param name="ext-rtp-ip" value="auto-nat"/>
12 <param name="ext-sip-ip" value="auto-nat"/>
```

设置 NAT 环境中公网的 RTP IP。该设置会影响 SDP 中的 IP 地址。有以下几种可能：

- 一个 IP 地址，如 12.34.56.78
- 一个 stun 服务器，它会使用 stun 协议获得公网 IP，如 stun:stun.server.com
- 一个 DNS 名称，如 host:host.server.com
- auto，它会自动检测 IP 地址
- auto-nat，如果路由器支持 NAT-PMP 或 UPNP，则可以使用这些协议获取公网 IP。

指定的时间内 RTP 没有数据传送，则挂机。

```
1 <param name="rtp-hold-timeout-sec" value="1800"/>
```

RTP 处理保持状态的最大时长。

```
1 <!-- <param name="vad" value="in"/> -->
2 <!-- <param name="vad" value="out"/> -->
3 <!-- <param name="vad" value="both"/> -->
```

语音活动状态检测，有三种可能，可设为入、出，或双向，通常来说“出”（out）是一个比较好的选择。

```
1 <!--<param name="alias" value="sip:10.0.1.251:5555"/>-->
```

给本 sip profile 设置别名。

```
1 <!--all inbound reg will look in this domain for the users -->
2 <param name="force-register-domain" value="$$\{domain\}"/>
3 <!--force the domain in subscriptions to this value -->
4 <param name="force-subscription-domain" value="$$\{domain\}"/>
5 <!--all inbound reg will stored in the db using this domain -->
6 <param name="force-register-db-domain" value="$$\{domain\}"/>
7 <!--force suscription expires to a lower value than requested-->
8 <!--<param name="force-subscription-expires" value="60"/>-->
```

以上选项默认是起作用的，这有助于默认的例子更好的工作。它们会在注册及订阅时在数据库中写入同样的域信息。如果你在使用一个 FreeSWITCH 支持多个域时，不要选这些选项。

```
1 <!--<param name="enable-3pcc" value="true"/>-->
```

该选项有两个值，true 或 proxy。 true 则直接接受 3pcc 来电；如果选 proxy，则会一直等待电话应答后才回送接受。

```
1 <!-- use at your own risk or if you know what this does.-->
2 <!--<param name="NDLB-force-rport" value="true"/>-->
```

在 NAT 时强制 rport。除非你很了解该参数，否则后果自负。

```
1 <param name="challenge-realm" value="auto_from"/>
```

设置 SIP Challenge 是使用的 realm 字段是从哪个域获取，auto_from 和 auto_to 分别是从 from 和 to 中获取，除了这两者，也可以是任意的值，如 freeswitch.org.cn。

```
1 <!--<param name="disable-rtp-auto-adjust" value="true"/>-->
```

大多数情况下，为了更好的穿越 NAT，FreeSWITCH 会自动调整 RTP 包的 IP 地址，但在某些情况下（尤其是在 mod_dingaling 中会有多个候选 IP），FreeSWITCH 可能会改变本来正确的 IP 地址。该参数禁用此功能。

```
1 <!--<param name="inbound-use-callid-as-uuid" value="true"/>-->
```

在 FreeSWITCH 是，每一个 Channel 都有一个 UUID，该 UUID 是由系统生成的全局唯一的。对于来话，你可以使用 SIP 中的 callid 字段来做 UUID. 在某些情况下对于信令的跟踪分析比较有用。

```
1 <!--<param name="outbound-use-uuid-as-callid" value="true"/>-->
```

与上一个参数差不多，只是在去话时可以使用 UUID 作为 callid。

```
1 <!--<param name="rtp-autofix-timing" value="false"/>-->
```

RTP 自动定时。如果语音质量有问题，可以尝试将该值设成 false。

```
1 <!--<param name="pass-callee-id" value="false"/>-->
```

默认情况下 FreeSWITCH 会设置额外的 X- SIP 消息头，在 SIP 标准中，所有 X- 打头的消息头都是应该忽略的。但并不是所有的实现都符合标准，所以在对方的网关不支持这种 SIP 头时，该选项允许你关掉它。

```
1 <!-- clear clears them all or supply the name to add or the name prefixed with ~ to remove
2     valid values:
3
4     clear
5     CISCO_SKIP_MARK_BIT_2833
6     SONUS_SEND_INVALID_TIMESTAMP_2833
7
8     -->
9 <!--<param name="auto-rtp-bugs" data="clear"/>-->
```

某些运营商的设备不符合标准。为了最大限度的支持这些设备，FreeSWITCH 在这方面进行了妥协。使用该参数时要小心。

```
1 <!-- the following can be used as workaround with bogus SRV/NAPTR records -->
2 <!--<param name="disable-srv" value="false" />-->
3 <!--<param name="disable-naptr" value="false" />-->
```

这两个参数可以规避 DNS 中某些错误的 SRV 或 NAPTR 记录。

最后的这几个参数允许根据需要调整 sofia 库中底层的时钟，一般情况下不需要改动。

```

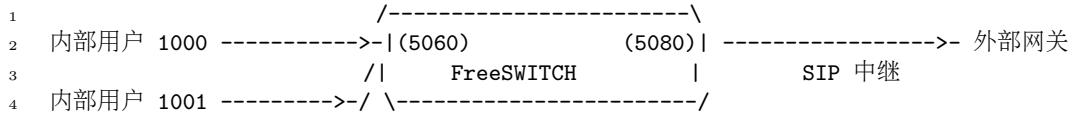
1  <!-- The following can be used to fine-tune timers within sofia's transport layer
2      Those settings are for advanced users and can safely be left as-is -->
3
4  <!-- Initial retransmission interval (in milliseconds).
5      Set the T1 retransmission interval used by the SIP transaction engine.
6      The T1 is the initial duration used by request retransmission timers A and E (UDP) as well as response retrans-
7  <!-- <param name="timer-T1" value="500" /> -->
8
9  <!-- Transaction timeout (defaults to T1 * 64).
10     Set the T1x64 timeout value used by the SIP transaction engine.
11     The T1x64 is duration used for timers B, F, H, and J (UDP) by the SIP transaction engine.
12     The timeout value T1x64 can be adjusted separately from the initial retransmission interval T1. -->
13 <!-- <param name="timer-T1X64" value="32000" /> -->
14
15
16 <!-- Maximum retransmission interval (in milliseconds).
17     Set the maximum retransmission interval used by the SIP transaction engine.
18     The T2 is the maximum duration used for the timers E (UDP) and G by the SIP transaction engine.
19     Note that the timer A is not capped by T2. Retransmission interval of INVITE requests grows exponentially
20     until the timer B fires. -->
21 <!-- <param name="timer-T2" value="4000" /> -->
22
23 <!--
24     Transaction lifetime (in milliseconds).
25     Set the lifetime for completed transactions used by the SIP transaction engine.
26     A completed transaction is kept around for the duration of T4 in order to catch late responses.
27     The T4 is the maximum duration for the messages to stay in the network and the duration of SIP timer K. -->
28 <!-- <param name="timer-T4" value="4000" /> -->
29
30 </settings>
31 </profile>
```

7.1.2 external.xml

它是另一个 UA 配置文件，它默认使用端口 5080。你可以看到，大部分参数都与 internal.xml 相同。最大的不同是 auth-calls 参数。在 internal.xml 中，auth-calls 默认是 true；而在 external.xml 中，默认是 false。也就是说，发往 5060 端口的 SIP 消息（一般只有 INVITE 消息）需要认证，而发往 5080 的消息则不需要认证。我们一般把本地用户都注册到 5060 上，所以，它们打电话时要经过认证，保证只有在们用户 directory 中配置的用户能打电话。而 5080 则不同，任何人均可以向该端口发送 SIP 请求。

如下图，本地用户 1000 注册到 5060 端口上，每次它向外打电话时（呼叫本地用户 1001 时也一样），它向 FreeSWITCH 的 5060 端口发送 INVITE 请求，FreeSWITCH 对其验证通过后（可

以是IP地址验证或Digest验证），才允许通话继续进行——如果呼叫 1001，则 FreeSWITCH 继续向用户 1001 发送 INVITE 请求；如果呼叫外部电话，则通过端口 5080 向外部网关发送 INVITE 请求。



还是看上图，在上面的例子中，FreeSWITCH 也可以通过 5060 端口对外部网关发送 INVITE 请求，效果是一样的。也就是说对于去话（outbound call, 出局通话）而言，两者在这里没有区别。但对于来话（inbound call, 入局通话）就不同了。为了让 FreeSWITCH 接收来话，一般有两种方式：

- 1) FreeSWITCH 作为一个 UAC（这里可以理解为软电话，如 X-Lite）使用从本地的 5080 端口通过 SIP 中继注册到外部网关，外部网关通过 SIP 消息中的 Contact 字段知道该 UAC 是在监听 5080 端口。当有电话进来时，外部网关就向 FreeSWITCH 所在机器的 5080 端口发送 INVITE 请求。
- 2) 某些网关是把来话和去话分开的，一般不需要或者根本不允许注册。那它们怎么知道我们的 FreeSWITCH 的 IP 和端口呢？它们一般是在申请的时候事先配置好的，就比方你去装电话时，告诉电信公司你家的门牌号一样，所不同的是这里你告诉人家的是你 FreeSWITCH 服务器的 IP 地址 和端口号（在这里是 5080）。如果有来话，外部网关就按照事先配置好的地址发送 INVITE 请求。

不管是使用以上哪种方式，当外部网关的 INVITE 请求到达 FreeSWITCH 时，我们不能对这些消息进行认证，因为外部网关不是我们本地的用户。所以我们在 external 这个 profile 中把 auth-calls 参数设为 false。

读到这里，大概你就明白了。如果你对外部网关使用 5060 的话，那么由于外部网关的来话请求不能通过验证，则 FreeSWITCH 会拒绝所有来话。当然可能你还是有疑问，既然 5080 端口允许所有来话，那么怎么保证安全呢？这个我们到后面讲到，请安全的时候会讲到，在此，就不多说了。

7.1.3 gateway

在上一节中我们已经提到，FreeSWITCH 需要通过外部网关向外打电话，而这个外部网关就称为 gateway。在 external.xml 中，我们可以看到它使用预处理指令装入了 external 目录下的所有文件：

这样做的好处是，我们可以把每个网关配置写到不同的文件中。默认的配置中包含了一个 example.xml，里面有好多配置选项。在这里，我们先从最简单的配置讲起。

参见第二章“配置SIP网关拨打外部电话”，添加一个新网关只需要在 external 目录中新建一个 XML 文件，名字可以随便起，如 gw1.xml。

```

1 <gateway name="gw1">
2   <param name="realm" value="SIP服务器地址, 可以是IP或IP:端口号"/>
3   <param name="username" value="SIP用户名"/>
4   <param name="password" value="密码"/>
5 </gateway>

```

其中，每个网关都有一个名字，由第一行 gateway name 指定。该名字与 xml 文件的名字可以相同，也可以不同。在 FreeSWITCH 内部，将以该名字唯一确定该网关。在整个 FreeSWITCH 中，网关名字必须唯一。否则，除第一个外，其它的都将被忽略。

realm 指定 SIP 网关服务器的地址，可以是域名或IP地址，如果端口号不是5060，则后面需要加上“:端口号”，如“1.2.3.4:5080”。该参数是可选的，如果没有，则默认跟 gateway name 一样。

username 和 password 指用户名和密码，这两个参数也是必需的。值得注意的是，有些网关使用 IP 地址验证，而不需要用户名和密码。但在 FreeSWITCH 中，你也必须设置这两个参数，它们的值将被忽略，所以，可以填上任意的值。

我们再来看一下 example.xml 都有哪些默认的参数默认。注意，默认情况下这些参数都是注释掉的，不起作用。

```

1 <include>
2   <!--<gateway name="asterlink.com">-->
3   <!--<param name="username" value="cluecon"/>-->
4   <!--<param name="password" value="2007"/>-->
5   <!--<param name="realm" value="asterlink.com"/>-->

```

其中，<include> 以及后面的 </include> 标签指明该文件被其它文件包含，它将在预处理阶段被去掉。而且，没有这两个标签也不会出错。但为了严谨性，建议不要省略这对标签。

```
1 <!--<param name="from-user" value="cluecon"/>-->
```

设置 SIP 消息中 From 字段的值，如果省略，则默认与 username 相同。

```
1 <!--<param name="from-domain" value="asterlink.com"/>-->
```

设置 From 字段中的 domain 值，默认与 real 相同。

```
1 <!--<param name="extension" value="cluecon"/>-->
```

来话中的分机号，默认与 username 相同。

```
1 <!--<param name="proxy" value="asterlink.com"/>-->
```

如果需要代理服务器，则设置该值，默认与 realm 相同。

```

1   <!--// send register to this proxy: *optional* same as proxy, if blank //-->
2   <!--<param name="register-proxy" value="mysbc.com"/>-->
```

如果需要注册到代理服务器，则设置该值，默认与 realm 同。

```
1   <!--<param name="expire-seconds" value="60"/>-->
```

设置注册时 Expires 字段的值，默认为 3600 秒。

```
1   <!--<param name="register" value="false"/>-->
```

如果网关不需要注册，则设为 false。默认为 true。有些网关必须需要注册才能打电话；而有的则不需要。另外，注册到网关上还允许从网关设备呼入我们的 FreeSWITCH.

```
1   <!--<param name="register-transport" value="udp"/>-->
```

设置 SIP 消息使用 udp 还是 tcp 传输。

```
1   <!--<param name="retry-seconds" value="30"/>-->
```

如果注册失败或超时，则多少秒后再重新注册。

```
1   <!--<param name="caller-id-in-from" value="false"/>-->
```

使用 From 字段中的值作为来电显示号码。

```
1   <!--<param name="contact-params" value="tport=tcp"/>-->
```

设置在 Contact 字段中额外的参数。

```
1   <!--<param name="ping" value="25"/>-->
```

每隔一段时间发送一个 Options 消息，如果失败，则会从该网关注销，并将其设置为 down 状态。通过周期性的发送无关紧要的 SIP 消息，有助于快速发现对方的状态变化，同时也有助于保持 NAT 的连接（如果在 NAT 环境中的话）。

```

1     <!--</gateway>-->
2   </include>
```

第八章 认识拨号计划

拨号计划 (*dialplan*) 是 FreeSWITCH 中至关重要的一部分。它的主要作用就是对电话进行路由 (从这一点上来说, 相当于一个路由表)。说的简明一点, 就是当一个用户拨号时, 对用户所拨的号码进行分析, 进而决定下一步该做什么。当然, 实际上, 它所能做的比你想象的要强大的多。

我们在第二章中已经提到过修改过拨号计划, 单从配置文件看, 还算比较简单直观。实际上, 它的概念也不是很复杂。如果你理解正则表达式, 那你应该能看懂系统自带的大部分的配置。但是, 在实际应用中, 有许多问题还是常常令初学者感到疑惑。主要的问题是, 要理解 Dialplan, 还需要了解 FS 是怎样工作的 (第五章), API 与 App 的区别等。

通过本章, 我们除了要了解 Dialplan 的基本概念和运作方式, 还要以理论与实践相结合的方式来进行学习, 使用初学者能快速上手, 有经验的人也能学到新的维护和调试技巧。

8.1 XML Dialplan

Dialplan 是 FreeSWITCH 中一个抽象的部分, 它可以支持多种不同的格式, 如类似 Asterisk 的格式 (由 mod_dialplan_asterisk 提供)。但在实际使用中, 用的最多的还是 XML 格式。下面, 我们就先讨论这种格式。

8.1.1 配置文件的结构

拨号计划的配置文件在 conf/dialplan 中, 在前面的章节中我们讲过, 它们是在 freeswitch.xml 中, 由 <X-PRE-PROCESS cmd="include" data="dialplan/*.xml"/> 装入的。

拨号计划由多个 Context (上下文/环境) 组成。每个 Context 中有多个 Extension (分支, 在简单的 PBX 中也可以认为是分机号, 但很显然, Extension 涵盖的内容远比分机号多)。所以, Context 就是多个 Extension 的逻辑集合, 它相当于一个分组, 一个 Context 中的 Extension 与其它 Context 中的 Extension 在逻辑上是隔离的。

下面是 Dialplan 的完整结构:

```

1  <?xml version="1.0"?>
2  <document type="freeswitch/xml">
3      <section name="dialplan" description="Regex/XML Dialplan">
4          <context name="default">
5              <extension name="Test Extension">
6                  </extension>
7          </context>
8      </section>
9  </document>

```

Extension 相当于路由表中的表项，其中，每一个 Extension 都有一个 name 属性。它可以是任何合法的字符串，本身对呼叫流程没有任何影响，但取一个好听的名字，有助于你在查看 Log 时发现它。

在 Extension 中可以对一些 condition （条件）进行判断，如果满足测试条件所指定的表达式，则执行相对应的 action （动作）。

例如，我们将下列 Extension 配置加入到 conf/dialplan/default.xml 中。并作为第一个 Extension。

```

1  <extension name="My Echo Test">
2      <condition field="destination_number" expression="^echo|1234$">
3          <action application="echo" data="" />
4      </condition>
5  </extension>

```

FreeSWITCH 安装时，提供了很多例子，为了避免与提供的例子冲突，强烈建议在学习时把自己写的 Extension 写在最前面。当然我说的最前面并不是 default.xml 的第一行，而是放到第一个 Extension 的位置，就是以下语句的后面（你通常能在第13-14行找到它们）：

```

1  <include>
2      <context name="default">

```

用你喜欢的编译器编辑好并存盘后，在 FreeSWITCH 命令行上（Console 或 fs_cli）执行 reloadxml 或按 F6 键，使 FreeSWITCH 重新读入你修改过的配置文件。并按 F8 键将 log 级别设置为 DEBUG，以看到详细日志。然后，将软电话注册上，并拨叫 1234 或 echo（大部分软电话都能呼叫字母，如 Zoiper，Xlite 可以使用空格键切换数字和字母）。

你将会看到很多 Log，注意如下的行：

```

1  Processing Seven <1000>->1234 in context default
2  parsing [default->My Echo Test] continue=false
3  Regex (PASS) [Echo Test] destination_number(1234) =~ /echo|1234$/ break=on-false
4  Action echo()

```

在我的终端上，上面的第一行是以绿色显示的。当然，为了排版方便，我省去了 Log 中的日期以及其它不关键的一些信息。

第一行，Processing 说明是在处理 Dialplan，**Seven** 是我的的 SIP 名字，**1000** 是我的分机号，**1234** 是我所拨叫的号码，这里，我直接拨叫了 1234。它完整意思是说，呼叫已经达到路由阶段，要从 XML Dialplan 中查找路由，该呼叫来自 Seven，分机号是1000，它所呼叫的被叫号码是 1234（或 echo，如果你拨打 echo 的话）。

第二行，呼叫进入 parsing (解析XML) 阶段，它首先找到 XML 中的一个 Context，这里是 **default**（它是在 user directory 中定义的，看第五章。user directory 中有一项 <variable name="user_context" value="default"/>，说明，如果 1000 这个用户发起呼叫，则它的 context 就是 default，所以要从 XML Dialplan 中的 default 这个 Context 查起）。它首先找到的第一个 Extension 的 name 是 **My Echo Test**（还记得吧？我们把它放到了 Dialplan 的最前面）。continue=false 的意思我们后面再讲。

第三行，由于该 Extension 中有一个 Condition，它的测试条件是 **destination_number**，也就是被叫号码，所以，FreeSWITCH 测试被叫号码（这里是 1234）是否与配置文件中的正则表达式相匹配。**^echo|1234\$** 是正则表达式，它匹配 echo 或 1234。所以这里匹配成功，Log 中显示 Regex (PASS)。当然既然匹配成功了，它就开始执行动作 echo（它是一个 App，在第 6 章我们讲过，这些App大部分来自于 [mod_dptools](#)），所以你就听到了自己的声音。

这是最简单的路由查找。前面我已经说了，系统自带了一些 Dialplan 的例子，也许在第二章你已经测试过了。下面，我们试一下系统自带的 echo 的例子。这次，我呼叫的是 9196。在 Log 中，还是从绿色的行开始看：

```

1 Processing Seven <1000>->9196 in context default
2 parsing [default->My Echo Test] continue=false
3 Regex (FAIL) [Echo Test] destination_number(9196) =~ /~echo|1234$/ break=on-false
4 parsing [default->unloop] continue=false
5 Regex (PASS) [unloop] ${unroll_loops}(true) =~ /~true$/ break=on-false
6 Regex (FAIL) [unloop] ${sip_looped_call}() =~ /~true$/ break=on-false
7 parsing [default->tod_example] continue=true
8 Date/Time Match (FAIL) [tod_example] break=on-false
9 parsing [default->holiday_example] continue=true
10 Date/Time Match (FAIL) [holiday_example] break=on-false
11 parsing [default->global-intercept] continue=false
12 Regex (FAIL) [global-intercept] destination_number(9196) =~ /~886$/ break=on-false
13 parsing [default->group-intercept] continue=false
14 Regex (FAIL) [group-intercept] destination_number(9196) =~ /~\*\*$/ break=on-false
15 parsing [default->intercept-ext] continue=false
16 Regex (FAIL) [intercept-ext] destination_number(9196) =~ /~\*\*(\d+)$/ break=on-false
17 parsing [default->redial] continue=false
18 Regex (FAIL) [redial] destination_number(9196) =~ /~(redial|870)$/ break=on-false
19 parsing [default->global] continue=true
20 Regex (FAIL) [global] ${call_debug}(false) =~ /~true$/ break=never
21
22 [] [] [] [] [] 此处省略五百字...
23
24 parsing [default->fax_receive] continue=false
25 Regex (FAIL) [fax_receive] destination_number(9196) =~ /~9178$/ break=on-false
26 parsing [default->fax_transmit] continue=false
27 Regex (FAIL) [fax_transmit] destination_number(9196) =~ /~9179$/ break=on-false
28 parsing [default->ringback_180] continue=false
29 Regex (FAIL) [ringback_180] destination_number(9196) =~ /~9180$/ break=on-false
30 parsing [default->ringback_183_uk_ring] continue=false
31 Regex (FAIL) [ringback_183_uk_ring] destination_number(9196) =~ /~9181$/ break=on-false
32 parsing [default->ringback_183_music_ring] continue=false
33 Regex (FAIL) [ringback_183_music_ring] destination_number(9196) =~ /~9182$/ break=on-false
34 parsing [default->ringback_post_answer_uk_ring] continue=false
35 Regex (FAIL) [ringback_post_answer_uk_ring] destination_number(9196) =~ /~9183$/ break=on-false
36 parsing [default->ringback_post_answer_music] continue=false
37 Regex (FAIL) [ringback_post_answer_music] destination_number(9196) =~ /~9184$/ break=on-false
38 parsing [default->ClueCon] continue=false
39 Regex (FAIL) [ClueCon] destination_number(9196) =~ /~9191$/ break=on-false
40 parsing [default->show_info] continue=false
41 Regex (FAIL) [show_info] destination_number(9196) =~ /~9192$/ break=on-false
42 parsing [default->video_record] continue=false
43 Regex (FAIL) [video_record] destination_number(9196) =~ /~9193$/ break=on-false
44 parsing [default->video_playback] continue=false
45 Regex (FAIL) [video_playback] destination_number(9196) =~ /~9194$/ break=on-false
46 parsing [default->delay_echo] continue=false
47 Regex (FAIL) [delay_echo] destination_number(9196) =~ /~9195$/ break=on-false
48 parsing [default->echo] continue=false
49 Regex (PASS) [echo] destination_number(9196) =~ /~9196$/ break=on-false
50 Action answer()
51 16tion echo() ----- D R A F T -----

```

你可以看到，前面的正则表达式匹配都没有成功（Regex (FAIL)），只是到最后匹配到 ^9196\$ 才成功（你看到 Regex (PASS) 了），成功后先应答（answer），然后执行 echo。

在这一节里，我们花了很多篇幅来讲解如此简单的问题。但实际上，我是想让你知道，这一节最重要的不是讲 Dialplan，而是告诉你如何看 Log。在邮件列表上，大多数新手遇到的问题都可以很轻松的从 Log 中看出来，但他们不知道怎么看，或者是看了也不理解。所以，在这里，我想请你再看一下我们的第一个例子。永远记住：遇到 Dialplan 的问题，从绿色的行开始看起（当然，如果你的终端不能显示颜色，那么，从 Processing 一行看起）。我们的第一个例子虽然只有短短的四行 Log，但是它包含了所有你需要的信息。

8.1.2 默认的配置文件结构

系统默认提供的配置文件包含三个 Context: default、features 和 public，它们分别在三个 XML 文件中。default 是默认的 dialplan，一般来说注册用户都可以使用它来打电话，如拨打其它分机或外部电话等。而 public 则是接收外部呼叫，因为从外部进来的呼叫是不可信的，所以要进行更严格的控制。如，你肯定不想从外部进来的电话再通过你的网关进行国内或国际长途呼叫。

当然，这么说不是绝对的，等你熟悉了 Dialplan 的概念之后，可以发挥你的想象力进行任何有创意的配置。

其中，在 default 和 public 中，又通过 INCLUDE 预处理指令分别加入了 default/ 和 include/ 目录中的所有 XML 文件。这些目录中的文件仅包含一些额外的 Extension。由于 Dialplan 在处理时是顺序处理的，所以，一定要注意这些文件的装入顺序。通常，这些文件都按文件名排序，如 00_，01_ 等等。如果你新加入 Extension，可以在这些目录里创建文件。但要注意，这些文件的优先级比直接写在如 default.xml 中低。我前面已经说过，由于你不熟悉系统提供的默认的 Dialplan，很可能出现与系统冲突的情况。当然，你已经学会如何查看 Log，所以能很容易的找到问题所在。但在本书中，我还是坚持将新加的 Extension 加在 Dialplan 中的最前面，以便于说明问题。

实际上，由于在处理 Dialplan 时要对每一项进行正则表达式匹配，是非常影响效率的。所以，在生产环境中，往往要删除这些默认的 Dialplan，而只配置有用的部分。但我们还不能删，因为里面有好多例子我们可以学习。

8.1.3 正则表达式

Dialplan 使用 Perl 兼容的正则表达式（PCRE, Perl-compatible regular expressions）匹配。熟悉编程的同学肯定已经很熟悉它了，为了方便不熟悉的同学，在这里仅作简单介绍：

```

1 ^1234$          ^ 匹配字符串开头, $ 匹配结尾, 所以本表达式严格匹配 1234
2 ^1234|5678$    | 是或的意思, 表示匹配 1234 或 5678
3 ^123[0-9]$     [ ] 表式匹配其中的任意一个字符, 其中的 - 是省略的方式, 表示 0 到 9, 它等于 [0123456789]
4                               也就是说它会匹配 1230, 1231, 1232 ... 1239
5 ^123\d$         同上, \d 等于 [0-9]
6 ^123\d+$        + 号表示1个或多个它前面的字符, 因为 + 前面是 \d, 所以它就等于1个或多个数字, 实际上,
7                               它匹配任何以123开头的至少4位数的数字串, 如1230, 12300, 12311, 123456789等
8 ^123\d*$        *号与+号的不同在于, 它匹配0个或多个前面的字符。
9                               所以, 它匹配以123开头的至少3位数的数字串, 如 123, 123789
10 ^123            跟上面一样, 由于没有结尾的$, 它匹配任何以123开头的数字串, 但除此之外,
11                               它还匹配后面是字母的情况, 如 123abc
12 123$           匹配任何以123结尾的字符串
13 ^123\d{5}$      {5}表示精确匹配5位, 包含它前面的一个字符。在这里, 它匹配以123开头的所有8位的电话号码
14 ^123(\d+)$     ( )在匹配中不起作用, 跟^123\d+是相同的, 但它对匹配结果有作用,
15                               匹配结果中除123之外的数字都将存储在$1这个变量中, 在下一步使用
16 ^123(\d)(\d+)$ 如果用它跟12345678匹配, 则匹配成功, 结果是 $1 = 4, $2 = 5678
17 .               最后说明, "." 匹配任意一个字符, 如果你写了 .* , 则它会匹配任意字符串

```

FreeSWITCH 提供了简单的 API 可以测试你写的正则表达式是否正确, 只需要在命令行上输入“`regex` 要匹配的字符串 | 正则表达式” 。如:

```

1 freeswitch> regex 1234 | \d
2 true
3 freeswitch> regex 1234 | \d{4}
4 true
5 freeswitch> regex 1234 | \d{5}
6 false
7 freeswitch> regex 1234 | ^123
8 true

```

简单的正则表达式比较容易理解, 更深入的学习请查阅相关资料。正则表达式功能很强大, 但配置不当也容易出现错误, 轻者造成电话不通, 重者可能会造成误拨或套拨, 带来经济损失。

参考资料: <http://zh.wikipedia.org/zh/正则表达式> , http://wiki.freeswitch.org/wiki/Regular_Expression 。

8.1.4 信道变量 - Channel Variables

在 FreeSWITCH 中, 每一次呼叫都由一条或多条“腿”(Call Leg)组成, 其中的一条腿又称为一个 Channel (信道), 每一个 Channel 都有好多属性, 用于标识 Channel 的状态, 性能等, 这些属性称为 Channel Variable (信道变量), 简写为 Channel Var 或 Chan Var 或 Var。

通过使用 `info` 这个 App, 可以查看所有的 Channel Var。我们先修改一下 Dialplan。

```

1 <extension name="Show Channel Variable">
2   <condition field="destination_number" expression="^1235$">
3     <action application="info" data="" />
4   </condition>
5 </extension>

```

加入 default.xml 中，为了复习上一节的内容，我们这一次加入 My Echo Test 这一 Extension 的后面，存盘，然后在 FreeSWITCH 命令行上执行 reloadxml。从软电话上呼叫 1235，可以看到有很多 Log 输出，还是从绿色的行开始看：

