

Kamailio 实战

Kamailio in Action

从零开始学习Kamailio，从实践到理论，
一步一步深入掌握Kamailio的精髓。



目 录

版权信息

内容简介

推荐序一

推荐序二

前言

第1章 Kamailio与SIP

 1.1 什么是Kamailio

 1.2 背景

 1.3 SIP

 1.4 Kamailio基本架构

第2章 理解Kamailio配置文件

 2.1 基本配置文件

 2.2 原生脚本

 2.3 Lua脚本

 2.4 Lua脚本的其他写法

第3章 Kamailio基本概念和组件

 3.1 core详解

 3.2 其他概念和组件

第4章 KEMI详解

 4.1 KEMI Lua入口

 4.2 KEMI函数

 4.3 在C函数中导出KEMI函数

 4.4 KEMI和伪变量

 4.5 核心和pv模块中的函数

 4.6 原生脚本与KEMI对比

 4.7 其他

第5章 Kamailio运行环境与实例

 5.1 运行Kamailio

 5.2 将SIP呼叫转发到FreeSWITCH

 5.3 从简单的路由脚本开始

 5.4 Kamailio命令行工具

5.5 Web管理界面

5.6 调试与排错

第6章 使用Kamailio做SIP路由转发

6.1 什么是路由

6.2 基本路由转发

6.3 使用dispatcher模块做路由转发和负载均衡

6.4 呼叫从哪里来

6.5 API路由

6.6 在KEMI脚本中调用原生脚本中的路由块

第7章 数据库操作

7.1 初始化数据库

7.2 配置数据库连接

7.3 在路由时进行SQL查询

7.4 其他函数和伪变量

7.5 常用数据库表结构

第8章 15个典型的路由示例

8.1 通过号码分析树进行路由

8.2 号码翻译

8.3 低成本路由

8.4 前缀路由

8.5 动态路由

8.6 缩位拨号

8.7 通过别名数据库路由

8.8 运营商路由

8.9 字冠域名翻译

8.10 用户注册和查询

8.11 向外注册

8.12 更多AVP示例

8.13 话单

8.14 SBC

8.15 WebRTC

第9章 性能

9.1 性能测试

9.2 拆解Kamailio高性能信令服务设计

第10章 安全

10.1 基本安全手段和策略

10.2 限呼

10.3 TLS

10.4 iptables

10.5 其他安全建议和相关链接

附录A 安装Kamailio

A.1 在Debian和Ubuntu上安装Kamailio

A.2 从源代码安装

附录B FreeSWITCH快速入门

B.1 FreeSWITCH简介

B.2 运行FreeSWITCH

B.3 环境变量

B.4 配置

B.5 常用命令

B.6 修改配置

B.7 增加声音文件

B.8 host模式网络

B.9 测试号码

附录C Lua快速入门

C.1 Lua与JavaScript的相似性

C.2 区别

C.3 其他

附录D Docker简介及常用命令

D.1 Docker简介

D.2 Docker安装

D.3 基本概念

D.4 常用命令

D.5 Docker Compose

附录E 模块索引表

后记

作者简介

版权信息

书名：Kamailio实战
作者：杜金房 吕佳婷
出版社：机械工业出版社
出版时间：2022-10-26
ISBN：9787111712473

内容简介

这是一本帮助读者真正把Kamailio用起来的专业工具书、百科全书，由《FreeSWITCH权威指南》的作者、世界知名通信专家、Kamailio中文社区联合创始人、FreeSWITCH-CN中文社区创始人兼执行主席撰写，来自上海交大、中南大学等多所高校以及声网、腾讯云等国内知名企业和社区的多位通信专家鼎力推荐。

本书涵盖Kamailio核心概念、运行原理、基本配置、路由转发逻辑、路由脚本撰写、数据库操作、性能测试、安全等内容，并包含大量实战案例，案例中的参数都可以拿来直接使用。

第1章重点介绍与SIP相关的基本概念和网络拓扑，以及Kamailio路由脚本的基本架构，可帮助读者全面认识Kamailio。

第2章和第3章深入讲解Kamailio的配置文件、基本概念和核心组件，以帮助读者深入理解Kamailio，并为后面把Kamailio用起来打好基础。

第4~7章分别介绍KEMI、Kamailio的运行环境、Kamailio做路由转发的方法，以及Kamailio中数据库的操作与使用方法，同时配有大量实际案例，这部分可帮助读者快速Kamailio用起来。其中包括路由脚本编写、命令行工具使用、调试手段、无状态转发和有状态转发、并行转发和串行转发、有负载均衡、API路由等重点内容。

第8章用15个案例进一步解Kamailio中常见的模块及其用法，以及一些高级话题，如SBC、媒体代理和拓扑隐藏、WebRTC相关的信令及媒体转换等。

第9章介绍与Kamailio性能相关的知识，包括构建高性能SIP服务器的注意事项以及Kamailio具有高性能的秘密。

第10章分享实际生产系统中常见的安全问题，并给出对应的解决方案。

另外，为了便于读者快速上手，本书还以附录的形式对Kamailio安装方法，以及FreeSWITCH、Lua、Docker入门知识进行了简单介绍。

Foreword

推荐序一

杜金房先生邀我为他的新书写一个推荐序，我非常高兴。我曾在通信行业工作，创业之前，在Motorola、3Com等公司研发通信产品10余年，对通信类的技术非常感兴趣。不过我之前对Kamailio不是很了解，因此利用周末时间学习了一下。

Kamailio是一款有20多年历史的开源软件，能历经这么多年仍然保持强大的生命力，足见其不凡。老杜的书写得深入浅出，让我轻松理解了Kamailio，也让我对老杜产生了由衷的佩服。

我现在做的产品TDengine就是开源的，认识老杜也是因为开源。大概是由于我比较高调，老杜很早就来到我们的用户微信群跟我们交流用TDengine来处理话单的存储和查询的感受。在2021年的一次开源大会上，老杜还拉着我在嘈杂的晚会一角聊了半个小时。

TDengine是我现在带领团队在做的创业项目。2016年下半年，我发现大家都习惯性采用Hadoop大数据平台来处理从机器、传感器、智能硬件上采集到的数据。其实这些数据是很有特点的，比如时序、结构化、很少删除等，我便想到应该充分利用这些数据特点，开发一个专用的大数据处理平台，来大幅提高系统的处理效率，减少系统研发和维护的复杂度。我个人判断，今后世界上90%的数据都是机器、传感器产生的，而不会是人产生的，这是一个巨大的市场，很有想象空间，因此我决定再次创业，专注于物联网、工业互联网领域，打造一个专用的高效时序大数据处理平台。

跟Kamailio类似，TDengine也是使用C语言开发的。当然，20多年前做Kamailio这样的软件，除了C语言几乎没什么其他选择，而现在我们有了更多的选择，比如被誉为互联网时代的C语言的Go语言等。但是因为我个人一直用C语言做研发，而且底层软件，包括几乎所有流行的数据库，都是用C语言开发的，所以我还是毫不犹豫地选择了C语言。老杜也是C语言的拥趸，看到他在FreeSWITCH开源项目中10多年间贡献了超过15万行代码，我还是非常佩服的。向开源项目做贡献跟自己做开源项目不一样，比如我自己的TDengine由我自主可控，可以提交任何我认为正确的代码，而向开源项目做贡献，要得到上游项目方的认可。老杜能够持续多年贡献这么多代码，本身就是一种实力的表现。

上个月老杜发起了一个开源项目：在FreeSWITCH中写一个TDengine的模块，使用TDengine存话单。话单本身就是TDengine的一个很好的应用场景，符合时序数据库的特点：数据量大，总体按时间有序，一次写、多次读且不能修改。在开发过程中他还遇到了TDengine客户端库与FreeSWITCH中符号表冲突的问题，并向我们提出了问题和解决方案。这也是开源的好处，如果不是他，我们可能要过很久才会发现这些潜在的问题，何况他还给我们带来了免费的解决方案。未来必将是开源软件主导的世界，因为这样会让整个社会的协作更有效率，让全球的开发者都能参与进来，产品的质量也更有保障。

IT技术表面上变化很快，但究其根本，变化很小，因为底层的原理都是一样的。话单，本质上就是时序数据。而时序数据的存储，与IT系统中的消息队列的存储几乎没有区别。数据库技术中的集群技术，包括高可靠、高可用等，在通信行业，至少30年前就在研究和应用了。通信行业的集群需要通过分布式技术，让系统处理能力实现水平扩展，让硬件实现热插拔，让软件实现在线升级。因此，如果你精通一个领域的知识，再转到另外一个相关领域，会很容易。

再说回Kamailio，这本书主要以Lua语言来写路由脚本，这对我来说很亲切。我们在TDengine客户端库中也用到了Lua语言。

这是第一本关于Kamailio的中文书，如果你在通信领域工作，特别是在做SIP相关工作，推荐你读一读这本书。

陶建辉
TDengine创始人
2022年5月写于美国加州

Foreword

推荐序二

收到杜总让我写推荐序的邀请后，我一口气看完了杜总发过来的手稿，感到既惶恐又激动。感到惶恐是因为我平时除了写代码之外，很少在其他的“写”上下功夫。一是因为“懒”，二是因为文采实在有限——我的老板经常对我写的材料感到无可奈何（在此，感谢我的老板的宽容，并给我大块的自由时间）。我担心我这写作水平把杜总这么好的书写坏。至于激动，是因为杜总不仅技术水平过硬，而且性格好，交际广。他认识那么多技术专家，却还能想起我，让我觉得荣幸之至。

既然要写推荐序，那么我就按部就班地写两方面的东西。一方面是Kamailio本身，另一方面是杜总。

关于Kamailio，我是在2012年开始使用的。在那之前，我们几个人开发了一套完整的VoIP产品，并于2009年卖给了美国一家公司。两年之后，我厌倦了安逸，萌生了再开发一套VoIP产品的念头。重起炉灶，从零开始是不现实的，那也是对全球源码贡献者的不尊重。当时在我们面前有几个优秀的开源栈：Resiprocate、Yate、FreeSWITCH、OpenSIPS、BellSIP、Kamailio、Sofia等。由于当时我们还有大量的客户使用E1/T1中继，所以还要考虑IMS、SIGTRAN、SS7相关模块的开发。经过将近半年的摸索、压测、比对，我们在系统方面选择了FreeSWITCH+Kamailio+Yate的组合，以求在终端方面提供基于IMS架构的SIP软电话和WebRTC网页软电话。

在这里，我简单说一下我对产品的理解。相对来说，我觉得FreeSWITCH、Kamailio等都不算是一个完整的产品。一个产品是需要从整体用户需求、稳定性、可靠性、可兼容性、可维护性等方面来综合设计的（先声明，我不是产品经理），比如负载均衡、路由分发、资源重组、信令媒体分离、信令协议栈适配、音视频媒体分离、传输和编解码算法分离……所以，很多国内精通FreeSWITCH或者Kamailio的程序员，不一定能做出一个很好的产品。

Kamailio和OpenSIPS都是OpenSER的分支，为什么我们选择了Kamailio呢？这和我们当时的产品思想有关。这主要体现在如下几方面。

第一方面，我们使用FreeSWITCH做媒体，并通过dialplan的Lua接口，使用Lua语言开发了一套B2BUA模块。这个B2BUA模块可以在不使用Socket的情况下，直接在应用层对接Kamailio的Lua模块并进行进程间的交互，从而形成SIPServer（Kamailio）+B2BUA+MediaServer（FreeSWITCH）的标准VoIP分布式架构。

第二方面，Kamailio能够很好地支持WebSocket交互，从而支持WebRTC通信。

第三方面，Kamailio能够方便对接Yate的SIGTRAN模块，进而驱动M3K、M5K等外部SIGTRAN设备。这样Kamailio就成了一个功能比较齐全的网关模块（支持SIP、SIGTRAN、WSS等）。

第四方面，Kamailio能够很完美地支持IMS架构中的P-CSCF和S-CSCF（虽然还是修改了一些代码）对接。现在随着5G的全面部署，4G中的IMS模块也变成了v-IMS模块，Kamailio能完美实现对新网元的功能对接。

当然，其他的开源栈并非不优秀，只是基于我们的产品设计，我们选择了Kamailio，并且它也确实很好用！

关于杜总，我们年纪相同，相识于2017年。当时，他正为声网的RTC四处奔波。实际上，我和杜总身上有一些共同的标签，比如技术控，所以，几次交流后就变成了好朋友。但杜总身上有很多我不具备的优秀品质，比如勤奋、热爱分享。

杜总出版了多本书，最有名的是《FreeSWITCH权威指南》。书很好，我看了好几遍。其实，我要比杜总更早接触FreeSWITCH和Kamailio。但我总是宁愿看晦涩的英文文档，也从没动过把自己踩的坑好好整理成中文书的念头。2019年，我和杜总曾计划联合写一本书，书名和目录都定下来了，但我却一直停工到现在。看到杜总又一本新书即将出版，我愈加觉得自己的“懒癌症”有点严重了！

杜总通过出书、讲课、建社区，极大地拉低了VoIP开发的技术门槛，培训出了无数优秀的技术人

员，他为RTC行业发展做出了有目共睹的贡献。

最后，对于这本书，我希望从事RTC开发的伙伴能够人手一本，因为它会帮大家降低时间成本，提高开发效率。

王文敏
中移在线高级技术专家

Preface

前言

早就想写这样一本书了。

自《FreeSWITCH权威指南》于2014年出版以来，我收获了好多读者，也收到了很多很好的反馈，很多读者说希望能看到一本关于SIP代理服务器的书。因为随着业务规模扩大，FreeSWITCH势必要做集群，而做集群就需要一个SIP代理服务器。Kamailio是一个很好的SIP代理服务器，在过去的几年里，我们也在很多项目中用到了它。与其说为了满足读者的期盼，不如说我自己想写一本关于Kamailio的书。因为我们团队的小伙伴要看，我们给客户做培训要用，而关于Kamailio的中文资料却少之又少。

我与Kamailio的渊源

我与Kamailio的故事还得从很久以前说起。



我大学毕业后第一份工作是在烟台电信，做程控交换机的运维，在大学里学的“程控数字交换与现代通信网”课程算是派上了用场。那时候，运营商使用的交换机还是程控交换机，非常庞大且笨重。我维护的交换机主要是上海贝尔S1240系列以及华为的CC08系列。感谢那些工作，让我做到了将理论与实践相结合，并对通信网和七号信令有了深入的理解。那些年，我也见证了电信改通信、通信改网通的行业变革。至于运营商混合所有制改革，那都是后来的事。

后来我又学习了Asterisk和FreeSWITCH，并加入一家北京的在线教育公司，使用FreeSWITCH做实时在线口语教学。良好的通信知识基础和开放的团队氛围让我很快学会了FreeSWITCH，并最终加入FreeSWITCH开源社区，成了FreeSWITCH开源项目的核心贡献者。我提交了很多补丁，涉及核心交换功能以及很多外围的模块，其中视频相关的代码和模块前期几乎都是我写的。后来，我整理了自己的学习笔记及博客文章，又编写了一些新内容，出版了《FreeSWITCH权威指南》一书。该书出版时我已经辞职，开始独立创业，主要提供FreeSWITCH、OpenSIPS、Kamailio相关技术开发和咨询工作。现在我的工作主要是带领团队基于这些开源项目打造与IP-PBX、软交换和呼叫中心相关的实时通信产品和服务。

随着我对FreeSWITCH的深入理解和使用、维护的系统规模的扩大，以及客户对安全性、稳定性的要求越来越高，单机版的FreeSWITCH已经无法满足我的需求，因此，我又学习了Kamailio以及OpenSIPS，并成功将它们应用于多个大型项目中。当时的Kamailio和OpenSIPS版本都还比较低，Kamailio中还有很多处于独立目录中的OpenSER模块。也许是无知者无畏，当时我们还直接拉了Kamailio的代码按客户的要求进行大改，后来虽然测试成功了，但是我们最终决定只用FreeSWITCH，而我们编写的那些代码也没有上线。不过我们因此积累了很多宝贵的一手经验。

再后来，我们在另一个大型项目中使用了OpenSIPS，用其与FreeSWITCH配合。当时我们使用了一种非常简单但有效的架构：SIP话机使用基于UDP的SIP协议通过OpenSIPS注册到FreeSWITCH，话机做主叫时会经过OpenSIPS做负载均衡，做被叫时FreeSWITCH直接呼叫话机而不经过OpenSIPS。这种架构简单好用，一直没出过问题，直到数年后我们遇到仅支持TCP且不太规范的SIP终端。

Kamailio和OpenSIPS都是OpenSER的延续版，在最初的版本中两者其实差别不大。我在



《FreeSWITCH实例解析》中写过一些OpenSIPS相关的内容。至于为什么是OpenSIPS而不是Kamailio，大概是因为OpenSIPS的名字中含有SIP吧，也可能是因为我们感觉国内的OpenSIPS用户要更多一些，但无论如何总得选一个吧。实际上，两者一直在更新版本，但主要功能没多大差别，在我的项目中两者也都用过，具体用哪个主要看甲方的偏好。

我跟Kamailio和OpenSIPS的主要作者在ClueCon上见过多次面，喝过啤酒，聊过天。他们人也很好，同是程序员，我和他们有聊不完的话题。现在，OpenSIPS团队在罗马尼亚，Kamailio主要作者

则在德国柏林。2017年我们还邀请OpenSIPS开发团队到我们主办的“FreeSWITCH开发者沙龙”上做过远程演讲。

最近在项目中使用Kamailio比较多，主要是因为我比较喜欢Kamailio中的KEMI接口，可以直接用Lua语言写路由逻辑。事实上，本书将主要介绍KEMI和Lua路由脚本。

Kamailio主要是一个代理服务器（Proxy），它不会主动发起呼叫，而是对呼叫SIP消息进行转发，因此不能“开箱即用”，你需要自己写一些转发逻辑。从另一个方面来讲，如果使用Kamailio，你必将会用到像FreeSWITCH或Asterisk那样的软件。简单来讲，Kamailio与FreeSWITCH最大的不同是——前者是一个代理服务器，而后者是一个B2BUA。

或许你已经了解了FreeSWITCH，事实上，本书中将会多次提到FreeSWITCH。虽然具备FreeSWITCH基础知识并不是阅读本书的必要条件，但如果你了解FreeSWITCH，那阅读本书会事半功倍。当然，本书也可以助你深入了解SIP，进而更了解FreeSWITCH。

本书面向的读者

在开始写作本书时，Kamailio刚刚庆祝完20周岁的生日，所以我希望以本书作为献给Kamailio的生日礼物，同时希望能帮助初学者快速掌握Kamailio，帮助资深运维和开发人员深入理解和灵活使用Kamailio。

具体来说，我希望本书能对以下人员有帮助。

1) FreeSWITCH从业者

这些人大部分应该已读过我的《FreeSWITCH权威指南》。相信本书能从另一个角度、另一个维度帮助他们理解SIP，理解通信逻辑。无论最终是否使用Kamailio，本书都会给他们带来帮助。即使他们自己不使用Kamailio，他们的对端也有可能在使用Kamailio，知己知彼，百战百胜；如果自己要使用Kamailio，阅读完本书自然能事半功倍。

2) VoIP系统、软交换系统、电信设备开发人员

这些开发人员必定会与SIP打交道，有的甚至要自主研发SIP协议栈和设备，这时就要与别人对接，Kamailio可以是一个很好的测试和验证平台。另外，他山之石可以攻玉，说不定看看Kamailio的源代码能得到很多启发。无论如何，参考Kamailio的KEMI，将Lua等嵌入式脚本语言融入这些电信设备，必将大大增加系统的可扩展性和兼容性。

3) Asterisk开发者



跟大多数Asterisk开发者一样，我也是读着《Asterisk，电话未来之路》《Trixbox不相信眼泪》一路走过来的。与FreeSWITCH类似，Asterisk也需要做集群，若做集群，则Kamailio是不二之选。

4) VoIP系统实施、维护人员

对于实施、维护呼叫中心、IP-PBX等系统的人员来说，本书也是不可多得的SIP教程。运维人员通常需要进行现场诊断、排错，还要分析SIP包等。扎实的SIP功底是高效做好这些事情的必要条件。Kamailio以及本书提到的一些SIP工具可以帮助模拟信令流程，进而帮助这些人更快地定位问题并排错。甚至，可以临时将Kamailio串联到系统中，处理不兼容设备间的信令适配问题。

5) 电信企业的维护人员、销售人员、决策人员

广大电信企业的人员在以往的工作中积累了大量的工作经验，但往往局限于华为、中兴等设备厂家提供的解决方案和技术架构。技术瞬息万变，在市场竞争日益激烈，国内电信政策调整并逐渐宽松之际（如虚拟运营商牌照的发放），只有了解另一种解题思路，掌握新技术，才能更好地把握市场方向，为客户提供更好的服务。

6) 电信企业的开发人员

现在，国内的电信企业内部都有自己的研究院，以支撑电信系统以及周边系统的选型与建设。在软件国产化、“信创”、自主可控的产业背景下，如果用到SIP处理，Kamailio当然是一个很好的选择。

7) 呼叫中心从业人员

可以预见，在不远的将来，将有很多呼叫中心是基于FreeSWITCH和Kamailio开发的。而本书中丰富的基础知识和详尽的功能介绍将对使用、管理呼叫中心系统起到很好的指导作用。

8) 在校教师和学生

大部分学校的教材只讲了VoIP原理及SIP，很枯燥。老师教育学生“要理论与实践相结合”，而本书正是理论与实践的最佳结合点。学生在本书中学到的知识和技能可以直接用在日后的工作中。

9) 互联网RTC从业人员

随着Web RTC的飞速发展，基于互联网的实时通信也发展迅猛，各种互联网教育、直播连麦、在线音视频会议等都会使用新兴的RTC技术。但Web RTC是一个媒体层的标准，没有规定信令，而Kamailio实现了SIP over WebSocket信令，支持浏览器中的WebRTC呼叫。另外，不管使用何种信令，基于互联网的RTC系统也难免与传统的通信系统对接，深入了解SIP才能更好地做好互联网与传统电信系统的无缝互联与融合。

10) 开发经理、技术决策人员

了解本书所讲的知识有助于技术选型和决策。本书虽然主要讲Kamailio，但对相关的技术和产品（如FreeSWITCH、OpenSIPS等）也都有对比和分析。全面了解各种技术有助于做出更好的决策。

11) OpenSIPS用户

Kamailio与OpenSIPS同根同源，很多概念和理论都是相通的。虽然本书的示例主要使用KEMI，在OpenSIPS中还没有对应的方法，但是，在呼叫流程的处理、SIP消息头域管理、号码变换、路由选择、负载均衡和高可用等方面都是相通的。事实上，KEMI在Kamailio中也是很新的概念，本书的示例也有很多是从原生脚本的示例中翻译过来的。如果读懂本书，就能将这些示例翻译回原生脚本，使其适用于Kamailio和OpenSIPS。

本书的内容及特色

本书从Kamailio的历史、基本概念和逻辑讲起，即使没有相关经验的读者也能轻松入门。如果读者还有一些通信相关的行业背景知识以及相关的计算机网络基础知识，读起来会更轻松。为了照顾对通信和SIP不太熟悉的读者，本书附带了大量的脚注信息和相关链接，供读者查阅。本书的附录部分也有对FreeSWITCH、Docker和Lua语言等相关基础知识的介绍，即使不熟悉这些内容的读者也能快速入门并无障碍地阅读本书。此外，如果你读过《FreeSWITCH权威指南》，你就有了阅读本书非常好的基础。

鉴于本书的章节安排，本书适合按从头到尾的顺序阅读。当然，所谓顺序阅读并不是需要逐字逐句阅读。尤其是第3章中列出的参数众多，初次阅读时观其大略即可。读完一遍后，再反复阅读某些章节，相信每次你都会有新的收获。

本书的主要特色是**KEMI**，以及通过**Lua**写路由脚本。**Lua**是一门轻量级的编程语言，非常适合写路由脚本。**Lua**除了用在Kamailio中外，还被广泛用在FreeSWITCH、Nginx（OpenResty）、PostgreSQL、VLC、Wireshark等知名软件中。

排版及约定

□有些代码行或日志输出较长，为适应版面，进行了人工换行和排版。

□提示符：对于命令行的输入输出来说，在Linux及Mac等UNIX类平台上，前面的\$或#为操作系统命令提示符；在不至于引起混淆的情况下，可能会省略系统提示符。

□注释：在Kamailio原生脚本中，统一使用#或//表示注释（有的情况下#也代表命令行提示符，请注意区分），而在Lua脚本和SQL语句中，使用--表示注释。

□本书给出的示例代码，如果在随书附赠的源代码中也有，一般会给出源代码文件名，如mtree.lua，以方便读者运行对照。

□为节省篇幅，本书的示例代码中加了大量的注释，这样处理可以使注释更便于阅读且更有针对性。所以代码内的注释也是本书很重要的部分。

资源和勘误

□kamailio.org是Kamailio社区官方网站，上面有各种资源。

□kamailio.org.cn是Kamailio中文网站，由我和SIP/VoIP专家James Zhu共同维护。

□book.dujinfang.com是本书的在线站点，提供本书的源代码下载及勘误服务等。

□dujinfang@gmail.com是我的电子邮箱，如果你对本书有任何意见、建议或批评，请发到该邮箱。

□<https://weibo.com/dujinfang>是我的个人微博，我很乐意与各位读者进行交流。

□RTS.cn是一个探讨开源和商业最佳结合的技术社区，有Kamailio、FreeSWITCH相关的技术交流以及年度技术论坛——RTSCon。

□FreeSWITCH-CN是FreeSWITCH中文社区的微信公众号，未来也会多发一些Kamailio的内容，欢迎大家搜索“FreeSWITCH-CN”关注。

由于我水平有限，书中错误和疏漏在所难免，欢迎广大读者朋友批评指正。

致谢

本书在写作时参考的大量资料都来自Kamailio开源社区及其官方网站上公开的信息，包括但不限于网页、演讲PDF、YouTube视频等，以及*SIP Routing with Kamailio*这本电子书[\[1\]](#)。Daniel-Constantin Mierla是Kamailio开发者之一，为本书的写作提供了很多指导和建议。Kamailio软件发展到今天，是德国的FhG FOKUS研究所、参与开源社区的众多个人开发者和公司共同努力的成果，在此一并致谢。

感谢机械工业出版社。机械工业出版社对中文原创计算机图书的支持让我倍感温馨。感谢杨福川编辑，他的图书出版理念给了我许多启发和写作的动力。感谢孙海亮编辑，他的耐心和细致保证了本书的质量和写作进度。

感谢我的妻子吕佳婧，她是本书的第二作者，也是本书的第一读者。她编辑整理了前三章和附录中的部分内容，也常常帮我修订文字错误。她基本上包揽了所有的家务和孩子的功课辅导，让我有更多的时间写作。

感谢我的儿子杜昱凝。他从很小就帮我测FreeSWITCH电话，现在已经是初中生的他也经常帮我调试Kamailio路由脚本并测试电话。也感谢他对我的理解，我总是忙于上班和写作，少了很多陪他玩耍的时间。

感谢我的同事韩小仿、景朝阳、杜林君、杨小金等，他们帮我提供了很多案例，做了很多测试和验证。感谢我的同事林彦君设计了本书封面。感谢烟台小樱桃网络科技有限公司，为本书的出版提供了人力和资金支持。没有他们，便没有此书。

杜金房
2022年5月于烟台

[1] 又称为*Kamailio Admin Book*，作者为Daniel-Constantin Mierla及Elena-Ramona Modroiu，参见<https://www.asipto.com/sw/kamailio-admin-book/>。

Chapter 1

第1章

Kamailio与SIP

Kamailio  是一个开源的SIP服务器，主要用作SIP代理服务器、注册服务器等，而

FreeSWITCH  是一个典型的SIP B2BUA，主要用于VoIP媒体相关的处理。

在学习FreeSWITCH以及SIP的过程中，经常有人问我：“SIP消息中那么多头域和参数，都是干什么用的？有些头域我从来也没有用过，是否真正有用？”我的回答是肯定的。FreeSWITCH只是一个应用场景，SIP是面向运营商设计的协议，在实际的部署环境中比单纯的FreeSWITCH要复杂得多。当然并不是所有人都能接触到运营商的环境，不过，现在是开源主导的世界，通过学习OpenSIPS或Kamailio，就可以更好地理解SIP了。

1.1 什么是Kamailio

Kamailio主要处理SIP，因此了解SIP对更快地学习Kamailio有很大帮助，而学好Kamailio又有助于进一步了解SIP，两者相辅相成。即使没有这两者太多的基础，相信通过本章的讲解，你也能初窥门径。

Kamailio基于GPLv2+开源协议发布，它可以支持每秒建立和释放成千上万次的呼叫（Call Attempt Per Second，CAPS），可用于构建大型的VoIP实时通信服务——音视频通信、状态呈现（Presence）、WebRTC、实时消息等；也可以构建易扩容的SIP-to-PSTN网关、IP-PBX系统，以及连接Asterisk、FreeSWITCH、SEMS等。

Kamailio具有如下特性：

- 支持异步的TCP、UDP、SCTP、TLS、WebSocket。
- 支持WebRTC，支持IPv4和IPv6。
- 支持IM消息及状态呈现。
- 支持XCAP和MSRP Relay。
- 支持异步操作。
- 支持VoLTE相关的IMS扩展。
- 支持ENUM、DID以及LCR路由。
- 支持负载均衡、主备用路由（Fail-Over）。
- 支持AAA（记账、鉴权和授权）。
- 支持很多SQL和NoSQL数据库后端，如MySQL、PostgreSQL、Oracle、Radius、LDAP、Redis、Cassandra、MongoDB、Memcached等。
- 支持消息队列，如RabbitMQ、Kafka、NATS等。
- 支持JSON-RPC、XML-RPC控制协议以及SNMP监控。



Kamailio从2001年开始开发^注，至今也有20余年的历史了。Kamailio的读法是Kah-Mah-Illie-Oh，或简单一点，Ka-Ma-ili-o，或Kama-ilio，谷歌翻译成“卡迈里奥”，但笔者觉得翻译成“卡马伊



里奥”或简称“卡马”^注更为合适。

Kamailio与FreeSWITCH配合使用最常用的场景是Kamailio作为注册服务器和呼叫负载均衡服务器（一般主备配置），FreeSWITCH进行媒体相关的处理（如转码、放音、录音、呼叫排队等），如图1-1所示。

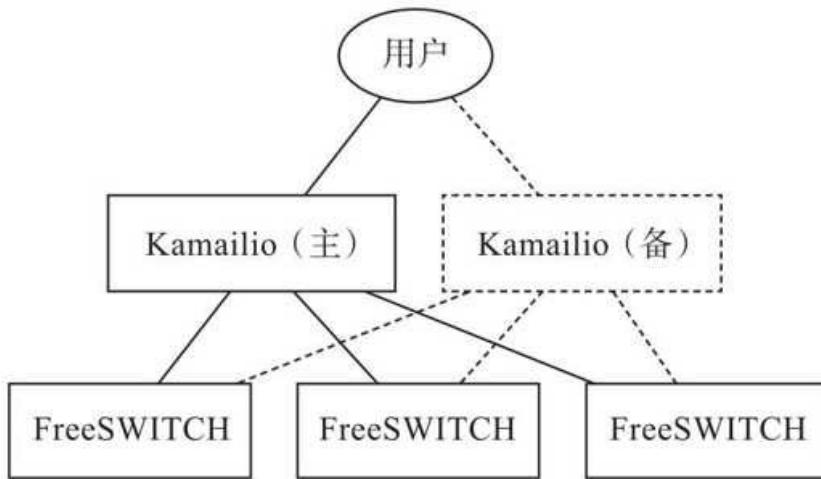


图1-1 Kamailio与FreeSWITCH配合使用

简单总结一下，Kamailio是一个：

- SIP服务器（SIP Server）。
- SIP代理服务器（SIP Proxy Server）。
- SIP注册服务器（SIP Registrar Server）。
- SIP地址查询服务器（SIP Location Server）。
- SIP重定向服务器（SIP Redirection Server）。
- SIP应用服务器（SIP Application Server）。
- SIP负载均衡服务器（SIP Load balance Server）。
- SIP WebSocket服务器（SIP WebSocket Server）。
- SIP SBC服务器（SIP SBC Server）。

相对而言，Kamailio不是：

- SIP软电话（SIP Phone）。
- 媒体服务器（Media Server）。
- 背靠背用户代理（Back-to-Back UA, B2BUA）。

它有以下特性：

- 快。
- 可靠。
- 灵活。

但它不做以下事情：

- 发起通话。
- 应答通话。

做音、视频等媒体处理。

1.2 背景

Kamailio起源于一个研究项目——SER。SER项目的全称是SIP Express Router，最早是由位于德国柏林的FhG FOKUS研究所开发的，并以GPL协议发布。核心研发人员有Andrei Pelinescu-Onciu、Bogdan-Andrei Iancu、Daniel-Constantin Mierla、Jan Janak以及Jiri Kuthan。2004年FhG FOKUS在SER的基础上启动了一个新项目IPtel（iptel.org），次年该项目的商业部分卖给了Tekelec，核心开发团队中部分成员去了iptel.org，而Bogdan和Daniel离开FhG FOKUS创建了一家新的公司Voice-



System，并开始维护开源版本的SER——OpenSER。后来，OpenSER分出两个项目



，一个是Kamailio，另一个是OpenSIPS。再后来，Kamailio与OpenSER项目合并，这使得Kamailio看起来更“正宗”一些。不管怎样，两者最初的代码都是一样的，但由于思路和方向不同，后来的版本差异就比较大了。本书中我们主要讨论Kamailio。下面是Kamailio的发展简史。

- 2001年9月，SER项目，Andrei Pelinescu-Onciu在德国FhG FOKUS研究所写下第一行代码。
- 2005年6月，分离为SER与OpenSER两个项目。
- 2008年8月，OpenSER分为Kamailio与OpenSIPS，首个Kamailio版本是1.4.0。
- 2008年11月，OpenSER与Kamailio代码合并，两者的模块可以通用，但分为不同的模块目录。
- 2009年3月，发布Kamailio v1.5.0，这是代码合并后的第一个大版本，该版本引入了很多新特性。
- 2013年3月，发布Kamailio v4.0.0，彻底整合了OpenSER的模块，使用同一个模块目录。
- 2017年2月，发布Kamailio v5.0.0，增加了KEMI支持，移除了MI控制接口，将相关功能统一到RPC管理接口。
- 2017年12月，发布Kamailio v5.1.0，增强了KEMI支持，增加了sipdump等9个新模块。
- 2018年11月，发布Kamailio v5.2.0，继续增强KEMI支持，增强了dispatcher模块，支持rtpengine转码功能，增加了acc_json等6个新模块。
- 2019年11月，发布Kamailio v5.3.0，继续增强KEMI支持，增强dispatcher模块，支持HAProxy协议，增加了rtp_media_server等6个新模块。
- 2020年7月，发布Kamailio v5.4.0，继续增强KEMI支持，支持STIR/SHAKEN，增加了JSON格式日志支持，增加了secsipid、kafka等5个新模块。
- 2021年5月5日，发布Kamailio v5.5.0，继续增强KEMI支持，增加rtpengine的Websocket控制接口支持，增加jwt等6个新模块。
- 2022年2月27日，发布Kamailio v5.5.4，本书开始写作时最新的版本，包含一些缺陷更新。
- 2022年5月，发布Kamailio v5.6.0，本书截稿时，该版本已冻结更新并准备发布。完整的特性列表尚未公布。当你拿到本书时，相信该版本就已经发布了。

Kamailio简史如图1-2所示。



1.3 SIP

SIP（Session Initiation Protocol，会话初始协议）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。

除此之外，还有大量相关的或是在SIP基础上扩展出来的RFC，如关于SDP的RFC 4566 、
关于会议的RFC 4579  等。

关于SIP，笔者找到对Henning Schulzrinne [\[1\]](#) 教授的一段采访 。采访中他回忆了SIP协议的诞生过程。

我最开始对音频和视频编码产生兴趣，是因为我的硕士论文与此有关。那时候PC声卡还没有广泛用于真正的音频采集，所以我只能在一台PDP-11/44迷你计算机上使用A/D转换器对信息进行转换。后来互联网和早期的SUN工作站（如SPARC）可以传输实时音视频了，我开始致力于开发和标准化用于在互联网上传输音视频的协议，如RTP和SIP。移动设备的广泛使用造就了“无处不在的系统”，包括现在广为人知的IoT。

我当时正在致力于一个被称为DARTnet的博士科研项目，这个项目可以在一个试验性的叠加网络中传输音频和视频，该网络当时使用了一个新的协议——ST-II（现在已弃用）。网络节点是早期可以实现网络路由器和叠加网络的SPARC工作站（来自SUN公司），这个系统需要配合相关工具和协议才能运行。我开始积极投入到音视频的各种技术工作中——从创造传输语音和视频的协议到开发支持流畅播放的算法。当时IETF为了支持组播骨干网（一种早期可以向千百个观众传输音视频的技术），开始开发所需协议，我也参与其中。这些工作促成了RTP的开发工作。RTP最初是针对组播设计的，后来被用于大多数“基于标准”的音视频传输，而不只限于组播。随着这项工作不断成熟，包括我在内参与这项工作的很多人认为，需要更好的方法来启动音视频会话，SIP因此诞生了。当时没人在意我们这个小组的研究，因为“真正的”电信工程师们正在研究时分复用电路交换机，而SIP是基于IP交换的。这反而成了我们的优势，没有人来指手画脚，我们的研究进程也较快。随着有线和无线行业开始认识到需要转入IP网络，SIP所需的底层协议标准实际上已经准备就绪了。

下面介绍SIP中一些基本概念。

1.3.1 SIP基础

SIP是一个基于文本的协议，从这方面来讲，SIP与HTTP、SMTP类似。一个典型的SIP请求如下：

```
INVITE sip:seven@xswitch.cn SIP/2.0
```

请求由三部分组成：INVITE表示发起一次呼叫请求；seven@xswitch.cn为请求的地址（Request URI，又称SIP URI或AOR（Adress of Record，用户的公开地址）；SIP/2.0是版本号。

SIP URI类似于一个电子邮件地址，其格式为“协议：名称@主机”。“协议”有SIP和SIPS（后者用于安全通信，如sips:seven@xswitch.cn）两种；“名称”可以是一串数字形式的电话号码，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。

一个SIP请求会得到一个响应，响应消息的第一行如下所示：

SIP/2.0 200 OK

其中中间部分为状态码，由三位数字构成：

- 1xx表示临时响应。
- 2xx表示成功响应。
- 3xx表示重定向。
- 4xx表示客户端引起的错误（如请求一个不存在的地址时就会收到著名的404状态码）或需要客户端进一步提供的认证信息（如401等）。
- 5xx表示服务器端的错误（服务器脚本出错等）。
- 6xx表示全局错误。

其中临时响应（1xx）不是必需的，其他所有的响应都称为最终响应，每一个SIP请求都应该有一个最终响应。

起始行以下为SIP消息头，如From和To等。有些SIP消息（如INVITE、200 OK）中会有更进一步的描述信息，这类信息称为正文（Body）。Body是可选的，并不是所有的请求或响应中都有Body字段。如果Content-Length为0或不存在，就没有Body。消息头中的Content-Length表示正文的长度。在所有的SIP头域结束后，会有一个空行，它标志着SIP头部的结束及消息正文的开始。

熟悉HTTP的读者会发现，SIP其实与HTTP类似。事实上SIP就是参考HTTP设计的，重用了很多HTTP中的内容，如Digest（摘要）认证等。HTTP是互联网最重要的协议，而SIP也在通信领域很快替代了二进制的H323协议成为主流。

1.3.2 SIP的基本概念和相关元素

SIP是一个对等的协议，类似于P2P。不像HTTP那样是“客户端-服务器”的结构，也不像传统电话那样必须有一个中心的交换机，SIP甚至可以在不需要服务器的情况下进行通信，只要通信双方都知道对方的地址（或者只有一方知道另一方的地址）。如图1-3所示，Bob拿起话机给Alice发送一个INVITE请求，说“一起吃饭吧”，Alice说“好的”，电话就接通了。

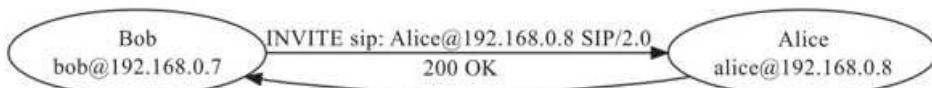


图1-3 SIP点对点通信



在SIP网络中，Alice和Bob的话机都称为UA。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，如图1-3中所示的Bob。另一种是UAS（UA Server，用户代理服务器端），它是接受请求并发送响应的一方，如图1-3中所示的Alice。由于SIP是对等的，所以当Alice呼叫Bob时（有时候Alice也会主动约Bob一起吃饭），Alice的话机就称为UAC，而Bob的话机会执行UAS的功能。一般来说，UA都会实现上述两种功能。

设想Bob和Alice是经人介绍刚刚认识的一对恋人。因为他们彼此还不熟悉，所以Bob想请Alice吃饭还需要一个中间人（M）传话，而这个中间人就称为代理服务器（Proxy Server），如图1-4所示。还有另一种中间人称为重定向服务器（Redirect Server），它以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，但我爱人知道，要不然我告诉你我爱人的电话，你直接问她吧，我爱人叫W。这样，M就成了一个重定向服务器（把Bob对他的请求重定向到W，这样Bob接下来要直接联系W），而W是真正的代理服务器。这两种服务器都是UAS，它们主要为一对欲通话的UA提供路由选择功能，如图1-5所示。

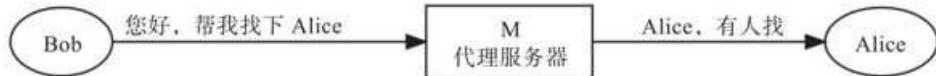


图1-4 代理服务器示意图

还有一种被称为注册服务器的UAS。试想这样一种情况：Alice还是个学生，没有自己的手机，但她又希望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，如果有事找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，哪天去姥姥家了，Alice也要把姥姥家的电话告诉M。总之，只要Alice换一个新的位置，它就要向M重新“注册”新位置的电话号码，以便M能随时找到她，这时候M就相当于一个注册服务器。注册服务器的另一个功能是“寻址”，比如Bob想要找Alice，那么他就要问M，用哪个电话号码可以联系到她，这时候M起的作用就是寻址，如图1-6所示。

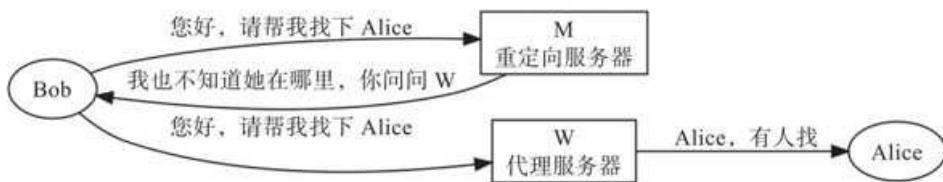


图1-5 重定向服务器示意图

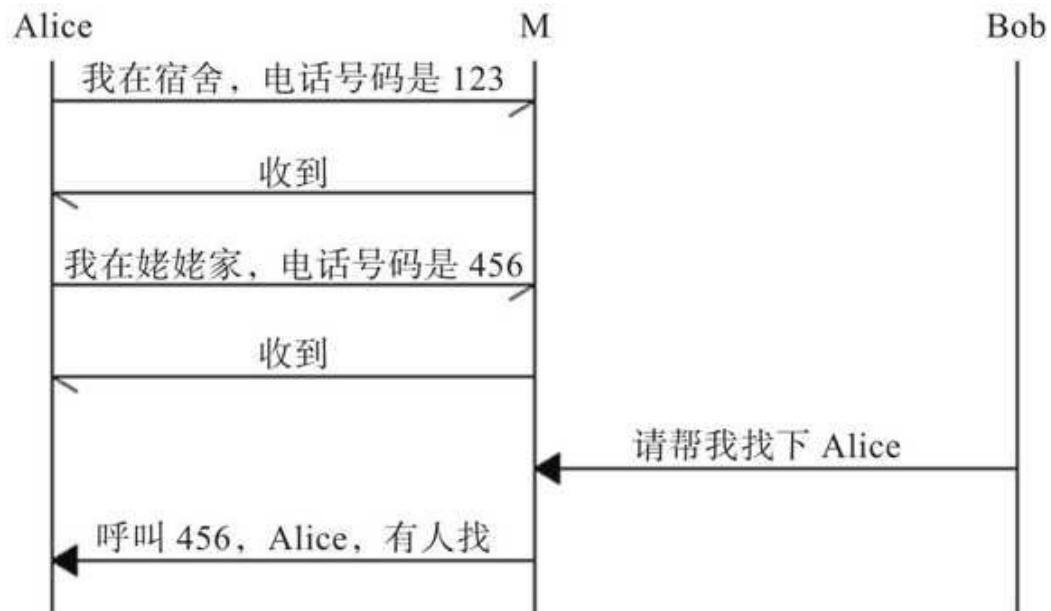


图1-6 注册服务器、寻址服务器示意图

还有一种特殊的UA称为B2BUA。需要指出，其实RFC 3261并没有定义B2BUA的功能，它只是实现一对UAS和UAC的串联。我们前面提到的FreeSWITCH就是一个典型的B2BUA。

此外，还有一个概念——边界会话控制器（Session Border Controller，SBC）。它主要位于一堆SIP服务器的边界，用于打通内外网的SIP通信、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。其应用位置和拓扑结构如图1-7所示。

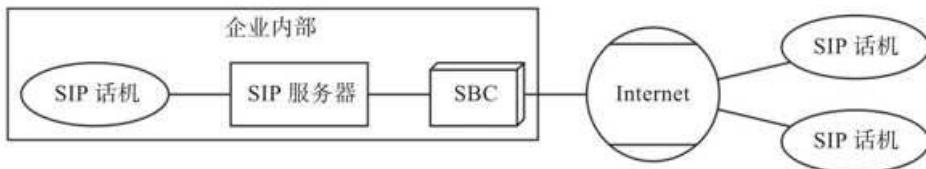


图1-7 SBC位置示意图

与FreeSWITCH相比，Kamailio是一个典型的代理服务器。不过，Kamailio也可以做注册服务器、SBC等。一般来说，Kamailio只处理SIP，但在某些场景中，如NAT穿越、拓扑隐藏等，也会配合MediaProxy或rtpengine（这两个都是媒体代理）一起工作。

1.3.3 SIP的基本方法和头域

SIP定义了6种基本方法，如表1-1所示。

表1-1 SIP的基本方法

基本方法	说 明
REGISTER	注册联系信息
INVITE	初始化一个会话
ACK	对 INVITE 消息的最终响应的证实
CANCEL	取消一个等待处理或正在处理的请求
BYE	终止一个会话
OPTIONS	查询服务器能力和，也可以用作 ping 测试

除此之外，SIP还定义了一些扩展方法，如SUBSCRIBE、NOTIFY、MESSAGE、REFER、INFO、

注
PRACK 等。

另外，无论是基本方法还是扩展方法，所有SIP消息都必须包含表1-2所示的6个头域。

表1-2 SIP消息必备头域

名 称	描 述
Call-ID	用于区分不同会话的唯一标志
CSeq	顺序号，用于在同一会话中区分事务
From	说明请求来源
To	说明请求接受方
Max-Forwards	限制跳跃点数和最大转发次数
Via	描述请求消息经过的路径

1.3.4 SIP URI

如图1-8所示，192.168.1.9是Kamailio服务器，而Bob和Alice分别在另外两台机器上。

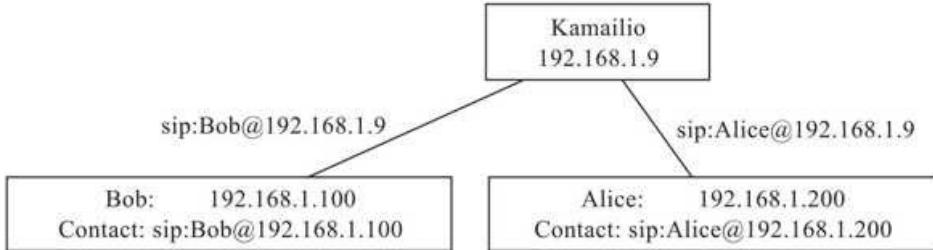


图1-8 Bob和Alice分别在另外两台机器上的情况

在图1-8所示情况下，Alice注册到Kamailio，Bob呼叫她时，使用她的服务器地址（因为Bob只知道服务器地址），即sip:Alice@192.168.1.9。Kamailio接到SIP呼叫请求后，查找本地数据库，发现Alice的实际地址（Contact地址，即联系地址）是sip:Alice@192.168.1.200，进而建立呼叫。

SIP URI除使用IP地址外，也可以使用域名，如sip:Alice@example.com。域名将使用DNS的A记录（对于IPv4）或AAAA记录（对于IPv6）进行查询，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。

这里再重复一下，Bob呼叫Alice时，Bob是主叫方，他已经知道服务器的地址，因此可以直接给服



务器发送INVITE消息，因而他是不需要注册的。而Alice不同，她是作为被叫的一方，为了让服务器能找到她，她必须事先通过REGISTER消息将自己“注册”到服务器上。

1.3.5 SDP和SOA

SIP负责建立和释放会话，一般来说，会话会包含相关的媒体，如视频和音频。媒体数据是由SDP（Session Description Protocol，会话描述协议）来描述的。SDP一般不单独使用，它与SIP配合使用时会放到SIP的正文（Body）中。

会话建立时，需要媒体协商，这样双方才能确定对方的媒体能力以交换媒体数据。Kamailio不处理媒体，但有时也可以配合rtpengine等做媒体代理、实现转码、完成NAT穿越等。在此，我们通过一个简单的FreeSWITCH例子介绍一下SDP是如何工作的。

我们来看一个FreeSWITCH参与的单腿呼叫的例子。客户端607呼叫FreeSWITCH默认的服务echo，它是一个回声服务，呼通后，主叫用户不仅能听到自己的声音，还能看到自己的视频（如果有的话）。为了更直观一些，我们使用Wireshark进行抓包和分析。图1-9显示了该SIP呼叫的流程。

由图1-9可知，客户端（192.168.1.118）呼叫FreeSWITCH（192.168.1.9），INVITE中带了SDP消息。其认证过程与我们上面讲到的类似。最后，FreeSWITCH回复200 OK对通话进行应答，然后双方互发RTP媒体流（G711A，即PCMA的音频和H264的视频）。

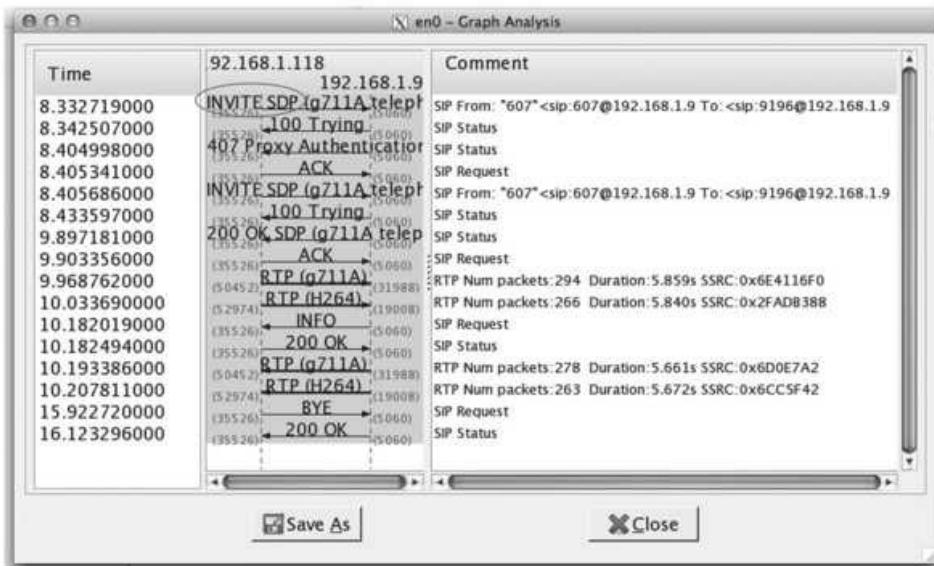


图1-9 带SDP的SIP呼叫

在图1-9中还可以看出，客户端的SIP端口号是35526，音频端口号是50452，视频端口号是52974；FreeSWITCH的端口号则分别是5060、31988和19008。到后面我们会在SIP消息中找到这些。

下面是一个完整的SIP INVITE消息：

```

recv 921 bytes from udp/[192.168.1.118]:35526
-----
INVITE sip:9196@192.168.1.9 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.118:35526;branch=z9hg4bK-d8754z-0a09c74c6345dc09-1--d8754z;rport
Max-Forwards: 70
Contact: <sip:607@192.168.1.118:35526>
To: <sip:9196@192.168.1.9>
From: "607"<sip:607@192.168.1.9>;tag=f49f383a
Call-ID: ZTQON2Y2NzI2ZjMxZTcwZTY0YTA5ODUyZDUzNRM2YjM
CSeq: 1 INVITE
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
Content-Type: application/sdp
Supported: replaces
User-Agent: Bria 3 release 3.5.0b stamp 69410
Content-Length: 381

v=0
o=- 1371880105304943 1 IN IP4 192.168.1.118
s=Bria 3 release 3.5.0b stamp 69410
c=IN IP4 192.168.1.118
b=AS:2064
t=0 0
m=audio 50452 RTP/AVP 8 0 98 101
a=rtpmap:98 ILBC/8000

a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=sendrecv
m=video 52974 RTP/AVP 123
b=TIAS:2000000
a=rtpmap:123 H264/90000
a=fmtp:123 profile-level-id=428014;packetization-mode=0
a=rtcp-fb:* nack pli
a=sendrecv

```

对于SIP头部我们前面已经了解得差不多了。其中的Content-Length跟HTTP中的类似，表示正文的长度。这里的正文类型是用Content-Type表示的，在这里它是application/sdp，表示正文中是SDP消息。同样，一个空行把SIP头部（Header）与SIP正文（Body）部分隔开。（SIP头部的结束是以“\r\n\r\n”为标志的。）

下面我们主要讨论SDP部分。

v (Version) , 表示协议的版本号。

o (Origin) , 表示源。各项的含义依次是username、sess-id、sess-version、nettype、addrtype、unicast-address。

s (Session Name) , 表示本SDP所描述的Session的名称。

c (Connection Data) , 连接数据。两个字段分别是网络类型和网络地址，以后的RTP流就会发到该地址上。注意，在NAT环境中我们要解决透传问题，就是要看这个地址，这在后文中也会讲到。

b (Bandwidth Type) , 带宽类型。

t (Timing) , 起止时间。0表示无限。

m=audio (Media Type) , 媒体类型。audio表示音频，50452表示音频的端口号，应该跟图1-9所示一致；RTP/AVP是传输协议，这里是RTP；后面是支持的Codec类型，与RTP流中的Payload Type（负荷类型）相对应，在这里分别是8、0、98和101。8和0分别代表PCMA和PCMU，它们属于静态编码，因而有一一对应的关系。而对于大于95的编码都属于动态编码，需要在后面使用“a=”进行说明。

a (Attributes) , 属性，用于描述上面音频的属性，如本例中98代表8000Hz的ILBC编码，101代表RFC2833 DTMF事件。a=sendrecv表示该媒体流可用于收和发，其他的还有sendonly（仅收）、recvonly（仅发）和inactive（不收不发）。

m=video (Media Type) , 媒体类型。video表示视频。可以看出它的端口号52974也跟图1-9所示一致。而且H264的视频编码对应的也是一个动态Payload Type，在本例中是123。

FreeSWITCH收到上述的请求后，进行编码协商。这里我们省去SIP交互的中间环节，直接看200（应答）消息：

```
send 1255 bytes to udp/[192.168.1.118]:35526
-----
SIP/2.0 200 OK

Via: SIP/2.0/UDP 192.168.1.118:35526;branch=z9hG4bK-d8754z-39cda633284f4f26-1---
d8754z-;rport=35526
From: "607"<sip:607@192.168.1.9>;tag=f49f383a
To: <sip:9196@192.168.1.9>;tag=9UXRpKBrZrc4N
Call-ID: ZTQ0N2Y2NzI2ZjMxZTcwZTY0YTA5ODUyZDUzNWM2Yjm
CSeq: 2 INVITE
Contact: <sip:9196@192.168.1.9:5060;transport=udp>
User-Agent: FreeSWITCH-mod_sofia/1.3.17+git-20130329t031728z~aca9257f93
Accept: application/sdp
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, INFO, UPDATE, REGISTER, REFER,
NOTIFY, PUBLISH, SUBSCRIBE
2013-06-22 13:48:26.932214 [DEBUG] mod_erlang_event.c:156 Sending event CHANNEL_
CALLSTATE to attached session 5b10acf6-daff-11e2-8a9b-a577a8aef831
Supported: timer, precondition, path, replaces
Allow-Events: talk, hold, conference, presence, dialog, line-seize, call-info,
sla, include-session-description, presence.winfo, message-summary, refer
Session-Expires: 120;refresh=ua
Min-SE: 120
Content-Type: application/sdp
Content-Disposition: session
Content-Length: 297
Remote-Party-ID: "9196" <sip:9196@192.168.1.9>;party=calling;privacy=off;screen=no

v=0
o=FreeSWITCH 1371848118 1371848119 IN IP4 192.168.1.9
s=FreeSWITCH
c=IN IP4 192.168.1.9
t=0 0
m=audio 31988 RTP/AVP 8 101
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=silenceSupp:off - - -
a=ptime:20
m=video 19008 RTP/AVP 123
a=rtpmap:123 H264/90000
```

SIP头域我们就不多讲了，列在这里只是为了让消息完整。下面直接看200返回的SDP数据，我们也能找到音视频的IP地址是192.168.1.9，端口号分别是31988和19008。该SDP也携带了FreeSWITCH协商后的编码PCMA（8）以及a=ptime项，ptime表示RTP数据的打包时间，其实这里也可以省略，默认就是20（毫秒）。至此，双方都有了对方的RTP地址和端口信息，它们就可以互发RTP流了。



媒体流的协商过程称为SOA（Service Offer and Answer，提议/应答），即首先有一方提供它支持的Codec类型，另一方基于此进行选择。如本例中，607先提议：“我支持PCMA、PCMU和ILBC编码，你看咱俩用哪种通信比较好？”FreeSWITCH回复说：“那我们就用PCMA吧。”然后双方就可以互发RTP流进行媒体交换了。当然，根据现有的媒体协商标准，FreeSWITCH也可以说：“我支持PCMA和PCMU两个编码，随便你发，用哪个都行。”在这种情况下，双方就必须准备好能收发两种编码的RTP流，不管对方用哪个发，都必须能正确接收。不过，到目前为止，FreeSWITCH还不支持同时回复两个编码，但是如果FreeSWITCH是请求方，对方回复了两种编码，FreeSWITCH是可以正确支持的。虽然应答方回复多个编码会增加复杂性，但标准就是这么规定的。

1.3.6 SIP承载



大家已经熟知，HTTP是用TCP承载的，而SIP支持TCP和UDP承载（当然也支持TLS等其他承载方式）。事实上，RFC 3261规定，任何SIP UA必须同时支持TCP和UDP。我们常见的SIP都是用UDP承载的。由于UDP是面向无连接的，故在大并发量的情况下与TCP相比，可以节省TCP由



于每个IP包都需要确认带来的额外开销。不过，在SIP包比较大的情况下，如果超出了IP层的最大传输单元（MTU，即Maximum Transmit Unit，通常最大是1500字节）的大小，在经过路由器时可能会被拆包，使用UDP承载的SIP消息就可能会发生丢失、乱序等，这时候就应该使用更可靠的传输层协议TCP。



在需要对SIP加密的情况下，可以使用TLS。TLS是基于TCP实现的。



在新的网络时代，又出了一个新的草案，名为SIP over WebSocket。当前，主流浏览器如Chrome和Edge已经实现了WebSocket，从而可以通过它承载SIP；而这些浏览器大多也实现了



WebRTC，这意味着它们可以通过Web浏览器与普通的SIP话机（甚至PSTN）进行音视频通话。SIP over WebSocket的承载为SIP/WS或SIP/WSS，其中后者是基于TLS实现的。WebRTC必须加密后才能传输，所以网上实际在用的信令协议都是SIP/WSS。

1.3.7 事务、对话和会话

Kamailio在大多数情况下都被用作SIP代理（SIP Proxy），典型的应用场景是处理用户注册、呼叫路由、负载均衡等。要理解SIP代理，我们还需要进一步理解如下概念。

1.事务

事务（Transaction）是指一个请求消息以及这个请求对应的所有响应消息的集合。对于INVITE事务来讲，除包含INVITE请求和对应的响应消息外，在非成功响应的情况下，还包括ACK请求。Via头域中的branch参数能够唯一确定一个事务。branch值相同的，代表为同一个事务。事务是由方法（事件）来引起的，一个方法（Method）的建立和到来都将建立新的事务。实际上当收到新消息时，就是根据branch来查找对应事务的。

一个事务由5个必要部分组成：From、To、Via头域中的branch参数、Call-ID和CSeq。这5个部分一起识别某一个事务，如果缺少任何一部分，该事务就会设置失败。事务是逐一跳转（Hop by Hop，每一个路由节点称为一跳，即一个Hop）的关系，即路由过程中交互的双方包括一个请求及其触发的所有响应（即若干临时响应和一个最终响应）。事务的生命周期用于表示从请求产生到收到最终响应的完整周期。

2.对话

对话（Dialog）是两个UA之间持续一段时间的点对点的SIP连接，它使UA之间的消息变得有序，同时给出请求消息的正确的路由。Call-ID、from-tag以及to-tag三个值的组合能够唯一标识一次对话。对话只能由INVITE或SUBSCRIBE来创建。