```

1 Processing Seven <1000>->1235 in context default
2 parsing [default->Echo Test] continue=false
3 Regex (FAIL) [Echo Test] destination_number(1235) =~ /^echo|1234$/ break=on-false
4 parsing [default->Show Channel Variable] continue=false
5 Regex (PASS) [Show Channel Variable] destination_number(1235) =~ /^1235$/ break=on-false
6 Action info()
7 ...
8 EXECUTE sofia/internal/1000@192.168.7.10 info()
9 2010-10-23 09:46:31.662281 [INFO] mod_dptools.c:1171 CHANNEL_DATA:
10 Channel-State: [CS_EXECUTE]
11 Channel-Call-State: [RINGING]
12 Channel-State-Number: [4]
13 Channel-Name: [sofia/internal/1000@192.168.7.10]
14 Unique-ID: [cfea988b-2dc4-42ec-b731-2cd7ea864fc6]
15 Caller-Direction: [inbound]
16 Caller-Username: [1000]
17 Caller-Dialplan: [XML]
18 Caller-Caller-ID-Name: [Seven]
19 Caller-Caller-ID-Number: [1000]
20 Caller-Network-Addr: [192.168.7.10]
21 Caller-ANI: [1000]
22 Caller-Destination-Number: [1235]
23 variable_direction: [inbound]
24 variable_uuid: [cfea988b-2dc4-42ec-b731-2cd7ea864fc6]
25 variable_sip_local_network_addr: [123.130.140.154]
26 variable_remote_media_ip: [123.130.140.154]
27 variable_remote_media_port: [8000]
28 variable_sip_use_codec_name: [PCMA]
29 variable_sip_use_codec_rate: [8000]
30 variable_sip_use_codec_ptime: [20]
31 variable_read_codec: [PCMA]
32 variable_read_rate: [8000]
33 variable_write_codec: [PCMA]
34 variable_write_rate: [8000]
35 variable_endpoint_disposition: [RECEIVED]
36 variable_current_application: [info]

```

为节省篇幅，我们删去了一部分。

可以看到，由于我们呼叫的是 1235，它在第三行测试 My Echo Test 的 1234 的时候失败了，接在接下来测试 1235 的时候成功了，便执行相对应的 Action - info 这个App。它的作用就是把所有 Channel Variables 都打印到 Log 中。

所有的 Channel Variable 都是可以在 Dialplan 中访问的，使用格式是 \${变量名}，如 \${destination_number}。将下列配置加入 Dialplan 中（存盘，reloadxml 不用再说了吧？）：

```

1 <extension name="Accessing Channel Variable">
2   <condition field="destination_number" expression="^1236(\d+)$">
3     <action application="log" data="INFO Hahaha, I know you called ${destination_number}"/>
4     <action application="log" data="INFO The Last few digits is $1"/>
5     <action application="log" data="ERR This is not actually an error, just jocking"/>
6     <action application="hangup"/>
7   </condition>
8 </extension>

```

这次我们呼叫 1236789，看看结果：

```

1 Processing Seven <1000>->1236789 in context default
2 parsing [default->Echo Test] continue=false
3 Regex (FAIL) [Echo Test] destination_number(1236789) =~ /echo|1234$/ break=on-false
4 parsing [default->Show Channel Variable] continue=false
5 Regex (FAIL) [Show Channel Variable] destination_number(1236789) =~ /^1235$/ break=on-false
6 parsing [default->Accessing Channel Variable] continue=false
7 Regex (PASS) [Accessing Channel Variable] destination_number(1236789) =~ /^1236(\d+)$/ break=on-false
8 Action log(INFO Hahaha, I know you called ${destination_number})
9 Action log(NOTICE The Last few digits is 789)
10 Action log(ERR This is not actually an error, just jocking)
11 Action hangup()
12
13 [DEBUG] switch_core_state_machine.c:157 sofia/internal/1000@192.168.7.10 Standard EXECUTE
14
15 EXECUTE sofia/internal/1000@192.168.7.10 log(INFO Hahaha, I know you called 1236789)
16 [INFO] mod_dptools.c:1152 Hahaha, I know you called 1236789
17 EXECUTE sofia/internal/1000@192.168.7.10 log(NOTICE The Last few digits is 789)
18 [NOTICE] mod_dptools.c:1152 The Last few digits is 789
19 EXECUTE sofia/internal/1000@192.168.7.10 log(ERR This is not actually an error, just jocking)
20 [ERR] mod_dptools.c:1152 This is not actually an error, just jocking
21 EXECUTE sofia/internal/1000@192.168.7.10 hangup()

```

跟前面一样，我们还是从绿色的行开始看。这一次，1236789 匹配了正则表达式 ^1236(+)，并将 789 存储在变量 \$1 中。然后在 8-11 行看到它解析出的四个 Action（三个 log 一个 hangup）。到这里为止，Channel 的状态一直没有变，还处在路由查找的阶段。在所有 Dialplan 解析完成后，Channel 状态才进行 Standard Execute 阶段。理解这一点是非常重要的，我们后面再做详细说明，但是在这里你要记住路由查找（解析）和执行分属于不同的阶段。当 Channel 状态进入执行阶段后，它才开始依次执行所有的 Action。log() 的作用就是将信息写到 Log 中，

它的第一个参数是 `lelevel`, 就是 Log 的级别, 有 INFO、Err、DEBUG 等, 不同的级别在彩色的终端上能以不同的颜色显示。 (详细的级别请参考http://wiki.freeswitch.org/wiki/Mod_logfile#Log_Levels) 。

你肯定看到彩色的 Log 了, 同时也看到了用 \$ 表示的 Channel Variable 被替换成了相应的值。

同时你也看到, 这次实验我们特意增加了几个 Action。一个 Action 通常有两个参数, 一个是 `application`, 代表要执行的 App, 另一个是 `data`, 就是 App 的参数, 当 App 没有参数时, `data` 也可能省略。

一个 Action 必须是一个合法的 XML 标签, 在前面, 你看到的 `context`, `extension` 等都是成对出现的, 如 `<extension> </extension>`。但由于 Action 比较简单, 一般采用简写的形式来关闭标签, 即 `<action />`。注意大于号前面的 “/”, 如果不小心漏掉, 在 `reloadxml` 时将会出现类似“+OK [[error near line 3371]: unexpected closing tag </condition>]”的错误, 而实际的错误位置又通常不是出错的那一行。这是在编辑 XML 文件时经常遇到的问题, 又比较难于查找。因此在修改时要多加小心, 并推荐使用具有语法高亮的功能的编辑器来编辑。

读到这里, 你或许还有疑问, 既然我们在 `info App` 的输出里没看到 `destination_number` 这一变量, 它到底是从哪里来的呢? 是这样的, 它在 `info` 中的输出是 `Caller-Destination-Number`, 但你在引用的时候就需要使用 `destination_number`。还有一些变量, 在 `info` 中的输出是 `variable_xxxx`, 如 `variable_domain_name`, 而实际引用时要去掉 `variable_` 前缀。不要紧张, 这里有一份对照表: http://wiki.freeswitch.org/wiki/Channel_Variables#Info_Application_Variable_Names_.28variable_xxxx.29

8.1.5 测试条件 - Conditions

在上面我们已经看到了最简单的测试条件, `<condition field="destination_number" expression="^1234$">`, 它使用正则表达式匹配测试一个变量是否满足预设的正则表达式。大部分的测试都是针对被叫号码 (`destination_number`) 的, 但你也可以对其它变量进行测试, 如 IP 地址 (注意由于正则表达式中 “.” 具有特殊意义, 所有需要用 “\.” 进行转义) :

```
1 <condition field="network_addr" expression="^192\.168\.7\.7$">
```

下面列出了所有的测试条件:

```

1 context      Dialplan 当前的 Context
2 rdnis        被转移的号码（在呼叫转移中设置）。
3 destination_number 被叫号码
4 Dialplan    Dialplan 模块的名字，如 XML, YAML, inline, asterisk, enum 等
5 caller_id_name 主叫（来电显示）的名称
6 caller_id_number 主叫号码
7 ani          主叫的自动号码识别 (Automatic Number Identification)
8 aniii        主叫类型，如投币电话 (payphone)
9 uuid         本 Channel 的唯一标志
10 source      呼叫源，来自哪一个FreeSWITCH 模块，如 mod_portaudio, mod_sofia等
11 chan_name   Channel 名字，如 portaudio/1234
12 network_addr 主叫的 IP 地址
13 year         当前的年，0-9999
14 yday        一年中的第几天，1-366
15 mon         月，1-12
16 mday        日，1-31
17 week        一年中的第几周，1-53
18 mweek       本月中的第几周，1-6
19 wday        一周中的第几天，1-7（周日等于 = 1，周一等于 = 2，有点奇怪吧？）
20 hour        小时，0-23
21 minute      分，0-59
22 minute-of-day 一天中的第几分钟，(1-1440)（午夜 = 1，1点 = 60，中午 = 720）

```

当然，除此之外，还接受用户在 user directory 中设置的变量（参见第5章），但要注意使用 \${} 对变量进行引用，如：

```
1 <condition field="${toll_allow}" expression="international">
```

如果在你的用户目录中设置了以下变量，则你可以进行相关测试：

```

1 <user id="1000">
2   <variables>
3     <variable name="my_test_var" value="my_test_value"/>
4   </variables>
5 </user>
6
7 <condition field="${my_test_var}" expression="my_test_value">
```

测试条件不可以嵌套，但可以迭加，如

```

1 <extension name="Testing Stacked Conditions">
2   <condition field="network_addr" expression="^192\.168\.7\.7$">
3     <condition field="destination_number" expression="^1234$">
4       <action application="log" data="INFO Hahaha, I know you called ${destination_number}"/>
5     </condition>
6   </condition>
7 </extension>
```

是错误的，但以下面是正确的：

```

1 <extension name="Testing Stacked Conditions">
2   <condition field="network_addr" expression="^192\.168\.7\.7$"/>
3   <condition field="destination_number" expression="^1234$">
4     <action application="log" data="INFO Hahaha, I know you called ${destination_number}"/>
5   </condition>
6 </extension>
```

注意第一个 condition，它的标签是在一行内用简写的形式关闭的，实际上，它等价于

```
1 <condition field="network_addr" expression="^192\.168\.7\.7$">></condition>
```

所以，它于下面测试 destination_number 的 condition 是迭加的关系。因此就构成了一个简单的“逻辑与”的关系，即，如果 IP 地址不匹配，或者 IP 匹配了但被叫号码不匹配，则跳过本 extension，继续解析下一项。因此我们说这两个平行的 condition 是“逻辑与”的关系。

除此之外，两个迭加在一起的 condition 还可以构成其它关系，使你可以只用 XML 就能完成比较复杂的路由配置，而无需编程。**break**参数就是做这个用的，它有以下几个值（为方便讨论，假设两个条件分别为 A 和 B）：

- on-false 在第一次匹配失败时停止（但继续处理其它的 extension），这是缺省的配置。结果相当于 A and B。
- on-true 在第一次匹配成功时停止（但会先完成对应的 Action，然后继续处理其它的 extension），不成功则继续，所以，结果是 ((not A) and B)
- always 不管是否匹配，都停止。
- never 不管是否匹配，都继续。

通过使用 **break** 参数，你可以写出类似 if-then-else 的结构。如上面的例子（因为没有 break 参数，所以默认是 on-false）就是：

```

1 //伪代码， =~ 表示正则表达式匹配， // 是注释，下同
2 if(network_addr =~ ^192\.168\.7\.7$/) then
3   if(destination_number =~ ^1234$/) then
4     //wirte log
5   end
6 end
```

我们再来看一个例子，假设你拨打 1234，如果来自 192.168.7.7这个IP，就播放早上好，如果来自 192.168.7.8，就说晚上好。则你可以这样设置：

```

1 <extension name="Testing Stacked Conditions">
2   <condition field="destination_number" expression="^1234$"/>
3   <condition field="network_addr" expression="^192\.168\.7\.7$"/>
4     <action application="playback" data="good-morning.wav"/>
5   </condition>
6 </extension>
7
8 <extension name="Testing Stacked Conditions">
9   <condition field="destination_number" expression="^1234$"/>
10  <condition field="network_addr" expression="^192\.168\.7\.8$"/>
11    <action application="playback" data="good-night.wav"/>
12  </condition>
13 </extension>

```

通过使用 **break** 参数，你就可以将两种情况写到一个 extension 里：

```

1 <extension name="Testing Stacked Conditions">
2   <condition field="destination_number" expression="^1234$"/>
3   <condition field="network_addr" expression="^192\.168\.7\.7$" break="on-true">
4     <action application="playback" data="good-morning.wav"/>
5   </condition>
6   <condition field="network_addr" expression="^192\.168\.7\.8$"/>
7     <action application="playback" data="good-night.wav"/>
8   </condition>
9 </extension>

```

这种情况跟上面两个 extension 的情况是等价的。虽然看起来不如第一种情况容易理解，但它把相似的功能逻辑上放到一个 extension 里，却比较直观。如果你从 192.168.7.7 上呼叫 1234，它会首先匹配 1234，进而匹配网络地址，匹配成功，便不再往下进行，但是，它会先把已经匹配到的条件中的 Action 执行完毕，所以，播放 good-morning。若你从 192.168.7.8 上呼叫，则它不能匹配 192.168.7.7，但由于 break 参数的值是 on-true，所以，在这里，它不会中止，而是会继续尝试匹配下面的 condition，进而播放 good-night。所以，它相当于：

```

1 if(destination_number =~ /^1234$/) then
2   if(network_addr =~ /^192\.168\.7\.7$/) then
3     // play good morning
4   else if(network_addr =~ ^192.168.7.8$) then
5     // play good night
6   end
7 end

```

always 和 never 不常用，发挥你的想象力，可以创造出类似 if a then b end; if c then d end; 之类的条件。

8.1.6 动作与反动作 - Action & Anti-Action

除使用 condition 的 break 机制来完成复杂的条件以外，你还可以使用“反动作”来达到类似的目的，如：

```

1 <extension name="Anction and Anti-Action">
2   <condition field="destination_number" expression="^1234$"/>
3   <condition field="network_addr" expression="^192\\.168\\.7\\.7$">
4     <action application="playback" data="good-morning.wav"/>
5     <anti-action application="playback" data="good-night.wav"/>
6   </condition>
7 </extension>
```

它说明，如果呼叫来自 192.168.7.7，则播放 good morning，否则，播放 good night（不管是不是来自 192.169.7.8）。因此，你可以看到，它没有 condition 条件那么强大，但在简单的条件下也经常使用，它相当于：

```

1 if(destination_number =~ /~1234$/) then
2   if(network_addr =~ /~192\\\.168\\\.7\\\.7$/) then
3     // play good morning
4   else
5     // play good night
6 end
7 end
```

8.1.7 工作机制深入剖析

在进一步了解 Dialplan 的工作机制之前，我们先来看一下 Channel 的状态机。如下图：

```

1 NEW --> INIT --> ROUTING --> EXECUTE --> HANGUP --> REPORTING --> DESTROY
2           |-----<----|-----|
3           Transfer
```

当新建（NEW）一个 Channel 时，它首先会初始化（INIT），然后进入路由（ROUTING）阶段，也就是我们查找解析 Dialplan 的时候。在这里，专门有一个术语叫 Hunting（在传统的交换机里，它译为选线，在这里，我就译为选路吧）。找到合适的路由入口后，它会执行（EXECUTE）一系列动作，最后无论哪一方挂机，都会进入挂机（HANGUP）阶段。后面的报告阶段一般用行统计、计费等，最后将 Channel 销毁，释放系统资源。

在 EXECUTE 状态，可能会发生转移（Transfer，该转移跟我们通常说的呼叫转移不太一样），它可以转移到同一 Context 下其它的 Extension，或者转移到其它 Context 下的 Extension，但无论哪种转移，发生转移时，都会重新进行路由，也就是重新 Hunt Dialplan。

我们在前面的章节也讲过，一定要记住 ROUTING 和 EXECUTE 是属于两个不同阶段的，只有 ROUTING 完毕后才会进行 EXECUTE 阶段的操作。当一个 Channel 进入 ROUTING 阶段

时，它首先会查找 Dialplan（英文叫 hit the Dialplan），即对 Dialplan 进行解析（是的，每个电话都会重新解析 Dialplan），解析 Dialplan 这一过程称为 Hunting。解析完毕（成功）后，会得到一些 Action，然后 Channel 进入 EXECUTE 阶段，依次执行所有的 Action。

我们用另一种方式实现类似上面的 good-morning/good-night 面的例子，但这一次稍稍有点不同：用户呼叫 1234，如果来自 192.168.7.7，则播放 good morning，如果来自 192.168.7.8，则播放 good night，否则，什么都不做。（注意：这个例子是错误的，它不会达到你预期的结果，但这是新手常常犯的错误，而遇到这种情况，大多数新手都会以为 FreeSWITCH 出问题了）：

```

1 <extension name="Testing Hunting and Executing" continue="true">
2   <condition>
3     <action application="set" data="greeting=no-greeting.wav"/>
4   </condition>
5 </extension>
6
7 <extension name="Testing Hunting and Executing" continue="true">
8   <condition field="network_addr" expression="^192\.168\.7\.7$">
9     <action application="set" data="greeting=good-morning.wav"/>
10  </condition>
11 </extension>
12
13 <extension name="Testing Hunting and Executing" continue="true">
14   <condition field="network_addr" expression="^192\.168\.7\.8$">
15     <action application="set" data="greeting=good-night.wav"/>
16   </condition>
17 </extension>
18
19 <extension name="Testing Hunting and Executing">
20   <condition field="destination_number" expression="^1234$"/>
21   <condition field="${greeting}" expression="^good">
22     <action application="playback" data="${greeting}"/>
23   </condition>
24 </extension>
```

首先，也许你已经看到，我们在前三个 extension 中都增加了一个参数 continue="true"。如果没有该参数，则默认为 false。在默认的情况下（试想一下前面的例子），在 Dialplan 的 Hunting 阶段，一旦根据前面介绍的 condition 匹配规则找到对应的 extension，就执行相应的 Action，而不会再继续查找其它的 extension 了，即使后面的 extension 也有可能匹配。

但有些情况下，Dialplan 中会有多个 extension 满足匹配规则，而我们希望所有对应的 Action 都能得到执行，我们就使用 continue="true" 参数。

此外，我们这次还在第一个 extension 中用了一个空的 condition，这个空的 condition 没有匹配规则，因此，它被认为匹配任何规则。所以，从表面上看，它应该等价于：

```

1 $greeting = no-greeting.wav
2
3 if (network_addr =~ /^192\.168\.7\.7$/) then
4     $greeting = good-morning.wav
5 else if (network_addr =~ /192\.168\.7\.8$/) then
6     $greeting = good-night.wav
7 end
8
9 if (destination_number =~ /~1234$/ and $greeting =~ /~good/) then
10    play $greeting
11 end

```

但实际上两者不是等价的。原因在于，对 Dialplan 的 Hunting 和 Executing 分属于不同的阶段。在 Hunting 阶段，只解析 Dialplan，并将所到的所有满足条件的 Action 都放到一个动作列表（队列）中，待呼叫流程进行到 Executing 阶段时，再依次执行动作列表中的动作。所以，上述的 XML 等价于：

```

1 $action_list[0] = "$greeting = no-greeting.wav"
2
3 if (network_addr =~ /^192\.168\.7\.7$/) then
4     $action_list[1] = "$greeting = good-morning.wav"
5 else if (network_addr =~ /192\.168\.7\.8$/) then
6     $action_list[1] = "$greeting = good-night.wav"
7 end
8
9 if (destination_number =~ /~1234$/ and $greeting =~ /~good/) then
10    $action_list[2] = "play $greeting"
11 end
12
13 //开始执行命令列表中的命令
14
15 foreach $action in $action_list do
16     execute $action
17 end

```

这下应该找出问题所在了。因为你已经看到，在 Hunting 阶段，\$greeting 这个变量的值始终是空值，因为没有执行任何动作设置这个值。所以在测试 \$greeting 与正则表达式 /~good/ 是否匹配时，永远是不通过的。因此最后的动作列表中，也只有两个动作 action_list[0] 和 action_list[1]。虽然在执行阶段 \$greeting 的值最终会被设置为 no-greeting, good-morning 或 good-night，但它再也不会回到 Hunting 阶段了（transfer 除外），因而也不可能再执行真正的 play 命令播放声音了。

好吧，我想我已经用了足够的篇幅来讲解这一问题了。如果你已经嫌我罗嗦了，那说明你已经懂了。如果你还是明白，那说明我的写作能力还有待提高。这样吧，我这里有两个建议：

1. 把这一节再看一遍

2. 把这些 XML 放到你的机器上试试，看一下 Log 的输出。并改一下某些参数对比一下 Log 有什么不同（如，把 true 改成 false）。

当然，我写了一个错误的例子，还是有责任把它改正的。请看下一节。

8.1.8 内连执行 - inline 参数

在上一节我们讲到一个错误的例子。原因是在 Dialplan 的 Hunting 阶段不执行 Action，但你却认为它应该执行。当然，如果你学会了看 Log，你是完全可以从 Log 中看出这一问题的。但实际上，大多数的新手都不会仔细地去看 Log，而且即使仔细看了，由于经验比较少，也不一定能找到问题所在。FreeSWITCH 的开发者和老手们在邮件列表中不厌其烦地解释和回答这种问题。后来，终于忍不住了，在 Action 上增加了一个 **inline** 参数。

好吧，现在就把所有带 set 的 Action 都加上 inline="true"，如：

```
1 <action inline="true" application="set" data="greeting=no-greeting.wav"/>
```

问题就迎刃而解了。是的，这就是你期望的结果。带有 inline 的 Action 在 Hunting 阶段便会执行。

改好后，认真对比一下 Log 输出与前面有什么不同。我相信，到这里你基本上完全理解 Dialplan 了。

当然，并不是所有的 App 都能 inline 执行的。适合 inline 执行的 App 必须能很快的执行，一般只是很快的存取某个变量，并且不能改变当前 Channel 的状态。

满足这样条件的 App 有：

```
1 check_acl      eval        event       export
2 log            presence    set         set_global
3 set_profile_var set_user   sleep      unset
4 verbose_events cidlookup curl       easyroute
5 enum           lcr        nibblebill odbc_query
```

当然，inline 参数也不是解决所有问题的万能钥匙，因为它会打乱执行顺序，如：

```
1 <action inline="true" application="set" data="var=1"/>
2 <action application="info"/>
3 <action inline="true" application="set" data="var=2"/>
4 <action application="info"/>
```

而实际上，在最后的输出中，两个 info 都显示 variable_var = 2。

8.1.9 实例解析

以上的论述应该涵盖了 Dialplan 的所有概念，当然，要活学活用，还需要一些经验。下面，我们讲几个真实的例子。这些例子大部分来自默认的配置文件。

Local_Extension

我们要看的第一个例子是 Local_Extension。FreeSWITCH 默认的配置提供了 1000 - 1019 共 20 个 SIP 账号，密码都是 1234。

```

1 <extension name="Local_Extension">
2   <condition field="destination_number" expression="^(10[01][0-9])$">
3     //actions
4   </condition>
5 </extension>
```

这个框架说明，用正则表达式 `(10[01][0-9])$` 来匹配被叫号码，它匹配所有 1000 - 1019 这 20 个号码。

这里我们假设在 SIP 客户端上，用 1000 和 1001 分别注册到了 FreeSWITCH 上，则 1000 呼叫 1001 时，FreeSWITCH 会建立一个 Channel，该 Channel 构成一次呼叫的 a-leg（一条腿）。初始化完毕后，Channel 进入 ROUTING 状态，即进入 Dialplan。由于被叫号码 1001 与这里的正则表达式匹配，所以，会执行下面这些 Action。另外，由于我们在正则表达式中使用了“`()`”，因此，匹配结果会放入变量 `$1` 中，因此，在这里，`$1 = 1001`。

```

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

`set` 和 `export` 都是设置一个变量，该变量的名字是 `dialed_extension`，值是 1001。

关于 `set` 和 `export` 的区别我们在前面已经讲过了。这里再重复一次：`set` 是将变量设置到当前的 Channel 上，即 a-leg。而 `export` 则也将变量设置到 b-leg 上。当然，这里 b-leg 还不存在。所以在这里它对该 Channel 的影响与 `set` 其实是一样的。因此，使用 `set` 完全是多余的。但是除此之外，`export` 还设置了一个特殊的变量，叫 `export_vars`，它的值是 `dialed_extension`。所以，实际上。上面的第二行就等价于下面的两行：

```

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

```

1 <!-- bind_meta_app can have these args <key> [a|b|ab] [a|b|o|s] <app> -->
2 <action application="bind_meta_app" data="1 b s execute_extension::dx XML features"/>
3 <action application="bind_meta_app" data="2 b s record_session::${recordings_dir}/${caller_id_number}.${strptime}">
4 <action application="bind_meta_app" data="3 b s execute_extension::cf XML features"/>
5 <action application="bind_meta_app" data="4 b s execute_extension::att_xfer XML features"/>
```

bind_meta_app 的作用是在该 Channel 是绑定 DTMF。上面四行分别绑定了 1、2、3、4 四个按键，它们都绑定到了 b-leg 上。注意，这时候 b-leg 还不存在。所以，请记住这里，我们下面再讲。

```
1 <action application="set" data="ringback=${us-ring}"/>
```

设置回铃音是美音（不同国家的回铃音是有区别的），\${us-ring} 的值是在 vars.xml 中设置的。

```
1 <action application="set" data="transfer_ringback=$${hold_music}"/>
```

设置呼叫转移时，用户听到的回铃音。

```
1 <action application="set" data="call_timeout=30"/>
```

设置呼叫超时。

```
1 <action application="set" data="hangup_after_bridge=true"/>
2 <!--<action application="set" data="continue_on_fail=NORMAL_TEMPORARY_FAILURE,USER_BUSY,NO_ANSWER,TIMEOUT"/>
3 <action application="set" data="continue_on_fail=true"/>
4
5 这些变量影响呼叫流程，详细说明见下面的 bridge。
6
7 <action application="hash"
8   data="insert/${domain_name}-call_return/${dialed_extension}/${caller_id_number}"/>
9 <action application="hash"
10  data="insert/${domain_name}-last_dial_ext/${dialed_extension}/${uuid}"/>
11 <action application="hash"
12  data="insert/${domain_name}-last_dial_ext/${called_party_callgroup}/${uuid}"/>
13 <action application="hash"
14  data="insert/${domain_name}-last_dial_ext/global/${uuid}"/>
```

hash 是内存中的哈希表数据结构。它可以设置一个键-值对（Key-Value pair）。如，上面最后一行上向 \${domain_name}-last_dial_ext 这个哈希表中插入 global 这么一个键，它的值是 \${uuid}，就是本 Channel 的唯一标志。

不管是上面的 set，还是 hash，都是保存一些数据为后面做准备的。

```
1 <action application="set"
2   data="called_party_callgroup=${user_data(${dialed_extension}@${domain_name} var callgroup})"/>
3 <!--<action application="export"
4   data="nolocal:sip_secure_media=${user_data(${dialed_extension}@${domain_name} var sip_secure_media)}
```

这一行默认是注释掉的，因此不起作用。nolocal 的作用我们已前也讲到过，它告诉 export 只将该变量设置到 b-leg 上，而不要设置到 a-leg 上。

```

1 <action application="hash"
2   data="insert/${domain_name}-last_dial/${called_party_callgroup}/${uuid}"/>

```

还是 hash.