对话是点到点（Peer to Peer）的关系，即真实的通信双方，其生命周期贯穿一个点到点会话的始终。

3.会话

会话（Session）是一次通信过程中所有参与者之间的关联关系以及它们之间的媒体流的集合，是端到端的。只有当媒体协商成功后，会话才能被建立起来。

如图1-10所示，根据前面的描述，图中有1个SIP对话和3个事务。从INVITE到200 OK是一个事务，从BYE到200 OK则是另一个事务。ACK是一个单独事务。在这个场景中，会话和对话是重合的。

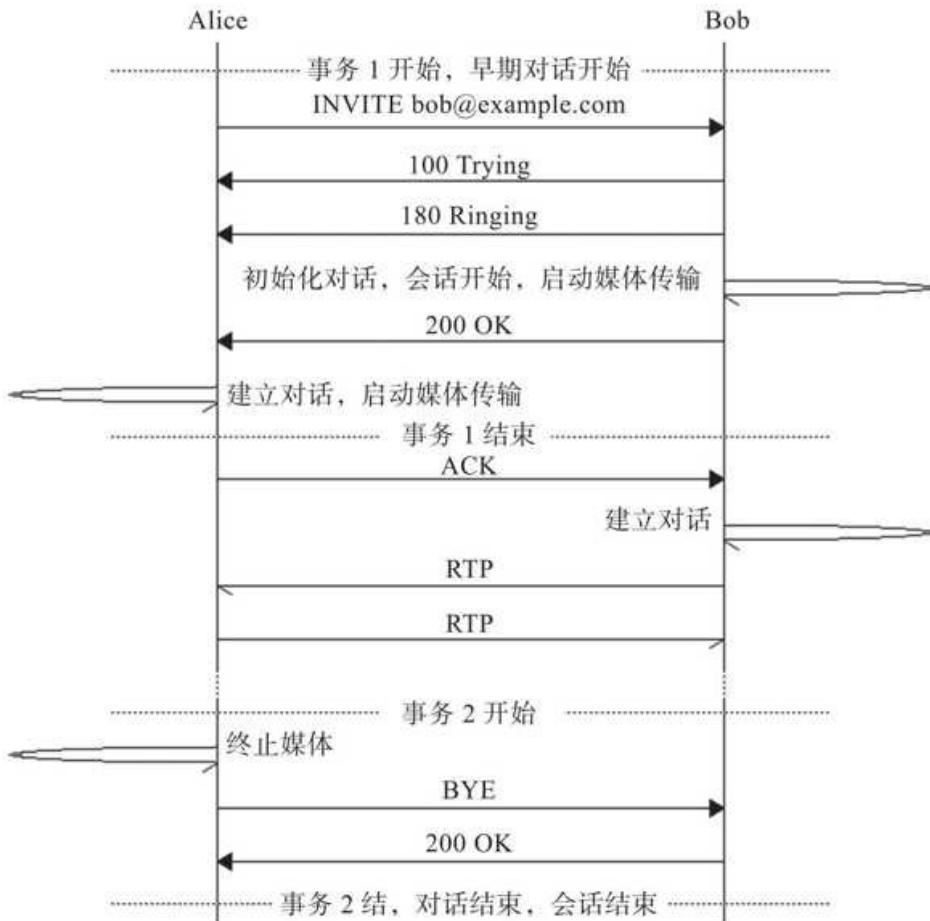


图1-10 事务与对话关系图

图1-11描述的是两个UA经过代理服务器转发的情况。事务和对话只存在于直接相连的UA间，而会话是端到端的——Alice和Bob之间的通话是一个会话。

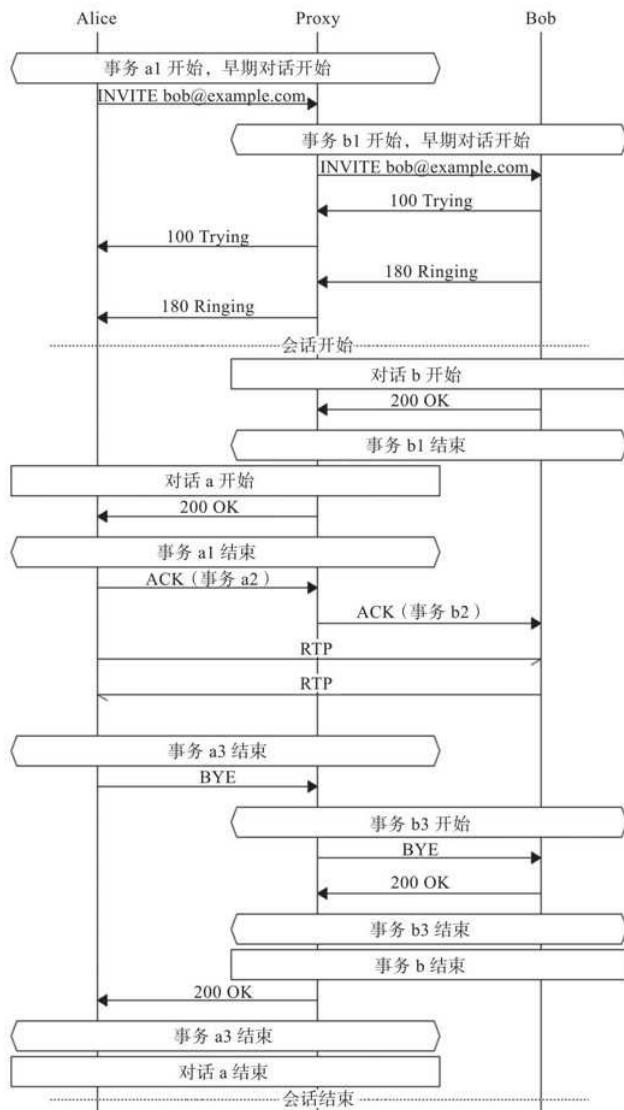


图1-11 事务、对话与会话关系图

4.CSeq

CSeq的生存期是一个会话。CSeq用于将一个会话中的请求消息进行序列化，以便对重复消息、“迟到”消息进行检测，以及对响应消息与相应请求消息进行匹配。它包含两部分：一个32位的序列号，一个请求方法。

通常在会话开始时确定一个初始值，其后在发送消息时将该值加1。主叫方与被叫方各自维护自己的CSeq序列，互不干扰。CSeq序列有点像TCP/IP中IP包的序列。

一个响应消息有与其对应的请求消息相同的CSeq值。

注意：SIP中CANCEL消息与ACK消息是比较特殊的。CANCEL消息的CSeq中的序列号总是跟其将要撤销（Cancel）的消息相同，而对于ACK消息，如果它所要确认的INVITE请求是非2xx响应，则ACK消息的CSeq中的序列号与对应INVITE请求的相同；如果是2xx响应，则不同，此时ACK被当作一个新的事务。

Call-ID、from-tag以及to-tag这三个值相同代表是同一个对话；branch值相同代表是同一个事务，否则代表不同的事务。

1.3.8 Stateless与Stateful

作为一个代理服务器，关键的作用就是路由SIP消息，即控制SIP消息从哪里来、到哪里去。当然，如果有必要的话，可以在中间修改SIP消息。

SIP代理服务器有两种工作状态——Stateless与Stateful，即无状态和有状态。在无状态下，代理服务器只是机械地路由消息，将收到的消息根据一定的规则转发到下一跳，它不关心会话、对话和事务。在这种情况下代理服务器不会维护状态机，因而比较轻量级，但同时，对于错误处理和计费应用来讲会有诸多限制。

在有状态的情况下，代理服务器在收到请求消息（如INVITE）时会启动一个状态机，跟踪一个事务，一直到收到200 OK或其他最终响应。所以，如果一个代理服务器在收到200 OK消息时知道与之关联的INVITE消息，那么该代理服务器就是有状态的。

在Kamailio中，无状态模式使用forward()转发消息，而有状态模式使用t_relay()转发，且可以在onreply_route()中处理响应消息。

在有状态的情况下，状态只维护在一个事务内，而不是整个对话。即状态只维护在从收到INVITE消息到200 OK消息的过程中，而不是在从INVITE到BYE的过程中。

有状态模式适合处理更复杂的应用，如语音信箱、会议、呼叫转移、计费等。

在Kamailio中，有状态模式的处理一般分为以下几个步骤。

(1) 验证请求合法性。

检查消息大小是否超长，消息是否完整。

检查Max-Forward头域，看是否有循环请求。

(2) 路由消息预处理。如果有Record-Route字段则对其进行处理。

(3) 确定处理请求目的地时是否涉及如下问题。

目标是本地注册用户吗？（可以在本地数据库中查到。）

本机是最终目的地吗？

是否需要转发到外部的域（其他服务器）？

(4) 消息转发。调用t_relay()进行转发。Kamailio将会自动处理所有与状态相关的工作，如重发等。

(5) 响应处理。如果收到响应消息，则进行处理，一般情况下这些都是自动完成的，但也可以在onreply_route()里进行处理，如“遇忙转移”业务，可以在收到“486 Busy Here”消息时，转到另一个号码或进入语音信箱进行处理。

1.3.9 严格路由和松散路由

Strict Router和Loose Router分别称为严格路由和松散路由。松散路由是SIP Version 2中才有的概念。

我们可以看到，在Router字段中设置的SIP URI经常有一个lr的属性，例如<sip:example.com;lr>，这就是表示这个地址所在的代理服务器是一个松散路由，如果没有lr属性，它就是一个严格路由。

松散路由实际上表示代理服务器依据RFC 3261处理Route字段的规则，而严格路由表示Proxy Server根据RFC 2357处理Route字段的规则。严格路由要求SIP消息的Request URI为其自身的地址。具体步骤如下。

(1) 松散路由和严格路由首先都会检查Router字段的第一个地址是否为自己，如果是，则从Router字段中删除自己。

(2) 严格路由在发往下一跳时将使用Router字段中的下一跳地址更新Request URI。

(3) 松散路由首先会检查Request URI是否为自己。如果不是，则不做处理；如果是，则取出Route字段中最后一个地址作为Request URI地址，并从Route字段中删去最后一个地址。

(4) 松散路由还会检查下一跳是否为严格路由。如果不是，则不做处理；如果是，则将Request URI添加为Route的最后一个字段，并用下一跳严格路由的地址更新Request URI。

由上可见，后面两步其实是松散路由为了兼容严格路由而做的额外工作。两者最大的区别体现在Request URI会不会变，如图1-12及图1-13所示。

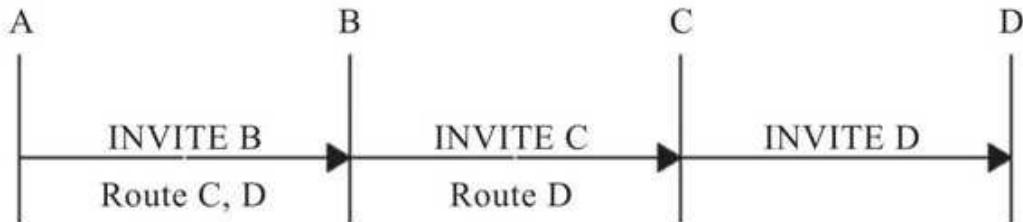


图1-12 严格路由



图1-13 松散路由

1.3.10 Record-Route

当一个代理服务器收到一个SIP消息时，它可以决定是否留在SIP传输的路径上，即后续的SIP消息是否还要经过它。比如在A呼叫B时，如果代理服务器只起到“找到B”的作用，则它可以将第一个消息原样传送，B回送的消息将可以不经过代理服务器而直接回到A上，这种方式称为Forward，如图1-14所示。



图1-14 Forward示意图

如果代理服务器想保留在SIP路径上，则它在将消息转发到下一跳之前要把它自己的地址加到Record-Route头域中。那么，当B在回复响应消息的时候，就会将消息发回到Record-Route指定的地址上，这种方式称为Relay，如图1-15所示。

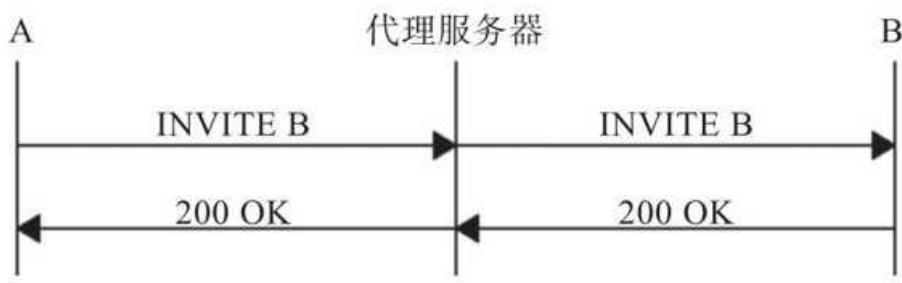


图1-15 Relay及Record-Route示意图

有了上述基础知识，下面我们就可以看看Kamailio的应用了。事实上，这些基础知识略显枯燥，也不是那么容易懂，大家可以先学习后面的内容，再回过来复习这部分，或许更有助于理解。

1.4 Kamailio基本架构

类似于FreeSWITCH，Kamailio也是由核心和可加载模块组成的。Kamailio的核心非常短小精悍，负责基本的SIP消息处理，而模块则扩展了核心的功能。Kamailio的模块实现了一些命令和函数，可以在配置文件（或称脚本）中使用，而配置文件则是这些命令和函数的黏合剂，实现相应的业务逻辑。

Kamailio的配置文件默认为kamailio.cfg，由以下几部分组成。

- 全局定义： 配置全局参数，如日志、调试级别、监听的IP地址和端口等。全局参数影响Kamailio核心以及所有模块。
- 模块： 模块使用loadmodule指令加载，加载后就可以使用模块里面的函数。
- 模块配置： 可以使用modparam()函数配置模块的参数，如“modparam()模块名，参数名，参数值”。
- 主路由块： 处理SIP请求，是最先接触到SIP消息的地方。
- 次级路由块： 类似于子函数，可以使用route()命令定义其他路由块。
- 回复路由块： 用于处理临时或最终响应的SIP消息（如200 OK等）。
- 失败路由块： 用于处理失败或异常，如忙或超时等。
- 分支路由块： 在对SIP进行Fork操作的时候，处理每个分支的逻辑。
- 本地路由块： 用于在Kamailio内部产生一条通过TM模块主动发送的消息（仅在作为UAS时）。

配置文件使用类似C语言的语法实现，示例如下（仅作为示例，并不是完整的配置）：

```
##### 全局参数 #####
### LOG Levels: 3=DBG, 2=INFO, 1=NOTICE, 0=WARN, -1=ERR
#ifndef WITH_DEBUG    # 预处理指令, 如果设置了该值, 则下面的配置生效, 否则 else 后面的配置有效
debug=3              # 设置调试级别
log_stderror=yes    # 是否将日志输出到标准错误上
#else
debug=2
log_stderror=no
#endif
memdbg=5            # 内存调试级别
memlog=5            # 内存日志级别

##### 模块 #####
# mpath="/usr/local/lib/kamailio/modules/"          # 模块路径
loadmodule "tm.so"                                     # 加载 tm 模块

# ----- tm 模块相关的参数 -----
modparam("tm", "failure_reply_mode", 3)
modparam("tm", "fr_timer", 30000)
modparam("tm", "fr_inv_timer", 120000)

##### 路由逻辑 #####
# 主路由, 收到消息后最先执行这里
request_route {
    # 初始化, 具体的路由块在后面, 相当于一个函数调用
    route(REQINIT);
    # NAT 检测
```

```

route(NATDETECT);

# CANCEL 处理
if (is_method("CANCEL")) {
    if (t_check_trans()) {
        route(RELAY);
    }
    exit;
}

# 处理对话内的 SIP 消息
route(WITHINLG);

# 只有初始请求会调用这些(没有 To tag 的情况)
if(t_precheck_trans()) {           # 处理重传
    t_check_trans();
    exit;
}
t_check_trans();

# 墓权
route(AUTH);

# 对于对话级的请求，添加 Record-Route 头域

remove_hf("Route"); # 先删除 Route 头域
if (is_method("INVITE|SUBSCRIBE")) # 仅对这两个方法有效
    record_route();      # 添加 Record-Route 头域

if (is_method("INVITE")) { # 仅对 INVITE 请求记账
    setflag(FLT_ACC); # 设置记账标志
}

route(SIPOUT);      # 将请求路由到外部 SIP 服务
route(REGISTRAR);   # 处理注册消息
route(LOCATION);    # 呼叫本地注册用户
}

# 次路由，接力转发 SIP 消息
route[RELAY] {
    if (!t_relay()) {
        si_reply_error();
    }
    exit;
}

# 对每个 SIP 请求进行初始化检查
route[REQINIT] {

}

# 处理对话内的 SIP 消息
route[WITHINLG] {

}

# 处理 SIP 注册
route[REGISTRAR] {

```

```
}

# 呼叫本地注册用户
route[LOCATION] {
}

# IP 墓权
route[AUTH] {
}

# NAT 检测
route[NATDETECT] {
}

# NAT 处理
route[NATMANAGE] {
}

# 对话内的 URI 相关处理
route[DLGURI] {
}

# 路由到其他 SIP 服务器
route[SIPOUT] {
}

# 分支路由，管理外呼的分支呼叫
branch_route[MANAGE_BRANCH] {
}

# 回复路由，处理对方回复的消息
onreply_route[MANAGE_REPLY] {
}

# 失败路由，失败时调用
failure_route[MANAGE_FAILURE] {
}
```

从这里可以看出，写Kamailio的配置文件基本上相当于用C语言写程序，所以，不仅需要懂SIP，还需要懂Kamailio的各种路由逻辑，即需要学习这门新的配置“语言”，这对维护人员或程序员来说要求还是比较高的。如果本书的名字叫“Kamailio从入门到放弃”，那本书到这里就可以结束了。但如果你不言放弃，那么请继续学习，后面还有更有趣的内容等着你。

[1] Henning Schulzrinne教授是多媒体领域的先驱人物。他是IEEE通信协会互联网技术委员会的联合主席，同时也是*Journal of Communications and Networks* 的编辑。他领导和开发了VoIP协议，并为SIP、RTP和RTSP等多媒体领域中的关键网络传输协议做出了重要贡献。他已发表250余篇期刊会议论文，以及70多个RFC标准。

Chapter 2

第2章

理解Kamailio配置文件

为了能直观地了解Kamailio，本章我们通过一个示例来讲解配置文件。



在Kamailio源代码的misc/examples/目录下，有一些配置文件，我们选取kemi目录下的配置文件作为例子进行讲解。

Kamailio从5.0版本开始支持KEMI（参见第4章），除了原生的配置方式外，还支持使用一些主流的脚本语言（如Lua、JavaScript、Python等）对路由进行配置。原生的配置方式只能调用Kamailio本身实现的函数和逻辑，而通过其他脚本语言进行配置，可以调用更多脚本语言的特性。下面先看一些例子。

2.1 基本配置文件

下面的例子来自Kamailio-basic-kemi.cfg，为节省篇幅，我们做了一些删减，并添加了一些注释，以方便读者阅读。有些重要的功能和模块我们还会在后面进行更详细的讲解，因此在此仅做简要注释，并不深入展开。

在配置文件中，行内以#开头的是注释，但以#!开头的是真正有效的内容，不是注释。配置文件也支持C语言风格的注释，如“/*这是一个注释*/”。

#!define会定义一个常量（与C语言中的#define类似），#!ifdef需要与#!endif配对使用。如果需要将这些定义注释掉，可以在前面再加上一个或多个#。可以通过定义这些常量来控制后面的配置执行哪些代码。

```
#!define WITH_MYSQL          # 启用 MySQL 支持

#!define WITH_AUTH           # 对 SIP 消息进行认证
#!define WITH_IPAUTH         # 进行 IP 认证
#!define WITH_USRLOCDB       # 使用用户 Location 数据库表
#!define WITH_NAT            # 启用 NAT 穿越
#!define WITH_ANTIFLOOD     # 启用防洪水攻击
#!define WITH_ACCDB          # 启用记账（话单）数据库记录
```

我们还可导入其他配置文件，如果对应的配置文件不存在，也不报错。导入功能的主要作用是在不改变主配置文件的情况下，从其他文件中读入一些个性化的配置。比如，我们可以把这些主配置文件存储在Git仓库中，而本地化个性化的配置文件却因人、因环境而异，也不适合存入公共的Git仓库里。导入文件的命令如下所示。

```
import_file "Kamailio-local.cfg"
```

如果启用MySQL数据库，则需要设置数据库连接字符串。#!ifndef表示仅在常量没定义的情况下才执行它后面的内容，也需要和#!endif配对使用，相关代码如下所示。

```
#!ifndef WITH_MYSQL # 如果启用 MySQL 支持的话
#!ifndef DBURL      # 定义一个数据库连接字符串，它将在 auth_db, acc, usrloc 等模块中使用
#define DBURL "mysql://Kamailio:Kamailiorw@localhost/Kamailio"
#endif
#ifndef
#define MULTIDOMAIN 0 * 默认不启用多域支持，这样会简单些，改成 1 则启用
```

定义一些标志（Flag），这些标志的名称和值可以随意定义，但若能遵守约定俗成的命名规则通常会使配置更易读。下面是两个常用的前缀。

□ **FLT_**: Flag Transaction，与事务相关的定义。

□ **FLB_**: Flag Branch，与分支相关的定义。

定义标志的示例如下。

```
#!define FLT_ACC 1
#!define FLT_ACCMISSED 2
#!define FLT_ACCFAILED 3
#!define FLT_NATS 5
#!define FLB_NATB 6
#!define FLB_NATSIPPING 7
```

下面是一些全局变量。

```
### LOG Levels: 3=DBG, 2=INFO, 1=NOTICE, 0=WARN, -1=ERR
#ifndef WITH_DEBUG
debug=3          # 设置 Debug 的级别, 值越大日志越详细
log_stderror=yes # 是否将日志输出到“标准错误”
#else
debug=2
log_stderror=no
#endif

memdbg=5          # 内存调试级别
memlog=5          # 内存日志级别

# 定义 Log 的前缀，在打印日志时带有这些前缀可方便查看日志
#ifndef WITH_CFGLUA
log_prefix="LUA (Srm): "
```

```

#else
#ifndef WITH_CFGPYTHON
log_prefix="PY2 (Srm): "
#endif
log_prefix="NAT (Srm): "
#endif
#endif

# 延迟相关的日志级别，可跟踪哪一部分执行了多长时间，便于后期优化
latency_cfg_log=2
latency_log=2
latency_limit_action=100000
latency_limit_db=200000
log_facility=LOG_LOCAL0

fork=yes # 是否启动到后台，一般在生产上都是启动到后台，在学习或调试时启动到前台会更方便
children=2 # 子进程个数（注意，如果开启了TCP或其他模块，都有可能会启动更多单独的子进程）

# 下面这几行内容默认是注释掉的
:disable_tcp=yes # 是否禁用TCP，默认为不禁用
#auto_aliases=no # 是否禁用通过反向DNS自动根据IP地址查找别名的功能，默认为不禁用
#alias="sip.mydomain.com" # 增加一个别名（就是认为这个域是该服务器需要处理的，也可以增加多行）
#listen=udp:10.0.0.10:5060 # 监听地址，默认会监听所有IP地址

port=5060 # SIP/HTTP 监听端口

#ifndef WITH_TLS
enable_tls=yes # 是否启用TLS
#endif

/* TCP 连接最大时长，如果 SIP 客户端处于 NAT 环境下，则 Kamailio 无法做反向主动连接，只能依赖客户端连上来，所以，默认值是略微超过 SIP 注册的最大有效时长（3600 秒）*/
tcp_connection_lifetime=3605

##### 下面是模块区域 #####
# mpath="/usr/local/lib/Kamailio/modules/" # 设置模块文件所在的路径

# 加载下列模块

#ifndef WITH MYSQL
loadmodule "db_mysql.so" # MySQL 模块
#endif

loadmodule "jsonrpcs.so" # JSON-RPC 模块
loadmodule "kex.so" # Kamailio 核心扩展模块
loadmodule "corex.so" # 核心扩展模块，有一些新函数可以支持动态参数
loadmodule "tm.so" # Transaction 管理模块，有状态（Stateful）转发时需要该模块
loadmodule "tmx.so" # tm 模块的扩展模块
loadmodule "sl.so" # Stateless，无状态转发模块
loadmodule "rr.so" # Record-Route 相关逻辑
loadmodule "pv.so" # PV 处理函数
loadmodule "maxfwd.so" # 管理 Max-Forward 头域的模块
loadmodule "usrloc.so" # User Location，用户注册信息管理模块
loadmodule "registrar.so" # 处理注册消息，但用户实际的注册信息（Contact）会在 usrloc 模块中管理
loadmodule "textops.so" # 字符串处理模块
loadmodule "siputils.so" # SIP 工具函数

```

```

loadmodule "xlog.so"          # 日志函数
loadmodule "sanity.so"        # SIP 完成性校验模块
loadmodule "ctl.so"           # 使用 binrpc 管理 Kamailio 的模块，支持 Unix Socket, UDP, TCP 等
loadmodule "cfg_rpc.so"        # 使用 RPC 动态获取和设置全局变量的模块
loadmodule "acc.so"            # Accounting, 记账模块，记录话单
loadmodule "kemix.so"          # KEMI 中用到的相关函数实现，详见后面介绍 KEMI 的章节
#ifndef WITH_AUTH
loadmodule "auth.so"          # 认证鉴权模块
loadmodule "auth_db.so"        # 认证鉴权 + 数据库模块
#endif
#ifndef WITH_IPAUTH
loadmodule "permissions.so"    # 使用 IP 地址列表鉴权的模块
#endif
#ifndef
#endif
#ifndef WITH_NAT
loadmodule "nathelper.so"      # NAT 穿越模块
loadmodule "rtpproxy.so"        # RTP Proxy 控制模块
#endif
#ifndef WITH_TLS
loadmodule "tls.so"            # TLS 协议支持模块
#endif
#ifndef WITH_DEBUG
loadmodule "debugger.so"        # 调试模块
#endif
#ifndef WITH_ANTIFLOOD
loadmodule "htable.so"          # 哈希表模块
loadmodule "pike.so"            # 防洪水攻击模块
#endif
#ifndef WITH_CFG LUA
loadmodule "app_lua.so"         # KEMI Lua 语言模块
#endif
#ifndef WITH_CGFPYTHON
loadmodule "app_python.so"       # KEMI Python 语言模块
#endif

# ----- 设置各模块的参数 -----
modparam("jsonrpcs", "pretty_format", 1) # 设置 JSON 格式为方便阅读的“漂亮”格式
modparam("tm", "failure_reply_mode", 3) # 失败回复模式。自动丢弃之前串行 Fork 产生的 Leg
modparam("tm", "fr_timer", 30000)          # 设置默认重传时间为 30 秒
modparam("tm", "fr_inv_timer", 120000)       # 设置 1×× 消息后 INVITE 重发超时
/* 用户注册数据管理，启用数据库持久化，模块加载时自动从数据库加载注册信息 */
modparam("usrloc", "preload", "location")
#ifndef WITH_USRLOCDB
modparam("usrloc", "db_url", DBURL)          # 数据库连接字符串，在前面定义
modparam("usrloc", "db_mode", 2)                # 数据库模式，2 表示将内存中的数据写入数据库
modparam("usrloc", "use_domain", MULTIDOMAIN) # 是否支持多域
#endif

# ..... 在此省略掉一些模块配置 ......

#ifndef WITH_CGFLUA # 加载 Lua 路由脚本
modparam("app_lua", "load", "/usr/local/etc/kamailio/kamailio-basic-kemi-lua.lua")
cfgengine "lua"
#else
cfgengine "native" # 加载原生路由脚本
include_file "/usr/local/etc/kamailio/kamailio-basic-kemi-native.cfg"
#endif
#endif

```

上面的配置文件在Kamailio启动时加载，并且加载后不能重加载，如果有变化，只能重启Kamailio。不过，其中用modparam()指定的参数，有一些可以在Kamailio运行时通过RPC命令动态修改，具体需要参考相应模块的说明文档。

从配置文件的最后几行可以看到，我们可以根据情况选择使用的路由脚本。在此保留app_python模块只是为了演示如何加载不同的脚本，为节省篇幅其他语言的用法在这里省略掉了。下面我们仅讨论原生脚本和Lua脚本。

2.2 原生脚本

同上面的配置一样，原生脚本也只能在Kamailio启动时加载一次，不能在运行时改变。

第一个路由块是request_route，它是最先接收SIP消息的地方。根据请求以及呼叫进展的情况，它会调用其他路由块进行处理（为了让代码逻辑更清楚一些），如下面的route(REQINIT)。注意，这些route()路由块内可能会调用exit实现返回，也就是说，不管在哪里执行了exit，对本SIP消息的处理就结束了，都不会再执行它后面的代码。

```
request_route { # 主路由，第一次接触到 SIP 消息的地方，每一个进来的 SIP 请求消息都在这里处理
    route(REQINIT); # 对每一个请求都进行初始化合法性检查，是一个独立的路由块，实现代码在后面
    route(NATDETECT); # 检测是否要处理NAT，在独立的路由块中执行，实现代码在后面，下同
    if (is_method("CANCEL")) {# CANCEL 逻辑，如果有相应的事务则路由到下一跳，否则直接退出
        if (t_check_trans()) {
            route(RELAY);
        }
        exit;
    }
    route(WITHINDLG); # SIP 对话内的消息处理
    ## 如果没有对话（没有 To tag 的消息），则继续进行下面的处理
    if(t_precheck_trans()) { # 重传处理
        t_check_trans();
        exit;
    }
    t_check_trans();
    route(AUTH); # 鉴权
    remove_hf("Route"); # 去掉原有的Route消息头（如果说有的话），并换成我们自己的，以便后续的消息还经过我们
    if (is_method("INVITE|SUBSCRIBE")) {# 对这两个方法增加Record-Route头域
        record_route();
    /* 仅对 INVITE 消息记账（话单），FLT_ACC 与 acc 模块中的 log_flag 对应，通话完成后会打印日志，如果 acc 模块开启了数据库设置，也可以将通话话单写入数据库 */
    if (is_method("INVITE")) {
        setflag(FLT_ACC); # do accounting
    }
    /* 如果请求的 domain 不是本服务器（不在本地 alias 列表中），则直接路由到对应的 domain*/
    route(SIPOUT);
    ## 如果请求的 domain 由本服务器负责，则继续执行下面的路由块
    route(REGISTRAR); # 处理注册消息
    if ($rU==null) { # 合法性检查
        # 如果 SIP 消息到了这里，那么在 RURI 中必须有 username 部分，否则报错
        sl_send_reply("484","Address Incomplete");
        exit;
    }
}
```

```

    }

    route[LOCATION]; # 呼叫本地的注册用户

}

route[RELAY] { # 接力转发
    # 对于转发的消息，执行更多的事件触发路由，如串行 Forking、RTP 转发处理等
    if (is_method("INVITE|BYE|SUBSCRIBE|UPDATE")) {
        # 如果设置了 'branch_route'，则会触发相应的事件路由
        if(!t_is_set("branch_route")) t_on_branch("MANAGE_BRANCH");
    }
    if (is_method("INVITE|SUBSCRIBE|UPDATE")) {
        if(!t_is_set("onreply_route")) t_on_reply("MANAGE_REPLY");
    }
    if (is_method("INVITE")) {
        if(!t_is_set("failure_route")) t_on_failure("MANAGE_FAILURE");
    }
    # 调用 t_relay() 进行转发，如果出错则返回错误消息
    if (!t_relay()) {
        sl_reply_error();
    }
    /* 到此就结束了，因此，如果在任意路由块里调用了该函数，对于本消息而言，脚本执行就终止。
       不再进行后续的操作了 */
    exit;
}

route[REQINIT] { # 对每一个 SIP 请求执行合法性检查
#ifndef WITH_ANTIFLOOD
    if(src_ip!=myself) { # 检查是否收到大量 SIP 消息（洪水攻击），将信任的 IP 地址放到白名单内，以防误伤
        if(Ssht(ipban=>$si) !=$null) { # 这个 sht 是一个共享内存的哈希表，用作 IP 地址黑名单
            # 如果能从表中找到，说明这个 IP 地址已经被防住了，直接退出
            xdbg("request from blocked IP - $rm from $fu (IP:$si:$sp)\n");
            exit;
        }
        # 使用 pike 模块检测从同一个 IP 地址收到的消息频率（具体频率可设置），如果请求频次超过限制，则将这个 IP 地址加入黑名单
        if (!pike_check_req()) {
            xlog("L_ALERT","ALERT: pike blocking $rm from $fu (IP:$si:$sp)\n");
            Ssht(ipban=>$si) = 1;
            exit;
        }
    }
    # 判断 User-Agent 是否是已知的扫描工具，如果是，则直接返回 200 OK，不处理
    if($ua == "friendly-scanner") {
        sl_send_reply("200", "OK");
        exit;
    }
#endif
    # 检查 Max-Forward 字段是否低到 10，防止 SIP 消息又绕回来产生死循环
    if (!mf_process_maxfwd_header("10")) {
        sl_send_reply("483","Too Many Hops");
        exit;
    }
    if(is_method("OPTIONS") && uri==myself && SrU==$null) { # 处理 OPTIONS 请求
        sl_send_reply("200","Keepalive");
        exit;
    }
}

```

```

if(!sanity_check("1511", "7")) { # 检查 SIP 消息的合法性
    xlog("Malformed SIP message from $si:$sp\n");
    exit;
}

route[WITHINDLG] { # 处理 SIP 对话内的请求
    if (!has_totag()) return;
    if (loose_route()) { # 同一对话内后续的请求应该从 Record-Routing 头域中选择下一跳
        route(DLGURI);
        if (is_method("BYE")) {
            setflag(FLT_ACC); # 记账
            setflag(FLT_ACCFAILED); # 即使呼叫失败也记账
        } else if ( is_method("ACK") ) {
            route(NATMANAGE); # ACK 需要无状态转发
        } else if ( is_method("NOTIFY") ) {
            record_route(); # 为对话内的 NOTIFY 增加 Record-Route 头域, 参见 RFC 6665
        }
        route(RELAY);
        exit;
    }
    if ( is_method("ACK") ) {
        if ( t_check_trans() ) {
            # 不是松散路由, 但却是有状态的 ACK, 该 ACK 应该是 487 后的 ACK
            # 或者是上游服务器返回 404 时的 ACK
            route(RELAY);
            exit;
        } else {
            exit; # ACK 没有对应的事务, 忽略并丢弃
        }
    }
    # 如果执行到这里, 那就不归我们管了, 回复 404 错误消息
    sl_send_reply("404", "Not here");
    exit;
}

route[REGISTRAR] { # 处理 SIP 注册请求
    if (!is_method("REGISTER")) return; # 仅处理注册消息, 否则直接返回
    if(isflagset(FLT_NATS)) { # NAT 相关的处理,略
        setbflag(FLB_NATB);
    }
#ifndef WITH_NATSIPPING
    setbflag(FLB_NATSIPPING); # 执行 SIP NAT pinging
#endif
}
/* 将注册消息中的 Contact 写到 location 表中 (可以是内存也可以是数据库) 并返回 200 OK,
   如果保存失败则返回错误 */
if (!save("location"))
    sl_reply_error();
exit;
}

route[LOCATION] { # 查找并呼叫本地注册用户
    # 比如两个用户 a 和 b 都注册到 Kamailio 中, 则 a 呼叫 b 时就会查询 location 表看 b 有没有注册
    if (!lookup("location")) { # 如果找不到则根据返回值进行出错处理
        $var(rc) = $rcr;
        t_newtran();
        switch ($var(rc)) {
            case -1:

```

```

        case -3:
            send_reply("404", "Not Found");
            exit;
        case -2:
            send_reply("405", "Method Not Allowed");
            exit;
    }
}

if (is_method("INVITE")) { # 当使用 usrloc 路由时, 也对未呼通的呼叫记账
    setflag(FLT_ACCMISSED);
}
route(RELAY); # 如果找到被叫用户的注册地址, 则向注册地址转发 INVITE 消息
exit;          # 退出
}

route[AUTH] { # 基于 IP 地址的认证
#ifndef WITH_AUTH
#ifndef WITH_IPAUTH
# 检查是否在 IP 地址白名单里
if((!is_method("REGISTER")) && allow_source_address()) {
    return; # 允许, 直接返回
}
#endif
if (is_method("REGISTER") || from_uri==myself) {
    # 处理注册请求, 检查 subscriber 表中有没有对应的用户名和密码信息
    if (!auth_check("$fd", "subscriber", "1")) {
        # 如果还没有认证则发起 chanllenge 认证, 返回 407
        auth_challenge("$fd", "0");
        exit;
    }
    # 已经验证通过, 在转发到下一跳前去掉认证相关的消息头
    if(!is_method("REGISTER|PUBLISH"))
        consume_credentials();
}
/* 如果主叫不是我们本地已知的用户, 则只允许呼叫本地注册用户, 否则不允许转发, 毕竟我们的服务
   器不是开放的转发服务器 (Open Relay) */
if (from_uri!=myself && uri!=myself) {
    sl_send_reply("403","Not relaying");
    exit;
}
#endif
return;
}

route[NATDETECT] { # 进行主叫 NAT 相关处理, 如修改 Contact 信息等。略
#ifndef WITH_NAT
force_rport();
if (nat_uac_test("19")) {
    if (is_method("REGISTER"))
        fix_nated_register();
    else {
        if(is_first_hop())
            set_contact_alias();
    }
    setflag(FLT_NATS);
}
#endif
}

```

```

        return;
    }

    route[NATMANAGE] { # RTPProxy 控制，在 NAT 环境下将会修改 SDP，通过 RTPProxy 进行媒体转发。略
        ifndef WITH_NAT
            if (is_request()) {
                if(has_totag()) {
                    if(check_route_param("nat=yes")) {
                        setbflag(FLB_NATB);
                    }
                }
            }
            if (!(isflagset(FLT_NATS) || isbflagset(FLB_NATB)))
                return;
            rtpproxy_manage("co");
            if (is_request()) {
                if (!has_totag()) {
                    if(t_is_branch_route())
                        add_rr_param(";nat=yes");
                }
            }
            if (is_reply()) {
                if(isbflagset(FLB_NATB))
                    set_contact_alias();
            }
        }
        endif
        return;
    }

    route[DLGURI] { # 在对话相关请求中可能要进行 URI 更新
        ifndef WITH_NAT
            if(!isdsturiset())
                handle_ruri_alias();
        }
        endif
        return;
    }

    route[SIPOUT] { # 路由到外部的 SIP 服务器 (domain 不属于我们)
        if (uri==myself) return; # 如果不是外部 SIP 服务器则返回
        append_hf("P-hint: outbound\r\n"); # 增加 SIP 头域以方便跟踪 SIP 消息
        route(RELAY); # 直接转发 SIP 消息
        exit;
    }

    branch_route[MANAGE_BRANCH] { # 外呼的分支
        xdbg("new branch [$T_branch_idx] to $ru\n"); # 打印日志
        route(NATMANAGE); # 转到 NAT 处理
    }

    onreply_route[MANAGE_REPLY] { # 收到答复消息时进行处理
        xdbg("incoming reply\n");
        if(status=="[12][0-9][0-9]")
            route(NATMANAGE); # 检查是否需要 NAT 处理
    }

    failure_route[MANAGE_FAILURE] { # 呼叫失败的处理
        route(NATMANAGE); # 检查是否需要 NAT 处理
        if (t_is_CANCELled()) { # 如果之前收到过 CANCEL 消息，则直接退出，否则继续处理
            exit;
        }
    }
}

```

上述路由脚本可以完成SIP用户注册互打电话这样的场景，也可以呼叫到其他的SIP服务器（只需要指定其他SIP服务器的域即可）。可以看出，针对不同的SIP消息（如INVITE、CANCEL等），在呼叫的不同阶段都有不同的处理方式，里面的业务逻辑也比较多（好多if...else判断）。一般来说，从头写一个路由脚本还是比较难的，如有需要，最好在官方提供的示例脚本的基础上进行修改。

如果要支持注册，需要将SIP客户端的注册信息写入subscriber表，具体实现方法我们将在后文详细描述。

上述脚本是Kamailio原生的路由配置脚本。在5.0以后的Kamailio中，我们都建议使用KEMI脚本并用Lua或Python等语言配置路由。但由于KEMI比较新，资料比较少，网上很多例子都是用原生脚本写成的，因此，笔者希望通过本节帮助大家理解Kamailio的路由逻辑，与KEMI进行对比也更有助

于进一步理解和学习Kamailio的配置方法。

2.3 Lua脚本

Lua脚本跟原生脚本在性能上没什么差距，但由于Lua是一门专业的编程语言，有编程背景的人写起来会更顺手一些。

Lua脚本仅用于路由部分，基本的配置还是要用到2.1节介绍的脚本。但如果在后续只更改Lua的路由部分，可以在命令行上使用kamcmd app_lua.reload进行重载，而无须重启Kamailio，这样就不会影响通话了。这也是Lua脚本的最大好处之一。

Lua脚本中有一个全局的KSR对象，用于调用Kamailio中的函数和逻辑，它是在app_lua模块中实现的。Kamailio旧版本中的Lua功能会产生sr对象，现在sr对象由app_lua_sr模块实现，这样做仅为了向后兼容，在未来的版本中可能会去掉相关设置。

Lua脚本中，以--开头的是单行注释语句，用--[]包起来的是多行注释语句。

注意

不要在Lua脚本中执行exit，它会导致整个Kamailio退出，如果只是结束当前消息的处理，则应该使用KSR.x.exit()。KSR.drop()会丢弃SIP消息，但路由脚本还会继续执行，如果需要停止执行，则需要在其后面调用KSR.x.exit()，或者直接将KSR.drop()换为KSR.x.drop()。

可以用luac -p /path/to/script.lua命令检查Lua脚本有没有语法问题。

下面看一个示例。

```

-- 全局变量，在这里还需要再定义一遍，要与主配置文件中相应的配置一致
FLT_ACC=1
FLT_ACCMISSSED=2
FLT_ACCFAILED=3
FLT_NATS=5
FLB_NATB=6
FLB_NATIPPING=7

-- ksr_request_route 这个名称是固定的，对应原生脚本中的 request_route，收到所有的 SIP 请求都会调用该函数进行处理
function ksr_request_route()
    ksr_route_reinit();
    -- 初始合法性检查，其实它就是一个一般的 Lua 函数 (function) 调用，函数名可以任意
    ksr_route_natdetect(); -- NAT 处理
    if KSR.is_CANCEL() then -- CANCEL 处理
        if KSR.tm.t_check_trans()>0 then
            ksr_route_relay();
        end
        return 1;
    end
    ksr_route_withindlg(); -- SIP 对话内的消息处理
    -- 如果没有对话（仅对于设有 To tag 的消息），则继续进行下面的处理
    if KSR.tm.t_precheck_trans()>0 then -- 重传处理
        KSR.tm.t_check_trans();
        return 1;
    end
    if KSR.tm.t_check_trans()==0 then return 1 end
    ksr_route_auth(); -- 鉴权
    -- 去掉原有的 Route 消息头（如果有的话），并换成我们自己的，以便后续的消息还经过我们
    KSR.hdr.remove("Route");
    if KSR.is_method_in("IS") then -- 如果method是INVITE或SUBSCRIBE
        KSR.rr.record_route();
    end
    if KSR.is_INVITE() then -- 仅对INVITE记账
        KSR.setflag(FLT_ACC); -- 记账
    end
    ksr_route_sipout(); -- 将 domain 不是本地的请求转发到外部服务器
    -- 下面都是我们自己的 domain 了
    ksr_route_registrar(); -- 处理注册
    if KSR.corex.has_ruri_user() < 0 then
        -- RURI 中必须有 user 部分，否则回复“484 地址不全”的消息
        KSR.s1.al_send_reply(484,"Address Incomplete");
        return 1;
    end
    ksr_route_location(); -- 查找并呼叫本地注册用户
    return 1;
end

function ksr_route_relay() -- 接力转发
    -- 对于转发的消息执行更多的事件触发路由，如串行 Forking、 RTP 转发处理等
    if KSR.is_method_in("IBSU") then
        if KSR.tm.t_is_set("branch_route")<0 then
            -- 如果设置了 branch_route，则会触发相应的事件路由
            KSR.tm.t_on_branch("ksr_branch_manage");
        end
    end
end

```

```

if KSR.is_method_in("ISU") then
    if KSR.tm.t_is_set("onreply_route")<0 then
        KSR.tm.t_on_reply("ksr_onreply_manage");
    end
end
if KSR.is_INVITE() then
    if KSR.tm.t_is_set("failure_route")<0 then
        KSR.tm.t_on_failure("ksr_failure_manage");
    end
end
if KSR.tm.t_relay()<0 then -- 调用 t_relay() 进行转发, 如果出错则返回错误消息
    KSR.sl.sl_reply_error();
end
-- 到此就结束了。因此, 如果在任意路由块里调用了该函数, 对于本消息而言, 脚本执行就终止, 就不
-- 进行后续的操作了
KSR.x.exit();
end

function ksr_route_reqinit() -- 对每一个 SIP 请求执行合法性检查
if not KSR.is_myself_srcip() then
    -- 检查是否收到大量 SIP 消息的洪水攻击, 将信任的 IP 地址放到白名单内, 以防误伤
    local srcip = KSR.kx.get_srcip();
    -- 这个 sht 是一个共享内存的哈希表, 用作 IP 地址黑名单
    if KSRhtable.sht_match_name("ipban", "eq", srcip) > 0 then
        -- 如果能从表中找到, 说明这个 IP 地址已经被防住了, 直接退出
        KSR.debug("request from blocked IP - " .. KSR.kx.get_method()
            .. " from " .. KSR.kx.get_furi() .. " (IP:" ..
            .. srcip .. ":" .. KSR.kx.get_srcport() .. ")\n");
        KSR.x.exit();
    end
    -- 使用 pike 模块检测从同一个 IP 地址收到消息的频率(具体频率可设置), 如果请求频次超过限
    频, 则将这个 IP 地址加入黑名单
    if KSR.pike.pike_check_req() < 0 then
        KSR.error("ALERT: pike blocking " .. KSR.kx.get_method()
            .. " from " .. KSR.kx.get_furi() .. " (IP:" ..
            .. srcip .. ":" .. KSR.kx.get_srcport() .. ")\n");
        KSRhtable.sht_seti("ipban", srcip, 1);
        KSR.x.exit();
    end
end
-- 判断 User-Agent 是否是已知的扫描工具, 如果是, 则直接返回 200 OK, 不处理
local ua = KSR.kx.get_ua();
if string.find(ua, "friendly") or string.find(ua, "scanner")
    or string.find(ua, "sipcli") or string.find(ua, "sippicious") then
    KSR.sl.sl_send_reply(200, "OK");
    KSR.x.exit();
end
-- 检查 Max-Forward 字段是否低到 10, 防止 SIP 消息又绕回来产生死循环
if KSR.maxfwd.process_maxfwd(10) < 0 then
    KSR.sl.sl_send_reply(483, "Too Many Hops");
    KSR.x.exit();
end
if KSR.is_OPTIONS() -- 处理 OPTIONS 请求
    and KSR.is_myself_ruri()
    and KSR.corex.has_ruri_user() < 0 then
    KSR.sl.sl_send_reply(200, "Keepalive");
    KSR.x.exit();
end

```

```

if KSR.sanity.sanity_check(l5ll, 7)<0 then -- 检查 SIP 消息的合法性
    KSR.err("Malformed SIP message from "
        .. KSR.kx.get_srcip() .. ":" .. KSR.kx.get_srcport() .."\n");
    KSR.x.exit();
end

-- 处理 SIP 对话内的请求
function ksr_route_withindlg()
    if KSR.siputils.has_totag()<0 then return 1; end
    -- 同一对话内后续的请求应该从 Record-Route 头域中选择下一路
    if KSR.rr.loose_route()>0 then
        ksr_route_dlguri();
        if KSR.is_BYE() then
            KSR.setflag(FLT_ACC); -- 记账
            KSR.setflag(FLT_ACCFAILED); -- 即使呼叫失败也记账
        elseif KSR.is_ACK() then
            ksr_route_natmanage(); -- ACK 需要无状态转发
        elseif KSR.is_NOTIFY() then
            -- 为对话内的 NOTIFY 增加 Record-Route 头域, 参见 RFC 6665
            KSR.rr.record_route();
        end
        ksr_route_relay();
        KSR.x.exit();
    end
    if KSR.is_ACK() then
        if KSR.tm.t_check_trans() >0 then
            -- 不是桥接路由, 却是有状态的 ACK, 该 ACK 应该是 487 后的 ACK
            -- 或者是上游服务器返回 404 时的 ACK
            ksr_route_relay();
            KSR.x.exit();
        else
            KSR.x.exit(); -- ACK 没有对应的事务, 忽略并丢弃
        end
    end
    -- 如果执行到这里, 那就不归我们管了, 返回 404 错误
    KSR.sl.sl_send_reply(404, "Not here");
    KSR.x.exit();
end

function ksr_route_registrar() -- 处理 SIP 注册请求
    if not KSR.is_REGISTER() then return 1; end -- 仅处理注册消息, 否则直接返回
    if KSR.isflagset(FLT_NATS) then -- NAT 相关的处理, 啊
        KSR.setbflag(ELB_NATB);
        KSR.setbflag(FLB_NATSIPPING); -- 执行 SIP NAT pinging, 啊
    end
    -- 将注册消息中的 Contact 写到 location 表中 (可以是内存也可以是数据库) 并返回 200 OK, 如果
    -- 保存失败则返回错误
    if KSR.registrar.save("location", 0)<0 then
        KSR.sl.sl_reply_error();
    end
    KSR.x.exit();
end

function ksr_route_location() -- 查找并呼叫本地注册用户
    -- 比如两个用户 a 和 b 都注册到 Kamailio 中, 则 a 呼叫 b 时就会查询 location 表看 b 有没有注册
    local rc = KSR.registrar.lookup("location");

```

```

if rc<0 then -- 如果找不到则根据返回值进行出错处理
    KSR.tm.t_newtran();
    if rc== -1 or rc== -3 then
        KSR.sl.send_reply(404, "Not Found");
        KSR.x.exit();
    elseif rc== -2 then
        KSR.sl.send_reply(405, "Method Not Allowed");
        KSR.x.exit();
    end
end
if KSR.is_INVITE() then -- 当使用 usrloc 路由时，也对未呼通的呼叫记账
    KSR.setflag(FLT_ACCMISSED);
end
-- 如果找到被叫用户的注册地址，则向注册地址转发 INVITE 消息
kar_route_relay();
KSR.x.exit();
end

function kar_route_auth() -- 基于 IP 地址的认证
    if not KSR.auth then
        return 1;
    end
    if KSR.permissions and not KSR.is_REGISTER() then -- 检查是否在 IP 地址白名单里
        if KSR.permissions.allow_source_address(1)>0 then
            return 1; -- 允许呼叫，直接返回
        end
    end
    -- 处理注册请求，检查 subscriber 表中有没有对应的用户名和密码信息
    if KSR.is_REGISTER() or KSR.is_myself_furi() then
        -- 对请求进行鉴权检查
        if KSR.auth_db.auth_check(KSR.kx.getf_fhost(), "subscriber", 1)<0 then
            -- 如果还没有认证则发起 challenge 认证，返回 407
            KSR.auth.auth_challenge(KSR.kx.getf_fhost(), 0);
            KSR.x.exit();
        end
        -- 已经验证通过，在转发到下一跳前去掉认证相关的消息头
        if not KSR.is_method_in("RP") then
            KSR.auth.consume_credentials();
        end
    end
    -- 如果主叫不是我们本地已知的用户，则只允许呼叫本地注册用户，否则不允许转发，毕竟我们的服务
    -- 不是开放的转发服务器 (Open Relay)
    if (not KSR.is_myself_furi())
        and (not KSR.is_myself_ruri()) then
        KSR.sl.sl_send_reply(403,"Not relaying");
        KSR.x.exit();
    end
    return 1;
end

function ksr_route_natdetect() -- 进行主叫 NAT 相关处理，如修改 Contact 信息等。略
    if not KSR.nathelper then return 1; end
    KSR.force_rport();
    if KSR.nathelper.nat_uac_test(19)>0 then
        if KSR.is_REGISTER() then
            KSR.nathelper.fix_nated_register();
        elseif KSR.siputils.is_first_hop()>0 then
            KSR.nathelper.set_contact_alias();

```

```

        end
        KSR.setflag(FLT_NATS);
    end
    return 1;
end

-- RTPProxy 控制, 在 NAT 环境下将会修改 SDP, 通过 RTPProxy 进行媒体转发。略
function ksr_route_natmanage()
    if not KSR.rtpproxy then
        return 1;
    end
    if KSR.siputils.is_request()>0 then
        if KSR.siputils.has_totag()>0 then
            if KSR.rr.check_route_param("nat=yes")>0 then
                KSR.setbflag(FLB_NATB);
            end
        end
    end
    if (not (KSR.isflagset(FLT_NATS) or KSR.isbflagset(FLB_NATB))) then
        return 1;
    end
    KSR.rtpproxy.rtpproxy_manage("co");
    if KSR.siputils.is_request()>0 then
        if KSR.siputils.has_totag()<0 then
            if KSR.tmx.t_is_branch_route()>0 then
                KSR.rr.add_rr_param(":nat=yes");
            end
        end
    end
    if KSR.siputils.is_reply()>0 then
        if KSR.isbflagset(FLB_NATB) then
            KSR.nathelper.set_contact_alias();
        end
    end
    return 1;
end

function ksr_route_dlguri() -- 在对话相关请求中可能要进行 URI 更新
    if not KSR.nathelper then return 1; end
    if not KSR.isdsturiset() then
        KSR.nathelper.handle_ruri_alias();
    end
    return 1;
end

function ksr_route_sipout() -- 路由到外部的 SIP 服务器 (domain 不属于我们)
    if KSR.is_myself_ruri() then return 1; end
    KSR.hdr.append("P-Hint: outbound\r\n"); -- 增加一个 SIP 头域以便跟踪 SIP 消息
    ksr_route_relay(); -- 直接转发 SIP 消息
    KSR.x.exit();
end

function ksr_branch_manage() -- 外呼的分支, 相当于原生脚本中的 branch_route[...]
    KSR.dbg("new branch [".. KSR.pv.get("ST_branch_idx")
            .. "] to " .. KSR.kx.get_ruri() .. "\n");
    ksr_route_natmanage();
    return 1;
end

```

```
function ksr_onreply_manage() -- 收到回复消息时进行处理，相当于原生脚本中的 onreply_route[...]
    KSR.dbg("incoming reply\n");
    local scode = KSR.kx.get_status();
    if scode>100 and scode<299 then
        ksr_route_natmanage();
    end
    return 1;
end

function ksr_failure_manage() -- 呼叫失败的处理，相当于原生脚本中的 failure_route[...]
    ksr_route_natmanage();
    if KSR.tm.t_is_CANCELed(>0) then
        return 1;
    end
    return 1;
end

function ksr_reply_route() -- 收到回复消息时进行处理，相当于原生脚本中的 reply_route[...]
    KSR.info("===== response - from Kamailio lua script\n");
    return 1;
end
```

从上面的Lua脚本中可以看出，它与原生脚本的功能和写法基本是一样的，也许它的代码量（行数）并不少，但读起来更清晰也更易懂，而且它是真正的编码语言，更便于写单元测试，甚至在没有Kamailio的情况下也可以测试脚本的正确性（关于单元测试，我们将在后文中讲解），这一点使用原生脚本是做不到的。

2.4 Lua脚本的其他写法

上一节介绍的Lua脚本是Kamailio自带的示例，与原生脚本写法一致是为了方便对照学习，并不是



说只有这一种写法。事实上，有一个lua-kamailio项目就使用了另一种写法。该项目的描述是：它实现了一个Kamailio Lua函数库，用于替换单一KEMI示例文件，以方便进行单元测试。我们先来看如下代码。

```
-- 文件名: headers.lua
-- 处理 SIP Header 相关的函数
rex = require "rex_pcre"
message = require "kamailio.message"

local headers = {}

function headers.get_request_user()
    return KSR.pv.get("$rU")
end

function headers.set_request_user(value)
    KSR.pv.sets("$rU", value)
end

return headers

-- 文件名: core.lua
-- 核心相关的函数
local core = {}

function core.exit()
    KSR.x.exit()
end

function core.set_flag(flag)
    KSR.setflag(flag)
end

function core.set_branch_flag(flag)
    KSR.setbflag(flag)
end

function core.is_flag_set(flag)
    return KSR.isflagset(FLT_NATS)
end

return core
```

以上只是一些代码片段，因篇幅所限，就不在此多解释了。通过将Lua代码分散到不同的文件中，使用模块化方式进行组织，可以使路由脚本代码看起来更具Lua风格。虽然上述项目并不十分活跃，但可以作为一个很有意义的参考，大家在以后的项目中可以自由发挥。

至此，大家应该对Kamailio中的SIP处理和路由逻辑有了大致的理解，可以进行后面的学习了。如果上面的代码让你感觉很难理解，先不要急，跳过去就好了。接下来，我们会针对每一个概念和主题进行详细讲解，那时再回来阅读本章，一切就显得很容易了。

Chapter 3

第3章

Kamailio基本概念和组件

Kamailio是一个多进程多线程的系统，这一点与FreeSWITCH不同，后者是一个多线程的系统。进程是操作系统（如Linux）管理的独立的资源调度单位，有独立的内存空间，管理起来比多线程更复杂一些。

Kamailio要支持跨进程的内存管理和通信，就会用到共享内存，以便不同的进程都能管理相同的数

据。Kamailio的很多设计都与此有关。下面一起来看一下Kamailio中的基本概念和组件 。

3.1 core详解

core（核心）是Kamailio的核心组件，可对外导出一些函数和参数，用于Kamailio的配置文件。在前文中我们讲到Kamailio的配置文件大致分为全局参数、模块设置、路由块几个部分。

下面就来看一下核心中包含的内容以及上述配置文件中的几个部分与核心组件的对应关系。

3.1.1 全局参数部分

除预处理相关的指令外，全局参数是配置文件中的第一部分。参数和格式一般是：“name=value”。其中name对应核心模块中对外导出的参数（3.1.7节中将详细描述），如果配置文件中的参数在核心实现中找不到（可能因拼写错误导致），Kamailio将报错并无法启动。

value的典型类型有3种：整型、布尔型和字符串型。其中字符串型需要用双引号引起来，但有一种“标志符”字符串，不需要使用双引号。

有的参数可能复杂一些，会包含整型、字符串型等，如listen参数，通常的格式如下：

```
proto:ip:port # 协议 :IP 地址 :端口
```

这里就包含了标志符字符串和整型变量。

下面是一些常见的例子。

```
log_facility = LOG_LOCAL0      # 标志符字符串, 日志设备
children = 4                   # 整型
disable_tcp = yes              # 布尔型
alias = "sip.xswitch.cn"       # 字符串型
listen = udp:192.168.7.7:5060  # 标志符字符串, 整型
```

一般来说行尾无须加分号，但是有的参数可能支持多行（如listen和alias），这时候就要加分号了，以免下一行被错误解析，示例如下。

```
alias = "sip.xswitch.cn";
```

如果你使用Kamailio保留字符串（内置标志符），则需要使用引号，如在下面的例子中，“dns”是保留字。

```
listen = tcp:127.0.0.1:5060 advertise "sip.dns.example.com":5060
```

3.1.2 模块设置部分

配置模块的加载，可以通过以下参数指定路径。

```
mpath = "/usr/local/lib/kamailio/modules/"
```

模块加载的例子如下。

```
loadmodule "debugger.so"           # 加载该模块
modparam("debugger", "cfgtrace", 1) # 设置模块参数
```

不同的模块有不同的参数，模块参数的格式是“modparam（"模块名称", "参数名称", 参数值）”。其中参数值也有字符串或整数之分，也可以引用#define定义的宏。

3.1.3 路由块部分

路由块部分是实际意义上的SIP消息处理的逻辑，使用Kamailio原生脚本语言描述。SIP消息的处理从下列路由块开始。

```
request_route { // 主路由块, SIP 消息最先触达的地方
    route(REQINIT); // 次路由块, 每个请求都调用一次
    ...
}

branch_route[MANAGE_BRANCH] { // 分支路由块, 在新分支创建时被调用
    xdbg("new branch [$T_branch_idx] to $ru\n");
    route(NATMANAGE);
}
```

各种路由块我们已经在第2章中讲过了，此处不再赘述。

3.1.4 通用元素

本节介绍Kamailio配置脚本中的通用元素。

1.6种主要的通用元素

Kamailio配置脚本中的通用元素主要有7种，本节介绍前6种，因为第七种预处理指令内容比较多，所以单独作为一节来介绍。

(1) 注释。支持以#和//的形式进行单行注释，以及以类似于C语言中的“/* */”的形式进行多行注释。示例如下。

```
# 这是一行注释
// 这是另一行注释
/*
 * 这是一个
 * 多行注释
 */
```

注意：以#!开头的是预处理指令（类似于C语言中的预处理），不是注释。

(2) 值。Kamailio有如下3种类型的值。

□ 整型：32位整数，如10。

□ 布尔型：1、true、on、yes或0、false、off、no。

□ 字符串型：单引号或双引号括起来的字符串，如"this is a string value"。

(3) 标志符。标志符是不加引号的字符串，用于匹配整数或布尔值。核心参数名、函数名、模块函数名、核心关键字、语句等都是标志符，如前面提到的dns、uri、return等。

(4) 变量。变量以\$开头。示例如下。

```
$var(x) = $rU + "@" + $fd;r
```

其中，\$var()为自定义变量，其他的为伪变量。\$rU代表Request User，\$fd代表From Domain，它们跟@拼接后赋值给一个变量x。更多的伪变量说明可以参见3.2节。

(5) 动作或语句。在路由模块中执行的指令，结尾需要加“;”。指令可以是核心或模块的函数调用语句、条件语句、循环语句或赋值语句等，语法类似于C语言。示例如下。

```
    sl_send_reply("404", "Not found");
    exit;
```