```

1 <action application="bridge"
2   data="${sip_invite_domain=$${domain}}user/${dialed_extension}@${domain_name}"/>

```

bridge 是最关键的部分。其实上面除 bridge 以外的 action 都可以省略，只是会少一些功能。

回忆一下第四章中的内容。用户 1000 其实是一个 SIP UA (UAC)，它向 FreeSWITCH (作为 UAS) 发送一个 INVITE 请求。然后 FreeSWITCH 建立一个 Channel，从 INVITE 请求中找到被叫号码 (destination_number=1001)，然后在 Dialplan 中查找 1001 就一直走到这里。

bridge 的作用就是把 FreeSWITCH 作为一个 SIP UAC，再向 1001 这个 SIP UA (UAS) 发起一个 INVITE 请求，并建立一个 Channel。这就是我们的 b-leg。1001 开始振铃，bridge 把回铃音传回到 1000，因此，1000 就能听到回铃音（如果 1001 有自己的回铃音，则 1001 能听到，否则，将会听到默认的回铃音 \${us-ring}）。

当然，实际的情况比我们所说的要复杂，因为在呼叫之前。FreeSWITCH 首先要查找 1001 这个用户是否已经注册，否则，会直接返回 USER_NOT_REGISTERED，而不会建立 b-leg。

bridge 的参数是一个标准的呼叫字符串 (Dial string)，以前我们也讲到过。domain 和 domain_name 都是预设的变量，默认就是服务器的 IP 地址。user 是一个特殊的 endpoint，它指本地用户。所以，呼叫字符串翻译出来就是（假设 IP 是 192.168.7.2）：

```
1 {sip_invite_domain=192.168.7.2}user/1001@192.168.7.2
```

其中，“{ }”里是设置变量，由于 bridge 在这里要建立 b-leg，因此，这些变量只会建立在 b-leg 上。与 set 是不一样的。但它等价于下面的 export：

```

1 <action application="export" value="nolocal:sip_invite_domain=192.168.7.2"/>
2 <action application="bridge" value="user/1001@192.168.7.2"/>

```

好了，到此为止电话路由基本上就完成了，我们已经建立了 1000 到 1001 之间的呼叫，就等 1001 接电话了。接下来会有几种情况：

- 被叫应答
- 被叫忙
- 被叫无应答
- 被叫拒绝
- 其它情况 ...

我们先来看一下被叫应答的情况。1001 接电话，与 1000 畅聊。在这个时候 bridge 是阻塞的，也就是说，bridge 这个 App 会一直等待两者挂机（或者其它错误）后才返回，才有可能继续执行下面的 Action。好吧，让我们休息一下，等他们两个聊完吧。

最后，无论哪一方挂机，bridge 就算结束了。如果 1000 先挂机，则 FreeSWITCH 会将挂机原因发送给 1001，一般是 NORMAL_RELEASE（正常释放）。同时 Dialplan 就再也没有往下执行的必要的，因此会发送计费信息，并销毁 a-leg。

如果 1001 先挂机，b-leg 就这样消失了。但 a-leg 依然存在，所以还有戏看。

b-leg 会将挂机原因传到 a-leg。在 a-leg 决定是否继续往下执行之前，会检查一些变量。其中，我们在前面设置了 hangup_after_bridge=true。它的意思是，如果 bridge 正常完成后，就挂机。因此，a-leg 到这里就释放了，它的挂机原因是参考 b-leg 得出的。

但由于种种原因 1001 可能没接电话。1001 可能会拒接（CAIL_REJECTED，但多数 SIP UA 都会在用户拒接时返回 USER_BUSY）、忙（USER_BUSY）、无应答（NO_ANSWER 或 NO_USER_RESPONSE）等。出现这些情况时，FreeSWITCH 认为这是不成功的 bridge，因此 hangup_after_bridge 变量就不管用了。这时候它会检查另一个变量 continue_on_fail。由于我们上面设置的 continue_on_fail=true，因此在 bridge 失败（fail）后会继续执行下面的 Action。

这里值得说明的是，通过给 continue_on_fail 不同的值，可以决定在什么情况下继续。如：

```
1 <action application="set" data="continue_on_fail=USER_BUSY,NO_ANSWER"/>
```

将只在用户忙和无应答的情况下继续。其它的值有：NORMAL_TEMPORARY_FAILURE（临时故障）、TIMEOUT（超时，一般时SIP超时）、NO_ROUTE_DESTINATION（呼叫不可达）等。

```
1 <action application="answer"/>
```

最后，无论什么原因导致 bridge 失败（我们没法联系上 1001），我们都决定继续执行。首先 FreeSWITCH 给 1000 回送应答消息。这时非常重要的。

```
1 <action application="sleep" data="1000"/>
2 <action application="voicemail" data="default ${domain_name} ${dialed_extension}" />
```

接下来，暂停一秒，并转到 1001 的语音信箱。语音信箱的知识等我们以后再讲。另外，默认配置中使用的是 loopback endpoint 转到 voicemail，为了方便说明，我直接改成了 voicemail。

回声

没什么好解释的，如果拨 9196，就能听到自己的回声

```

1 <extension name="echo">
2   <condition field="destination_number" expression="^9196$">
3     <action application="answer"/>
4     <action application="echo"/>
5   </condition>
6 </extension>

```

延迟回声

与 echo 基本一样，但回声会有一定延迟，5000 是毫秒数。

```

1 <extension name="delay_echo">
2   <condition field="destination_number" expression="^9195$">
3     <action application="answer"/>
4     <action application="delay_echo" data="5000"/>
5   </condition>
6 </extension>

```

8.2 内连拨号计划 - Inline Dialplan

首先，Inline dialplan 与上面我们讲的 Action 中的 inline 参数是不同的。

XML Dialplan 支持非常丰富的功能，但在测试或编写程序时，我们经常用到一些临时的，或者是很简单的 Dialplan，如果每次都需要修改 XML，不仅麻烦，而且执行效率也会有所折扣。所以，我们需要一种短小、轻便的 Dialplan 以便更高效地完成任务。而且，通过使用 Inline dialplan，也可以很方便的在脚本中生成动态的 Dialplan 而无需使用复杂的 reloadxml 以及 mod_xml_curl 技术等。

与 XML Dialplan 不同，它没有 Extension，也没有复杂的 Condition。而只是象 XML Dialplan 中那样简单的 Action 的叠加。它有一种很紧凑的语法格式：

```
1 app1:arg1,app2:arg2,app3:arg3
```

从语法可以看出，它只是多个 App 以及参数组成的字符串，App 之间用逗号分隔，而 App 与参数之间用冒号分隔。如果参数中有空格，则整个字符串都需要使用单引号引起。在我们上面的例子中，你通过拨打 9196 来找到对应的 XML dialplan，在这里，我们可以直接在命令行上写出对应的 inline 形式：

```

1 originate user/1000 echo inline
2 originate user/1000 answer,echo inline

```

读到这里，你可能要问，它与“originate user/1000 &echo”有什么区别呢？在回答这个问题之前，我们需要先看一下 originate 的语法：

```

1 originate <call_url> <exten>|&<application_name>(<app_args>)
2 [<dialplan>] [<context>] [<cid_name>] [<cid_num>] [<timeout_sec>]

```

首先，它的第一个参数是呼叫字符串，第二个参数可以是 & 加上一个 App，App 的参数要放到 () 里，如：

```

1 originate user/1000 &echo
2 originate user/1000 &playback(/tmp/sound.wav)
3 originate user/1000 &record(/tmp/recording.wav)

```

这是最简单的形式，首先，originate 会产生一个 Channel，它会呼叫 user/1000 这个用户。请注意，这是一个单腿的通话，因此只有一个 Channel。但一个 Channel 有两端，一端是 1000 这个用户，另一端是 FreeSWITCH。在 user/1000 接电话后（严格说是收到它的 earlymedia 后），FreeSWITCH 即开始在该 Channel 上执行 & 后面的 App。但这种形式只能执行一个 App，如果要执行多个，就需要将电话转入 Dialplan：

```

1 originate user/1000 9196
2 originate user/1000 9196 XML default

```

上面两个命令是一样的。它的作用是，在 user/1000 接电话后，电话的另一端（也就是 FreeSWITCH）需要对电话进行路由，在这里，它要将电话路由到 9196 这个 Extension 上，第一条命令由于没有指定是哪个 Dialplan，因此它会在默认的 XML Dialplan 中查找，同时，XML Dialplan 需要一个 Context，它默认就是 default。它的效果是跟你直接用软电话拨打 9196 这个分机一样的（所不同的是呼叫的方向问题，这种情况相当于回拨）。

当然，除此之外，它还可以加一些可选的参数，用于指定来电显示（Caller ID）的名字 (cid_name) 和号码 (cid_number)，以及超时的秒数，如：

```

1 originate user/1000 9196 XML default 'Seven Du' 9196 30

```

当然，我们这里学了 Inline Dialplan，所以你也可以这样用：

```

1 originate user/1000 echo inline

```

请注意，在 XML Dialplan 中，9196 是一个分机号，而 Inline 中的 echo 则是一个 App，当然你也可以顺序执行多个 App：

```

1 originate user/1000 answer,playback:/tmp/please_leave_a_message.wav,record:/tmp/recording.wav inline
2 originate user/1000 playback:/tmp/beep.wav,bridge:user/1001 inline

```

有时候，App 的参数中可能会有逗号，因而会与默认的 App 间的逗号分隔符相冲突，以下的 m 语法形式将默认的逗号改为 ^ 分隔（以下三行实际上为一行）。

```
1 originate user/1000
2 'm:^;playback/tmp/beep.wav^bridge:
3 {ignore_early_media=true,originate_caller_id_number=1000}user/1001'
```

当然你也可以用在任何需要 Dialplan 的地方，如（以下两行实为一行）

```
1 uuid_transfer 2bde6598-0f1a-48fe-80bc-a457a31b0055
2 'set:test_var=test_value,info,palyback:/tmp/beep.wav,record:/tmp/recording.wav'
```

除此之外，还有其它的 Dialplan 形式，我们在这里就不再介绍了，要查看你的系统支持多少 Dialplan，使用如下命令：

```
1 freeswitch@seven-macpro.local> show dialplan
2
3 type,name,ikey
4 dialplan,LUA,mod_lua
5 dialplan,XML,mod_dialplan_xml
6 dialplan,asterisk,mod_dialplan_asterisk
7 dialplan,enum,mod_enum
8 dialplan,inline,mod_dptools
```


第九章 嵌入式脚本

前面提到，FreeSWITCH 支持使用你喜欢的各种程序语言来控制呼叫流程。你不仅可以用它们写出灵活多样的IVR，给用户带来更好的体验，更重要的是你可以通过它们很好地与你的业务进行无缝集成，以节省你的后台业务处理及管理成本。

使用程序语言来做这些事情有两种方式：第一种是嵌入式脚本，第二种是独立的程序。如果使用后者，理论上讲，你可以使用任何你喜欢的语言，只要该语言支持 TCP Socket。关于使用 Socket 方式来控制 FreeSWITCH 的编程方法我们将在下一章讲解。本章主要关注嵌入式脚本。

9.1 什么是嵌入式脚本？

其实前面我们学到的 XML dialplan 已经体现了其非凡的配置能力，它配合 FreeSWITCH 提供的各种 App 也可以认为是一种脚本。当然，毕竟 XML 是一种描述语言，功能还有限。FreeSWITCH 通过嵌入其它语言的解析器支持很多流行的编程语言。

一般来说，编程语言分为两种：编译型语言（如C）和解释型语言（如 javascript, perl 等）。使用解释型语言编写出来的脚本不需要编译，因而非常灵活方便。典型地，FreeSWITCH 支持的语言有：

- Lua
- Javascript
- Python
- Perl
- Java

其它脚本语言如 Php, Ruby 以前是支持的，由于它们有内存及性能问题，且没有志愿者维护，现在已经被列为 [Unsupported](#) 了。

9.2 应用场景

一般来说，这些嵌入式脚本主要用于写 IVR，即主要用来控制一路通话的呼叫流程。虽然它们也可以控制多路通话（在后面我们也会讲到这样的例子，但这不是他们擅长的功能）。

当然，这里说的一路通话不是说它们只能控制唯一一路通话。以 Lua 为例，你可以把呼叫路由到一个 lua 脚本，当有电话进来时，FreeSWITCH 会为每一路通话启动一个线程，控制每一路通话的 lua 脚本则在相应的线程内执行，互不干扰。Java 语言需要 Java 的虚拟机环境，比这个要复杂些。

9.3 Lua

这是一门小众语言，听起来，它可能不像其它语言（如 Java）那样“如雷贯耳”，但由于其优雅的语法及小巧的身段受到很多开发者的青睐，尤其是在游戏领域¹。

在 FreeSWITCH 中，Lua 模块是默认加载的。在所有嵌入式脚本语言中，它是最值得推荐的语言。首先它非常轻量级，mod_lua.so 经过减肥(strip)后只有272K；另外，它的语法也是相当的简单。有人做过对比说，在嵌入式的脚本语言里，如果 Python 得 2 分，Perl 拿 4，Javascript 得 5，则 Lua 语言可得 10 分。可见一斑。

另外，Lua 模块的文档也是最全的。我在使用其它模块或写 Event Socket 程序时也经常参考 Lua 模块的文档。

9.3.1 语法简介

- Lua 语言的注释为以两个“-”开头，支持单行和多行注释，如：

```
-- 这是一行注释  
--[[ 这是多行注释  
    第二行也被注释掉了  
]]
```

- Lua 变量不需要类型声明
- Lua 支持类似面向对象的编程，所有对象都是一个 Table(Lua 中独有的概念)。
- Lua 支持尾递归、闭包。

详细的资料请参阅有关资料，底线是 — 如果你会其它编程语言，在30分钟内就能学会它。

¹我相信有很多人知道它是缘于2010年一则新闻中说一个14岁的少年用它编出了 iPhone 上的名为 Bubble Ball 的游戏，该游戏下载量曾一度超过史上最流行的“愤怒的小鸟”。

9.3.2 将电话路由到 Lua 脚本

在 dialplan XML 中，使用

```
1 <action application="lua" data="test.lua"/>
```

便可将进入 dialplan 的电话交给 lua 脚本接管。脚本的默认路径是安装路径的 scripts/ 目录下，当然你也可以指定绝对路径，如 /tmp/test.lua。需要注意在 windows 下目录分隔符是用 “\”，所以有时候需要两个“\”，如“c:\\\\test\\\\test.lua”。

9.3.3 Session 相关函数

FreeSWITCH 会自动生成一个 session 对象（实际上是一个 table），因而可以使用 Lua 面象对象的特性编程，如以下脚本放播放欢迎声音(来自 [Hello Lua](#))。

```
1 -- answer the call
2 session:answer();
3
4 -- sleep a second
5 session:sleep(1000);
6
7 -- play a file
8 session:streamFile("/tmp/hello-lua.wav");
9
10 -- hangup
11 session:hangup();
```

大部分跟 session 有关的函数是跟 FreeSWITCH 中的 App 是一一对应的，如上面的 answer()、hangup() 等，特别的， streamFile() 对应 playback() App。如果没有对应的函数，也可以通过 session:execute() 来执行相关的 App，如 session:execute("playback", "/tmp/sound.wav") 等价于 session:streamFile("/tmp/sound.wav")。

需要注意，lua 脚本执行完毕后默认会挂断电话，所以上面的 Hello Lua 例子中不需要明确的 session:hangup()。如果想在 lua 脚本执行完毕后继续执行 dialplan 中的后续流程，则需要在脚本开始处执行

```
1 session:setAutoHangup(false)
```

如下列场景，test.lua 执行完毕后（假设没有 session:hangup()，主叫也没有挂机），如果没有 setAutoHangup(false)，则后续的 playback 动作得不到执行。

```
1   <extension name="eavesdrop">
2       <condition field="destination_number" expression="^1234$">
3           <action application="answer"/>
4           <action application="lua" data="test.lua"/>
5           <action application="playback" data="lua-script-complete.wav"/>
6       </condition>
7   </extension>
```

更多的函数可以参考相关的 wiki 文档: http://wiki.freeswitch.org/wiki/Mod_lua

9.3.4 非 Session 函数

Lua 脚本中也可以使用跟 Session 不相关的函数，最典型的是 freeswitch.consoleLog()，用于输出日志，如：

```
1   freeswitch.consoleLog("NOTICE", "Hello lua log!\n")
```

另外一个是 freeswitch.API，它允许你执行任意 API，如

```
1   api = freeswitch.API();
2   reply = api:executeString("sofia", "status");
```

9.3.5 独立的 Lua 脚本

独立的 Lua 脚本可以直接在控制台终端上(使用 luarun)执行，这种脚本大部分可用于执行一些非 Session 相关的功能，后面我们会讲到相关例子。

9.3.6 数据库

在 Lua 中，可以使用 [LuaSQL](#) 连接各种关系型数据库，但据说 LuaSQL 与某些版本的数据库驱动结合有内存泄漏问题，配置起来也比较复杂。

另一种连接数据库的方式是直接使用 freeswitch.Dbh。它可以直接通过 FreeSWITCH 内部的数据库连接句柄来连接 sqlite 数据库或任何支持 ODBC 的数据库。下面是一个来自 FreeSWITCH wiki 的例子。

```
1 local dbh = freeswitch.Dbh("dsn","user","pass") -- when using ODBC
2 -- OR --
3 -- local dbh = freeswitch.Dbh("core:my_db") -- when using sqlite
4
5 assert(dbh:connected()) -- exits the script if we didn't connect properly
6
7 dbh:test_reactive("SELECT * FROM my_table",
8                     "DROP TABLE my_table",
9                     "CREATE TABLE my_table (id INTEGER(8), name VARCHAR(255))")
10
11 dbh:query("INSERT INTO my_table VALUES(1, 'foo')") -- populate the table
12 dbh:query("INSERT INTO my_table VALUES(2, 'bar')") -- with some test data
13
14 dbh:query("SELECT id, name FROM my_table", function(row)
15     stream:write(string.format("%5s : %s\n", row.id, row.name))
16 end)
17
18 dbh:query("UPDATE my_table SET name = 'changed'")
19 stream:write("Affected rows: " .. dbh:affected_rows() .. "\n")
20
21 dbh:release() -- optional
```

9.4 Javascript

相对于 Lua, 大家可能对 Javascript 更熟悉一些。Javascript 是 Web 浏览器上最主流的编程语言, 它最早是设计出来用于配合 HTML 渲染页面用的, 近几年由于 [Node.js](#) 的发展使它在服务器端的应用也已发扬光大。它遵循 [EMCAScript](#) 标准。

通过加载 mod_spidermonkey 模块可以使用 js 解析器, 模块 mod_spidermonkey_odbc 则支持在 Javascript 脚本中连接 ODBC 数据库。

除语法不同外, 用法上与 Lua 类似, 如使用 javascript (它是一个App) 执行一个 session 相关的脚本, 或 jsrun (它是一个API) 执行一个非 session 相关的脚本。

9.5 其它脚本语言

其它脚本语言的使用也类似, 读者可参照使用。值得一提的是, FreeSWITCH 有一个 mod_managed 模块支持 Windows .NET 架构下的语言 (F#, VB.NET, C#, IronRuby, Iron-Python, JScript.NET), 通过 mono 也可以支持其它平台 (如 Linux) 。

第十章 Event Socket

相信好多读者都已经等待本章好久了。Event Socket 是操控 FreeSWITCH 的瑞士军刀。它可以通过 Socket 使用FreeSWITCH提供的所有 App 和 API函数。由于使用 Socket, 它几乎可以跟任何语言开发的程序通信，也就是说，它几乎可以跟任何系统进行集成。

FreeSWITCH 使用 [swig](#) 来支持多语言。简单来讲，FreeSWITCH用C语言写了一些库函数，通过swig包装成其它语言接口。现在已知支持的语言有 C、Perl、Php、Python、Ruby、Lua、Java、Tcl、以及由 managed 支持的 .Net 平台语言如 C#, VB.NET 等。

值得一提的是，Event Socket 并没有提供什么新的功能，它只是提供了一个开发接口，所有的通道处理及媒体控制还都是由 FreeSWITCH 提供的 App 和 API 来完成的。当然，读到这里没有必要失望，我这么说只是希望读者能更专注于这个接口的概念，以便更好地理解这里的逻辑。

10.1 架构

[Event Socket](#) 有两种模式，内连（inbound）模式和外连（outbound）模式。注意，这里所说的内外是针对 FreeSWITCH而言的。

10.1.1 外连模式

见下图，FreeSWITCH 作为一个TCP客户端连接到一个 TCP 服务器上。



那么这个 TCP 服务器是谁呢？它是需要用户自己去实现的，实际上这就是用户需要开发的部分。在这个服务器里，用户可以实现自己的业务逻辑，以及连接数据库获取数据帮助决策等。

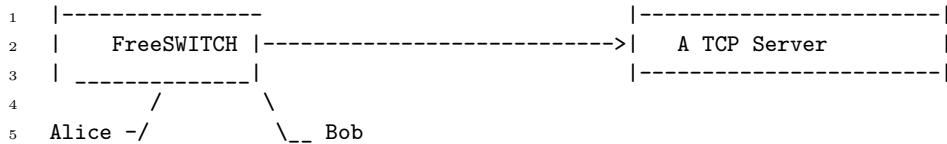
那么，怎么让 FreeSWITCH 去连接你的 TCP Server 呢？回想一下第四章。FreeSWITCH 是一个 B2BUA，当 Alice 呼叫 Bob 时，首先电话会到达 FreeSWITCH(SIP)，建立一个单腿的 Channel (a-leg)，然后电话进入路由状态，FreeSWITCH 查找 dialplan，找到通过以下动作建立一个到 TCP Server 的连接：

```
1 <action application="socket" data="127.0.0.1:8084"/>
```

到这里，还是只有一个 Channel。socket 是一个 App，它会把这个 Channel 置为 Park 状态，然后把 FreeSWITCH 作为一个 TCP 客户端连接到 TCP Server 上，把当前呼叫的相关信息告诉它，并询问下一步该怎么做。当然，这里 FreeSWITCH 跟 TCP Server 说的一种言，只有他们两个人懂，与 SIP 及 RTP 没有任何关系。也就是说，TCP Server 只是发布控制指令，并不实际处理语音数据。

接下来，TCP Server 收到 FreeSWITCH 的连接请求后，进行决策，如果它认为 Alice 想要呼叫 Bob，它就给 FreeSWITCH 发一个 bridge App 消息，告诉它应该继续呼叫 Bob(给 Bob 发 INVITE SIP 消息)。

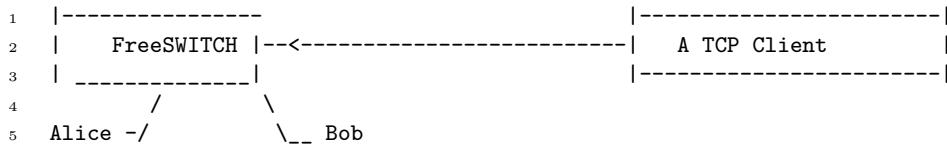
如下图。在 Alice 挂机之前，FreeSWITCH 会一直向 TCP Server 汇报 Channel 的相关信息，所以这个 TCP Server 就掌握这通电话所有的详细信息，也可以在任何时间对他们发号施令。



Alice 挂机后，与 TCP Server 的连接也会断开，释放资源。读到这里，读者应该明白了。之所以叫做 TCP Server，是因为它应该是一个服务器应用，永远在监听一个端口（在上面的例子中是 8084）等待有人连接。如果需要支持多个连接，这个服务器就应该使用 socket 的 select 机制或做成多线程/多进程的。

我们上面说过，所谓 inbound 和 outbound 都是针对 FreeSWITCH 而言的。由于在这种模式下，FreeSWITCH 要连接到 TCP Server，因此称为外连模式。

10.1.2 内连模式



在这种模式下，FreeSWITCH 作为一个服务器，而用户的程序可以作为一个客户端主动连接到 FreeSWITCH 上。同样，FreeSWITCH 允许多个客户端连接。每个客户端连接上来以后，可

以订阅一些事件。上面我们说过，FreeSWITCH要通过Event Socket向外部发送信息，这些信息就是以事件的形式体现的。同样，在内部，好多功能也是事件驱动的。

用户的 TCP Client 收到这些事件后，可以通过执行 App 和 API 来控制 FreeSWITCH 的行为。

10.1.3 模式小结

总起来讲，对于外连模式来讲，由于“socket”来身是一个 App，进而它所连接的 TCP Server 也好像是这个 App 功能的一部分，在 Alice 这个 Channel 的内部工作，它发布的命令通常也是一些App。只是在 bridge 到 Bob 后它又好像是一个中间人，或第三者。

而对于内连模式，很明显外部的 TCP Client 就是一个第三者，它通常不是 Channel 的一部分，而是监听到一个感兴趣的事件以后，通过API (uuid_ 一族的API) 来对 Channel 进行操作

10.2 Event Socket 协议

无论是哪种模式，FreeSWITCH要跟外部的程序交流思想，需要说一种语言。上面我们为了区分 SIP、RTP等协议，说这是一种方言。就姑且称这种方言为 ESL (Event Socket Language) 吧？实际上 ESL 是 Event Socket Library 的编写，这儿撞衫纯属巧合。

ESL 协议是纯文本的协议。它的思想来自于大家熟悉的HTTP协议及SIP协议。如果不熟悉，可以翻回第四章看看。这里再稍微罗嗦一点点——该协议的特点就是大部分消息（一些命令除外）都有一些字段组成的消息头（Header），某些消息有消息体（Body），某些则没有。消息头中一个字段叫 Content-Length，标志了消息体的长度。消息头和消息体之间用两个回车换行|r|n|r|n 隔开。

用户程序（TCP Server/Client）可以使用 sendmsg 发送 App 或 发送其它命令。

下面举两个例子：

10.2.1 外连

我的 dialplan 如下：

```
1 <extension name="socket">
2   <condition field="destination_number" expression="^1234">
3     <action application="socket" data="localhost:8084 async full"/>
4   </condition>
5 </extension>
```

当电话呼叫 1234 时，便连接 socket。注意这里的两个参数 async 和 full。其中，async 表示异步执行。缺省为同步的，比方说，在同步状态下，如果 FreeSWITCH 正在执行 playback 操

作，而 playback 是阻塞的，因而在 playback 完成之前向它发送任何消息都是不起作用的，但异步状态可以进行更灵活的控制。当然，异步状态增加了灵活性的同时也增加了开发的复杂度，这个读者在实际的开发中可以对比一下他们的区别。另一个参数 full 指明可以在外部程序中使用全部的 API，默认只有少量的 API 是有效的。至于哪些 API 跟这个参数相关，也留给读者练习。一句如果你不确定，加上 full 是没有错的。

好了，电话来了，由于我们的 TCP Server 没有启动，因此连接会失败，电话就断掉了。

所以我们需要一个 TCP Server，用任何语言写一个 Server 都需要很多文字去解释，因此，在这里，我们使用 netcat 这个工具来讲解。

netcat 是一般 linux 系统自带了一个工具，它可以做服务器也可以作客户端。如果做为客户端，你可以想像它类似于你更熟悉 Telnet。虽然它叫 netcat，但程序的名字叫 nc。如果系统默认没有安装，可以使用 *apt-get install netcat* (Ubuntu/Debian) 或 *yum install netcat* (CentOS) 来安装。

打开一个终端A，启动一个 Server A，监听 8084 端口（其中 -l 表示监听 listen，-k 表示客户端断开后继续保持监听。注意，有些版本的 netcat 参数稍有不同，请看相关的 man 文档）

```
nc -l -k localhost 8084
```

打开另一个终端B，启动一个 Client B 连上它

```
nc localhost 8084
```

然后在这个客户端中打些字，回车，在终端A中就应该能看到你输入的文字。

好了，接下来按 Ctrl+C 退出终端B，该让 FreeSWITCH 上场了。一会儿，FreeSWITCH 就相当于终端B。

拿起电话打 1234，电话路由到 socket，FreeSWITCH 就会连到 Server A 上。这时候你听不到任何声音，因为 Channel 处于 Park 状态。但是你在 Server A 里也看不到任何连接的迹象。不过，如果你打开另一个终端，以下命令可以显示 8084 端口处于连接 (ESTABLISHED) 状态。

```
1  seven@seven-macpro:~$ netstat -an|grep 8084
2  tcp4      0      0  127.0.0.1.8084          127.0.0.1.60588      ESTABLISHED
3  tcp4      0      0  127.0.0.1.60588          127.0.0.1.8084      ESTABLISHED
4  tcp4      0      0  127.0.0.1.8084          *.*                  LISTEN
```

好了，看我的。回到终端A，键入 connect 然后打两下回车，奇迹出现了吧？你应该会看到类似下面的输出：

```
1  Event-Name: CHANNEL_DATA
2  Core-UUID: 4bfcc9bd-6844-4b45-96a7-4feb2a4f9795
3  ..... 此外省略XXXX字
```

这便是 FreeSWITCH 发给 Server A 的第一个事件，里面包含了该 Channel 所有的信息。下面让该 Channel 何去何从完全看你的了。给它放点音乐听吧？

```
1 sendmsg
2 call-command: execute
3 execute-app-name: playback
4 execute-app-arg: local_stream://moh
```

我建议你直接粘贴上面的命令到 Server A 的窗口里，记得打两下回车啊。sendmsg 和作用就是发送一个 App 命令（这里是 playback）给 FreeSWITCH，然后 FreeSWITCH 就乖乖地照着做。如果你玩够了，就把上面的 playback 换成 hangup，再发一次，就挂断了。明白了吧？这些命令就跟直接写到 dialplan 里一样，不同的是你的程序可以控制什么时候发什么命令。

10.2.2 内连

FreeSWITCH 会启动一个 Event Socket TCP Server，IP、端口号和密码均可以在 conf/autoload_configs/event_socket.conf.xml 里配置。读到这里推荐你先去看一下那个文件。

还记得上面我们在终端B里用 nc 做客户端吧？很简单：

```
1 nc localhost 8021
```

连接上以后，你会看到如下消息：

```
1 Content-Type: auth/request
```

它告诉你，应该输入密码进行验证。输入

```
1 auth ClueCon
```

记得打两下回车了，如果说你就嫌我罗嗦了。至于为什么是 8021，为什么是 ClueCon 不用我说了吧？我已经告诉你应该去哪里看了。

如果一切顺利的话，你现在已经作为一个客户端连接到 FreeSWITCH 上了。为所欲为吧：

```
1 api version
2
3 api status
4
5 api sofia status
6
7 api uptime
```

你肯定经常在 fs_cli 中使用这些命令。不错，只是前面多了个 api。其实你也应该已经猜到了，fs_cli 也是作为一个客户端使用ESL与FreeSWITH通信的，只是它帮你做了好多事，你不用手工敲一些协议细节了。

收一下事件吧？键入

```
1 event plain all
```

可以订阅所有的事件，当然如果你看不过来可以少订点，如：

```
1 even plain CHANNEL_CREATE
```

是的，这次你又猜对了，在 fs_cli 中你是这么做的

```
1 /event plain CHANNEL_CREATE
```

看一下 fs_cli 的源代码 (fs_cli.c) 吧，因为它包含了所有你想要的东西，包括下面我们要讲的。

10.3 Event Socket 库

当然，事情永远都不像看起来那么简单。如果所有协议都需要手工敲的话，那电话一多你就累了。不过，这些早就有人帮你做好了，这就是你一直期望学习的 ESL (Event Socket Library)。

在上面的实验中，你可以注意到有好多类似这样的输出：

```
1 Content-Type: command/reply
```

这些是针对你输入命令的响应，同时如果你订阅职事件的话你还能收到好多事件，如果有许多通话的话，如果你又不停地控制他们的话，会不会很凌乱啊？会不会产生竞争条件 (Race Condition) 啊？答案是不会，因为ESL都帮你做好了，包装成了函数，而且有各种语言的版本。当然，对于不支持的语言，最得需要你自己写了。比如，笔者在N年前玩Flex时就曾自己写了一个 <http://wiki.freeswitch.org/wiki/FsAir>。

虽然你在实际应用中一般不需要知道上面的细节，但我相信对于理解整个架构还是有好处的。

好了，该言归正传了。

10.3.1 Event Socket 的例子

言归正传了讲什么的，其实没的讲了，因为你已经全懂了，自己再看几个例子就行了：http://wiki.freeswitch.org/wiki/Event_Socket_Library

总之一句话，所有的控制都是通过App和API，具体应用逻辑就看你自己的发挥了。

Ruby 客户端

看一个 Ruby 的例子吧？不管你是否熟悉Ruby，以下语言都是很好懂的：