(6) 表达式。表达式是由一组语句、变量、函数和操作符组成的值。示例如下。

```
if(!t_relay())
if($var(x)>10)
"sip:" + $var(prefix) + $rU + "@" + $rd
```

2. 预处理指令

类似于C语言，Kamailio配置文件中也有一些预处理指令，简述如下。

1) **include_file**

include_file使用方法如下。

```
include_file "path_to_file"
```

include_file类似于C语言中的#include指令，用于引入一个文件。path_to_file是被引入的文件路径，可以是相对路径也可以是绝对路径，但必须是一个字符串（不能是变量）。include_file预处理指令在Kamailio启动时执行，一旦被引入的文件有修改，必须重启Kamailio才能生效。如果文件是一个相对路径，则默认查找当前目录，如果找不到，则查找调用include_file指令的那个文件所在的目录。被引入的文件可以包含任何配置，也可以继续使用include_file指令引入其他文件，但最大不能超过10级。该指令也可以使用#!include_file或!!include_file代替。

如果被引入的文件不存在，则Kamailio会报错并停止启动。

include_file指令的使用示例如下。

```
request_route {
    ...
    include_file "routes.cfg"
    ...
}
```

其中，routes.cfg文件内容示例如下。

```
if (!mf_process_maxfwd_header("10")) {
    sl_send_reply("483", "Too Many Hops");
    exit;
}
```

2) **import_file**

import_file使用方法如下。

```
import_file "path_to_file"
```

import_file类似于include_file，但在找不到文件时不会报错。

3) **define**

define类似于C语言中的宏定义，决定配置文件中哪部分需要执行。宏定义外的配制信息在Kamailio启动时会丢弃，以节省系统资源。

系统支持以下关键字定义方式。

- #!define NAME: 定义一个关键字NAME。
- #!define NAME VALUE: 定义关键字并赋值。
- #!ifdef NAME: 检查关键字是否被定义。
- #ifndef: 检查关键字是否没有被定义。
- #!else: 否则, 后接分支语句。
- #!endif: 结束ifdef/ifndef的定义。
- #!trydef: 如果对应关键字没有定义则进行定义, 否则对应关键字无效。
- #!redefine: 强制重定义, 即使已经定义。

系统内置定义的关键字:

- KAMAILIO_X[_Y[_Z]]: Kamailio版本号。

- MOD_X: 当模块X加载后被赋值。

在Kamailio运行时可以在命令行上使用kamctl core.ppdefines_full命令获取完整关键字列表。

使用宏指令有以下好处:

- 快速启用或禁用某些特性, 如默认配置中的NAT穿越、Presence、鉴权等特性。
- 执行条件语句无法完成的任务, 如在全局参数、模块设置中选择不同的条件。
- 当切换不同的应用场景时, 比条件语句更高效。如在开发环境和生产环境中可以使用不同的参数。

define使用示例如下。

```

#define DEV_MODE      # 定义开发模式

#ifndef DEV_MODE    # 在开发模式下, 以下语句生效
debug = 5
log_stderr = yes
listen = 192.168.1.1
#else              # 在生产环境下, 取消 DEV_MODE 定义, 此时以下语句生效
debug = 2
log_stderr = no
listen = 10.0.0.1
#endif

...
#ifndef DEV_MODE    # 开发环境和生产环境分别连接不同的数据库
modparam("acc|auth_db|usrloc", "db_url",
        "mysql://kamailio:kamailio@localhost/kamailio")
#else
modparam("acc|auth_db|usrloc", "db_url",
        "mysql://kamailio:kamailiorw@10.0.0.2/kamailio_production")
#endif

...
#ifndef DEV_MODE
route{DEBUG} {
    xlog("SCRIPT: SIP $rm from: $fu to: $ru - srcip: $si\n");
}
#endif

```

```
...
route {
#ifndef DEV_MODE
    route(DEBUG);
#endif
}
```

可以用define定义字符串或整型值的宏。示例如下。

```
#define MYINT 123
#define MYSTR "xyz"
```

定义的宏在运行时会被替换，这一点跟C语言类似，如以下代码所示。

```
$var(x) = 100 + MYINT;
```

上述代码将被替换为如下形式。

```
$var(x) = 100 + 123;
```

define支持多行定义。示例如下。

```
#define MYLOOP $var(i) = 0; \
              while($var(i)<5) { \
                  xlog("++++ $var(i)\n"); \
                  $var(i) = $var(i) + 1; \
              }
```

在路由块中调用示例，方法如下。

```
route {
    ...
    MYLOOP
    ...
}
```

目前系统最大只能定义256个值。

注意

多行定义最终会在内部变成一行，因此，如果在多行定义中使用注释，可能会出现不可预知的结果，所以，如果要使用注释，那么只能在最后一行使用。

4) defenv

defenv表示从系统环境变量中获取值，并用define定义为Kamailio宏。示例如下。

```
#!defenv SHELL
```

假设系统的SHELL环境变量值为/bin/bash，则上述代码等价于以下定义。

```
#!define SHELL /bin/bash
```

也可以使用以下方式。

```
#!defenv MYSHELL=SHELL
```

上述定义等价于如下形式。

```
#!/define MYSHELL /bin/bash
```

当然也可以使用后面将要讲到的#!substdef与\$env(NAME)来实现defenv的功能，但用defenv更简洁。

5) subst

subst用于在配置文件中进行字符串替换。注意它只替换关键字标志符（没有引号的标志符）。示例如下。

```
#!subst "/regexp/subst/flags"
```

其中，flags是可选的，值可以是i（代表忽略大小写）或g（代表全局替换）。

下面的代码会将db_url中的DBPASSWD标志符替换为xyz。

```
#!subst "/DBPASSWD/xyz/"  
modparam("acc", "db_url", "mysql://user:DBPASSWD@localhost/db")
```

6) substdef

substdef的使用方法如下。

```
#!substdef "/ID/subst/"
```

substdef类似于subst，但额外增加了一个#!define ID subst定义。示例如下。

```
#!substdef "/DBPASSWD/xyz/"  
modparam("acc", "db_url", "mysql://user:DBPASSWD@localhost/db")
```

上述代码等价于如下形式。

```
#!subst "/DBPASSWD/xyz/"  
#!define DBPASSWD xyz  
modparam("acc", "db_url", "mysql://user:DBPASSWD@localhost/db")
```

7) substdefs

substdefs使用方法如下。

```
#!substdefs "/ID/subst/"
```

substdefs类似于substdef，但额外增加的#!define ID "subst"定义中，"subst"值会带有双引号，适用于字符串需要带双引号的场景。

3.1.5 核心关键字

核心关键字仅对SIP消息处理有效，主要用于if表达式中的判断。

(1) **af**：即Address Family，IP地址的类型，取值有INET和INET6，分别对应IPv4和IPv6。如以下代码会在SIP消息来自IPv6地址时打印日志。

```
if (af == INET6) {
    log("这条 SIP 消息使用了 IPv6\n");
}
```

(2) **dst_ip** : Dest IP, 即目的IP地址（收到本SIP消息的本地的IP地址），在系统中有多个网卡和IP地址的情况下，可以通过它判断目的IP地址是哪一个。使用示例如下。

```
if (dst_ip == 127.0.0.1) {
    log("这条 SIP 消息是在回环网络接口 (Loopback Interface) 上收到的\n");
};
```

注意，这里的127.0.0.1不要加双引号，如果加上系统会认为其是字符串进而会进行DNS反向地址解析，消耗系统资源。

(3) **dst_port** : Dest Port, 即目的端口（收到消息的SIP端口）。如以下代码在目的地端口为5061时打印日志。

```
if (dst_port == 5061) {
    log("这条 SIP 消息是在 5061 端口上收到的\n");
};
```

(4) **from_uri** : 从From头域获取到的URI。如下代码表示收到INVITE消息中的from_uri与".*@xswitch.cn"正则表达式匹配时打印日志。

```
if (is_method("INVITE") && from_uri =~ ".*@xswitch.cn") {
    log("主叫来自 xswitch.cn\n");
};
```

(5) **method** : SIP消息中的方法名。如下代码表示收到REGISTER（注册）消息时打印日志（与is_method("REGISTER")等价，参见上例）。

```
if (method == "REGISTER") {
    log("收到一条注册 (REGISTER) 消息\n");
};
```

(6) **msg:len** : SIP消息长度。如下代码表示收到SIP消息的长度超过2048时回复“413消息超长”并退出。

```
if (msg:len>2048) {
    sl_send_reply("413", "message too large");
    exit;
};
```

(7) **proto** : SIP消息底层承载协议类型。如下代码判断SIP消息是否基于UDP进行承载，若是则打印日志。

```
if (proto == UDP) {
    log("SIP 消息使用了 UDP 协议承载\n");
};
```

(8) **status** : 状态码，一般用于onreply_route路由块中，获取响应消息的状态。如果用于标准的路由块中，则它指最后一次发出的状态码。如下代码表示当收到的回复消息中状态码为200时打印日志。

```
if (status == "200") {  
    log(" 收到一条 200 OK 回复消息 \n");  
}
```

(9) **snd_af** : Send Adress Family, 即将要发送的地址类型, 在onsend_route路由块中有效。参见关于af的介绍。

(10) **snd_ip** : Send IP, 即将要使用的发送IP地址, 在onsend_route路由块中有效。参见关于dst_ip的介绍。

(11) **snd_port** : Send Port, 即将要使用的发送端口, 在onsend_route路由块中有效。参见关于dst_port的介绍。

(12) **snd_proto** : Send Protocol, 即发送协议, 在onsend_route路由块中有效。参见关于proto的介绍。

(13) **src_ip** : SIP消息的来源IP地址。如下代码判断来源IP地址（注意IP地址不要加引号）为127.0.0.1时打印日志。

```
if (src_ip == 127.0.0.1) {  
    log(" 该 SIP 消息来自 localhost ! \n");  
}
```

(14) **src_port** : SIP消息上一跳的来源端口。示例代码如下。

```
if (src_port == 5061) {  
    log(" 该 SIP 消息的源端口是 5061\n");  
}
```

(15) **to_ip** : To IP, 在onsend_route路由块中有效, 为SIP消息To头域中的IP地址。

(16) **to_port** : To Port, To头域中的端口, 类似to_ip。

(17) **to_uri** : To URI, To头域中的URI, 类似to_ip。如下代码判断To URI是否匹配正则表达式, 若是则打印日志。

```
if (to_uri =~ "sip:.*@xswitch.cn") {  
    log(" 该 SIP 消息中的 To 域是 xswitch.cn\n");  
}
```

(18) **uri** : Request URI, 即请求URI。如下代码判断uri是否与正则表达式相匹配, 若是则打印日志。

```
if (uri =~ "sip:.*@xswitch.cn") {  
    log(" 该 SIP 消息的 Request URI 中的域是 xswitch.cn\n");  
}
```

3.1.6 核心值

核心提供的一些预定义的宏值（标志符）可以用于if表达式判断, 可以与核心关键字进行比较（如是否相等）, 简述如下。

1.myself

myself代表“自己”, 它是一个集合, 包括在配置文件中配置的本地的IP地址、主机名(hostname)、别名(alias)等, 主要用于检查收到的SIP消息是在“自己”的管辖范围(同一SIP集群)内处理, 还是要转发到外部其他的SIP服务器(别人的SIP服务器)进行处理。

可以使用alias指令向myself列表中添加别名。示例如下。

```
if (uri == myself) { // 判断 Request URI 是否应该自己处理  
    log(" 收到一条 SIP 消息，这是我的菜，让我来进行处理\n");  
};
```

注意

这里的“自己”并不表示Kamailio不转发该SIP消息，事实上，大多数情况下Kamailio都需要将SIP消息转发到下一跳，只是这个下一跳一般是“自己”内部的SIP服务器，而不是“别人”（如别的运营商）的服务器。

也可以使用is_myself()函数实现与myself同样的功能。

2.其他

其他核心标志符有INET（可用于检查来源是否是IPv4地址）、INET6（IPv6）、UDP（UDP协议，可以与proto关键字进行比较）、TCP（TCP协议，可以与proto关键字进行比较）、TLS（TLS协议，可以与proto关键字进行比较）、SCTP（SCTP协议，可以与proto关键字进行比较）、WS（WS协议，可以与proto关键字进行比较）、WSS（WSS协议，可以与proto关键字进行比较）。

下面是一些代码示例，因比较直观，就不多解释了。

```
if (af == INET) {  
    log(" 这条 SIP 消息使用了 IPv4\n");  
}  
  
if (proto == WSS) {  
    log(" 这条 SIP 消息是通过 WSS (安全的 WebSocket) 协议承载的\n");  
}
```

3.1.7 核心参数

Kamailio配置文件中有一些全局参数，这些参数是比较核心的，关乎整个系统的运行。略述如下。

(1) advertised_address: 通告地址，用于设置Via头域中的IP地址，用于内外网IP不一致的情况。默认情况下Via头域中会使用本地用于发送SIP消息的IP地址。举例如下。

```
advertised_address = "1.2.3.4"  
advertised_address = "kamailio.org"
```

注意

该参数未来可能会取消（因为它是全局的）。推荐使用listen指令的advertise参数代替它，该参数仅对listen指定的网络接口有效，因而不同的网卡，IP地址或端口可以有不同的通告地址。

(2) advertised_port: 设置Via头域中的端口，参见advertised_address。举例如下。

```
advertised_port = 5080
```

提示

该参数也可能被取消，原因与advertised_address相同。

(3) alias: 设置别名，以便myself可以检查是否是本地需要处理的主机名或域名。有必要同时设置端口，否则loose_route()将不能按预期工作。举例如下。

```
alias = other.domain.com:5060
alias = another.domain.com:5060
```

提示

如果域名中有与Kamailio冲突的关键字，必须用引号引起来，如forward、exit、drop，甚至包括-等。

(4) **async_workers**: 指定default组中以异步调用方式启动了多少个子进程（Worker进程）。这些子进程用于执行异步任务，异步任务可能来自async、acc、sqlops等模块。默认为0，即异步功能不启用。举例如下。

```
async_workers = 4
```

(5) **async_nonblock**: 设置default组的异步进程中的内置Socket是否为阻塞模式。默认为0，即为阻塞模式。举例如下。

```
async_nonblock = 1
```

(6) **async_usleep**: 设置不同任务间需要等待的微秒值，在async_nonblock=1时有效，默认值为0。举例如下。

```
async_usleep = 100
```

(7) **async_workers_group**: 对异步进程进行分组，具体语法如下。

```
async_workers_group = "name=X;workers=N;nonblock=[0|1];usleep=M"
```

对上述设置中的相关属性解释如下。

- **name**: 组名，用于需要异步处理的函数，如sworker_task(name)。
- **workers**: 表示分组中需要建立多少个子进程。
- **nonblock**: 将内部通信Socket设为无阻塞模式。
- **usleep**: 设置不同任务间需要等待的微秒值，在async_nonblock=1时有效。

async_workers_group的默认值为空字符串（""），下面是一个示例。

```
async_workers_group="name=reg;workers=4;nonblock=0;usleep=0"
```

如果name为default，它会覆盖掉async_workers等相关指令设置的值。

关于异步任务，可以参见event_route[core:pre-routing]事件以及sworker模块，在本书后面会给出一个用Lua实现的示例（参见6.5.1节）。

(8) **auto_aliases**: Kamailio默认会监听所有IP地址，并对所有IP地址做DNS反向解析，将结果加入别名列表，以便可以进行myself检测。将该参数设为no可以禁用反向DNS解析。举例如下。

```
auto_aliases = no
```

(9) **auto_bind_ipv6**: 默认为0，不启用。如果启用，则会像IPv4那样自动绑定所有IPv6地址。

```
auto_bind_ipv6 = 1
```

(10) bind_ipv6_link_local: 默认为0, 如果设为1, 则将绑定IPv6本地回环地址, 目前仅支持UDP。举例如下。

```
bind_ipv6_link_local = 1
```

(11) check_via: 检查回复消息中最上面的Via地址是否为本地地址, 默认为0, 不检查。

(12) children: 表示每个UDP监听多少个子进程, 默认为8。如果你有多个 (n个) UDP监听地址, 则将启动 $n \times 8$ 个子进程。举例如下。

```
children = 4
```

对于TCP/TLS监听, 使用tcp_children。

(13) chroot: 决定是否支持chroot, 该参数的值必须是一个合法的chroot路径。chroot是一个操作系统特性, 其通过将当前进程能访问的文件系统切换到一个独立的“沙箱”环境, 以提高系统安全性。举例如下。

```
chroot = /other/fakeroot
```

(14) corelog: 设置Kamailio核心的调试日志级别, 默认为-1 (即L_ERR), 值越大打印的日志越多。有时出现一些错误日志并不意味着真的出错了, 比如在解析SIP消息时出错, 原因可能是对端发送了不合法的SIP消息, 这不是Kamailio本身的问题。

corelog使用方法如下。

```
corelog = 1
```

具体的日志级别可以参见下面要讲的debug参数。

(15) debug: 设置调试级别, 值越大消息越多, 默认值为0。建议生产环境中在-1到2之间取值。下面是取值列表。

L_ALERT	-5
L_BUG	-4
L_CRIT2	-3
L_CRIT	-2
L_ERR	-1
L_WARN	0
L_NOTICE	1
L_INFO	2
L_DBG	3

使用建议如下。

debug=3: 打印所有日志, 用于调试, 不建议用于生产环境, 因为这将产生大量日志, 不仅会拖慢系统还会塞满硬盘空间。

debug = 0: 只打印警告、错误和严重错误信息。

debug=-6: 禁用所有日志。

可以使用RPC接口在运行时动态控制这些参数。示例如下。

```
kamcmd cfg.get core debug          # 获取核心 debug 参数值  
kamcmd cfg.set_now_int core debug 2    # 将核心 debug 参数值设为整数 2  
kamcmd cfg.set_now_int core debug -- -1 # 设为 -1，由于 -1 包含一个 - 号，因此前面用 -- 避免歧义
```

与内存有关的日志参见memlog和memdbg。更多信息参见
<https://www.kamailio.org/wiki/tutorials/3.2.x/syslog>。

(16) disable_core_dump: 禁用Core Dump（内核转储），取值可为yes或no，默认值为no。默认情况下Core Dump的值为unlimited或一个很高的值，Kamailio在崩溃时会将崩溃瞬间的内存快照写入一个core文件，Core Dump过大将消耗大量的文件系统空间。该参数可以禁用Core Dump。举例如下。

```
disable_core_dump = yes
```

(17) disable_tls: 又称tls_disable，用于决定是否禁用TLS。默认为否，但是需要加载TLS模块



才能用（该模块的使用方法可以参考10.3节）。举例如下。

```
disable_tls = yes
```

(18) enable_tls: 又称tls_enable，决定是否启用TLS，与disable_tls相反，默认值为启用。

(19) force_rport: 决定是否强制启用rport机制，取值为yes或no，类似于force_rport()函数，但全局有效。

rport是一种NAT穿越机制，启用后，SIP消息中将带有“;rport”参数，如果Kamailio位于NAT后面（NAT内）并发出SIP消息，对方在回复时，可以向从接收到的SIP消息中学习到的源地址发送相关信息，而不是根据SIP协议发向Via或Contact地址（这些地址可能都在NAT后面）。

(20) fork: 如果将该参数设为yes，则系统会启动到后台，此时会启动很多进程，每个IP地址以及每个协议都会启动children倍数的进程。

如果将该参数设为no，则仅启用它找到的第一个IP地址，相当于Kamailio使用-F选项启动。

(21) fork_delay: 在慢启动时，该参数表示在启动每个进程之前暂停多少微秒，默认为0。举例如下。

```
fork_delay = 5000
```

(22) group: 又称gid，即Group ID，用于设置Kamailio启动后的进程组ID。举例如下。

```
group = "siprouter"
```

(23) http_reply_parse: 又称http_reply_hack，当其启用时（值为yes），Kamailio可以以SIP的方式解析HTTP请求的返回结果（但不会当成SIP消息被处理）。默认为no，即无法解析HTTP返回结果。

(24) ip_free_bind: 又称ipfreebind或ip_nonlocal_bind。设置Kamailio是否可以监听本地不存在的IP地址。在双机热备（常见形式为HA，全称为High Availability，即主备高可用）的情况下，对外服务的IP地址称为业务IP地址或浮动IP地址，该IP地址只会绑定到主机上，而在备机上启动Kamailio时若尝试监听浮动IP地址将出错。通常将该参数设为1，这样在备机上没有绑定浮动IP地址的情况下启动Kamailio进行监听也不会出错。举例如下。

```
ip_free_bind = 1
```

(25) **listen:** 设置服务器监听的协议和地址。该指令可以有多行，以便监听多个IP地址。举例如下。

```
listen = 10.10.10.10
listen = eth1:5062
listen = udp:10.10.10.10:5064
```

如果没有listen指令，则Kamailio将监听所有IP地址。Kamailio启动时会打印它监听的所有IP地址。如果使用IPv6地址，则应该有中括号。举例如下。

```
listen=udp:[2a02:1850:1:1::18]:5060
```



可以使用**advertise**指定一个通告IP地址（一般是外网IP地址），它将影响Via头域以及Record-Route消息头中的IP地址，举例如下。

```
listen = udp:10.10.10.10:5060 advertise 11.11.11.11:5060
```

通告IP地址必须是ip:port形式的，协议将从监听的Socket上取得。该参数常用的场景是Kamailio处于NAT环境中，实际监听的IP地址（本机内网IP地址，如本例中的10.10.10.10）与外网IP地址（如本例中的11.11.11.11）不一致。

监听时可以指定一个唯一的名字（name），以简化外发SIP消息时选择Socket的流程。如rr和path模块可以使用这个name来更快找到外发后续SIP消息使用的相应的Socket。

name属性值必须是有引号的字符串。举例如下。

```
listen = udp:10.0.0.10:5060 name "s1"
listen = udp:10.10.10.10:5060 advertise 11.11.11.11:5060 name "s2"
listen = udp:10.10.10.20:5060 advertise "mysipdomain.com" name "s3"
listen = udp:10.10.10.30:5060 advertise "mysipdomain.com" name "s4"

route {
    ...
    $fsn = "s4"; // SIP消息将从这个名称对应的 Socket 上发出
    t_relay();
}
```

注意

系统内部并没有针对name进行唯一性检查。如果多个listen有同一个name，则使用第一个找到的name。

(26) **loadmodule:** 加载模块。模块路径在loadpath或mpath指定的路径中查找。如果模块路径中只有模块名字或“名字.so”，则Kamailio会尝试加载“名字/名字.so”，这在使用源代码编译的环境中非常有用。举例如下。

```
loadpath "/usr/local/lib/kamailio:/usr/local/lib/kamailio/modules/"

loadmodule "/usr/local/lib/kamailio/modules/db_mysql.so"
loadmodule "modules/usrloc.so"
loadmodule "tm"
loadmodule "dialplan.so"
```

(27) **loadmodulex:** 类似于loadmodule，但其参数中可以有变量，而loadmodule的参数只能是字符

串。

(28) `loadpath`: 又称`mpath`, 用于设置模块搜索路径, 可以用“`:`”分隔多个路径。Kamailio在加载时将顺序查找“路径名/模块名.so”或“路径名/模块名/模块名.so”。举例如下。

```
loadpath "/usr/local/lib/kamailio/modules:/usr/local/lib/kamailio/mymodules"  
loadmodule "uri"  
loadmodule "tm"
```

(29) `local_rport`: 类似于`add_local_rport()`函数, 但全局有效, 默认为`off`。举例如下。

```
local_rport = on
```

(30) `log_engine_data`与`log_engine_type`: 设置日志引擎相关的数据。具体的日志引擎在模块中实现, 不同模块对数据的支持不同。举例如下。

```
log_engine_type = "udp"  
log_engine_data = "127.0.0.1:9"
```



更多信息可参见`log_custom`模块 的相关说明。



(31) `log_facility`: 如果Kamailio要将日志写入`Syslog`, 则可以通过该参数设置与日志对应的`Facility`, 这在将日志写入独立的日志文件时非常有用。具体的参数含义请参考系统`syslog(3)`的相关说明。该参数默认值为`LOG_DAEMON`, 也可以改为其他的值。举例如下。

```
log_facility = LOG_LOCAL0
```

(32) `log_name`: 设置`Syslog`的日志前缀, 又称`Syslog Tag`, 默认值为应用程序的名字(`kamailio`)或完整路径。当在同一台服务器上运行多个Kamailio实例时可用于区分日志。举例如下。

```
log_name = "kamailio-proxy-1"
```

(33) `log_prefix`: 用于在日志行中添加一个前缀。前缀可以引用与在运行时SIP消息解析后对应的变量, 具体的变量求值方法与`log_prefix_mode`有关。

该参数设置仅对处理SIP消息的路由块有效, 对于没有SIP消息的路由块, 如在`timer`、`evapi`的`Worker`进程中, 该参数无效。

看下面的示例, 其中日志前缀设为消息类型(1为请求, 2为响应) +CSeq+Call-ID:

```
log_prefix = "{Smt Shdr(CSeq) $ci} "
```

(34) `log_prefix_mode`: 配置`log_prefix`的求值方法, 默认值为0, 仅在收到SIP消息时求值一次, 适用于变量不变(后续的配置变量不会改变变量的值)的场景。如果将该参数设置为1, 则变量将在每次配置变更后都重新求值, 这适用于可能由于配置变更引起变量变化的情况, 如`$cfg(line)`。举例如下。

```
log_prefix_mode = 1
```

注

(35) `log_stderr`: 控制Kamailio是否将日志输出到标准错误，取值有yes和no，默认是后者。举例如下。

```
log_stderr = yes
```

(36) `cfgengine`: 设置路由块的解析引擎，默认值为"native"（等价于"default")，其他取值由加载的模块导出的名称决定，如"lua"由app_lua模块导出。举例如下。

```
cfgengine = "lua"
```

(37) `maxbuffer`: 在自动探测阶段可以使用的最大内存字节数，默认值为262144。举例如下。

```
maxbuffer = 65536
```

(38) `max_branches`: 设置对于一个SIP Transaction生成的最大分支数量。分支通常由核心的append_branch()函数生成，也可由tm模块的并行或串行转发功能生成。该参数默认值为12，取值区间为1~31。举例如下。

```
max_branches = 16
```

(39) `max_recursive_level`: 最大递归深度，在调用子路由块或if...else语句时都会增加递归深度，默认值为256。举例如下。

```
max_recursive_level = 500
```

(40) `max_while_loops`: 设置最大循环次数，默认值为100，其作用是防止产生死循环。如果设为0，则保护将会失效，但在明显的死循环语句出现时（如while(1){...}）仍会打印警告日志。举例如下。

```
max_while_loops = 200
```

(41) `mcast`: 该参数用于设置组播的网卡，如果不设置该参数，操作系统会根据内核的路由表对网卡进行选取。该参数会在每一个listen参数后重置，所以不会有副作用。举例如下。

```
mcast = "eth1"  
listen = udp:1.2.3.4:5060
```

(42) `mcast_loopback`: 设置组播包是否发往loopback地址，取值为yes或no，默认为后者。举例如下。

```
mcast_loopback = yes
```

(43) `mcast_ttl`: 设置组播TTL（Time to Live），默认使用操作系统的默认值（通常为1）。举例如下。

```
mcast_ttl = 32
```

(44) `memdbg`: 又称mem_dbg。该参数用于指定内存调试器的日志级别。如果memdg生效，则内存管理器范围内的任何与内存相关的请求（alloc、free等）都会被记录（如果在编译时启用了NO_DEBUG宏，则该参数无效）。默认值为L_DBG，即memdbg=3。如果设置了memdbg=2，则只

有当debug大于或等于2时，内存调试日志才会触发。

(45) memlog: 又称mem_log，该参数用于指定内存统计的日志级别。如果开启了memlog，Kamailio会记录内存统计信息，该信息会在关闭Kamailio或Kamailio收到SIGUSR1信号时打印，以方便调试内存泄漏相关的问题。默认值为L_DBG，即memlog=3。运行机制与memdbg相同。

(46) 其他内存参数: Kamailio还有几个其他内存相关的参数，如mem_join、mem_safety、mem_summary，这些参数都与开发调试相关，我们就不多介绍了。



(47) mhomed: 该参数用于在多网卡环境下设置Kamailio外发SIP消息的选择策略。默认值为0，即不开启，因为开启后会比较费资源。在默认情况下，Kamailio会使用收到消息的那个网卡转发SIP消息，如果转发时协议有变化（如TCP进UDP出），或在Kamailio单独外发消息时，会选择它能找到的第一个网卡进行外发，而不管目的地址是什么。事实上，Kamailio的很多模块都支持手动指定网卡（或网卡IP地址）的参数或函数，这些参数或函数可以静态配置或在路由脚本中动态指定。

如果将mhomed设为1，则Kamailio会尝试选择一个能到达对方的网卡（网址），基本逻辑是这样的：Kamailio向目标IP地址打开一个UDP Socket，然后获取与该Socket对应的本地IP地址（由操作系统根据路由表算出），然后关闭该Socket，而取到的IP地址可用于Via或Record-Route头域。举例如下。

```
mhomed = 1
```

(48) mlock_pages: 锁定内存页以避免被操作系统换到swap内存（通常位于更慢的硬盘）上，取值有yes或no，默认为no。这是个策略问题，如果Kamailio跟其他系统共用内存，那么开启该参数Kamailio会抢占一些先机。但如果Kamailio是专机专用，则在内存实在不够用又不能倒换时，会导致申请内存失败。

(49) modinit_delay: 模块初始化后等待的时间间隔，默认值为0，单位是微秒。主要用于对单位时间内连接频率有限制的系统（如数据库等），其可使这些系统启动慢一些以免连接失败。举例如下。

```
modinit_delay = 100000
```

(50) modparam: 该参数类似于一个函数，用于设置模块参数。具体的模块参数和值因模块而异。该参数我们在前面已多次讲过了，不再赘述。举例如下。

```
modparam("usrloc", "db_mode", 2)
modparam("usrloc", "nat_bflag", 6)
```

(51) onsend_route_reply: 决定是否执行onsend_route路由块，默认为0，即禁用，可以使用1、yes、on等值开启。在Kamailio收到一个SIP回复消息且该消息即将被转发出去时会调用onsend_route路由块。举例如下。

```
onsend_route_reply = yes
```

(52) open_files_limit: 操作系统通常对一个进程打开文件的数量有限制（典型的值是1024，由操作系统ulimit设置），如果该数值大于系统限制，则Kamailio会尝试将系统限制提升为该参数设置的值（当然Kamailio必须以root方式启动）。文件、Socket连接等都会占用文件数，尤其在使用TCP连接时，每一个连接都会占用一个连接数。举例如下。

```
open_files_limit = 2048
```

(53) phone2tel: 该参数启用后, Kamailio会检查SIP URI中是否带有user=phone参数, 如果带有该

注

参数, 则将SIP URI转换为TEL URI。该参数默认值为开启状态(值为1), 可以用以下方法关闭。

```
phone2tel = 0
```

(54) pmtu_discovery: 该参数如果启用, 则会在发送消息时设置IP包的DF(Don't Fragment, 禁止分片)位, 默认值为0。举例如下。

```
pmtu_discovery = 1
```

(55) port: 设置Kamailio的监听端口, 默认为5060。举例如下。

```
port = 5080
```

(56) sip_warning: 默认为0, 如果设为1, 则所有Kamailio产生的SIP回复消息中都会多一个Warning头域。该头域会包含一些内部调试信息, 但可能会泄露内部网络架构, 故在生产环境中要慎用该参数。举例如下。

```
sip_warning = 0
```

(57) socket_workers: 设置每一个监听Socket的Worker进程数。常用于listen之前。当用于listen之前时, 对于UDP或SCTP Socket, 它会覆盖掉children或sctp_children设置的值。对于TCP或TLS, 它会增加额外的TCP进程数, 这些进程只处理它们监听的Socket上的消息(如UDP消息不会被分发到这些Work中进行处理)。该参数的值会在每个listen参数后重置为默认值, 所以, 如果有多个listen并不想使用默认值的话, 则每次都要重设。如果不使用该参数, 则children、tcp_children以及sctp_children会起作用。我们来看如下配置。

```
children = 4
socket_workers = 2
listen = udp:127.0.0.1:5080
listen = udp:127.0.0.1:5070
listen = udp:127.0.0.1:5060
```

使用上述配置Kamailio将启动两个Worker进程用于处理5080端口的消息, 由于第一个listen后socket_worker值被重置(失效), 因此5070和5060端口都会根据children参数的设置启动4个进程, 总共有10个进程。

再来看如下配置。

```
children = 4
socket_workers = 2
listen = tcp:127.0.0.1:5080
listen = tcp:127.0.0.1:5070
listen = tcp:127.0.0.1:5060
```

上述配置一共有6个进程: 先启动2个进程, 专门处理5080端口的消息; 再启动4个进程, 每个进程都会处理5070和5060端口的消息。

(58) statistics: Kamailio有内置的统计功能, 统计使用计数器实现。计数器可以被读、写和清除。计数器可以在核心中定义(如tcp计数器), 也可以在外部模块中定义(如tmx模块中的2xx_transactions)。计数器可以由核心自动更新(如tcp计数器), 也可以通过statistics模块提供的函数在路由脚本或通过MI命令更新。\$stat()是一个只读的伪变量, 可用于读取计数器的值。举例如

下。

```
modparam("statistics", "variable", "NOTIFY") // 定义一个 NOTIFY 计数器  
if (method == "NOTIFY") {  
    update_stat("NOTIFY", "+1"); // 更新计数器  
}  
  
xlog("Number of received NOTIFYs: $stat(NOTIFY)"); // 打印计数器值
```

命令行示例如下。

```
kamctl fifo get_statistics NOTIFY # 获取 NOTIFY 计数器的值  
kamctl fifo reset_statistics NOTIFY # 将计数器重置为 0  
kamctl fifo clear_statistics NOTIFY # 获得计数器值并重置为 0  
  
kamcmd mi get_statistics lxx_replies # kamcmd 命令也类似
```

(59) **tos**: 该参数用于设置IP包的TOS(Type Of Service)值，对UDP和TCP都有效。举例如下。

```
tos = IPTOS_LOWDELAY  
tos = 0x10  
tos = IPTOS_RELIABILITY
```

上述代码中各参数的具体含义需要基于TCP/IP协议进行理解，限于篇幅，在此就不多解释了，读者可自行学习。

(60) **udp_mtu**: RFC 3261规定，所有SIP的实现必须都支持非可靠和可靠的传输层协议，即UDP和TCP（或TLS、SCTP）。这主要是因为TCP/IP层在网络上传输，在消息超过路由器的MTU值（通常为1500或更小）时会发生分片，而SIP常用的UDP在发生分片时无法保证完成有效重组，所以，如果SIP包比较长，则应该使用可靠的协议发送消息（如TCP或TLS）。RFC 3261建议使用1300，如果UDP包的大小超过1300KB则换用TCP发送，该值仅针对SIP消息的实际占用字节数，不包含TCP/IP层的开销。**udp_mtu**的默认值为0，即不会切换底层协议。举例如下。

```
udp_mtu = 1300
```

(61) **udp_mtu_try_proto**: 与**udp_mtu**配合使用，如果**udp_mtu**大于0，且SIP消息字节数大于**udp_mtu**指定的值，则换成该参数指定的协议发送相关消息。可参考**udp_mtu_try_proto(proto)**函数，该函数可以在路由脚本中使用。该参数的默认值为UDP，即不生效，如果需要使用，则推荐用TCP，其他可选值为TLS、SCTP。

注意

虽然RFC 3261规定所有SIP实现必须同时支持UDP和TCP，但并不是所有SIP实现都遵守这个规定。另外，对于某些NAT或防火墙内部的客户端TCP连接也不可达，因此，开启该参数要小心，一般来说开启这类参数的原则是：“确保你知道你在做什么。”

(62) **user**: 又称uid。运行Kamailio进程的操作系统为uid，Kamailio会通过suid（Set User ID）切换到该uid。举例如下。

```
user = "kamailio"
```

(63) **user_agent_header**: 设置User-Agent头域，且必须是完整的消息头域（包括“User-Agent:”，但不包括行结束符CRLF）。举例如下。

```
user_agent_header = "User-Agent: The Best SIP Server"
```

3.1.8 DNS相关参数

Kamailio有内置的DNS解析器，该解析器支持DNS缓存，默认是启用的。内置的DNS解析器启用后操作系统的解析器将不再起作用，因而写在“/etc/hosts”里的域名是不起作用的。如果内置DNS缓存被禁用（use_dns_cache=no），则会使用系统的DNS解析器。

tm模块的DNS失败转移功能须直接引用内部DNS缓存中的数据（为了节省内存），基于DNS的失败转移机制仅在内置DNS缓存启用时才有效。表3-1列出了内置DNS与系统DNS的对比。

表3-1 内置DNS与系统DNS对比

功 能	内置 DNS	系统 DNS
是否支持缓存解析后的 DNS 记录	是	否
是否支持 NAPTR/SRV 解析和权重	是	是
是否支持基于 DNS 的失败转移机制	是	否

如果启用内置DNS缓存，域名记录可以通过RPC命令以手动的形式添加或删除。除此之外，DNS还有很多配置参数可用于配置DNS的查询策略，受篇幅所限在此就不一一介绍了，详见源代码目录中的doc/tutorials/dns.txt文件。

3.1.9 TCP相关参数或选项

以下参数会影响Kamailio中TCP的行为，这些参数可以根据实际应用场景进行调整。

(1) disable_tcp: 用于设置是否全局禁用TCP，默认值为no，即不禁用。可以使用如下方法禁用。

```
disable_tcp = yes
```

(2) tcp_accept_aliases: 当收到一个TCP承载的SIP消息时，如果Via头域中包含alias参数，则后续的交互会创建一个新的TCP连接，其将连到Via指定的端口中。关于alias参数的作用可以参见IETF



关于SIP连接重用的标准draft-ietf-sip-connect-reuse-00.txt。Kamailio遵循该标准，但仅针对Via头域中的端口部分（Kamailio会忽略主机部分的查询，因为这通常需要进行DNS查询，用起来不太安全，而且在真实应用中是否真的有用也不确定）。

提示

在NAT穿越中，最好不用tcp_accept_aliases参数，而是使用nathelper模块中的fix_nated_[contact|register]函数进行相关处理。

tcp_accept_aliases参数默认值为no，可以使用以下方法改为yes。

```
tcp_accept_aliases = yes
```

(3) tcp_accept_haproxy: 启用HAProxy协议。开启该选项可以使内部的TCP协议栈在连接刚刚建立时接收PROXY-protocol-formatted消息头，PROXY-protocol-formatted消息头由HAProxy代理服务



定义，Kamailio支持HAProxy协议的v1（人类可读的文本格式）和v2（二进制格式）两个版本。当Kamailio位于TCP负载均衡器（如HAProxy或AWS的ELB）后面时通常需要该参数。负载均衡器可以通过PROXY-protocol-formatted消息头提供远端客户端的IP地址等信息，以便Kamailio可以做基于IP地址的ACL验证。

注意

开启tcp_accept_haproxy选项以后不符合HAProxy协议的普通TCP连接将会被拒绝。

tcp_accept_haproxy的默认值为no，可以改为yes。举例如下。

```
tcp_accept_haproxy = yes
```

(4) tcp_accept_hep3: 该参数使Kamailio接收HEP3数据包。HEP3是一种SIP封装格式，用于SIP监控，其他SIP服务器将SIP通过HEP3发送到Kamailio后，Kamailio可以将消息存入数据库，以便后续



查询分析。具体参见Homer Capture Server。

tcp_accept_hep3默认值为no，即不开启，可以通过以下方式开启。

```
tcp_accept_hep3 = yes
```

(5) tcp_accept_no_cl: 控制当收到的消息中没有Content-Length头域时是否报错。在SIP中，当使用TCP承载时，该头域是必须存在的，但是在基于HTTP 1.1的XCAP中却不是如此。后者在传输大消息时会分成很多段（Chunk）。开启该参数后最好在路由块中进行完整性检查，如果是SIP消息，就检查是否存在Content-Length头域，以验证消息是否符合RFC 3261的规定。

tcp_accept_no_cl默认值为no，即不开启，可以使用以下方法开启。

```
tcp_accept_no_cl = yes
```

(6) tcp_accept_unique: 默认值为0，如果设置为1，则会检查是否有来自相同IP地址和端口的TCP连接，如果有则拒绝。举例如下。

```
tcp_accept_unique = 1
```

(7) tcp_async: 又称tcp_buf_write，用于决定是否启用异步发送。如果启用，则所有阻塞的TCP连接和发送都会缓存到发送队列异步发送。默认值为yes。举例如下。

```
tcp_async = yes
```

提示

该参数对TLS也有效。

(8) tcp_children: 表示将要创建的TCP进程数，如果不设置该参数，则会使用children参数的值。举例如下。

```
tcp_children = 4
```

(9) tcp_clone_rcvbuf: 用于决定是否复制一份消息接收缓冲区收到的消息。默认值是0，表示不需要复制，直接从缓冲区解析就行（因为解析并不会破坏缓冲区），但如果需要用到在消息缓冲区中修改内容的函数，如msg_apply_changes()，就需要复制。举例如下。

```
tcp_clone_rcvbuf = 1
```

(10) tcp_connection_lifetime: 设置TCP连接存活时长，超过该时长，设置的非活动TCP连接将会被断开。该参数的默认值是120，单位为秒。如果设为0，则将导致TCP连接很快断开。

SIP协议并不需要一直保持TCP连接，如有需要，服务端会反向向客户端主动建立连接。但在实际

场景中可能会有很多客户端都位于NAT或防火墙后面，反向连接无法正常建立，这时候服务端就不应该主动断开连接。

一般来说SIP注册的最大有效时长是3600秒，建议设一个比它稍大的值。举例如下。

```
tcp_connection_lifetime = 3605
```

(11) `tcp_connection_match`: 默认值为0，如果设置为1，则会尝试更严格的Socket匹配方法，即会将本地端口也考虑在内，否则只考虑本地IP以及远端IP端口。举例如下。

```
tcp_connection_match = 1
```

(12) `tcp_connect_timeout`: 表示TCP客户端向远程发起连接的超时时长。在发起连接时，如果网络不可达，或在对端防火墙丢弃（DROP）所有IP包但没有明确拒绝时，就会发生超时。该参数默认值为10，单位为秒。可以将参数值调小以便更快地检测到网络问题。举例如下。

```
tcp_connect_timeout = 5
```

(13) `tcp_conn_wq_max`: 设置TCP异步发送的缓冲区的大小，每个连接一个缓冲区。如果缓冲区满了将会导致发送出错并断开连接。该参数默认值为32K（32768），若禁用`tcp_buf_write`，则该参数无效。举例如下。

```
tcp_conn_wq_max = 65536
```

(14) `tcp_crlf_ping`: 开启SIP TCP保活功能，周期性地发送CRLF+CRLF（两个回车换行符，即\r\n\r\n），对方会回复一个CRLF，该机制也称为Ping-Pong（乒乓）。该参数默认值为yes。举例如下。

```
tcp_crlf_ping = yes
```

(15) `tcp_defer_accept`: 延迟接收当前数据直到收到后续数据，这在一定程度上可以提高性能，尤其是在服务器上有很多TCP连接的情况下。该参数仅对Linux和FreeBSD有效，详细信息参见对应的tcp(7)TCP_DEFER_ACCEPT及ACCF_DATA(0)手册。举例如下。

```
tcp_defer_accept = yes # FreeBSD, 默认值为no, 即不启用  
tcp_defer_accept = 3   # Linux, 超时的秒数, 默认值为不启用该功能
```

(16) `tcp_delayed_ack`: 如果开启该参数，第一个ACK消息将会延迟到与第一个包含数据的包一起发送。该参数目前仅对Linux有效，参见linux tcp(7)TCP_QUICKACK。该参数在支持它的系统上（目前只有Linux）默认值是yes。举例如下。

```
tcp_delayed_ack = yes
```

(17) `tcp_fd_cache`: 如果启用该参数（值为yes），则发送的FD（File Description，与TCP连接对应的文件描述符）将会被缓存在调用`tcp_send`函数进行发送的进程中，这会提升一些性能，代价是TCP连接释放会慢一些，保持在打开状态的FD也会比不启用该参数时多一些。该参数的默认值为yes。举例如下。

```
tcp_fd_cache = yes
```

(18) `tcp_keepalive`: 启用TCP保活功能，参见Socket的SO_KEEPALIVE选项。注意，该参数是TCP层的，跟SIP层的Ping-Pong保活不同。该参数的默认值为yes。举例如下。

```
tcp_keepalive = yes
```

(19) `tcp_keepcnt`: 该参数用于设置有多少TCP保活包发出后，如果对方无响应，则断开连接。仅支持Linux操作系统。参见Socket的TCP_KEEP_CNT选项。该参数的默认值为不设置。举例如下。

```
tcp_keepcnt = 3
```

(20) `tcp_keepidle`: 用于设置TCP空闲（没有任何数据发送）多长时间（秒）后发保活包。仅支持Linux。参见Socket的TCP_KEEP_IDLE选项。该参数的默认值为不设置。举例如下。

```
tcp_keepidle = 120
```

(21) `tcp_keepintvl`: 表示在上一次检查失败的情况下，保活检查时间间隔（秒）。具体见Socket的TCP_KEEPINTVL选项。仅支持Linux操作系统。该参数的默认值为不设置。举例如下。

```
tcp_keepintvl = 10
```

(22) `tcp_linger2`: 表示处于FIN_WAIT2状态的TCP连接的存活时间，会覆盖`tcp_fin_timeout`参数。参见linux `tcp(7)`TCP_LINGER2。仅支持Linux。该参数的默认值为不设置。

```
tcp_linger2 = 10
```

(23) `tcp_max_connections`: 表示最大TCP连接数，超过该值将不再接受（accept）新连接。默认值在操作系统的`tcp_init.h`文件中定义，如“#define DEFAULT_TCP_MAX_CONNECTIONS 2048”。举例如下。

```
tcp_max_connections = 4096
```

(24) `tcp_no_connect`: 设为yes可禁止Kamailio向外发起TCP连接（对TLS也有效）。该值也可以在运行时通过“`kamcmd cfg.set_now_int tcp no_connect 1`”修改。

(25) `tcp_poll_method`: 设置使用的选举（poll）方法（操作系统针对多个TCP连接采用的调度策略），默认情况下会选择最适合的方法。该参数的值可以在操作系统源代码的`io_wait.c`及`poll_types.h`中找到，如none、poll、epoll_lt、epoll_et、sigio_rt、select、kqueue、/dev/poll等。举例如下。

```
tcp_poll_method = select
```

(26) `tcp_rd_buf_size`: 用于设置TCP读缓冲区的大小，如果连接数比较少，但每个连接上的消息比较多，那么增大该参数值可以提高性能，但需要占用更多内存。在连接数多的系统上最好将该值设得小一点以节省内存。该值也会影响到通过TCP接收的SIP或HTTP消息的最大值。

该参数的值在源代码中有一个固定上限（目前是16MB），如果需要更大的值，则需要修改源代码并重新编译Kamailio。在某些情况下，你需要在运行时分配更多的私有内存或共享内存（可以在Kamailio启动参数中设置）。

`tcp_rd_buf_size`的默认值为4096，可以在运行时修改。静态配置如下。

```
tcp_rd_buf_size = 65536
```

(27) `tcp_send_timeout`: 当Kamailio需要发送数据时，如果超过该参数设置的秒数还无法成功发送，则断开TCP连接。默认值为10秒。将该参数的值改小一些有助于更快地检测到坏掉的TCP连接。举例如下。

```
tcp_send_timeout = 3
```

(28) `tcp_source_ipv4`、`tcp_source_ipv6`: 设置所有外连TCP的源地址（IPv4及IPv6），若设置失败则使用默认IP地址。举例如下。

```
tcp_source_ipv4 = 127.0.0.1 # IPv4 地址  
tcp_source_ipv6 = ::1       # IPv6 地址
```

(29) `tcp_syncnt`: 设置SYN包的最大重传次数，超过该值将断开TCP连接。可参考linux `tcp(7)TCP_SYNCNT`。仅在Linux系统下有效。该参数的默认值为不设置。举例如下。

```
tcp_syncnt = 5
```

(30) `tcp_wq_max`: 用于设置全局允许缓存的最大发送字节数。仅当`tcp_buf_write`启用时有效。该参数的默认值为10MB，举例如下。

```
tcp_wq_max = 字节数
```

(31) `tcp_reuse_port`: 重用TCP端口。一般情况下，如果Kamailio作为UAS监听了5060端口，则它作为UAC在外发起连接时本端的端口就不能是5060了，而是由操作系统随机选择一个端口。启用该参数后会允许TCP通过指定的端口向外发起连接（即使已被占用）。该参数使用Socket的`SO_REUSEPORT`选项，Linux（内核版本要高于3.9.0版本）、FreeBSD、OpenBSD、NetBSD、MacOSX对该选项均有支持。该参数当且仅当在编译Kamailio以及运行Kamailio的操作系统上都支持`SO_REUSEPORT`选项时才生效。该参数的默认值为no，即不启用TCP端口。可以使用如下方法启用该参数。

```
tcp_reuse_port = yes
```

3.1.10 TLS相关参数

很多TLS相关的属性都可以在`tls`模块参数中设置，这里仅介绍核心中的TLS参数。

(1) `tls_port_no`: 用于设置Kamailio监听的TLS端口。该参数的默认值为5061。举例如下。

```
tls_port_no = 6061
```

(2) `tls_max_connections`: 用于设置TLS最大连接数，与TCP最大连接数类似。该参数的默认值为2048。举例如下。

```
tls_max_connections = 4096
```

3.1.11 SCTP概述



SCTP（Stream Control Transmission Protocol，流控制传输协议）是一个传输层协议，提供的服务有点像TCP，但又结合了UDP一些优点，可以提供可靠、高效、有序的数据传输协议。相比之下，TCP针对的是字节类的消息，而SCTP针对的是帧类的消息。

SCTP主要的贡献是对多重连外线路（多网卡）提供支持，一个终端可以由一个或多个IP地址组成，使得传输可在主机间或网卡间做到网络容错透明。

SCTP最初被设计用于在IP网上上传输电话协议（SS7，即七号信令），以求把SS7信令网络的一些可

靠特性引入IP网。IETF在这方面的工作称为信令传输（SIGTRAN）。

Kamailio有完善的SCTP支持，也有很多可调整的参数。不过支持SCTP的其他系统较少，因而业界很少使用。限于篇幅，这部分参数我们就不多介绍了。

3.1.12 UDP相关参数

UDP是主要的SIP承载协议。在SIP包字节数不超过全链路上最小MTU的情况下，UDP一般工作得很好。由于UDP是面向无连接的，所以不像TCP那样有连接数限制，性能也比较好。当然，UDP在NAT和防火墙穿越方面会比TCP差一些。下面是一些UDP参数。

(1) `udp4_raw`: `udp4_raw`用于设置是否启动Raw Socket。Raw Socket称为原始套接字，启用Raw Socket后应用程序可以自行组织UDP包头。在Kamailio中开启Raw Socket支持后在多CPU的环境下有40%~50%的性能提升。该参数支持以下值。

- 0: 禁用。
- 1: 启用。
- 1: 自动。

如果设为自动，则在支持Raw Socket的操作系统上Kamailio将尽量启用Raw Socket功能（Kamailio以root启动或非root用户使用CAP_NET_RAW权限启动）。Linux和FreeBSD都支持Raw Socket功能。对于其他BSD以及Darwin（Mac OS）系统，需要在编译时通过-DUSE_RAW_SOCKET配置参数启用支持。在Linux上，如果网络全链路上有MTU小于1500的网络设备，则还应该将`udp4_raw_mtu`设置为一个较小的值。

`udp4_raw`参数也可以在运行时设置（参数为`core.udp4_raw`）。`udp4_raw`的启用方法如下。

```
udp4_raw = on
```

(2) `udp4_raw_mtu`: 如果启用`udp4_raw`，也应该同时设置该参数值。该参数的值应该小于整个网络上全链路中所有设备中的MTU最小值。该参数默认值为1500。该参数仅在Linux上需要设置，在BSD系统的UNIX上，内核会自动使用合适的值。该参数也可以在运行时设置（通过`core.udp4_raw_mtu`参数进行设置）。

(3) `udp4_raw_ttl`: 设置Raw UDP Socket的TTL（Time to Live，生存时间）。如果启用`udp4_raw`，也可以设置该参数值。默认为自动（auto）模式，值为-1，即TTL与普通UDP Socket相同。该参数也可以在运行时设置（参数`core.udp4_raw_ttl`）。

3.1.13 核心函数

核心函数（Core Variables）主要用于路由块中，下面对核心函数进行简单介绍。

(1) `add_local_rport`: 在SIP消息的Via头域中增加rport参数。rport采用的是一种NAT穿越机制，告诉对方将回复消息发到接收消息来源地址，而不是向SIP协议中指定的IP地址发送。详情可参阅RFC 3581。举例如下。

```
add_local_rport();
```

(2) `break`: 类似于C语言中的break语句，可以从switch语句或while循环中跳出。

(3) `drop`: 表示丢弃SIP消息并停止路由脚本执行（包括后续的隐含动作）。例如，该函数在`branch_route`中被调用，则相应的分支会被丢弃（而不是执行隐含动作转发该消息）。如果该函数在默认的`onreply_route`中执行，则任何响应消息都会被丢弃。但如果在具名的`onreply_route`中执行（有状态事务转发），则只能丢弃临时响应消息（1xx），而对最终响应消息（2以上开头的消息）无效。详见如下代码注释。

```

onreply_route { // 默认响应处理函数
    if(status == "200") {
        drop(); // 有效，丢弃该响应消息
    }
}

onreply_route[FOOBAR] { // 具名响应处理函数
    if(status == "200") {
        drop(); // 无效，该函数会被忽略，因为这不是一个临时响应消息，无法丢弃
    }
}

```

(4) **exit**: 停止路由脚本的执行，等效于return(0)。它不会响应后续隐含的动作。举例如下。

```

route {
    if (route(2)) { // 调用另一个路由块并检查返回值
        xlog("L_NOTICE", "method $rm is INVITE\n"); // 返回值为1
    } else {
        xlog("L_NOTICE", "method is $rm\n"); // 返回值为-1或0都会执行这一句
    }
}

route[2] {
    if (is_method("INVITE")) {
        return(1);
    } else if (is_method("REGISTER")) {
        return(-1);
    } else if (is_method("MESSAGE")) {
        sl_send_reply("403", "IM not allowed");
        exit; // 等效于return(0)
    }
}

```

(5) **force_rport**: 又称add_rport。rport参数在RFC 3581中定义，用于帮助NAT穿越。该函数会在收到的SIP消息中的第一个Via头域增加rport参数，就如同对方（客户端）发送时带上了该参数。这样，Kamailio有回复消息时就会发到来源IP地址，而不是Via头域中的IP地址（这两个IP地址可能不一样），并在回复消息的第一个Via头域上增加received参数携带的来源IP地址和端口，以便同事务中后续的SIP消息交互都能顺利进行。

提示

核心参数中也有一个与force_rport同名的参数（参见3.1.7节），其可作用于全局，而此处的函数仅针对当前SIP消息所在的事务有效。

举例如下。

```
force_rport();
```

(6) **force_send_socket**: 强制待发SIP消息使用指定的Socket。该Socket必须是配置文件中listen指令指定的其中一个。如果协议不匹配（如UDP消息被强制发送到一个TCP Socket上），那么Kamailio将在与被发送消息使用的协议相同的协议（本例中是UDP）中自动选择一个与指定Socket最相近的Socket向外发送。该函数不支持伪变量，如果需要使用伪变量，可以使用corex模块中的set_send_socket函数代替该函数。举例如下。

```
force_send_socket(10.10.10.10:5060);
force_send_socket(udp:10.10.10.10:5060);
```

(7) **force_tcp_alias**: 等同于add_tcp_alias，其语法如下。

```
force_tcp_alias(port)
```

当kamailio收到一个TCP SIP消息时，会为当前的TCP Socket连接增加一个TCP端口别名，后续SIP消息可通过该别名指定向这个Socket发送，从而帮助实现NAT穿越。如果不指定port参数，则会从第一个Via头域中提取。当这个TCP连接断开时，所有与之相关的TCP端口别名均会被删除。

(8) forward: 无状态转发，将SIP消息转发到与\$du伪变量对应的目的地。举例如下。

```
$du = "sip:10.0.0.10:5060;transport=tcp";
forward();
```



(9) isflagset: 测试当前处理的SIP消息是否被设置了某一个flag。flag取值范围为0~31。举例如下。

```
if(isflagset(3)) {
    log("flag 3 is set\n");
};
```

Kamailio也支持命名flag，这种flag需要在配置文件的开始处定义。举例如下。

```
flags test, a:1, b:2; // 定义三个 flag
route{
    setflag(test);      // 设置 test 这个 flag
    if (isflagset(a)){ // 等价于 isflagset(1)
        ....
    }
    resetflag(b);      // 等价于 resetflag(2)
}
```

(10) is_int: 测试伪变量是否为整数值。举例如下。

```
if(is_int("$avp(foobar)")) {
    log("foobar contains an integer\n");
};
```



(11) log: 打印日志。举例如下。

```
log("just some text message\n");
```

(12) prefix: 在SIP消息的R-URI的用户名部分增加前缀字符串。例如，以下代码实现的是在所有被叫号码前增加两个0。

```
prefix("00");
```

(13) return: return函数用于在子路由块中返回，返回值为整型。在调用子路由块的代码中可以使用\$retcode或\$?测试返回值，return(0)相当于exit()。

在布尔表达式测试中，负数和0相当于false，正数相当于true。如果没有返回值，或者子路由块执行到最后没有return语句，则隐含返回1。参见以下示例。

```
route {
```

```

        if (route[2]) { // 调用子路由块并测试返回值
            xlog("L_NOTICE", "method $rm is INVITE\n");
        } else {
            xlog("L_NOTICE", "method $rm is REGISTER\n");
        }
    }

    route[2] {
        if (is_method("INVITE")) {
            return(1);
        } else if (is_method("REGISTER")) {
            return(-1);
        } else {
            return(0);
        }
    }
}

```

(14) revert_uri: 用于重置R-URI，即将R-URI设置为刚刚收到SIP消息时的R-URI值，即丢弃所有与之相关的修改。举例如下。

```
revert_uri();
```

(15) rewritehostport: 等同于sethostport、sethp，用于重写R-URI中的域部分，而用户部分不动。例如收到一个SIP消息，首行为如下形式。

```
INVITE 1234@1.2.3.4:5070 SIP/2.0
```

其中，1234为用户部分，1.2.3.4:5070为域部分，而1234@1.2.3.4称为R-URI，我们可以使用如下代码进行域重写。

```
rewritehostport("5.6.7.8:5080");
```

重写后，对应的SIP消息变为如下形式。

```
INVITE 1234@5.6.7.8:5080 SIP/2.0
```

(16) rewritehostporttrans: 等同于sethostporttrans、sethpt，功能与rewritehostport类似，但可以在增加transport参数的同时修改底层协议。举例如下。

```
rewritehostporttrans("1.2.3.4:5080;transport=tls");
```

(17) rewritehost: rewritehost的功能类似于rewritehostport，但仅修改主机部分，其他不动。举例如下。

```
rewritehost("5.6.7.8");
```

(18) rewriteport: 又称setport、setp。类似于rewritehostport，但仅修改端口，其他不动。举例如下。

```
rewriteport("5070");
```

(19) rewriteuri: 又称seturi，将当前URI重写为指定值。举例如下。

```
rewriteuri("sip:test@kamailio.org");
```

(20) rewriteuserpass: 又称setuserpass、setup。修改R-URI的用户名和密码部分。这是旧的方法，现在SIP一般都使用Digest实现相同的功能，不再使用此方法。举例如下。

```
rewriteuserpass("alice:password");
rewriteuserpass("password");
```

上述代码将R-URI修改为alice:password@ip:port，其中ip:port还保留原来的值。

(21) rewriteuser: 又称setuser、setu，用于重写R-URI的用户部分（可用于修改被叫号码）。举例如下。

```
rewriteuser("newuser");
```

(22) route: 用于执行次级路由块（类似于编程语言中的函数调用），其参数既可以是次级路由块的名字，也可以是字符串标志符，还可以由变量拼接而成。举例如下。

```
route(REGISTER_REQUEST);
route(@received.proto + "_proto_" + $var(route_set));
```

(23) set_advertised_address: 用于设置SIP消息的通告地址（通常是公网IP地址），与配置文件中的advertised_address指令相同但优先级更高，后者是全局的，而该函数仅作用于当前SIP消息所在的事务。举例如下。

```
set_advertised_address("xswitch.cn");
```

(24) set_advertised_port: 与配置文件中的advertised_port语句相同但优先级更高，仅作用于当前SIP消息所在的事务。举例如下。

```
set_advertised_port(5080);
```

(25) set_forward_no_connect: 在SIP协议中，对于TCP或TLS并不要求一直保持连接，如果没有可用的连接，Kamailio会自动建立一个。但在对端处于NAT或防火墙后的时候连接可能就无法建立。该函数就用于禁止建立新连接，并在转发SIP消息时查找一个已存在的连接，如果没有则转发失败。该函数仅用于面向连接的协议，如TCP、TLS及SCTP（功能暂未实现）等，对UDP无效。

具体的连接行为因路由块而异。

- 在普通路由块中，影响无状态转发和有状态的tm。对于有状态的tm，会影响所有的分支以及可能的重传（当然TCP或TLS没有重传）。
- 对于无状态onreply_route[0]，等效于set_reply_*(0)，建议使用set_reply_*(0)相关的函数。
- 对于有状态的onreply_route[非0值]无效。
- 对于branch_route，仅影响当前的分支，包含Transaction内所有消息，如CANCEL等。
- 对于onsend_route，则与branch_route类似。