```

1  require 'ESL'
2
3  con = ESL::ESLconnection.new('127.0.0.1', '8021', 'ClueCon')
4  esl = con.sendRecv('api sofia status')
5  puts esl.getBody

```

上述例子连接到 FreeSWITCH，执行 *sofia status* 命令，并输出结果。

C 客户端示例

在源代码目录 `libs/esl/` 有个 `test_client.c`，运行它后会使用 inbound 模式连接 FreeSWITCH。

```

1  int main(void)
2  {
3      // 初始化 handle
4      esl_handle_t handle = {{0}};
5
6      // 连接服务器，如果成功 handle 就代表这个连接了，参数意义都很明显
7      esl_connect(&handle, "localhost", 8021, NULL, "ClueCon");
8
9      // 发送一个命令，并接收返回值
10     esl_send_recv(&handle, "api status\n\n");
11
12     // last_sr_event 应该是 last server response event，即针对上面命令的响应
13     // 如果在此之前收到事件（在本例中不会，因为没有订阅），事件会存到一个队列里，不会发生竞争条件
14
15     if (handle.last_sr_event && handle.last_sr_event->body) {
16         // 打印返回结果
17         printf("%s\n", handle.last_sr_event->body);
18     } else {
19         // 对于 api 和 bgapi 来说（上面已经将命令写死了），应该不会走到这里；
20         // 但其它命令可能会到这里
21         printf("%s\n", handle.last_sr_reply);
22     }
23
24     // 断开连接
25     esl_disconnect(&handle);
26     return 0;
27 }

```

C 服务器示例

同样的目录中有一个 `testserver.c`, 运行于 *outbound* 模式, 它是多线程的, 每次有电话进来时, 通过 *dialplan* 路由到 *socket*, FreeSWITCH 便向它发起一个连接。

我们先从 `main()` 函数开始看。第3行把 `log` 级别开到最大, 这样能看到详细的 `log`, 包括所有协议的细节。然后它通过 `esl_listen_threaded` 启动一个 `socket` 监听 8084 端口。如果有连接到来时, 它便回调 `mycallback` 函数。

```
1 int main(void)
2 {
3     esl_global_set_default_logger(7);
4     esl_listen_threaded("localhost", 8084, mycallback, 100000);
5
6     return 0;
7 }
```

回调函数如下:

```

1 static void mycallback(esl_socket_t server_sock, esl_socket_t client_sock, struct sockaddr_in *addr)
2 {
3     esl_handle_t handle = {{0}}; // 初始化一个句柄, 用于标志一个 ESL 连接
4     int done = 0;
5     esl_status_t status;
6     time_t exp = 0;
7
8     esl_attach_handle(&handle, client_sock, addr); // 将handle与socket绑定
9     esl_log(ESL_LOG_INFO, "Connected! %d\n", handle.sock);
10
11    // 我们只关心与本 channel 相关的事件
12    esl_filter(&handle, "unique-id", esl_event_get_header(handle.info_event, "caller-unique-id"));
13
14    // 订阅这些事件
15    esl_events(&handle, ESL_EVENT_TYPE_PLAIN, "SESSION_HEARTBEAT CHANNEL_ANSWER"
16                " CHANNEL_ORIGINATE CHANNEL_PROGRESS CHANNEL_HANGUP "
17                " CHANNEL_BRIDGE CHANNEL_UNBRIDGE CHANNEL_OUTGOING CHANNEL_EXECUTE"
18                " CHANNEL_EXECUTE_COMPLETE DTMF CUSTOM conference::maintenance");
19
20    esl_send_recv(&handle, "linger"); // 发送一个命令 linger, 开启逗留模式, 相关说明见下文
21    esl_execute(&handle, "answer", NULL, NULL); // 执行 answer App
22    esl_execute(&handle, "conference", "3000@default", NULL); // 将来话送到 3000 这个会议中
23
24    // 循环, 等待挂机
25    while((status = esl_recv_timed(&handle, 1000)) != ESL_FAIL) {
26        if (done) { // 如果有结束标志
27            if (time(NULL) >= exp) { break; } // 忙等待, 超时后跳出循环
28        } else if (status == ESL_SUCCESS) {
29            const char *type = esl_event_get_header(handle.last_event, "content-type");
30            // 挂机后会收到该事件
31            if (type && !strcasecmp(type, "text/disconnect-notice")) {
32                const char *dispo = esl_event_get_header(handle.last_event, "content-disposition");
33                esl_log(ESL_LOG_INFO, "Got a disconnection notice disposition: [%s]\n", dispo ? dispo : "");
34                if (!strcmp(dispo, "linger")) {
35                    done = 1;
36                    // 挂机后, 可能还有后续的事件, 我们多等几秒钟
37                    esl_log(ESL_LOG_INFO, "Waiting 5 seconds for any remaining events.\n");
38                    exp = time(NULL) + 5;
39                }
40            }
41        }
42    }
43    // 清理现场
44    esl_log(ESL_LOG_INFO, "Disconnected! %d\n", handle.sock);
45    esl_disconnect(&handle);
46 }

```

linger 为“逗留”模式。如果不开启该模式，则主叫挂机后，FreeSWITCH 会立即断开它主动建立的 socket，会造成一些后续的事件收不到，如 CHANNEL_HANGUP_COMPLETE, CHANNEL_DESTROY 等。逗留模式告诉 FreeSWITCH 晚一些断开这个 socket。

使用 ESL 发送 SIP MESSAGE 消息

Wiki 上有一个使用 Perl 发送 MESSAGE 的应用，这里用C写了一下（在 testclient 基础上修改的）：

http://wiki.freeswitch.org/wiki/Mod_sms#Sending_a_Message_via_ESL

执行以下程序后，在 1000 这个 SIP 电话上将会收到一个 *hello* 消息（如果你IP地址正确的話）。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <esl.h>
4
5 int main(void)
6 {
7     esl_handle_t handle = {{0}};
8     struct esl_event *event;
9     struct esl_event_header header;
10
11    // 初始化一个 event
12    esl_event_create_subclass(&event, ESL_EVENT_CUSTOM, "SMS::SEND_MESSAGE");
13    // 增加头
14    esl_event_add_header_string(event, ESL_STACK_BOTTOM, "to", "1000@192.168.0.7");
15    esl_event_add_header_string(event, ESL_STACK_BOTTOM, "from", "seven@192.168.0.7");
16    esl_event_add_header_string(event, ESL_STACK_BOTTOM, "sip_profile", "internal");
17    esl_event_add_header_string(event, ESL_STACK_BOTTOM, "dest_proto", "sip");
18    esl_event_add_header_string(event, ESL_STACK_BOTTOM, "type", "text/plain");
19    // 增加 body
20    esl_event_add_body(event, "hello");
21    // 连接,
22    esl_connect(&handle, "localhost", 8021, NULL, "ClueCon");
23    // 先发个命令试试?
24    esl_send_recv(&handle, "api version\n\n");
25
26    if (handle.last_sr_event && handle.last_sr_event->body) {
27        printf("%s\n", handle.last_sr_event->body);
28        // 到这里上面的命令应该是成功了
29        printf("sending event....\n");
30        // 把 event 发出去
31        esl_sendevent(&handle, event);
32        // 释放内存
33        esl_event_destroy(&event);
34    } else {
35        // 应该不会到这里
36        printf("%s\n", handle.last_sr_reply);
37    }
38
39    esl_disconnect(&handle);
40    return 0;
41 }

```

接下来做什么？多实验，函数的参考看 wiki。有空多看看 fs_cli.c，里面有所有的细节。

第十一章 FreeSWITCH实战

在前面的章节里，我们介绍了一些基础知识。所谓理论联系实际，掌握任何一门技能都少不了不断的练习。本书的名称是《FreeSWITCH实战》，就让我们在本章言归正传。

实战是本书写作的初衷。同时也与FreeSWITCH官方的第二本书*FreeSWITCH Cookbook*不谋而合。本章收集了作者在多年的工作过程中的一些实例，以及一些博客文章。有一些是作者在学习的过程中留下来的，有一些是在实际的环境中应用的。无论如何，笔者希望通过这些实际的例子，抛砖引玉，与大家共同学习和提高。

11.1 用FreeSWITCH实现IVR

IVR的全称是Interactive Voice Response，就是我们经常说的电话语音菜单。FreeSWITCH支持非常强大的语音菜单——你可以写简单的XML，或更使用灵活的Lua等脚本语言，以及 Event Socket，Erlang Socket等等。

这里，简单介绍一下XML。其实语音菜单说简单也简单，说难也难。让我们先来一个感性的认识—FreeSWITCH默认的配置已包含了一个功能齐全的例子，随便拿起一个分机，拨5000，就可以听到菜单提示了，当然，默认的提示是英文的，大意是说欢迎来到FreeSWITCH，拨1进入FreeSWITCH会议；拨2进入回音（echo）程序（这时候可以听到自己的回音）；拨3听等待音乐（MOH，Music on Hold），拨4会转到FreeSWITCH开发者Brian West的SIP电话上；拨5你会听到一只尖叫的猴子；拨6进入下级菜单；拨9重听，拨1000-1019之间的号码则会转到对应分机。

11.1.1 最简单的菜单

感受这些之后，让我们先来配置一种最简单的情形。一些廉价的企业小交换机通常会提供这样的功能——“您好，欢迎致电XX公司，请直拨分机号，查号请拨0”。在此，我们假定使用FreeSWITCH的默认配置，分机号为1000-1019，前台分机号为0，拨0则转人工台，查号或转接其它分机。

系统默认的配置文件存是`conf/autoload_configs/ivr.conf`，XML格式，它装入`conf/ivr_menus/`目录下的所有`.xml`文件。系统有一个示例的IVR配置，叫`demo_ivr`，也就是你刚才拨5000听到的那个（如果你真拨了的话）。

真正的菜单放到一对`<menus>` `</menus>`中，而每一个`<menu>` `</menu>`即是一个菜单。它应该有一个唯一的名字（name），以便在拨号计划（dialplan）中引用。在前面已经讲过，被装入的XML文件最外层一般都有一对`include`标签，以保证XML文档的完整性。实现我们目标菜单的配置如下：

```

1  <include>
2  <menus>
3      <menu name="welcome"
4          greet-long="custom/welcome.wav"
5          greet-short="custom/welcom_short.wav"
6          invalid-sound="ivr/ivr-that_was_an_invalid_entry.wav"
7          exit-sound="voicemail/vm-goodbye.wav"
8          timeout="15000"
9          max-failures="3"
10         max-timeouts="3"
11         inter-digit-timeout="2000"
12         digit-len="4">
13
14         <entry action="menu-exec-app" digits="0" param="transfer 1000 XML default"/>
15         <entry action="menu-exec-app" digits="/^(10[01][0-9])$/" param="transfer $1 XML default"/>
16
17     </menu>
18 </menus>
19 </include>

```

我们指定菜单的名字是`welcome`, `greet-long`即为最开始播放的语音—“您好，欢迎致电XX公司，请直拨分机号，查号请拨0”，该语音文件默认的位置应该是`/usr/local/freeswitch/sounds/`。你应该事先把声音文件录好，放到`custom/welcome.wav`（当然，你也可以使用其它路径，如`/home/your_name/ivr/welcome.wav`）。并且，由于PSTN交换机都是使用PCM编码，所以，`welcome.wav`文件的格式应设为单声道，8000Hz能获得较好的音质并占用较少的系统资源。

如果用户长时间没有按键，刚应重新提示拨号，但重新提示应该简短，如直接说“请直拨分机号，查号请拨0”。所以，可以录制这么一个声音文件放到`custom/welcome_short.wav`。

`invalid-sound`: 如果用户按错了键，则会使用该提示。如果你安装时指定了`make sounds-install`，则该文件应该是默认存在的，只是它是英文的，如果你需要中文的提示，可以自己录一个放到`custom`目录中。

`exit-sound`: 不说也知道，最后菜单退出时（一般是超时），会提示“Good Bye”。

`timeout`指定超时时间；`max-failures`为容忍用户按键错误的次数。`max-timeouts`即最大超时次数。`inter-digit-timeout`为两次按键的最大间隔（毫秒），如用户拨分机号1001时，如果拨了10，等2秒，然后再按01，这时系统实际收到的号码为10，则会播放`invalid-sound`以提示错误。

`digit-len`说明菜单项的长度，在本例中，用户分机号长度为4位。

该`menu`中有两个选项，第一个是在用户按0时，通过`menu-exec-app`执行一个命令（参见[mod_command](#)），在此处它执行`transfer`，将来话转到分机1000。

如果来电用户知道分机号，则可以直接拨分机号，而不用经过前台转接，节约时间。在该例中，正则表达式“`/^(10[01][0-9])$/`”会匹配用户输入1000-1019之间的分机，

以上菜单设定好后，需要在控制台中执行 `reloadxml`（或按F6键）才可以配置生效。

配置完成后就可以在控制台上进行测试（呼叫1001，接听后进入ivr菜单）：

```
1 FS> originate user/1001 &ivr(welcome)
```

测试成功后，你就可以配置 *dialplan* 把并户来话转接到菜单了，在你的 *dialplan* 中加入一个 *extension*（请注意，你加到正确的 *dialplan* 中，不确定的话，在 *default* 和 *public* 中都加上应该没错，实际上，如果这里出了问题，你应该回去看第八章）：

```
1 <code>
2   <extension name="incoming_call">
3     <condition field="destination_number" expression="^1234$">
4       <action application="answer" data="" />
5       <action application="sleep" data="1000" />
6       <action application="ivr" data="welcome" />
7     </condition>
8   </extension>
9 </code>
```

呼叫1234，是不是可以听到语音菜单了？注意，以实际应用中，你可能要把1234改成你实际的DID。

11.1.2 默认菜单简介

明白了以上简单的菜单，就很容易理解更复杂一点的配置了。上面我们提到，系统默认提供了一个名叫 *demo_ivr* 的菜单。最初的语音提示 *greet-long/greet-short* 是用 *phrase* 实现的。*phrase* 是用 XML 定义的一些短语，最终也是播放声音文件，但在多语言系统中会更灵活。在此，我们不讨论 *phrase*，你可以简单的认为它就是一个声音文件。

菜单选项大多都是根据用户按键使用 *menu-exec-app* 执行相应的命令。*menu-sub* 表示会执行一个下级菜单，这样，在下级菜单中（此处是 *demo_ivr_submenu*）便可以用 *menu-top* 来返回上级菜单。

基本上就这么多。通过设置多级菜单，以及与 *dialplan* 配合，根据不同的情况进行跳转，可以实现相当复杂的一些功能。如果这些还不够，可以尝试一下更高级的 *LUA* 菜单或使用 *Event Socket* 实现。

11.1.3 调试

打开控制台或 *fs_cli*，按 F8 将 *loglevel* 调到 *debug* 状态，能看到详细的执行过程。如果看到红色的（如果你的控制台不支持彩色，看 *ERROR* 的吧），可能是配置错误，不过一般会是声音文件找不到之类的，检查相应路径下是否有对应的声音文件。

11.2 在FreeSWITCH中执行长期运行的嵌入式脚本—Lua语言例子

众所周知，FreeSWITCH中可以使用嵌入式的脚本语言`javascript`、`lua`等来控制呼叫流程。而更复杂一点操作可能就需要使用[Event Socket](#)了。其实不然，嵌入式的脚本也可以一直运行，并可以监听所有的`Event`，就像使用`Event Socket`起一个单独的`Daemon`一样。

这里我们以`lua`为例来讲一下都有哪些限制以及如何解决。

首先，在控制台中执行一个`Lua`脚本有两种方式，`lua`和`luarun`。二者不同就是`lua`是在当前线程中运行的，所以，它会阻塞；而`luarun`会产生一个新的线程，不会阻塞当前的线程执行。

另外，你也可以写到`lua.conf`配置文件中，这样它就能随FreeSWITCH一起启动。

```
<param name="startup-script" value="gateway_report.lua"/>
```

脚本后面可以加参数，如“`luarun test.lua arg1 arg2`”，在脚本中，就可以通过`argv[1]`，`argv[2]`来获得参数的值。而`argv[0]`则是脚本的名字。

如果要让脚本一直运行，脚本中必须有一个无限循环。你可以这样做：

```
while true do
    -- Sleep for 500 milliseconds
    freeswitch.msleep(500);
    freeswitch.consoleLog("info", "blah...");
```

但这样的脚本是无法终止的，由于FreeSWITCH使用`swig`支持这些嵌入式语言，而有些语言没有退出机制，所以，所有语言的退出机制都没有在FreeSWITCH中实现，即使`unload`相关的语言模块也不行，也是因为如此，为了避免产生问题，所有语言模块也都不能`unload`。

另外，使用`freeswitch.msleep()`也不安全，Wiki上说：“*Do not use this on a session-based script or bad things will happen*”。

既然是长期运行的脚本，那，为什么为停止呢？是的，大部分时间你不需要，但，如果你想修改脚本，总不会每次都重启FreeSWITCH吧？尤其是在调试的时候。

那么，还有别的办法吗？

答案是肯定的，我们可以使用事件机制构造另一个循环：

```
con = freeswitch.EventConsumer("all");
for e in (function() return con:pop(1) end) do
    freeswitch.consoleLog("info", "event\n" .. e:serialize("xml"));
end
```

上面的代码中，`con`被初始化成一个事件消费者。它会一直阻塞并等待FreeSWITCH发出一个事件，并打印该事件的XML表示。当然，事件总会有的。如每个电话初始化、挂机等都会有相应的事件。除此之外，FreeSWITCH内部也会20秒发出一个*heartbeat*事件，这样你就可以定时执行一些任务。

当然如果使用 `con:pop(0)`也可以变成无阻塞的，但你必须在循环内部执行一些`sleep()`以防止脚本占用太多的资源。

通过这种方法，你应该就能想到办法让脚本退出了。那就是，另外执行一个脚本触发一个*custom*的事件，当该脚本监测到特定的*custom*事件后退出。当然你也可以不退出，比方说，打印一些信息以用于调试。

我写了一个`gateway_report.lua`脚本。就用了这种技术。思路是：监听所有事件。如果收到`hangup`，则判断是通过哪个`gateway`出去的，并计算一些统计信息。如果需要保存这些信息，可以有以下几种方式：

- `fire_event`，即触发另一个事件，这样，如果有其它程序监听，就可以收到这个事件，从而可以进行处理，如存入数据库等。
- `http_post`，发一个`HTTP post`请求到一个`HTTP server`，`HTTP server`接收到请求后进行下一步处理。其中，`http_post`是无阻塞的，以提高效率，即只发请求，而不等待处理结果。
- `db`，可以通过`luasql`直接写到数据库，未完全实现
- 当然你也可以直接通过`io.open`写到一个本地文件，未实现...

由于这种脚本会在FreeSWITCH内部执行，需要消耗FreeSWITCH的资源，因此，在大话务量（确切来说是“大事件量”）的情况下还是应该用Event Socket。

完整的代码在 http://svn.freeswitch.org/svn/freeswitch/trunk/seven/lua/gateway_report.lua

其它参考资料：

- FreeSWITCH提供的API: http://wiki.freeswitch.org/wiki/Mod_lua
- Lua语言: <http://www.lua.org/>

11.3 一个在FreeSWITCH中外呼的脚本

有一天，一个朋友问我能否实现在FS中外呼，然后放一段录音，我说当然能，写个简单的脚本就行。但后来他说还要知道呼叫是否成功，我说，那就需要复杂一点了。

当然，这个应用很简单，就没必要使用`event_socket`那些复杂的东东。写了一个Lua脚本，基本能满足要求了。

实现思路是将待呼号码放到一个文件(*number_file_name*)中，每个一行，然后用*Lua*依次读每一行，呼通后播放*file_to_play*，并将结果写到*log_file_name*中。为保证对方应答后才开始播放，需要*ignore_early_media*参数，否则，对方传回铃音或彩铃时播放就会开始，而那不是我们想要的。

先设置一些常量/变量：

```
prefix = "{ignore_early_media=true}sofia/gateway/cnc/"
number_file_name = "/usr/local/freeswitch/scripts/number.txt"
file_to_play = "/usr/local/freeswitch/sounds/custom/8000/sound.wav"
log_file_name = "/usr/local/freeswitch/log/dialer_log.txt"
```

简单起见，包装一个函数：

```
function debug(s)
    freeswitch.consoleLog("notice", s .. "\n")
end
```

定义呼叫函数。*freeswitch.Session*会呼叫一个号码，并一直等待对方应答。然后，使用*streamFile*播放一段声音，挂机。最后，函数返回挂机原因 *hangup_cause*。

```
function call_number(number)
    dial_string = prefix .. tostring(number);

    debug("calling " .. dial_string);
    session = freeswitch.Session(dial_string);

    if session:ready() then
        session:sleep(1000)
        session:streamFile(file_to_play)
        session:hangup()
    end
    -- waiting for hangup
    while session:ready() do
        debug("waiting for hangup " .. number)
        session:sleep(1000)
    end

    return session:hangupCause()
end
```

实际的代码是从下面开始执行的。首先第1行打开存放电话号码的文件（准备读），和呼叫日志（第2行，准备写，追加）。

然后是无限循环(*while*)，每次读取一行，当读到空行或文件尾时，退出。

while 循环中，读到的每一行实际上是一个号码，调用上面定义的*call_number()*函数进行呼叫，并将呼叫结果写到*log_file*中。

```
number_file = io.open(number_file_name, "r")
log_file = io.open(log_file_name, "a+")

while true do

    line = number_file:read("*line")
    if line == "" or line == nil then break end

    hangup_cause = call_number(line)
    log_file:write(os.date("%H:%M:%S ") .. line .. " " .. hangup_cause .. "\n")
end
```

很简单，很强大，是吧？

将脚本存到*scripts*目录中（通常是 */usr/local/freeswitch/scripts/*），起名叫*dialer.lua*，在FreeSWITCH控制台或*fs_cli*中执行：

```
1 luarun dialer.lua
```

完整的脚本：

- <http://fisheye.freeswitch.org/browse/freeswitch-contrib/seven/lua/dialer.lua?hb=true>

另外还有一个 *batch_dialer*:

- http://fisheye.freeswitch.org/browse/freeswitch-contrib/seven/lua/batch_dialer.lua?hb=true

参考：

- FreeSWITCH提供的API: http://wiki.freeswitch.org/wiki/Mod_lua
- Lua语言: <http://www.lua.org/>

11.4 也谈 FreeSWITCH 中语音识别

有段时间有一个语音识别的项目，便轻轻地研究了一下，虽没有达到预期的效果，但过程还是比较有趣，所以就记录了下来。

题目是这样的：给定一些潜在客户，用 FreeSWITCH 自动呼叫，如果用户应答，则转至 IVR，播放欢迎信息甚至转至人工座席；如果客户不应答，则获取不应答原因。

该想法想要达到的目标是：1) 客户关怀。客户注册即可收到关怀电话（当然前提是留下电话号码。OK，发短信是另一种方式，但我这里讨论的是语音）；2) 数据清洗，过滤无效客户。从不同渠道来的客户数据良莠不齐，有的甚至50%以上都无法打通，所以，把这部分数据过滤掉显然是很有意义的。

虽然我们在外呼中使用 *SIP*，但被叫用户在 *PSTN*，而 *PSTN* 信令网一个很令人讨厌的地方就是返回的信令不准确，所以，你无法从信令层面获取被叫用户的状态（空号，忙等），而只能从语音层面去“听”。当然，听，对于人来说是没有问题的，但对于机器来说，就不轻松了，它需要使用语音识别 (ASR, *Automatic Sound Recognition*) 技术来实现。

Sphinx 应该是开源的语音识别公认的比较好的软件。幸运的是 FreeSWITCH 有一个 *pocket_sphinx* 模块。它既能进行连续的识别，也能针对关键词进行识别，在测试阶段，成功率还是比较高的。但实际上我们真正要测的数据由于中英文混杂，没有收到好的效果。

11.4.1 样本

目前 *PSTN* 网上有各种语音数据，除了各种各样的彩铃之外，便是五花八门的语音提示，而且，针对同一种挂机原因，有各种不同版本的语音提示。为了获取样本，我打了不同省市不同运营商的电话并录音：

```
1 originate sofia/gateway/blah/139xxxxxxxx &record(/tmp/testx.wav)
```

作为测试，我选择了以下几种：

1. 您拨的号码是空号，请查证再拨....(无限循环)
2. 对不起，您拨打的用户不方便接听您的电话，请稍后再拨。Sorry, the subscriber you have dialed is not convenient to answer now, please dial again later. (循环...)
3. 您好，您所拨打的号码是空号，请核对后再拨。您好，您所拨打的号码是空号，请核对后再拨。Sorry, the number YOU dialed doesn't exist, please check it and dial again.(循环... 一个问题是你有必要强调吗？)
4. 号码是空号，请查证后再拨。Sorry, The number you have dialed is not in service, please check the number and dial again. (循环)

如果你听一下，你会发现真是太难听了。那么大的电话公司，不能找个专业的人录音吗？
(个人感觉 *test4.wav* 还是比较专业)

- test1.wav: http://commondatastorage.googleapis.com/dujinfang.com/sounds/hangup_cause/test1.wav
- test2.wav: http://commondatastorage.googleapis.com/dujinfang.com/sounds/hangup_cause/test2.wav
- test3.wav: http://commondatastorage.googleapis.com/dujinfang.com/sounds/hangup_cause/test3.wav
- test4.wav: http://commondatastorage.googleapis.com/dujinfang.com/sounds/hangup_cause/test4.wav

11.4.2 第一种方案，关键词

我将几个关键字写进了 grammar 中，如：

```
grammar hpcause;

<service> = [ service ];
<rejected> = [ convenient ];
<busy> = [ busy ];
<konghao> = [ konghao ];
<exist> = [ exist ];
public <hpcause> = [ <service> | <rejected> | <busy> | <konghao> | <exist> ];
```

实际测试中，我甚至将“空号”的拼音“”*konghao*”作为关键词加上去，的确有时候能识别出来。由于中英文混杂，识别率太低。试过纯英文的环境比较理想。

11.4.3 第二种方案，连续识别

当然我也试过连续的语音识别，效果都不理想。*pocket_sphinx* 是支持中文的，但配置比较复杂，而且我也怀疑它在中英文混合识别方面的效果到底如何。

11.4.4 第三种方案，只录音，采用外部程序识别

要想在 FreeSWITCH 中准确识别这么复杂的情况看来是不现实的。另一种想法就是只录音，而采用外部程序（可能还是 *Sphinx*）来识别。可以针对中英文各识别一次，去掉不能识别的部分，我相信效果还是可以的。但没有试过。

11.4.5 第四种方案, Google Voice

实际上 *Google Voice* 有一个很有趣的功能就是 *Voice Mail*, 当你的电话无法接通时, 它可以录音, 并能转换成文本。我今天忽然想到, 能否让 *Google Voice* 来替我们做这项工作? 如果行, 对于每个 *Voice Mail* 我们都能收到一封电子邮件, 岂不是绝了?

我赶紧试了以下命令 (人工换行) :

```
1 originate {ignore_early_media=true}sofia/gateway/blah/1717673xxxx
2   'sleep:3,playback:/home/app/t/test4.wav' inline
```

上面, 我呼叫我的 *Google Voice* 号码, 并播放声音文件, 为了等待 *Google Voice* 启动 *Voice Mail*, 暂停了3秒。其中使用了 FreeSWITCH 的 *inline dialplan*。

不得不说, 人家 *Google Voice* 的功力就是比较深, 以下是呼叫结果 (虽然它花了好长时间生成这些文本) :

1. But how much you called her some cat content, a lot. But how much you called her some cat content walk. But how much you called her chin cat content walk. But how much you called her some cat content water. But how much you called her some cat content walk. But how much you called her team cat content walk. But how much you called her teams have content walk. But how much you called her team cat content walk. But how much you called her team captain sample. But how much you called her some cat content. Blah. But how much you called her team packing 10 o'clock. But.

由于第一条语音是纯中文的, 对比一下, 可以发现 GV 的想象力是比较丰富的。

1	您 拨 的	号 码 是	空 空 号
2	But	How much	Called her

2. Dave, date which he didn't full full time. Thank you can. Looking for some sure. Okay well, sorry the subscriber to get that out. If not, yes right now. Please check it out later, date which he didn't hopeful. I think you can. Looking for some chance. Okay well, sorry. The subscriber to get that out. If not, yes uncertain, so. Please straight out later. If You Keep Doing phone found antiquing employer some chocolates as well. Sorry. The subscriber you have dialed is not company as uncertain. So please straight out later. Hey Vicky, but I think in full full time. Thank you been looking for. So, I'm sure. Okay well, sorry. The subscriber you have dialed is house company as of right now lease without a later Hello. Hey Vicky, like, I can hopeful. I think you can get them some chocolates at. Well, sorry. The subscriber you have dialed is not come again it's on sale now. Please straight out later. Hey Vicky, but I don't know 4 phone so I think you can get some fire some chocolate at 40 sorry. The subscriber you have dialed is not come again it's on sat. Now, Please, straight out later.

“not convenient” 跟 “not come again” 相比意思应该差不多吧? 哈 :D。

3. Com critical table, the com the job without the how much you can call team critical examples of the the number in the Dells doesn't exist. Please check it and Dale again the com job without a, how much you can call few critical table, the com the job without a, how much you can call team critical to thank both of these, the number in the Dells doesn't exist. Please check it and down again.

至少有关键字 “doesn't exist”

4. This is not in service. Please check the number and dial again, yeah. How, the bloodhound I should call me a call. If you could take a look at all. The number you have dialed is not in service. Please check the number and dial again.