示例：

```

route {
    if (lookup()) { // 查找注册用户的联系地址
        // 注册用户一般都位于NAT设备后面，反向TCP可能无法建立连接，禁用无意义的尝试
        set_forward_no_connect();
        t_relay(); // 转发，如果找不到对应的TCP连接，就会失败
    }
}

```

(26) `set_forward_close`: 在转发（forward）当前SIP消息后即断开连接。可用于设置了`set_forward_no_connect()`的路由块中。断开连接时可能导致无法收到响应消息。

(27) `set_reply_no_connect`: 类似于`set_forward_no_connect()`，但是仅针对响应消息（本地产生或转发的响应消息），具体行为因使用场景而不同。

□在普通路由块中，影响本事务中所有的响应消息（包括本地产生的或转发的）以及所有本地通过`sl_reply()`产生的无状态响应。

□`onreply_route`: 影响当前的响应消息，即设有`send_flags`的那一条消息（在多分支情况下胜出的那一个，包括最终响应和临时响应）。

□`branch_route`: 忽略。

□`onsend_route`: 忽略。

(28) `set_reply_close`: 类似于`set_forward_close`，只不过其仅作用于响应消息。

(29) `setflag`: 在当前处理的SIP消息上设置一个flag，取值范围为0~31。值的具体含义由用户决定，用于记录当前SIP消息的处理状态，以便进行一些特殊处理（典型的处理行为有写话单、记录消息是否经过鉴权等）。可以参考`isflagset`以便加深对`setflag`的理解。

(30) `strip`: 去掉R-URI中用户部分的前n位，俗称“吃位”。比如在企业PBX的场景中，打内线可直接拨小号，而打外线要加0，但是外面的落地线路不认识这个0，就可以在送出前将0“吃”掉，只发送后面的号码。举例如下（其中n=1代表吃掉1位）。

```
strip(1);
```

(31) `strip_tail`: 类似于`strip`，但吃掉号码尾部的n位。举例如下。

```
strip_tail(3);
```

(32) `udp_mtu_try_proto(proto)`: 其中proto的取值有TCP、TLS、SCTP等，类似于全局配置参数`udp_mtu_try_proto`。本函数仅作用于当前的SIP消息。举例如下。

```
if($rd == "10.10.10.10")
    udp_mtu_try_proto(TLS);
```

(33) `userphone`: 在R-URI上添加user=phone参数。

3.1.14 自定义全局参数

可以在Kamailio配置脚本（如`kamailio.cfg`）中自定义全局参数，而自定义的全局参数可以在路由块中使用，且参数值可以在运行时通过RPC命令修改，整个过程均无须重启Kamailio。自定义全局参数的格式如下。

```
group.variable = 值 desc "描述"
```

其中，`group`可以是任意字符串，表示一个参数分组，类似于一个命名空间。参数值可以是字符串

或整数。举例如下。

```
pstn.gw_ip = "1.2.3.4" desc "PSTN 网关地址"
```

可以使用如下方法在路由块中访问该参数的值。

```
$ru = "sip:" + $rU + "@" + $sel(cfg_get.pstn.gw_ip);
```

注意

有些字符串在Kamailio内部用作关键字，因而不能用于变量名（或变量名的一部分，即以“.”分隔的部分，如true不能用，但可以用truely），这些字符串有：yes、true、on、enable、no、false、off、disable、udp、UDP、tcp、TCP、tls、TLS、sctp、SCTP、ws、WS、wss、WSS、inet、INET、inet6、INET6、sslv23、SSLv23、SSLV23、sslv2、SSLv2、SSLV2、sslv3、SSLv3、SSLV3、tlsv1、TLSv1、TLSV1。

3.1.15 脚本语句

脚本语句类似于编程语言（如C语言）。在路由代码中可以使用如下脚本语句。

1.if ... else

if...else语句的语法如下：

```
if(expr) {
    actions;
} else {
    actions;
}
```

其中，expr为逻辑表达式，actions为执行的动作（函数或其他语句）。在expr中可使用以下逻辑运算符。

□ ==: 等于。

□ !=: 不等于。

□ =~: 正则表达式匹配，使用Posix Regular Expressions语法而不是PCRE，如使用“[:digit:]{3}”而不是“\d\d\d”匹配三位数字。关于Posix正则表达式的更多内容可参阅其他相关资料。

□ !~: 正则表达式不匹配。

□ >: 大于。

□ >=: 大于或等于。

□ <: 小于。

□ <=: 小于或等于。

□ &&: 逻辑与。

□ ||: 逻辑或。

□ !: 逻辑非。

□ [...]: 测试表达式，类似于Shell中的test，方括号中可以是任意数学表达式。

示例：

```
if(is_method("INVITE"))
{
    log(" 收到一条 INVITE 消息\n");
} else {
    log(" 收到一条消息，但不是 INVITE\n");
}
```

2.switch

脚本中的switch类似于大多数其他语言中的switch语句，测试变量是否匹配某一个分支。

注意，break只能在case中使用，如果想从路由块中其他地方返回，则需要使用return。

示例：

```
route {
    route(1); // 执行路由块 1
    switch($retcode) // 测试返回值
    {
        case -1;
            log("process INVITE requests here\n");
            break;
        case 1;
            log("process REGISTER requests here\n");
            break;
        case 2;
        case 3;
            log("process SUBSCRIBE and NOTIFY requests here\n");
            break;
        default:
            log("process other requests here\n");
    }

    switch($rU) // R-URI 的用户部分
    {
        case "101";
            log("destination number is 101\n");
    }
}
```

```

        break;
    case "102":
        log("destination number is 102\n");
        break;
    case "103":
    case "104":
        log("destination number is 103 or 104\n");
        break;
    default:
        log("unknown destination number\n");
    }
}

route[1]{
    if(is_method("INVITE"))
    {
        return(-1);
    };
    if(is_method("REGISTER"))
        return(1);
    }
    if(is_method("SUBSCRIBE"))
        return(2);
    }
    if(is_method("NOTIFY"))
        return(3);
    }
    return(-2);
}

```

注意

在子路由块中返回0（如return(0)）将会导致整个脚本停止执行，慎用。大家应尽量使用非0值。

3.while

脚本中的while表示循环语句，类似于大部分其他语言中的while。举例如下。

```

$var(i) = 0; // 初始化变量
while($var(i) < 10) // 如果变量值小于 10 则循环
{
    xlog("counter: $var(i)\n"); // 打印变量值
    $var(i) = $var(i) + 1; // 将变量加1
}

```

3.1.16 脚本操作符

可以在路由脚本中执行变量赋值等操作。

1.赋值

赋值语句类似于C或其他语言，通过=实现，左边是待赋值的变量，右面是值或值表达式。左侧可以使用以下变量。

- 非排序列表的AVP（见后面示例）。
- 变量，如\$var(...)。
- 共享内存变量，如\$shv(...).
- \$ru: 设置R-URI。
- \$rd: 设置R-URI的域（Domain）部分。

□ \$rU: 设置R-URI的用户（User）部分。

□ \$rp: 设置R-URI的端口（Port）部分。

□ \$du: 设置目的地URI（Dest URI）。

□ \$fs: 设置发送Socket。

□ \$br: 设置分支（branch）。

□ \$mf: 设置消息flag。

□ \$sf: 设置脚本flag。

□ \$bf: 设置分支flag。

示例：

```
$var(a) = 123;
```

对于AVP，可以直接删除所有已存在的值并设为一个新值。举例如下。

```
$avp(i:3)[*] = 123; // 设为123  
$avp(i:3)[*] = $null; // 设为空
```

2.字符串操作

+可用于字符串串联。举例如下。

```
$var(a) = "test";  
$var(b) = "sip:" + $var(a) + "@" + $fd; // 结果为 sip:test@from-domain
```

3.数学运算

Kamailio支持的数学运算符如表3-2所示。

表3-2 Kamailio支持的数学运算符

序号	运 算 符	含 义
1	+	加
2	-	减
3	*	乘
4	/	除以
5	mod	取模（类似其他语言中的%）
6		比特位或
7	&	比特位与
8	^	比特位异或

(续)

序号	运算符	含义
9	<code>~</code>	比特位取反
10	<code><<</code>	左移一比特位
11	<code>>></code>	右移一比特位

示例（为保障运算优先级推荐使用括号）如下。

```
$var(a) = 4 + ( 7 & (~2) );
```

数学运算可用于条件表达式中。示例如下。

```
if( $var(a) < 4 )
log("var 变量的右起第 3 位为 1\n");
# 4 = 0B 0000 0100, 因而从右数第三位为 1
```

4. 操作符

Kamailio有以下操作符。

□类型操作符：(int)将它后面的变量转为整型；(str)将它后面的变量转为字符串型。

□字符串比较：eq为等于；ne为不等于。

□整数比较：ieq为等于；ine为不等于。

当整型和字符串型比较时，会有隐含类型转换。

□`0 == ""`：返回true，其中""会隐含转换为整数0，相当于比较`0 == 0`。

□`0 eq ""`：返回false，比较时整数0会隐含转换为"0"，因而相当于"0"eq ""。

□`"a" ieq "b"`：返回true，隐含转换为`(int) "a" = 0`以及`(int) "b" = 0`。

□`"a"=="b"`：返回false。

注意

在Kamailio内部，只要有可能，`==`和`!=`都会在启动时自动转换为对应的eq、ne、ieq、ine等操作符。

对于比较操作符`+`、`==`和`!=`，Kamailio会试图猜出用户期望的类型并进行转换，一般是右侧迁就左侧的类型，但如果左侧是未定义的变量（用`undef`表示）则会导致无法适配。下面是一些转换原则。

□`+`：在`undef+expr`中，`undef`会被转换为空字符串，即`""+expr`。

□`==`和`!=`：在`undef==expr`中，`undef`会转换为跟`expr`同类型的值再进行比较（若`expr`也为`undef`，则`undef==undef`返回true，内部都会转换为字符串再比较）。

□表达式求值转换：自动转换为整型或字符串型，如`int(undef) == 0`、`int("") == 0`、`int("123") == 123`、`int("abc") == 0`、`str(undef) == ""`、`str(123) == "123"`。

□`defined expr`：如果`expr`已定义则返回true，否则返回false。注意，只有独立的avp或pvar可能是未定义的，其他的表达式都是已定义的。

□`strlen(expr)`：表示将返回表达式的值转换为字符串后的长度。

□ **strempty(expr)**: 如果表达式能转换为空字符串，则返回true，否则返回false。相当于expr==""。

示例：

```
if (defined $v && !strempty($v)) $len=strlen($v);
```

Kamailio是一个历史悠久的项目，支持的功能和特性也非常多，很多参数和特性又依赖于更深层次的知识，如chroot、Syslog以及Socket参数等，这些都跟操作系统甚至Socket通信理论相关。不过，不了解这些知识也不影响对本书的阅读，Kamailio默认的配置文件就提供了比较通用的默认值，我们把相关的参数列在这里是为了给大家一个简单的参考和直观的印象，以便大家更好地理解本书给出的脚本。

除了上述内容之外，Kamailio核心中还有其他一些参数和函数，但限于篇幅我们并没有一一列举。

上述内容足以帮大家阅读和理解本书内容，也足以应对日常使用问题，更多内容请参考

<https://www.kamailio.org/wiki/cookbooks-devel/core>。

3.2 其他概念和组件

上一节介绍了与core相关的概念和组件，这是Kamailio中最重要的部分之一。除了上面介绍的内容，还有一些与Kamailio相关的概念和组件需要大家了解。

3.2.1 伪变量

伪变量（Pseudo-Variables）是以\$开头的一些变量，可以获取路由中的相关数据。有的伪变量是只读的，如\$ci为SIP消息中Call-ID头域的值，只读的原因是显而易见的，因为你无法修改收到的SIP消息的Call-ID；有的变量则可以动态改变，如\$du是下一跳的URI，通过改变它可以改变SIP消息下一跳的地址。举例如下。

```
$du = "127.0.0.1:5060";
t_relay();
```



常用的伪变量简述如下。

- \$ru: 读取或设置R-URI。
- \$rd: 读取或设置R-URI中的Domain部分。
- \$rU: 读取或设置R-URI中的User部分。
- \$rp: 读取或设置R-URI中的Port部分。
- \$du: 读取或设置Dest URI。
- \$fs: 读取或设置发送Socket。
- \$br: 读取或设置Branch。
- \$mf: 读取或设置Message Flag。
- \$sf: 读取或设置Script Flag。
- \$bf: 读取或设置Branch Flag。
- \$ci: Call-ID，只读。
- \$si: 来源IP地址，只读。
- \$rm: SIP Method，只读。
- \$avp(id): 获取AVP中id的值，参见AVP。
- \$sht(htable=>key): 获取htable中key的值。

3.2.2 htable



htable（Hash Table）即哈希表，在共享内存中实现，主要用于缓存系统，可以通过DMQ在不同的Kamailio实例间同步。

哈希表可以使用伪变量存取，如\$sht(htname=>name)。哈希表可以在加载时从数据库读取数据，也

可以在运行时动态修改数据，非常适用于做缓存。

在第2章中我们使用哈希表防止洪水攻击时用到了ipban。见下面的代码，ipban是哈希表的名称，size是哈希表的大小（后面详解），而autoexpires为自动过期的秒数（过期后IP地址自动解封）。

```
modparam("htable", "htable", "ipban=>size=8;autoexpire=300;")
```

见下面代码，对于每个请求的IP地址（“\$si”代表来源IP地址），我们都检查ipban表中是否有值，如果有，则直接丢弃。

```
if($sht(ipban=>$si) !=$null) {  
    drop;  
}
```

如果我们判断从某个IP地址发来的请求过于频繁（通过pike模块可以做到，略），则将该IP地址写入ipban表，具体如下。

```
$sht(ipban=>$si) = 1;
```

哈希表可以使用dbtable参数在启动时从数据库表中加载数据，也可以使用dmqreplicate参数将数据同步到另一个配对的Kamailio实例中。示例如下。

```
modparam("htable", "htable", "a=>size=4;autoexpire=7200;dbtable=htable_a;")  
  
modparam("htable", "htable", "b=>size=5;")  
modparam("htable", "htable", "c=>size=4;autoexpire=7200;initval=1;dmqreplicate=1;")
```

这里所说的哈希表就是我们在数据结构中学到的哈希表。这里的size就是表的大小，而不是表中可以存储多少个元素。实际的槽位会有 2^{size} 个。如图3-1所示，当有两个相同的元素算出同一个哈希值时（碰撞），会依次存到一个链表中，而不会发生存不下的情况（只要内存足够大）。也就是说，如果只有一个槽，那么也能存下很多元素，不过，这时哈希表就成了一个链表（当然，Kamailio的size最小取值是2，也就是说最少会有 $2^2=4$ 个槽位）。

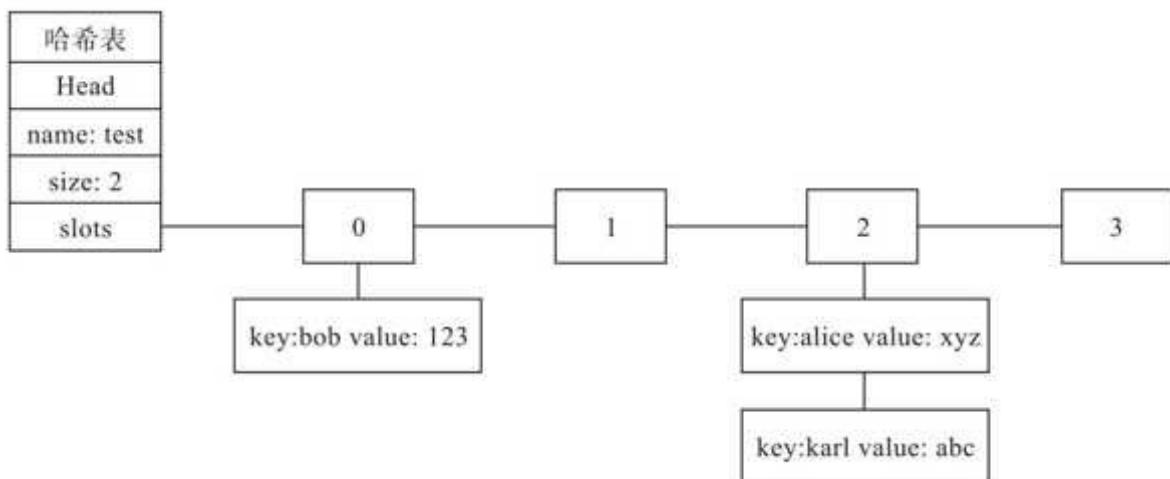


图3-1 有4个槽位的哈希表

详细的使用方法我们在后文还会讲（如9.2.5节、10.2节等）。

3.2.3 AVP

AVP（Attribute-Value Pairs），即属性-值对（也就是键值对）。键和值可以是数字或字符串。

AVP与事务关联，即若在收到INVITE消息时设置了一个AVP，那么在收到200 OK时也能使用这个AVP，但在收到BYE消息时不能使用，因为BYE属于另一个事务。AVP可以在route、branch_route以及failure_route路由块中使用。

AVP的使用格式如下。

```
$avp(avp_flags:avp_name)
```

AVP的使用格式还可以如下。

```
$avp(avp_alias)
```

下面对AVP使用时涉及的各个参数进行简单解读。

1.avp_flags

avp_flags的使用格式如下。

```
type_flags [script_flags]
```

在上述代码中，相关参数说明如下。

(1) type_flags是必选项，其可能的取值如下。

```
I | i | S | s
```

其中：

□ I或i表示整数。

□ S或s表示字符串。

(2) script_flags的取值为0~255，它是一个8位的无符号整数，即最大可以设置8个不同的标志位，如果省略，它会匹配所有标志。

2.avp_name

avp_name表示字符串或整数。如果是字符串，则不能包含空格且仅支持字符a~z、A~Z、0~9和_；如果是整数，则取值范围为0~ 2^{16} -1。

3.avp_alias

avp_alias表示标志符字符串，只支持字符a~z、A~Z、0~9和_。下面看几个例子。

□ \$avp(i:11): 这个AVP的ID是一个整数11。

□ \$avp(s:foo): 这个AVP的ID是一个字符串“foo”。

□ \$avp(bar): 这个AVP的ID是一个标志符字符串“bar”。

□ \$avp(i3:123): 这个AVP的ID是一个整数123，但有两个script_flag，即00000011=3，其中等号前的是二进制数，等号后的是十进制数。

这些AVP的值可以是字符串或整数，如“\$avp(i:1)=1”、“\$avp(i:2)='blah'"等。AVP可以在路由块代码中动态设置，也可以从数据库中读取。在数据库中读取时，AVP可以关联到用户上，也可以关联到Domain上。详情参见<https://kamailio.org/docs/modules-devel/modules/avpops.html>。

3.2.4 模块

Kamailio的模块可以根据需要有选择地进行加载，这可扩展核心的功能。不同的模块有不同的功能，模块在加载时向核心注册它的能力，并导出一些函数供配置文件及路由脚本调用。模块在初始化时，读取配置文件中通过modparam设置的参数。有的参数是纯静态的，有的参数可以在运行时通过RPC命令修改。

Kamailio最新的文档列出了近250个模块，在后面讲解示例脚本时会对一些常用的一些模块进行简单注释。在此，我们并不准备对所有模块进行一一介绍，而是在后文中用到它们时再讲解。



Kamailio的文档比较完善，每个模块都有单独的文档，模块的文档一般分为以下几个部分。

- 简介。
- 相关依赖（是否依赖其他模块）。
- 详细解释。
- 参数（在kamailio.cfg的模块参数部分可以设置的参数）。
- 函数（在路由块中可以执行的函数）。
- 对外导出的伪变量。
- RPC命令（在运行时可以改变参数或重载、查看模块内部数据的相关命令）。
- 事件路由（当相应事件发生时可以执行的回调路由块或回调函数）。

上面是模块文档的结构，这也在一定程度上反映了模块的结构。单纯列举模块文档未免有些枯燥，而且，即使最简单的路由转发功能也通常是由多个模块配合完成的。因此，本书将在后面的章节中结合实际的例子深入讲解各模块的功能，在此就不多介绍了。

Chapter 4

第4章

KEMI详解



Kamailio（继承自SER）默认使用类似于C语言的DSL（我们称之为原生脚本语言）进行配置和写路由逻辑，这在SER代码的初始版本发布时是最好的选择。

虽然Kamailio提供了大量的函数和功能，DSL在大部分时候工作得也很好，但在复杂的场景中，尤其是在跟外部路由逻辑交互的场景中，会受到一些限制。另外，DSL也很难在运行时重载，这主要是由当时的设计架构决定的，因此每次修改路由逻辑后都不得不重启Kamailio。所以，需要有一种



更好的方法写路由逻辑。随着各种嵌入式语言技术的不断成熟，便出现了KEMI。

有了KEMI，Kamailio的核心配置文件就可以只负责全局参数、加载模块、配置模块参数这三个部分了。这些都是在Kamailio启动时一次性执行并加载到内存的，但是有很多参数也可以在运行时通过相应的RPC进行控制。而实际的路由逻辑可以由KEMI路由脚本描述。KEMI现在支持的脚本语言有Lua、JavaScript、Python 2、Python 3、Ruby和Squirrel。

Kamailio以前支持的以内部执行方式（Inline Execution）执行的其他语言现在仍可以被支持，如.NET（C#等）、Perl、Java等，但目前并没有提供KEMI接口（未来也许会提供）。

相比原生的路由脚本，使用KEMI的好处主要有以下几个。

- 重载路由脚本时无须重启Kamailio。
- 学习现成的脚本语言比学习Kamailio DSL语言要容易。
- 现有的脚本语言更完善、更强大、更灵活。
- 针对各种脚本语言都有很多现成的库，这些库提供各种各样的功能。
- 提供各种现成的工具，利用这些工具可方便地进行调试、排错及写单元测试等。

在本书中，我们仅以Lua语言为例进行讲解。Lua语言虽非常简洁但功能强大，在FreeSWITCH中也可以使用Lua。如果你熟悉任何一种编程语言，几乎都可以在30分钟内学会基本的Lua语法。没有学过Lua的同学也可以通过附录C快速入门Lua。

4.1 KEMI Lua入口

Lua入口在app_lua模块中实现，Kamailio支持Lua 5.1和Lua 5.2。可以在kamailio.cfg中使用如下方法加载Lua路由脚本。

```
loadmodule "app_lua.so"
modparam("app_lua", "load", "/path/to/script.lua")
cfgengine "lua"
```

当收到SIP请求消息时，Kamailio将执行ksr_request_route()函数，它相当于原生脚本中的request_route{}路由块。如果该函数不存在，Kamailio在运行时会报错。

当收到SIP响应消息时，Kamailio将尝试执行ksr_reply_route()函数，即使该函数不存在也不会报错，它相当于原生脚本中的reply_route{}路由块。

当Kamailio每次向外发起一个请求时，将执行ksr_onsend_route()函数，即使该函数不存在也不报错，它相当于原生脚本中的onsend_route{}路由块。该函数也可以用于外发响应消息时的回调，但需要专门配置。

可以通过KSR.tm.t_on_branch(...)函数设置一个分支路由回调函数，具体方法如下。

```
if KSR.tm.t_is_set("branch_route") < 0 then
    KSR.tm.t_on_branch("ksr_branch_manage");
end
```

同样，可以使用KSR.tm.t_on_reply(...)、KSR.tm.t_on_failure()、KSR.tm.t_on_branch_failure(...)设置在回复路由、失败路由和分支失败路上进行回调。

与事件回调路由相关的函数由各模块的event_callback参数指定，如在sipdump模块中通过event_callback参数设置一个回调函数，在收发SIP消息时该函数会被回调，配置代码如下。

```
modparam("sipdump", "event_callback", "ksr_sipdump_event")
```

提示

旧的sr Lua模块仍然可用，但由于是过时的调用方式，在本书中我们就不介绍了。

原生脚本函数与KEMI脚本函数的对应关系如表4-1所示。

表4-1 原生脚本函数与KEMI脚本函数的对应关系

原生脚本	KEMI 脚本 (如 Lua)
request_route	ksr_request_route()
reply_route	ksr_reply_route()
onsend_route	ksr_onsend_route()
route[NAME]	任何函数名
event_route[NAME]	通过模块中的 event_callback 参数指定的函数
branch_route[NAME]	通过 KSR.tm.t_on_branch("f") 指定的函数
onreply_route[NAME]	通过 KSR.tm.t_on_reply("E") 指定的函数
failure_route[NAME]	通过 KSR.tm.t_on_failure("f") 指定的函数
event_route{tm:branch-failure:NAME}	通过 KSR.tm.t_on_branch_failure("f") 指定的函数

4.2 KEMI函数

KEMI提供的函数都在KSR对象内。KSR中的函数通常返回整型或布尔型的值（这是历史原因导致的，Lua还是调用了核心或模块中在C语言中实现的函数），只有极少数例外。每个函数最多有6个参数。

4.2.1 函数整型返回值规则

函数的整型返回值通常遵循以下规则。

- 大于0：函数成功执行，对返回值的逻辑判断应该返回true。
- 小于0：函数执行失败，对返回值的逻辑判断应该返回false。
- 等于0：脚本执行应该终止，注意要用KSR.x.exit()终止而不是exit。

布尔型的返回值是true或false，可以直接用于逻辑判断。有的函数是void类型的，不返回任何值。有的函数返回xval类型的值，如KSR.pv模块中获取伪变量值的函数。如果函数返回xval，则结果可以是字符串、整数或nil（对应C语言中的NULL）。

函数的参数一般只能是整数或字符串，整数不使用引号，而字符串需要使用单引号或双引号。

KEMI导出的很多函数都与原生脚本有相应的对应关系，通用的规则如下。

- 如果函数的参数是整数，则KEMI函数以整数方式输入（在原生脚本中所有输入都是字符串形式）。
- 如果原生函数有可选的参数，则对应的KEMI函数会导出多个类似的函数，每个都有不同个数的参数。这主要是因为并不是所有语言都支持可变参数。如原生脚本中的forward(...)函数就对应KSR.forward()和KSR.forward_uri()两个函数。

在运行时，可以使用kamcmd app_lua.api_list命令打印可用的KEMI函数。

4.2.2 函数返回0的情况

在原生脚本中，如果函数返回0，则Kamailio会自动中断脚本的执行，但在KEMI中不是这样的。所以，如果KEMI中的函数返回0，要想中断脚本执行，后面还应该调用KSR.x.exit()或KSR.x.drop()

KEMI中只有少数函数会返回0，为方便使用，列举如下。

- tm模块中，主要包括t_check_trans()和t_newtran()。
- websocket模块中，主要包括ws_handshake()。

代码示例如下。

```
function ksr_request_route()
    if KSR.tm.t_check_trans() == 0 then
        KSR.x.exit();
    end
end
```

上面这些函数也会返回大于或小于0的值，上面的示例仅处理了返回值为0的情况，在实际应用中应该检查所有情况并做相应处理。对于类似的情况本书后面不再赘述。

4.2.3 模块函数

不同模块中的函数都会映射到“KSR.模块名”包中，如acc模块中的函数对应KSR.acc，dispatcher模块中的函数对应KSR.dispatcher等。实际的函数名和参数可以通过对应指令进行查询，比如对于Lua语言可通过kamcmd app_lua.api_list命令查询。KEMI函数通常与原生脚本语言中的同名函数有一定的对应关系，在4.6.1节我们还会对此进行介绍。



在各模块相关的文档中 [\[1\]](#)，也有对导出函数的说明和详细解释。具体的函数和参数比较多，就不在这里展开了，在后文中，我们都会以Lua语言为例来讲述各种路由场景和用到的模块示例。

4.3 在C函数中导出KEMI函数

对于KEMI函数，由于大部分是在模块中实现的，而且需要加载后才能导出，为了避免所有模块都依赖于语言相关的模块，因此使用了函数映射方式实现。也就是说，在核心实现一些函数，并使用一个映射表将这些函数映射到模块中。映射关系是在Kamailio启动时实现的，因为是一对一的数组映射，所以查找起来非常快。目前使用的映射表大小是1024。这个值以后可以改变，但目前所有模块导出的函数加起来也不到1000个，而且在实际使用时也不可能同时加载所有模块，因此目前这个值足够了。导出函数比较少的另一个原因是，有一些函数或功能在相关的语言（如Lua）中有原生的实现，Kamailio中的函数根本没必要再导出。

可以使用如下方法将模块中的函数导出到KEMI中。

- 声明一个sr_kemi_t数组。
- 使用mod_register()函数将模块中的函数注册到KEMI，或者在核心启动时使用sr_kemi_modules_add()将模块中的函数注册到KEMI。

sr_kemi_t在核心的kemi.h中定义。

```
#define SR_KEMI_PARAMS_MAX 6

typedef struct sr_kemi {
    str mname; /* sub-module name */
    str fname; /* function name */
    int rtype; /* return type (supported SR_KEMIP_INT, SR_KEMIP_BOOL, SR_KEMIP_XVAL) */
    void *func; /* pointer to the C function to be executed */
    int ptypes[SR_KEMI_PARAMS_MAX]; /* array with the type of parameters */
} sr_kemi_t;
```

下面是sl模块导出两个函数的例子（C语言）。

```
C function sl_send_reply_str(...) is exported as sl.sreply(...)
C function send_reply(...) is exported as sl.freply(...)
static sr_kemi_t sl_kemi_exports[] = {
    { str_init("sl"), str_init("sreply"),
        SR_KEMIP_INT, sl_send_reply_str,
        { SR_KEMIP_INT, SR_KEMIP_STR, SR_KEMIP_NONE,
          SR_KEMIP_NONE, SR_KEMIP_NONE, SR_KEMIP_NONE } },
    { str_init("sl"), str_init("freply"),
        SR_KEMIP_INT, send_reply,
        { SR_KEMIP_INT, SR_KEMIP_STR, SR_KEMIP_NONE,
          SR_KEMIP_NONE, SR_KEMIP_NONE, SR_KEMIP_NONE } },
    { { 0, 0 }, { 0, 0 }, 0, NULL, { 0, 0, 0, 0, 0, 0 } }
};

int mod_register(char *path, int *dlflags, void *p1, void *p2)
{
    sr_kemi_modules_add(sl_kemi_exports);
    return 0;
}
```

注意，值为全0或NULL的元素（如上述代码中sl_kemi_exports的第三个元素）将结束这个导出数组。Lua中导出函数与Kamailio内部结构对应关系如表4-2所示。

表4-2 Lua中的导出函数与Kamailio内部结构的对应关系

导出到 KEMI 的函数	Kamailio 内部结构
sr_kemi_lua_exec_func_0	sr_kemi_lua_export_t(t_relay)
...	...
sr_kemi_lua_exec_func_100	sr_kemi_lua_export_t(sl_send_reply)
...	...
NULL	NULL
...	...

导出函数的第一个参数为sip_msg_t*类型（它表示当前处理的SIP消息），后面跟6个整数或str*类型（注意不是char*，Kamailio有自己的字符串类型）的参数。当遇到SR_KEMIP_NONE参数时会提前终止，但有的编译器会报错，因此推荐提供所有参数。

导出的函数将存放在_sr_kemi_core数组中，并在kemi.c中可以找到。

sip_msg_t*后的6个参数并不能在所有情况下都自由组合，相关限制如下。

□ 1个参数，可以是int或str*。

□ 2~5个参数，可以是int或str*的任意组合。

□ 6个参数，必须全是str*。如需要其他组合也可以添加，但目前尚无此需求。

返回值有以下几种。

□ SR_KEMIP_INT：返回值是整数且必须符合如下规则。

○ 小于0：错误，在逻辑比较中按false处理。

○ 大于0：成功，在逻辑比较中按true处理。

○ 等于0：脚本逻辑应立即终止执行。

○ 例外：有一些Getter类的取值函数不在此列，如KSR.kx.get_status()、KSR.kx.get_timestamp()等。

□ SR_KEMIP_BOOL：返回值在原生脚本语言中可以是true或false，但在C语言中必须为1或0。

□ SR_KEMIP_XVAL：返回值视当前情况而异，可以是整数或字符串。如KSR.pv.get(...)这样的函数会返回伪变量类型的值，返回这种值的函数应该最多只有两个输入参数。

导出函数的C语言示例代码如下（在此主要给大家一个直观的印象，不多解释，即使写Lua路由脚本也不需要了解C语言）。

```
static int sr_kemi_lua_exec_func_1023(lua_State *L) {
    return sr_kemi_lua_exec_func(L, 1023);
}
typedef struct sr_kemi_lua_export {
    lua_CFunction pfunc;
```

```
    sr_kemi_t *ket;
} sr_kemi_lua_export_t;

static sr_kemi_lua_export_t _sr_kemi_lua_export_list[] = {
    { sr_kemi_lua_exec_func_0, NULL },
    ...
    { sr_kemi_lua_exec_func_1023, NULL },
    {NULL, NULL}
};

sr_kemi_t *sr_kemi_lua_export_get(int idx) {
    if(idx<0 || idx>=SR_KEMI LUA_EXPORT_SIZE) return NULL;
    return _sr_kemi_lua_export_list[idx].ket;
}

int sr_kemi_lua_exec_func(lua_State* L, int eidx)
{
    sr_kemi_t *ket;
    ket = sr_kemi_lua_export_get(eidx);
    return sr_kemi_lua_exec_func_ex(L, ket, 0);
}
```

4.4 KEMI和伪变量

KSR.pv模块导出的函数可以处理Kamailio相关的伪变量。这些函数应该由Kamailio核心或相关的语言模块导出，而不需要依赖其他模块。当然，具体有多少变量还跟加载了哪些模块有关，如\$T(..)依赖于tmx模块。

伪变量是针对Kamailio原生的脚本语言设计的，因此在KEMI脚本中使用起来有些限制，下面就对此进行介绍。

4.4.1 伪变量静态名称限制

注意，这部分内容非常重要。

在很多场景下，伪变量的名字需要是一个静态字符串，这在解析kamailio.cfg时会进行验证。然而，在引入KEMI以后，Kamailio无法控制脚本语言（如Lua）中的解析，导致可能产生很多（或无限多个）伪变量，进而会塞满Kamailio的私有内存空间。

如在使用htable的场景中，如果使用了原生的kamailio.cfg，且使用了\$sht(test=>x)（一个静态的ID）以及\$sht(test=>\$rU)（根据RURI自动生成的ID）这样的伪变量，则伪变量只会在Kamailio启动时生成，且只会有两个。

在使用KEMI时，使用KSR.pv.get("\$sht(test=>x)")和KSR.pv.get("\$sht(test=>\$rU)")取值仍然是可以的，因为系统会生成两个伪变量标志符。但是，如果使用KSR.pv.get("\$sht(test=>..KSR.kx.get_ruser(..))")获取\$sht(test=>\$rU)的值，则不同的R-URI会生成不同的字符串，如\$sht(test=>alice)、\$sht(test=>bob)、\$sht(test=>carol)等，随着请求越来越多将可能生成无限多个这种字符串，直至耗尽内存。

Kamailio提供了一些机制用于处理上述问题，特别是在htable中，会使用专门的函数访问htable的项目，或清除某些伪变量。但这些处理机制并不是所有模块中都有，如sqlops、ndb_redis、ndb_mongodb中就没有。所以，应尽量避免使用动态字符串作为伪变量的名字。

下面的代码通过\$var(x)提供了一种避免使用动态字符串作为伪变量名的方法。

```
KSR.pvx.var_sets("x", "alice");
shtx = KSR.pv_get("$sht(test=>$var(x))");
KSR.pvx.var_sets("x", "bob");
shtx = KSR.pv_get("$sht(test=>$var(x))");
KSR.pvx.var_sets("x", "carol");
shtx = KSR.pv_get("$sht(test=>$var(x))");
```

提示

有些模块返回SQL或NoSQL结果集，结果集的ID是可以被重复使用的，这些结果集返回的内容存放在Kamailio的私有内存中，可以被不同的Kamailio工作进程安全地访问。

4.4.2 针对特定伪变量的函数

KSR.kx包针对一些特定的伪变量导出了一些get或set函数（分别用于获取或设置变量的值），特别是用于那些存取SIP消息头域（Request-URI、From-URI、From-URI-username）的函数。如果有可能，应该优先使用这些函数，如KSR.kx.get_ruri()、KSR.kx.get_furi()。

有些函数有相应的gete和getw变体，这些变体与pv模块里的函数的含义相同，其中返回值的区别如下。

- KSR.kx.get_ua(): 不存在则返回nil。
- KSR.kx.gete_ua(): 不存在则返回空字符串。

□ KSR.kx.getw_ua(): 如果结果为nil则返回字符串“<null>”，以方便打印日志。

KSR.pvx导出了一些用于存取私有内存值（如\$var(...)）或共享内存值（如\$shv(...)）的函数以及XAVP的变体。例如，下面两种方法是等效的，推荐使用后者。因为前者相当于调用原生脚本中的\$var()函数，而后者直接使用KEMI导出的函数。

```
KSR.pv.sets("$var(x)", "alice");
KSR.pvx.var_sets("x", "alice");
```

KSR.sqllops包提供了一些函数以替代\$dbr(...), 如下面这两种方法是等效的，推荐使用后者。

```
KSR.sqllops.sql_query("ca", "select username from subscriber limit 1", "ra");
if KSR.pv.get("$dbr(ra=>rows)") > 0 then
    local username = KSR.pv.get("$dbr(ra=>[0,0])");
end

KSR.sqllops.sql_query("ca", "select username from subscriber limit 1", "ra");
if KSR.sqllops.sql_num_rows("ra") > 0 then
    local username = KSR.sqllops.sql_result_get("ra", 0, 0);
end
```

在实际使用时，推荐查看脚本中用到的每个模块是否有相关的KEMI函数，如果有，则使用它们提供的方法应该比使用伪变量更方便。

4.5 核心和pv模块中的函数

Kamailio的核心直接提供了一些函数，主要用于实现核心功能以及写日志（有的由xlog模块提供），如KSR.function(params)这样的函数。

KSR.pv子模块提供了通用的伪变量存取函数，而KSR.hdr子模块则提供了存取SIP头域的函数，相关的模块有htable、kemix、textops、textopsx等。举例如下。

```
KSR.debug("a debug message from Lua script\n");
KSR.hdr.remove("Route");
```

4.5.1 核心中的常用函数



常用函数列表及简介如下。

- KSR.add_local_rport(): 在Via头域中增加rport参数，参见第3章同名函数。
- KSR.log(str level, str msg): 其中level的取值有dbg、info、notice、warn、crit、err。
- KSR.debug(...): 打印debug级别的日志，相当于KSR.log("dbg", ...), 下同。
- KSR.info(...): 打印info级别的日志。
- KSR.notice(...): 打印notice级别的日志。
- KSR.warn(...): 打印warn级别的日志。
- KSR.crit(...): 打印crit (critical) 级别的日志。
- KSR.err(...): 打印err (error) 级别的日志。
- KSR.force_rport(): 强制使用rport，参见3.1.7节。
- KSR.is_method(...): 测试请求方法，返回布尔值，如KSR.is_method("INVITE")用于测试是否为INVITE请求。
 - A: ACK。
 - B: BYE。
 - C: CANCEL。
 - I: INVITE。
 - K: KDMQ。
 - M: MESSAGE。
 - N: NOTIFY。
 - O: OPTIONS。
 - E: PRACK。

◦ P: PUBLISH。

◦ F: REFER。

◦ R: REGISTER。

◦ S: SUBSCRIBE。

◦ U: UPDATE。

◦ G: GET。

◦ T: POST。

◦ V: PUT。

◦ D: DELETE。

□ KSR.is_INVITE(): 测试请求方法是否为INVITE。此外，KSR.is_ACK()、KSR.is_BYE()、KSR.is_CANCEL()、KSR.is_REGISTER()、KSR.is_MESSAGE()、KSR.is_SUBSCRIBE()、KSR.is_PUBLISH()、KSR.is_NOTIFY()、KSR.is_OPTIONS()、KSR.is_REFER()、KSR.is_INFO()、KSR.is_UPDATE()、KSR.is_PRACK()这几个函数与之都类似，不再赘述。

□ KSR.is_KDMQ(): 用于设置Kamailio在不同实例间同步数据的方法。

□ KSR.is_GET(): HTTP GET方法。

□ KSR.is_POST(): HTTP POST方法。

□ KSR.is_PUT(): HTTP PUT方法。

□ KSR.is_DELETE(): HTTP DELETE方法。

□ KSR.is_myself(...): 用于设置myself的规则，见前面的章节。

□ KSR.is_myself_ruri(): 测试R-URI是否在myself范围内。

□ KSR.is_myself_srcip(): 测试来源IP地址是否在myself范围内。

□ KSR.is_TCP(): 测试是否是TCP。

□ KSR.is_UDP(): 测试是否是UDP。

□ KSR.is_SCTP(): 测试是否是SCTP。

□ KSR.is_WSX(): 测试是否是WS或WSS协议。

□ KSR.is_proto(...): 用于判断当前消息使用的是什么协议，只要配置了参数字符串中的一种即返回true。如KSR.is_proto("EW")返回true则表示是TLS或WSS协议，参数字符串是以下字符的组合。

◦ e或E: 代表是TLS协议。

◦ s或S: 代表是SCTP。

◦ t或T: 代表是TCP。

◦ u或U: 代表是UDP。

◦ v或V: 代表是WS协议。

◦ w或W: 代表是WSS协议。

- KSR.is_IPv4(): 测试是否是IPv4。
- KSR.is_IPv6(): 测试是否是IPv6。
- KSR.to_IPv4(): 测试目标地址是否是IPv4。
- KSR.to_IPv6(): 测试目标地址是否是IPv6。
- KSR.to_TCP(): 测试是否使用TCP外发。
- KSR.to_TLS(): 测试是否使用TLS协议外发。
- KSR.to_SCTP(): 测试是否使用SCTP外发。
- KSR.to_UDP(): 测试是否使用UDP外发。
- KSR.to_WSS(): 测试是否使用WSS外发。
- KSR.to_WSX(): 测试是否使用WS或WSS协议外发。
- KSR.setflag(...): 设置flag。
- KSR.resetflag(...): 重置flag。
- KSR.isflagset(...): 测试flag是否已设置。
- KSR.setbflag(...): 设置branch flag。
- KSR.resetbflag(...): 重置branch flag。
- KSR.isbflagset(...): 测试branch flag是否已设置。
- KSR.seturi(...): 设置R-URI。
- KSR.setuser(...): 设置R-URI中的用户部分。
- KSR.force_rport(...): 强制rport参数。
- KSR.set_advertised_address(): 设置通告地址。
- KSR.set_advertised_port(): 设置通告端口。
- KSR.forward(): 无状态转发到\$du, 如果\$du不存在, 则转发到\$ru。
- KSR.forward_uri(str uri): 无状态转发到URI。
- KSR.route(...): 执行路由块。

4.5.2 pv模块相关函数

pv模块提供了与伪变量存取相关的函数。伪变量的名称必须是合法的名称。关于伪变量, 参见3.2.1节。

有些全局的变量类型存放在系统共享内存中, 这些必须在kamailio.cfg中用modparam()定义。如与\$shv(...)相关的变量类型必须进行以下定义。

```
modparam("pv", "shvset", "name=value")
```

pv模块提供的函数都在KSR.pvx包中。下面就来介绍这些函数, 其中第一行均为函数原型描述, 如xval KSR.pv.get(str "pvname")表示: 返回值为xval类型 (即可以是字符串、整数、或null值) ; 输入

参数是str（字符串）类型的；pvname是参数名占位符，在这个特定的函数中它表示伪变量的名字。

1.KSR.pv.get(...)

函数原型（包含参数类型和返回值类型，下同）：

```
xval KSR.pv.get(str "pvname")
```

返回伪变量的值，返回值可能是字符串、整数或null，如果伪变量不存在则返回值为null，如：

```
KSR.debug("ruri is: " + KSR.pv.get("$ru") + "\n");
```

2.KSR.pv.gete(...)

函数原型：

```
xval KSR.pv.gete(str "pvname")
```

其他介绍同KSR.pv.get(...), 但在返回值为null的情况下将返回空字符串（""），如：

```
KSR.debug("avp is: " + KSR.pv.gete("$avp(x)") + "\n");
```

3.KSR.pv.getvn(...)

函数原型：

```
xval KSR.pv.getvn(str "pvname", int vn)
```

返回整数，但在返回值为null的情况下将返回默认值vn，如：

```
KSR.debug("avp is: " + KSR.pv.getvn("$avp(x)", 0) + "\n");
```

4.KSR.pv.getvs(...)

函数原型：

```
xval KSR.pv.getvs(str "pvname", str "vs")
```

返回字符串，在返回值为null的情况下返回默认值vs，如：

```
KSR.debug("avp is: " + KSR.pv.getvs("$avp(x)", "foo") + "\n");
```

5.KSR.pv.getw(...)

函数原型：

```
xval KSR.pv.getw(str "pvname")
```

在返回值为null的场景下返回字符串“<null>”，以方便打印它的值，如：

```
KSR.debug("avp is: " + KSR.pv.getw("$avp(x)") + "\n");
```

6.KSR.pv.seti(...)

函数原型:

```
void KSR.pv.seti(str "pvname", int val)
```

将伪变量赋值为整数, 如:

```
KSR.pv.seti("$var(x)", 10);
```

7.KSR.pv.sets(...)

函数原型:

```
void KSR.pv.sets(str "pvname", str "val")
```

将伪变量赋值为字符串, 如:

```
KSR.pv.sets("$var(x)", "kamailio");
```

8.KSR.pv.unset(...)

函数原型:

```
void KSR.pv.unset(str "pvname")
```

将伪变量设为null, 如:

```
KSR.pv.unset("$avp(x)");
```

9.KSR.pv.is_null(...)

函数原型:

```
bool KSR.pv.is_null(str "pvname")
```

如果伪变量为null, 则返回true, 如:

```
if(KSR.pv.is_null("$avp(x)) {  
    ***  
}
```

4.5.3 KSR.hdr子模块

KSR.hdr子模块提供了SIP消息头域处理的相关函数。

1.KSR.hdr.append(...)

函数原型:

```
int KSR.hdr.append(str "hdrval")
```

若追加一个头域, 则将会追加到最后一个头域后面, 注意后面应该有"\r\n", 如:

```
KSR.hdr.append("X-My-Hdr: " + KSR.pv.getw("$si") + "\r\n");
```

2.KSR.hdr.append_after(...)

函数原型:

```
int KSR.hdr.append_after(str "hdrval", str "hdrname")
```

在某个头域（第一个匹配的头域）后面增加一个头域，如:

```
KSR.hdr.append_after("X-My-Hdr: " + KSR.pv.getw("$si") + "\r\n", "Call-ID");
```

3.KSR.hdr.insert(...)

函数原型:

```
int KSR.hdr.insert(str "hdrval")
```

在第一个头域前面插入一个头域，如:

```
KSR.hdr.insert("X-My-Hdr: " + KSR.pv.getw("$si") + "\r\n");
```

4.KSR.hdr.insert_before(...)

函数原型:

```
int KSR.hdr.insert_before(str "hdrval", str "hdrname")
```

在某个头域（第一个匹配的头域）后面增加一个头域，如:

```
KSR.hdr.insert_before("X-My-Hdr: " + KSR.pv.getw("$si") + "\r\n", "Call-Id");
```

5.KSR.hdr.remove(...)

函数原型:

```
int KSR.hdr.remove(str "hdrval")
```

删除所有匹配函数中参数指定名字的头域，如:

```
KSR.hdr.remove("X-My-Hdr");
```

6.KSR.hdr.rmapappend(...)

函数原型:

```
int KSR.hdr.rmapappend(str "hrc", str "hadd")
```

删除所有匹配的头域并增加一个头域，结尾必须有“\r\n”，如:

```
KSR.hdr.rmapappend("X-My-Hdr", "X-My-Hdr: abc\r\n");
```

其实它相当于:

```
KSR.hdr.remove("hnm")
KSR.hdr.append("hadd")
```

7.KSR.hdr.is_present(...)

函数原型:

```
int KSR.hdr.is_present(str "hdrval")
```

如果头域存在则返回大于0的值, 如:

```
if(KSR.hdr.is_present("X-My-Hdr") > 0) {
    ***
}
```

8.KSR.hdr.append_to_reply(...)

函数原型:

```
int KSR.hdr.append_to_reply(str "hdrval")
```

在与该请求消息对应的回复消息上增加一个头域, 如:

```
KSR.hdr.append_to_reply("X-My-Hdr: " + KSR.pv.getw("Ssi") + "\r\n");
```

9.KSR.hdr.get(...)

函数原型:

```
xval KSR.hdr.get(str "hname")
```

返回头域的值, 如果不存在则返回null, 如:

```
v = KSR.hdr.get("X-My-Hdr");
```

10.KSR.hdr.get_idx(...)

函数原型:

```
xval KSR.hdr.get(str "hname", int idx)
```

如果头域有多个, 则返回第idx个。idx从0开始计数。如果不存在, 则返回null; 如果idx为负数, 则返回倒数第idx个, 如:

```
v = KSR.hdr.get_idx("X-My-Hdr", 1);
```

11.KSR.hdr.gete(...)

函数原型:

```
xval KSR.hdr.gete(str "hname")
```

同KSR.hdr.get(), 但如果头域不存在则返回空字符串而不是null。适用于字符串拼接的场景 (null值通常会拼接出错), 如:

```
v = KSR.hdr.gete("X-My-Hdr");
```

12.KSR.hdr.gete(...)

同KSR.hdr.get_idx(), 但如果头域不存在则返回空字符串而不是null, 如:

```
v = KSR.hdr.gete_idx("X-My-Hdr", -1);
```

13.KSR.hdr.getw(...)

函数原型:

```
xval KSR.hdr.getw(str "hname")
```

同KSR.hdr.get(), 但如果头域不存在则返回字符串“<null>”而不是null, 如:

```
v = KSR.hdr.getw("X-My-Hdr");
```

14.KSR.hdr.getw_idx(...)

函数原型:

```
xval KSR.hdr.getw_idx(str "hname", int idx)
```

同KSR.hdr.get_idx(), 但如果头域不存在则返回字符串“<null>”而不是null, 如:

```
v = KSR.hdr.getw_idx("X-My-Hdr", 2);
```

15.KSR.hdr.match_content(...)

函数原型:

```
bool KSR.hdr.match_content(str "hname", str "op", str "mval", str "eidx")
```

如果头域的值与表达式匹配, 则返回true, 否则返回false。相关参数如下。

□ **hname:** 头域名称。

□ **op:** 运算符, 有以下选项。

○ **eq:** Equal, 等于。

○ **ne:** Not Equal, 不等于。

○ **sw:** Starts With, 以字符串开头。

○ **in:** Include, 包含。

□ **mval:** 匹配的值。

□ **eidx:** 作用同名头域的第几条, 可能的取值如下。

○ **f:** First, 第一个。

○ **l:** Last, 最后一个。

◦ a: All, 所有头域。

◦ o: 至少有一个匹配。

示例:

```
if (KSR.hdr.match_content("X-My-Hdr", "in", "test", "o")) {  
    ...  
}
```

4.5.4 特殊的KEMI函数

KSR.x子模块针对不同语言提供了一些特殊的函数。

1.KSR.x.modf

函数原型:

```
int KSR.x.modf(str "fname", params...)
```

执行Kamailio模块导出函数fname，后面的参数必须是字符串，它们将作为函数的参数传入，如:

```
KSR.x.modf("sl_send_reply", "200", "OK");
```

注意

尽量使用专门的KSR函数，而不是这个函数。如果你必须使用该函数，则应检查一下C语言的源代码中是否有fixup和fixup-free函数。如果你不知道如何检查，可以到sr-users邮件列表发邮件询问。

2.KSR.x.exit(...)

函数原型:

```
void KSR.x.exit()
```

KSR.x.exit(...)相当于原生脚本中的exit()，用于停止当前SIP消息路由脚本的执行，可参见4.6.3节。

注意

对于Lua语言而言，直接调用exit()会停止整个Kamailio，所以建议在任何时候都不要用。

3.KSR.x.drop(...)

函数原型:

```
void KSR.x.drop()
```

KSR.x.drop(...)用于停止当前脚本的执行并将SIP消息丢弃。有的语言中没有这个函数，在停止脚本继续执行前可以使用KSR.set_drop()，如在Python中:

```
KSR.set_drop()  
exit()
```

4.6 原生脚本与KEMI对比

本书主要以KEMI为例进行讲解，但网上大部分的资料和例子还都是基于原生脚本的，理解两者的异同会有助于学习，尤其是那些已经熟悉旧版本Kamailio的读者。

4.6.1 函数名

原生语言相关的函数命名规则如下。

- 类似于C语言的语法和命名规范。
- 只有函数名，命名空间靠函数名前缀实现，如my_function(params)、t_relay()。
- 很多模块都使用相同的前缀，如对于tm模块有t_relay、t_reply、t_replicate。
- 函数名通过脚本解释器导出到核心。
- 很多的函数都是在模块中导出的。

KEMI脚本语言相关的函数命名规则如下。

- 面向对象的命名规则。
- 模块变成一个对象，即“KSR.模块名.函数名（参数名）”，如：

```
KSR.tm.t_relay()
```

- 很多函数名与原生语言的函数名相同，即使看起来有些冗余，如：

```
KSR.acc.acc_request(...)
```

- 有些函数是从核心导出的，但大部分是从模块导出的。
- 特殊的KSR子模块有KSR.hdr、KSR.pv、KSR.x等。

4.6.2 函数的参数

原生语言的函数参数说明如下。

- 函数返回整型值。
- 对于路由块中的返回值，解释器也会处理。
 - 返回值小于0，对应false逻辑。
 - 返回值大于0，对应true逻辑。
 - 返回值等于0，中止当前消息的脚本执行。

脚本示例：

```
# handle retransmissions
if (!is_method("ACK")) {
```

```

    if(t_precheck_trans()) {
        t_check_trans();
        exit;
    }
    t_check_trans();
}

```

对KEMI脚本语言中相关函数所涉参数说明如下。

- 返回值可以是布尔型、整型或字符串型的值。
- 从pv模块中取值，可以返回字符串。
- 大部分函数返回整型值。
- 部分函数返回布尔型的值，绝大部分核心函数返回布尔型的值。
- KSR.tm.t_check_trans()、KSR.tm.t_newtran()、KSR.websocket.handle_handshake()这几个函数返回0，需要特殊处理。

脚本示例：

```

-- 处理消息重传
if not KSR.is_ACK() then
    if KSR.tmx.t_precheck_trans()>0 then
        KSR.tm.t_check_trans();
        return 1;
    end
    if KSR.tm.t_check_trans()==0 then
        return 1;
    end
end

```

4.6.3 停止当前脚本执行

在原生语言中要停止当前脚本执行可以使用exit()、drop()、return(0)等方法实现。

对于KEMI脚本的停止操作，说明如下。

- 注意：在某些语言（如Lua）中可直接调用exit()停止整个Kamailio。
- 在Python中可以使用exit()或os.exit()。原生的drop()对应KSR.set_drop()+os.exit()。
- 在其他KEMI脚本（Lua、JavaScript、Squirrel）中，可用KSR.x.exit()、KSR.x.drop()、return等方法停止。
- 不要忘记返回值为0的函数，需要特殊处理。

4.7 其他

上面我们提到了KEMI的诸多好处，但在实际应用中，性能也是一个重要指标。那么，KEMI脚本与原生脚本在性能方面有什么区别呢？实际测试显示，原生脚本跟KEMI中支持的其他语言运行效率差不多，甚至与用Lua和Python写的脚本没有本质的区别（在通用环境中，Python脚本性能会弱一些，这主要是因为Python比Lua更复杂），详见9.1.2节。

通过对本章的学习，我们了解了KEMI的使用方法以及核心中的大部分功能函数，并与原生脚本中的用法做了对比。除此之外，还有一些值得注意的地方，简单总结如下。

与原生脚本比起来，使用KEMI有很多优势，主要表现在如下方面。

□ KEMI支持的语言都是真正的脚本语言，所以其拥有更多的语句、表达式、函数、库等。

□ 更少的维护量（对于Kamailio开发者来讲），因为语言解释器是由别人（专人）维护的。

□ 语言本身有更多的文档、更多的学习资源，甚至很多人都已经熟悉这些语言了。

□ 支持运行时重载。

○ 修改路由逻辑无须重启Kamailio。

○ 重载后将在路由到下一次请求时生效。

□ 性能与原生脚本没有明显区别。

○ Lua、JavaScript、Squirrel都直接使用静态函数映射（导出）。

○ 由于Python使用动态对象，所以其运行会略慢一些。

当然，使用KEMI也不是只有好处没有坏处。受KEMI的历史及架构所限，对KEMI脚本中函数的使用也受到一些限制，这会导致有时候在KEMI脚本中所写的代码反而比在原生脚本中所写的代码更长。

下面介绍KEMI脚本的一些弱点，以及一些使用建议（以Lua为例）。

在KEMI中必须通过函数访问伪变量而不能直接引用，也不能将伪变量直接用在字符串中，示例如下。

```
KSR.pv.get("$rU")
KSR.pv.sets("$rU", "test")
KSR.pv.seti("$var(x)", 10)
KSR.pv_is_null("$rU")
```

提示

可以将结果存到Lua语言的局部变量中，如src_ip=KSR.pv.get("\$si")，但这样的代码会显得有些啰嗦。



在KEMI中函数静态参数需要预编译，但是大多数情况下预编译是没必要的，因为含变量的参数在运行时每次都会求值（值会变），而预编译会带来一些开销。

综上，有些正则表达式替换类的操作以及其他与字符串、整数运算等相关的操作可以直接使用KEMI脚本语言提供的功能，而不需要调用Kamailio本身提供的函数。

最后说一点，目前并不是所有模块都支持KEMI，比如IMS相关的模块中就有很多不支持。因此对

注

于Kamailio项目，希望大家到GitHub上提交Pull Request，共同把这个开源项目做得更好。

好了，有了上述这些基本知识，从下一章开始，我们就可以实际运行Kamailio，跑一些真正的路由脚本了。

Chapter 5

第5章

Kamailio运行环境与实例

有了前面几章的基础知识，下面就可以实际运行Kamailio并研究一些实际案例了。

5.1 运行Kamailio

典型的Kamailio仅支持Linux操作系统，并且可直接在Linux操作系统上安装。在Linux上安装Kamailio的方法请参考附录A。我们在此仅讨论如何在Docker环境中运行Kamailio。Docker可以在Windows、MacOS和Linux等宿主机操作系统上运行，所以不管你使用何种操作系统，都可以很方便地运行Kamailio和本书的示例。

5.1.1 环境准备

使用Docker运行Kamailio以及本书相关的示例前，我们需要先准备一下环境。下面是对环境依赖的简要说明，对于这些工具不熟悉的读者，也可以在本书后面的附录中找到更多的解释和学习资源。

1.Docker



Docker是一种容器技术，现在，它已经成了进行Linux相关的软件开发和部署的事实上的标准。Docker是一个开源项目，它彻底释放了计算虚拟化的威力，极大地提高了应用的维护效率，降低了云计算应用开发的成本。通过使用Docker，不仅可以让应用的部署、测试和分发都变得前所未有的高效和轻松，而且在工作结束后，可以一键删除所有镜像和临时文件，而不会在操作系统上的各种目录中留下垃圾（Docker本身需要占用一些存储空间，如果你的硬盘空间足够大，那么就不用过多关心是否占用存储空间的问题了）。更重要的是，通过Docker可以把各种环境隔离开来，避免不同环境之间产生冲突。比如笔者在开发过程中，经常用到Kamailio以及FreeSWITCH的不同版本，它们又分别依赖于很多不同的第三方软件库，不同的库又有很多不同的版本，使用Docker容器就可以很好地避免产生冲突。

如果你还不熟悉Docker，那至少应该听说过虚拟机。虚拟机是在真正的物理机上虚拟出来的“电脑”，有虚拟的CPU、内存、硬盘、网卡等。而Docker技术在虚拟机的基础上更进一步，通过Linux内核和内核功能（例如Cgroup和Namespace）来分隔进程，以便各进程可以相对独立地运行，而且能共享宿主机的内核和网络资源等。附录D提供了一些面向Docker初学者的入门指导，有需要的读者可以参考。下面假定读者已经在自己的电脑上安装了Docker。

2.xswitch-free

在实际使用时，Kamailio通常不会单独使用，而是与FreeSWITCH或Asterisk等搭配使用，因此在讲解实际案例时也不免会用到它们。在本书中，我们以使用FreeSWITCH为例来讲解。



xswitch-free是一个FreeSWITCH Docker镜像，它是XSwitch云平台使用的Docker镜像，只是删除了一些私有模块，会一直保持更新。xswitch-free主要是为了帮助大家快速学习和使用FreeSWITCH，这样初学者就可以专注于学习FreeSWITCH本身，而不需要从头研究如何搞定一大堆依赖，并从源代码开始编译FreeSWITCH。

关于xswitch-free和FreeSWITCH本身，如果你想进一步了解，可以参考附录B。

5.1.2 在命令行上运行Kamailio

在本书中，我们使用kamailio/kamailio-ci:5.5.2-alpine这个Docker镜像。当你读到本书时，可能有更新的版本，大家可以查阅<https://hub.docker.com/r/kamailio/kamailio-ci>了解。

注意

我们这里没有使用latest版本，主要原因是latest在Docker标签中的含义永远是指最新版本，但它不是一个特定版本。建议大家在使用时选择更明确的版本。

以下命令可以启动Docker容器。

```
docker run --name kamailio --rm kamailio/kamailio-ci:5.5.2-alpine -m 64 -M 8
```

启动后，可以看到类似如下的输出：

```
Listening on
  udp: 127.0.0.1:5060
  udp: 172.17.0.4:5060
  tcp: 127.0.0.1:5060
  tcp: 172.17.0.4:5060
Aliases:
  tcp: 3ea18bb32dc7:5060
  tcp: localhost:5060
  udp: 3ea18bb32dc7:5060
  udp: localhost:5060
```

上述输出表示Kamailio已经运行并监听了上面的IP地址和端口，它的默认配置文件类似于第2章中介绍的配置文件。注意，这时Kamailio是无法使用的，因为Docker容器本身运行在NAT网络环境中。我们此时可以进入容器内部查看相关的配置文件，示例如下。

```
docker exec -it kamailio sh          # 进入容器内部 Shell
cat /etc/kamailio/kamailio.cfg       # 查看配置文件内容
```

如果你的宿主机是Linux，可以以host模式启动。

```
docker run --net=host --name kamailio --rm kamailio/kamailio-ci:5.5.2-alpine
```



这时候，就可以尝试用SIP客户端注册了。随便注册两个不同的账号（Kamailio默认脚本没有鉴权）就可以互打电话了。

在macOS和Windows上host模式可能不适用，可以继续以NAT模式启动，并将端口映射出来，示例如下。

```
docker run --name kamailio --rm \
-p 5060:5060/udp -p 5060:5060 \
kamailio/kamailio-ci:5.5.2-alpine -m64 -M8
```

使用NAT模式时，需要修改Kamailio的配置文件，这时候可以先进入Shell进行修改。修改完配置文件后再启动Kamailio。以下命令可启动容器并进入Shell。（由于使用--entrypoint参数覆盖了自启动入口，所以这里不会启动Kamailio。）

```
docker run --name kamailio --rm \
-p 5060:5060/udp -p 5060:5060 \
--entrypoint=/bin/sh -it \
kamailio/kamailio-ci:5.5.2-alpine -m64 -M8
```

在容器Shell内执行ip ad命令可以看到网卡eth0的IP地址，记下来（如笔者的是172.17.0.4，后面会用到），然后执行如下命令来启动Kamailio。

```
kamailio
```

注

默认Kamailio将会启动到后台。可以安装一个ngrep 来抓包并查看SIP消息。

```
apk add ngrep  
ngrep -p -q -Wbyline port 5060
```

apk是Alpine Linux上的软件包管理工具。在国内直接安装apk可能比较慢，可以尝试使用国内镜像，这里我们使用阿里云的镜像站。

```
echo "http://mirrors.aliyun.com/alpine/v3.13/main/" > /etc/apk/repositories  
echo "http://mirrors.aliyun.com/alpine/v3.13/community/" >> /etc/apk/repositories  
apk add ngrep # 这样安装就会快很多
```

下面可以尝试注册了。注意，由于是在NAT网络环境下，实际注册的地址是宿主机的地址，而不是容器内的地址。如笔者的宿主机IP地址是192.168.7.7，但注册时仍然要使用容器内的IP地址作为Domain（否则Kamailio不能通过myself检测）。用户名和密码可以任意，如图5-1所示。

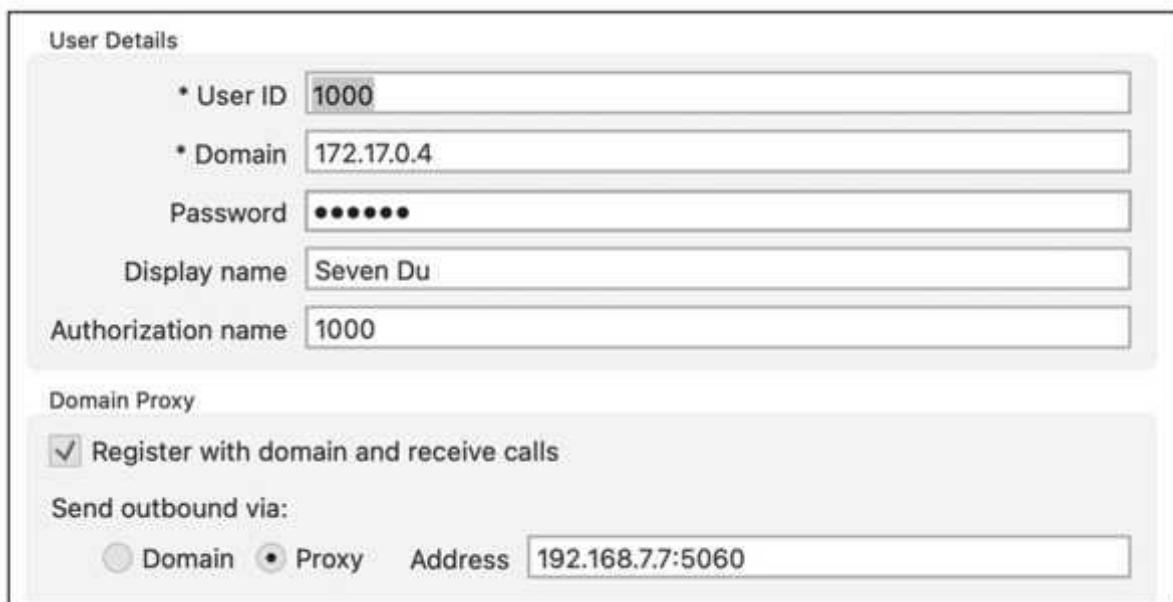


图5-1 使用Bria SIP客户端注册到Kamailio

注册两个账号互打（1000呼叫1001），发现不能成功应答。在ngrep中可以看到信令流程（后面详细介绍了大部分信令流程）。注意，消息中的第一行U ip:port->ip:port...是ngrep加上去的，表示消息的协议（U即UDP协议）和收发IP地址。每一行后的“.”也是ngrep加上去的，实际是\r\n。

Kamailio收到客户端1000发来的INVITE消息。客户端位于宿主机上，但由于NAT的缘故（笔者用的是macOS上的Docker），Kamailio（172.17.0.4）看到的来源IP地址是NAT网关的地址（172.17.0.1），而不是实际的来源IP地址。注意这个消息里的To是没有tag的。另外，为了方便读者阅读与对照学习，下面列出了大部分完整的SIP消息。大家在初次阅读的时候可以先浏览一下消息流程，再回过头来仔细研究各SIP头域在不同阶段的变化，在读到后面的例子时，也可以回过来查看本章的SIP消息。（当然如果自己动手实验会更直观，如果再能与本书介绍的SIP消息对比学习，会有更多收获。）

```

U 172.17.0.1:63184 -> 172.17.0.4:5060 #300
INVITE sip:1001@172.17.0.4 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:56507;branch=z9hG4bK-524287-1---a9172c1036a2975f;rport.
Max-Forwards: 70.
Contact: <sip:1000@172.17.0.1:61036;riinstance=6a14537ceb5c0433>.
To: <sip:1001@172.17.0.4>.

From: "Seven Du"<sip:1000@172.17.0.4>;tag=935eab27.
Call-ID: 88307ODUxMDUzN2QxZjQxYzcyMDVlNTNmN2MzMDExNDgzMGM.
CSeq: 1 INVITE.
Allow: SUBSCRIBE, NOTIFY, INVITE, ACK, CANCEL, BYE, REFER, INFO, OPTIONS, MESSAGE.
Content-Type: application/sdp.
Supported: replaces.
User-Agent: Bria 5 release 5.0.3 stamp 88307.
Content-Length: 154.
.
v=0.
o=- 1633434566283077 1 IN IP4 192.168.7.7.
s=Bria 5 release 5.0.3 stamp 88307.
c=IN IP4 192.168.7.7.
t=0 0.
m=audio 54880 RTP/AVP 9 8 0.
a=sendrecv.