事实证明老四的英文说的不错，对英文部分的识别几乎一字不差。

11.4.6 结论

看来中英混合识别的难度还是比较大的。但我想，不管PSTN的语音再怎么变态，毕竟它是有限的，我们只要采集所有的语音样本，做到90%以上识别率应该是不难做到的。

11.5 在 FreeSWITCH 中使用 google translate 进行文本语音转换

今天，偶然发现 google translate 一个很酷的功能 — TTS。

在浏览器中输入（为排版方便进行了人工换行）：

```
1 http://translate.google.com/translate_tts?  
2 q=hello+and+welcome+to+w+w+dot+dujinfang+dot+com&tl=en
```

然后立即就可以播放声音。又试了一下下面这个，也好用。

```
1 http://translate.google.com/translate_tts?q=欢迎光临七哥的博客&tl=zh
```

我在Mac上分别用 Safari, Chrome 和 FireFox 都测试通过。

那么，能不能在 FreeSWITCH 里用呢？当然，FreeSWITCH 通过 *mod_shout* 支持 *mp3*！

默认的 FreeSWITCH 中 *mod_shout* 是不编译的，所以需要自己编译。到源代码目录下，执行

```
1 make mod_shout-install
```

就装好了（当然，前提是已经用源代码安装了 FreeSWITCH 的情况，参见第二章）。

在 FreeSWITCH 命令行上装入模块：

```
1 load mod_shout
```

测试一下（人工换行，实际应为一行）：

```
1 originate user/1000 &playback(shout://translate.google.com/translate_tts?
2 q=hello+and+welcome+to+www+dot+dujinfang+dot+com&tl=en)
```

太爽了。但中文的没有成功，不知道为什么。

当然你也可以写到 *dialplan* 中，然后呼叫 1234 试一下 :D（为了排版方便，我换行了，记着 shout 那一行别断行）

```
<extension name="Free_Google_Text_To_Speech">
  <condition field="destination_number" expression="^1234$">
    <action application="answer" data="" />
    <action application="playback"
      data="shout://translate.google.com/translate_tts?
      q=hello+and+welcome+to+www+dot+dujinfang+dot+com&tl=en"/>
  </condition>
</extension>
```

11.6 FreeSWITCH 与 H323

FreeSWITCH 中有两个 *H323* 的实现，*mod_opal* 与 *mod_h323*。两者都使用 *ptlib*，后者比较新一点。以前曾经测试过 *mod_opal*，但没有成功，改天试了一下 *h323*，成功了。

实际步骤跟 Wiki 上描述的差不多，我使用的是 *ptlib-2.8.2* 和 *h323plus-20100525*。OS 为 CentOS 5.5。

安装过程中出现找不到头文件的错误，我没按 Wiki 上指示的 *copy* 文件，而是直接修改 *src/mod/end_points/mod_h323/Makefile* 把 */usr/include* 及 */usr/lib* 都改为 */usr/local/include* 及 */usr/local/lib*，然后 *make && make install* 成功。

由于我测试的 FS 在一台远程服务器上，NAT 环境中，而 *h323* 穿越 NAT 就在我看来比 SIP 复杂多了。于是我装了一个 PPTP VPN，远程拨入后，加了一个 listener(其中 192.168.6.100 为 VPN server 端IP)：

load mod_h323 测试连接成功。

mod_h323 使用了 *h323plus* 库，貌似可以在编译时通过 *enable-h264* 之类的支持视频，但 *mod_h323* 中还没有支持视频的选项。无论如何，音频还是相当好的。当然测试时也发现有锁的情况，通话非正常中断可能会出问题，也遇到一次 *core dump*。鉴于模块还在开发中，有些错误是在所难免的。

11.7 视频会议

FreeSWITCH的会议功能非常强大，也支持视频。

使用视频会议首先，要在 sofia profile 中设置支持的视频编码，简单起见我直接在 vars.xml 中设置了：

```
1 <X-PRE-PROCESS cmd="set" data="global_codec_prefs=PCMU,PCMA,GSM,H264,H263-1998,H263"/>
2 <X-PRE-PROCESS cmd="set" data="outbound_codec_prefs=PCMU,PCMA,GSM,H264,H263-1998,H263"/>
```

拿两个视频电话互拨，先保证都有视频。当然，为了防止多种视频编码在协商时可能出现问题，可以在FreeSWITCH或终端软件中仅使用一种视频编码。

11.7.1 普通视频会议

测试会议前，将所有视频设备设成同一种视频编码，如H263或H264。拨打默认的电话会议号码3000，电话会议正常。会议流程是这样的：

- 每个会议里都有一个 *video* 线程
- 每个会议里会有一个标志，叫做 *floor*，一般来说，当前正在发言的人会拥有这个 *floor*
- 拥有 *floor* 的人的视频会广播到所有的终端上，包括它自己

如果在会议中，另一个人开始讲话，视频就会发生切换，但切换的画面会出现马赛克，而且有些慢，即使在局域网环境中也如此。其实这个问题跟终端电话的实现有关。FreeSWITCH会在*floor*切换时给终端发送一个RTCP Fast Update¹，终端收到该命令后应该立即发送关键帧，这样就应该不会出现马赛克。但有的话机不支持该协议。

11.7.2 多画面融屏

前一段时间，有个合作伙伴拿了個视频处理芯片，想尝试做一种视频融屏的会议应用，研究了一下，后来倒也成功了。

需要说明的是，这不是 FreeSWITCH 的基本功能，但实现思路还是比较有借鉴意义的，同时也体现了FreeSWITCH强大的可定制性，因此不妨在这儿提一下。

该视频芯片提供SDK，我们很快在其提供的 *demo*代码基础上做了一个四分屏融屏程序，思路很简单，进去四路rtp视频流，出来一路rtp，如下图。为简单起见，我们都使用同一种视频编码—H264。

¹<http://tools.ietf.org/html/rfc5168>

```

1          /-<-<-<-<-<-<-<-<-<-<-<-\ 
2          /-----\           /-----|-----\ 
3 Phone1 --<----SIP/RTP----->---| FreeSWITCH |           | 
4          \-----/----->---| Video Mixer |           | 
5 Phone2 --<----SIP/RTP----->---/ / / | |----->---| 
6 Phone3 --<----SIP/RTP----->---/ / / |----->---| 
7 Phone4 --<----SIP/RTP----->---/ |----->---\-----/ 
8

```

我们改了一些 FreeSWITCH 代码，在会议开始时，每加入一方就将其视频直接转发到 *Video Mixer* 视频处理芯片上，它会把各路视频融合，并返回一路视频，FreeSWITCH 再启动一个线程接收该视频，并发送到所有视频终端上。所有终端都能看到所有与会者融合以后的画面，而混音功能仍由 FreeSWITCH 提供，非常帅。

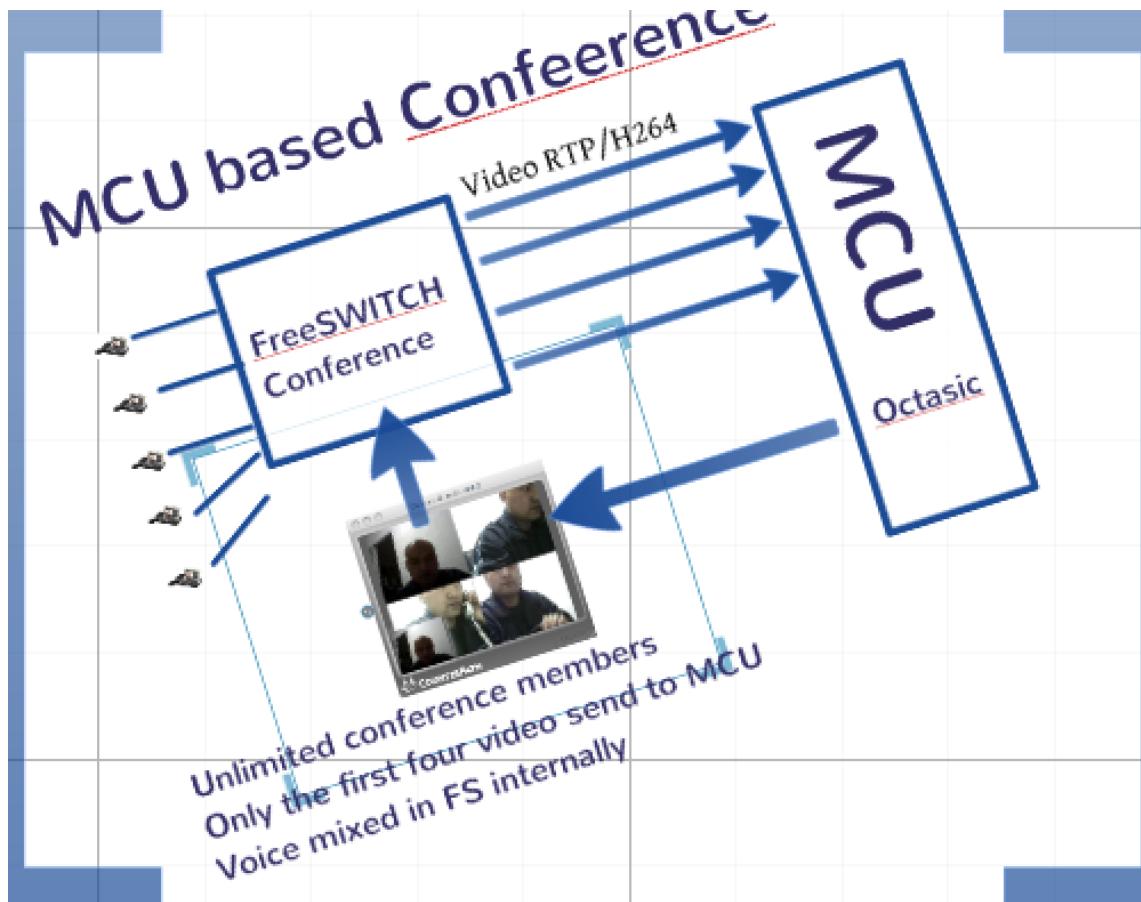


图 11.1: 视频融屏

11.8 多台 FreeSWITCH 服务器级联

其实，只要你吃透了前些章节的内容，做 FreeSWITCH 级联是没有任何问题的。但这个问题还常常被众网友问到，我就索性再写一篇。

11.8.1 双机级联

假设你有两台 FreeSWITCH 机器，分别为A和B，同样IP分别为 192.168.1.A 和 192.168.1.B。每台机器均为默认配置，也就是说在每台机器上 *1000 ~ 1019* 这 20 个号码可以互打电话。位于同一机器上的用户称为“网内用户”，如果需要与其它机器上的用户通信，则其它机器上的用户就称为“网外用户”。

现在你需要在两台机器之间的用户互拨，因此你想了一种拨号方案。如果*A**1000*想拨打*B**1000*，则*B**1000*相对于*A**1000*来说就是“网外用户”。就一般的企业PBX而言，一般拨打外网用户就需要加一个特殊的号码，比方说“*0*”。这样，“*0*”就称为“出局字冠”。

好了，我们规定，不管是*A*的用户还是*B*上的用户，拨打外网用户均需要加“*0*”。下面我们仅配置*A*打*B*，把*B*打*A*的情况留给读者练习。

在*A*机上，把以下 *dialplan* 片断加到 *default.xml* 中：

```

1 <extension name="B">
2   <condition field="destination_number" expression="^0(.*)$">
3     <action application="bridge" data="sofia/external/sip:$10192\168\1\.B:5080"/>
4   </condition>
5 </extension>
```

其中，*expression=* 后面的正则表示式表示匹配以*0*开头的号码，“吃”掉*0*后，把剩下的号码送到*B*机的*5080*端口上。

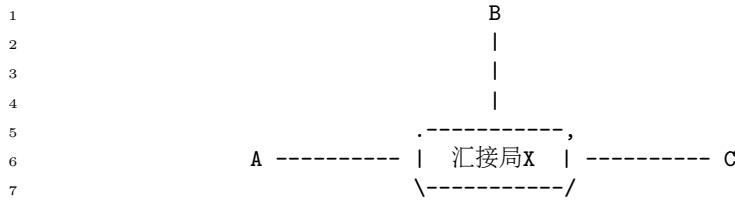
所以，如果用户*1000*在*A*上拨 *01000*，将会发送 *INVITE sip:1000@192.168.1.B:5080* 到*B*上。*B*收到后，由于*5080*端口默认走public dialplan*，所以查找 *public.xml*，找到*1000*后将电话最终接续到*B*机的*1000*用户。

除了SIP外，我还在两台机器上分别加了两块E1板卡，中间用交叉线直连，这样的话，我希望拨9开头就走E1到对端，设置如下：

```

1 <extension name="B_E1">
2   <condition field="destination_number" expression="^9(.*)$">
3     <action application="bridge" data="freetdm/1/a/$1"/>
4   </condition>
5 </extension>
```

11.8.2 汇接模式



其实你搞定了第一种模式以后，这种汇接模式也就很简单了。无非你需要动一动脑子做一下拨号计划，比方说到A拨0，B拨1，到C拨2之类的。然后在汇接局配置相关的 *dialplan* 就OK了。

遇到 *dialplan* 的问题还是再看一下第八章，还是那句话，使用 F8 打开详细的 *LOG*，打一个电话，从绿色的行开始看。

11.8.3 安全性

上面的方法只使用 5080 端口从 *public dialplan* 做互通，而发送到 5080 端口的 INVITE 是不需要鉴权的，这意味着，你任何人都可以向它发送 INVITE 从而按你设定的路由规则打电话。这在第一种模式下问题可能不大，因为你的 *public dialplan* 仅将外面的来话路由到本地用户。但在汇接局模式下，你可能将一个来话再转接到其它外部网关中去，那你就需要好好考虑一下安全问题了，因为你肯定不希望全世界的人都用你的网关打免费电话。

一般说来，解决这个问题有两种方式，那就是让所有来话都经过认证鉴权后再进行路由（本地用户发到 5060 端口上都是需要鉴权的）。

考虑双机级联的情况，你只需要在 A 上配置一个到 B 的网关（将下列内容存成 XML 文件放到 *conf/sip_profiles/external/b.xml*）：

```

1 <include>
2   <gateway name="b">
3     <param name="realm" value="192.168.1.B"/>
4     <param name="username" value="1000"/>
5     <param name="password" value="1234"/>
6   </gateway>
7 </include>

```

同时把 A 上的 *dialplan* 改成：

```

1 <extension name="B">
2   <condition field="destination_number" expression="^0(.*)$">
3     <action application="bridge" data="sofia/gateway/b/$1"/>
4   </condition>
5 </extension>

```

这样，*A*上的用户可以呼通所有*B*上的用户，从*B*的用户来看，好像所有电话都是从本机的1000这个用户打进来的（这就是网关的概念，因为对于*B*来说，*A*机就相当于一个普通的SIP用户1000。当然你从*A*上理解，*B*就是给你提供了一条SIP中继，如果在*B*上解决了“主叫号码透传”以后，*B*就相当于一条真正的中继了）。如果说还是不容易理的话，想像一下*B*是联通或电信的服务器网关，你是不能控制的，而它只给了你一个网关的IP，用户名，和密码，你把它配到你的*A*上，就可以呼通电信能呼通的任何固定电话或手机了。

11.9 万能 FreeSWITCH directory 脚本

好多人问我如何使用 mod_xml_curl 进行用户验证，每次回答指导都很费劲。某天我抽用 PHP 写了一个万能脚本，希望对大家有帮助。

FreeSWITCH 默认使用静态的 XML 文件配置用户，但如果需要动态认证，就需要跟数据库关联。FreeSWITCH 通过使用 mod_xml_curl 模块完美解决了这个问题。实现思路是你自己提供一个 Web 服务器，当有用户注册（或 INVITE）请求时，FreeSWITCH 向你的 Web 服务器发送请求，你查询数据库生成一个标准的XML文件，FreeSWITCH 进而通过这一文件对用户进行认证。

11.9.1 脚本

好了，别的不多说了，看脚本(用PHP实现):

```

1  <?php
2      $user = $_POST['user'];
3      $domain = $_POST['domain'];
4      $context = $_POST['Hunt-context'];
5      $password = "1234";
6  ?>
7  <document type="freeswitch/xml">
8  <section name="directory">
9      <domain name=<?php echo $domain;?>">
10     <params> <!-- 注意下一行有一个人工换行 -->
11         <param name="dial-string" value="<${presence_id}=${dialed_user}@${dialed_domain} ${sofia_contact}(${dialed_user}@${dialed_domain})}" />
12     </params>
13     <groups>
14         <group name="default">
15             <users>
16                 <user id=<?php echo $user; ?>">
17                     <params>
18                         <param name="password" value="<?php echo $password; ?>"/>
19                         <param name="vm-password" value="<?php echo $password; ?>"/>
20                     </params>
21                     <variables>
22                         <variable name="toll_allow" value="domestic,international,local"/>
23                         <variable name="accountcode" value="<?php echo $user; ?>"/>
24                         <variable name="user_context" value="<?php echo $context ?>"/>
25                         <variable name="effective_caller_id_name" value="FreeSWITCH-CN"/>
26                         <variable name="effective_caller_id_number" value="<?php echo $user;?>"/>
27                         <variable name="outbound_caller_id_name" value="$$ ${outbound_caller_name}"/>
28                         <variable name="outbound_caller_id_number" value="$$ ${outbound_caller_id}"/>
29                         <variable name="callgroup" value="default"/>
30                         <variable name="sip-force-contact" value="NDLB-connectile-dysfunction"/>
31                         <variable name="x-powered-by" value="http://www.freeswitch.org.cn"/>
32                     </variables>
33                 </user>
34             </users>
35         </group>
36         </groups>
37     </domain>
38 </section>
39 </document>

```

之所以称这是万能脚本，是因为它根本不查询数据库，任何注册请求只要密码是 1234 就都能通过注册。

好了，把上述PHP文件放到你的服务器上，确保它能正确执行。

接下来配置你的 FreeSWITCH, conf/autoload_configs/xml_curl.conf.xml

```

1 <configuration name="xml_curl.conf" description="cURL XML Gateway">
2   <bindings>
3     <binding name="directory">
4       <param name="gateway-url"
5           value="http://localhost/~seven/freeswitch/directory.php"
6           bindings="directory"/>
7     </binding>
8   </bindings>
9 </configuration>

```

然后

```

1 reloadxml
2 reload mod_xml_curl

```

拿起你的SIP电话注册试试吧，别忘了万能密码是 1234。

然后怎么办？把最开头的几行换能你的业务逻辑（查询数据库等），就实现你自己的认证了。

上面的 php 脚本也放到 *github* 上了：<https://gist.github.com/1086122>

11.9.2 调试

- *load mod_xml_curl* 错误

mod_xml_curl 默认是不编译的，到你的源代码目录中执行 *make mod_xml_curl-install*

- 还是不行

同学，“不行”是个很笼统的说法，遇到问题，你要找出哪里出错了，至少，要知道发生了什么现象。在 FS 中执行 *xml_curl debug_on*，FS 会把每次请求生成的 XML 存到类似 */tmp/xxx.xml* 的一个文件里，看看里面有什么。

11.10 使用 Erlang 控制呼叫流程

11.10.1 概述

Erlang 是一门函数式编程语言。相比其它语言来说，它是一门小众语言，但它具有轻量级多进程、高并发、热代码替换等地球人不能拒绝的特性，并非常易于创建集群应用(Cluster)。更重要的是，它天生就是为了编写电信应用的。

除电信领域外，它在游戏及金融领域也担负着重要的作用。就 FreeSWITCH 来讲，[OpenACD](#) 及 [Whistle](#) 已是比较成熟项目。

FreeSWITCH 有一个原生的模块叫 `mod_erlang_socket`，它作为一个隐藏的Erlang节点(Hidden Node)可以与任何 Erlang 节点通信。

以下假定读者有一定 Erlang 基础，其基本原理和语法不再赘述。

11.10.2 安装 Erlang 模块

Erlang 模块默认是不安装的，首先请确认你的机器上已经安装 Erlang 的开发环境。我总是通过源代码安装 (`./configure && make && make install`)，如果通过管理工具安装的请确认有 `erlang-devel` 或 `erlang-dev` 包。

确认安装好 Erlang 环境以后，在 FreeSWITCH 源代码目录中重新执行 `./configure`，以产生相关的 Makefile。

然后执行

```
1 make mod_erlang_event-install
```

在 FreeSWITCH 控制台上执行

```
1 load mod_erlang_event
```

11.10.3 把电话控制权转给 Erlang

在该例子中，Erlang 程序作为一个节点运行，它类似于 Event Socket 概念中的外连套接字 (outbound socket)。

在我们开始之前，先检查 Erlang 模块的设置，确认 `erlang_event.conf.xml` 中有以下三行（其它的行不动）。

```
<param name="nodename" value="freeswitch@localhost"/>
<param name="encoding" value="binary"/>
<param name="cookie" value="ClueCon"/>
```

其中第一行的 `nodename` 是为了强制 FreeSWITCH 节点的名字，如果不配置的话，FreeSWITCH 会自己的节点起一个最适合的名字，但我们发现它自己起的名字并不总是正确，因此在这里为了防止引起任何可能的错误，人工指定一个名字。注意这里我们使用了短名字，如果你的 Erlang 程序在另一台机器上，你应该使用长名字。

第二行，我们使用二进制编码。默认的节点间通信使用文本编码，文本编码比二进制编码效率要低一些。

第三行，设置该节点的 Cookie。注意要与 Erlang 节点的要相同。

在 Erlang 代码里，我们先定义一些宏：

```
-define(FS_NODE, 'freeswitch@localhost').
-define(WELCOME_SOUND, "tone_stream://%(100,1000,800);loops=1").
-define(INPUT_NUMBER_SOUND, "tone_stream://%(100,1000,800);loops=2").
```

在这里，欢迎音（WELCOME_SOUND）是一个”嘀“声，请输入一个号码（INPUT_NUMBER_SOUND）使用两个”嘀“声。我们这么做的目的是使读者能直接编译代码而不用依赖于任何声音文件，当然，为了能更好的理解这个例子，你应该换成更观的声音文件，如“/tmp/welcome.wav”（您好，欢迎，请输入一个号码）。

为了使用整个通信过程更加透明化，我没有使用缺省的 freeswitch.erl 库，而是自己写了几个函数用于在 Erlang 程序和 FreeSWITCH 间传递真正的消息。

11.10.4 把电话发给 Erlang

在 outbound 模式下，FreeSWITCH（可以看作一个客户端）会连接到你的 Erlang 程序节点（可以看作一个服务器）上，并把进来的电话控制权送给它。

在 Dialplan 中有两种设置方法：

```
1 <extension name="test">
2     <condition field="destination_number" expression="^7777$">
3         <action application="erlang" data ="ivr:start test@localhost"/>
4     </condition>
5 </extension>
6
7 <extension name="test">
8     <condition field="destination_number" expression="^7778$">
9         <action application="erlang" data ="ivr:! test@localhost"/>
10    </condition>
11 </extension>
```

7777 法

7777 法是我起的名字。在这种方法里，如果你呼叫 7777，FreeSWITCH 会首先将当前的 Channel 给 Park 起来，并给你的 Erlang 程序（test@localhost 节点）发送一个 RPC 调用，调用的函数为：ivr:start(Ref)。其中，ivr:start 是你在上述 XML 中定义的，Ref 则是由 FreeSWITCH 端生成的针对该请求的唯一引用。在 Erlang 端，start/1 函数应该 spawn 一个新的进程，并且并新进程的 PID 返回，FreeSWITCH 收到后会将与该 Channel 相关的所有事件（Event）送给该进程。

```

1  start(Ref) ->
2      NewPid = spawn(fun() -> serve() end),
3      {Ref, NewPid}.

```

上述代码产生一个新进程，新进程将运行 `serve()` 函数，同时原来的进程会将新进程的 `Pid` 返回给 FreeSWITCH。

这是用 Erlang 控制呼叫的最简单的方法。任何时候来了一个呼叫，FreeSWITCH 发起一个远端 RPC 调用给 Erlang，Erlang 端启动一个新进程来控制该呼叫。由于 Erlang 的进程都是非常轻量级的，因而这种方式非常优雅（当然，Erlang 端也不用每次都启动一个新进程，比如说可以事先启动一组 `Pid`，看到进来的请求时选择一个空闲的进程来为该电话服务。当然，正如我们一再强调的，Erlang 中的进程是非常轻量级的，产生一个新进程也是相当快的，所以，没有必要用这种传统软件中“进程池”那么复杂方法）。

我们将在后面再讨论 `serve()` 函数。

7778 法

除了使用 RPC 生成新的进程外，还可以用另外一种方法产生新进程。如上面 7778 所示，与 7777 不同的是，其中的 `ivr:start` 中的 `start` 换成了“`!`”。该方法需要你首先在 Erlang 端启动一个进程，该进程监听也有进来的请求。当 FreeSWITCH 把电话路由到 Erlang 节点时，“`!`”语法说明，FreeSWITCH 会发送一个 `{getpid, ...}` 消息给 Erlang，意思是说，我这里有一个电话，告诉我哪个 `Pid` 可以处理，我好把相应的事件发送给它。这种方法比上一种方法稍微有点复杂，但程序控制更加灵活。

```

1  start() ->
2      register(ivr, self()),
3      loop().
4
5  loop() ->
6      receive
7          {get_pid, UUID, Ref, FSPid} ->
8              NewPid = spawn(fun() -> serve() end),
9              FSPid ! {Ref, NewPid},
10             ?LOG("Main process ~p spawned new child process ~p~n", [self(), NewPid]),
11             loop();
12             _X ->
13                 loop()
14         end.

```

上面的代码中，用户应在 Erlang 节点启动后首先执行 `start/0` 以启动监听。`start()` 时，将首先注册一个名字，叫 `ivr`，然后进入 `loop` 循环等待接收消息。一旦它收到 `{get_pid, ...}` 消息，就 `spawn` 一个新的进程，并把新进程的 `Pid` 发送给 FreeSWITCH，然后再次等待新的消息。在这里，新产生的进程同样执行 `serve()` 函数为新进来的 Channel 服务。

11.10.5 呼叫控制

我们来做这样一个例子。当有电话进入时，它首先播放欢迎音：“您好，欢迎致电 XX 公司...”，然后让用户输入一个电话号码：“请输入一个分机号”，接下来我们会检查号码并转接到该号码，如果不正确，则重新让用户输入。

```

serve() ->
    receive
        {call, {event, [UUID | Event]}} ->
            ?LOG("New call ~p~n", [UUID]),
            send_msg(UUID, set, "hangup_after_bridge=true"),
            send_lock_msg(UUID, playback, ?WELCOME_SOUND),
            send_msg(UUID, read, "1 4 " ?INPUT_NUMBER_SOUND " erl_dst_number 5000 #"),
            serve();
        {call_event, {event, [UUID | Event]}} ->
            Name = proplists:get_value(<<"Event-Name">>, Event),
            App = proplists:get_value(<<"Application">>, Event),
            ?LOG("Event: ~p, App: ~p~n", [Name, App]),
            case Name of
                <<"CHANNEL_EXECUTE_COMPLETE">> when App =:= <<"read">> ->
                    case proplists:get_value(<<"variable_erl_dst_number">>, Event) of
                        undefined ->
                            send_msg(UUID, read, "1 4 " ?INPUT_NUMBER_SOUND
                                " erl_dst_number 5000 #");
                        Dest ->
                            send_msg(UUID, bridge, "user/" ++ binary_to_list(Dest))
                    end;
                _ -> ok
            end,
            serve();
        call_hangup ->
            ?LOG("Call hangup~n", []);
        _X ->
            ?LOG("ignoring message ~p~n", [_X]),
            serve()
    end.

```

到这里，我们再回想一下，FreeSWITCH 将一个新来电置为 Park 状态，然后向你的 Erlang 程序要一个 Pid，得到后会将所有与该 Channel 相关的消息发送给该 Pid（而该 Pid 现在正在运行 serve() 函数等待新消息）。如果一切如我们所愿，该 Pid 收到的第一个消息永远是 {call, {event, [UUID / Event]}}。好了，接下来就看你的了。

首先，为了学习方便，你打印了一条 Log 消息。然后，你通过设置 *hangup_after_bridge=true* 来确保该 Channel 能正常挂断（你可以看到，跟在 dialplan 中的设置是一样的）。接下来，你播

放 (playback) 了欢迎音。注意，这里的 `send_lock_msg()` 很关键，它确保 FreeSWITCH 在播放完当前文件后再执行收到的下一条消息。

后续的消息都类似 `{call_event, {event, [UUID | Event]}}`。所以，如果你收到 CHANNEL_EXECUTE_COMPLETE 消息后，检测到其 `Application` 参数是 `read` 时，它表示用户按下了按键（或者超时），在上面的代码中，它就会 `bridge` 到某一分机（或者在输入超时的情况下重新询问号码）。

当收到 `call_hangup` 消息时，意味着电话已经挂机了，我们的Erlang进程没有别的事可做，就轻轻的消亡了。

下面是 `send_msg` 和 `send_lock_msg` 函数：

```

1  send_msg(UUID, App, Args) ->
2      Headers = [{"call-command", "execute"},
3                  {"execute-app-name", atom_to_list(App)}, {"execute-app-arg", Args}],
4      send_msg(UUID, Headers).
5  send_lock_msg(UUID, App, Args) ->
6      Headers = [{"call-command", "execute"}, {"event-lock", "true"},
7                  {"execute-app-name", atom_to_list(App)}, {"execute-app-arg", Args}],
8      send_msg(UUID, Headers).
9  send_msg(UUID, Headers) -> {sendmsg, ?FS_NODE} ! {sendmsg, UUID, Headers}.

```

11.10.6 使用状态机实现

事实上，一个 `channel` 在不同的时刻有不同的状态，所以呼叫流程控制非常适合用状态机实现。Erlang OTP 有一个 `gen_fsm behaviour`，使用它可以非常简单的实现状态机。

本例中我使用了一个 `gen_fsm` 的增强版本，叫做 `gen_f fsm2`。它在原先的基础上增加了以下几个函数。

在收到 FreeSWITCH 侧的大部分消息时回调 `Module:StateName(Message, StateData)`，如

```
1  wait_bridge({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, Event}, State)
```

在收到 CHANNEL_HANGUP_COMPLETE 及 CHANNEL_DESTROY 类的消息时回调以下函数

```

1  Module:handle_event({channel_hangup_event, UUID, Event}, StateName, State) on
2  Module:handle_event({channel_destroy_event, UUID, Event}, StateName, State) on CHANNEL_DESTROY

```

在这里我们使用 `gen_f fsm` 来实现上面的例子。不过在这时里我们增加了一个语言提示：“请选择提示语言，1 为普通话，2 为英语...”，以使用读者对本例有更直观的印象。

²<https://gist.github.com/1354864>

见下面的代码。首先我们定义一个记录（见第XX行）来“记住” FSM 进程中的一些东西。简单起见，我们只是使用上面提到的7778法来启动 FSM 进程。

进程启动 (*init*) 时，我们简单的过渡到 *welcom* 状态并等待，收到第一条消息后开始播放欢迎声音，然后让主叫用户输入一个按键以选择语言，同时进程转换到 *wait_lang* 状态并等待。其中，我们将主叫号码 (caller id) 等记在状态机的 *state* 变量中。

当用户有按键输入时，会收到 *CHANNEL_EXECUTE_COMPLETE* 消息，根据按键它会通过设置 *sound_prefix* 信道变量以支持不同语种的声音。接下来会继续让用户输入一个号码，并进入 *wait_number* 状态。

在 *wait_number* 状态可能会收到一个合法的号码，然后进行呼叫，或者用户输入超时，它就会播放声音提示用户重新输入一个号码。

当进程转移到 *wait_hangup* 状态时（表示电话已接通，双方正在对话），它会把收到的消息统统打印出来，所以你会看到 *CHANNEL_BRIDGE*, *CHANNEL_UNBRIDGE* 等消息。当然收到这些消息后你可以选择更新数据库有更有用的事件，这里就不再多说了。

当然，你肯定已经猜出来了，当回调函数执行到 *handle_event({channel_hangup_event, ...})* 时进程会清理现场并终止。

```
-module(fsm_ivr).
-behaviour(gen_fsfsm).

-export([start/1, init/1, handle_info/3, handle_event/3, terminate/3]).
-export([welcome/2, wait_lang/2, wait_number/2, wait_hangup/2]).

-define(FS_NODE, 'freeswitch@localhost').
-define(WELCOME_SOUND, "tone_stream://%(100,1000,800);loops=1").
-define(INPUT_NUMBER_SOUND, "tone_stream://%(100,1000,800);loops=2").
-define(SELECT_LANG_SOUND, "tone_stream://%(100,1000,800);loops=3").
-define(LOGFmt, Args), io:format("~b " ++ Fmt, [?LINE | Args])).

-record(state, {
    fsnode      :: atom(),           % freeswitch node name
    uuid        :: undefined | string(), % channel uuid
    cid_number  :: undefined | string(), % caller id
    dest_number :: undefined | string() % called number
}).

start(Ref) ->
    {ok, NewPid} = gen_fsfsm:start(?MODULE, [], []),
    {Ref, NewPid}.

init([]) ->
    State = #state{fsnode = ?FS_NODE},
```

```

{ok, welcome, State}.

%% The state machine
welcome({call, _Name, UUID, Event}, State) ->
    CidNumber = proplists:get_value(<<"Caller-Caller-ID-Number">>, Event),
    DestNumber = proplists:get_value(<<"Caller-Caller-Destination-Number">>, Event),
    ?LOG("welcome ~p", [CidNumber]),
    send_lock_msg(UUID, playback, ?WELCOME_SOUND),
    case u_utils:get_env(dtmf_type) of
        {ok, inband} ->
            send_msg(UUID, start_dtmf, "");
        _ -> ok
    end,
    send_msg(UUID, read, "1 1 " ?SELECT_LANG_SOUND " erl_lang 5000 #"),
    {next_state, wait_lang,
     State#state{uuid = UUID, cid_number = CidNumber, dest_number = DestNumber}}.

wait_lang({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, Event}, State) ->
    case proplists:get_value(<<"Application">>, Event) of
        <<"read">> ->
            DTMF = proplists:get_value(<<"variable_erl_lang">>, Event),
            LANG = case DTMF of
                <<"1">> -> "cn";
                <<"2">> -> "fr";
                _ -> "en"
            end,
            send_msg(UUID, set, "sound_prefix=/usr/local/freeswitch/sounds/" ++ LANG);
        _ -> ok
    end,
    send_msg(UUID, read, "1 4 " ?INPUT_NUMBER_SOUND " erl_dst_number 5000 #"),
    {next_state, wait_number, State};
wait_lang(_Any, State) -> {next_state, wait_lang, State}. % ignore any other messages

wait_number({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, Event}, State) ->
    case proplists:get_value(<<"Application">>, Event) of
        <<"read">> ->
            case proplists:get_value(<<"variable_erl_dst_number">>, Event) of
                undefined ->
                    send_msg(UUID, read,
                            "1 4 " ?INPUT_NUMBER_SOUND " erl_dst_number 5000 #"),
                    {next_state, wait_number, State};
                Dest ->
                    send_msg(UUID, bridge, "user/" ++ binary_to_list(Dest)),
                    {next_state, wait_hangup, State}
            end
    end

```

```

        end;
    _ -> {next_state, wait_number, State}
end;
wait_number(_Any, State) -> % ignore any other messages
    {next_state, wait_number, State}.

wait_hangup({call_event, Name, _UUID, Event}, State) ->
    ?LOG("Event: ~p~n", [Name]),
    {next_state, wait_hangup, State};
wait_hangup(_Any, State) -> % ignore any other messages
    {next_state, wait_hangup, State}.

handle_info(call_hangup, StateName, State) ->
    ?LOG("Call hangup on ~p~n", [StateName]),
    {stop, normal, State};
handle_info(_Info, StateName, State) ->
    {next_state, StateName, State}.

handle_event({channel_hangup_event, UUID, Event}, StateName, State) ->
    %% perhaps do bill here
    HangupCause = proplists:get_value(<<"Channel-Hangup-Cause">>, Event),
    ?LOG("Hangup Cause: ~p~n", [HangupCause]),
    {stop, normal, State}.

terminate(normal, _StateName, State) -> ok;
terminate(Reason, _StateName, State) ->
    % do some clean up here
    send_msg(State#state.uuid, hangup, ""),
    ok.

%% private functions
send_msg(UUID, App, Args) ->
    Headers = [{"call-command", "execute"}, {"execute-app-name", atom_to_list(App)}, {"execute-app-arg", Args}],
    send_msg(UUID, Headers).
send_lock_msg(UUID, App, Args) ->
    Headers = [{"call-command", "execute"}, {"event-lock", "true"}, {"execute-app-name", atom_to_list(App)}, {"execute-app-arg", Args}],
    send_msg(UUID, Headers).
send_msg(UUID, Headers) -> {sendmsg, ?FS_NODE} ! {sendmsg, UUID, Headers}.

```

11.11 呼叫是怎样工作的?

到这里，我假设读者已经仔细阅读了前面一些章节，并做过一些实验。在这里再强调一次，实验是非常重要的，如果你还没有做，请再翻一下前面的章节，至少安装上FreeSWITCH，注册两个电话互相拨打，并把一些示例号码都拨打测试一下。

本节我们分析一下一个呼叫在FreeSWITCH里面到底是怎么工作的，涉及到的一些基础概念我们会简单复习一下，复习不到的请回去翻阅一下前面的章节。

首先，我们假设你用的是默认的配置，并从1000呼叫1001。我们把1000称为“主叫”，而1001则称为“被叫”。

1000的SIP话机作为UAC会发送INVITE请求到FreeSWITCH的5060端口，也就是到达mod_sofia的internal这个profile所配置的UAS（见conf/sip_profiles/internal.xml）。该UA收到正确的INVITE后会返回100响应码，表示我收到你的请求了，稍等片刻。该UAS中对所有收到的INVITE都要进行鉴权（因为auth-calls=true）。它会先检查ACL（访问控制列表。一般用于IP鉴权，我们暂时还没学到ACL，留到后面去讲）。默认ACL检查是不通过的，因此就会走到密码鉴权阶段，SIP没有发明新的鉴权方式，而是使用与HTTP协议中一样的Digest Auth³进行鉴权的。其特点就是密码并不在网络上传输，因而比较安全。读者可以翻到第四章看一下具体的认证流程，一般是UAS回401，然后UAC重新发送带鉴权信息的INVITE。UAS收到后，便将鉴权信息提交到上层的FreeSWITCH代码，FreeSWITCH就会到directory中查找相应的用户，在这里它会找到conf/directory/default/1000.xml，并根据其中配置的密码信息进行鉴权。如果鉴权不通过，则回送403等错误信息，会话就结束了。

如果鉴权通过，FreeSWITCH就取到了用户的信息，比较重要的是user_context，在我们的例子中它的值为default。接下来电话进入路由（Routing）阶段，开始查找dialplan。由于用户的context是default，因此路由就从default这个dialplan查起（由conf/dialplan/default.xml定义）。

查找dialplan的流程已经在第八章讲过了，总之它会帮你找到1001这个用户，并执行bridge user/1001。user/1001称为呼叫字符串，它会再次查找directory，找到conf/directory/default/1001.xml。由于这里1001是被叫，因此它会进一步查找直到找到1001实际注册的位置。实际上，它实际的注册位置是用dial-string参数来表示的，由于所有用户的规则都一样，因此该参数被放到conf/directory/default.xml中，你会看到（人工换行，2-4行实际上为一行）：

```

1 <param name="dial-string"
2   value="{sip_invite_domain=${dialed_domain},
3   presence_id=${dialed_user}@${dialed_domain}}
4   ${sofia_contact(${dialed_user}@${dialed_domain})}">
5 </params>
```

其中，最关键的是sofia_contact这个API调用，它会查找数据库，找到1001实际注册的Contact地址，并返回真正的呼叫字符串。如在我的机器上：

³http://en.wikipedia.org/wiki/Digest_access_authentication

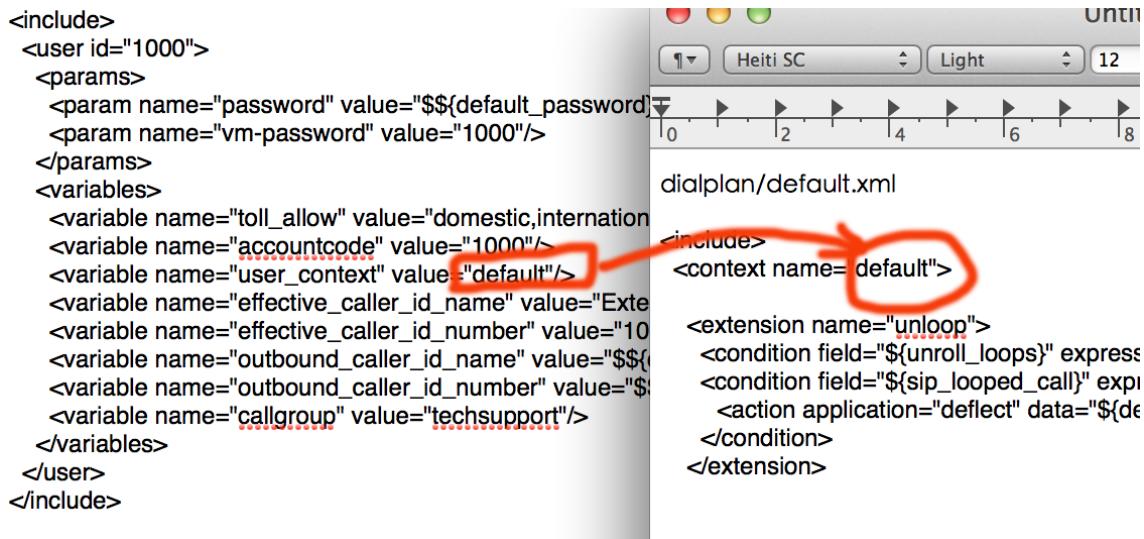


图 11.2: Context的对应关系

```

1 freeswitch@localhost> sofia_contact 1001@192.168.1.109
2
3 sofia/internal/sip:1001@192.168.1.109:51641

```

找到呼叫字符串后，FreeSWITCH又启动另外一个会话作为一个UAC给1001发送INVITE请求，如果1001摘机，回送200，FreeSWITCH再向1000回200，通话开始。

FreeSWITCH是一个B2BUA，上面的过程建立了一对通话，其中有两个Channel。

参照第四章，读者不妨跟踪SIP消息试一下，在控制台上打开跟踪的命令是：

```
1 sofia profile internal siptrace on
```

关掉

```
1 sofia profile internal siptrace off
```

FreeSWITCH的默认配置中，external.xml对应的profile是不鉴权的，因此凡是送到5080端口的INVITE都不需要鉴权。那么什么情况下或者怎样将INVITE送到5080呢？有以下两种情况：

第一种是，FreeSWITCH作为一个客户端，添加一个网关，该网关放到 sip_profiles/external/的XML文件中，它就会被包含到 sip_profiles/external.xml 中。它向其它的服务器注册时，其Contact地址就是 IP:5080，如果有来话，对方的服务器就会把INVITE送到它的5080端口，跟上面我们呼叫1001时把INVITE送到51641端口是一个道理。

第二种情况，大部分SIP UA允许你直接把INVITE送到任意一个端口，这一般用在中继方式对接的情况。

你也可以看到，在 external.xml 中 auth-calls=false。所以，当它收到INVITE时就不会进行鉴权了，而是直接对来话进行路由。

而路由需要一个Context，这个Context从哪里来呢？仔细看一下external.xml，里面有这么一句：

```
1 <param name="context" value="public"/>
```

所以路由就会查找public这个dialplan，而其中只定义了很少的extension，也就是说不经过认证的INVITE请求一般只能做比较少的事情，如只处理来话。

读者这里，读者也许会说，我怎么看到 internal.xml 里的context也是public呢？

是的，但是 internal 这个 profile 还有 auth-calls=true，而对于一个来话而言，如果经过了认证，就找到了一个用户，就会使用我们最前面所说的 user_context，而不是使用这个profile中的context。

那么，这个profile的context什么时候用呢？

我前面已经说过我们还没讲到ACL，不过，如果你正确配置了ACL的话，对有些IP地址发来的INVITE请求可以不鉴权，那么自然就不知道是哪个用户打来的电话，因此就无法找到对应的user_context，那就只能使用profile的context了。当然，这个context你可以改成任意值，只要你配置了相应的dialplan就行了。

值得一提的是，对于没有经过鉴权的呼叫，同样可以有主、被叫号码，只不过主叫号码是不可信的，对方可以指定显示任意号码。如果你讲究严谨的话，就需要多做些工作了。

11.12 在呼叫失败的情况下向主叫用户播放语音提示

有些时候，在呼叫失败的情况下我们希望给主叫用户一个有意义的语音提示。如，有时我们在打电话时会听到“您拨的电话正在通话中，请稍后再拨...”，或“电话无应答...”之类的提示，我们在 FreeSWITCH 里也可以这样做。

实现起来其实很简单，默认的配置在呼叫失败时会转到 voicemail (语音信箱)，我们只需要在这里修改，让他播放一个语音提示，然后再进入语音信箱（或直接挂断也行）。

找到 <extension name="Local_Extension">部分的最后几行

```
<action application="bridge" data="user/${dialed_extension}@${domain_name}" />
<action application="answer" />
<action application="sleep" data="1000" />
<action application="bridge"
      data="loopback/app=voicemail:default ${domain_name} ${dialed_extension}" />
```

其中，第一个 bridge 是说明去呼叫被叫号码，如果呼叫失败，则 dialplan 会继续往下走，依次是：

- 应答 (answer)
- “睡”一会 (sleep)
- 进入语音信箱 (voicemail)

OK, 我们只需要把最后一个bridge那行改成

```
<action application="playback" data="${originate_disposition}.wav"/>
```

重新打电话试一下吧，如果被叫忙，则 originate_disposition 变量的值就会是 USER_BUSY，用户没注册就是 USER_NOT_REGISTERED 之类的，你只需要保证相关目录下有相对应的声音文件即可（如果log中提示找不到声音文件的话试试自己录一个）。

当然，呼不通的原因可能有很多，你总不可能录上所有的声音文件是吧，有两种方法可以解决这个问题：

- 1) 使用一个 lua (或其它语言) 的脚本

```
<action application="lua" data="/tmp/xxx.lua"/>
```

在 lua 脚本中可以拿到这个 originate_disposition 变量，从而可以使用 if/then/else 之类的逻辑播放各种声音文件。这里就不多讲了。

- 2) 当然，如果你脚本也不想编辑的话，实际上 FreeSWITCH 的 dialplan 功能是非常强大的，你只需要将呼叫转到播放不同声音文件的 dialplan:

```
<action application="transfer" data="play-cause-${originate_disposition}"/>
```

然后创建如下 dialplan extension:

```
<extension name="Local_Extension_play-cause">
    <condition field="destination_number" expression="^play-cause-USER_BUSY$">
        <action application="playback" data="/tmp/sounds/user-busy.wav"/>
    </condition>
</extension>

<extension name="Local_Extension_play-cause">
    <condition field="destination_number" expression="^play-cause-USER_NOT_REGISTERED$">
```

```

<action application="playback" data="/tmp/sounds/user-not-registered.wav"/>
</condition>
</extension>

<extension name="Local_Extension_play-cause">
<condition field="destination_number" expression="^play-cause0(.*)$">
    <!-- for all other reasons, play this file -->
    <action application="log" data="WARNING hangup cause: $1"/>
    <action application="playback" data="/tmp/sounds/unknown-error.wav"/>
</condition>
</extension>

```

当然，能播放上面的声音文件还有一个前提，就是在第一个 bridge 前面要有以下两行：

```

<action application="set" data="hangup_after_bridge=true"/>
<action application="set" data="continue_on_fail=true"/>

```

其中，第一行的作用是：在一个成功的 *bridge* 之后，如果被叫用户挂断电话，我们也没有必要再向主叫用户播放提示音了，因此直接挂机。这一行也可以没有，那么你可以在后面的 originate_disposition 里如果发现值是“NORMAL_CLEARING”（正常挂机）的情况再决定是否播放相关语音。

第二行的作用是：如果呼叫失败（空号，拒接等），继续往下走，否则(值为 false 的情况)到这里就挂机了。该变量的值还可是各种用逗号分隔开的挂断原因，如，下面的配置表示只有遇到用户忙或无应答这两种情况才播放语音，其它的就直接挂机。

```
<action application="set" data="continue_on_fail=USER_BUSY,NO_ANSWER"/>
```

常见的挂断原因取值还有 NORMAL_TEMPORARY_FAILURE（普通临时失败）、TIMEOUT（超时）、NO_ROUTE_DESTINATION（无法路由/空号）等几种⁴。

11.13 在同一台服务器上启动多个 FreeSWITCH 实例

有时候，需要用到多个FreeSWITCH进行测试，或者需要在一台服务器上部署多个“不兼容”的系统。我们在这节探讨一下怎么做。

11.13.1 背景故事

几年前我还在Idapted工作的时候，由于需要连接Skype及Google Talk。就曾经做过这样的部署（如下图，附录中也有）。

⁴更详细的请参见：<http://wiki.freeswitch.org/wiki/Hangup.causes>

```

1           |--- PSTN gateways
2 /-----\   |--- FS-skype
3 | FS    |-----|--- FS-gtalk
4 \-----/   |--- FS-skype2
5           |--- more ...

```

当时主要的考虑是Skype(mod_skypiax, 后来改名为mod_skypopen)模块不太稳定, 所以我们把带Skype的FreeSWITCH启动到另一个实例, 这样就避免了由于Skype模块崩溃影响所有业务。后来, 我们在这上面又做了扩展, 即再启动一个带Skype的实例。对于主的FreeSWITCH而言, 他们就相当于两个Skype网关。平时可以负荷分担的工作, 其中一个崩溃, 另一个也可以正常工作, 起到了HA (High Availability, 高可用) 效果。

后来我们又启动了一个带mod_dingaling的FreeSWITCH实例, 用于跟Google Talk互通。除了当时mod_dingaling也有点稳定性问题外, 我们主要是为了让呼叫处理更方便, 如, 不管是呼叫什么用户, 呼叫字符串都是sofia/gateway/, 编程就方便了许多。

11.13.2 练习

闲话少叙, 我们今天虽不能复现当时的场景, 但基本概念是差不多的。我们今天的练习就是在同一机器上启动两个FreeSWITCH实例而互相不冲突, 甚至, 可以用各种方式互通。

我们都应该知道FreeSWITCH默认的配置文件在 /usr/local/freeswitch/conf。这里假设第一个实例已启动并正确运行。

首先, 我们要复制一份新的环境 (放到freeswitch2目录中, 以下的操作都在该目录中) :

```

1 mkdir /usr/local/freeswitch2
2 cd /usr/local/freeswitch2
3 cp -R /usr/local/freeswitch/conf .
4 mkdir log
5 mkdir db
6 ln -sf ./freeswitch/sounds .

```

其中第1行创建一个新目录, 第3行把旧的配置文件复制过来, 第4、5行分别创建log和db目录, 最后一行做个符号链接, 确保有正确的语音文件。

然后, 需要修改一些配置以防止端口冲突。第一个要修改的是conf/autoload_configs/event_socket.conf.xml, 把其中的8021改成另一个端口, 比方说9021。

修改 conf/vars.xml, 把其中的5060,5080也改成其它的, 如7060,7080。

默认的这样就行了, 当然如果你还加载了其它的模块, 注意要把可能引起冲突的资源都改一下。比如因为我用到 mod_erlang_event 模块, 我就需要改autoload_configs/erlang_event.conf.xml中的listen-port和nodename。

下面我们可以启动试试了:

```
1 cd /usr/local/freeswitch2/
2 /usr/local/freeswitch/bin/freeswitch -conf conf -log log -db db
```

以上命令分别用 `-conf`、`-log`、`-db` 指定新的环境目录。启动完成后将进入控制台。如果想使用 `fs_cli`, 则可以打开另外一个终端窗口, 连接 (还记得我们把端口改成9021了吧?) :

```
1 /usr/local/freeswitch/bin/fs_cli -P 9021
```

找个软电话注册到7060端口试试, 比如我用Xlite注册的地址就是 192.168.1.100:7060。

11.13.3 进阶

当然, 上面的两个FreeSWITCH实例都运行的同一份代码。有时候, 你还可能运行两个不同版本的FreeSWITCH。你可以在编译的时候指定一个不同的安装目录, 如:

```
1 ./configure --prefix=/opt/freeswitch
2 make && make install
```

这样就可以将FreeSWITCH安装到`/opt/freeswitch`目录中, 如果执行`/opt/freeswitch/bin/freeswitch`, 它就默认使用`/opt/freeswitch/conf`下面的配置文件, 我们也不需要再copy一份了。

当然, 如果需要两个实例同时运行, 你还是要改其中的一个的某些端口, 以避免冲突。改完后运行:

```
1 /opt/freeswitch/bin/freeswitch
```

好玩吧? 如果用于生产环境, 你可能也想把新的配置加到自启动文件里去, 这些就留给读者自己练习了。

当然, 如果你真的用于生产环境, 你还有更多的事情要做。比方说其中一项就是需要修改两个`switch.conf.xml`文件, 把其中的`rtp-start-port`及`rtp-end-port`改成不冲突的值。这两个参数控制FreeSWITCH的rtp端口池, 并发量小的情况下可能看不出来, 一旦并发量大了, 配置不好就有可能产生端口冲突。当然, 更进一步的说明超出了本节的范围。希望读者能多进行实践。

11.13.4 让两个实例相互通信?

我记得已经写过了, 看《多台FreeSWITCH服务器级联》一节。

11.13.5 小结

通过本节, 相信你对FreeSWITCH配置文件、运行方式、及 `fs_cli` 等工具又多了一层认识。当然, 即使你不需要启动多个实例, 相信也能从这一节学到一些东西。

第十二章 附录

12.1 FreeSWITCH背后的故事(译)

Anthony Minessale/文 Seven Du/译

本文由Anthony Minessale写于2007年5月。来自 www.freeswitch.org¹。翻译它是因为我觉得永远都不会过时... — 译者注

我开发FreeSWITCH已经近两年的时间了。在我们第一个发行版即将发布的黎明之际，我想花一点时间来与大家分享一下软件背后的故事，并透露一点即将到来的消息。本故事也将会登在 [OST Magazine](#)第一期上。呵，去下载一份吧，免费的！

写FreeSWITCH的想法诞生于2005年春的一次Asterisk开发者大会上。当时，我为Asterisk 1.2版贡献了大量的代码和新特性，并为其以后的发展提了好多思路。我们都认为在当时的Asterisk代码树中存在很严重的限制，并且面临着一些问题——修正一些问题时不仅会牺牲很多特性，而且还会花大量的时间。一种思路是建立新的代码分支，通过将大家已经熟悉的代码分开，新的分支就不会影响Asterisk用户的使用，从而能减轻好多开发的压力。问题是开发者都不希望将精力分散到同一份代码的两个版本上，因为那样他们就不得不将BUG修正和代码修改在两个分支间拷来拷去。我的建议是：在一个单独的代码库上从头创建一个Asterisk 2.0的分支，等一切就绪后再对外发布。我想那个想法确实打动了一些开发人员，但现实是，由于讨论的时间过长，最终大家都失去了兴趣。不过，我没有。

很明显，我是唯一一个认真考虑这一问题的。接下来我用了几天的时间在做一个白日梦——如何设计一个新的电话系统。之后，我再也无法抑制，便创建了一个新的目录开始从头写choir.c。是的，Choir是我为该工程选的第一个名字。我希望不同的通信部件能够步调一致地协同工作，就像教堂的唱诗班那样优雅和谐（perfect harmony）。我又用了5天时间把我想到的点子都组织到几个文件里。最初的努力并不能装配成一个电话交换机。我知道，我需要创建一个稳定的核心，并应该能跨平台。所以最简单的是使用Apache APR库来搭建一个基本的子系统，让它能够动态装载共享模块并悠闲地停在屏幕上等待shutdown命令。实现这些代码用了另外6个月的时间。

在那5天里，我知道了写一个达到那种要求的可伸缩性程序并不是一件简单的事情。我最初的目标是捡起那些Asterisk丢掉的东西并加以改善。但随着思考的深入，我越来越觉得，实际上我想

¹<http://www.freeswitch.org/node/60>

改进的是最基础的设计和功能。最终我得出结论，Asterisk不能实现一些我期望的功能是因为它不是我所需要的软件。也就在那时，我知道我不是要编写另一个PBX，而是要开发一个完全不同级别的应用程序。我用了后续的几个月时间组织了第一届ClueCon年会来讨论如何合理的设计Choir。我希望在继续写任何代码之前能确信我有正确的计划。从这一点上说，我仍在做相当的一部分Asterisk开发，并且我也在我写的一些第三方模块中来实验我的想法。另外，我也让我的同事们花了相当多的时间来争论在电话领域里如何“正确”地做事情，这也算是一种前瞻性吧。

在那年的ClueCon会议上，我有幸遇到一些在电话领域里很有影响力的开发者，并把很多灵感带回了家。同时，Asterisk阵营中的关系开始有些紧张，因为有几个开发者也打算建立新的分支。新的项目称为OpenPBX（现在叫Call Weaver），从某种意义上讲它引起了Asterisk社区的一次严重地分裂。最初，OpenPBX把我的许多第三方Asterisk模块加入到他们的新代码中，并就如何增强稳定性方面咨询我的建议。其中有一点，甚至他们还来看我是如何在我的新项目中实现的。可能我们能再打起精神来重写那些老的计划，但最终没有实现。不过，我想，最意义深远的一刻是——当有人问我：“多长时间以后它才能打电话呢？”我不知道，所以我决定搞清楚。简短的回答是一星期。

与我想达到的目标相比，能打电话只是一个小小的胜利。我还有很多要做的事情。这一点不起眼的业绩得不到太多关注。好消息是，这一项目最终上路了。我深居简出，用了三个月时间试图能做出点能吸引公众眼球的东西。那段时间，项目的名字曾改为Pandora并最终成为FreeSWITCH。2006年1月我们公开的SVN仓库和一个邮件列表上线了。那时仅有几个模块和一个很短的特性列表。但我们有了一个可以工作的内核，它能够在包括Mac OS X, Linux和BSD的几个UNIX变种上编译和运行，并能在Microsoft Windows上以一个控制台程序运行。

随着时间的推移，我们也越加有动力。有时也停下来犯几个错误，然后继续前进，写几个新的模块。在试验了四个不同的SIP终点模块后，我们决定使用Sofia SIP。也曾试验过5个不同的RTP协议栈，最终决定还是我们自己写。同时我也开发了mod_dingaling来做与Google Talk的接口，以及一个多特性的会议桥。在第一年里，我主要关注如何在使核心尽量稳定的基础上，提供几个外部接口以便于其它模块的开发。如用于IVR的嵌入式的javascript，一个XML-RPC接口和一个用于远程控制和事件监控的基于TCP的Socket的接口。

第二年的ClueCon大会，那一天仿佛就是我白日梦醒来的日子。我第一次向我的同行们演示了FreeSWITCH。近九个月后，在距离第一个想法两年之际，第三届ClueCon一个月前，我们达到了开发的BETA阶段——FreeSWITCH 1.0发布在际了。我们也吸引了一些勇敢的开发者一道。他们已经把FreeSWITCH用于生产系统，并给我们提供了保证我们第一个发布版本成功的很重要的反馈。

在发布之前，最后一点工作是我新写的一个叫做OpenZAP的开源的TDM抽象库。使用BSD协议的mod_openzap模块将取代当时特定于Sangoma的mod_wanpipe，并提供Sangoma及其它几种TDM硬件（只需要开发对应的模块）支持。OpenZAP也将为模拟和ISDN信令提供一个简单的接口。OpenZAP的指导思想是——应用程序能使用同样的API去控制任何它所支持的TDM硬件。它提供一种方式能将所有不同的特性规范化。如果一种卡缺少某种特性，那么它就能以软件的形式实现，不管是在OpenZAP库中还是在与生产商硬件API通信的接口程序中。

我们在如此短的时间内做这么多事听起来好像是不可能的，但我想，最终，还是像格言里说的：“需要是发明之母。”接下来的路还很长。但我想就此机会感谢所有曾经帮助我们走了这么远的人。下面列表中是所有在我们AUTHORS文件中的人：

- Anthony Minessale II (就是我!)
- Michael Jerris (我们极具价值的编译专家和跨平台专家 cross-platformologist, [呵, 这个词是我创造的])
- Brian K. West (我们挚爱的Mac权威, 没有他的帮助, 我们将寸步难行)
- Joshua Colp (帮助我们做了第一个SIP模块, 虽然现在我们已经不用了)
- Michal “cypromis” Bielicki (他从第一天就加入进来了, 感谢信任!)
- James Martelletti (把mono集成进了FreeSWITCH.)
- Johny Kadarisman (帮我们弄好了python模块)
- Yossi Neiman (写了mod_cdr收集通话详单)
- Stefan Knoblich (在我们的SIP之旅上帮助甚多)
- Justin Unger (找到很多BUG)
- Paul D. Tinsley (SIP presence以及其它好的建议)
- Ken Rice (为什么有这个名字? 它给了我们做了很多测试和补丁)
- Neal Horman (在会议模块上有巨大贡献)
- Michael Murdock (我们CopperCom的朋友, 有大量反馈和补丁)
- Matt Klein (大量SIP帮助, 将帮我们确保FreeSWITCH运行于FreeBSD.)
- Justin Cassidy (幕后工作者, 确保一切正常)
- Bret McDanel (敢吃螃蟹的人, 试验了绝大多数的功能, 最早发现了很多隐藏的BUG, 我指的是复活节彩蛋!)