```

Kamailio回复100 Trying:

```

U 172.17.0.4:5060 -> 172.17.0.1:63184 #301
SIP/2.0 100 trying -- your call is important to us.
Via: SIP/2.0/UDP 192.168.7.7:56507;branch=z9hG4bK-524287-1---a9172c1036a2975f;rpo
rt=63184;received=172.17.0.1.
To: <sip:1001@172.17.0.4>.
From: "Seven Du"<sip:1000@172.17.0.4>;tag=935eab27.
Call-ID: 88307ODUxMDUzN2QxZjQxYzcyMDVlNTNmN2MzMDExNDgzMGM.
CSeq: 1 INVITE.
Server: kamailio (5.5.2 (x86_64/linux)).
Content-Length: 0.

```

Kamailio查找本地的location表，找到1001的注册地址，并发送INVITE。可以看到，Kamailio根据自己的IP地址增加了一个Via头域，由于IP消息经过一次Kamailio转发，因此Max-Forwards字段值比收到时减了1，由70变成了69。

```

U 172.17.0.4:5060 -> 172.17.0.1:61170 #302
INVITE sip:1001@172.17.0.1:61170;ob SIP/2.0.
Record-Route: <sip:172.17.0.4;lr>.
Via: SIP/2.0/UDP 172.17.0.4;branch=z9hG4bK5e32.b38ef4b0ac75f1ae986114cf130ba0d2.0.
Via: SIP/2.0/UDP 192.168.7.7:56507;received=172.17.0.1;branch=z9hG4bK-524287-1---a9172c1036a2975f;rport=63184.
Max-Forwards: 69.
Contact: <sip:1000@172.17.0.1:61036;riinstance=6a14537ceb5c0433>.
To: <sip:1001@172.17.0.4>.
From: "Seven Du"<sip:1000@172.17.0.4>;tag=935eab27.
Call-ID: 88307ODUxMDUzN2QxZjQxYzcyMDVlNTNmN2MzMDExNDgzMGM.
CSeq: 1 INVITE.
Allow: SUBSCRIBE, NOTIFY, INVITE, ACK, CANCEL, BYE, REFER, INFO, OPTIONS, MESSAGE.
Content-Type: application/sdp.
Supported: replaces.
User-Agent: Bria 5 release 5.0.3 stamp 88307.
Content-Length: 154.
.
v=0.
o=- 1633434566283077 1 IN IP4 192.168.7.7.
s=Bria 5 release 5.0.3 stamp 88307.
c=IN IP4 192.168.7.7.
t=0 0.

```

```
m=audio 54880 RTP/AVP 9 8 0.  
a=sendrecv.
```

Kamailio收到1001客户端发来的100 Trying:

```
U 172.17.0.1:61170 -> 172.17.0.4:5060 #303  
SIP/2.0 100 Trying.  
Via: SIP/2.0/UDP  
172.17.0.4;received=192.168.7.7;branch=z9hG4bK5e32.b38ef4b0ac75f1ae986114cf130ba0d2.0.  
Via: SIP/2.0/UDP 192.168.7.7:56507;rport=63184;received=172.17.0.1;branch=z9hG4  
bK-524287-1---a9172c1036a2975f.  
Record-Route: <sip:172.17.0.4;lr>.  
Call-ID: 88307ODUxMDUzN2QxZjQxYzcyMDV1NTNmN2MzMDExNDgzMGM.  
From: "Seven Du" <sip:1000@172.17.0.4>;tag=935eab27.  
To: <sip:1001@172.17.0.4>.  
CSeq: 1 INVITE.  
Content-Length: 0.
```

Kamailio收到客户端1001发来的180 Ringing, 我们看到, 这里的To头域有一个tag参数:

```
U 172.17.0.1:61170 -> 172.17.0.4:5060 #304  
SIP/2.0 180 Ringing.  
Via: SIP/2.0/UDP  
172.17.0.4;received=192.168.7.7;branch=z9hG4bK5e32.b38ef4b0ac75f1ae986114cf130ba0d2.0.  
Via: SIP/2.0/UDP 192.168.7.7:56507;rport=63184;received=172.17.0.1;branch=z9hG4  
bK-524287-1---a9172c1036a2975f.  
Record-Route: <sip:172.17.0.4;lr>.  
Call-ID: 88307ODUxMDUzN2QxZjQxYzcyMDV1NTNmN2MzMDExNDgzMGM.  
From: "Seven Du" <sip:1000@172.17.0.4>;tag=935eab27.  
To: <sip:1001@172.17.0.4>;tag=pJ1xu0ubvp6fOHMRyd2PkxQOJhFD1Qjd.  
CSeq: 1 INVITE.  
Contact: <sip:1001@172.17.0.1:61170;ob>.  
Allow: PRACK, INVITE, ACK, BYE, CANCEL, UPDATE, INFO, SUBSCRIBE, NOTIFY, REFER,  
MESSAGE, OPTIONS.  
Content-Length: 0.
```

将180 Ringing转发给主叫客户端1000:

```
U 172.17.0.4:5060 -> 172.17.0.1:63184 #305  
SIP/2.0 180 Ringing.  
Via: SIP/2.0/UDP 192.168.7.7:56507;rport=63184;received=172.17.0.1;branch=z9hG4  
bK-524287-1---a9172c1036a2975f.  
Record-Route: <sip:172.17.0.4;lr>.  
Call-ID: 88307ODUxMDUzN2QxZjQxYzcyMDV1NTNmN2MzMDExNDgzMGM.  
From: "Seven Du" <sip:1000@172.17.0.4>;tag=935eab27.  
To: <sip:1001@172.17.0.4>;tag=pJ1xu0ubvp6fOHMRyd2PkxQOJhFD1Qjd.  
CSeq: 1 INVITE.  
Contact: <sip:1001@172.17.0.1:61170;ob>.  
Allow: PRACK, INVITE, ACK, BYE, CANCEL, UPDATE, INFO, SUBSCRIBE, NOTIFY, REFER,  
MESSAGE, OPTIONS.  
Content-Length: 0.
```

收到被叫1001的应答消息:

```
U 172.17.0.1:61170 -> 172.17.0.4:5060 #306  
SIP/2.0 200 OK.  
Via: SIP/2.0/UDP  
172.17.0.4;received=192.168.7.7;branch=z9hG4bK5e32.b38ef4b0ac75f1ae986114cf130ba0d2.0.
```

```

Via: SIP/2.0/UDP 192.168.7.7:56507;rport=63184;received=172.17.0.1;branch=z9hG4
bK-524287-1---a9172c1036a2975f.
Record-Route: <sip:172.17.0.4;lr>.
Call-ID: 883070DUxMDUzN2QxZjQxYzcyMDV1NTNmN2MzMDExNDgzMGM.
From: "Seven Du" <sip:1000@172.17.0.4>;tag=935eab27.
To: <sip:1001@172.17.0.4>;tag=pJIxuOubvp6f0HMRyd2PkxQOJhFD1Qjd.
CSeq: 1 INVITE.
Allow: PRACK, INVITE, ACK, BYE, CANCEL, UPDATE, INFO, SUBSCRIBE, NOTIFY, REFER,
MESSAGE, OPTIONS.
Contact: <sip:1001@172.17.0.1:61170;ob>.
Supported: replaces, 100rel, norefersub.
Content-Type: application/sdp.
Content-Length: 253.
.
v=0.
o=- 3842423366 3842423367 IN IP4 172.17.0.1.
s=pjmedia.
b=AS:84.
t=0 0.
a=X-nat:0.
m=audio 4000 RTP/AVP 8.
c=IN IP4 172.17.0.1.
b=TIAS:64000.
a=rtcp:4001 IN IP4 172.17.0.1.
a=sendrecv.
a=rtpmap:8 PCMA/8000.
a=ssrc:761892578 cname:6f013e233a96a4c4.

```

将应答消息转发给主叫1000:

```

U 172.17.0.4:5060 -> 172.17.0.1:63184 #307
SIP/2.0 200 OK.
Via: SIP/2.0/UDP 192.168.7.7:56507;rport=63184;received=172.17.0.1;branch=z9hG4
bK-524287-1---a9172c1036a2975f.
Record-Route: <sip:172.17.0.4;lr>.
Call-ID: 883070DUxMDUzN2QxZjQxYzcyMDV1NTNmN2MzMDExNDgzMGM.
From: "Seven Du" <sip:1000@172.17.0.4>;tag=935eab27.
To: <sip:1001@172.17.0.4>;tag=pJIxuOubvp6f0HMRyd2PkxQOJhFD1Qjd.
CSeq: 1 INVITE.
Allow: PRACK, INVITE, ACK, BYE, CANCEL, UPDATE, INFO, SUBSCRIBE, NOTIFY, REFER,
MESSAGE, OPTIONS.
Contact: <sip:1001@172.17.0.1:61170;ob>.
Supported: replaces, 100rel, norefersub.
Content-Type: application/sdp.
Content-Length: 253.
.
v=0.
o=- 3842423366 3842423367 IN IP4 172.17.0.1.
s=pjmedia.
b=AS:84.
t=0 0.
a=X-nat:0.
m=audio 4000 RTP/AVP 8.
c=IN IP4 172.17.0.1.
b=TIAS:64000.
a=rtcp:4001 IN IP4 172.17.0.1.
a=sendrecv.
a=rtpmap:8 PCMA/8000.
a=ssrc:761892578 cname:6f013e233a96a4c4.

```

收到被叫1001的应答消息:

```

U 172.17.0.1:61170 -> 172.17.0.4:5060 #308
SIP/2.0 200 OK.

```

将应答消息转发给主叫1000:

```

U 172.17.0.4:5060 -> 172.17.0.1:63184 #309
SIP/2.0 200 OK.

```

收到被叫1001的应答消息：

```
U 172.17.0.1:61170 -> 172.17.0.4:5060 #311  
SIP/2.0 200 OK.
```

将应答消息转发给主叫1000：

```
U 172.17.0.4:5060 -> 172.17.0.1:63184 #312  
SIP/2.0 200 OK.
```

至此，呼叫接通了，但由于Kamailio没有收到主叫的ACK证实消息，因而也没有给被叫发ACK，导致被叫在接听后持续发200 OK。以上转发呼叫的流程如图5-2所示。

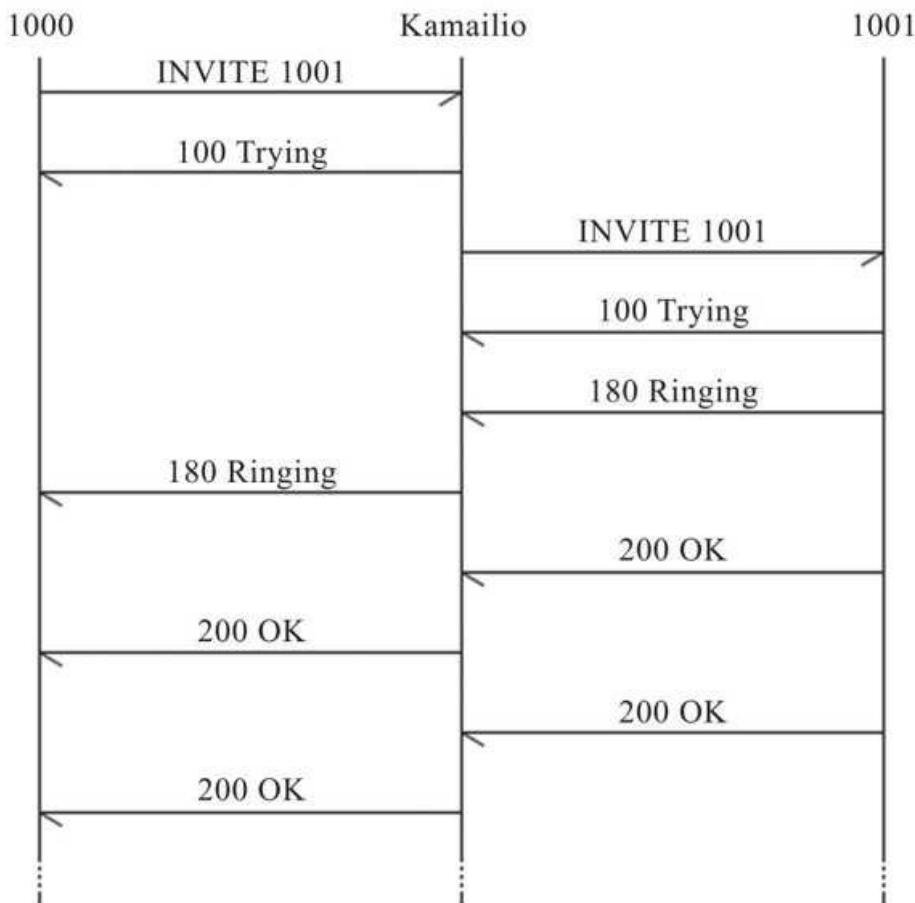


图5-2 通过Kamailio转发呼叫的流程图

之所以会出现上述情况，是因为主叫1000收到200 OK消息后要给Record-Route:<sip:172.17.0.4;lr>这个地址回复SIP消息，但1000在宿主机上，无法回复容器内部的这个IP地址，因而造成ACK消息无法送达。该问题在实际场景中也很常见。

下面我们来解决这个问题。

(1) 停掉Kamailio，具体命令如下。

```
killall kamailio
```

(2) 修改配置文件vi/etc/kamailio/kamailio.cfg，找到# listen=udp:10.0.0.10:5060，然后在其后面增

注

加如下两行代码。

```
listen=udp:172.17.0.4:5060 advertise 192.168.7.7:5060
listen=tcp:172.17.0.4:5060 advertise 192.168.7.7:5060
```

重新启动Kamailio，可以看到监听地址变了，具体如下。

```
# kamailio
Listening on
    udp: 172.17.0.4:5060 advertise 192.168.7.7:5060
    tcp: 172.17.0.4:5060 advertise 192.168.7.7:5060
Aliases:
    tcp: 4e2e9ac37dcf:5060
    udp: 4e2e9ac37dcf:5060
```

(3) 重新注册所有客户端并呼叫，然后挂机，一切正常。我们可以看到Kamailio发送给主机的200 OK消息变为如下形式（Record-Route头域变成了NAT的外网地址，即宿主机的IP地址。为节省篇幅，这里给出的消息内容进行了一些精减）：

```
U 172.17.0.4:5060 -> 172.17.0.1:62943 #16
SIP/2.0 200 OK.
Via: SIP/2.0/UDP
192.168.7.7:56507;rport=62943;received=172.17.0.1;branch=z9hG4bK-524287-1---084a851047dbfa5f.
Record-Route: <sip:192.168.7.7;lr>.
Call-ID: 88307MWZkZDk5MTN1NWViYjBjMmNlMjVkJzljNWY0MWMzjI.
```

当然，上面仅是SIP层面的NAT处理。有关RTP层的消息（SDP相关）可以参见8.14节的示例。

下面分享几个小技巧。

- Kamailio默认会启动到后台，如果在配置文件中增加fork=false，则会启动到前台，可以直接用Ctrl+C退出，而无须通过killall命令关停Kamailio。这在学习时比较方便。
- 在Docker的NAT模式下，UDP通道映射有时保持不好，经常需要重新注册才能打通电话，这时可以换TCP注册。
- Kamailio启动到前台时，建议修改配置文件，即增加Kamailio的日志级别（如debug=3）以便查看更多日志。
- 如果启动容器时有同名的容器存在，或存在映射端口冲突，则启动会失败。在这种情况下可以换一个名字或端口启动。也可以用docker ps命令查看所有正在运行的容器，用docker stop停止并用docker rm删掉这些导致冲突的容器（删除之前要确保不再需要了），再启动新的容器。

5.1.3 将配置文件保存到宿主机

在上面的例子中，我们直接在容器内修改Kamailio的配置文件，在容器重建时会丢失修改。为了能长期保存劳动成果，可以将宿主机上的文件或文件夹映射到容器里面。不过，前提是得先有这个文件夹。下面先启动一个临时容器，从里面复制一个etc文件夹出来。

```
docker create --name kamailio kamailio/kamailio-ci:5.5.2-alpine
docker cp kamailio:/etc/kamailio /tmp/
docker rm kamailio
ls -l /tmp/kamailio
```

重新启动容器，并通过-v参数将宿主机的目录映射到容器里面。如果想让Kamailio自动启动，可以

去掉--entrypoint参数这一行，否则需要手动启动。具体的代码如下。

```
docker run --name kamailio --rm \
-p 5060:6060/udp -p 5060:6060 \
-v /tmp/kamailio:/etc/kamailio \
--entrypoint=/bin/sh -it \
kamailio/kamailio-ci:5.5.2-alpine -m64 -M8
```

这样就可以随时在宿主机上用你喜欢的编辑器编辑配置文件并在容器内部重启Kamailio了。

5.1.4 使用Docker Compose管理容器

在命令行上输入比较长的命令会比较麻烦，而且有时候容器中的应用还需要连接数据库等，这时就需要启动多个容器，会更复杂。Docker Compose使用可编排的YAML配置文件，可以同时管理很多容器的启停，非常方便。



使用如下命令克隆本书代码示例仓库。

```
git clone https://git.xswitch.cn/book/kamailio-book-examples.git
cd kamailio-book-examples
```

上述仓库中的例子使用Makefile维护。Makefile是make工具中使用的工程配置文件，一般来说，它里面有很多目标（Target），可以作为命令行make的参数，如make network就会执行network: 后面的命令。如果你的系统上没有make，也可以直接查看Makefile文件的内容，找到相应的目标后面的命令并执行，如make network实际上是执行docker network create kamailio-example命令，下同。



初始化环境，仅需执行一次，它会产生.env文件。

```
make setup
```

修改docker/.env文件里的端口号、IP地址等，使其成为适合你自己环境的地址。

初始化网络。仅在每次宿主机启动后第一次使用网络时需要进行初始化操作，具体命令如下。

```
make network
```

启动容器：

```
make up
```

进入容器：

```
make sh
```

启动Kamailio：

```
/start-kam.sh
```

为了方便学习与调试，我们没有让Kamailio自动启动，而是手动执行上述启动命令，这样可以随时通过Ctrl+C组合键退出Kamailio。可以在宿主机上并行开启其他终端并使用make sh命令进入同一个容器。如果在生产系统上需要自动启动Kamailio，可以修改kam.yml中的entrypoint参数。

所有容器名都以kb-（Kamailio-Book的缩写）开头，这样可以避免与宿主机上其他系统冲突。

默认的配置文件是从源代码目录下复制过来的，为了简化，这里我们修改了文件名，对应关系如下：

- kamailio-basic-kemi.cfg与/etc/kamailio/kamailio.cfg对应。
- kamailio-basic-kemi-lua.lua与/etc/kamailio/kamailio.lua对应。

为了能跑通本书的示例脚本，我们做了一些修改（下面内容仅供参考，以随书附赠的代码中的实际文件为准），下面逐一介绍。

注释掉kamailio.cfg中以下选项（两个#开头代表注释掉），因为我们没有MySQL。

```
##!define WITH_MYSQL  
##!define WITH_AUTH  
##!define WITH_USRLOCDB  
##!define WITH_NAT  
##!define WITH_ACCDB
```

增加如下模块：

```
loadmodule "cfgutils.so"
```

增加如下选项：

```
#!define WITH_CFGLUA
```

Lua脚本路径调整为：

```
modparam("app_lua", "load", "/etc/kamailio/kamailio.lua")
```

增加以下监听项：

```
listen=udp:KAM_IP_LOCAL:KAM_SIP_PORT advertise KAM_IP_PUBLIC:KAM_SIP_PORT  
listen=tcp:KAM_IP_LOCAL:KAM_SIP_PORT advertise KAM_IP_PUBLIC:KAM_SIP_PORT
```

其中，大写的变量是从/start-kam.sh中来的，本地IP地址是自动计算的（因为不同的人用的IP是不同的），外部IP地址和端口是从环境变量（来自.env文件）中获取的。

启动完毕后，就可以像上一节一样进行注册和分机互打了。后面的例子都是在这两个配置文件的基础上进行的。

5.2 将SIP呼叫转发到FreeSWITCH

首先，启动FreeSWITCH容器并进入控制台，具体命令如下。

```
make up-fs1          # 启动 FreeSWITCH 容器  
make bash-fs1        # 进入 FreeSWITCH 容器中的 Shell  
fs cli              # 进入 FreeSWITCH 控制台
```

进入控制台后可以看到SIP消息等。默认的FreeSWITCH容器提供了以下号码，这些号码可以用于

 测试：

- 10000200：返回200 OK后挂机。
- 10000404：返回404，空号。
- 10000486：返回486，用户忙。
- 10009196：回声，不会主动挂机。
- 9196：同10009196。

修改kamailio.cfg，让它使用我们的Lua示例脚本（examples.lua），后文中大部分使用该脚本。

```
modparam("app_lua", "load", "/etc/kamailio/examples.lua")
```

示例脚本定义了FS1_URI和FS1_UDP两个字符串常量——代表FreeSWITCH的URI和UDP地址。在笔者的环境中，FreeSWITCH的地址是172.18.0.3:5080，但在不同的环境中IP地址可能不同，所以这里的地址可能要换成你自己的FreeSWITCH地址。如果你使用了随书附赠的示例代码实现的Docker环境，那么其中的kb-fs1就是FreeSWITCH的域名。该域名是由Docker自动维护的，它跟容器服务的名字相关。使用域名访问比使用IP地址要方便些。本书后面的例子尽量使用域名而非实际的IP地址。

脚本中实现了一个k_dofile()函数，其用于读入其他示例文件，这样就可以很简单地测试不同的例子了。另外，我们会使用SIP客户端测试发送SIP消息，但是这里要注意，并不是所有的客户端都支持不注册就发SIP消息，所以，我们写了一个ksr_register_always_ok()函数（见代码清单5-1），该函数对所有的注册消息都返回“200 OK”。如果需要测试对注册消息的处理或转发，可以将该函数注释掉。

代码清单 5-1

```
FS1_URI = 'sip:kb-fsl:5080'
FS1_UDP = 'udp:kb-fsl:5080'

function k_dofile(file)
    local BASE="/etc/kamailio/examples/"
    dofile(BASE .. file)
end

function ksr_register_always_ok()
    if KSR.is_method_in("R") then
        KSR.sl.sl_send_reply(200, "OK");
        KSR.x.exit()
    end
end

-- 读入其他文件
k_dofile("stateless_forward.lua")
```

重启Kamailio使上述设置生效。之后，修改Lua脚本，如果只是修改了Lua脚本而没有修改 kamailio.cfg，则可以直接在命令行重载Lua脚本，且无须重启Kamailio，具体命令如下：

```
kamcmd app_lua.reload
```

上述命令没有任何输出，这表示成功了。一般来说，只要没有错误输出，都表示成功了。具体的转发脚本（stateless_forward.lua）将在6.2.2节讲解。

5.3 从简单的路由脚本开始

本节从一个最简单的路由脚本开始，逐渐扩展到更多功能。下面是一个简单的路由脚本simple-log.lua，其作用是收到任何SIP消息后，打印一条日志，然后回复“200 OK”。

simple-log.lua的内容如下：

```
function ksr_request_route()
    KSR.info("Got a SIP Message, method=" .. KSR.pv.gete('$rm') ..
        " from IP: " .. KSR.pv.gete("$si") .. "\n")
    KSR.sl.send_reply("200", "OK")
end
```

把代码清单5-1中的k_dofile行（最后一行）换成如下内容即可以使用上述路由脚本：

```
k_dofile("simple-log.lua")
```

启动Kamailio后，使用任何SIP客户端给上述路由脚本发任何消息，都会收到“200OK”。如我们可以使用sipexer发送SIP OPTIONS消息：

```
sipexer 192.168.7.7:5060
```

在Kamailio中可以看到如下日志：

```
sr_kemi_core_info(): Got a SIP Message, method=OPTIONS from IP: 172.22.0.1
```

关于sipexer我们将在5.4.5节讲述。

5.4 Kamailio命令行工具

在使用和开发Kamailio的过程中，经常需要对运行中的Kamailio进行控制、状态查询等操作。Kamailio本质上是一种客户端-服务器（Client-Server）的结构，客户端软件可以通过RPC（远程过程调用）对Kamailio进行控制、状态查询。由于历史原因，Kamailio提供了多种命令行工具，下面就分别介绍它们的特点和使用方法。

- **kamctl**: Shell脚本，名字是Kamailio Control Tool的缩写，用于通过JSON-RPC控制Kamailio。
- **kamdbctl**: Shell脚本，名字是Kamailio Database Control Tool的缩写，用于操作Kamailio的数据仓库。
- **kamcmd**: C语言写的程序，名字是Kamailio Command Line BinRPC Tool的缩写，使用BinRPC控制Kamailio。
- **kamcli**: Python 3脚本，名字是Kamailio Command Line Tool的缩写，用于控制Kamailio。
- **sipexer**: 一个SIP命令行工具，用于给Kamailio发送SIP消息。

5.4.1 kamctl

kamctl是一个Shell命令行工具，可调用操作系统自带的命令，如echo、cat、grep、awk等。kamctl通过JSON-RPC与Kamailio进行交互，比如直接在Shell命令行上执行kamctl uptime命令（获取Kamailio启动了多长时间），在本书写作时，会输出如下结果。

```
{
  "jsonrpc": "2.0",
  "result": {
    "now": "Sun Mar  6 02:02:06 2022",
    "up_since": "Sun Mar  6 02:01:38 2022",
    "uptime": 28
  },
  "id": 655
}
```



Linux上有一个jq 命令可以用于解析JSON字符串。以下命令可获取result部分。

```
# kamctl uptime | jq .result
{
  "now": "Sun Mar  6 02:10:04 2022",
  "up_since": "Sun Mar  6 02:01:38 2022",
  "uptime": 506
}
```

以下命令可获取result.uptime部分。

```
# kamctl uptime | jq .result.uptime
513
```

细心的读者可能已注意到了，上面两次的uptime值是不同的，这是因为在实际的场景下，这个时间是一直在变化的。

此外，也可以使用如下命令查看Kamailio启动的进程数，这在研究Kamailio进程模型时非常有用，如在笔者的环境中输出如下代码。

```

# kamctl ps

{
    "jsonrpc": "2.0",
    "result": [
        {
            "IDX": 0,
            "PID": 617,
            "DSC": "main process - attendant"
        },
        {
            "IDX": 1,
            "PID": 620,
            "DSC": "udp receiver child=0 sock=172.22.0.2:35060 (192.168.7.7:35060)"
        },
        {
            "IDX": 2,
            "PID": 622,
            "DSC": "udp receiver child=1 sock=172.22.0.2:35060 (192.168.7.7:35060)"
        },
        {
            "IDX": 3,
            "PID": 624,
            "DSC": "slow timer"
        },
        {
            "IDX": 4,
            "PID": 626,
            "DSC": "timer"
        },
        {
            "IDX": 5,
            "PID": 628,
            "DSC": "secondary timer"
        },
        {
            "IDX": 6,
            "PID": 630,
            "DSC": "JSONRPCS FIFO"
        },
        {
            "IDX": 7,
            "PID": 631,
            "DSC": "JSONRPCS DATAGRAM"
        },
        {
            "IDX": 8,
            "PID": 633,
            "DSC": "ctl handler"
        },
        {
            "IDX": 9,
            "PID": 635,
            "DSC": "Http Async Worker"
        },
        {
            "IDX": 10,
            "PID": 637,
            "DSC": "WEBSOCKET KEEPALIVE"
        },
        {
            "IDX": 11,
            "PID": 638,
            "DSC": "WEBSOCKET TIMER"
        },
        {
            "IDX": 12,
            "PID": 641,
            "DSC": "tcp receiver (generic) child=0"
        },
        {
            "IDX": 13,
            "PID": 644,
            "DSC": "tcp receiver (generic) child=1"
        },
        {
            "IDX": 14,
            "PID": 646,
            "DSC": "tcp main process"
        }
    ],
    "id": 1198
}

```

使用不加参数的kamctl命令会列出简明的使用帮助。其中，`rpc`子命令用于进行RPC调用，如在修改了dispatcher模块的配置文件或数据库后，可以使用如下命令重载和查询分发策略。

```

kamctl rpc dispatcher.reload
kamctl rpc dispatcher.list

```

关于dispatcher模块提供的RPC方法，可以参见6.3节以及模块相关说明文档。不同的模块都会导出不同的方法，具体的介绍可以从模块相关的参考文档中获取。

为了让kamctl能连接到Kamailio，需要在Kamailio中加载jsonrpcs模块，并开启FIFO（或UNIXSocket）连接支持（默认都开启，详见随书附赠代码中的kamailio.cfg中的示例）。在Kamailio启动后可以通过如下命令检查连接情况。

```
# ls -l /run/kamailio/          # 执行该命令会列出目录下的所有文件  
total 0                         # 0个普通文件，但有3个特殊文件（在UNIX中一切都是文件）  
srw----- 1 root    root        0 Mar  6 02:01 kamailio_ctl  
prw-rw---- 1 root    root        0 Mar  6 02:01 kamailio_rpc fifo  
srw-rw---- 1 root    root        0 Mar  6 02:01 kamailio_rpc.sock
```

提示

熟悉Linux的读者都应该知道，ls -l命令用于列目录，每一行都代表一个文件，其中第一个字符-代表普通文件、d代表目录。在上述示例中，s代表Socket文件，p代表Pipe，即先入先出（FIFO）管道文件。

kamctl执行时会检查一个配置文件（/etc/kamailio/kamctlrc），该文件是一个典型的Linux配置文件，里面的配置项都是“参数=值”的形式，其中#代表注释，例如下面的代码。

```
CTLENGINE="RPCFIFO"  # 设置通过 FIFO 连接 Kamailio 服务器  
RPCFIFOPATH="/var/run/kamailio/kamailio_rpc fifo" # 设置 FIFO 管道文件
```

kamctl也可以操作数据库，如通过kamctl add username@domain password添加一个用户等。操作数据库之前需要在配置文件中配置数据库的连接参数，关于数据库和连接参数，我们将在第7章讲解。

值得注意的是，上面的配置文件其实就是配置环境变量，也可以在命令行上直接使用环境变量，这在同一台服务器上需要连接多个Kamailio时比较有用，示例如下。

```
RPCFIFOPATH="/var/run/kamailio/kamailio_rpc_server1 fifo" kamctl ps  
RPCFIFOPATH="/var/run/kamailio/kamailio_rpc_server2 fifo" kamctl ps
```

当然，kamctl本身就是一个Shell脚本，在高级定制化的应用中你可以找到它，根据你的需要也可以自行修改。

5.4.2 kamdbctl

与kamctl类似，kamdbctl也是一个Shell脚本，它们甚至都使用同一个kamctlrc配置文件。

kamdbctl的主要作用是创建和初始化数据库，向数据库中添加数据等，不带参数的命令会输出一个帮助，下面的代码中仅用文替换掉必要的英文说明。

```
# kamdbctl
/usr/sbin/kamdbctl 5.4.0

usage: kamdbctl create <db name or db_path, optional> .( 创建数据库 )
kamdbctl drop <db name or db_path, optional> ..( 删除整个数据库和表 )
kamdbctl reinit <db name or db_path, optional>. ( 删掉整个数据库并重新初始化 )
kamdbctl backup <file> .....( 将数据库备份成文件 )
kamdbctl restore <file> .....( 从备份文件中恢复数据库 )
kamdbctl copy <new_db> .....( 从现有数据库中复制一个新数据库 )
kamdbctl presence .....( 增加 presence 相关的表 )
kamdbctl extra .....( 增加有 extra 标记的不太常用的表 )
kamdbctl dbuid .....( 增加 uid 相关的表 )
kamdbctl dbonly .....( 仅创建空数据库, 不创建表 )
kamdbctl grant .....( 设置数据库用户权限 )
kamdbctl revoke .....( 收回数据库用户权限 )
kamdbctl add-tables <gid> .....( 仅创建 gid 标志的组中的表 )
kamdbctl pframework create .....( 创建一个 provisioning framework
示例文件 )
```

kamdbctl只是一个简单的脚本程序，如果某些参数不符合你的要求（如你不是用root用户或kamailio用户连接数据库），可以直接修改该脚本。

该脚本功能有限，因此这里就不多介绍了。如果脚本达不到你的要求，你除了可以直接修改脚本外，还可以直接手工使用SQL操作数据库。更多数据库相关的操作我们将在第7章中讲到。

5.4.3 kamcmd



kamcmd是使用C语言写的一个命令行工具，它可以像kamctl那样一次性执行命令，也可以批量执行。此外，它还有一个交互式的命令行环境，在该环境下可以很方便地翻阅和再执行历史命令。举例如下（其中kamcmd>为提示符）。

```
# kamcmd          # 进入交互多命令行

kamcmd 1.5
Copyright 2006 iptelorg GmbH
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.

kamcmd> ps      # 执行ps命令, 类似于 kamctl ps
617 main process - attendant
620 udp receiver child=0 sock=172.22.0.2:35060 (192.168.7.7:35060)
622 udp receiver child=1 sock=172.22.0.2:35060 (192.168.7.7:35060)
624 slow timer
626 timer
628 secondary timer
630 JSONRPCS FIFO
631 JSONRPCS DATAGRAM
633 ctl handler
635 Http Async Worker
637 WEBSOCKET KEEPALIVE
638 WEBSOCKET TIMER
641 tcp receiver (generic) child=0
644 tcp receiver (generic) child=1
646 tcp main process
```

使用kamcmd -h命令会输出如下帮助信息。

```

# kamcmd -h

version: kamcmd 1.5
Usage: kamcmd [options] [-s address] [ cmd ]
Options:
  -s address Kamailio 服务的 UNIX Socket 地址或主机名
  -R name 强制回复 Socket 名称，针对 UNIX Datagram Socket 模式
  -D dir 如果使用 UNIX Datagram Socket 模式但又没有（通过 -R）强制指定回复 Socket 名称时
           自动在该目录下创建一个
  -f format 结果打印的格式。format 是一个包含 %v 的字符串，它的值从回复消息中读取，
           如果要在结果中打印 %v 字符串，则可以使用 %%v 这种方法
  -v      详细模式
  -V      版本号

  -h      本帮助信息
address:
  [proto:]name[:port] proto 的取值有 tcp、udp、unixs 或 unixd，如
  tcp:localhost:2049、unixs:/tmp/kamailio_ctl
cmd:
  method [arg1 [arg2...]]
arg:
  字符串或数字。如果要将数字强制解析成字符串，可以在前面加 "s:"，如 s:1

```

一些kamcmd命令示例如下。

```

kamcmd -s unixs:/tmp/kamcmd_ctl system.listMethods
kamcmd -f "pid: %v desc: %v\n" -s udp:localhost:2047 core.ps
kamcmd ps # 使用默认的控制 Socket 连接
kamcmd # 使用默认的 Socket 连接并进入交互模式
kamcmd -s tcp:localhost # 连接该 tcp 主机的默认端口并进入交互模式

```

kamcmd交互模式默认直接使用RPC通信，因而可以直接执行RPC方法，示例如下。

```

kamcmd> dispatcher.reload
kamcmd> dispatcher.list

```

可以使用help列出当前系统支持的RPC方法，示例如下。

```

kamcmd> help
app_lua.api_list          # 核心和模块中导出的 Lua API 列表
app_lua.list                # Lua 脚本列表
app_lua.reload              # 重载 Lua 脚本
dispatcher.add               # 增加分发策略项
dispatcher.list              # 分发策略列表
dispatcher.reload             # 重载分发策略
dispatcher.remove             # 删除一个分发策略项
builtin: ?                   # 内置命令，相当于 help
builtin: help                 # 本帮助信息
builtin: version              # Kamailio 版本
builtin: quit                  # 退出交互式环境
builtin: exit                  # 退出，同 quit
builtin: warranty              # 支持
builtin: license                # 许可证

```

限于篇幅，其他方法在此就不多列举了，后面我们讲到对应的模块时，还会进行讲解。

5.4.4 kamcli



kamcli 是一个“新轮子”，使用Python 3编写，旨在代替kamctl。因为后者是使用Shell脚本编写的，所以能力有限。Python是真正的编程语言，功能丰富，处理RPC请求以及数据库连接等操作都有现成的库，而且在大多数Linux系统上都有安装。

kamcli依赖于一些Python 3环境，在常见的Linux系统上可以使用如下方式安装。

```
Debian/Ubuntu: apt-get install python3 python3-pip python3-setuptools python3-dev  
CentOS: yum install python3 python3-devel python3-pip python3-setuptools
```

在本书使用的Kamailio Docker环境中（使用Alpine Linux），可以使用如下方式安装kamcli。首先，安装编译所需的依赖库，具体命令如下。

```
apk add git py3-pip python3-dev gcc g++ musl-dev mysql-client mariadb-dev
```

然后就可以使用如下方式下载源代码并安装了。

```
$ git clone https://github.com/kamailio/kamcli.git      # 克隆源代码  
$ cd kamcli                                         # 进入源代码目录  
$ pip3 install -r requirements/requirements.txt       # 安装相应的 Python 依赖库  
$ pip3 install mysqlclient                           # 安装 MySQL Python 客户端库  
$ pip3 install --editable .                         # 安装 kamcli
```

安装完毕后可以执行如下命令查看帮助信息。

```
kamcli --help
```

kamcli使用.ini格式的配置文件，路径查找顺序如下：

- (1) ./kamcli/kamcli.ini。
- (2) ./kamcli.ini。
- (3) /etc/kamcli/kamcli.ini。
- (4) ~/.kamcli/kamcli.ini。
- (5) 在命令行上使用-c或--config指定的路径。

可以使用如下方法安装配置文件。

```
cd kamcli # 源代码主目录  
kamcli config install           # 安装到全局系统目录，如 /etc/kamcli/kamcli.ini  
sudo kamcli config install -u   # 安装到用户目录，如 $HOME/.kamcli/kamcli.ini
```

kamcli默认使用YAML格式进行输出，如执行uptime和ps的命令，其输出格式如下。

```
# kamcli uptime  
  
id: 5841  
jsonrpc: '2.0'  
result:  
  now: Sun Mar  6 08:31:57 2022  
  up_since: Sun Mar  6 07:47:52 2022  
  uptime: 2645  
  
# kamcli ps  
  
  0 2898 main process - attendant  
  1 2901 udp receiver child=0 sock=172.22.0.2:35060 (192.168.7.7:35060)  
  2 2903 udp receiver child=1 sock=172.22.0.2:35060 (192.168.7.7:35060)  
  3 2905 slow timer  
  4 2907 timer  
... (略)
```

可以通过如下方式执行JSON-RPC命令。

```
kamcli -d jsonrpc dispatcher.list
```

或进入一个交互式Shell环境（使用Ctrl+D或: q可退出），示例如下。

```
kamcli shell
```

然后输入命令（如dispatcher list）并按回车键就可以了。在输入命令的过程中，还会弹出相关提示，可以通过按Tab键做命令补全，如图5-3所示。

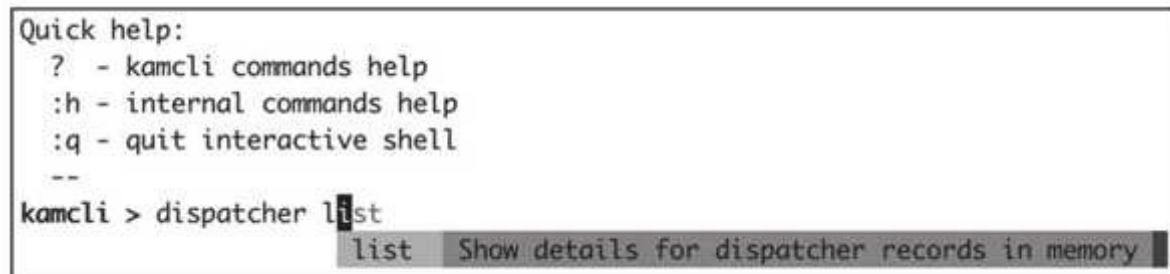


图5-3 kamcli命令提示和Tab补全

提示

kamcli使用Python 3的SQLAlchemy库连接数据库，其虽然可以支持多种数据库，但到本书完稿时，其作者仅在MySQL上做过测试。

更多的使用方法和配置文件参数可以参见源代码中的README说明，在此就不赘述了。

5.4.5 sipexer



上面的命令行工具都是通过JSON-RPC控制Kamailio服务器本身的，而sipexer是一个SIP客户端工具。由于在运行Kamailio的时候经常需要测试各种不同的SIP消息，但是标准的SIP客户端软件都不方便随意改动SIP消息中的各种头域（何况在进行SIP测试时除了正常消息外有时候还要故意测试收发错误的消息）。因此，Kamailio的作者Daniel Constantin Mierla自己写了这个工具。

sipexer这个名字是随便选的，只是为了容易读写。如果想给它一个实际意义，那么可以把它当作SIP EXECutor的缩写。

sipexer使用Go语言编写，因而它直接使用了Go语言的模板系统做SIP消息模板，在命令行上填充不同的值，这些值就可以转换成真正的SIP消息。sipexer支持UDP、TCP、TLS以及WebSocket（WS和WSS）传输协议。

sipexer的安装和使用都非常简单，读者可以直接克隆其GitHub镜像并按README里的方法编译安装（需要Go语言开发环境）：

```
git clone https://github.com/miconda/sipexer
```

如果没有Go语言环境，也可以直接从如下地址下载适用于你的操作系统的版本，然后直接运行二进制客户端进行安装：

```
https://github.com/miconda/sipexer/releases
```

下面看一下sipexer的使用方法。比如，可以通过如下几种方法发送SIP OPTIONS消息：

```
sipexer
sipexer 127.0.0.1
sipexer 127.0.0.1 5060
sipexer udp 127.0.0.1 5060
sipexer udp:127.0.0.1:5060
sipexer sip:127.0.0.1:5060
sipexer "sip:127.0.0.1:5060;transport=udp"
```

下面是一个简明的sipexer特性列表：

- 发送OPTIONS请求，可以快速检查SIP服务器是否还活着。（另一个类似的工具是sipsak。）
- 支持注册、注销操作，可以设置自定义的Expires以及Contact头域，支持明文或HA1加密的密码。
- 自定义SIP头域。
- 使用模板系统自定义SIP消息。
- SIP消息中的各个部分可以通过命令行参数传入，也可以通过JSON文件传入。
- 模拟SIP通话（仅SIP信令，不含媒体），如从INVITE到BYE的场景。
- 在SIP通话中响应其他SIP请求，如通过OPTIONS做通话保活的请求。
- 使用SIP MESSAGE发送IM（Instant Message，即时）消息。
- 支持彩色显示，可以更容易找到想要看的值，方便调试。
- 支持很多传输协议，如IPv4、IPv6、UDP、TCP、TLS及WebSocket（WebRTC）。
- 可发送任何类似的SIP消息，如INFO、SUBSCRIBE、NOTIFY等。

下面是一些示例仅供参考。

```
# 指定一个新的 R-URI:
sipexer -ruri sip:alice@server.com udp:127.0.0.1:5060

# 指定源端口 55060
sipexer -laddr 127.0.0.1:55060 udp:127.0.0.1:5060

# 发送注册请求，自动产生 Contact 头域，指定 Expires 为 600 秒，用户名为 alice，密码为 test123
sipexer -register -cb -ex 600 -au alice -ap test123 udp:127.0.0.1:5060

# 注册请求，Expires 为 60 秒，等待 20000 毫秒，选择注销
sipexer -register -vl 3 -co -com -ex 60 -fuser alice -cb -ap "abab..." -hal -sd
-sw 20000 udp:127.0.0.1:5060

# 设置 From-User 头域为 carol
sipexer -sd -fu "carol" udp:127.0.0.1:5060

# 同上
```

```

sipexer -sd -fv "fuser:carol" udp:127.0.0.1:5060
# From User 为 carol, To User 为 david. R-URI 中的用户名部分与 To 中的相同
sipexer -sd -fu "carol" -tu "david" -su udp:127.0.0.1:5060

# 同上
sipexer -sd -fv "fuser:carol" -fv "tuser:david" -su udp:127.0.0.1:5060

# 增加新头域
sipexer -sd -xh "X-My-Key:abcdefg" -xh "P-Info:xyzw" udp:127.0.0.1:5060

# 发送 MESSAGE 消息及 Body, 分别使用 UDP, TCP, TLS 以及 WSS
sipexer -message -mb 'Hello!' -sd -su udp:127.0.0.1:5060
sipexer -message -mb 'Hello!' -sd -su tcp:127.0.0.1:5060
sipexer -message -mb 'Hello!' -sd -su tls:127.0.0.1:5061
sipexer -message -mb 'Hello!' -sd -su wss://server.com:8443/sip

# 发送 INVITE 消息, 使用默认的 From 用户 alice 和 To 用户 bob
sipexer -invite -vl 3 -co -com -sd -su udp:server.com:5060

# alice 呼叫 bob, 使用 HAl 格式的密码验证 (具体加密字符串略), 10000 毫秒后发送 BYE, 日志级别为 3
且支持彩色
sipexer -invite -vl 3 -co -com -fuser alice -tuser bob -cb -ap "4a4a4a4a4a..." -hal -sw 10000 -sd -su udp:server.com:5060

```

命令行中的最后一个参数一般都是目标地址，可以省略。如果省略，则等效于127.0.0.1:5060。目标地址支持以下格式。

(1) SIP URI格式，举例如下。

```
sip:user@server.com:5080;transport=tls
```

(2) SIP Proxy Socket地址（proto:host:port）格式，举例如下。

```
tls:server.com:5061
```

(3) WSS URL格式，举例如下。

```
wss://server.com:8442/webrtc
```

仅有主机名或IP地址部分，则默认端口为5060，如example.com。

host:port格式，默认传输协议为UDP，如127.0.0.1:5060。

proto:host格式，默认端口为5060，如udp:127.0.0.1。

host port格式，默认传输协议为UDP，如127.0.0.1:5060。

proto host格式，默认端口为5060，如udp 127.0.0.1。

proto host port格式，同proto:host:port，如：udp 127.0.0.1:5060。

除此之外，更多的参数格式以及模板定义可以参阅项目README中的相关说明。我们也会在后面的例子中看到对sipexer工具的实际应用。值得一提的是，该工具比较新，在本书快要完成的时候才发布，因此，本书中只有少数例子使用了该工具。

5.5 Web管理界面



Siremis 是Kamailio的一个图形用户界面，主要由Asipto公司开发。Siremis使用PHP开发，可以管理Kamailio数据库、调用Kamailio RPC请求等。Siremis主要是一个管理员工具，一直在更新，功能还是挺全的，不过，整体界面风格还是很早以前的，不怎么好看。但对于初学者而言，有一个图形界面对学习有很大帮助，至少学起来不那么枯燥，所以，下面我们也简要介绍一下它的安装和使用。

注意

Siremis要用到数据库，我们会在第7章介绍数据库相关知识，所以，如果需要实验，可以先翻过去看看。其实不理解Kamailio的数据库关系也不大，比如创建一个假的Kamailio数据库也能“骗过”Siremis。但无论如何，一些基本的数据库知识（如怎么连接MySQL）还是要有的。

Siremis会连接两个数据库：一个是Siremis自己的，存储它的用户、组、权限、菜单设置等；另一个是Kamailio的数据库，里面有SIP用户数据（subscriber表）、注册数据（location表）、分发策略数据（dispatcher表）等。

为了简单，我们还是用Docker的方式安装Siremis。首先下载源代码：

```
git clone https://github.com/asipto/siremis.git
```

编译Docker镜像：

```
cd siremis/misc/docker  
docker build -t siremisdev-debian10 -f Dockerfile.debian10-gitdev .
```

编译完成后，可以直接使用如下命令启动镜像：

```
docker run --rm --name siremisdev-debian10 -p 8080:80 --network kamailio-example  
siremisdev-debian10
```

其中，我们使用了--network kamailio-example指定这个网络。如果你使用了本书附赠的代码中的示例，则这个网络可以连接Kamailio容器以及MySQL数据库容器（通过主机名kb-mysql）。

启动Siremis后，可以使用如下命令进入容器并查询其IP地址（后面也许有用），举例如下。

```
docker exec -it siremisdev-debian10 bash  
ip ad      # 这条命令在容器内执行
```

Siremis支持MySQL和PostgreSQL两种数据库，但对后者的支持不大完善，因此，在这里我们以MySQL为例。

Siremis可以自动创建数据库并导入数据，但需要使用数据库超级用户root和密码（不能没有密码，否则后面会导致奇怪的错误）连接数据库，因为普通用户没有创建数据库的权限。如何设置MySQL数据库的连接权限超出了本书的范围，不了解的读者可自行学习。如果Siremis没有创建权限，也可以手动创建数据库。例如，用root连接MySQL可执行如下命令。

```

CREATE database siremis;
CREATE user 'siremis'@'%';
GRANT ALL PRIVILEGES ON siremis.* to 'siremis'@'%' identified by 'siremis';
CREATE user 'kamailio'@'%';
GRANT ALL PRIVILEGES ON siremis.* to 'kamailio'@'%' identified by 'kamailio';
FLUSH PRIVILEGES;

```

为了安全，可以针对特定IP地址授权，举例如下。

```

CREATE user 'siremis'@'172.22.0.4';
GRANT ALL PRIVILEGES ON siremis.* to 'siremis'@'172.22.0.4' identified by 'siremis';

```

用浏览器打开http://localhost:8080/siremis，首次进入会开启设置向导，在设置向导中会提示选择数据库类型和数据库连接参数，如图5-4所示。



图5-4 安装Siremis时选择数据库连接参数

在图5-4中，我们使用了kb-mysql这个主机名连接MySQL，这是我们随书附赠代码的Docker环境中设置的名称，当然也可以改成IP地址，尤其是在连接非Docker中MySQL数据库的情况下。

另外，图5-4所示界面中有4个选项，分别如下。

- Create Siremis DB:** 决定是否让Siremis自动创建数据库。如果你所选的数据库用户具有创建数据库的权限（如root），就选择这个，否则就不选。
- Import Default Data:** 创建Siremis所需要的表和默认数据，这个选项一般都需要选，否则不知道如何创建所需的表和数据。
- Update SIP DB:** 用于更新Kamailio数据库中的acc表和与数据统计相关的表，以便符合Siremis的要求。如果你不确定是否需要更新相关内容，那么可以先不选该选项，等熟悉了再进行重装并选择。

Replace DB Config: 这个选项默认是开启的，它会根据你填的参数更新/var/www/siremis/siremi/Config.xml。后期如果你发现这个文件有错，也可以手动修改该文件。

如果一切顺利，就可以按向导进入下一步了，默认的登录用户名和密码都是admin，这两项可以在登录后修改。如果有任何错误，停掉Docker镜像，删掉数据库，然后重来即可。

登录后可以点击SIP Admin Menu选项卡进入Kamailio相关的配置，你会看到熟悉的Subscriber List、Location List、Dispatcher List等链接（见图5-5），其实它们就是对应Kamailio中相应的模块和表的管理入口。你可以尝试修改一些内容，对照数据库看看它改了哪些表里面的数据。

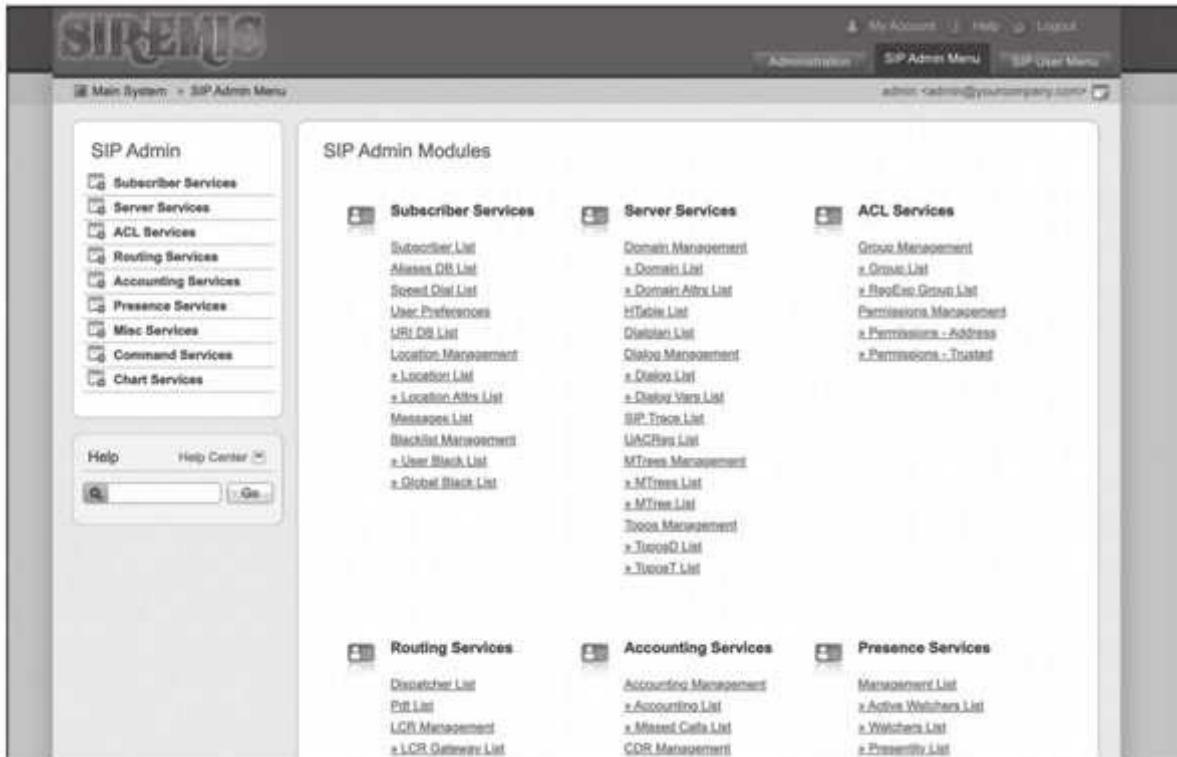


图5-5 Siremis的SIP Admin Menu页

如果在界面上通过RPC方式控制Kamailio执行一些RPC命令（比如一般修改Dispatcher List后要执行dispatcher.reload重载数据），则需要在Kamailio中做对应的配置。上面我们讲过kamctl等控制软件，这些软件都是通过本地FIFO连接的方式进行RPC控制的，但我们现在要使Siremis运行在独立的Docker容器中，这一般需要通过Socket方式连接Kamailio，而Socket方式需要考虑安全问题（你肯定不想整个互联网上的用户都能在你的Kamailio上执行RPC）。在Kamailio中启用基于HTTP的RPC服务也很简单，可参见6.5.1节相关内容。

5.6 调试与排错

在实际使用时，我们常常需要跟踪SIP消息。除了上面讲过的ngrep外，还有一些其他的实用工具。Kamailio本身也提供了一些模块。为了便于读者在学习后文的过程中跟踪调试，我们先来学习一下这些工具。

5.6.1 使用sipdump模块跟踪SIP消息

sipdump模块可以将收发的SIP消息写入本地文件或转入指定路由。写入的文件支持纯文本形式、标准的pcap包以及带有P-KSR-SIPDump头的pcap包等。

如果想将收发的SIP消息直接写入文本文件，只需启用sipdump模块，然后将mode的值设置为1即可。配置如下。

```
loadmodule "sipdump.so"
modparam("sipdump", "enable", 1)
modparam("sipdump", "mode", 1)
```

其中，mode参数有以下可能的取值。

- 1：将SIP消息写入纯文本文件，默认会写入/tmp/目录。
- 2：将SIP消息发送到事件路由，由事件路由决定如何处理。
- 4：将SIP消息写入pcap包文件。
- 8：将SIP消息插入P-KSR-SIPDump头域，并写入pcap包文件。

mode参数是一个比特位整数，也就是上述取值可以相加，如3相当于1+2，可以同时既写入本地文件又触发事件路由。

上述模式中除了事件路由形式外，其他都是将SIP消息写入文件，这些模式可以设置写入文件的路径、文件前缀以及文件分割间隔，举例如下。

```
modparam("sipdump", "folder", "/var/log/kamailio") # 设置写入路径
modparam("sipdump", "fprefix", "ksipdump-")          # 设置生成的文件名前缀
modparam("sipdump", "rotate", 3600)                   # 设置文件分割的间隔，单位是秒
# 设置清理过期文件，单位是秒。这里需要注意，此参数暂时只在 master 的分支上支持，5.5.2 版本及以前
# 版本不支持
modparam("sipdump", "fage", 172800)
```

如果使用纯文本文件，则可以在打一通电话以后查看生成的文件，也可以在打电话的过程中使用tail -f /tmp/文件进行实时跟踪。

如果使用事件路由模式，需要配置对应的事件路由，如增加以下参数配置：

```
modparam("sipdump", "event_callback", "ksr_sipdump_event") # 定义回调事件路由
```

然后在KEMI模式的Lua脚本内设置回调的事件路由，具体如下。

```
function ksr_sipdump_event(evname)
    KSR.info("from: " .. KSR.sipdump.get_src_ip() .. " to: " ..
    KSR.sipdump.get_dst_ip() ..
    " tag: " .. KSR.sipdump.get_tag() .. "\n" .. KSR.sipdump.get_buf());
end
```

上述的示例是在事件路由中收到SIP消息后便直接以日志形式输出，当然也可以写入数据库、消息队列并通过HTTP、EVAPI或其他任何协议发送到其他地方。与其他系统交互的方式我们在后文中

还会讲到，初学者直接看日志就好了。

5.6.2 其他SIP相关工具简介

除了前面介绍的工具外，还有其他一些实用工具。这些工具也各有所长，安装和使用也都比较简单，也有相关的入门使用文档。下面仅对相关工具进行罗列和简单介绍，感兴趣的读者可以自行深入了解。

- **sipgrep:** sipgrep可以类比前面介绍的ngrep，其可以对抓到的pcap包进行回放，参见<https://github.com/sipcapture/sipgrep>。
 - **Wireshark:** 老牌的抓包工具，有图形化的界面和文本抓包程序tshark。对VoIP友好，它甚至有一个专门的Telephony（电话）菜单。参见<https://www.wireshark.org/>。
 - **sngrep:** 另一个文本界面的抓包分析工具，可以在文本界面上展示SIP流程。参见<https://github.com/ironotec/sngrep>。
 - **sipsak:** 被誉为SIP的瑞士军刀，可以很方便地发送SIP消息，非常适合用于轻量级的SIP OPTIONS检测。参见<https://github.com/nils-ohlmeier/sipsak>。
 - **sipp:** SIP消息构建和压力测试工具，可以使用XML描述SIP消息模板。参见<http://sipp.sourceforge.net/>。
- 除此之外，还有很多SIP工具，这里就不一一列举了。Kamailio的作者miconda也在GitHub上维护了一些SIP资源列表，大家可以自行感受一下Kamailio作者都在用什么、关心什么，相信对学习Kamailio大有帮助，具体参见<https://github.com/miconda/sip-resources>。

Chapter 6

第6章

使用Kamailio做SIP路由转发

Kamailio最主要的作用就是转发SIP消息。根据需要，Kamailio可以做无状态和有状态的路由转发，并且可以进行并行或串行的多目的地转发，而且在转发过程中可以修改SIP消息头、SDP，以及对主、被叫号码进行号码变换等。下面先来看看什么是路由，以及一些实际路由脚本的例子。

6.1 什么是路由

对于“路由”这个词，一般人可能既熟悉又陌生。熟悉的是，基本上家家都有路由器；陌生的是，这个路由器跟我们这里说的“路由”是一个东西吗？

路由，对应的英文是Route，作为动词，即选路的意思；作为名词，对应一条选路规则，或一条路径。家用的路由器（英文称为Router）作用和我们所说的路由差不多，只不过其为上网收发IP包进行选路。本书讨论的Kamailio，可以认为是一个SIP路由器。

简单来说，Kamailio的路由就是控制SIP包从哪里来、到哪里去。下面抛开技术细节，通过一个日常生活中的例子，来了解路由相关的名词术语。

如图6-1所示，假设有A、B、C、D四座城市，C（英文正好为Center）位于中心，A、B、D间互访都需要经过C。C就对应我们这里所说的Kamailio服务器。

各城市之间的传输路径就称为中继。中继是有方向的。在本例中，中继是“双向”的，即A能到C，C也能到A，其他亦然。但假设某天A发生了疫情，C规定，所有从A来的人都不准进入C，但C的人仍可去往A，这相当于路径CA是通的，但AC不通，中继就变成了单向的。中继的方向是相对的，CA相对A来说，就是“单入”中继，相对C来说，就是“单出”中继。

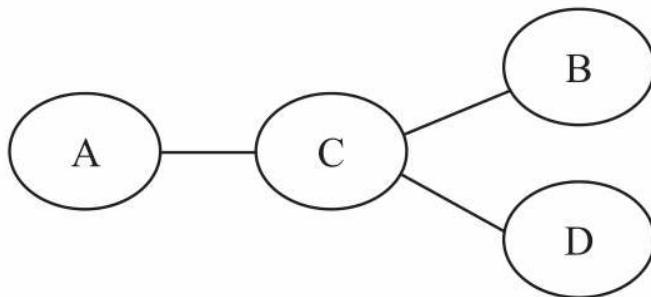


图6-1 路由组网示意图

假设有人从A出发经C去往B，但从C出发有两条路（CB和CD），不知道怎么走，就要找人“问路”，这个“问路”和“回答”的过程就称为“查找路由”，即“选路”。一旦选择了正确的路线，就可以继续前往“目的地”（Destination，简称Dest），当然，在Kamailio中，“去往目的地”对应的是CB这条中继，B也称为C的“下一跳”（Next Hop）。

从C到B可能有多种交通方式，如高铁、驾车、飞机等，可以认为是多条中继，多条中继组成一个中继组。假设有100个人同时从C到B，可能有20个人选择乘飞机，70个人选择坐高铁，10个人选择自驾，这些不同的选择合在一起称为中继组的分配策略，分配策略的前提是不同的中继有不同的容量和费用。如果A发生地震，所有人都经C到其他城市避难，这时候可能因为AC间的路不够宽（或高铁、飞机的班次不够多，对应到Kamailio就是中继资源不够用）而发生“拥塞”。

如果有人从A出发经C到B，本想乘飞机，但由于天气原因航班取消（中继故障），不得以改乘高铁，这就称为重选路由，这种重选是“串行”（Serial）的。

假设A急需一种药品，B或D都有可能提供这种药品，派出一人到达C市后，通信中断，经多方打听仍不确认在B还是D能找到该药品，只能人到了才知道。为了增加成功率，C也派出一人帮忙，A的人去B，C的人去D，这样不管B还是D有药品都能成功被找到，这称为“并行”（Parallel）转发。

政策发生变化，C要在每条路的出入两个方向上都设置检查站并收费。如果对进入C的人收费，称为“入中继计费”，反之，称为“出中继计费”。

当然在上面提到A发生疫情的情况，A的人无法经过C到B，但D的人可以经C到B。所以，在C上，对于到同一个目的地（这里是B）的问路请求，还要检查这个人是从哪个城市来的，以确定是否准许通过，这个“来源”的城市就称为“呼叫源”（Source）。所以，呼叫源也是路由的一部分。

假设甲乙两个人都来自A，且要经C去往B，而从C到B有多种交通方式。两人到达C后，仅根据“呼叫源”和“目的地”无法区分甲乙两人后续行程，但甲买了飞机票，乙买了高铁票，到C后就可以通过不同的中继路由到B。所以说甲乙预先买的票决定了后续行程，甲乙购买的票就称为“路由码”。在Kamailio中，路由码可以在SIP头域中传送，也可以在主、被叫号码中传送，相当于C把路由选择的部分权利开放给了A。

如果很不幸，C发生了疫情，则A、B、D之间的交通就中断了。所以，在此，C属于一个“单点故障”点，这时要有备用的方案。一般采用两个中心城市的设计，如图6-2所示的C1和C2。这在Kamailio网关上称为“双平面”。这种架构下，C1或C2其中一个城市发生疫情不影响A与B、D之间的交通（通信）。

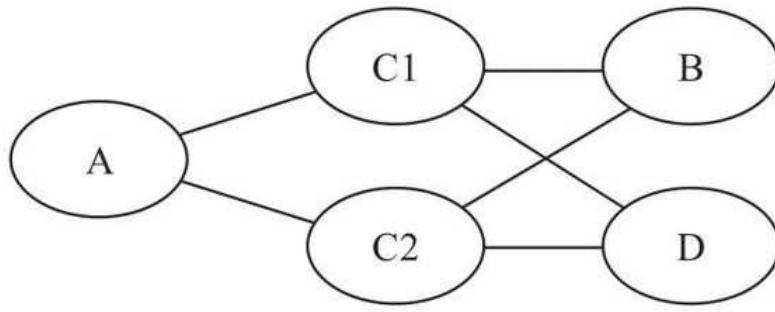


图6-2 双平面架构示意图

再回到图6-1所示情形，如果把C换成Kamailio网关，A、B、D换成电话交换机，把城市里的“人”换成“电话”，那么，在Kamailio (C) 的中继AC上来了一通电话，这通电话是A（呼叫源）打来的，主叫号码是A，被叫号码是B，C查找本地的路由表，发现一条路由AB，目的地是CB这条中继，然后将通话发到CB中继上，电话到达B端的交换机，这便是Kamailio进行路由查找和转发的过程。

6.2 基本路由转发

“千里之行，始于足下”，本节先从最基本的基本路由转发脚本开始，一步一步地带大家领略Kamailio的强大魅力。

6.2.1 最简单、最安全的路由转发

最简单的转发就是不转发。越简单，越安全，使用如下的Lua脚本，可以解决一切安全问题。

```
examples/simple-secure.lua:
function ksr_request_route()
    KSR.x.drop();
end

function ksr_reply_route()
    KSR.x.drop();
end
```

重启Kamailio，如果你打开ngrep，可以看到，不管客户端怎么疯狂注册或呼叫Kamailio，它都没有任何响应。“大音希声”，沉默就是最有力的反抗。`drop`函数的作用是丢弃收到的SIP消息且不做任何响应。代码胜过千言万语，经过前面的学习，这块代码已经不需要解释了。

6.2.2 无状态转发

可以直接使用如下函数将SIP呼叫请求转发到FreeSWITCH。

```
stateless_forward.lua:
function ksr_request_route()
    ksr_register_always_ok()
    -- KSR.forward_uri("sip:172.18.0.3:5080")
    KSR.forward_uri("sip:kb-fs1:5080")
end
```

也可以使用如下方式（`examples/stateless_forward2.lua`）将SIP呼叫请求转发到FreeSWITCH。

```
function ksr_request_route()
    ksr_register_always_ok()
    KSR.pv.sets("$du", "sip:kb-fs1:5080")
    KSR.forward()
end
```

`KSR.forward()`函数没有参数，会查找\$du这个伪变量进行路由，如果\$du不存在，则会转发到\$ru。

这里，`KSR.forward()`和`KSR.forward_uri()`两个函数都对应原生脚本中的`forward(...)`函数（参见4.2节）。

6.2.3 有状态转发

使用如下脚本做有状态转发。

```
stateful-relay.lua:
function ksr_request_route()
    ksr_register_always_ok()

    KSR.rr.record_route();
    KSR.pv.sets("$du", FS1_URI)
    KSR.tm.t_relay()
end
```

t_relay()函数也有以下两种变体。

```
function ksr_request_route()
    ksr_register_always_ok()
    KSR.tm.t_relay_to_proxy(FS1_UDP)
    -- KSR.tm.t_relay_to_proxy_flags(FS1_UDP, 0)
end
```

注意，t_relay_to_proxy(str proxy)的参数是一个Socket地址字符串而不是URI，格式是protocol:ip:port；而t_relay_to_proxy_flags(str proxy, int flags)多了一个flags参数，它是一个二进制位标志的整数，取值有如下几个。

□ 0x1：不发送100 Trying回复消息。

□ 0x2：在内部错误时不发送回复消息。

□ 0x4：禁用DNS Failover。

当然，取值可以是上述值的任意组合，如 $0x3 = 0x1|0x2$ （对十六进制数进行二进制“或”操作，也可以认为是相加），结果即二进制0B00000011。

从上面可以看出，消息转发的本质是设置\$du并通过forward及relay相关的函数把SIP消息以无状态或有状态的方式转发出去。读者可以自行执行上述脚本并对比两者产生的SIP消息的异同。

另外，从上面的语法中也可以看出，这种转发可以进行协议转换。比如进来的消息是UDP，可以转换成TCP，此时只需要将那个protocol参数设成TCP即可。如果设置\$du，则可以在后面加上“;transport=tcp”参数以将传输协议设为TCP。

6.2.4 并行转发

并行转发即Parallel Forking，指将SIP消息裂变成两个或多个（称为多个分支），分别转发到不同的目的地，首先响应的目的地将获胜，并继续进行SIP交互，而响应慢的或根本不响应的目的地将被放弃。我们看下面的示例代码。

```
forking-parallel.lua:
function ksr_request_route()
    ksr_register_always_ok()
    KSR.corex.append_branch_uri('sip:10000486@' .. FS1_IP_PORT)
    KSR.corex.append_branch_uri('sip:10000200@' .. FS1_IP_PORT)
    KSR.pv.sets("$du", FS1_URI)
    KSR.tm.t_relay()
end
```

在上述代码中，我们增加了两个并行的分支（branch）呼叫9196，用ngrep观察SIP消息。我们可以看到，SIP客户端（来自172.18.0.1）发来的消息（见包#1）被Kamailio（172.18.0.2）转发后变成3个INVITE（3个不同的分支，见包#3、#4、#5中Via头域中的“branch=”参数），其中两个被FreeSWITCH（172.18.0.3）回复了482 Merged（见包#6、#8），一个被正常应答（见包#11），具体如下。

U 172.18.0.1:59926 -> 172.18.0.2:5060 #1
INVITE sip:9196@172.18.0.2 SIP/2.0.
Via: SIP/2.0/UDP 172.18.0.1:59458;rport;branch=z9hG4bKPj1D9dfFjR2VoKI4E5boggGLiF93CG8Jzpo.

U 172.18.0.2:5060 -> 172.18.0.1:59926 #2
SIP/2.0 100 trying -- your call is important to us.
Via: SIP/2.0/UDP 172.18.0.1:59458;rport=59926;branch=z9hG4bKPj1D9dfFjR2VoKI4E5boggGLiF93CG8Jzpo;received=172.18.0.1.

U 172.18.0.2:5060 -> 172.18.0.3:5080 #3
INVITE sip:9196@172.18.0.2 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534eb3db.0.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59926;branch=z9hG4bKPj1D9dfFjR2VoKI4E5boggGLiF93CG8Jzpo.

U 172.18.0.2:5060 -> 172.18.0.3:5080 #4
INVITE sip:10000486@kb-fs1:5080 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534eb3db.1.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59926;branch=z9hG4bKPj1D9dfFjR2VoKI4E5boggGLiF93CG8Jzpo.

U 172.18.0.2:5060 -> 172.18.0.3:5080 #5
INVITE sip:10000200@kb-fs1:5080 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534eb3db.2.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59926;branch=z9hG4bKPj1D9dfFjR2VoKI4E5boggGLiF93CG8Jzpo.

U 172.18.0.3:5080 -> 172.18.0.2:5060 #6
SIP/2.0 482 Request merged.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534e

```
b3db.l;received=172.18.0.2.  
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59926;branch=z9hG4bKP  
j1D9dffFjR2VoK14E5boggGLiF93CG8Jzpo.
```

```
U 172.18.0.2:5060 -> 172.18.0.3:5080 #7  
ACK sip:10000486@kb-fs1:5080 SIP/2.0.  
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534e  
b3db.l.
```

```
U 172.18.0.3:5080 -> 172.18.0.2:5060 #8  
SIP/2.0 482 Request merged.  
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534e  
b3db.2;received=172.18.0.2.  
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59926;branch=z9hG4bKP  
j1D9dffFjR2VoK14E5boggGLiF93CG8Jzpo.
```

```
U 172.18.0.2:5060 -> 172.18.0.3:5080 #9  
ACK sip:10000200@kb-fs1:5080 SIP/2.0.  
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534e  
b3db.2.  
Max-Forwards: 70.
```

```
U 172.18.0.3:5080 -> 172.18.0.2:5060 #10  
SIP/2.0 100 Trying.  
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534e  
b3db.0;received=172.18.0.2.
```

```
U 172.18.0.3:5080 -> 172.18.0.2:5060 #11  
SIP/2.0 200 OK.  
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK499b.adb337a77624a1d5653d993d534e  
b3db.0;received=172.18.0.2.  
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59926;branch=z9hG4bKP  
j1D9dffFjR2VoK14E5boggGLiF93CG8Jzpo.
```

FreeSWITCH回复482的原因是它不能处理同一个Call-ID有多个并行分支的情况。你可以试着将呼叫并行发到多个不同的FreeSWITCH，在实际使用时一般也会发到多个不同的IP地址。

值得一提的是，FreeSWITCH也支持类似的并行呼叫，但不是有多个分支，而是有多个独立的呼叫。在FreeSWITCH中的用法如下。

```
<action application="bridge" data="user/1000,user/1001,user/1002"/>
```

在上面的情况下，FreeSWITCH会向3个分机同时发送INVITE消息，其中一个接听，另外两个会自动挂断（FreeSWITCH发CANCEL消息）。

当然，FreeSWITCH也支持串行呼叫，如果第一个不接则呼叫第二个，如果第二个也不接则呼叫第三个，举例如下。

```
<action application="bridge" data="user/1000|user/1001|user/1002"/>
```

提示



Kamailio是一个Proxy，而FreeSWITCH是一个B2BUA，这两者都可以称为Fork。通过对比FreeSWITCH消息就可以看出，在前者产生的SIP消息中，Call-ID都是一样的，只是分支不同；而后者产生的SIP消息中，Call-ID是不同的。这便是Kamailio与FreeSWITCH对Fork处理的最大不同之

处。

6.2.5 串行转发

串行转发即Serial Forking，也叫Fallback（转移到备用方案）或Failover（故障转移），也就是在第一个呼不通的情况下呼叫下一个。看下面的代码，我们通过t_on_failure()设置一个回调函数ksr_failure_manage，当呼叫失败时，它将执行该函数。这里我们重新修改目的地（改变被叫号码或IP地址，或两者都改）。

```
forking-serial.lua:
function ksr_request_route()
    ksr_register_always_ok()

    if KSR.is_INVITE() then
        if KSR.tm.t_is_set("failure_route")<0 then
            KSR.tm.t_on_failure("ksr_failure_manage");
        end
    end

    KSR.pv.sets("$du", FS1_URI)
    KSR.tm.t_relay()
end

function ksr_failure_manage()
    if KSR.tm.t_is_canceled()>0 then
        return 1
    end

    local status_code = KSR.tm.t_get_status_code()
    KSR.warn("call failed with status=" .. status_code .. " retrying ...\\n")

    -- KSR.cfgutils.sleep(8)
    KSR.pv.sets("$tU", '9196')
    KSR.pv.sets("$rU", '9196')
    KSR.pv.sets("$du", FS1_URI)
    KSR.tm.t_relay()
end
```



注意，由于目的地是FreeSWITCH，而FreeSWITCH对Fork的呼叫支持不好，因此，FreeSWITCH会返回482 Merged消息。不过，不管FreeSWITCH返回什么，从SIP消息角度看，我们的Fork都是成功了。如果要测试功能的Fork，可以取消KSR.cfgutils.sleep(8)这一行的注释。



KSR.cfgutils.sleep(8)表示等8秒以后再重试。在生产环境中不要这么用，因为它将阻塞Lua脚本，这里我们仅为演示Fork功能，所以才会这样用。实际使用时一般会Fork到不同的IP地址，也就不会有这样的问题了。

串行的Fork也是使用不同的分支。呼叫10000404将返回404 Not Found（见包#6），然后使用新的分支重发INVITE（见包#8），信令流程如下，读者可以仔细阅读对比它与并行Fork的异同。

U 172.18.0.1:59634 -> 172.18.0.2:5060 #2
INVITE sip:10000404@172.18.0.2 SIP/2.0.
Via: SIP/2.0/UDP 172.18.0.1:59458;rport;branch=z9hG4bKPjpyL0JX4dV7FosiO.X3DtQbpTheCnm6J.
Max-Forwards: 70.

U 172.18.0.2:5060 -> 172.18.0.1:59634 #3
SIP/2.0 100 trying -- your call is important to us.
Via: SIP/2.0/UDP 172.18.0.1:59458;rport=59634;branch=z9hG4bKPjpyL0JX4dV7FosiO.X3DtQbpTheCnm6J;received=172.18.0.1.

U 172.18.0.2:5060 -> 172.18.0.3:5080 #4
INVITE sip:10000404@172.18.0.2 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2eb11.0.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59634;branch=z9hG4bKPjpyL0JX4dV7FosiO.X3DtQbpTheCnm6J.

U 172.18.0.3:5080 -> 172.18.0.2:5060 #5
SIP/2.0 100 Trying.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2eb11.0;received=172.18.0.2.

U 172.18.0.3:5080 -> 172.18.0.2:5060 #6
SIP/2.0 404 Not Found.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2eb11.0;received=172.18.0.2.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59634;branch=z9hG4bKPjpyL0JX4dV7FosiO.X3DtQbpTheCnm6J.

U 172.18.0.2:5060 -> 172.18.0.3:5080 #7
ACK sip:10000404@172.18.0.2 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2eb11.0.

U 172.18.0.2:5060 -> 172.18.0.3:5080 #8
INVITE sip:9196@172.18.0.2 SIP/2.0.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2

eb11.1.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59634;branch=z9hG4bK
jpyL0JX4dV7FosiO.X3DtQbpThepCnm6J.
Max-Forwards: 70.

U 172.18.0.3:5080 -> 172.18.0.2:5060 #9
SIP/2.0 100 Trying.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2:
eb11.1;received=172.18.0.2.
From: "Seven Du" <sip:1001@172.18.0.2>;tag=dgChP-wnoEYxwLXmJoDH8t1PiSPwJa9p.
To: sip:9196@172.18.0.2.
Call-ID: 55CUfNzHZntbKTpJdBcpjqmJQ4cbCURz.
CSeq: 20743 INVITE.
User-Agent: FreeSWITCH-mod_sofia/1.10.7-dev+git~20210727T022117Z-f03d765022~64bit.
Content-Length: 0.

U 172.18.0.3:5080 -> 172.18.0.2:5060 #10
SIP/2.0 200 OK.
Via: SIP/2.0/UDP 192.168.7.7:5060;branch=z9hG4bK01d2.94945a65c65187e9c9f16f79f6d2:
eb11.1;received=172.18.0.2.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59634;branch=z9hG4bK
jpyL0JX4dV7FosiO.X3DtQbpThepCnm6J.

U 172.18.0.2:5060 -> 172.18.0.1:59634 #11
SIP/2.0 200 OK.
Via: SIP/2.0/UDP 172.18.0.1:59458;received=172.18.0.1;rport=59634;branch=z9hG4bK
jpyL0JX4dV7FosiO.X3DtQbpThepCnm6J.

6.3 使用 dispatcher模块做路由转发和负载均衡

dispatcher应该是Kamailio中最常用的模块，它有很多负载均衡算法，也支持并行和串行的Fork。

6.3.1 基本用法

使用前，我们在book.cfg中加入以下配置。

```
loadmodule "dispatcher.so"

modparam("dispatcher", "list_file", "/etc/kamailio/dispatcher.list")
modparam("dispatcher", "ds_probing_mode", 3)
modparam("dispatcher", "flags", 2)
modparam("dispatcher", "ds_probing_threshold", 3)
modparam("dispatcher", "ds_ping_interval", 5)
modparam("dispatcher", "ds_ping_reply_codes",
"class=2;code=403;code=488;class=3")
modparam("dispatcher", "xavp_dst", "_dsdst_")
modparam("dispatcher", "xavp_ctx", "_dsctx_")
```

首先，我们需要一个list_file（列表文档）参数，其他参数都是可选的。对于其他参数，我们留到后面再讲。dispatcher.list的内容如下。

```
100 sip:kb-fs1:5080
200 sip:rts.xswitch.cn:20003;transport=tcp
300 sip:kb-fs1:5080
300 sip:rts.xswitch.cn:20003;transport=tcp
```

list_file的第一列是组号，后面跟的是一个URI，表示转发的地址。在此，我们设置了3个组，其中第3个组里有2个URI。为方便起见，列表中的每一行称为一个中继。

重新启动Kamailio后，我们可以使用kamcmd dispatcher.list命令查看内存中的情况，具体如下。

```

{
    NRSETS: 3
    RECORDS: {
        SET: {
            ID: 100
            TARGETS: {
                DEST: {
                    URI: sip:kb-fs1:5080
                    FLAGS: AX
                    PRIORITY: 0
                }
            }
        }
        SET: {
            ID: 300
            TARGETS: {
                DEST: {
                    URI: sip:rts.xswitch.cn:20003;transport=tcp
                    FLAGS: AX
                    PRIORITY: 0
                }
                DEST: {
                    URI: sip:kb-fs1:5080
                    FLAGS: AX
                    PRIORITY: 0
                }
            }
        }
        SET: {
            ID: 200
            TARGETS: {
                DEST: {
                    URI: sip:rts.xswitch.cn:20003;transport=tcp
                    FLAGS: AX
                    PRIORITY: 0
                }
            }
        }
    }
}

```

呼叫10000200，可以将呼叫转发到fs1上。dispatcher.lua的内容如下。

```

function ksr_request_route()
    ksr_register_always_ok()

    if KSR.is_INVITE() then
        if KSR.tm.t_is_set("failure_route") < 0 then
            KSR.tm.t_on_failure("ksr_failure_manage");
        end
    end

    if KSR.dispatcher.ds_select_dst(100, 4) < 0 then
        KSR.sl.send_reply(404, "No destination")
        KSR.x.exit()
    else
        KSR.tm.t_relay()
    end
end

```

其中ds_select_dst(str group, int algorithm)用于在组中选择一条中继并将SIP消息通过它发出去。group即我们在list_file中写的组号，algorithm是选择的算法，上述代码中将此参数设为4，这代表轮循（Round Robin）。其他参数我们后面会讲到。

如果把组号换成300，多打几个电话，就可以看到电话将在两个目的地间平均分配。

注意，由于我们没有判断呼叫的来源（我们认为所有SIP请求都来自SIP客户端），因而，如果FreeSWITCH主动发送了BYE消息，我们还是会路由到FreeSWITCH，这就造成了循环，导致挂不了机。真正的脚本不仅需要正确处理BYE消息，还应该考虑呼叫来源。用ds_is_from_list()函数就可以对呼叫来源进行判断。在此我们先假设所有请求都是来自SIP客户端（BYE消息也是由客户端先发送）。对于呼叫来源的判断将在6.4节中讲到。

6.3.2 dispatcher模块

dispatcher模块实现了以下分配策略（其中开头的数字为分配策略的ID，这里特意没有按从小到大排序）。

(1) **4—轮循：** 呼叫将在多个中继间轮循选取，如第一个呼叫走第一条中继，下一个呼叫走第二条……

(2) **9—百分比：** 根据设定的百分比进行路由。必须设置权重字段。权重字段加起来必须等于100，如果不到100，则最后一条中继（根据中继ID排序）将使用剩余的权重。如只有两个中继，我们将其权重分别设置为20、20，则实际上第二条中继的权重为80。

(3) **11—按相对比例分配：** 根据设定的相对比例进行路由。必须设置rweight参数，其值大于0并且小于等于100。比如有两个中继，第一个权重设置为1，第二个权重设置为4，那么分配到第一个中继的概率是 $1/(1+4)=20\%$ ，分配到第二个中继的概率是 $4/(1+4)=80\%$ 。

(4) **8—根据优先级分配：** 选择优先级最高的中继，如果呼叫失败，则会走次优先级的中继（顺序尝试，Serial Forking）。

(5) **10—根据负载分配：** 该模式比较复杂，使用时有诸多限制，需要进行特殊配置，因而除非特殊情况否则不建议使用。

使用该模式必须进行特殊配置，且一旦完成配置就无法再更改了（除非重启）。

必须设置duid和maxload字段，比如duid=fs1;maxload=300。duid是destination unique id的缩写，每个中继的duid必须具有唯一性。

只处理INVITE请求，不处理PUBLISH请求等。

要配置ds_hash_size等模块参数，比如modparam("dispatcher", "ds_hash_size", 9)以便记录中继的当前负载。

在收到BYE或CANCEL请求时要调用函数ds_load_update，在收到2[0-9][0-9]回应时要调用ds_load_update函数，在收到3|4|5|6失败回应时要调用ds_load_unset函数。

超过最大负载之后怎么处理？可以将模块参数use_default设置为1，即把最后一个中继作为最终选项，这就等于提供了一个候补方案。

(6) **6—随机分配：** 将呼叫随机分配到所有状态是Active的中继上，但机会不一定均等。

(7) **0—根据Call-ID分配：** 根据Call-ID计算出hash， $\text{hash} = \text{hash} \% \text{nr}$ ，最后计算出来的就是选中的中继。 nr 是number of items in dst set的缩写，即总中继数。

(8) **1—根据From URI分配：** 根据From URI计算出hash，其他跟0一样。适合的场景是，在主叫号码(From)相同的情况下，呼叫请求总是分配到同一个中继。

(9) **2—根据To URI分配：** 根据To URI计算出hash，其他跟0一样。适合的场景是，在被叫号码相同的情况下，呼叫请求分配到同一个中继。

(10) **3—根据Request URI分配：** 根据Request URI计算出hash，其他跟0一样。

(11) **5—根据Username分配：** 根据认证用户名来计算hash，如果认证用户名不存在，那么就改

用轮询（Round Robin）的方式。该策略用于希望同一个注册用户总是分配到同一个中继的场景。

(12) 7—根据PV字符串分配：需要配置模块参数hash_pvar，比如modparam("dispatcher", "hash_pvar", "\$fU")。在下面所示这个例子中，根据From User计算hash，也可以配置成PV字符串的组合，比如modparam("dispatcher", "hash_pvar", "\$fU@\$ci")，还可以配置成自定义伪变量，比如modparam("dispatcher", "hash_pvar", "\$var(myhash)")。

```
$var{myhash} = "1234"; # 在调用 ds_select_dst() 之前给自定义伪变量赋值
if(!ds_select_dst("1", "7")) {
    send_reply("404", "No destination");
    exit;
}
```

(13) 13—延迟优化：该策略的英文名为Latency Optimized。该策略会根据Ping检测到的延迟调整中继的优先级，并根据调整后的优先级对中继进行排序，在多个最高优先级（优先级相同）的中继间使用轮循算法。该策略需要设置权重（内部会映射成优先级字段priority）。内部在属性(attributes)里面设置cc=1字段，cc (Congestion Control) 是拥塞控制的意思。

如果cc = 0，那么计算公式为：

```
ADJUSTED_PRIORITY = PRIORITY - (ESTIMATED_LATENCY_MS/PRIORITY)
```

如果cc=1，那么计算公式为：

```
CONGESTION_MS = CURRENT_LATENCY_MS - NORMAL_CONDITION_LATENCY_MS
ADJUSTED_PRIORITY = PRIORITY - (CONGESTION_MS/PRIORITY)
```

如表6-1所示，系统将呼叫只分给最高优先级（即30）的中继。

表6-1 延迟优化中继优先级表

中继号	优先级	估计延迟	调整后的优先级	实际分配比例
1	30	21	30	33%
2	30	91	27	0%
3	30	61	28	0%
4	30	19	30	33%
5	30	32	29	0%
6	30	0	30	33%
7	30	201	24	0%

如果ds_ping_latency_stats有效，则该策略会自动调整中继的优先级：每当Ping检测到的延迟毫秒数等于优先级的值时，优先级减1，也就是说，检测到比较大的延迟则降低优先级。

(14) 12—并行：有多个分支，也就是同时呼叫这个组里面所有的中继。如果其中一个接听，则其他的都会挂掉。

6.3.3 优先级路由及备用路由

我们前面用到一个简单的dispatcher.list，它完整的格式如下（字段间以空格分隔）。

```
组号 (int) 目的地 (sip uri) flags(int) 优先级 (int) 其他参数 (str)
```

示例数据如下。

```

300 sip:kb-fs1:5080 0 10
300 sip:rts.xswitch.cn:20003;transport=tcp 0 0

```

由于上述第一个中继优先级高，所以如果根据dispatcher模块设置的分配策略是8（见6.3.2节），则优先走第一个，失败时可以走下一个。需要在dispatcher.lua中加入以下代码。

```

function ksr_failure_manage() -- 呼叫失败后执行该函数
    if KSR.tm.t_is_canceled() > 0 then return 1 end -- 如果主叫挂机，就返回
    local status_code = KSR.tm.t_get_status_code() -- 获取失败消息的状态码
    KSR.warn("call failed with status=" .. status_code .. " retrying ...\\n")
    -- 只有状态码在该范围内，并且分支有超时设置且该分支收到回复时才进入下面代码块
    if KSR.tm.t_check_status("[4-5][0-9][0-9]") or
        (KSR.tm.t_branch_timeout() > 0 and KSR.tm.t_branch_replied() < 0) then
        local ret = KSR.dispatcher.ds_next_dst() -- 从中继分配表中选择下一个可用的中继
        if ret > 0 then -- 如果还有可用中继
            dst = KSR.pv.gete('$xavp(_dsdst_0)=>uri') -- 找到下一个中继，以便打印日志
            KSR.notice("dispatch FAILED, trying next " .. dst) -- 打印日志，方便调试
            KSR.info("--- SCRIPT: going to <" .. KSR.pv.get("$ru") .. "> via <" ..
                KSR.pv.get("$du") .. ">")
            KSR.tm.t_relay() -- 发送。之前ds_next_dst()函数会改变目标地址
        else -- 如果没有可用中继就打印错误日志
            KSR.notice("dispatch FAILED, no more dst to try")
        end
    end
    KSR.x.exit()
end

```

由于dispatcher模块使用了modparam("dispatcher", "xavp_dst", "_dsdst_")参数，当调用ds_select_dst()时，中继信息将存到_dsdst_这个AVP中。如果呼叫失败，则可以从AVP中查看是否有下一个中继，然后进行路由。或者，如上面的代码那样，直接调用ds_next_dst()获取下一个可用中继，如果有，就继续转发。这跟上面讲过的串行转发类似。

6.3.4 按权重路由

dispatcher模块实现了weight和rweight两种权重方式，这两种方式略有不同（具体见6.3.5节）。这两种方式均需要在目的地集中加入权重参数，具体如下。

```

300 sip:kb-fs1:5080 0 0 weight=80;rweight=80;maxload=20
300 sip:rts.xswitch.cn:20003;transport=tcp 0 0 weight=20;rweight=20

```

要想使上述设置生效，需要使用kamcmd dispatcher.reload或者kamctl dispatcher reload命令重载数据集，也可以直接重启Kamailio。再次测试可以发现，权重越高分到的电话就越多。

6.3.5 特殊参数

上一节中讲的weight、rweight等都是中继数据的特殊参数，本节就来讲讲这些特殊参数。Kamailio中涉及的中继数据特殊参数如下。

- duid：一个唯一标志，有的算法需要这个标志。
- maxload：用于并发控制，必须是正整数，如果这个中继上的并发数达到该值，则不会再选到这个中继，直到它上面的一个呼叫挂掉。如果是0则表示该中继无效。
- weight：用于权重相关的算法，必须是0~100，所有中继中的weight加起来应该等于100。
- rweight：相对权重，必须是1~100。
- socket：决定通过哪个socket发SIP消息，也用于发送OPTIONS请求。

6.3.6 从数据库中加载

dispatcher.list中的数据也可以从数据库中加载。需要设置db_url参数以连接一个数据库表，默认的数据库表名是dispatcher。这里所说的数据库表可以是真正的数据库表也可以是一个视图，可以用table_name修改对应的表名。

注

表结构如下（以PostgreSQL为例，来自源代码目录下的utils/kamctl/postgres/dispatcher-create.sql）。

```
CREATE TABLE dispatcher (
    id SERIAL PRIMARY KEY NOT NULL,
    setid INTEGER DEFAULT 0 NOT NULL,
    destination VARCHAR(192) DEFAULT '' NOT NULL,
    flags INTEGER DEFAULT 0 NOT NULL,
    priority INTEGER DEFAULT 0 NOT NULL,
    attrs VARCHAR(128) DEFAULT '' NOT NULL,
    description VARCHAR(64) DEFAULT '' NOT NULL
);
```

往表中填入相应数据，内容跟dispatcher.list类似，然后调用kamcmd dispatcher.reload即可完成数据的加载工作。

dispatcher模块功能很强大，其中包含的各种算法也基本能满足实际使用的要求。通过该模块可将数据一次性加载到内存中，在实际路由时无须再查询数据库，因而非常高效。

dispatcher模块中还有很多参数，限于篇幅这里就不一一细讲了。更多细节请参阅模块的说明文档：<https://www.kamailio.net/docs/modules-devel/modules/dispatcher.html>。

6.4 呼叫从哪里来

我们看一个典型的场景，如图6-3所示：PSTN来话被分配到K1上，集群去话也由K1路由。用户SIP话机注册到K2上，打电话时经过FreeSWITCH再通过K1发往PSTN。

对于K1而言，需要判断电话是从PSTN网关来的还是从FreeSWITCH来的。

对于K2而言，需要判断注册或呼叫信息是来自SIP话机还是来自FreeSWITCH。

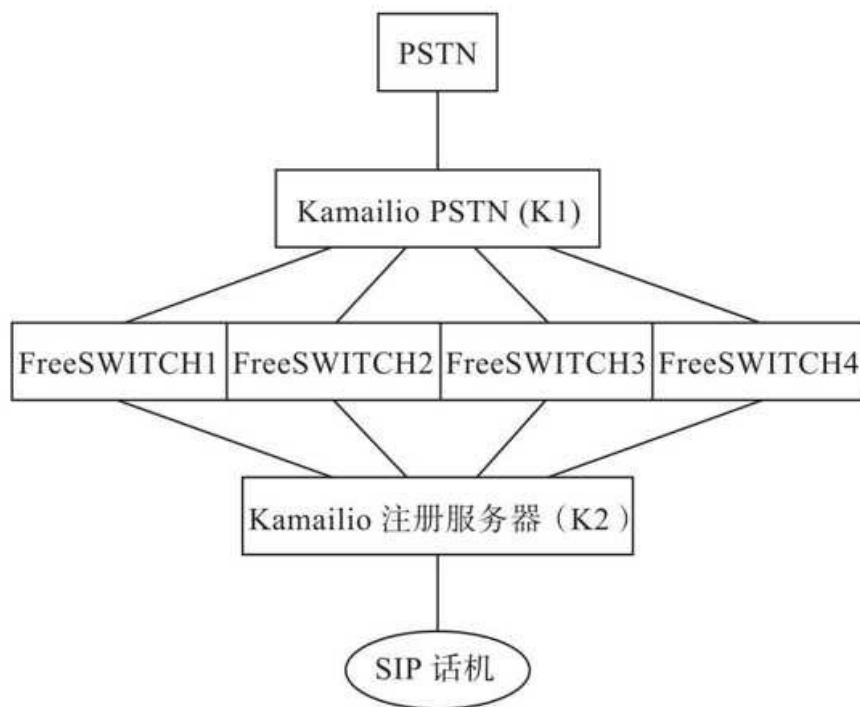


图6-3 呼叫逻辑关系示意图

当然，我们可以将K1和K2合二为一，那样的话，我们的Kamailio路由脚本就需要判断4个方向的信息。在实际使用时，为保持路由脚本简洁，我们建议“专K专用”。

为了判断呼叫从哪里来，我们需要一些策略和工具，还需要了解实际的场景和网络拓扑。在最简单的场景下，PSTN的IP地址是已知的，而且不会有多个，本地SIP电话的IP地址段是固定的，FreeSWITCH的地址也是固定的，因而，可以根据IP地址段判断呼叫从哪里来。但在更复杂的场景中，比如K2上的用户在公网上，IP地址就可能是所有国内的IP地址，甚至是全球的IP地址，且包括IPv4和IPv6两种。这样判断起来就会复杂得多。

6.4.1 根据IP地址段判断

先来看以下的示例代码。

```

FS_IPs = {
    "10.10.0.1/24",
    "172.17.0.2/32",
    "172.22.0.3/32",
}

function is_from_fs_ip()
    local srcaddr = KSR.pv.gete("$si");

    for idx, val in pairs(FS_IPs) do
        if KSR.ipops.ip_is_in_subnet(srcaddr, val) > 0 then
            return true;
        end
    end
    return false;
end

```

其中，`ip_is_in_subnet()`可以判断IP地址是否属于某个IP地址段，`FS_IPs`中的/24表示掩码的位数，这



是CIDR（无类别域间路由）表示的方法。通过上述函数就可以判断呼叫的来源了。但这种方式需要将IP地址信息静态写到Lua脚本中。当然，也可以将`FS_IPs`放到单独的Lua配置文件脚本中用`dofile()`读进来，以后要修改IP地址只需要修改Lua配置文件。

6.4.2 使用**dispatcher**模块判断

如果已经使用了**dispatcher**做负载均衡分发，那就说明我们已经知道一些IP地址段和目标主机的对应关系了。当这些主机发起呼叫时，我们就可以根据已知的信息判断呼叫来源是否在我们的IP地址段中。比如，可以使用`KSR.dispatcher.ds_is_from_list()`判断IP地址是否属于某一个组。

`ds_is_from_list`语法如下。

```
ds_is_from_list([groupid [, mode [, uri] ]])
```

如果来源IP地址或URI与对应的组中的地址匹配，则返回true（在Lua中返回大于0的值），否则返回false。相关参数说明如下。

□ **groupid**: 可选参数，其输入值可以是一个整数，还可以是一个值为整数的变量。如果该参数的值不存在，则会在所有组的所有地址中尝试匹配；如果该参数有值，则仅在指定的组中匹配。

□ **mode**: 可选参数（如果指定第三个参数，则该参数是必选参数），用于指定匹配算法，它是一个比特位整数，具体含义如下。

- 0: 所有的IP地址、端口、协议必须匹配。
- 1: 忽略端口。
- 2: 忽略协议。
- 3: 忽略端口和协议（3 = 0b11）。

□ **uri**: 可选参数。如果该参数为空，则表示使用来源IP地址、端口和协议；否则其必须是一个合法的SIP URI（但仅匹配IP地址、端口和协议，其他参考都会忽略）。该参数可以是一个静态或动态字符串（变量），URI的domain部分可以是IP地址也可以是主机名。

如果`ds_is_from_list`函数返回true（地址匹配），则`setid_pvname`指定的参数将会被赋值为相应的`groupid`，其他参数将赋值到`attrs_pvname`指定的变量中。

注意，出于向后兼容的考虑，如果没有任何参数，或仅提供`groupid`参数，则仅匹配IP地址和端口（忽略协议差异）。举例如下。

```

ds_is_from_list()
ds_is_from_list("100")
ds_is_from_list("100", "3")

ds_is_from_list("100", "3", "sip:127.0.0.1:5080")

```

注意，由于Lua不支持函数重载，因此KEMI提供了几个Lua函数变体，具体如下。

```

KSR.dispatcher.ds_is_from_list()
KSR.dispatcher.ds_is_from_list(100)
KSR.dispatcher.ds_is_from_list_mode(100, 3)
KSR.dispatcher.ds_is_from_list_uri(100, 3, "sip:127.0.0.1:5080")

```

6.4.3 使用permissions模块判断

permissions模块有两个参数可以配置允许（allow）和拒绝（deny）对应的地址文件，具体如下。

```

modparam("permissions", "default_allow_file", "/etc/permissions.allow")
modparam("permissions", "default_deny_file", "/etc/permissions.deny")

```

地址文件的结构如下。

```
(groupid,int) (address,str) (netmask,int,o), (port,int,o) (tag,str,o)
```

其中，int表示整数、str表示字符串、o表示可选参数。

- groupid:** 组号。
- address:** IP地址，可以是IPv4或IPv6。
- netmask:** 掩码中1的位数，如24表示255.255.255.0，32表示255.255.255.255。
- port:** 端口。
- tag:** 标签，如果执行该模块相应的函数后结果能匹配（返回true），则可以在peer_tag_avp参数指定的AVP中保存该值，以便后面读取。

地址文件示例如下。

```

1 127.0.0.1 32 0 tag1
1 10.0.0.10

2 192.168.1.0 24 0 tag2
2 192.168.2.0 24 0 tag3

3 [1:5ee::900d:c0de]

```

示例1 使用allow_source_address(group_id)检查SIP消息的来源IP地址是否属于某个组，然后根据来源进行路由。

```

if KSR.permissions.allow_source_address(1) > 0 then
    KSR.info("Coming from address group 1")
elseif KSR.permissions.allow_source_address(2) > 0 then
    KSR.info("Coming from address group 2")
end

```

示例2 使用allow_address_group（地址、端口）匹配来源地址和端口，如果端口为0则表示可以匹配任意端口。根据是否匹配决定走的路由，或者决定是否发起Challenge验证。

```

local si = KSR.pv.gete("$si")
KSR.info("request: si: " .. si)
address_group = KSR.permissions.allow_address_group(si, 0)
tlog('info', "address_group: " .. address_group)

if address_group == -1 then -- 未找到
    KSR.info("si: " .. si .. " not found, rejecting call")
    KSR.x.drop()
elseif address_group == 2 then -- 来自 FreeSWITCH Media Servers ...
    return ksr_dispatch_route_from_media_server()
end

```

除此之外，还有更多的匹配函数和算法，它们适用于不同的场景，详情参考
<https://kamailio.org/docs/modules-devel/modules/permissions>。

6.4.4 使用geoip2模块判断

GeoIP是一个地理IP地址数据库，可以用于判断一个IP地址属于哪个国家、地区以及城市，多用于和国际呼叫相关的业务。Kamailio中有一个geoip模块，但那个模块比较旧，且不支持KEMI，因此在这里我们使用geoip2模块。

geoip2模块可以执行对MaxMind GeoIP2数据库的实时查询，该数据库中的数据是IP地址到地理位置



的映射。使用前需要先下载MaxMind GeoIP2数据库

在配置文件中配置模块和GeoIP库加载路径，具体如下。

```

loadmodule "geoip2.so"
modparam("geoip2", "path", "/etc/kamailio/GeoLite2-City_2021/GeoLite2-City.mmdb")

```

以下代码可实现查询、打印来源IP地址的属性，并丢弃所有来自国外IP地址的消息。

```

geoip2.lua:
function ksr_request_route()
    if KSR.geoip2 and KSR.geoip2.match(KSR.pv.gete("$si"), "src") > 0 then
        KSR.info("ip = " .. KSR.pv.gete("$si"))
        KSR.info("cc = " .. KSR.pv.gete("$cip2(src=>cc)") .. "\n") -- 国家代码
        KSR.info("regn = " .. KSR.pv.gete("$cip2(src=>regn)") .. "\n") -- 地区名称
        KSR.info("regc = " .. KSR.pv.gete("$cip2(src=>regc)") .. "\n") -- 地区代码
        KSR.info("city = " .. KSR.pv.gete("$cip2(src=>city)") .. "\n") -- 城市
        if KSR.pv.gete("$cip2(src=>cc)") ~= "CN" then -- 丢弃所有IP地址来源国家不是CN
            KSR.x.drop();
        end
    end
end

```

下面是运行两个示例产生的日志，仅供参考。

```

ip = 113.116.53.141
cc = CN
regn = Guangdong
regc = GD

city = Shenzhen

ip = 146.88.240.4
cc = US
regn = Michigan
regc = MI
city = Southfield

```

使用上述方式一般判断粒度会比较粗，其精准度完全依赖于IP地址库的精确度。在国内ipip.net是一个专业的IP地址数据库提供商，很多互联网大厂都是他们的客户。不过笔者没有使用过他们的产品，所以这里就不介绍了，对IP地址要求比较高的用户可以自行研究。

6.5 API路由

随着业务越来越广泛，大家对动态数据处理的需求越来越强烈，而且在很多场景中都需要对其他系统进行实时对接，比较典型的对接方式是HTTP接口方式、消息中间件（MQ）方式等。这种通过API动态查询并产生路由的方式就称为API路由。Kamailio也有一些支持模块可实现跟其他系统的对接。

6.5.1 通过HTTP查询路由

HTTP API是最常用的API，其简单且易于实现。下面来看一个例子。

1.准备环境

我们首先需要准备一个HTTP服务器，以便提供路由查询。你可以使用你喜欢的任何语言写一个服务器，不过在这里，既然我们学习了Kamailio，就用Kamailio做一个HTTP服务器。

Kamailio支持HTTP，首先打开配置文件中的WITH_XHTTP宏，实际上，它将解锁以下配置。

```
#!ifdef WITH_XHTTP
tcp_accept_no_cl=yes
loadmodule " xhttp.so"
modparam(" xhttp", "event_callback", "ksr_xhttp_event")
#endif
```

WITH_XHTTP宏使用 xhttp 模块并提供 HTTP 服务。说起来，SIP 其实跟 HTTP 差不多，所以支持 SIP 的 xhttp 模块，也可以轻松支持 HTTP。上述代码中配置了 xhttp 参数，这个参数用于告诉 Kamailio，收到 HTTP 请求时执行 ksr_xhttp_event 这个函数。下面的代码用于实现这个函数，详细信息在代码注释里。

```
function ksr_xhttp_event(evname)
    -- 打印日志，打印请求 IP 地址
    KSR.info("==== http request: " .. evname .. " " .. "Ri:" .. KSR.pv.get("SRI") .. "\n")
```

```

-- 为了安全，我们要求客户端在HTTP头中传递一个Token
-- 为了简单，我们使用硬编码的1234
if KSR.hdr.get("Authorization") ~= "Bearer 1234" then
    KSR.hdr.append_to_reply('WWW-Authenticate: Bearer error="invalid_token"\r\n')
    KSR xhttp xhttp_reply("401", "Unauthorized", "", '{"code": 401, "message": "invalid_token"}')
    return
end

-- 获取Content-Type头域和请求Body
local content_type = KSR.pv.get("Sct") or ""
local req_body = KSR.pv.get("Srb")

-- 我们使用JSON格式传递参数
if not req_body or content_type ~= "application/json" then
    KSR xhttp xhttp_reply(400, "Client Error", "text/plain", "invalid content_type or body")
    return
end

-- 将参数打印出来
KSR.notice("request body: " .. req_body .. "\n")
-- 如果需要，我们也可以解析JSON参数，根据JSON内容决定返回什么样的值
-- local jbody = cjson.decode(req_body)

-- 为了简单，我们固定返回一个路由地址，实际使用时可以根据请求参数决定返回内容
local reply = {
    route = FSI_URI
}
local body = cjson.encode(reply)
KSR xhttp xhttp_reply(200, "OK", "application/json", body)
return 1
end

```



其中，`cjson` 是一个Lua模块，可以在Kamailio容器的命令行上使用如下命令安装`cjson`。

```

make sh          # 进入容器
apk add lua lua-cjson  # 安装

```

有了HTTP服务器，我们就可以启动Kamailio，并用curl客户命令向它发送HTTP请求了。具体如下。

```

curl -H "Authorization: Bearer 1234" -H 'Content-Type: application/json' -d '{}'
localhost:5060/
{"route":"sip:kb-fs1:5080"}

```



`curl`即cURL，它是一个HTTP命令行客户端。上述命令表示向`localhost:5060/`发起HTTP请求，使用-H添加两个头域，并使用-d传送一个空的JSON对象字符串。服务器收到后（并不检查请求参数）返回一个JSON对象，里面有一个`route`信息。

2.客户端请求路由脚本

有了上述这个用于测试的HTTP服务器，我们就可以再增加HTTP客户端了，HTTP客户端使用了



`http_client` 这个模块，实现代码如下。

```
loadmodule "http_client"
```

路由脚本如下。

```
http_client.lua:
local cjson = require 'cjson'

function ksr_request_route()
    ksr_register_always_ok()

    -- 获取请求信息，并打印日志
    local rm = KSR.pv.get("rm")
    local fu = KSR.pv.get("fu")
    local ru = KSR.pv.get("ru")
    local si = KSR.pv.get("si")
    KSR.info("request: si: " .. si .. " rm: " .. rm .. " from " .. fu .. " to " ..
        ru .. "\n")

    -- 构造请求 URL，在此我们从 kamailio.cfg 的宏里获取要请求的 IP 地址和端口号
    local url = "http://" .. KSR.kx.get_def("KAM_IP_LOCAL") .. ":" .. KSR.kx.get_def
        ("KAM_SIP_PORT")
    -- 构造请求参数
    local req = {
        rm = rm,
        fu = fu,
        ru = ru,
        si = si,
    }
    -- 将请求参数转成 JSON 字符串
    local post_body = cjson.encode(req)
    -- 发送 HTTP 请求，结果会放到 $var(result) 中
    local code = KSR.http_client.query_post_hdks(url, post_body,
        "Content-Type: application/json\r\nAuthorization: Bearer 1234",
        "$var(result)")

    -- 打印状态码和返回值
    KSR.info("code: " .. code .. "\n")
    KSR.info("result: " .. KSR.pvx.var_get("result") .. "\n")

    if code == 200 then
        -- 从返回的 JSON 信息中获取路由，并转发
        result = cjson.decode(KSR.pvx.var_get("result"))
        KSR.forward_uri(result.route)
    else
        KSR.sl.sl_send_reply(500, "Internal Error")
    end
end
```

在这个例子中，Kamailio既作为HTTP客户端又作为服务器，当收到SIP呼叫时其通过HTTP POST请求查询路由并转发。当然，简单的请求也可以使用GET（对应KSR. http_client.query(url, pv) 方法获取）。

3.异步HTTP请求

同步请求会阻塞SIP处理，如果在并发量比较大且HTTP服务器响应比较慢的情况下，会影响性能，这时候可以使用异步请求。异步请求将HTTP请求推到独立的进程中执行，执行完毕后再通过回调的方式回传结果进而唤醒原来的事务继续处理。



异步HTTP请求需要加载http_async_client模块。

```
loadmodule "http_async_client"
modparam("http_async_client", "workers", 2) # 启动几个 worker 进程
```

路由脚本如下。

```

http_async_client.lua:
local cjson = require 'cjson'

function ksr_request_route()
    ksr_register_always_ok()

    local rm = KSR.pv.gete("$rm")
    local fu = KSR.pv.gete("$fu")
    local ru = KSR.pv.gete("$ru")
    local si = KSR.pv.gete("$si")
    KSR.info("request: si: " .. si .. " rm: " .. rm .. " from " .. fu .. " to " .. ru .. "\n")

    if not KSR.is_method_in("I") then -- 为了简单，仅处理 INVITE 消息
        return
    end

    -- 构造 URL
    local url = "http://" .. KSR.kx.get_def("KAM_IP_LOCAL") .. ":" .. KSR.kx.get_def("KAM_SIP_PORT")
    local req = {rm = rm, fu = fu, ru = ru, si = si}
    local post_body = cjson.encode(req)
    -- 将请求相关的参数放到 $http_req() 相关的 PV 中
    KSR.pv.sets("$http_req(method)", "POST")
    KSR.pv.sets("$http_req(hdr)", "Content-Type: application/json")
    KSR.pv.sets("$http_req(hdr)", "Authorization: Bearer 1234")
    KSR.pv.sets("$http_req(body)", post_body)
    KSR.pv.seti("$http_req(suspend)", 1) -- 请求时挂起当前事务
    KSR.tm.t_newtran() -- 创建一个新的事务以便它可以被挂起
    -- HTTP 执行结束后，回调第二个参数中指定的函数
    local code = KSR.http_async_client.query(url, "ksr_async_callback")
    KSR.info("code: " .. code .. "\n")
    if code < 0 then
        KSR.err("suspend error\n")
    elseif code == 0 then
        KSR.info("suspended\n")
    else
        KSR.info("not suspended\n")
    end
end

-- HTTP 请求结束后继续执行事务
function ksr_async_callback(evname)
    -- KSR.info("callback: " .. evname .. "\n")

    local http_ok = KSR.pv.get("$http_ok") -- 请求是否成功
    local http_rs = KSR.pv.get("$http_rs") -- 结果状态码
    local http_rb = KSR.pv.get("$http_rb") -- 结果 Body
    KSR.info("http_ok=" .. http_ok .. " http_rs=" .. http_rs .. " http_rb=" .. http_rb .. "\n")

    if http_rs == 200 then
        -- 如果返回正确的结果，则解析 JSON 信息，根据返回结果中的 route 值转发
        result = cjson.decode(http_rb)
        KSR.pv.sets("$du", result.route)
        KSR.tm.t_relay()
    else
        KSR.tm.t_send_reply(500, "Internal Error")
    end
end

```

异步执行需要经过“挂起-恢复”的过程，需要使用回调函数，因而代码写起来有点复杂，但理解并掌握了异步执行的原理，用起来会非常顺利。

4.KSR.http_async_client.query(url, callback)函数

KSR.http_async_client.query(url, callback)函数用于发送HTTP或HTTPS请求，其涉及的参数如下。

url: 字符串，请求URL。

callback: 字符串，回调函数名称。

如果在执行该函数前有一个事务（用t_newtran()创建），则事务会被挂起，路由脚本结束（以便处理下一个请求），并在HTTP请求结束（或失败、超时）时恢复事务的执行。

如果该函数在非事务的环境中使用，或者在\$http_req(suspend)=0的环境下使用，路由脚本将继续执行，但是回调函数仍将被调用（不影响SIP消息处理流程，可用于写日志或写数据库等），在回调函数中也可以获取到HTTP请求结果。

该函数的返回值如下。

- 0: 停止后续脚本执行，事务将在回调中恢复执行。
- 1: 在无事务或\$http_req(suspend) = 0场景下继续后续脚本执行。
- 1: 出错。

5.请求参数

HTTP的请求参数用\$http_req(key)PV指定，其中的PV是只写的。其中，key的取值如下。

- all: 将值设为\$null，用于将所有参数设为默认值（在模块参数中设置的值）。
- hdr: 设置、修改、删除HTTP头域。多次设置该值将使结果中出现多个头域。
- body: 请求的Body。
- method: 用于设置方法，支持GET、POST、PUT和DELETE。默认为GET，但在存在Body的情况下，默认为POST。
- timeout: 超时的毫秒数，一般来说该值应该小于tm.fr_timer的值，因为后者优先级更高。
- tls_client_cert: TLS客户端证书。
- tls_client_key: TLS客户端证书密钥。
- tls_ca_path: TLS CA证书路径。
- authmethod: 鉴权方法，它是一个比特位整数，因此可以同时支持多个方法。但支持多个方法时Kamailio会发送一个额外的请求以获取服务端支持的鉴权方法。所以仅设置一个特定的方法有助于提高性能。默认值为3，即执行BASIC和Digest验证（3=1+2），其他可能的取值如下。
 - 1: HTTP BASIC（基本）验证。
 - 2: HTTP Digest（摘要）验证。
 - 4: GSS-Negotiate（通用安全服务协商）验证，微软提供的一种验证方式。
 - 8: NTLM（NT Lan Manager）验证，微软提供的一种安全协议。
 - 16: HTTP Digest with IE flavour，微软IE特有的一种摘要验证方式。
- username: 设置验证的用户名。
- password: 设置验证密码。
- suspend: 如果设为0，则不挂起；如果设为1，则挂起。
- tcp_keepalive: 决定是否支持TCP keepalive。
- tcp_ka_idle: 设置TCP keepalive空闲时间。
- tcp_ka_interval: 设置TCP keepalive间隔。
- follow_redirect: 若为非0值，则表示cURL处理HTTP 3xx重启向消息；若为0则该参数不起作用。该参数默认值可以由curl_follow_redirect全局参数指定。

6.HTTP返回结果

上述函数异步返回的结果可以通过\$http_*相关的PV值获取，下列PV值仅在回调函数中有效。

- \$http_ok: HTTP请求成功为1，失败为0（检查\$http_err并获取详细信息）。
- \$http_err: 请求失败原因的字符串描述，不出错则为>null。
- \$http_rs: HTTP状态码，如200。
- \$http_rr: HTTP返回消息中的原因字符串，如200消息中的“OK”、404消息中的“Not Found”等。
- \$http_hdr(Name): 获取HTTP头域，如果有多个同名头域，也可以使用\$(http_hdr(Name)[N]获取第N个头域。
- \$http_mb及\$http_ml: HTTP响应的缓冲区（包括响应头域）以及缓冲区长度。
- \$http_rb及\$http_bs: HTTP响应的Body及Body的长度。