12.2 FreeSWITCH 与 Asterisk 比较²

Anthony Minessale/文 Seven Du/译

VoIP通信, 与传统的电话技术相比, 不仅仅在于绝对的资费优势, 更重要的是很容易地通过开发相应的软件, 使其与企业的业务逻辑紧密集成。Asterisk作为开源VoIP软件的代表, 以其强大的功能及相对低廉的建设成本, 受到了全世界开发者的青睐。而FreeSWITCH作为VoIP领域的新秀, 在性能、稳定性及可伸缩性等方面则更胜一筹。本文原文在<http://www.freeswitch.org/node/117>, 发表于2008年4月, 相对日新月异的技术来讲, 似乎有点过时。但本文作为FreeSWITCH背后的故事, 仍很有翻译的必要。因此, 本人不揣鄙陋, 希望与大家共读此文, 请不吝批评指正。—译者注

²FreeSWITCH vs Asterisk

FreeSWITCH 与 Asterisk 两者有何不同？为什么又重新开发一个新的应用程序呢？最近，我听到很多这样的疑问。为此，我想对所有在该问题上有疑问的电话专家和爱好者们解释一下。我曾有大约三年的时间用在开发 Asterisk 上，并最终成为了 FreeSWITCH 的作者。因此，我对两者都有相当丰富的经验。首先，我想先讲一点历史以及我在 Asterisk 上的经验；然后，再来解释我开发FreeSWITCH的动机以及我是如何以另一种方式实现的。

我从2003年开始接触 Asterisk，当时它还不到1.0版。那时对我来讲，VoIP还是很新的东西。我下载并安装了它，几分钟后，从插在我电脑后面的电话机里传出了电话拨号音，这令我非常兴奋。接下来，我花了几天的时间研究拨号计划，绞尽脑汁的想能否能在连接到我的Linux PC上的电话上实现一些好玩的东西。由于做过许多Web开发，因此积累了好多新鲜的点子，比如说根据来电显示号码与客户电话号码的对应关系来猜想他们为什么事情打电话等。我也想根据模式匹配来做我的拨号计划，并着手编写我的第一个模块。最初，我做的第一个模块是app_perl，现在叫做res_perl，当时曾用它在Asterisk中嵌入了一个Perl5的解释器。现在我已经把它从我的系统中去掉了。

后来我开始开发一个Asterisk驱动的系统架构，用于管理我们的呼入电话队列。我用app_queue和现在叫做AMI（大写字母总是看起来比较酷）的管理接口开发了一个原型。它确实非常强大。你可以从一个T1线路的PSTN号码呼入，并进入一个呼叫队列，坐席代表也呼入该队列，从而可以对客户进行服务。非常酷！我一边想一边看着我的可爱的Web页显示着所有的队列以及他们的登录情况。并且它还能周期性的自动刷新。令人奇怪的是，有一次我浏览器一角上的小图标在过了好长时间后仍在旋转。那是我第一次听说一个词，一个令我永远无法忘记的词 — 死锁。

那是第一次，但决不是最后一次。那一天，我几乎学到了所有关于GNU调试器的东西，而那只是许多问题的开始。队列程序的死锁，管理器的死锁。控制台的死锁开始还比较少，后来却成了一个永无休止的过程。现在，我非常熟悉“段错误(Segmentation Fault)”这个词，它真是一个计算机开发者的玩笑。经过一年的辛勤排错，我发现我已出乎意料的非常精通C语言并且有绝地战士般的调试技巧。我有了一个分布于七台服务器、运行于DS3 TDM信道的服务平台。与此同时，我也为这一项目贡献了大量的代码，其中有好多是我具有明确版权的完整文件³。

到了2005年，我已经俨然成了非常有名的Asterisk开发者。他们甚至在CREDITS文件以及《Asterisk, 电话未来之路》这本书中感谢我。在Asterisk代码树中我不仅有大量的程序，而且还有一些他们不需要或者不想要的代码，我把它们收集到了我的网站上。(至今仍在 <http://www.freeswitch.org/node/50>)

Asterisk 使用模块化的设计方式。一个中央核心调入称为模块的共享目标文件以扩展功能。模块用于实现特定的协议（如SIP）、程序（如个性化的IVR）和其它外部接口（如管理接口）等。Asterisk的核心是多线程的，但它非常保守。仅仅用于初始化的信道以及执行一个程序的信道才有线程。任何呼叫的B端都与A端都处于同一线程。当某些事件发生时（如一次转移呼叫必须首先转移到一个称作伪信道的线程模式），该操作把一个信道所有内部数据从一个动态内存对象中分离出来，放入另一个信道中。它的实现在代码注释中被注明是“肮脏的”⁴。反向操作也是如此，当销毁一个信道时，需要先克隆一个新信道，才能挂断原信道。同时也需要修改CDR的结构

³<http://www.cluecon.com/anthm.html>

⁴/* XXX This is a seriously wacked out operation. We're essentially putting the guts of the clone channel into the original channel. Start by killing off the original channel's backend. I'm not sure we're going to keep this function, because while the features are nice, the cost is very high in terms of pure nastiness. XXX */

以避免将它视为一个新的呼叫。因此，对于一个呼叫，在呼叫转移时经常会看到3或4个信道同时存在。

这种操作成了从另一个线程中取出一个信道事实上的方法，同时它也正是开发者许许多多头痛的源头。这种不确定的线程模式是我决定着手重写这一应用程序的原因之一。

Asterisk使用线性链表管理活动的信道。链表通过一种结构体将一系列动态内存串在一起，这种结构体本身就是链表中的一个成员，并有一个指针指向它自己，以使它能链接无限的对象并能随时访问它们。这确实是一项非常有用的技术，但是，在多线程应用中它非常难于管理。在线程中必须使用一个信号量（互斥体，一种类似交通灯的东西）来确保在同一时刻只有一个线程可以对链表进行写操作，否则当一个线程遍历链表时，另一个线程可能会将元素移出。甚至还有比这更严重的问题——当一个线程正在销毁或监听一个信道的同时，若有另外一个线程访问该链表时，会出现“段错误”。“段错误”在程序里是一种非常严重的错误，它会造成进程立即终止，这就意味着在绝大多数情况下会中断所有通话。我们所有人都看到过“防止初始死锁”⁵这样一个不太为人所知的信息，它试图锁定一个信道，在10次不成功之后，就会继续往下执行。

管理接口（或AMI）有一个概念，它将用于连接客户端的套接字(socket)传给程序，从而使你的模块可以直接访问它。或者说，更重要的是你可以写入任何你想写入的东西，只要你所写入的东西符合Manager Events所规定的格式（协议）。但遗憾的是，这种格式没有很好的结构，因而很难解析。

Asterisk的核心与某些模块有密切的联系。由于核心使用了一些模块中的二进制代码，当它所依赖的某个模块出现问题，Asterisk就根本无法启动。如果你想打一个电话，至少在Asterisk 1.2中，除使用app_dial和res_features外你别无选择，这是因为建立一个呼叫的代码和逻辑实际上是在app_dial中，而不是在核心里。同时，桥接语音的顶层函数实际上包含在res_features中。

Asterisk的API没有保护，大多数的函数和数据结构都是公有的，极易导致误用或被绕过。其核心非常混乱，它假设每个信道都必须有一个文件描述符，尽管实际上某些情况下并不需要。许多看起来是一模一样的操作，却使用不同的算法和截然不同的方式来实现，这种重复在代码中随处可见。

这仅仅是我在Asterisk中遇到的最多的问题一个简要的概括。作为一个程序员，我贡献了大量的时间，并贡献了我的服务器来作为CVS代码仓库和Bug跟踪管理服务器。我曾负责组织每周电话会议来计划下一步的发展，并试图解决我在上面提到过的问题。问题是，当你对着长长的问题列表，思考着需要花多少时间和精力来删除或重写多少代码时，解决这些问题的动力就渐渐的没有了。值得一提的是，没有几个人同意我的提议并愿意同我一道做一个2.0的分支来重写这些代码。所以在2005年夏天我决定自己来。

在开始写FreeSWITCH时，我主要专注于一个核心系统，它包含所有的通用函数，即受到保护又能提供给高层的应用。像Asterisk一样，我从Apache Web服务器上得到很多启发，并选择了一种模块化的设计。第一天，我做的最基本的工作就是让每一个信道有自己的线程，而不管它要做什么。该线程会通过一个状态机与核心交互。这种设计能保证每一个信道都有同样的、可预测的路径和状态钩子，同时可以通过覆盖向系统增加重要的功能。这一点也类似其它面向对象的语言中的类继承。

做到这点其实不容易，容我慢慢讲。在开发FreeSWITCH的过程中我也遇到了段错误和死锁（在前面遇到的多，后来就少了）。但是，我从核心开始做起，并从中走了出来。由于所有信道

⁵ Avoiding initial deadlock

都有它们自己的线程，有时候你需要与它们进行交互。我通过使用一个读、写锁，使得可以从一个散列表（哈希）中查找信道而不必遍历一个线性链表，并且能绝对保证当一个外部线程引用到它时，一个信道无法被访问也不能消失。这就保证了它的稳定，也不需要像Asterisk中“Channel Masquerades”之类的东西了。

FreeSWITCH核心提供的的大多数函数和对象都是有保护的，这通过强制它们按照设计的方式运行来实现。任何可扩展的或者由一个模块来提供方法或函数都有一个特定的接口，从而避免了核心对模块的依赖性。

整个系统采用清晰分层的结构，最核心的函数在最底层，其它函数分布在各层并随着层数和功能的增加而逐渐减少。

例如，我们可以写一个大的函数，打开一个任意格式的声音文件向一个信道中播放声音。而其上层的API只需用一个简单的函数向一个信道中播放文件，这样就可以将其作为一个精减的应用接口函数扩展到拨号计划模块。因此，你可以从你的拨号计划中，也可以在你个性化的C程序中执行同样的playback函数，甚至你也可以自己写一个模块，手工打开文件，并使用模块的文件格式类服务而无需关注它的代码。

FreeSWITCH由几个模块接口组成，列表如下：

拨号计划(Dialplan): 实现呼叫状态，获取呼叫数据并进行路由。

终点(Endpoint): 为不同协议实现的接口，如SIP, TDM等。

自动语音识别/文本语音转换(ASR/TTS): 语音识别及合成。

目录服务(Directory): LDAP类型的数据库查询。

事件(Events): 模块可以触发核心事件，也可以注册自己的个性事件。这些事件可以在以后由事件消费者解析。

事件句柄(Event handlers): 远程访问事件和CDR。

格式(FORMATS): 文件模式如wav。

日志(Loggers): 控制台或文件日志。

语言(Languages): 嵌入式语言，如Python和JavaScript。

语音(Say): 从声音文件中组织话语的特定的语言模块。

计时器(Timers): 可靠的计时器，用于间隔计时。

应用(Applications): 可以在一次呼叫中执行的程序，如语音信箱(Voicemail)。

FSAPI(FreeSWITCH 应用程序接口) 命令行程序，XML RPC函数，CGI类型的函数，带输入输出原型的拨号计划函数变量。

XML 到核心XML的钩子可用于实时地查询和创建基于XML的CDR。

所有的FreeSWITCH模块都协同工作并仅仅通过核心API或内部事件相互通信。我们非常小心地实现它以保证它能正常工作，并避免其它外部模块引起不期望的问题。

FreeSWITCH的事件系统用于记录尽可能多的信息。在设计时，我假设大多数的用户会通过一个个性化的模块远程接入FreeSWITCH来收集数据。所以，在FreeSWITCH中发生的每一个重

要事情都会触发一个事件。事件的格式非常类似于一个电子邮件，它具有一个事件头和一个事件主体。事件可被序列化为一个标准的Text格式或XML格式。任何数量的模块均可以连接到事件系统上接收在线状态，呼叫状态及失败等事件。事件树内部的mod_event_socket可提供一个TCP连接，事件可以通过它被消费或记入日志。另外，还可以通过此接口发送呼叫控制命令及双向的音频流。该套接字可以通过一个正在进行的呼叫进行向外连接(Outbound)或从一个远程机器进行向内 (Inbound)连接。

FreeSWITCH中另一个重要的概念是中心化的XML注册表。当FreeSWITCH装载时，它打开一个最高层的XML文件，并将其送入一个预处理器。预处理器可以解析特殊的指令来包含其它小的XML文件以及设置全局变量等。在此处设置的全局变量可以在后续的配置文件中引用。

如，你可以这样用预处理指令设置全局变量：

```
1 <X-PRE-PROCESS cmd="set" data="moh_uri=local_stream://moh"/>
```

现在，在文件中的下一行开始你就可以使用 `$(moh_uri)`，它将在后续的输出中被替换为 `local_stream://moh`。处理完成后XML注册表将装入内存，以供其它模块及核心访问。它有以下几个重要部分：

- 配置文件：配置数据用于控制程序的行为。
- 拨号计划：一个拨号计划的XML表示可以用于 mod_dialplan_xml，用以路由呼叫和执行程序。
- 短语：可标记的IVR分词是一些可以“说”多种语言的宏。
- 目录：域及用户的集合，用于注册及账户管理。

通过使用XML钩子模块，你可以绑定你的模块来实时地查询XML注册表，收集必要的信息，以及返回到呼叫者的静态文件中。这样你可以像一个WEB浏览器和一个CGI程序一样，通过同一个模型来控制动态的SIP注册，动态语音邮件及动态配置集群。

通过使用嵌入式语言，如Javascript, Java, Python和Perl等，可以使用一个简单的高级接口来控制底层的应用。

FreeSWITCH工程的第一步是建立一个稳定的核心，在其上可以建立可扩展性的应用。我很高兴的告诉大家在2008年5月26日将完成FreeSWITCH 1.0 PHOENIX版。有两位敢吃螃蟹的人已经把还没到1.0版的FreeSWITCH 用于他们的生产系统。根据他们的使用情况来看，我们在同样的配置下能提供Asterisk 10倍的性能。

我希望这些解释能足够概括FreeSWITCH和Asterisk的不同之处以及我为何决定开始FreeSWITCH项目。我将永远是一个Asterisk开发者，因为我已深深的投入进去。并且，我也希望他们在以后的Asterisk开发方面有新的突破。我甚至还收集了很多过去曾经以为已经丢失的代码，放到我个人的网站上供大家使用，也算是作为我对引导我进入电话领域的这一工程的感激和美好祝愿吧。

Asterisk是一个开源的PBX，而FreeSWITCH则是一个开源的软交换机。与其它伟大的软件如 Call Weaver、Bayonne、sipX、OpenSER以及更多其它开源电话程序相比，两者还有很大发

展空间。我每年都期望能参加在芝加哥召开的 ClueCon 大会，并向其它开发者展示和交流这些项目(<http://www.cluecon.com>)。

我们所有人都可以相互激发和鼓励以推进电话系统的发展。你可以问的最重要的问题是：“它是完成该功能的最合适的工具吗？”

12.3 FreeSWITCH 中文FAQ

这是 [FreeSWITCH 官方 FAQ](#) 的中文翻译。截止日期: 2010-05-01.

12.3.1 基本问题

啊，你们提供了一个如此强大的系统，那你们有什么心愿吗？

这里有每个开发者的 Wishlist:

- Anthony Minessale (anthm) - <http://www.amazon.com/gp/registry/wishlist/1NQC79YV4RB83>
- Mike Jerris (MikeJ) - <http://www.amazon.com/gp/registry/wishlist/1ELF2W9FAIN2N>
- Brian West (bkw_) - <http://www.amazon.com/gp/registry/wishlist/1BWDJUX5LYQE0>
- Raymond Chandler ([intra]lanman) - <http://www.amazon.com/gp/registry/wishlist/27XDISBBI4NOU>

在 sofia 呼叫字符串中，% 和 @ 有何不同？

这很简单，你可以有多种选择。如果 domain 是一个 profile 的别名，那你你可以用 sofia/domain_name/user; 如果 domain 不是别名，那可你可以使用 sofia/profile_name/user%domain; 但如果你想呼叫一个远端的 SIP URI，并且对方不需要认证，那么你可以直接使用 sofia/profile_name/remoteuser@remoteip

`${var}` 与 `$$ {var}` 有何不同？

`${var}` 回在每次执行到 dialplan 时进行扩展，而 `$$ {var}` 则是在模块加载或 reloadxml 时一次性扩展的。更详细的信息见 conf/vars.xml 中的注释。

set 和 export 程序有何区别?

set 在当前 channel 上设置变量，而 export 在 (a-leg 和 b-leg) 两个 channel 上都设置。当然，你也可以只在 b-leg 上设置，如: ##### PBX 与 软交换的区别是什么？只是语义问题吗？

一个 PBX 通常用于公司内部提供小型的语音信箱、分机、电话会议等服务。它主要关注于不同的分机间能互相通信。

而一个软交换是一个连接多种网络的设备，通常可以路由不同协议的电话，把电话从一个终端路由到另一个终端（如一个PBX设置）。当然 FreeSWITCH 也可以做为一个 PBX 使用，但它的功能远不止于此。FreeSWITCH 可以装载很多不同的模块，运行起来相当于一个包含好多 PBX 的集群。通常的 PBX 都将全部 PBX 功能置于单一的核心中，这使用它们在想用作软交换时比 FreeSWITCH 要困难地多。

FreeSWITCH™ 是用什么语言写的？

很多种语言。核心是用 C 写的。大部分的模块是 C 和 C++。某些模块则使用了其它一些语言，包括但不限于 JavaScript/ECMAScript、Lua、Perl、Python、Ruby、Java 和 .NET。

你怎么评价你已前做过的 Asterisk 开发？

那些工作并没有白费。我有时还在使用它，甚至有时我还提供这方面的咨询服务。我花了很多年贡献代码，我也为 Asterisk 开发了许多第三方的模块，现在在我的 [asterisk stuff](#) 页面还可以找到。我只是简单地认为 FreeSWITCH 是电话的未来。

FreeSWITCH 能做什么？

FreeSWITCH 终点模块实现了 SIP, IAX2, H.323 (with OPAL), Skype, Jingle (Google Talk), 声卡（通过 mod_portaudio）以及 TDM 语音卡（Sangoma 或 Zaptel 兼容卡）。你可以把它用做一个 VoIP 通信服务器，协议转换网关，以及执行由 JavaScript、Perl、Lua 或 C# 写的 IVR 语音菜单脚本。它有一个事件引擎可以通知你任何正在发生的事件，还有 XML RPC 接口，你可以用它通过 HTTP 与 系统核心进行交互。

有一个用于读取数据的抽象层，它支持 XML、外部的 HTTP 服务器、INI 文件，以及目录查询（如 LDAP）等。拨号计划通过 XML 与 Perl 兼容的正则表达式的结合对电话进行路由。

什么？你刚才说它还可以和 Google Talk 通信？

是的，2006 年 3 月我自己写了一个 XMPP 电话信令库与 Google 的 Google Talk 进行通信，你可以用一个 Jabber 账号同时与任意多路 Google Talk 客户端通话，当然也可以像 SIP 或 H.323 那样实现 IVR。如果通信的两端都是 FreeSWITCH，你还可以绕过 NAT 发送扩展的数据，如 Caller ID 和 DNIS。这里有一个关于 Google Talk 与 SIP 电话通信的教程：http://www.alijawad.org/cms/index.php?option=com_content&task=view&id=21&Itemid=2。

有人将 FreeSWITCH 用于生产环境吗?

是的。

它同时支持多少路通话? 有基准测试吗?

这依赖于你的程序以及配置。你需要用你的程序进行压力测试来获取你的极限。你所能做的完全依赖于你的需求。请不要在 FreeSWITCH 邮件列表中问这样的问题, 因为你总是会得到相同的官方回答: “我们只对每一个特定的 FreeSWITCH 部署进行基准测试, 因为不同的部署方式会得到不同的值。如果你需要, 你可以获取该项目的商业支持。我们曾经历过回答这种问题的教训, 所以我们的策略是不在公共的论坛上对该问题发表任何意见”。

你们做该项目多久了?

最早的发行版本是在 2006 年, 只能做少的功能。实际上我从 2005 年 10 月起就利用业余时间开发。

什么时候才有第一个真正的发行版?

第一个真正的发行版是版本 1.0.0, 发行于 2008 年 5 月 26 日, 星期一。后来, 有来其它的 5 个发行版。现在最新的版本是 1.0.6, 发布于 2010 年 6 月。所有发行版可以在我们的[文件服务器](#)上找到。

什么是 ClueCon?

[ClueCon](#) 是电话系统开发者的年度盛会, 在芝加哥举行。会上会有来自不同的 VoIP 项目的领袖及其它开发者展示、讨论及交流意见。它为期三天, 是一个绝好的机会可以在白天交流技术, 晚上则享受芝加哥的美景。

什么电话能在 FreeSWITCH 下工作?

见 Interop List: http://wiki.freeswitch.org/wiki/Interop_List

FreeSWITCH 一定要以 root 用户运行吗?

不。

12.3.2 获取帮助

有帮助文档吗？

是的，我们的 wiki 上有超过 500 页的文档：<http://wiki.freeswitch.org/wiki/Documentation>。也有中文的文档在：<http://www.freeswitch.org.cn/document>。

你们支持 IRC 吗？ Q: Do you guys support IRC?

是的。志愿者和 FreeSWITCH 专家聚集在 irc.freenode.net 上的 #freeswitch 频道。你可以使用很多 ICR 客户端，如 mIRC for Windows、Limechat for Mac/OSX、Irssi 或 XChat for Unix 类的系统、ChatZilla for Mozilla 浏览器或任何其它标准的 IRC 程序。在里面，我们主要说英语，不过，我们确实有一个支持多种语言的自动的翻译服务。

是否有一个电话会议系统我可以参与有关 FreeSWITCH 的讨论呢？

是的。我们支持多种方式呼叫我们。以下的方式都会到同一个地方： SIP: 888@conference.freeswitch.org H.323: 888@conference.freeswitch.org Google Talk/Jingle: freeswitch@gmail.com Google Talk/Jingle: 888@jabber.asterlink.com IAX: guest@conference.freeswitch.org/888

是否有邮件列表？

当然，请到 <http://lists.freeswitch.org> 登记。

12.3.3 调试与排错

有没有关于排错与汇报 BUG 的指南？我该从哪里开始呢？

从这里开始，Reporting Bugs - http://wiki.freeswitch.org/wiki/Reporting_Bugs，它会回答你很多问题。

当我从 FreeSWITCH 呼叫 Snome 电话时，在控制台上看到“a=crypto in RTP/AVP, refer to RFC 3711”，怎么办？

您需要到 Snome 的配置界面，identity->rtp，并把 RTP/SAVP 设成 optional。

我的 FreeSWITCH 不影响任何 SIP 请求，我也用 tcpdump 检查了，发送端的正常的，但 FreeSWITCH 就是没有反应，怎么回事呢？

一般来说是你的防墙惹得祸，在 tcpdump 中能看到防火墙之外的数据包，你最好关掉防火墙试试，如“service iptables stop”或“/etc/init.d/iptables stop”。

我刚装好了 FreeSWITCH，但在启动的时候显示错误：SQL ERR[no such table: <table_name>

通常这条信息之后会有一条“Auto Generating Table!”的信息。那说明这是正常的。因为你是第一次使用，当 FreeSWITCH 找不到 sqlite 数据库或表时，它会自己创建。只要以后重启 FreeSWITCH 时不再出此错误，就没事。

show channels、conference list、以及其它控制台命令什么也显示不出来。

可能是核心数据库的结构乱了，这些命令都是从数据库中获取数据，删掉所有数据库（在Linux/Unix上用 rm -rf /usr/local/freeswitch/db/*）重启 FreeSWITCH 试试看。

我在 Win32 上装了 FreeSWITCH 但是不能启动，我应该检查什么？

你需要 msver80d.dll，它应该在你系统 system 或 system32 目录中已经存在的。如果你找不到这个 dll，那么你可以试试安装 Microsoft Visual C++ 2005 Redistributable 包。

如何调试 SIP？

参见：http://wiki.freeswitch.org/wiki/Debugging_Freeswitch 及 <http://wiki.freeswitch.org/wiki/Sofia>。

如何带 debug 符号编译 FreeSWITCH？

```
1 export CFLAGS="-g -ggdb"
2 export MOD_CFLAGS="-g -ggdb"
3 ./configure
4 make megaclean (如果所有它依赖的库都需要 debug 的话，或者)
5 make sure (如果只需要 FreeSWITCH)
```

我收到 “Invalid Application <name>” 是什么意思？

该错误最可能的原因是你没有加载正确的模块。

mod_spidermonkey_odbc 出错: ::Error SQLConnect=-1 errno=0 [unixODBC][Driver Manager]Data source name not found, and no default driver specified

确认你的 ODBC 驱动 (odbc.ini) 及数据源 (odbinst.ini) 在 /usr/local/freeswitch/etc 目录下, (这些文件通常在 /etc 目录, 做个符号链接就行)。参见: http://wiki.freeswitch.org/wiki/Mod_spidermonkey_odbc#General_Configuration

ICMP error 是什么错误?

由于某种原因, 你的连接请求被拒绝了, 详见http://wiki.freeswitch.org/wiki/Connection_Refused。

在 Ubuntu 64-bit (gutsy/intrepid) 上启动时出现: segmentation fault

ncurses 库有一个问题。解决的办法是: apt-get install libncurses5-dev 重新编译 libedit: cd libs/libedit make clean sh configure.gnu make cd ../../.. make clean make install

12.3.4 后台运行

我如何让 FreeSWITCH 运行在后台?

```
1 freeswitch -nc
```

当 FreeSWITCH 运行在后台时, 是否有类似 telnet 的客户端能连上去呢?

是的。只要 mod_event_socket 模块已加载, 你就可以使用 fs_cli 连上去。它在 FreeSWITCH 的 bin 目录。

FreeSWITCH 运行在后台时, 我如何停止它呢?

使用命令: freeswitch -stop 或在 fs_cli 中, 运行 fs_cli> fsctl shutdown

如何让 FreeSWITCH 以更高的优先级运行?

```
1 freeswitch -hp
```

如何将 FreeSWITCH 注册为一个 Win32 服务?

在你安装 FreeSWITCH 的路径中，执行 freeswitch -install 或者，删除该服务 freeswitch -uninstall

现在，该服务安装在“网络服务”项目中，在某些机器上，该项目可能没有足够的权限来运行 FreeSWITCH。在这种情况下，你需要修改它所属的用户。双击服务项目，到“登录”标签，将其修改为一个合适的用户，如“本地系统账户”或你建立的新账户。

你可以在命令行模式下启动和停止 FreeSWITCH:

```
1 net start freeswitch  
2 net stop freeswitch
```

如果使用“freeswitch -install”建立的启动项目不能启动，试试其它的办法，如：<http://sw4me.com/wiki/Winserv>。下载 winserv 并放到某一位置，如“C:\Program Files\winserv”。然后可以使用以下命令安装服务（人工换行）：

```
1 C:\Program Files\winserv\winserv.exe" install FreeSWITCH  
2 "C:\Program Files\FreeSWITCH\freeswitch.exe" -nc
```

如何在一台服务器上运行多个 FreeSWITCH 实例?

参见：http://wiki.freeswitch.org/wiki/Advanced_configuration#Multiple_FreeSWITCH_instances_on_one_box

12.3.5 硬件兼容性

它是否能运行在 Amazon Elastic Cloud 上?

是的，见：http://wiki.freeswitch.org/wiki/Amazon_EC2。

它能运行在 Xen 虚拟机里吗?

是的，EC2 就是用的 Xen。

它能运行在没有 MMU的机器上吗？比方说 Blackfin ?