- \$http_time(name): cURL提供的HTTP请求/响应时长。name取值如下。
 - total: 总传输时长。
 - lookup: DNS查询时长。
 - connect: 连接时长。
 - appconnect: 从建立连接到TLS握手结束的时长。
 - pretransfer: 从请求开始到可以发起HTTP传输的时长。
 - starttransfer: 从请求开始直到收到第一个字节的时长。
 - redirect: 从重定向开始（可能有多次重定向）到最终开始真正传输的时长。

6.5.2 rtjson

在上面的HTTP脚本中我们使用了自己定义的JSON消息，这用起来简单方便，但在复杂的场景中可



能代码会比较冗长。Kamailio有一个rtjson模块，其中定义了一个“相对标准”的方法，可以比较方便地处理串行或并行转发。

rtjson模块使用的JSON结构如下。

- version: 建议设为"1.0"，因为后续可能会有新版本，目前不检查该版本号。
- routing: 取值为serial或parallel，分别代表串行转发或并行转发。
- routes: 表示一个路由数组，包含多个路由。

路由参数如下。

- uri: Request URI，即请求URI。
- dst_uri: Destination URI，即目的地URI。
- path: 以逗号分隔的PathURI地址列表，如<sip:127.0.0.1:5084>和<sip:127.0.0.1:5086>。
- socket: 本地Socket。
- headers: 可以指定From头域、To头域及添加的其他头域，当变更From或To头域时需要uac模块支

持，若仅用于串行转发，则需要使用rtjson_update_branch()进行相关设置。

□ branch_flags：分支标志，仅用于串行转发，需要使用rtjson_update_branch()进行设置。

□ fr_timer：设置tm模块的fr_timer值，仅用于串行转发，需要使用rtjson_update_branch()进行设置。

□ fr_inv_timer：设置tm模块的fr_inv_timer值，仅用于串行转发，需要使用rtjson_update_branch()进行设置。

其他值会被rtjson模块忽略，但额外的值也可以作为随路数据使用，并使用jansson模块或 cJSON解析，如可以使用事务ID元组（index, label）恢复一个挂起的事务。

以下JSON示例来自rtjson模块的文档，其中routes是一个数组（用[]表示），数组中每一个对象（用{}表示）代表一条路由。

```
{  
    "version": "1.0",  
    "routing": "serial",  
    "routes": [  
        {  
            "uri": "sip:bob@b.example.org:5060",  
            "dst_uri": "sip:192.0.2.1:5060",  
            "path": "<sip:192.0.2.2:5084>, <sip:192.0.2.2:5086>",  
            "socket": "udp:192.0.2.20:5060",  
            "headers": {  
                "from": {  
                    "display": "Alice",  
                    "uri": "sip:alice@a.example.org"  
                },  
                "to": {  
                    "display": "Bob",  
                    "uri": "sip:bob@b.example.org"  
                },  
                "extra": "X-Hdr-A: abc\r\nX-Hdr-B: bcd\r\n"  
            },  
            "branch_flags": 8,  
            "fr_timer": 5000,  
            "fr_inv_timer": 30000  
        },  
        {  
            "uri": "sip:bob@b.example.org:5060",  
            "dst_uri": "sip:192.0.2.10:5060",  
            "path": "<sip:192.0.2.2:5084>, <sip:192.0.2.2:5086>",  
            "socket": "udp:192.0.2.20:5060",  
            "headers": {  
                "from": {  
                    "display": "Alice",  
                    "uri": "sip:alice@a.example.org"  
                },  
                "to": {  
                    "display": "Bob",  
                    "uri": "sip:bob@b.example.org"  
                },  
                "extra": "P-Asserted-Identity: <sip:alice@a.example.org>\r\n"  
            },  
            "branch_flags": 8,  
            "fr_timer": 5000,  
            "fr_inv_timer": 30000  
        },  
        {  
            "uri": "sip:bob@b.example.org:5060",  
            "dst_uri": "sip:192.0.2.11:5060",  
            "path": "<sip:192.0.2.2:5084>, <sip:192.0.2.2:5086>",  
            "socket": "udp:192.0.2.20:5060",  
            "headers": {  
                "from": {  
                    "display": "Alice",  
                    "uri": "sip:alice@a.example.org"  
                },  
                "to": {  
                    "display": "Bob",  
                    "uri": "sip:bob@b.example.org"  
                },  
                "extra": "P-Asserted-Identity: <sip:alice@a.example.org>\r\n"  
            },  
            "branch_flags": 8,  
            "fr_timer": 5000,  
            "fr_inv_timer": 30000  
        }  
    ]  
}
```

下面是一个示例路由脚本。为简单起见，我们只是使用了阻塞方式的HTTP请求获取rtjson（也可以使用异步方式，只是略复杂），当然，使用任何其他方式获取JSON也是可以的。下面的示例是一

个串行转发、http_client、rtjson三者结合的例子，前文中解释过类似的内容，所以这里不再额外解释。

```
rtjson.lua:
local cjson = require 'cjson'

function ksr_request_route()
    ksr_register_always_ok()

    -- 如果转发失败，则由失败函数处理，可以继续转发
    if KSR.is_INVITE() then
        if KSR.tm.t_is_set("failure_route") < 0 then
            KSR.tm.t_on_failure("ksr_failure_manage");
        end
    end

    local rm = KSR.pv.gete("Srm")
    local fu = KSR.pv.gete("Sfu")
    local ru = KSR.pv.gete("Sru")
    local si = KSR.pv.gete("$si")
    KSR.info("request: si: " .. si .. " rm: " .. rm .. " from " .. fu .. " to " .. ru .. "\n")

    local url = "http://" .. KSR.kx.get_def("KAM_IP_LOCAL") .. ":" ..
        KSR.kx.get_def("KAM_SIP_PORT")
    local req = {
        rm = rm,
        fu = fu,
        ru = ru,
        si = si,
    }
    local post_body = cjson.encode(req)
    local code = KSR.http_client.query_post_hdrs(url, post_body, "Content-Type:
        application/json\r\nAuthorization: Bearer 1234", "$var(result)")

    KSR.info("code: " .. code .. "\n")
    KSR.info("result: " .. KSR.pvx.var_get("result") .. "\n")

    if code == 200 then
        -- 如果成功获取JSON，则用它初始化rtjson，并转发
        KSR.rtjson.init_routes(KSR.pvx.var_get("result"));
        KSR.rtjson.push_routes();
        KSR.tm.t_relay()
    else
        KSR.s1.s1_send_reply(500, "Internal Error")
    end
end

function ksr_failure_manage()
    if KSR.tm.t_is_canceled()>0 then
```

```

        return 1
    end

    local status_code = KSR.tm.t_get_status_code()
    KSR.warn("call failed with status=" .. status_code .. " retrying ...\\n")

    -- 如果上次转发失败, 则检查 JSON 的路由数组中是否还有后续的路由, 如果有, 则继续
    if KSR.rtjson.next_route() then
        KSR.tm.t_relay()
    else
        KSR.tm.t_send_reply(500, "No more routes available")
    end
end

function ksr_xhttp_event(evname)
    KSR.info("---- http request:" .. evname .. " " .. "Ri:" .. KSR.pv.get("$Ri") .. "\\n")

    if KSR.hdr.get("Authorization") == "Bearer 1234" then
        KSR.hdr.append_to_reply('WWW-Authenticate: Bearer error="invalid_token"\\r\\n')
        KSR.xhttp.xhttp_reply("401", "Unauthorized", "", '{"code": 401, "message": "invalid_token"}')
        return
    end

    local content_type = KSR.pv.get("ScT") or ""
    local req_body = KSR.pv.get("$rb")

    if not req_body or content_type == "application/json" then
        KSR.xhttp.xhttp_reply(400, "Client Error", "text/plain", "invalid content_"
            "type or body")
        return
    end

    -- local jbody = cjson.decode(req_body)
    KSR.notice("request body: " .. req_body .. "\\n")

    -- 构造并返回一个 rtjson 结构的 JSON
    local reply = {
        version = "1.0",
        routing = "serial",
        routes = {
            dst_uri = FS1_URI,
            headers = {
                from = {
                    display = "Caller",
                    uri = "sip:caller@domain.com",
                },
                to = {
                    display = "Callee",
                    uri = "sip:callee@domain.com"
                },
            },
            branch_flags = 8,
            fr_timer = 5000,
            fr_inv_timer = 30000,
        },
        {
            dst_uri = FS1_URI,
            headers = {

```

```

        from = {
            display = "Caller",
            uri = "sip:caller@domain.com",
        },
        to = {
            display = "Callee",
            uri = "sip:callee@domain.com"
        },
    },
    branch_flags = 8,
    fr_timer = 5000,
    fr_inv_timer = 30000,
)
}
local body = cjson.encode(reply)
KSR xhttp xhttp_reply(200, "OK", "application/json", body)
return 1
end

```

从上述脚本中可以看出，我们写好Lua代码后，实际的路由逻辑由JSON数据驱动，可以根据JSON进行串行或并行转发。比如，假设你有两个后台服务器，想做1:1负荷分担，则基于上述例子，在HTTP服务器每次返回的结果中反转一下数组中的路由顺序即可。当然，上述示例中两个路由的地址都是一样的，你可以根据现场的情况改成不一样的。

6.5.3 evapi



evapi 模块提供一个事件消息流程。它只是建立一个TCP Socket连接，甚至都没有“协议”，所有Socket上收发的消息都由用户自定义。**evapi**仅支持服务器模式，也就是说你要实现一个对应的TCP客户端与它通信，并通过数据控制路由逻辑。

1.evapi服务

通过加载evapi模块，我们就可以创建一个TCP服务，这跟xhttp模块非常相似，不同之处是我们可以自己定义协议。

模块加载和参数配置如下。

```

loadmodule "evapi.so"
modparam("evapi", "bind_addr", "127.0.0.1:8888")      # 启动一个 TCP 服务监听 8888 端口
modparam("evapi", "event_callback", "ksr_evapi_event")# 服务回调函数
modparam("evapi", "netstring_format", 0)

```

其中，`netstring_format`有如下两个取值。

0：不使用Netstring格式，即使用的是裸数据，想传什么传什么。

1：使用Netstring格式。

Netstring格式是一种很简单的数据封装格式：开始处为使用ASCII字符串表示的数据长度，接着是一个冒号，然后是真正的字符串数据，最后是一个逗号。如在Netstring格式中，“hello world!”的十六进制ASCII码表示为：

```
31 32 3a 68 65 6c 6c 6f 20 77 6f 72 6c 64 21 2c
```

其中，31是字符1的ASCII码，32是2的ASCII码，也就是数据长度为12，3a是冒号，中间是“hello world!”，最后是一个逗号，即：“12:hello world!。”

“0;”表示一个空字符串，对应的Netstring格式中的ASCII码为“30 3a 2c”。

为了简单，下面的例子我们不使用Netstring格式（netstring_format = 0）。代码及注释如下。

```
evapi-event.lua:  
function ksr_evapi_event(evname)    -- TCP 服务回调函数，如果有请求进来，就执行该函数  
    if evname == "evapi:message-received" then -- 收到请求消息  
        local msg = KSR.pv.get("Sevapi(msg)") -- 获得消息内容  
        if msg:find("stats") == 1 then          -- 如果内容以 stats 开头  
            local request_body = '{"jsonrpc": "2.0", "method": "stats.fetch",  
                "params": ["all"], "id": 1}'  
            KSR.jsonrpcs.exec(request_body)      -- 构造一个 JSON-RPC 请求以获取内部状态  
            local code = KSR.pv.get("$_jsonrpl(code)")  
            local response_body = KSR.pv.get("$_jsonrpl(body)")  
            KSR.evapi.relay(response_body)        -- 返回结果 JSON 字符串  
        end  
    end  
end
```



下面我们可以用nc命令给Kamailio模块发一个字符串，去获取Kamailio所有的统计信息。命令如下。

```
echo 'stats' | nc 127.0.0.1 8888
```

执行上述命令就可以看到Kamailio返回的内容统计信息，部分结果如下。

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "core.bad_URIs_rcvd": "0",  
        "core.bad_msg_hdr": "0",  
        "core.drop_replies": "0",  
        "core.drop_requests": "0",  
        "core.err_replies": "0",  
        "core.err_requests": "0",  
        "core.fwd_replies": "0",  
        "core.fwd_requests": "0",  
        "core.rcv_replies": "0",  
        "core.rcv_replies_18x": "0",  
        "后面省略很多行....."  
    }  
}
```

如果你想尝试Netstring格式，可以在配置文件中把netstring_format参数配置为1，对应的命令行调整如下。

```
echo '5:stats,' | nc 127.0.0.1 8888
```

2. 使用evapi进行路由

下面我们一起来看一个使用evapi获取路由目的地并进行路由的例子。evapi是异步执行的，因而需要采用Kamailio内部的t_suspend（挂起）和t_continue（继续执行）机制。

1) 协议设计

evapi没有提供任何协议，一切协议都由你自行决定。虽然Netstring也算是一个协议，但它只能算是一个传输层的协议，它只规定了消息长度和边界。在此，我们需要使用Netstring，因为evapi要使用TCP连接。TCP本身是一个字节流，是没有消息边界的，有了Netstring，用起来就会简单很多。

除了Netstring以外，我们还需要定义一个业务层的协议。在此，我们使用JSON-RPC。RPC的全称是Remote Procedure Call，即远程过程调用，一般由“请求-响应”组成。JSON-RPC即使用JSON格式

描述的RPC，它规定了请求、响应以及事件消息。本质上，JSON-RPC也是一个“信封”，实际的数据在“信封内部”传输。JSON-RPC协议定义非常简单，JSON-PRC消息是一个JSON对象，其中的属性如下。

- **jsonrpc**: 协议版本号。
- **id**: 请求的唯一标志，可以是数字、字符串或null（在此我们不讨论null）。如果id不存在，则认为是一个事件通知，而无须回复。
- **params**: 请求的参数，可以是一个对象（以{}表示）或数组（以[]表示）。
- **result**: 响应消息中的返回结果，也可以是对象或数组。
- **error**: 响应消息中的错误，是一个对象，里面有code和message两个必需的属性，以及可选的data属性（用于提供一些额外信息）。

注意result和error是互斥的，两者不应该同时出现。详细的JSON-RPC协议标准可以参阅相关文档



。下面列举一些实际的消息示例。

Kamailio永远是一个evapi服务器，我们需要实现一个客户端连接到它，然后向它提供路由。

客户端连接到Kamailio后，首先发一个登录请求，请求的内容如下（注意params里的内容仍然可以根据你的需要随意定义）。

```
{
    "jsonrpc": "2.0",
    "id": "0",
    "method": "login",
    "params": {
        "username": "evapi",
        "password": "secret"
    }
}
```

服务器会返回一个登录成功的响应消息，具体如下。

```
{
    "jsonrpc": "2.0",
    "id": "0",
    "result": {
        "code": 200,
        "message": "login success"
    }
}
```

然后客户端就安静地等待服务器的路由请求。如果Kamailio侧有呼叫到来，需要查找路由，此时会向evapi客户端发送一个路由请求。

```
{  
    "jsonrpc": "2.0",  
    "id": "53340:1276135139",  
    "method": "route",  
    "params": {  
        "ru": "sip:9196@192.168.7.8:35060;transport=tcp",  
        "rm": "INVITE",  
        "fu": "sip:1001@192.168.7.8",  
        "si": "172.22.0.1",  
        "dest": "9196"  
    }  
}
```

其中，`id`是当前事务的`index`和`label`的组合，`params`中的参数可以自己随意定义，在此我们定义`dest`为被叫号码，这些参数我们还会在后面实际的脚本中详细解释。

客户端收到上述消息后，就可以根据`params`中的信息去查找数据库或调用其他的API并生成一条路由，响应消息示例如下。

```
{  
    "jsonrpc": "2.0",  
    "id": "53340:1276135139",  
    "result": {  
        "route": "sip:kb-fsl:5080"  
    }  
}
```

然后，Kamailio就可以根据这里的`route`参数进行路由了。

2) 客户端代码

客户端代码在这里也是使用Lua实现的，大家可以在随书附赠的代码的`examples/evapi_client`目录中找到。如果你使用本书推荐的Docker配置，可以在容器中使用下列命令安装相关的依赖。

```
apk add lua lua-socket lua-cjson
```

然后就可以按下面的方式启动客户端了。

```
lua evapi_client.lua
```

客户端启动后，就会主动连接Kamailio的`evapi`监听端口，主要代码详解如下。

```

evapi_client.lua:
json = require("cjson")           -- JSON 支持
socket = require("socket")         -- TCP Socket 支持
tcp = assert(socket.tcp())        -- 确保 Socket 是可用的

host, port = "127.0.0.1", 8888    -- 服务地址和端口

-- 下面是后端 FreeSWITCH 的 IP 地址和端口等配置，跟 examples.lua 中的定义相同
FS1_IP = 'kb-fal'
FS1_PORT = '5080'
FS1_IP_PORT = FS1_IP .. ':' .. FS1_PORT
FS1_URI = 'sip:' .. FS1_IP_PORT

function write_netstring_data(str) -- 通过 Socket 发送 Netstring 格式的字符串
    local len = #str
    tcp:send(tostring(len) .. ":" .. str .. ",")
end

function read_netstring_data()      -- 在 Socket 上读 Netstring 格式的字符串，内容略
end

print("connecting to " .. host .. ":" .. port)
tcp:connect(host, port)          -- 连接 Kamailio EVAPI 服务，此处并未检查是否出错
print("connected")
-- 发送 login 请求
write_netstring_data('{ "jsonrpc": "2.0", "id": "0", "method": "login", "params": {
"username": "evapi", "password": "secret" } }')

while true do                   -- 无限循环等待接收 Socket 消息
    local data, status = read_netstring_data() -- 读取 Netstring 格式的消息
    if status == "closed" then break end       -- 如果 Socket 断开则停止
    print("--- read_netstring_data:\n" .. data) -- 打印收到的消息

    local req = json.decode(data)             -- 解析 JSON 消息，得到一个 Lua Table
    if req and req.id and req.method == 'route' and req.params then -- 简单合法性检查
        local dest = req.params.dest          -- 被叫号码
        print("received a call to " .. dest .. "\n") -- 打印调试消息
        local response = {}                  -- 构造响应消息
        response.id = req.id                -- id 要原样返回
        response.jsonrpc = "2.0"            -- JSON-RPC 版本号固定
        local result = {}                  -- 真正的结果数据
        response.result = result
        result.route = FS1_URI            -- 路由字符串
        write_netstring_data(json.encode(response)) -- 将 Lua Table 转成字符串并发送
    end
end

```

3) 服务端代码

evapi的路由逻辑我们上面已经解释得很清楚了，在此我们直接看代码，这些代码在随书附赠的evapi.lua文件中。

```

evapi.lua:
local cjson = require 'cjson'           -- 加载 cjson

function ksr_request_route()
    ksr_register_always_ok()
    -- 获取并打印 SIP 请求

```

```

local rm = KSR.pv.gete("$rm")
local fu = KSR.pv.gete("$fu")
local ru = KSR.pv.gete("$ru")
local si = KSR.pv.gete("$si")
local dest = KSR.pv.gete("$dest")
KSR.info("request: si: " .. si .. " rm: " .. rm .. " from " .. fu .. " to " ..
        ru .. " dest " .. dest .. "\n")

if not KSR.is_method_in("I") then -- 在此我们只处理 INVITE 消息，其他消息将忽略
    return
end

KSR.tm.t_newtran() -- 启动一个新的事务，以便挂起
local tindex = KSR.pv.gete("ST(id_index)") -- 获取新事务的 index，一个无符号整数
local tlabel = KSR.pv.gete("ST(id_label)") -- 获取新事务的 label，一个无符号整数
KSR.info("transaction: index = " .. tindex .. " label = " .. tlabel .. "\n")
-- 构造 RPC 请求，它是一个 Lua Table，将事务 index 和 label 放到 id 中
local req = {
    jsonrpc = "2.0", method = "route",
    id = tindex .. ":" .. tlabel,
    params = {
        rm = rm, fu = fu, ru = ru, si = si, dest = dest
    }
}
local rpc = cjson.encode(req) -- 将请求的 Lua Table 转换成字符串
KSR.info("event: " .. rpc .. "\n")
local code = KSR.evapi.async_relay(rpc)
if code == 1 then -- 正常执行，事务被挂起，结果将异步返回
    KSR.info("Transaction suspended\n")
else -- 出错
    KSR.err("Transaction suspend error, code = " .. code .. "\n")
end
-- 路由脚本到此为止，不会阻塞，可以继续处理下一次请求
end

function ksr_evapi_event(evname) -- 当收到客户端发来的消息时将调用该函数
    if evname == "evapi:message-received" then
        local msg = KSR.pv.gete("Sevapi(msg)") -- 获得收到的消息，已去除 Netstring 结
        构后的真正 JSON 字符串
        KSR.info('EVAPI SERVER Received: ' .. msg .. "\n") -- 打印该字符串
        local rpc = cjson.decode(msg) -- 解析 JSON 消息
        if rpc then -- 解析成功
            if rpc.id and rpc.method == 'login' then -- 是一个 RPC 请求消息，是登录请求
                KSR.info("client login with user: " .. rpc.params.username ..
                        " password: " .. rpc.params.password .. "\n")
                local res = { -- 构造响应消息，简单起见我们允许所有
                    客户端登录
                    jsonrpc = "2.0",
                    id = rpc.id,
                    result = {
                        code = 200,
                        message = "login success",
                    }
                }
                local response = cjson.encode(res)
                KSR.evapi.relay(response) -- 返回响应消息。该函数会在 evapi 的 worker
                进程中执行，不会阻塞
            elseif rpc.id and rpc.result then -- 这是一个 RPC 消息
                tindex, tlabel = rpc.id:match("(.+):(.)") -- 取出事务 index 和 label
                KSR.pv.sets('$var{evmsg}', msg) -- 把这个消息存到变量里
                -- 唤醒并继续执行原有的事务，需要事务 index 和 label，以及一个回调函数
                KSR.tm.t_continue(tindex, tlabel, 'ksr_evapi_continue')
            end
        end
    end
end

function ksr_evapi_continue() -- 事务唤醒后，执行该回调函数
    local msg = KSR.pv.gete("$var{evmsg}") -- 从变量中获取路由描述的 RPC 消息
    KSR.info("Transaction resumed, continue. msg: " .. msg .. "\n")
    local rpc = cjson.decode(msg) -- 重新解析一遍
    KSR.pv.sets("$du", rpc.result.route) -- 通过改变 $du 来重设路由
    KSR.tm.t_relay() -- 转发 SIP 消息
end

```

通过上述脚本我们可以看出，上述异步执行的机制跟我们前面讲过的HTTP异步执行类似。上面我

们之所以可以使用JSON-RPC，是因为它有现成的协议标准。事务的挂起和唤醒需要事务的index和label参数，我们通过rpc.id将它们传到客户端，客户端再原样送回来，而这也正符合JSON-RPC协议对id的定义。当然我们也可以将这两个参数分别放到params里，并从result中返回，只是放到id里貌似更“优雅”。evapi并没有规定具体的协议，在实际使用时完全由你说了算，只要服务端和客户端一致就可以了。当然，你也可以使用6.5.2节介绍的rtjson协议。

4) 小结

为了方便讲解，我们上面提到的路由脚本写得比较简单，在实际使用时，需要进行更多的合法性检查，理论上所有通过Socket的应用都不能信任对端的数据（比如一个恶意的客户端发送了超长数据或者故意将字符串字段写成整数），因此，对收到的JSON数据的每一个参数都需要小心检查。当然，既然Kamailio要到客户端去查找路由，这就说明它对客户端还是有基本信任的，在实际使用时你肯定也会用相应的鉴权措施（如IP地址白名单和密码等）保护你的evapi端口不被非法连接。具体的安全措施超出了本节的范围，留给读者自行学习。

此外，在实际使用时，如果有多个evapi客户端连接上来，服务端会向所有客户端发送路由请求。这可能不是你想要的，需要你自行保证只有一个客户端连接到evapi服务上。Kamailio也提供了一个evapi_set_tag(tname)函数，可以给客户端打一个标签，然后就可以使用下列函数与具有特定标签(etag)的客户端进行交互了：evapi_multicast(evdata, etag)、evapi_async_multicast(evdata, etag)、evapi_unicast(evdata, etag)、evapi_async_unicast(evdata, etag)。

通过给客户端打标签，Kamailio的evapi服务端可以有选择性地与多个客户端交互，完成更复杂的路由功能，并使整体服务更健壮（如可实现一个客户端死掉由另一个接替等）。高级用法有多种，具体的我们在此就不多讲了，也留给读者自行学习。

6.6 在KEMI脚本中调用原生脚本中的路由块

并不是所有模块都支持KEMI，有时候有些功能在原生的路由块中实现起来比在Lua中实现更方便，或者你已经有了一些写好的路由块但不想花很多时间改由Lua实现，这时候，也可以从Lua脚本中调用原生脚本中的路由块。看下面的例子。

原生路由块如下。

```
route[NATIVE] {
    append_hf("P-hint: appended in NATIVE route blocks\r\n");
    Svar(exit) = 1;
    return;
}
```

Lua路由块实现代码（native.lua）如下。

```
function ksr_request_route()
    KSR.route("NATIVE")          -- 调用原生脚本中的 NATIVE 路由块并返回结果
    if KSR.pvx.var_get("exit") == 1 then
        KSR.x.exit()
    end
end
```

所以，原生的路由脚本可以调用Lua脚本中的路由块，Lua脚本也可以调用原生脚本中的路由块，你中有我，我中有你，就看你怎么组合。如果你已经有了很多原生的路由脚本，想迁移到Lua中，也可以通过这个方法一步一步替换，而不用一下子完成。当然，更重要的是可以通过该示例，进一步理解Kamailio中的路由处理逻辑，比如路由块其实相当于一个函数调用，并且可以有返回值。

Chapter 7

第7章

数据库操作

Kamailio本身不依赖于数据库，但在实际应用中，用户、中继及路由信息等，通常都会存储在数据库中。虽然Kamailio可以通过各种API（如上一章讲过的API路由）获取这些数据，但是直连数据库进行数据存取还是最直接的方式。Kamailio也支持很多类型的数据库，如常用的关系型数据库 PostgreSQL、MySQL、MariaDB、Oracle、SQL Server、SQLite等，以及非关系型数据库 MongoDB、Redis、Berkeley DB等。

为提高效率，Kamailio在大部分时候都是一次性把数据从数据库读到内存中。但有时候，也可以实时查询数据、获取数据。通过本章我们一起来看一下Kamailio中数据库的配置和使用方法。

7.1 初始化数据库

Kamailio支持主流的数据库，下面我们仅以PostgreSQL和MySQL为例讲一下对数据库进行初始化和配置的方法。

7.1.1 PostgreSQL

Kamailio自带了一个数据库初始化脚本kamdbctl，我们使用它来创建数据库，但该脚本不是很好用。它首先需要一个配置文件/etc/Kamailio/kamctrlc，在该文件中可以配置连接数据库的参数。

```
## 你的 SIP domain
SIP_DOMAIN=xswitch.cn
## 数据库引擎
DBENGINE=PGSQL

## 数据库服务器
DBHOST=kb-pg
## 数据库端口
DBPORT=5432
## 数据库名
DBNAME=Kamailio
## 数据库读写用户名，用于读写数据
DBRWUSER="Kamailio"
## 数据库读写用户密码
DBRWPW="Kamailio"
## 数据库只读用户名
DBROUSER="Kamailio"
## 数据库只读用户密码
DBROPW="Kamailio"
## 超级用户。该用户需要能创建数据库
DBROOTUSER="root"
## 超级用户密码
DBROOTPW="root"
```

另外，kamdbctl在创建数据库的过程中要多次询问数据库密码，为了避免出错，我们通过将密码写入.pgpass文件解决该问题。.pgpass是连接PostgreSQL的标准方法，感兴趣的读者可以阅读相关资料。

以下指令仅供参考，无须实际执行。

```
echo 'gw-pg:5432:kamailio:root:root' > ~/.pgpass
chmod 0600 ~/.pgpass
```

在db.yml中，我们使用POSTGRES_HOST_AUTH_METHOD=trust环境变量，所以，任何用户的访问都被认为是可信任的。出现这种情况的主要原因是kamdbctl尝试新建数据库，如果创建失败它就不往下执行了。在实际生产环境中，数据库往往是由超级用户建好的，理论上脚本只管建表就行。但数据库安全不是本书的重点，因此我们直接以root超级用户登录并建表。如果后续考虑安全性，可以去掉trust环境变量。

上述配置文件在我们的仓库中已经有了，为了简单我们也把相关指令直接写到Makefile中，只需要执行如下代码。

```
make up-pg          # 启动数据库容器
make create-kam-pg # 创建数据库
```

执行过程中将会在控制台上显示所有命令。在创建过程中，脚本可能会问你是否创建其他的表，一般回答“y”即可。如果任何地方出错，可以使用以下命令重来。

```
make down-pg          # 停止数据库容器  
rm -rf cache/pgdata # 清除缓存数据  
make up-pg           # 启动数据库容器  
make create-kam-pg   # 创建数据库
```

如果一切顺利，数据库就创建成功了。用如下方式查看一下。

```
make bash-pg          # 进入数据库容器  
psql kamailio         # 进入数据库  
  
\d                   # 显示数据库表  
  
kamailio=# select * from version;      # 查询版本号  
kamailio=# select * from dispatcher;    # 查询 dispatcher 表  
 id | setid | destination | flags | priority | attrs | description  
----+-----+-----+-----+-----+-----+-----  
(0 rows)
```

使用Ctrl+C组合键可以退出psql。更多的命令和使用方法请参阅PostgreSQL相关文档。

如果按上述方法创建数据库总出问题，也可以找到原始的建表语句进行手动创建，原始的SQL文件在/usr/share/kamailio/postgres/目录下。手动创建的方法如下（命令仅供参考）。

```
docker cp kb-kam:/usr/share/Kamailio/postgres /tmp/  
docker cp /tmp/postgres kb-pg:/tmp/  
make bash-pg  
psql  
CREATE user kamailio with password 'kamailio';  
CREATE DATABASE kamailio owner 'kamailio';  
exit;  
cd /tmp/postgres  
cat *.sql | psql kamailio
```

7.1.2 MySQL

不排除很多人熟悉并喜欢用MySQL。如果你使用MySQL，只需要修改kamctlrc，即将DBENGINE改成MYSQL即可，这样在文件的末尾会加载kamctlrc.mysql相关的配置，从而覆盖kamctlrc中的配置。

```
case $DBENGINE in  
  MYSQL|MySQL|MySQL)  
    ,/etc/kamailio/kamctlrc.mysql  
esac
```

执行如下命令创建数据库。

```
make create-kam-mysql
```

如果在上述代码执行过程中提示输入密码，直接按回车即可，因为这里我们没有使用密码。对于后续的几个提示，直接选Yes (“y”) 即可。如果出错，可以按如下方法删掉cache/mysql目录重来。

```
make down-mysql  
rm -rf cache/mysql  
make up-mysql  
make create-kam-mysql
```

如果按上述方法创建数据库总出问题，也可以找到原始的建表语句进行手动创建，原始的SQL文件在/usr/share/Kamailio/mysql/目录下。

7.2 配置数据库连接



数据库操作在sqllops 模块中实现，因而需要加载该模块。另外，要使用哪个数据库，还需要加载对应的数据库模块，如db_postgres或db_mysql。下面以前者为例。若要测试例子中的代码，可以取消掉Kamailio.conf中##WITH_PG前面的注释（改成一个#）。

```
loadmodule "db_postgres.so"
loadmodule "sqllops.so"
modparam("sqllops", "sqlcon", EXAMPLEDBURL)
```

其中，EXAMPLEDBURL是一个宏，它是一个字符串，定义如下。

```
#define EXAMPLEDBURL "example=>postgres://kamailio:kamailio@kb-pg/kamailio"
```

相应的语法如下。

```
#define EXAMPLEDBURL "连接 ID=> 数据库连接字符串"
#define EXAMPLEDBURL "连接 ID=> 数据库类型:// 用户名 : 密码 @ 主机名或 IP 地址 / 数据库名称"
```

其中连接ID用于在后续的路由脚本中引用，此处是example。sqlops参数可以有多行，可用于连接多个不同的数据库，甚至不同类型的数据。

7.3 在路由时进行SQL查询

在路由时可以使用如下个方法查询数据库。

```
sql_query("连接 ID", sql, "结果")
sql_xquery("连接 ID", sql, "结果")
sql_pvquery("连接 ID", sql, "结果")
```

上述三个函数作用一样，只是返回值和处理方法不同。

1.函数返回值

sql_query、sql_xquery、sql_pvquery的返回值如下。

- -1: 参数或查询出错。
- 1: 执行成功，如果是SELECT查询，则至少返回一行。
- 2: 执行成功，如果是SELECT查询，则没有任何行返回。

2.sql_query

如果sql_query函数执行成功，则可以通过\$dbr获取结果。\$dbr是一个伪变量，它的语法如下。

```
$dbr {result=>key}
```

其中，result为sql_query的第三个参数，key取值如下。

- rows: 行数。
- cols: 列数。
- [row, col]: 获取对应行、列上的值。
- colname[N]: 第 N 列的名称。

3.sql_xquery

sql_xquery函数允许通过列名获取结果。如果执行成功，则可以通过AVP列获取结果，其中，AVP的名称是函数的第三个参数，举例如下。

```
sql_xquery('example', sql, 'result')
```

获取第 i 行的方法如下。

```
$xavp(result[i])
```

获取第 i 行且列名为col_name的值的方法如下。

```
$xavp(result[i]=>col_name)
```

为了方便获取查询结果，我们定义了如下两个函数。

```

function xdb_get_row(xavp, irow) -- 获取一行数据
    return KSR.pv.get("$xavp(" .. xavp .. "[" .. tostring(irow) .. "])")
end

function xdb_gete_col(xavp, irow, col_name) -- 在行上根据列名获取数据
    return KSR.pv.gete("$xavp(" .. xavp .. "[" .. tostring(irow) .. "]>" ..
        col_name .. ")")
end

```

注意，在上述情况下，将会生成很多不同的AVP，我们曾在4.4.1节介绍伪变量静态名称限制时讨论过这个问题。但在上述情况下，一般来说返回的行数和列数都是有限的，生成的AVP数量也是有限的，占用内存不会很大，因而是可以接受的。

下面是一个进行数据库查询的例子，解读详见里面的注释。

```

function ksr_request_route()
    ksr_register_always_ok()

    local sql = "SELECT * FROM version"

    -- 查询example连接的数据库，结果放入version中
    local ok = KSR.sqlops.sql_query("example", sql, "version");
    local nrows = tonumber(KSR.pv.get("$dbr(version=>rows)"))
    local ncols = tonumber(KSR.pv.get("$dbr(version=>cols)"))
    KSR.info("number of rows in table: " .. nrows .. "\n");
    local i = 0
    local j = 0
    -- 打印列名，注意，i 和 j 都是 C 语言中的下标，因而从 0 开始，而不像 Lua 中从 1 开始
    while j < ncols do
        KSR.info("col#" .. j .. ":" .. KSR.pv.gete("$dbr(version=>colname[" .. j .. "])") .. "\n")
    end
end

```

```
j = j + 1
end
-- 遍历打印所有行和列
while i < nrows do
    j = 0
    while j < ncols do
        KSR.info("#" .. i .. "-" .. j .. ":" .. KSR.pv.get("$dbr(version=>[" ..
            i .. "," .. j .. "])") .. "\n")
        j = j + 1
    end
    i = i + 1
end

-- 释放内存
KSR.sqlops.sql_result_free("version")

-- 第二种查询方式
ok = KSR.sqlops.sql_xquery("example", sql, "version")
if ok >= 0 then
    KSR.info('xquery sql ok\n')
    KSR.sqlops.sql_result_free("version")
    i = 0
    -- 循环打印所有行和列
    while xdb_get_row("version", i) do
        KSR.info("id=" .. xdb_gete_col("version", i, "id") ..
            " table_name=" .. xdb_gete_col("version", i, "table_name") ..
            " table_version=" .. xdb_gete_col("version", i, "table_version") ..
            "\n")
        i = i + 1
    end
else
    KSR.error("SELECT ERROR " .. ok .. "\n")
end

-- 在实际场景下可以根据数据库中的查询结果进行路由
-- 但这里我们只是演示数据库的查询，因而简单返回 404。
KSR.sl.sl_send_reply(404, "Not Found")
end
```

7.4 其他函数和伪变量

Kamailio还提供了一些其他的函数和伪变量用于与数据库交互。

1.sql_pvquery

sql_pvquery函数可以将查询结果存入多个伪变量中，可以是任何可以写入（非只读）的伪变量，如\$var、\$avp、\$xavp、\$ru、\$du、\$sht等。

sql_pvquery的简单使用示例如下，详细使用方法可以参阅相关文档。

```
KSR.sqlops.sql_pvquery("example", "select 'col1', 2, NULL, 'sip:test@example.com'",  
    "$var(a), $avp(col2), $xavp(item[0]>s), $ru")
```

2.sql_query_async

sql_query_async函数可以异步执行SQL。该函数需要底层数据库模块的支持，db_mysql模块已支持



sql_query_async, db_postgres模块目前还不支持 sql_query_async。异步执行的SQL会在独立的进程中执行（必须设置async_works核心参数以便支持异步操作），因而无法获取结果，其主要用在往数据库里插入数据之类的场景。使用示例如下。

```
sql = "INSERT INTO version (default, 'TEST-TEST', 0)"  
KSR.sqlops.sql_query_async("example", sql)
```

3.\$sqlrows

\$sqlrows是一个伪变量，可以获取SQL影响的行数（主要是受INSERT、UPDATE、DELETE等影响的数据库中的行数）。使用示例如下。

```
local sql = "INSERT INTO ...."  
KSR.sqlops.sql_query("example", sql, "result")  
KSR.pv.get("$sqlrows(example)")
```

7.5 常用数据库表结构

在Kamailio源代码的utils/kamctl/目录下，可以找到不同数据库的SQL定义的目录，如postgres、mysql等。以postgres目录为例，下列命令只列出了目录中的前10个文件。

```
ls | head  
  
acc-create.sql  
alias_db-create.sql  
auth_db-create.sql  
avpops-create.sql  
carrieroute-create.sql  
cpl-create.sql  
dialog-create.sql  
dialplan-create.sql  
dispatcher-create.sql  
domain-create.sql
```

Kamailio中有一个特殊的version表，用于记录各模块的版本号，它是在standard-create.sql中定义的，建表语句如下。

```
CREATE TABLE version (  
    id SERIAL PRIMARY KEY NOT NULL,  
    table_name VARCHAR(32) NOT NULL,  
    table_version INTEGER DEFAULT 0 NOT NULL,  
    CONSTRAINT version_table_name_idx UNIQUE (table_name)  
);  
  
INSERT INTO version (table_name, table_version) values ('version','1');
```

每个模块相关的表在创建后都会向version表插入一条记录。如果在模块加载时，模块代码中的版本号和数据库中的版本号不匹配，则该模块会被拒绝加载。作为参考，常用的dispatcher表的创建语句如下。

```
CREATE TABLE dispatcher (  
    id SERIAL PRIMARY KEY NOT NULL,  
    setid INTEGER DEFAULT 0 NOT NULL,  
    destination VARCHAR(192) DEFAULT '' NOT NULL,  
    flags INTEGER DEFAULT 0 NOT NULL,  
    priority INTEGER DEFAULT 0 NOT NULL,  
    attrs VARCHAR(128) DEFAULT '' NOT NULL,  
    description VARCHAR(64) DEFAULT '' NOT NULL  
);  
  
INSERT INTO version (table_name, table_version) values ('dispatcher','4');
```

此外，上述语句在标准的Kamailio的安装目录下也能找到，如/usr/share/kamailio/postgres/。以下网址提供了更详细的说明，但可能跟你使用的版本不一致，故仅供参考：
<https://www.kamailio.org/docs/db-tables/kamailio-db-devel.html>。

具体数据库的使用方法大部分跟使用的模块相关，我们将在后面随模块一起介绍。

Chapter 8

第8章

15个典型的路由示例

有了前面的基础，本章我们来看更多的路由示例。这些示例并不能归为同一类，而是使用了不同的模块，颇具代表性。为便于理解，我们仅展示其中最关键、最本质的部分，略去一些对错误处理状态的检查。比如，有些示例只能转发客户端方向发来的呼叫，对FreeSWITCH侧发来的呼叫没有进行识别和处理；有些示例只能转发INVITE请求，对CANCEL和BYE消息没有进行处理。Kamailio是一个“有状态”的系统，SIP本身就有许多“状态”，在实际应用中还是要像我们在2.3节讲过的那样使用完整的脚本。本章只专注于介绍独立的模块和应用场景。注意，有些示例使用了一些明显很假的IP地址，如1.2.3.4、5.6.7.8等，在实际练习的时候要换成你自己的IP地址。

8.1 通过号码分析树进行路由

在通信领域，常见的路由选择都是根据被叫字冠进行的。Kamailio有一个mtree模块，该模块提供一个树形数据结构，将被叫号码放到内存中树的节点上，可以使查询速度非常快，这种做法适合做字冠匹配。mtree模块也需要一个数据库模块配合，数据最初存储在数据库里，在系统启动时把数据从数据库读入内存中。后续如果数据库中的数据有更新，也可以通过命令行重新加载数据。

mtree模块配置如下。

```
loadmodule "mtree"
modparam("mtree", "db_url", DBURL)      # 数据库 URL
modparam("mtree", "db_table", "mtrees") # 数据库表的名称
```

相关的建表SQL语句如下。

```
CREATE TABLE mtree (
    id SERIAL PRIMARY KEY NOT NULL,          -- ID, 自增长
    tprefix VARCHAR(32) DEFAULT '' NOT NULL,   -- 字冠
    tvalue VARCHAR(128) DEFAULT '' NOT NULL,    -- 对应的内容，可以放任何字符串
    CONSTRAINT mtree_tprefix_idx UNIQUE (tprefix) -- 索引
);

CREATE TABLE mtrees (
    id SERIAL PRIMARY KEY NOT NULL,          -- ID, 自增长
    tname VARCHAR(128) DEFAULT '' NOT NULL,   -- 树的名称
    tprefix VARCHAR(32) DEFAULT '' NOT NULL,   -- 字冠
    tvalue VARCHAR(128) DEFAULT '' NOT NULL,    -- 对应的内容，可以放任何字符串
    CONSTRAINT mtrees_tname_tprefix_tvalue_idx UNIQUE (tname, tprefix, tvalue)
);
```

在本例中我们仅使用mtrees表，往数据库中插入以下两条数据。

```
INSERT INTO mtrees (tname, tprefix, tvalue) VALUES ('mytree', '9',
'sip:1.2.3.4:5060');
INSERT INTO mtrees (tname, tprefix, tvalue) VALUES ('mytree', '9196',
'sip:5.6.7.8:5060');
```

重启Kamailio，或使用kamcmd mtree.reload命令让mtree模块重读数据库。这样数据就会被读入内存，后续在路由脚本中就不需要再访问数据库了，这保证了查询高效。可以使用如下命令进行验证。

```
# kamcmd mtree.match mytree s:9 0
mytree
{
    PREFIX: 9
    TVALUE: sip:1.2.3.4:5060
}

# kamcmd mtree.match mytree s:91 0
mytree
{
    PREFIX: 91
    TVALUE: sip:1.2.3.4:5060
}

# kamcmd mtree.match mytree s:9196 0
mytree
{
    PREFIX: 9196
    TVALUE: sip:5.6.7.8:5060
}
```

其中，命令行中的“s:”表示后面的第一个值是一个字符串，最后一个值表示匹配模式，这些内容下面会讲到。

mtree使用Longest Matching Prefix算法进行路由匹配，即优先匹配最长的字冠前缀。从上面的命令输出中也可以看出，当输入是9196时，匹配到了9196那条路由，其他的情况都只匹配到了9那一条。

路由脚本示例如下（mtree.lua）。

```
function ksr_request_route()
    ksr_register_always_ok()

    local rU = KSR.pv.get("SrU")           -- 获取 Request User 部分，即被叫号码
    matched = KSR.mtree.mt_match("mytree", rU, 0) -- 到树中查找是否有匹配的字冠
    KSR.info("rU = " .. rU .. " matched = " .. matched .. "\n") -- 打印日志
    if matched == 1 then                  -- 若匹配
        KSR.info("matched value: " .. KSR.pv.get("Savp(s:tvalue)") .. "\n") -- 打印日志
        KSR.forward_uri(KSR.pv.get("Savp(s:tvalue)")) -- 根据匹配的内容路由
        KSR.x.exit()                         -- 退出
    else
        KSR.err("No match\n")
        KSR.sl.sl_send_reply(404, "Not Found") -- 返回 404 Not Found
    end
end
```

如果路由匹配成功，则匹配到的值默认会存到\$avp(s:tvalue)这个PV中（或由模块的pv_value参数指定的AVP）。本例我们只是简单存储了一个URI字符串，在实际使用时可以存任何有意义的字符串。

mt_match(mtree, pv, mode)函数的参数说明如下。

- **mtree:** 字符串，树的名字，数据库和对应的内存中可以存放多棵树。
- **pv:** 字符串，表示要匹配的值，如主叫号码、被叫号码等。
- **mode:** 匹配模式。取值为0或2，其中0表示结果PV中存放匹配的结果；2表示结果PV是一个数组（默认为tvalues），存放所有匹配的结果，其中第[0]个位置存放匹配的最长的那个值。

更多参数和使用方法可以参阅相关文档：

<https://kamailio.org/docs/modules-devel/modules/mtree.html>。

8.2 号码翻译

本例我们使用dialplan模块。该模块主要提供号码匹配和翻译功能。该模块的相关配置如下。

```
loadmodule "dialplan"
modparam("dialplan", "db_url", DBURL)          # 数据库 URL
modparam("dialplan", "attrs_pvar", "$avp(dp_attrs)") # 属性对应的 PV
```

对应的数据库建表语句如下。

```
CREATE TABLE dialplan (
    id SERIAL PRIMARY KEY NOT NULL, -- ID, 自增长
    dpid INTEGER NOT NULL,          -- dialplan ID
    pr INTEGER NOT NULL,           -- 优先级
    match_op INTEGER NOT NULL,      -- 规则的匹配方式, 0 表示完全相等, 1 表示正则表达式
    match_exp VARCHAR(64) NOT NULL, -- 匹配表达式 (正则表达式或者字符串)
    match_len INTEGER NOT NULL,     -- 匹配长度
    subst_exp VARCHAR(64) NOT NULL, -- 替换表达式
    repl_exp VARCHAR(256) NOT NULL, -- 替换表达式 (类似 sed)
    attrs VARCHAR(64) NOT NULL     -- 匹配成功后的属性, 可以通过 $avp(dp_attrs) 访问
);
```

往数据库中插入下面的数据，每一个数据代表一条翻译规则。

```
INSERT INTO dialplan (dpid, pr, match_op, match_exp, match_len, subst_exp,
                      repl_exp, attrs)
VALUES (1, 0, 0, '102', 0, '', '212341234', 'equal match');

INSERT INTO dialplan (dpid, pr, match_op, match_exp, match_len, subst_exp,
                      repl_exp, attrs)
VALUES (2, 0, 1, '0([0-9]{8})', 0, '0([0-9]{8})', '\1', 'regexp match');
```

在数据库中运行“SELECT * FROM dialplan;”，会看到如下数据。

id	dpid	pr	match_op	match_exp	match_len	subst_exp	repl_exp	attrs
1	1	0	0	102	1	0	212341234	equal match
2	2	0	1	0([0-9]{8})	0	0([0-9]{8})	\1	regexp match

(2 rows)

重启Kamailio，或使用kamcmd dialplan.reload命令让dialplan模块重读数据库，然后可以使用下列命令进行验证。

```
# kamcmd dialplan.dump 1          # 导出 DPID 为 1 的记录
{
    DPID: 1
    ENTRIES: {
        ENTRY: {
            PRIO: 0
            MATCHOP: 0      # 完全匹配
            MATCHEXP: 102
            MATCHLEN: 0
            SUBSTEXP: <null string>
            REPLEXP: 212341234
            ATTRS: equal match
        }
    }
}
```

在命令行上执行如下号码翻译测试程序。

```
# kamcmd dialplan.translate 1 s:102
{
    Output: 212341234
    Attributes: equal match
}
```

由上面的输出可以看到，号码102被翻译成了212341234。

接下来我们看DPID为2的记录。

```
# kamcmd dialplan.dump 2
{
    DPID: 2
    ENTRIES: [
        ENTRY: [
            PRIO: 0
            MATCHOP: 1 # 采用正则表达式匹配

            MATCHEXP: 0([0-9]{8}) # 正则表达式
            MATCHLEN: 0
            SUBSTEXP: 0([0-9]{8}) # 执行替换的正则表达式，括号里面的内容可以引用
            REPLEXP: \1 # 替换结果引用，此处表示使用第一个匹配结果
            ATTRS: regexp match
        ]
    ]
}
```

使用上述翻译规则的翻译示例如下。

```
# dialplan.translate 2 s:012345678
{
    Output: 12345678
    Attributes: regexp match
}
```

上面的例子中，号码012345678先按照MATCHEXP的配置进行0([0-9]{8})的匹配，匹配成功后就跟MATCHEXP没什么关系了。接下来做替换操作，替换要依据SUBSTEXP和REPLEXP进行。现在SUBSTEXP配置的值是0([0-9]{8})，号码012345678拆分成了两个部分，即0和第一个括号里的内容12345678；REPLEXP配置的值是\1，也就是引用第一个括号里的内容，这样最后得到的结果是12345678，也就是删除了前缀0，俗称“吃掉”0。

路由脚本示例如下（dialplan.lua）。

```
function ksr_request_route()
    ksr_register_always_ok()
    -- 执行dialplan替换，结果存到 $var(new_rU) 中
    rc = KSR.dialplan.dp_replace(2, KSR.pv.get("SrU"), "$var(new_rU)")
    if rc == 1 then -- 替换成功
        KSR.info("dp_attr = " .. KSR.pv.get("Savp(dp_attrs)") .. "\n") -- 打印日志
        KSR.pv.sets("SrU", KSR.pv.get("Svar(new_rU)")) -- 设置新的 Request User
        KSR.pv.sets("StU", KSR.pv.get("Svar(new_rU)")) -- 设置新的 To User
        KSR.pv.sets("Stn", KSR.pv.get("Svar(new_rU)")) -- 设置新的 To User 名称
        KSR.pv.sets("Sdu", FS1_URI) -- 固定转发到 FreeSWITCH 1
        KSR.tm.t_relay() -- 执行路由转发
    else -- 匹配失败则返回空号
        KSR.sl.sl_send_reply(404, "Not Found")
        KSR.x.exit()
    end
end
```

dialplan的匹配也是在内存中进行的，脚本运行过程中不会访问数据库，因而执行起来非常快。

8.3 低成本路由

本例使用lcr模块。lcr是Least Cost Routing的简称，即低成本路由。该模块可以根据配置，在要到达的目的地有多个可选网关时，选择成本（以优先级和权重体现）最低的那个。配置如下。

```
loadmodule "lcr.so"
modparam("lcr", "db_url", DBURL)
modparam("lcr", "rule_id_avp", "$avp(lcr_ruleid)")      # 规则 ID 对应的 AVP
modparam("lcr", "gw_uri_avp", "$avp(lcr_gwuri)")        # 网关 URI 对应的 AVP
modparam("lcr", "ruri_user_avp", "$avp(lcr_ruri_user)") # Request URI 对应的用户 AVP
```

相关数据库建表语句如下。

```
CREATE TABLE lcr_rule (
    id SERIAL PRIMARY KEY NOT NULL,                      -- lcr 规则表
    lcr_id SMALLINT NOT NULL,                            -- ID, 自增长
    prefix VARCHAR(16) DEFAULT NULL,                     -- 字冠
    from_uri VARCHAR(64) DEFAULT NULL,                   -- From URI
    request_uri VARCHAR(64) DEFAULT NULL,                -- R-URI
    mt_tvalue VARCHAR(128) DEFAULT NULL,                 -- 保存 mtree 的值
    stopper INTEGER DEFAULT 0 NOT NULL,                  -- 是否停止后面的解析
    enabled INTEGER DEFAULT 1 NOT NULL,                  -- 是否启用后面的解析
    CONSTRAINT lcr_rule_lcr_id_prefix_from_uri_idx UNIQUE (lcr_id, prefix, from_uri)
);

CREATE TABLE lcr_rule_target (
    id SERIAL PRIMARY KEY NOT NULL,                     -- lcr 路由目的地表
    lcr_id SMALLINT NOT NULL,                           -- lcr 路由 ID
    rule_id INTEGER NOT NULL,                          -- 规则 ID
    gw_id INTEGER NOT NULL,                           -- 网关 ID
    priority SMALLINT NOT NULL,                        -- 优先级
    weight INTEGER DEFAULT 1 NOT NULL,                 -- 权重
    CONSTRAINT lcr_rule_target_rule_id_gw_id_idx UNIQUE (rule_id, gw_id)
);

CREATE TABLE lcr_gw (
    id INTEGER PRIMARY KEY NOT NULL,                  -- lcr 网关表
    lcr_id SMALLINT NOT NULL,                         -- ID, 自增长
    gw_name VARCHAR(128),                            -- 网关名称
    ip_addr VARCHAR(50),                            -- IP 地址
    hostname VARCHAR(64),                           -- 城名
    port SMALLINT,                                 -- 端口号
    params VARCHAR(64),                            -- 其他参数
    uri_scheme SMALLINT,                           -- URI 类型: 1 = sip; 2 = sips
    transport SMALLINT,                           -- SIP 传输方式: 1 = UDP; 2 = TCP; 3 = TLS;
                                                -- 4 = SCTP
    strip SMALLINT,                                -- 跟随变量 $prid 的含义完全一样
    prefix VARCHAR(16) DEFAULT NULL,                -- 是否在开头加前缀
    tag VARCHAR(64) DEFAULT NULL,                  -- 字冠
    flags INTEGER DEFAULT 0 NOT NULL,               -- 标签
    defunct INTEGER DEFAULT NULL,                  -- 标志
                                                -- 失效标志, 是一个 UNIX 时间戳
);

```

lcr模块涉及三张表，其中lcr_rule定义id跟prefix的对应关系；lcr_rule_target定义rule_id跟gw_id的对应关系；lcr_gw定义网关的详细参数，主要包括IP地址、端口及hostname等，其中hostname对应匹配后的路由设置规则，简述如下。

□若hostname为空，那么结果中\$du为空，\$ru指向IP地址（对应上述ip_addr字段）。

□若hostname非空，那么\$du指向IP地址（等于配置了呼出代理），而\$ru则指向hostname。

现在我们插入如下测试数据。

```

INSERT INTO lcr_rule (lcr_id, prefix, from_uri) VALUES (1, '44', NULL);
INSERT INTO lcr_rule (lcr_id, prefix, from_uri) VALUES (1, '442', NULL);
INSERT INTO lcr_rule (lcr_id, prefix, from_uri) VALUES (1, '443', NULL);
INSERT INTO lcr_rule (lcr_id, prefix, from_uri) VALUES (1, '49', NULL);
INSERT INTO lcr_rule (lcr_id, prefix, from_uri) VALUES (1, '49800', NULL);
INSERT INTO lcr_rule (lcr_id, prefix, from_uri) VALUES (1, '491', NULL);

INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 1, 1, 10, 1);
INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 2, 2, 10, 1);
INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 3, 2, 10, 1);
INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 4, 2, 10, 1);
INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 5, 1, 10, 1);
INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 5, 3, 20, 1);
INSERT INTO lcr_rule_target (lcr_id, rule_id, gw_id, priority, weight) VALUES
(1, 6, 1, 10, 1);

INSERT INTO lcr_gw (lcr_id, gw_name, ip_addr, hostname, port, params, uri_scheme)
VALUES (1, 'gw1', '10.1.1.101', NULL, 5060, NULL, 1);
INSERT INTO lcr_gw (lcr_id, gw_name, ip_addr, hostname, port, params, uri_scheme)
VALUES (1, 'gw2', '10.1.1.102', 'gw.com', 5070, NULL, 1);
INSERT INTO lcr_gw (lcr_id, gw_name, ip_addr, hostname, port, params, uri_scheme)
VALUES (1, 'gw3', '10.1.1.103', NULL, 5060, NULL, 1);

```

运行下面的SELECT查询，可以看得比较清楚。

```

SELECT * from lcr_rule;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | lcr_id | prefix | from_uri | request_uri | mt_tvalue | stopper | enabled |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | 1 | 44 | NULL | NULL | NULL | 0 | 1 |
| 2 | 1 | 442 | NULL | NULL | NULL | 0 | 1 |
| 3 | 1 | 443 | NULL | NULL | NULL | 0 | 1 |
| 4 | 1 | 49 | NULL | NULL | NULL | 0 | 1 |
| 5 | 1 | 49800 | NULL | NULL | NULL | 0 | 1 |
| 6 | 1 | 491 | NULL | NULL | NULL | 0 | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+

SELECT * from lcr_rule_target;
+-----+-----+-----+-----+-----+-----+
| id | lcr_id | rule_id | gw_id | priority | weight |
+-----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1 | 10 | 1 |
| 2 | 1 | 2 | 2 | 10 | 1 |
| 3 | 1 | 3 | 2 | 10 | 1 |
| 4 | 1 | 4 | 2 | 10 | 1 |
| 5 | 1 | 5 | 1 | 10 | 1 |
| 6 | 1 | 5 | 3 | 20 | 1 |
| 7 | 1 | 6 | 1 | 10 | 1 |
+-----+-----+-----+-----+-----+-----+

SELECT * from lcr_gw; -- (此处省略了部分列以方便排版)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | lcr_id | gw_name | ip_addr | hostname | port | params | transport | strip | prefix |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | 1 | gw1 | 10.1.1.101 | NULL | 5060 | NULL | NULL | NULL | NULL |
| 2 | 1 | gw2 | 10.1.1.102 | gw.com | 5070 | NULL | NULL | NULL | NULL |
| 3 | 1 | gw3 | 10.1.1.103 | NULL | 5060 | NULL | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

路由脚本示例如下（lcr.lua）。

```

function ksr_request_route()
    ksr_register_always_ok()
    local rU = KSR.pv.get("SrU")           -- 获取 Request User 部分
    if KSR.lcr.load_gws(1, rU) == 1 then   -- 查询网关
        if KSR.lcr.next_gw() == 1 then     -- 根据找到的内容替换 R-URI, Dest URI 等
            KSR.info("ruleid = " .. KSR.pv.get("Savp(lcr_ruleid)") .. "\n") -- 打印
            KSR.info("$du = " .. KSR.pv.get("Sdu") .. "\n")
            KSR.info("$ru = " .. KSR.pv.get("Sru") .. "\n")
            KSR.tm.t_relay()               -- 发往下一站
            KSR.x.exit()
        end
    end
    KSR.sl.send_reply(503, "No available gateways") -- 未找到路由则返回错误
end

```

使用上述路由脚本，如果呼叫441000，则先查lcr_rule表，匹配第一条规则（rule_id为1）；再查lcr_rule_target表，rule_id1匹配的gw_id为1；最后查lcr_gw表，对应的网关IP地址是10.1.1.101，端口是5060。实际运行的结果是：ruleid为1，\$du为空，\$ru为sip:441000@10.1.1.101:5060。这跟上面的分析完全吻合。

如果呼叫442000，则先查lcr_rule表，rule_id1和rule_id2都符合，但是rule_id2的prefix更长，于是最终匹配的是rule_id2；再查lcr_rule_target表，rule_id2匹配的gw_id为2，请注意，gw2配置的hostname为gw.com，这会导致\$du和\$ru不一致。实际运行的结果是：ruleid为2，\$du为sip:10.1.1.102:5070，\$ru为sip:442000@gw.com。

如果呼叫49800123，匹配的rule_id是5，而lcr_rule_target为这个rule_id配置了两个网关，其中gw1的优先级是10，gw2的优先级是20。优先级的取值范围是0~255，值越小其优先级越高，所以，最后选中的网关只能是优先级更高的gw1。实际运行的结果是：ruleid为5，\$du为空，\$ru为sip:49800123@10.1.1.101:5060。

8.4 前缀路由

本节要讲的示例会用到prefix_route模块，该模块根据数据库中的一组字冠（prefix，也称号码前缀）进行路由，它在加载时会把数据库中的数据加载到内存的二叉树中，这样后续的查询会非常快。该模块是一个很老的模块，不过也可以部分支持KEMI，且该模块有一定的参考意义，因此我们把它列在这里。

运行kamdbctl create命令时不会自动创建prefix_route表，手动建表语句如下。

```
CREATE TABLE prefix_route (
    prefix  VARCHAR(64) NOT NULL DEFAULT '', -- 字冠
    route   VARCHAR(64) NOT NULL DEFAULT '', -- 路由
    comment VARCHAR(64) NOT NULL DEFAULT '' -- 注释
);
```

往表里面插入一些测试数据，如下所示。

```
INSERT INTO prefix_route (prefix, route, comment) VALUES ('010', 'BJ', 'BeiJing');
INSERT INTO prefix_route (prefix, route, comment) VALUES ('020', 'GZ', 'GuangZhou');
INSERT INTO prefix_route (prefix, route, comment) VALUES ('021', 'SH', 'ShangHai');
INSERT INTO prefix_route (prefix, route, comment) VALUES ('0535', 'YT', 'YanTai');
```

使用如下配置。

```
loadmodule "prefix_route.so"
modparam("prefix_route", "db_url", DBURL) # 数据库 URL
```