现在还不能，可能以后也不能。没有MMU，就不能运行当下流行的操作系统，如Linux等。当然 Blackfin 对 ucLinux 支持得很好，但是 ucLinux 是一个精简的 Linux，它被设计于在没有 MMU 的有限的环境下也能运行。

有别的电话软件已经移植到了基于 Blackfin 的机器上，如 IP04，<http://www.rowetel.com>。如果程序清晰和简单，那么它将会非常好。但是，用户必须非常小心地运行那些不会产生内存碎片的程序。否则的话，就需要经常的进行重启。现在，FreeSWITCH还没有为如此受限的环境开发这么一个版本，所以，也没有人在做往 ucLinux 平台的移植。

12.3.6 编译

我是否需要下载所有外部的程序库（libs）？

不需要。make install 脚本会根据你选择编译的模块自动下载它所依赖的库。

我不想安装到 /usr/local/freeswitch/, 如何更改安装路径？

```
1 ./configure --prefix=/your/install/dir
```

我如何选择编译哪些模块？

修改源代码目录中的 modules.conf，把你想要编译的模块前面的 # 去掉，把不想编译的模块前面加个 #。在 Windows 系统上你需要使用 Visual Studio 提供的方法在 configuration manager 中修改模块依赖关系。

如何使用 Microsoft Visual C++ 2008 Express Edition 编译 FreeSWITCH？

一定要按照 Microsoft 的 install instructions 要求去做。Visual C++ 页面在 <http://www.microsoft.com/express/vc/>。当 VC++ 2008 Express 安装完成后，打开 solution 文件 Freeswitch.2008.express.sln（如果你看到很多关于不支持的目录的错误，很可能是你打开了 Freeswitch.2008.sln，那个文件是 MS Visual C++ edition 的 solution 文件）。点击 Build，编译 Solution 并等待编译完成。完成后会在 debug 目录中生成可执行文件 freeswitch.exe。

注意：我们不再支持 Microsoft Visual C++ 2005 is no longer supported。

我在 CentOS（可能其它发行版也会有）上遇到一个问题，提示“/lib/cpp” failing sanity check？

在 CentOS 4.4 上运行 “./configure” 会出现如下错误：“configure: error: C++ preprocessor”/lib/cpp" fails sanity check”。试试使用 yum install gcc-c++ compat-gcc-32 compat-gcc-32-c++ 检查依赖关系并且同意（如果你同意的话）。

运行 `make megaclean` 时出错

试试用下列顺序执行命令：

```
1 ./bootstrap.sh  
2 ./configure  
3 make megaclean  
4 make installall
```

我没有 Microsoft Visual C++，在 Windows 上是否有已经编译好的版本呢？。

有，参见：http://wiki.freeswitch.org/wiki/Installation_Guide#Precompiled_Binaries

12.3.7 SIP

如何设置 SIP 客户端认证？

参见：<http://wiki.freeswitch.org/wiki/Sofia>

如何设置 Music on Hold (MOH)？

参见 [Sofia Configuration Files](#) 中的 hold-music 选项，它可以对每个中继进行设置。或者，也可以通过信道变量 `hold_music` 在 XML 拨号计划中对每个信道进行设置。

在长时间收不到 RTP 后可以自动挂断电话吗？

是的。在 sofia profile 有两个参数管这个事。它们是 `rtp-timeout-sec` 和 `rtp-hold-timeout-sec`。

如果在 FS 控制台上看到 SIP 用户的注册情况？

可以在控制台上使用如下命令。显示 profile 信息和注册信息：

```
1 sofia status profile internal
```

或仅显示注册信息： `sofia status profile internal reg`

12.3.8 IAX2

如何设置 IAX2 客户端认证?

我们有用户目录，但没时间去修改 libiax2 以支持注册。对于外呼电话可以使用 iax/user:pass@remotehost/exten 以支持注册。要正确地支持所有东西，必须有人从头重写 IAX 协议栈。

注：自 Asterisk 升级至 1.6 后，协议不再兼容，也没有人再愿意更新，因此 IAX2 也在 FreeSWITCH 代码树中移到 unsupported 目录中。

FreeSWITCH 支持模块线路（FXS/FSO）吗？

OpenZAP 支持模拟语音卡。详见 <http://wiki.freeswitch.org/wiki/OpenZAP>。

FreeSWITCH 支持 ISDN BRI/BRA 线路吗（S0 Basic Rate Interface）？

初步的支持已经加入了ISDN 模块中（ozmod_isdn）。支持TE（用户）和 NT（网络）模式，包括NT 模式中的拨号音和交叠接收。zaptel + bristuff / dahdi（ozmod_zt）及一个单口的 HFC-S PCI 卡上测试通过。注意：一些高级特性如（transfer、hold）等还不支持。

更新：现在可以使用 Sangoma boost 协议栈与 Sangoma BRI 卡在 FreeSWITCH 中支持 BRI。另见：<http://wiki.freeswitch.org/wiki/FreeTDM>，

FreeSWITCH 是否支持 PRI（E1/J1/T1）？

OpenZAP 支持 PRI 卡（以及模块卡）。OpenZAP 代替了 mod_wanpipe，最初，对 Sangoma PRI cards 的支持是加在 mod_wanpipe 中实现的。

12.3.9 程序

如何在拨号计划中使用 JavaScript（ECMAScript）？

确定 mod_spidermonkey 模块已经编译并加载，在拨号计划中执行：参见 <http://wiki.freeswitch.org/wiki/Javascript> 文档中 FreeSWITCH 对 javascript 的扩展。

如何让 FreeSWITCH 在没有控制台的情况下运行？

```
1 freeswitch -nc (No Console)
```

其它选项有：

```
1 freeswitch -hp (High Priority mode)
2 freeswitch -vg (valgrind, 调试时有用)
```

如何在 FreeSWITCH 中发起一个呼叫?

见: API 命令: Originate http://wiki.freeswitch.org/wiki/Mod_commands#originate。
关于 Sofia sip URL 的语法, 见 <http://wiki.freeswitch.org/wiki/Sofia>。

reloadxml 能重载所有 XML 文件吗 ?

它只是将所有 XML 加载到内存, 并不意味着所有的改变都生效。拨号计划和用户目录会刷新, 它也会触发一个事件 (依赖于 ENUM 设置) 以重载 ENUM。而 sofia profile 的设置不会更新。但你可以使用 sofia 命令使其刷新, 有些改变需要重启某个 sofia profile。

会议设置会在下次创建一个会议时生效。当会议正在进行时不会起作用。

12.3.10 呼叫路由

我如何把 endpoints 放到不同的 context 中, 而不同的 context 又有不同的 extension ?

有几种不同的实现方法:

- 每个 profile 对应一个 context, 每一次都需要一个独立的 IP:Port。
- 在注册数据中使用不同的 domain, 它会自动路由 context。
- 把所有电话都指到一个公共的 context 中, 并使用 execute_extn 或 transfer 转移到其它地方
- 把它们送到一个 IVR, 然后决定下一步去哪里。
- 使用 xml_curl 建立动态的 dialplan, 根据你知道的呼叫数据来决定下一步应该做什么

如何在整个服务器上使用单一的 domain?

如果你想对所有请求提供服务, 并且在单一的 domain 下, 你可以找到 sip_profiles/internal.xml 中的 force-register-domain 一行, 去掉该行的注释, 并在 vars.xml 中设置对应的 domain 即可。在这里我不是十分确信, 但你不能在 directory.xml 中设置另一个唯一的 domain, 它必须与 vars.xml 中的匹配。如果你想让所有注册用户都能在同一个 domain 中列出来 (或者排序), 使用 force-register-db-domain 参数, 如:

```
1 <param name="force-register-domain" value="domainname"/>
2 <param name="force-register-db-domain" value="domainname"/>
```

12.3.11 配置 FreeSWITCH

是否有一个配置 FreeSWITCH 的图形界面？

FreePBX 开发者正在开发 FreePBX V3，它支持 FreeSWITCH。现在，你可以获取一个 [pre-release 的版本](#)。它是一个很有前途的项目，我们鼓励任何对开源 FreeSWITCH GUI 前端感兴趣的人都去支持他们的努力。

FusionPBX，是一个支持多平台，开源的 FreeSWITCH WEB 界面，它基于 BSD 许可证发布。它已证明是一个可定制的、非常灵活的WEB界面。它的后台数据库支持 SQLite、PostgreSQL、MySQL 等。它运行起来非常稳定，从小型的到大型的环境中都已经有所应用。

另一个选择是 WikiPBX，它基于 MPL 许可证发布，使用 Python 基于 Django 框架开发。

XML 糟透了，还有其它选择吗？

是也不是。有其它的选择，但不一定是更好的选择。关于 FreeSWITCH 中 XML 配置的讨论已经足够了，见下面这些资料：

- Anthony 关于为什么选择 XML 的解释 <http://www.freeswitch.org/node/123>
- Anthony 关于创建一个 mod_yaml 的讨论 <http://lists.freeswitch.org/pipermail/freeswitch-users/2008-June/004021.html>

另外，参见 [mod_dialplan_asterisk](#) 以获得更多关于使用 INI 格式的 extensions.conf 来配置 dialplan 的信息。（说明：它不灵活，并且不如 XML 强大。）

以上的例子都说明，如果 XML 你阻止你尝试 FreeSWITCH 的唯一原因的话，那么我们推荐你还是先用默认的配置试一试。你会感到惊奇，因为仅需修改一点点 XML 就可能干好多好多的事情。并且，你也会由于你能通过修改 XML 而做到的事情令人刮目相看。

另外一种不使用 XML Dialplan 控制呼叫逻辑的方法是通过 mod_event_socket 或称 ESL。使用这种它，你就可以不用 XML，而使用编译或解释型的程序语言来控制你的呼叫逻辑。

12.4 Idapted的FreeSWITCH实践⁶

FreeSWITCH是如何创造数千个就业机会并改变人们命运的？

⁶与本文对应的英文版本发布在 http://wiki.freeswitch.org/wiki/FreeSWITCH_Testimonial_on_Idapted.com，但本文不是原英文版的翻译。本文写于2009年ClueCon前夕，是本书作者与原Idapted创始人/CTO Jonathan Palley一起写的，JP将FreeSWITCH带到中国，作者从那里开始学习FreeSWITCH。当时由FreeSWITCH支持的EQ英语平台曾是国内最大的一对一在线口语教学平台。

“改变命运”，说起来有点言过其实，但从某种意义上说，我们确实做到了。FreeSWITCH已成为我们业务逻辑中非常关键的一部分，在此，我们想与大家分享一下我们的一点使用经验，并希望能以此来感谢FreeSWITCH社区中所有帮助过我们的人。

我们到底做了什么？不像其它公司，我们并不使用FreeSWITCH来经营电信业务，而是主要做技术和教育(<http://www.eqenglish.com>)。在我国，学生要到说英语的国家留学或工作，必须要通过相关的英语考试。但是，他们大都缺少真正的英语环境，很少能跟母语是英语的人交谈；即使有，也通常很昂贵。与此同时，在美国，却有很多人在寻找在家工作的机会。这些人通常具有良好的教育背景和很好的工作经验，通过我们的课程培训，他们就能成为很好的老师。是的，我们有自己的课程体系。比方说，雅思考试并不死抠语法，而是更注重学生是否能准确流利地用英语表达自己的思想。但目前很多教科书上还都是照本宣科，使很多学生得不到要领。而我们的老师都能启发学生做到些。不管你是否相信，我们的绝大部分学生都能通过一段时间的培训达到自己期望的分数，而在此之前，他们有的考了几年都未达到。当然，我们绝不是应试教育。就在几小时前，有一个学生还给我们的课程顾问打电话，喜极而泣。同时，对我们的老师而言，他们正在做着一项以前在美国业界还没有过的工作—<http://www.idapted.com>。

FreeSWITCH到底是如何帮助我们的成功的呢？首先，它是我们整个语音系统的核心，担负着录音、电话路由、多协议连接、与WEB系统无缝集成等重要任务，这些都是我们成功的关键。更重要的是，要横跨太平洋连接中、美两国，考虑不同国家的网络环境，建立多个服务器，这都是很严峻的挑战。如果采用普通的商业软件解决方案的话实现起来会非常困难，甚至是不可能的，而FreeSWITCH作为一款开源软件，却能很好地做到了这一点。

在此之前，我们也曾使用过其它的VoIP平台，用的最多的是Asterisk。诚然，Asterisk也是一款很优秀的开源软件，但当时我们经常会遇到一些模块死锁，录音超长等问题，一直无法解决。特别是随着我们的业务逻辑日渐复杂、用户一多起来之后，它更显得有些力不从心。幸运的是，我们遇到了FreeSWITCH。我们用了一天的时间阅读FreeSWITCH的源代码(这正是我们喜欢开源软件的原因)，当时几乎是拍案而起：“这不正是我们需要的吗？”接下来，我们用了几天的时间熟悉和测试各个模块，并向FreeSWITCH开发者提了好多修改建议，同时也得到了他们和社区很多的帮助。我们从2008年4月正式开始使用FreeSWITCH。

我们很快实现了所有的业务逻辑。美国的老师通过我们自己编写的VoIP客户端连接到FreeSWITCH以及我们的WEB系统上，学生则可以根据上课进度连接老师进行一对一对话。说起来倒也简单，等学生预习完所有课程后，通过在WEB界面上输入一个手机号(或其它电话号码，甚至是Skype或gtalk用户名)，我们的平台就会呼叫该号码，并选择最好的老师与之交谈。同时，老师和学生的电脑上都会同步显示当前课程的内容。

我们遇到最大的挑战是老师和学生分别在太平洋的两岸。曾经出现过几次亚、太之间的光缆中断事故，对我们的语音通信造成了很大的影响。后来，我们不得不通过在多个不同的数据中心架设多台冗余的服务器来减少单一路径对我们的影响。

另外，我们也在生产服务器上运行FreeSWITCH的多个实例。藉此我们可以在使用最新代码的同时又减少BUG的影响(说实在的，其实FreeSWITCH的BUG通常修的很快，也是极少数的trunk代码比上一个稳定发行版更稳定的工程之一)。我们的总体结构是通过一个“主”FreeSWITCH(FS)连接到其它FreeSWITCH实例上，专门负责skype的(FS-skype)就只加载skype模块mod_skypiax，专门负责Google Talk的(FS-gtalk)就只加载Gtalk模块mod_dingaling。FreeSWITCH允许我们通过设置不同的配置文件来运行同一个或多个版本的代码。因此我们设置了/usr/local/freeswitch, /usr/local/skype, /usr/local/gtalk等等的目录结构

来存放不同的配置文件。更重要的一点是，如果某个FreeSWITCH实例崩溃了(如FS-skype。说实话，FS-skype现在已经很稳定，但我们最初使用时该模块刚处于测试阶段，所以有时会崩溃)，电话就会自动路由到其它的服务器，甚至可能到其它数据中心的服务器(我们在香港和大陆的几个数据中心里放了冗余的服务器，如图)。

```
1          |-----PSTN gateways
2  /-----\      |--- FS-skype
3  |   FS  |-----|--- FS-gtalk
4  \-----/      |--- FS-skype2
5          |--- more ...
```

我们使用了以下几个主要模块：

- mod_sofia(SIP模块)

FreeSWITCH使用sofia-sip提供SIP支持。我们使用SIP连接老师，而通过SIP-PSTN网关呼叫学生。使用PSTN网络呼叫学生的好处是，学生可以直接输入一个手机号（或固定电话）接听电话，而不需要在电脑上进行任何配置。这就免去了我们帮助他们解决网络、麦克风、耳机等问题的麻烦(相当多的学生对设置这些东西不是很熟悉，所以要远程处理这些问题，工作量是很大的)。FreeSWITCH在路由选择方面很智能，所以我们在一个网关失败后很容易的选择其它的网关。而且，FreeSWITCH的NAT穿透力是非常棒的，它支持AutoNAT，可以在不需要STUN的情况下在NAT网络上工作。

- mod_skypiax(Skype模块)

尽管手机易于使用，但有些学生是为了学业奔波于异地，由于漫游的原因无法用手机接听。而且我们也发现确实有许多学生喜欢带着耳机跟老师连线。正好FreeSWITCH有一个模块可以连接Skype。最初我们使用时，它极不稳定(这也是为什么我们启动多个FreeSWITCH实例的原因之一)。老实说，当时它确实给我们带来了很大的麻烦，但想想它仅仅是阶段的代码，本来就有此风险。幸运的是，模块作者Giovanni Maruzzelli(意大利人)非常勤奋，他很快修复了许多BUG，甚至还抽出时间登录到我们服务器上帮助查找问题(我们欠他一个大大的人情)。为了使该模块更有用，我们也写了许多patch，如使用ANY和RR接口进行顺序和循环选线以及一些BUG修复等。另外一些特性如continue-load-on-fail和auto-skype-user尚未合并到trunk代码中去。非常感谢FreeSWITCH社区给了我们一个可以奉献的平台。

- mod_erlang_events(Erlang模块)

它是实现我们新的业务逻辑的另一个关键模块。我们有非常复杂的实时队列以及路由系统，以及很多相关的WEB页面和TCP套接字(socket)。这些很难直接在FreeSWITCH中实现。而Erlang天生的多进程支持使它成为实现这些技术最适合工具。我们用Erlang开发了粘合FreeSWITCH与前端Ruby on

Rails应用的中间件，通过HTTP和共享数据库与前端通信，而在后端，则通过mod_erlang_events与FreeSWITCH集成。每当有一个学生与老师连线时，我们都创建一个新的Erlang进程，所有的消息与状态机均在该进程内部完成，而不影响其它进程。同时，它还根据收到的FreeSWITCH事件(event)，通过实时推送技术(comet socket)同步更新学生和老师端的WEB界面。

- mod_conference(会议模块)

我们使用mod_conference召开老师的电话会议。我们的老师分布在美国的各个州，而我们的总部在北京。通过网络电话会议，我们不仅可以节省通信成本，更重要的是我们可以方便的进行会议录音，播放声音文件，记录会议出席情况等，也便于及时有效地对老师进行培训。

- mod_fifo(队列模块)

另外，我们还把它用作办公室PBX(提供中英双语的语音菜单)以及呼叫中心(销售)和客服系统。mod_fifo本是个很简单的模块，我们提交了一些Patch，使得它能完成相对复杂的呼叫队列功能。销售及客服系统均与我们自己编写的CRM系统(客户关系管理系统)紧密集成，大大提高了我们销售的业绩和客户服务水平。

FreeSWITCH非常强大和稳定，我们只是使用了其功能的一小部分。当然，除了其强大的功能外，更重要的是它的开放性及社区支持。首先，它是开源的，当遇到问题时我们很容易地查看源代码找到问题所在。其次，它的社区是活跃地、友好地，如果我们提交一个BUG，一经确认通常在一天之内就修复了(如果不是时差的关系可能会更快)。再次，我们可以提交自己的补丁程序或添加新功能，只要是有用的，他们都很乐于接受。值得一提的是，Anthony Minessale及其核心团队帮助我们实现了完整的SIP支持。回到1.0.0时代，Anthony曾在mod_sofia上做了大量工作，最终使得SIP客户端可以在不需要STUN的情况下在NAT网络上工作。这一点的重要性并不在于他做了多少改变，而是他帮助我们确确实实实现了一个有竞争力的系统，同时我们认为这一特性对FreeSWITCH也相当重要。

总的来说，FreeSWITCH给我们带来了新的技术并成就了新的商业模型。而强大的社区支持又让它能做的更好。我们每天都很兴奋地在它上面工作，以推动这项技术向更深更广的方向发展。同时我们非常确信这是未来最理想的电话平台。

第十三章 后记

从决定写书，到现在算起来已经有两年多了。当时就预料到这个过程可能非常长，因此决定在互联网上采用知识共享协议（Creative Commons，简称CC）在 FreeSWITCH-CN¹ 网站上发布。本书的前面一些章节远在FreeSWITCH官方的第一本书《FreeSWITCH 1.0.6》上市之前就发布了，因此本人常以为该书实为FreeSWITCH第一本书。

前期，空闲时间比较多，写得也勤一些。写得快的时候，也不便于一下子全发出来，因此，便慢慢地“攒”了几章，隔几天以后发。这样看起来还好像有个良好的进度。到后来，工作渐渐忙了起来，便写得慢了。手一松，手里存的就渐渐都发完了，也就不好再控制进度了。但我总会在FreeSWITCH 官方的新书发布前突击写两篇，以避免别人说我是抄的。但即使是这样，还经常听人说：“你翻译的书真好啊！”我非常感谢他们说“好”，同时也会告诉他们我不是翻译的（除了附录里的一些经典资料）。

前几章放到网上以后，得到了大家很好的反馈。每次看到网友在网站鼓励的留言以及邮件列表里良好的反馈，即使时间再忙，也会尽量多写一些。但写跟讲不一样。讲起来可以胡说，或一两句带过，但写的话却不能模棱两可，因而需要查找好多资料，花的时间就比看上去多得多。

写得慢了，反馈就自然就少了，因此写作的动力也就小了。同时，我也渐渐发现好多文章被转载到许多个人博客上。从一方面讲我是比较欣慰的，因为被人转载也算是被认可吧；但从另一方面讲，CC协议虽然不禁止转载，但看到好多人不加到原网页链接，也不注明作者姓名，反而当作自己的原创，我就不怎么高兴了。因此写作的兴趣也大减。偶尔有点心得什么的就随便发篇博客，对里面的错别字也不那么注意了。这对于我忠实的读者而言，我是深有歉意的。所以有时候等着文章被转走之后，重新再看的时候会把错别字改掉。

有些网友也很细心地帮我找错别字，我心里非常感激。有这些忠实的读者还是挺令人欣慰的。他们不时地鼓动我出书，好多都说出了以后必定第一个购买。我不记得他们的名字，但是我很感激。可出书还毕竟不是那么简单的。首先我没有那么多时间把这本书写完，其次该书的读者群还比较小，因此觉得这个想法离实现还比较远。

后来，我清点了一下以前写的一些博客，稍做整理，串起来，便有了《实战》一章，这本书总算是走入正题了。

有一天，一位朋友告诉我说：你太牛了，文章在百度文库上到处都是。我听了很诧异，查看后发现有人甚至用 Word 排版收集了书里全部的内容，唯独缺少的是作者的姓名。或许正赶上

¹<http://www.freeswitch.org.cn>

那两天我忙得比较烦，在我使用正常的投诉武器未得到满意的反馈之后我发了一些微博，并写了《原来百度公司是这样对待版权的—致百度的一封公开信》²。算是起到了一些效果吧—虽然百度没有道歉，但至少他们偷偷地把那些文章都删掉了。当然春风吹又生是以后的事。

又后来我在豆丁和其它网站上也看到同样的情况，尝试投诉了一下，从结果看他们好像没有百度那么自觉。

罢了，写我的书，让他们抄去吧。只不过，受害的是我的忠实读者们。本来，如果心情好的话我是可以多写几篇的！

就在前几天，又有一位网友说喜欢我的书并问我什么时候能出版。我告诉他：“出书不是一时半会儿就能发生的，但是如果有人捐赠的话我可能会写得快些。”这位网友二话没说，立即要了我的银行账号，并迅速转账。虽然钱不多，但是我感觉这是很有意义的。我记住了他的名字—Tim Yang。

这也是我写书以来收到的第一笔捐赠。虽然钱不多，但足以使我下决心把这本书印出来。虽然本书还没有最终完成，但我想，如果在今年首次社区开发者沙龙上给大家发一发，大家应该会喜欢吧？

但事实证明“印出来”也不是一件容易的事。

我最早开始写作的时候，使用 Latex 排版。Latex 是史上最酷的排版工具，我也很喜欢，但一直没有掌握其要领。而且，它处理起中文来总是非常麻烦。后来便转到 Markdown。其语法比较简单，很容易放到我的个人博客以及 FreeSWITCH-CN 网站上。不用再太关注排版的细节，使我也可以省出更多的时间用于写作。

曾经有个以前的同事想帮我整理一个 Word 版的，无奈用 Word 对程序代码排版太麻烦，排出来又不好看。对于书中的很多配置文件和程序代码，排出来的格式更是惨不忍。

后来借助于 Pandoc，终于可以直接从 Markdown 生成漂亮的 HTML 了，同时通过 pdflatex 也能生成非常精美的 PDF。只是让它能很好地处理中文上我还是花了好长时间。但为了能赶在沙龙之前把书印出来，我还是尽量把很多美好夜晚都享受了。

在排版的同时，我发现了很多的错别字，而且也发现，以前写的那些东西由于太随意的放到博客上，好多够读起来都不怎么通顺。要把这些句子都一一推敲，我恐怕再过一个月也弄不好。因此，我只好尽量在快速浏览中把我发现的一些改好。其它的，留给读者慢慢帮我挑吧。

由于前面的章节时间都比较久远，我也花了些时间更新并重写了一些部分，补齐了以前标记为 TODO 的一些章节，同时又增加了网友呼声最高的 *Event Socket* 一章。

我的好朋友 Kenny Bloom 听说我要印书，非常高兴地帮我制作了本书的封面。

样书印出来了，看起来感觉还不错。有些朋友建议我把定价也印上，辛勤工作嘛，总应该有点回报。但我想，卖书不是我的本意。我建立了 FreeSWITCH-CN 中文社区，就是希望与众多网友多多交流经验，共同学习，而且标上定价，就改变了性质，有“非法出版物”之嫌。诚然，我也希望我的劳动能得到一些回报，尤其是看到我的文章被许多人不顾版权声明胡乱转载的时候。但我不想让少数人的不仁伤害了我的道义，而且我更喜欢的方式是国际上开源社区的方式—Donation。我想尝试一下—如果有人捐赠，无论多少，便送一本书，如何？而且，我也想从大家

²<http://www.freeswitch.org.cn/2012/02/27/yuan-lai-bai-du-gong-si-shi-zhe-yang-dui-dai-ban-quan-de-zhi-bai-du-de-yi-feng-gong-kai-xin.html>

的捐赠中拿出一部分，再捐赠到FreeSWITCH官方的开发者社区中。大家为中文社区做贡献，就是为全世界的社区做贡献！事实上，我当时真是这么想的。但事实上，我错了，因为在样书印出来之后我忽然想起一个词——“公开募捐”。咨询了我的律师，果然，如果我那么做是有隐患的。这也完全颠覆了我的开源社区梦。这意味着——我将无法再收取捐赠。同时，我也只好在我的书的扉页上加上了一个不伦不类的“内部培训教材”，并在本书网站上取消了一切与捐赠有关的词。

好在，为自由软件社区做贡献并不一定是捐钱捐物这么狭隘。汇报Bug，翻译中文资料，在邮件列表中及QQ群中帮助其它同学，分享使用经验与代码，有时候甚至一些小小的鼓励都会对整个社区有莫大的帮助。前一段时间，在网友CH.LAU及默言的动员下，我们开始了FreeSWITCH中文文档翻译计划—<http://wiki.freeswitch.org.cn/>，并取得了一些进展。希望大家也一起参与进来，为自由软件的壮大发展尽一份力量。

最后，附上一些网友的留言。他们的支持和鼓励是督促我写作的不竭的动力：

- vontall: 我在google reader里订阅了本站，内容非常的好。目前我是在用Asterisk，但对FreeSWITCH也很有兴趣。未来之路一书对于Asterisk的推广是显而易见的，FreeSWITCH也确实缺乏这么一份资料。感谢您的无私奉献。
- liuhyu2000: 刚刚看过这篇，真好，我已经喜欢上了！
- Clark Lee: 很好的说明，完全入门freeswitch，谢谢，期待你更多的作品
- Phoenix Huang: 写得真好. 楼主加个油，出书吧，哥们必定支持
- voip123: 神一样的技术领袖！
- Jeromy: 谢谢，希望能够再写一下怎么和PSTN互通
- wzs: 看完这个真的认识更多。别的地方都找不到这样的资料。真是太感谢了。
- truelie: 真是太神奇了，感谢分享！就是不知道，SIP电话注册的时候，sofia模块是如何通知FS调用mod_xml_curl执行自己的gateway-url？
- crazy2005king: 以前对SIP根本没概念，通过看本篇文章终于入门了
- crazy2005king: 看了好几遍这篇文章，写得太好了
- bihu: 非常感谢，学习了。
- wavecb: 辛苦了，FreeSwitch中国贡献第一人啊
- Henry: 你的sample code, xml_curl 看了年都碰，你一放我就懂了，之前都在用xml_odbc.
- Toddler: 非常高，讲的很通俗易懂，感谢。
- rewing: 大牛，忍不住赞一个 虽然，我还要再看一遍才能吸收，不过相对来说，这已经是最好的教材了
- liuhyu2000: 好极了，讲的这么通俗，开心啊我！

- slickqt: 好,期待下一篇.
- Richard: 谢谢你分享这么好的内容, 感谢你的辛勤工作!
- chengcheng: 故事讲的真好啊! 有大师的范啊!
- Henry: SS7的内容清楚简单,VoIP 玩了几年了SS7还没像今天这么清楚过,谢谢
- lyck: 不错,写的很好,明白了很多内容,多谢
- 冰神月使: 顶! SIP协议非常不错

由于时间仓促, 疏漏之处在所难免, 请广大读者朋友不吝批评指正。

电子邮箱: book@freeswitch.org.cn 。

是为后记, 与诸君共勉!

杜金房

2012年6月14日于北京和园国际青年旅舍

本书未完, 待续

作者简介

杜金房（Seven Du），男。FreeSWITCH-CN中文社区创始人，FreeSWITCH代码贡献者。

2001年毕业于烟台大学，同年进入烟台电信工作，负责交换机、网管系统维护，并开发了大量网管及办公系统。经历了电信改通信、通信改网通等一系列变革。

2008到北京，加入Idapted，开始使用FreeSWITCH。

2009年创办FreeSWITCH-CN（www.freeswitch.org.cn）。

2011年创办北京信悦通科技有限公司（x-y-t.com），提供FreeSWITCH咨询服务、解决方案及商业支持。

2011及2012年，参加在美国芝加哥举办的ClueCon全球VoIP开发者大会，并演讲。