注意，prefix_route模块在匹配到相应的字冠后只能调用原生路由块中的路由，因此，我们需要在原生路由块中定义路由。不过，在原生路由块中，我们可以再通过lua_runstring()函数执行Lua中的路由块。比如，我们在book.cfg中添加如下路由。

```
route[BJ] {
    lua_runstring("ksr_prefix_route('BJ')");
}
route[GZ] {
    lua_runstring("ksr_prefix_route('GZ')");
}
route[SH] {
    lua_runstring("ksr_prefix_route('SH')");
}
route[YT] {
    lua_runstring("ksr_prefix_route('YT')");
}
```

prefix_route模块启动时自动读prefix_route表，并在kamailio.cfg（本书中使用的是book.cfg）中检查对应的路由。如果找不到相应的路由定义，就会报错。下面是一个报错的例子，仅供参考。

```
CRITICAL: prefix_route [prefix_route.c:72]: add_route(): route name 'SH' is not defined
```

Lua脚本和注释如下（prefix_route.lua）。

```

function ksr_request_route()
    ksr_register_always_ok()
    KSR.prefix_route.prefix_route_uri() -- 根据 Request URI 进行查找，如果找到就会触发相应的原生路由块
end

function ksr_prefix_route(prefix) -- 当被叫号码与我们配置的字冠匹配时，原生路由块又会调用该函数

    KSR.info("prefix = " .. prefix .. "\n") -- 打印匹配到的字冠：prefix
    if prefix == "BJ" then -- 根据 prefix 参数路由到不同的目的地
        KSR.pv.sets("$du", FS1_URI) -- 设置 Dest URI，路由下一跳
    elseif prefix == "GZ" then
        KSR.pv.sets("$du", "192.168.1.101:5060")
    elseif prefix == "SH" then
        KSR.pv.sets("$du", "192.168.1.102:5060")
    elseif prefix == "YT" then
        KSR.pv.sets("$du", "192.168.1.103:5060")
    else
        KSR.sl.sl_send_reply(404, "Not Found")
        return
    end
    KSR.rr.record_route()
    KSR.tm.t_relay() -- 转发到下一跳
end

```

现在呼叫010开头的号码，就会匹配到"BJ"路由块并自动路由到与FS1_URI对应的目的地，呼叫0535开头的号码路由到"YT"。

我们曾在6.6节讲过，在Lua脚本中调用原生路由块的例子，本例则是一个在Lua脚本中调用原生路由块，原生路由块又调用Lua脚本中的函数的例子。在原生路由块中除使用lua_runstring函数调用Lua脚本中的函数外，还可以使用lua_run函数调用，后者可以支持最多3个字符串函数参数，如上面原生脚本中的调用可以改写成以下形式（该函数只有一个参数"BJ"）：

```
lua_run("ksr_prefix_route", "BJ");
```

8.5 动态路由

本节要讲的示例会使用drouting模块，drouting的全称是Dynamic Routing，即动态路由，其支持动态选择最佳网关。其中LCR（低成本路由）特性可以看作它的一种特殊情况。该模块支持很多功能特性，具体如下。

- 路由：可以基于被叫字冠、主叫/组、时间、优先级进行路由。
- 号码处理：包括删号、插号、设置默认规则、呼入呼出处理、触发路由脚本。
- 失败处理：包括对串行转发失败、基于权重的负载均衡失败、随机选择网关失败等的处理。

在本例中我们使用以下配置。

```
loadmodule "drouting"
modparam("drouting", "ruri_avp", "$avp(dr_ruri)") # 将 R-URI 存入该 AVP
modparam("drouting", "db_url", DBURL) # 数据库 URL
```

本例中的数据结构如下。

```
CREATE TABLE dr_gateways (
    gwid SERIAL PRIMARY KEY NOT NULL, -- 网关表
    type INTEGER DEFAULT 0 NOT NULL, -- 网关 ID
    address VARCHAR(128) NOT NULL, -- 地址
    strip INTEGER DEFAULT 0 NOT NULL, -- 删号位数
    pri_prefix VARCHAR(64) DEFAULT NULL, -- 插号
    attrs VARCHAR(255) DEFAULT NULL, -- 属性
    description VARCHAR(128) DEFAULT '' NOT NULL -- 描述
);

CREATE TABLE dr_rules (
    ruleid SERIAL PRIMARY KEY NOT NULL, -- 路由规则表
    groupid VARCHAR(255) NOT NULL, -- 规则 ID
    prefix VARCHAR(64) NOT NULL, -- 组 ID
    timerec VARCHAR(255) NOT NULL, -- 字冠
    priority INTEGER DEFAULT 0 NOT NULL, -- 时间规则
    routeid VARCHAR(64) NOT NULL, -- 优先级
    gwlist VARCHAR(255) NOT NULL, -- 路由 ID
    description VARCHAR(128) DEFAULT '' NOT NULL -- 网关列表
);
```

我们现在来做一个相对简单的测试，首先往数据库中插入以下数据。

```
INSERT INTO dr_gateways (gwid, address, description) VALUES
(1, '192.168.1.100:5080', 'ivr');
INSERT INTO dr_gateways (gwid, address, description) VALUES
(2, '192.168.1.101:5080', 'fax');
INSERT INTO dr_gateways (gwid, address, description) VALUES
(3, '192.168.1.102:5080', 'conference');

INSERT INTO dr_rules (groupid, prefix, timerec, gwlist, routeid, description)
VALUES ('0', '1234', '', '1', 0, 'route to ivr');
INSERT INTO dr_rules (groupid, prefix, timerec, gwlist, routeid, description)
VALUES ('0', '5678', '', '2', 0, 'route to fax');
```

上面的规则是：如果被叫字冠是1234，那么路由到ivr；如果被叫字冠是5678，则路由到fax。

路由脚本示例如下（drouting.lua）。

```
function ksr_request_route()
    ksr_register_always_ok()
    KSR.info("Request URI = " .. KSR.pv.get("$ru") .. " before drouting\n")
    rc = KSR.drouting.do_routing("0") -- 执行drouting路由查找，组ID为0
    if rc == 1 then
        -- 执行成功
        KSR.info("Destination URI = " .. KSR.pv.get("'$ru") .. " after drouting\n")
        KSR.tm.t_relay() -- 转发
        KSR.x.exit()
    else
        KSR.sl.sl_send_reply(404, "Not Found")
        KSR.x.exit()
    end
end
```

此外，drouting模块还有很多有用的参数，具体如下。

□ **gwlist:** 可以放多个网关，比如"1,2"，第一个网关呼叫失败之后可以选择下一个网关继续呼叫。

□ **timerec:** 规则生效的时间段。它有一个特定的标准格式[\[1\]](#)，比如“20220101T-114900|00H05M|daily”，该字符串以|作为分隔符，第一段表示规则生效的起始时间，第二段表示持续时间，第三段表示频率。连起来就是从2022年1月1日早上11点49分起生效，生效时间持续5分钟，然后每天都在这个时间段生效。

关于drouting模块更多的参数和用法可以参考

<https://kamailio.org/docs/modules-devel/modules/drouting.html>。

8.6 缩位拨号

缩位拨号即使用比较短的号码代替难以记忆的很长的电话号码。speeddial模块提供了缩位拨号功能。本例使用如下配置。

```
loadmodule "speeddial"
modparam("speeddial", "db_url", DBURL) # 数据库 URL
```

本例中数据库建表语句如下。

```
CREATE TABLE speed_dial (
    id SERIAL PRIMARY KEY NOT NULL, -- 缩位拨号表
    username VARCHAR(64) DEFAULT '' NOT NULL, -- ID, 自增长
    domain VARCHAR(64) DEFAULT '' NOT NULL, -- 用户名
    sd_username VARCHAR(64) DEFAULT '' NOT NULL, -- 城
    sd_domain VARCHAR(64) DEFAULT '' NOT NULL, -- 缩位拨号用户名
    new_uri VARCHAR(255) DEFAULT '' NOT NULL, -- 缩位拨号城
    fname VARCHAR(64) DEFAULT '' NOT NULL, -- 新 URI
    lname VARCHAR(64) DEFAULT '' NOT NULL, -- 名字
    description VARCHAR(64) DEFAULT '' NOT NULL, -- 姓氏
    CONSTRAINT speed_dial_speed_dial_idx UNIQUE (username, domain, sd_domain,
        sd_username)
);
```

往数据库中插入下面的数据。

```
INSERT INTO speed_dial (username, sd_username, new_uri) VALUES ('1001', '10',
    'sip:666010@192.168.1.100');
INSERT INTO speed_dial (username, sd_username, new_uri) VALUES ('1001', '11',
    'sip:888011@192.168.1.100');
```

主叫1001如果拨打10，被叫号码就会变成新地址sip:666010@192.168.1.100，如果拨打11则会变成sip:888011@192.168.1.100。

路由脚本示例如下（speed-dial.lua）。

```
function ksr_request_route()
    ksr_register_always_ok()
    rc = KSR.speeddial.lookup("speed_dial") -- 缩位拨号查询翻译
    if rc == 1 then

        KSR.tm.t_relay() -- 查找成功则发到下一跳
        KSR.x.exit()
    else
        KSR.sl.sl_send_reply(404, "Not Found") -- 否则返回 404 Not Found
        KSR.x.exit()
    end
end
```

speeddial模块还有另外一个函数——lookup_owner，该函数接收一个额外的参数，可以传入一个字符串作为username，具体如下。

```
rc = KSR.speeddial.lookup_owner("speed_dial", KSR.pv.get("$fu"))
```

8.7 通过别名数据库路由

alias_db是用户别名模块，与usrloc模块不同，后者在内存里面查找用户的注册信息，而alias_db模块总是在数据库里面查找。它相当于给用户提供另一个可路由的别名。虽然在数据库中查找比较慢，但是进行数据更新操作比较简单（更新数据库就直接生效，而无须担心缓存和重载数据）。该模块可以在同一个路由脚本中查不同的表。

alias_db模块的配置如下。

```
loadmodule "alias_db"
modparam("alias_db", "db_url", DBURL)          # 数据库 URL
modparam("alias_db", "use_domain", MULTIDOMAIN) # 是否使用多租户
```

本例中数据库建表语句如下。

```
CREATE TABLE dbaliases (
    id SERIAL PRIMARY KEY NOT NULL,           -- 别名表
    alias_username VARCHAR(64) DEFAULT '' NOT NULL, -- ID, 自增长
    alias_domain VARCHAR(64) DEFAULT '' NOT NULL, -- 别名用户名
    username VARCHAR(64) DEFAULT '' NOT NULL,   -- 别名域
    domain VARCHAR(64) DEFAULT '' NOT NULL      -- 用户名
);
```

往数据库中插入以下测试数据。

```
INSERT INTO dbaliases (alias_username, username, domain)
VALUES ('200', '10000200', 'rts.xswitch.cn:20003;transport=tcp');
```

路由脚本示例如下（alias_db.lua）。

```
function ksr_request_route()
    ksr_register_always_ok()
    local ru = KSR.pv.get("$ru")           -- 获取 Request URI
    KSR.info("$ru = " .. ru .. " before alias db lookup\n")
    rc = KSR.alias_db.lookup("dbaliases") -- 表名称
    if rc == 1 then                      -- 查询成功
        local new_ru = KSR.pv.get("$ru")  -- 新的 Request URI (有的版本可能是 $du)
        KSR.info("$ru = " .. new_ru .. " after alias_db lookup\n")
        KSR.tm.t_relay()                -- 转发
        KSR.x.exit()
    else
        KSR.err("alias_db lookup failed\n") -- 查询失败
        KSR.sl.sl_send_reply(404, "Not Found")
    end
end
```

呼叫200，新的\$ru变成如下形式。

```
sip:10000200@rts.xswitch.cn:20003;transport=tcp
```

如果插入新的数据，会即时生效，而无须要重启Kamailio或重加载数据。alias_db模块适用于一些需要进行号码翻译的场合。

8.8 运营商路由

carrieroute模块是为运营商设计的，可以支持路由、负载均衡、黑名单等，功能非常强大。

carrieroute模块主要涉及三张表，数据库建表语句如下。

```
CREATE TABLE carrier_name (
    id SERIAL PRIMARY KEY NOT NULL,          -- 运营商名称表
    carrier VARCHAR(64) DEFAULT NULL        -- ID, 自增长
);
CREATE TABLE domain_name (                  -- 域名表
    id SERIAL PRIMARY KEY NOT NULL,          -- ID, 自增长
    domain VARCHAR(64) DEFAULT NULL         -- 域
);
CREATE TABLE carrieroute (                -- 运营商路由表
    id SERIAL PRIMARY KEY NOT NULL,          -- ID, 自增长
    carrier INTEGER DEFAULT 0 NOT NULL,       -- 运营商 ID
    domain INTEGER DEFAULT 0 NOT NULL,         -- 域
    scan_prefix VARCHAR(64) DEFAULT '' NOT NULL, -- 扫描字冠
    flags INTEGER DEFAULT 0 NOT NULL,          -- 标志
    mask INTEGER DEFAULT 0 NOT NULL,
    prob REAL DEFAULT 0 NOT NULL,
    strip INTEGER DEFAULT 0 NOT NULL,          -- 删号
    rewrite_host VARCHAR(255) DEFAULT '' NOT NULL, -- 重写主机地址
    rewrite_prefix VARCHAR(64) DEFAULT '' NOT NULL, -- 重写字冠前缀
    rewrite_suffix VARCHAR(64) DEFAULT '' NOT NULL, -- 重写后缀
    description VARCHAR(255) DEFAULT NULL       -- 描述
);

```

往数据库中插入以下测试数据，再运行SELECT语句。

```
INSERT INTO carrier_name VALUES (1, 'mobile');
INSERT INTO carrier_name VALUES (2, 'unicom');

INSERT INTO domain_name VALUES (1, 'prepaid');
INSERT INTO domain_name VALUES (2, 'postpaid');

INSERT INTO carrieroute VALUES (1, 1, 1, '137', 0, 0, 0.5, 0, '192.168.1.120:9999',
                               '', '', NULL);
INSERT INTO carrieroute VALUES (2, 1, 1, '137', 0, 0, 0.5, 0, '192.168.1.121:9999',
```

```

      '', '' , NULL);
INSERT INTO carrierroute VALUES (3, 1, 1, '', 0, 0, 1, 0, '192.168.1.122:9999', '',
      '', NULL);
INSERT INTO carrierroute VALUES (4, 2, 1, '133', 0, 0, 1, 0, '192.168.2.100:5060',
      '', '', NULL);
INSERT INTO carrierroute VALUES (5, 2, 1, '', 0, 0, 1, 0, '192.168.2.101:5060', '',
      '', NULL);

SELECT * FROM carrier_name;
+----+-----+
| id | carrier |
+----+-----+
| 1 | mobile |
| 2 | unicom |
+----+-----+

SELECT * FROM domain_name;
+----+-----+
| id | domain |
+----+-----+
| 1 | prepaid |
| 2 | postpaid|
+----+-----+

SELECT * FROM carrierroute; -- 为方便排版，省略无内容的列: rewrite_prefix, rewrite_sufix, description
+----+-----+-----+-----+-----+-----+-----+-----+
| id | carrier | domain | scan_prefix | flags | mask | prob | strip | rewrite_host |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 137 | 0 | 0 | 0.5 | 0 | 192.168.1.120:9999 |
| 2 | 1 | 1 | 137 | 0 | 0 | 0.5 | 0 | 192.168.1.121:9999 |
| 3 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 192.168.1.122:9999 |
| 4 | 2 | 1 | 133 | 0 | 0 | 1 | 0 | 192.168.2.100:5060 |
| 5 | 2 | 1 | 1 | 0 | 0 | 1 | 0 | 192.168.2.101:5060 |
+----+-----+-----+-----+-----+-----+-----+-----+

```

其中：

carrier_name表保存的是运营商的名称。

domain_name表里面的domain直译是域，上面的例子中一个域是prepaid（预付费），另外一个域是postpaid（后付费）。这里的域名跟DNS域名或者SIP域名没有任何关系，只是为了区分不同的类别。

carrieroute表保存的是该模块的路由规则，其中：

- 第一条规则是针对mobile这个运营商的（carrier=1），domain是prepaid（domain=1），呼叫137开头的号码，会自动路由到192.168.1.120:9999。
- 第二条规则除了rewrite_host不同之外，其他都与第一条一样。
- 前两条规则合起来看，就是呼叫137开头的号码，会平均分配到192.168.1.120:9999和192.168.1.121:9999，因为第一条规则和第二条规则的prob（概率）都是0.5。
- 第三条规则的作用是，呼叫其他字冠，并自动路由到192.168.1.122:9999。
- 第四条和第五条规则是为另外一个运营商（carrier=2）配置的路由规则。

carrieroute模块不支持KEMI，原生路由示例如下。

8.9 字冠域名翻译

本例使用pdt模块，pdt是Prefix-Domains Translation的缩写。该模块用于字冠和域的翻译，常用于通过DID查找号码所属域的场景。

本例中的pdt模块配置如下。

```
loadmodule "pdt"
modparam("pdt", "db_url", DBURL)      # 数据库 URL
modparam("pdt", "check_domain", 0)      # 检查域
modparam("pdt", "prefix", "0")          # 也可以不配置 prefix
```

本例中的数据结构如下。

```
CREATE TABLE pdt (
    id SERIAL PRIMARY KEY NOT NULL, -- ID, 自增长
    sdomain VARCHAR(255) NOT NULL,   -- Source domain, 即源域, 可以是*, 匹配任何域
    prefix VARCHAR(32) NOT NULL,     -- Request URI 中用户部分的字冠
    domain VARCHAR(255) DEFAULT '' NOT NULL, -- 翻译出来的域
    CONSTRAINT pdt_sdomain_prefix_idx UNIQUE (sdomain, prefix)
);
```

向数据库中插入以下数据。

```
INSERT INTO pdt (sdomain, prefix, domain) VALUES ('sip.xswitch.cn', '123',
    'rts.xswitch.cn');
INSERT INTO pdt (sdomain, prefix, domain) VALUES ('sip.xswitch.cn', '124',
    'rts.xswitch.cn');
```

对pd_translate(sdomain, rewrite_mode)函数的参数说明如下。

sdomain: 字符串类型，即Source domain，可以用KSR.pv.get("\$fd")获取，也可以是*（pdt表里面sdomain也应该同时设置为*）

rewrite_mode: 整型，改写模式，取值如下。

0：前缀与前导前缀一起被删除，应用于本例中，新的R-URI是sip:9001@rts.xswitch.cn。

1：仅删除前导前缀，应用于本例中，新的R-URI是sip:12391001@rts.xswitch.cn。

2：不改变R-URI的用户部分，应用于本例中，新的R-URI是sip:012391001@rts.xswitch.cn。

pdt模块可以按上面的规则把符合前缀的R-URI处理成新的R-URI，比如可以将sip:01239001@sip.xswitch.cn转换为sip:9001@rts.xswitch.cn。其中，号码01239001分为3个部分，即0、123和9001，简析如下。

0：对应pdt模块参数prefix。

123：对应pdt表中第一行里面的prefix。

9001：为前面两个prefix匹配后剩余的部分。

重启Kamailio，或使用kamcmd pdt.reload命令让pdt模块重读数据库表，然后就可以使用如下命令查看匹配规则了。

```
# kamcmd pdt.list
{
    SDOMAIN: sip.xswitch.cn
    RECORDS: (
        ENTRY: (
            DOMAIN: rts.xswitch.cn
            PREFIX: 123
        )
        ENTRY: (
            DOMAIN: rts.xswitch.cn
            PREFIX: 124
        )
    )
}
```

路由脚本示例如下。

```
function ksr_request_route()
    ksr_register_always_ok()
    if KSR.pdt.pd_translate(KSR.pv.get("$fd"), 0) == 1 then -- 翻译
        KSR.info("new ru: " .. KSR.pv.get("$ru") .. "\n")
        KSR.tm.t_relay()
    else
        KSR.sl.sl_send_reply(404, "Not Found")
        KSR.x.exit()
    end
end
```

8.10 用户注册和查询

本节要讲的示例中会用到registrar和usrloc模块。usrloc的全称是User Location，即用户位置。usrloc模块用于保存注册用户的位置信息（联系地址），并可以定期同步到数据库。

usrloc模块的一般配置如下（在使用前可以去掉kamailio.cfg中WITH_USRLOCDB前的注释）。

```
loadmodule "usrloc.so"

modparam("usrloc", "db_url", DBURL)          # 数据库 URL
modparam("usrloc", "db_mode", 2)                # 写到内存，并定期同步到数据库
modparam("usrloc", "timer_interval", 60)        # 定时器的周期，单位是秒
```

db_mode一般配置为2，推荐使用该模式，效率很高，所有修改都在内存中进行，且会定期把内存中的注册信息同步到数据库的location表中。Kamailio重启时也会自动把location表的内容读到内存。

一般来说usrloc模块会配合registrar模块一起使用，本例中我们使用如下路由脚本（location.lua）。

```
function ksr_request_route()
    if KSR.is_method_in("R") then
        KSR.registrar.save("location", 0);           -- 注册消息
        KSR.sl.sl_send_reply(200, "OK");             -- 写入 location 表
                                                -- 返回 200 OK
        KSR.x.exit()
    elseif KSR.is_method_in("I") then
        local rc = KSR.registrar.lookup("location"); -- 查 location 表
        if rc < 0 then -- 如果找不到则根据返回值进行出错处理
            KSR.tm.t_newtran();                      -- 启动一个新的事务
            KSR.sl.send_reply(404, "Not Found");      -- 返回 404 Not Found
            KSR.x.exit();                           -- 退出
        end
        KSR.tm.t_relay();                         -- 查到注册信息，转发
    end
end
```

通过usrloc模块向Kamailio注册，注册信息就会存入location表，可以使用kamcmd ul.dump查询内存中的注册信息。1~2分钟后，就可以使用SQL语句查询数据库中的注册信息（写入有一定延迟）了，比如下面的查询就可正常执行。

```
SELECT * FROM location;
```

此外，也可以把位置信息保存到redis-server中，配置如下。

```
loadmodule "db_redis.so"                                # 把 redis-server 当成数据库
modparam("db_redis", "schema_path", "/usr/share/kamailio/db_redis/kamailio")
modparam("db_redis", "keys", "location=entry;username&usrdom:username")
modparam("usrloc", "db_mode", 1) # 模式为 1. Redis 数据库跟 usrloc 模块的内存完全同步
modparam("usrloc", "db_url", "redis://kamailio:123456@127.0.0.1:6379/0")
```

其中，shema_path相当于SQL建表语句。由于Redis是一个根据键值存储的数据库，所以要进行映射。比如，本例中用到的location表的结构如下。

```
# cat /usr/share/kamailio/db_redis/kamailio/location
id/int,ruid/string,username/string,domain/string,contact/string,received/
string,path/string,expires/time,q/double,callid/string,cseq/int,last_modified/
time,flags/int,cflags/int,user_agent/string,socket/string,methods/int,instance/
```

```
string,req_id/int,server_id/int,connection_id/int,keepalive/int,partition/int,  
9
```

下面是注册后使用redis-cli查询注册记录的例子。

```
# SMEMBERS location::index::usrdom  
1) "location:usrdom::1001"  
  
# HGETALL location:entry::1001  
1) "q"  
2) "-1.000000"  
3) "server_id"  
4) "0"  
5) "path"  
6) ""  
7) "contact"  
8) "sip:1001@192.168.1.120:9999;transport=udp"  
省略更多输出.....
```

usrloc模块没有输出导出函数，但有两个RPC函数，具体如下。

- **ul.dump:** 打印所有注册记录的详细信息。如果记录数目少，用这个命令比较方便，但如果注册用户太多，则慎用。
- **ul.lookup:** 用于查找用户的注册信息，如kamcmd ul.lookup location 1001@example.com。

8.11 向外注册

在本书大多数示例中，我们采用的都是IP地址对IP地址的对接方式，这要求对端的SIP服务器信任我们Kamailio的IP地址。有时候，在跟一些运营商对接时，他们只能提供注册式的网关，即所有的呼叫请求都需要经过Challenge验证，我们的Kamailio必须注册到对方的服务器上才能接收来话。

Kamailio有uac模块，该模块可以让Kamailio作为一个客户端注册到对方的服务器上，从这一点上来说，跟在FreeSWITCH中添加一个网关类似。uac模块配置如下。

```
local_rport=yes

# loadmodule "uac.so"
modparam("uac", "restore_mode", "none")
modparam("uac", "reg_db_url", DBURL)          # 数据库 URL
modparam("uac", "reg_timer_interval", 10)        # 注册间隔
modparam("uac", "reg_retry_interval", 30)        # 重试间隔
modparam("uac", "reg_contact_addr", "KAM_IP_PUBLIC:KAM_SIP_PORT") # Contact IP 地址和端口
```

注意，之所以使用上述的local_rport=yes，是因为本例中我们是在NAT后面向公网的服务器注册，这样有助于NAT穿透，详见3.1.7节。

相关的建表语句如下。

```
CREATE TABLE uacreg (
    id SERIAL PRIMARY KEY NOT NULL,
    l_uuid VARCHAR(64) DEFAULT '' NOT NULL,           -- 注册用户唯一标志字符串
    l_username VARCHAR(64) DEFAULT '' NOT NULL,        -- 本地用户名
    l_domain VARCHAR(64) DEFAULT '' NOT NULL,          -- 本地域
    r_username VARCHAR(64) DEFAULT '' NOT NULL,        -- 远端用户名
    r_domain VARCHAR(64) DEFAULT '' NOT NULL,          -- 远端域
    realm VARCHAR(64) DEFAULT '' NOT NULL,            -- 域
    auth_username VARCHAR(64) DEFAULT '' NOT NULL,      -- 塞权用户名
    auth_password VARCHAR(64) DEFAULT '' NOT NULL,      -- 密码
    auth_ha1 VARCHAR(128) DEFAULT '' NOT NULL,          -- a1哈希码
    auth_proxy VARCHAR(255) DEFAULT '' NOT NULL,        -- 代理服务器，以 sip: 开头
    expires INTEGER DEFAULT 0 NOT NULL,                 -- 过期时间，秒
    flags INTEGER DEFAULT 0 NOT NULL,                   -- 注册标志
    reg_delay INTEGER DEFAULT 0 NOT NULL,               -- 注册延时
    contact_addr VARCHAR(255) DEFAULT '' NOT NULL,     -- 联系地址
    socket VARCHAR(128) DEFAULT '' NOT NULL,           -- 指定使用的 Socket，可以为空
    CONSTRAINT uacreg_l_uuid_idx UNIQUE (l_uuid)         -- 索引
);
```

可以使用以下命令添加注册记录。

```
kamcmd uac.reg_add s:1000 s:1000 seven.local s:1000 seven.local seven.local \s:1000
s:1234 . sip:kb-fsl:5070 900 0 6 . .

kamcmd uac.reg_add s:1001 s:1001 demo.xswitch.cn s:1001 demo.xswitch.cn demo,
xswitch.cn \ s:1001 s:1234 . sip:demo.xswitch.cn:10160 900 0 600 . .
```

注意，上述命令中的参数，如果是纯数字的字符串参数，则需要在前面加上“s:”。如果某个参数使用默认值，可以输入一个“.”占位。上述命令中的参数必须按如下顺序来（含义与数据库中的注释一致）排列：l_uuid、l_username、l_domain、r_username、r_domain、realm、auth_username、auth_password、auth_ha1、auth_proxy、expires、flags、reg_delay、contact_addr、socket。

也可以将注册信息插入数据库，具体实现如下。

```

INSERT INTO uacreg (l_uuid, l_username, l_domain, r_username, r_domain, realm,
auth_username, auth_password, auth_proxy, expires)
VALUES('1000', '1000', 'seven.local', '1001', 'seven.local', 'seven.local',
'1000', '1234', 'sip:kb-fsl:5070', 900);

INSERT INTO uacreg (l_uuid, l_username, l_domain, r_username, r_domain, realm,
auth_username, auth_password, auth_proxy, expires)
VALUES('1001', '1001', 'demo.xswitch.cn', '1001', 'demo.xswitch.cn', 'demo.xswitch.
cn', '1001', '1234', 'sip:demo.xswitch.cn:10160', 900);

```

插入数据库后需要重启Kamailio或使用kamcmd uac.reg_reload重载数据（注意，这将清除手动用uac.reg_add添加的记录）。使用下列命令可以查看注册状态。

```

kamcmd uac.reg_dump          # 列出所有注册信息
kamcmd uac.reg_info l_uuid s:1001 # 仅列出 l_uuid 为 1001 的注册信息

```

上述命令输出如下。

```

{
    l_uuid: 1001
    l_username: 1001
    l_domain: demo.xswitch.cn

    r_username: 1001
    r_domain: demo.xswitch.cn
    realm: demo.xswitch.cn
    auth_username: 1001
    auth_password: 1234
    auth_hai: none
    auth_proxy: sip:demo.xswitch.cn:10160
    expires: 900
    flags: 20
    diff_expires: 365
    timer_expires: 1651133927
    reg_init: 1651132958
    reg_delay: 60
    contact_addr: 127.0.0.1:5060
    socket: .
}

```

当Kamailio收到对端网关来话时，可以使用uac_reg_lookup()检查来话是否来自我们注册的网关，如果是，则根据需要进行转发。

```

function ksr_request_route()
    ksr_register_always_ok()
    if KSR.uac.uac_reg_lookup(KSR.pv.get("SrU"), "$ru") == 1 then
        KSR.info("网关来话 [" .. KSR.pv.get("Sou") .. " => " .. KSR.pv.get("Sru")
        .. "]\n");
        if KSR.registrar.lookup("location") < 0 then -- 查找本地是否有对应的注册用户
            KSR.sl.sl_send_reply("404", "Not Found")
        end
        KSR.rr.record_route();
        KSR.tm.t_relay()
    end
end

```

如果是本地注册用户来话，则发往对应的网关。首先，我们来看如何找到网关。在本例中，我们使用一对一的用户和网关，相当于每个本地用户都有一个对应的外线。

```

local fU = KSR.pv.gete("$fU")
local next_uri = ksr_get_next_uri(fU)          -- 根据 From User 查询对应的网关
if next_uri then                                -- 找到后根据网关绑定的地址进行路由
    KSR.info("next_uri: " .. next_uri .. "\n")
    KSR.pv.sets("$du", next_uri)
    KSR.rr.record_route();
    KSR.tm.t_on_failure("handle_trunk_auth")
    KSR.tm.t_relay()
end

```

其中，`ksr_get_next_uri()`是一个自定义函数，它会对uacreg表进行SQL查询，找到auth_proxy字段指定的地址。具体的查询方法可以参见随书附赠的代码以及第7章。

`KSR.tm.t_on_failure("handle_trunk_auth")`用于设置一个回调函数，并对网关回复的407消息进行认证。认证信息也需要从数据库中获取。具体代码如下。

```

function handle_trunk_auth()
    if KSR.tm.t_is_canceled() > 0 then
        return 1
    end
    local status_code = KSR.tm.t_get_status_code()

    if status_code == 407 then -- 需要认证
        local auth_username = KSR.pv.gete("$fU")
        -- 通过 From User 到 uacreg 表中查找相应的域及密码，以便进行鉴权，该函数也是一个自定义函数
        local realm, auth_password = ksr_get_auth_data(auth_username)
        -- uac_auth 函数需要这些信息进行鉴权，相应的 AVP 是可以通过 uac 模块参数进行设置
        KSR.pv.sets("$avp(arealm)", realm)
        KSR.pv.sets("$avp(auser)", auth_username)
        KSR.pv.sets("$avp(apasswd)", auth_password)
        if (KSR.uac.uac_auth() > 0) then -- 生成鉴权信息并鉴权
            if KSR.tm.t_relay() < 0 then
                KSR.sl.sl_reply_error()
            end
        else
            KSR.sl.sl_reply_error()
            KSR.x.exit()
        end
    end
end

```

当使用uac模块对呼叫进行鉴权时，默认CSeq不变，这可能造成鉴权失败（如FreeSWITCH会回复482 Merged消息），可以通过加载dialog模块并设置如下参数解决。

```
modparam("dialog", "track_cseq_updates", 1)
```

除此之外，还需要在发往下一跳之前用对话管理函数进行处理，以便能更新CSeq计数器。举例如下。

```
if KSR.is_INVITE() then KSR.dialog.dlg_manage() end
```

另外，在随书附赠的示例代码中，为了方便起见，FreeSWITCH没有配置鉴权。如果要测试本节的示例，最好修改FreeSWITCH配置文件，把鉴权打开。如修改/usr/local/freeswitch/conf/sip_profiles/default.xml，把以下两行注释掉。

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

在本例中，我们实现了uac向外注册，并实现了本地用户呼叫网关、网关呼叫本地用户，以及相应

的呼叫鉴权处理。受篇幅所限，这里没有列出所有代码，完整版可以参阅随书附赠的代码（uac.lua）。uac模块仅实现了向外注册和构造鉴权信息等，但应对呼叫鉴权时还需要提供相应的鉴权信息。本例中是使用查uacreg数据表实现的。如果每次呼叫都需要查数据库，可能会大大降低系统的吞吐量，在对性能要求比较高的场合也可以事先将相关数据缓存到内存中。关于缓存的实现在此我们就不举例了，感兴趣的读者可以自行尝试。

8.12 更多AVP示例

我们在3.2.3节讲过AVP的基本概念，这一节我们来看更多的示例，主要涉及avp、xavp、xavi、xavu等AVP变量及其用法。

提示

在Kamailio中，avp变量在SIP事务期间有效。如果想让其在整个对话期间都有效，可以使用\$dlg_var（需要先加载dialog模块）。

下面是一个使用avp的比较简单的例子。

```
KSR.pv.sets("$avp(y)", "Hello Kamailio") -- 设置字符串值  
KSR.info(KSR.pv.get("$avp(y)") .. "\n") -- 打印 Hello Kamailio
```

avp变量可以放多个值，先来看如下的例子。

```
KSR.pv.sets("$avp(x)", "one")  
KSR.pv.sets("$avp(x)", "two")  
KSR.pv.sets("$avp(x)", "three")  
  
KSR.info(KSR.pv.get("$avp(x)") .. "\n") -- three  
KSR.info(KSR.pv.get("$avp(x)[0]") .. "\n") -- three  
KSR.info(KSR.pv.get("$avp(x)[1]") .. "\n") -- two  
KSR.info(KSR.pv.get("$avp(x)[2]") .. "\n") -- one
```

avp变量的保存方式类似于堆栈，先进后出，赋新值不会覆盖旧值，最后放入的值的索引为[0]。

xavp是扩展的avp，xavp变量同样是在事务期间有效，但是支持更复杂的数据结构，并可按“索引=>字段名”形式来访问。举例如下。

```
KSR.pv.sets("$xavp(ua=>username)", "1001") -- 创建新的xavp  
KSR.pv.sets("$xavp(ua[0]=>password)", "1234") -- 字符串值  
KSR.pv.seti("$xavp(ua[0]=>expires)", 3600) -- 整数值  
  
KSR.pv.sets("$xavp(ua=>username)", "1002") -- 创建新的xavp，之前的那个ua变成了ua[1]  
KSR.pv.sets("$xavp(ua[0]=>password)", "6666")  
KSR.pv.seti("$xavp(ua[0]=>expires)", 600)  
  
KSR.info("username = " .. KSR.pv.get("$xavp(ua[0]=>username)") .. "\n") -- 1002  
KSR.info("password = " .. KSR.pv.get("$xavp(ua[0]=>password)") .. "\n") -- 6666  
KSR.info("expires = " .. KSR.pv.get("$xavp(ua[0]=>expires)") .. "\n") -- 600  
  
KSR.info("username = " .. KSR.pv.get("$xavp(ua[1]=>username)") .. "\n") -- 1001  
KSR.info("password = " .. KSR.pv.get("$xavp(ua[1]=>password)") .. "\n") -- 1234  
KSR.info("expires = " .. KSR.pv.get("$xavp(ua[1]=>expires)") .. "\n") -- 3600
```

xavi跟xavp类似，只是对键的大小写不敏感，i是insensitive（不敏感）的缩写。xavu跟xavp类似，但是值有唯一性，而且没有索引，u是unique（唯一）的缩写。

有些模块可以配置xavp参数，比如registrar模块、dispatcher模块等。sqlops是操作数据库的SQL模块，也可以将从数据库中读取的值存到xavp中，以便根据行号及字段名进行访问，参见7.3节。

下面再举一个registrar处理的例子。

在kamailio.cfg或book.cfg中增加如下配置。

```
modparam("registrar", "xavp_rcc", "ulrcd") # record 保存到 ulrcd 变量  
# record 的所有信息都要保存，包括 ruid(record 的主键). contact. expires. received. path 等  
modparam("registrar", "xavp_rcc_mask", 0)
```

路由脚本如下（xavp.lua）。

```
function ksr_request_route()  
    if KSR.registrar.save("location", 0)<0 then  
        KSR.si.sl_reply_error()  
        KSR.x.exit()  
    end  
    KSR.pvx.pv_xavp_print()  
    KSR.info("ruid = " .. KSR.pv.gete("$xavp(ulrcd[0]=>ruid)") .. "\n")  
    KSR.info("contact = " .. KSR.pv.gete("$xavp(ulrcd[0]=>contact)") .. "\n")  
    KSR.info("received = " .. KSR.pv.gete("$xavp(ulrcd[0]=>received)") .. "\n")  
    KSR.info("expires = " .. KSR.pv.getvn("$xavp(ulrcd[0]=>expires)", 0) .. "\n")  
end
```

运行上述脚本，就可以打印出Kamailio存放的位置信息。笔者注册、测试的部分日志如下。

```
xavx_print_list_content(): +++++ start XAVP list: 0x40025fb108 (0) (level=0)  
xavx_print_list_content():      *** (1:0 ~ 0x40025fb108) XAVP name: ulrcd  
xavx_print_list_content():      XAVP id: 2077578204  
xavx_print_list_content():      XAVP value type: 6  
xavx_print_list_content():      XAVP value: <xavp:0x40025fb060>  
xavx_print_list_content(): +++++ start XAVP list: 0x40025fb060 (0x40025fb128)  
    (level=1)  
xavx_print_list_content():      *** (1:1 ~ 0x40025fb060) XAVP name: expires  
xavx_print_list_content():      XAVP id: 1784956455  
... 遗略很多行 ...  
sr_kemi_core_info(): ruid = uloc-626a0761-1eb8-1  
sr_kemi_core_info(): contact = sip:1000@192.168.3.180:63908;transport=TCP;ob  
sr_kemi_core_info(): received =  
sr_kemi_core_info(): expires = 300
```

8.13 话单

本例使用acc模块，acc是Accounting（即记账）的简称，通俗来讲就是记话单。运行kamdbctl create 创建的数据库默认就包含了两张acc相关的表——acc表和missed_calls表。一般建议运行kamctl acc initdb给这两张表扩充字段，以使其记录的内容更加丰富。在默认的kamailio.cfg中也会有相应的扩展语句，也可以手动执行这些SQL语句来添加相应的列，相关的SQL内容如下。

```
ALTER TABLE acc ADD COLUMN src_user VARCHAR(64) NOT NULL DEFAULT '';
ALTER TABLE acc ADD COLUMN src_domain VARCHAR(128) NOT NULL DEFAULT '';
ALTER TABLE acc ADD COLUMN src_ip varchar(64) NOT NULL default '';
ALTER TABLE acc ADD COLUMN dst_user VARCHAR(64) NOT NULL DEFAULT '';
ALTER TABLE acc ADD COLUMN dst_user VARCHAR(64) NOT NULL DEFAULT '';
ALTER TABLE acc ADD COLUMN dst_domain VARCHAR(128) NOT NULL DEFAULT '';
ALTER TABLE missed_calls ADD COLUMN src_user VARCHAR(64) NOT NULL DEFAULT '';
ALTER TABLE missed_calls ADD COLUMN src_domain VARCHAR(128) NOT NULL DEFAULT '';
ALTER TABLE missed_calls ADD COLUMN src_ip varchar(64) NOT NULL default '';
ALTER TABLE missed_calls ADD COLUMN dst_user VARCHAR(64) NOT NULL DEFAULT '';
ALTER TABLE missed_calls ADD COLUMN dst_user VARCHAR(64) NOT NULL DEFAULT '';
ALTER TABLE missed_calls ADD COLUMN dst_domain VARCHAR(128) NOT NULL DEFAULT '';
```

在默认的kamailio.cfg中有如下配置。

```
loadmodule "acc.so"

modparam("acc", "early_media", 0)
modparam("acc", "report_ack", 0)
modparam("acc", "report_cancels", 0)
modparam("acc", "detect_direction", 0)
modparam("acc", "log_flag", FLT_ACC)          # 使用哪个Flag标志要进行记账。写入日志
modparam("acc", "log_missed_flag", FLT_ACMMISSED) # 使用哪个Flag标志未接的电话。写入日志
modparam("acc", "log_extra",
        "src_user=$fU;src_domain=$fd;src_ip=$si;"           # 可增加的额外的字段
        "dst_user=$tU;dst_user=$rU;dst_domain=$rd")      # 可增加的额外的字段
modparam("acc", "failed_transaction_flag", FLT_ACFAILED) # 使用哪个Flag标志失败的电话
modparam("acc", "db_flag", FLT_ACC)                # 写入数据库使用的Flag，一般与
                                                    # log_flag 相同
modparam("acc", "db_missed_flag", FLT_ACMMISSED) # 写入数据库使用的标志未接电话的Flag
modparam("acc", "db_url", DBURL)                  # 数据库 URL
modparam("acc", "db_extra",
        "src_user=$fU;src_domain=$fd;src_ip=$si;"           # 可增加的额外的字段，前提在上面
        "dst_user=$tU;dst_user=$rU;dst_domain=$rd")      # 可增加的额外的字段，前提在上面
                                                    # 扩展了数据表
```

如果想将记账信息异步插入数据库，那么可以把db_insert_mode参数配置为2，具体如下。

```
modparam("acc", "db_insert_mode", 2) # async insert
```

还需要在kamailio.cfg里面配置异步的Worker进程数，具体如下。

```
async_workers = 2 # 或者其他值
```

完整的路由脚本我们在2.3节已经详细讲过了，下面的脚本仅列出相关要点（acc.lua）。

```

FLT_ACC=1           -- 记账相关的标志，要跟 kamailio.cfg 中一致
FLT_ACCMISSED=2
FLT_ACCFAILED=3

function ksr_request_route()
    ksr_register_always_ok()

    if KSR.is_INVITE() or KSR.is_BYE() then -- 为了简单，呼叫开始和结束都记
        KSR.setflag(FLT_ACC) -- 设置事务标志，表示要记账（跟 acc 模块的 log_flag 参数对应起来）
        KSR.setflag(FLT_ACCFAILED) -- 如果呼叫失败，则同时保存到 acc 表和 missed_calls 表
    end

    KSR.rr.record_route();
    KSR.pv.sets("$du", FS1_URI)
    KSR.tm.t_relay()
end

```

执行上述脚本，打通电话后，就可以在日志中看到如下内容了（为排版方便，这里对换行位置进行了细微改动，下同）。

```

1(6982) NOTICE: LUA {INVITE}: acc [acc.c:287]: acc_log_request(): ACC: transaction
answered:

timestamp=1650987418;method=INVITE;from_tag=ROZ2Lz0WdQ1DRDQ0uLkSzolJjZsEY09Y;
to_tag=XrXaQUF72rHta;call_id=g0Qzw3zYzraQbLpb6.0B9kaVEzBb0gb9;code=200;reason=OK;
src_user=1001;src_domain=192.168.7.8;src_ip=172.22.0.1;dst_user=9196;dst_domain=192.168.7.8

```

挂机后看到如下内容。

```

2(6984) NOTICE: LUA {BYE}: acc [acc.c:287]: acc_log_request(): ACC: transaction
answered:

timestamp=1650987420;method=BYE;from_tag=ROZ2Lz0WdQ1DRDQ0uLkSzolJjZsEY09Y;
to_tag=XrXaQUF72rHta;call_id=g0Qzw3zYzraQbLpb6.0B9kaVEzBb0gb9;code=200;reason=OK;
src_user=1001;src_domain=192.168.7.8;src_ip=172.22.0.1;dst_user=9196;dst_domain=192.168.7.8

```

如果在kamailio.cfg中开启了WITH_ACCDB，则可以使用如下语句查询话单。

```
SELECT * FROM acc;
```

笔者测试的查询结果如下。

id	method	from_tag	to_tag	src_user
callid		sip_code	sip_reason	time
src_domain	src_ip	dst_user	dst_user	dst_domain
1	INVITE	QA7AFWKNTW3ktVZxZJak8ggZceLBuYRq	1v2Dy7jNQva5r	Br28Q2ng.
A7odJD7MESFokuSU93ZDdrw	200	OK	2022-04-26 15:39:44	1001
192.168.7.8	172.22.0.1	9196	9196	192.168.7.8
2	BYE	QA7AFWKNTW3ktVZxZJak8ggZceLBuYRq	1v2Dy7jNQva5r	Br28Q2ng.
A7odJD7MESFokuSU93ZDdrw	200	OK	2022-04-26 15:39:46	1001
192.168.7.8	172.22.0.1	9196	9196	192.168.7.8

{2 rows}

8.14 SBC

SBC（Session Border Controller，边界会话控制器）相当于一个SIP防火墙，部署在运营商侧以及企业侧的“边界”位置。一般来说，SBC如果部署在企业网，会部署在企业网的DMZ区；如果是双网卡，则一个网卡对外，一个网卡对内。这样是为了保护SIP安全。

SBC没有一个统一的规范，但通常都有防SIP攻击、限流、代理注册、代理媒体、协助NAT穿透、信令和媒体加解密转换、音视频编码转换等功能。

SBC可以是一个普通的SIP代理（Proxy），也可以是一个背靠背用户代理（B2BUA）。Kamailio可以作为一个SBC使用，配合rtpproxy或FreeSWITCH使用可以代理媒体，也可以分别实现Proxy和B2BUA功能。

这一节的内容和示例并不限于SBC，但它们或多或少都与SBC相关，因此我们统一将它们放到这里。

8.14.1 代理注册

Kamailio本身可以作为注册服务器使用，注册信息存储在Kamailio中。作为SBC使用时，Kamailio通常不带用户，而是将用户注册消息转发到其他SIP服务器进行处理，如图8-1所示。我们以FreeSWITCH为例，让它作为SIP服务器使用，而Kamailio就可以作为SBC，内网SIP终端直接注册在FreeSWITCH上，而外网SIP终端通过Kamailio（SBC）注册到FreeSWITCH上。

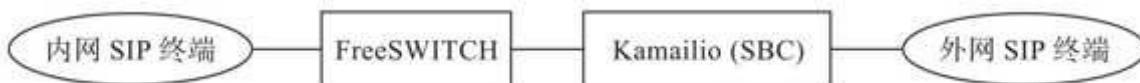


图8-1 SBC拓扑示意

我们知道，注册的作用主要是把Contact地址记录在注册服务器上，当注册用户做被叫时注册服务器就可以找到它。在图8-1所示的情况下，内网用户直接注册到FreeSWITCH，FreeSWITCH可以直接记住Contact地址，但当外网用户注册时，FreeSWITCH不仅要记住用户的Contact地址，还要记住Kamailio的地址（也可以预先设置），因为当内网用户呼叫外网用户时，SIP INVITE消息要经过Kamailio才能送达外网用户。

1. 使用path模块辅助代理注册

我们来看一个例子。这里要用到path模块，并使用如下配置。

```
loadmodule "path.so"
modparam("path", "use_received", 1) # 使用接收到的 SIP 客户端的来源地址而非 Contact 中的地址
```

在我们的路由脚本中，需要判断消息是从客户端来的还是从FreeSWITCH侧来的。在此使用dispatcher模块来做这个判断。dispatcher.list文件的内容如下。

```
100 sip:kamailio-fs1:5060
```

Lua脚本如下（path.lua）。

```

function ksr_request_route()
    if KSR.siputils.has_totag() < 0 then
        if KSR.is_OPTIONS() then
            KSR.sl.sl_send_reply(200, "Keepalive") -- 响应 OPTIONS 请求
            KSR.x.exit()
        else
            KSR.rr.record_route() -- 添加 Record-Route 头域
        end
    end

    if KSR.dispatcher.ds_is_from_list(100, 3) > 0 then -- 消息是从 FreeSWITCH 侧来的
        ksr_route_from_fs()
    else
        ksr_route_to_fs()
    end
end

function ksr_route_from_fs()
    KSR.tm.t_relay() -- 如果消息是从 FreeSWITCH 侧来的，直接路由即可，因为 R-URI 中包含了
    正确的信息
    KSR.x.exit()
end

function ksr_route_to_fs() -- 消息是从客户端来的

    if KSR.siputils.has_totag() < 0 then -- 首个 INVITE 消息或注册消息
        KSR.path.add_path_received() -- 关键函数，把消息源地址加入 Path 头域
        if KSR.dispatcher.ds_select_dst(100, 4) < 0 then -- 调用 dispatcher 选择 FreeSWITCH
            KSR.sl.send_reply(404, "No destination")
        else
            KSR.tm.t_relay() -- 路由转发
        end
        KSR.x.exit()
    else
        -- 处理对话内的 SIP 消息
        KSR.tm.t_relay()
        KSR.x.exit()
    end
end

```

运行上述脚本，使用客户端注册，在FreeSWITCH中可以看到，在注册消息中多了一个Path头域，具体如下。

```

recv 1017 bytes from udp/[172.22.0.3]:35060 to udp/[192.168.3.180]:5070 at 2022-
04-28 01:24:49.061884:
-----
REGISTER sip:seven.local;transport=tcp SIP/2.0
... 此处省略很多行 ...
Path: <sip:192.168.7.8:35060;lr;received=sip:172.22.0.1:64174%3Btransport%3Dtcp>

```

有了这个Path信息，当该用户做被叫时，FreeSWITCH会先把呼叫发送到Kamailio（上述Path中的sip:192.168.7.8:35060部分是Kamailio的地址）。

但这个add_path_received在此加入的是Kamailio的外网地址，而在笔者的Docker环境中Kamailio跟FreeSWITCH是通过内网对接的。如果FreeSWITCH向Kamailio发送消息时使用外网地址，则Kamailio将无法区分呼叫是来自FreeSWITCH还是来自客户端。事实上，使用上述脚本，当FreeSWITCH呼叫客户端时，可能会产生无限循环（Kamailio认为从FreeSWITCH侧来的呼叫是来自客户端，因而又发回FreeSWITCH）。当然，如果你使用host模式的Docker网络或不使用Docker，运行该示例是没有问题的。大部分的消息死循环都是由于Kamailio的配置不当导致判断错了呼叫来

 源引发的。

2. 手动设置Path头域

在此我们来看一个真实的案例，如图8-2所示。xswitch.cn是一个多租户的云通信平台，每个租户都有一个二级域名，每个租户运行一个Docker容器（里面是FreeSWITCH），客户端通过SBC注册到不同的租户上。客户端一般位于NAT后面。各服务器位于腾讯云上，每个服务器有对应的公网地址和私网地址（10.10.x.x段），DNS解析到公网地址上。容器使用NAT方式运行，因而容器内又有另一个地址（172段或192段）。

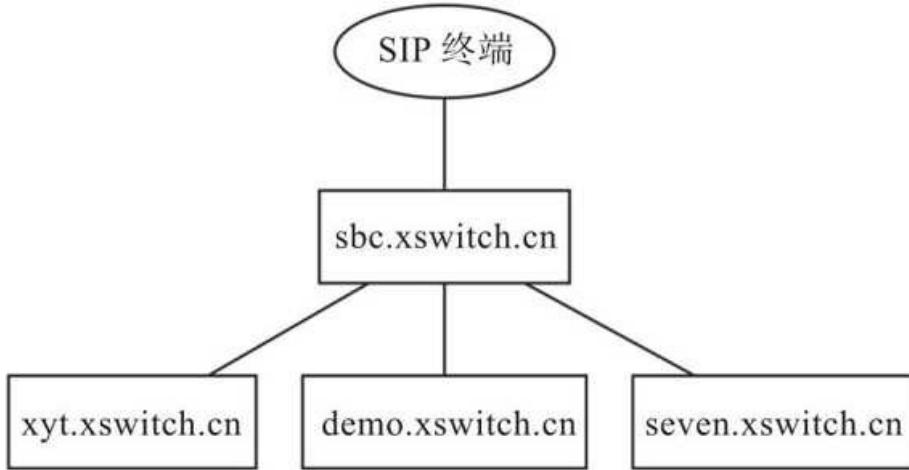


图8-2 xswitch.cn云平台多租户架构

以demo这个租户为例，客户端账号为1001，当它注册时，将代理服务器地址设为



sbc.xswitch.cn:1001，域设为demo.xswitch.cn，以便能路由到对应租户的容器。注册时触发的Lua代码如下。

```
D2IP = {} -- 为了简单，这里做了一个映射表，将域名映射到与 Docker 容器对应的服务器内网 IP 地址和端口
D2IP["xyt.xswitch.cn"] = "sip:10.10.0.8:18860"
D2IP["demo.xswitch.cn"] = "sip:10.10.0.15:10160"
D2IP["seven.xswitch.cn"] = "sip:10.10.0.13:20000"
D2IP["rts.xswitch.cn"] = "sip:10.10.0.8:20001"

-- 处理终端注册消息
function ksr_route_registrar()
    if not KSR.is_REGISTER() then return 1; end
    local next_url = D2IP[KSR.pv.get("$rd")]; -- 获取注册的域，并通过域从映射表中找到容器 SIP 消息的地址

    if next_url then
        KSR.pv.sets("$du", next_url); -- 如果找到
        KSR.path.add_path_received(); -- 设置下一跳的地址
        -- KSR.path.add_path_received() -- 在此我们不用这个函数，也是因为它添加的是公网地址，不是我们想要的
        local received = KSR.pv.get("$$ut") -- 手动获取接收到的 SIP 消息的地址
        if received then
            KSR.hdr.append("Path: sip:10.10.0.13:1001;lr;received=" .. urlencode(received) .. "\r\n")
        end
        ksr_route_relay(); -- 将注册信息转发到对应的容器
    else
        KSR.sl.sl_reply_error(); -- 否则返回错误
    end
    KSR.x.exit();
end
```

来自客户端的呼叫信息处理函数与上面的注册处理流程类似，区别是前者不需要增加那个Path头

域。

用客户端向demo服务注册，注册信息如下。从下面的信息中可以看到，Path头域中包含了SBC的内网IP地址和端口，以及SBC获取到的SIP终端的外网地址（以urlencode方式编码，编码之前的内容为：received=sip:112.238.21.184:63367;transport=udp）。如果SIP终端作为被叫，则FreeSWITCH会将INVITE消息先发到sip:10.10.0.13:1001（即Kamailio），Kamailio再将消息发到那个received=记录的地址上。

```
REGISTER sip:demo.xswitch.cn SIP/2.0
From: "1001" <sip:1001@demo.xswitch.cn>;tag=nGFBU3mWbCyqJjr310y2kWl-nE0dteXv
To: "1001" <sip:1001@demo.xswitch.cn>
Path: sip:10.10.0.13:1001;lr;received=sip%3A112.238.21.184%3A63367%3Btransport%3Dudp
```

在FreeSWITCH中使用sofia_contact 1001@demo.xswitch.cn命令可获取该终端的联系地址，内容如下（其中，fs_path为FreeSWITCH中记录的下一跳的地址）。

```
sofia/default/sip:1001@112.238.21.184:63367;ob;fs_path=sip%3A10.10.0.13%3A1001
```

该终端作为被叫时SIP消息如下（在FreeSWITCH中看到的）。

```
send 1516 bytes from udp/[140.143.134.19]:10160 to udp/[10.10.0.13]:1001 at 2022-04-28 02:32:43.015215:
-----
INVITE sip:1001@112.238.21.184:63367;ob SIP/2.0
```

在Kamailio侧，在客户端做被叫的脚本中直接执行KSR.tm.t_relay()即可，因为上述的R-URI已经是可以路由的了。也就是说，在本示例中，SBC（Kamailio）并不存储注册信息，注册信息都是存储在实际的SIP服务器（FreeSWITCH）中。

8.14.2 NAT穿透

NAT无处不在。你家里用于上网的宽带路由器一般都是NAT设备，当用手机4G或5G网络上网冲浪的时候你实际上也是在一个巨大的NAT网络后面。NAT的全称是网络地址转换，它的出现主要是为了解决IPv4网络地址不足的问题，但同时也给SIP、RTP协议的穿透带来了复杂性。

Kamailio中与NAT处理有关的模块有很多，比如registrar、usrloc、nat_traversal以及nathelper等。主要的NAT操作在nathelper模块中完成，该模块主要的功能就是帮助SIP信令穿透NAT。

nathelper模块主要包含了at_uac_test、set_contact_alias、handle_ruri_alias、fix_nated_register、fix_nated_contact这几个函数，从函数名也大体可以看出它们的作用。下面我们结合一些实例对上述函数进行讲解。

1.nat_uac_test

nat_uac_test函数用于探测终端是不是藏在NAT后面，该函数的原型如下。

```
int nat_uac_test(int flags)
```

其中，flags是探测标志，其取值的含义如下。

- 1：搜索Contact头域，查找RFC 1918或RFC 6598地址的出现次数。
- 2：将Via头域中的地址与信令的源IP地址进行比较。如果Via头域不包含端口，则它使用默认的SIP端口5060。
- 4：搜索最上面的Via头域，查找RFC 1918或RFC 6598地址出现的次数。
- 8：在SDP中搜索RFC 1918或RFC 6598地址出现的次数。

□ 16: 测试源端口是否与Via头域中的端口不同。如果Via头域不包含端口，则使用默认的SIP端口5060。

□ 32: 测试信令的源IP地址是RFC 1918还是RFC 6598。

□ 64: 测试信令的源连接是否为WebSocket。

□ 128: 测试Contact头域中URI端口是否与SIP请求的源端口不同。

□ 256: 测试SDP连接地址是否与源IP地址不同。它还适合用于多个连接地址行的场景。

所有标志都可以按位组合（二进制进行逻辑或运算，十进制直接相加），组合后的测试规则中任何一个检测出NAT，都会返回1，否则返回-1。

我们来看下面的例子。

```
function ksr_route_natdetect()
    if KSR.nathelper.nat_uac_test(19)>0 then -- 19 = 16 + 2 + 1, 三种组合
        KSR.setflag(FLT_NATS); -- 设置事务标志, 检测到了NAT
    end
end

function ksr_route_registrar()
    if not KSR.is_REGISTER() then return 1; end
    if KSR.isflagset(FLT_NATS) then -- 检查是否检测到NAT
        KSR.setbflag(FLB_NATB); -- 设置分支标志, ksr_route_natmanage 函数会检查这个标志
        -- 开启 SIP NAT 保活 (pinging)
        -- 设置分支标志, 当 usrioc 模块检测到这个标志之后, 将周期性地往这个终端发送 SIP
        -- OPTIONS 消息
        KSR.setbflag(FLB_NATSIPPING);
    end
end
```

2.set_contact_alias

set_contact_alias函数会把“;alias=ip~port~transport”添加到Contact头域的后面，形成新的Contact头域。这相当于Kamailio既知道你的内网地址，又知道你的外网地址。

我们来看一个实际的例子。Kamailio部署在阿里云，UA藏在NAT后面，向Kamailio发送INVITE消息。注意，这里Contact中的IP地址是一个RFC 1918私网地址。

```
INVITE sip:9196@106.14.57.231 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.132:5080;rport;branch=z9hG4bKFtjvpFeX9KjeS
Max-Forwards: 70
From: <sip:1001@106.14.57.231>;tag=vSB127etce8vg
To: <sip:9196@106.14.57.231>
Call-ID: 64416dc6-08d3-123b-3f8c-08002722ff3e
CSeq: 47901194 INVITE
Contact: <sip:gw+kam@192.168.1.132:5080;transport=udp;gw=kam>
```

kamailio.lua调用了set_contact_alias函数，当INVITE消息被转发出去的时候，Contact增加了“;alias=113.116.52.68~5080~1”，这个地址是Kamailio收到SIP消息时看到的远端地址，相当于SIP终端的外网地址。

```

INVITE sip:172.19.176.216:7080 SIP/2.0
Record-Route: <sip:106.14.57.231;lr;did=82.02a;rtp=bridge>
Via: SIP/2.0/UDP 106.14.57.231:5060;branch=z9hG4bKb881.221119b4cec486a1e41954aeec
    76f671.0
Via: SIP/2.0/UDP 192.168.1.132:5080;received=113.116.52.68;rport=5080;branch=
    z9hG4bKftjvpFeX9KjeS
Max-Forwards: 69
From: <sip:1001@106.14.57.231>;tag=vSB127etce8vg
To: <sip:9196@106.14.57.231>
Call-ID: 64416dc6-08d3-123b-3f8c-08002722ff3e
CSeq: 47901194 INVITE
Contact: <sip:gw+kam@192.168.1.132:5080;transport=udp;gw=kam;alias=
    113.116.52.68-5080-1>

```

3.handle_ruri_alias

handle_ruri_alias函数用于检查请求URI里面是否有“;alias=ip~port~transport”。如果有，那么据此设置目的URI（\$du），也就是说，Kamailio向终端发包时，不再按照标准的SIP发给Contact指定的地址，而是发到它自己检测出的终端的外网地址，也就是当初通过set_contact_alias保存到Contact头域中的alias=记住的地址。

4.fix_nated_register

fix_nated_register函数用于修复NAT过来的SIP注册请求。也就是说，当Kamailio检测到客户端位于NAT后面时，SIP注册请求里面的Contact头域将会被忽略。创建一个新的URI把源IP地址、端口以及协议组合到一起（在下面的例子中是sip:113.116.52.68:9999），把这个URI保存到AVP变量（nathelper模块的received_avp参数）中，此时会将这个URI加到200 OK消息的Contact头域里面，并保存到用户的location表里面，作为客户端的外网地址。如果以后呼叫用户的时候，就使用这个外网地址，而不是原来Contact头域中的私网地址。

下面是一个200 OK的例子，注意里面的Contact头域。

```

SIP/2.0 200 OK
Via: SIP/2.0/UDP 113.116.52.68:9999;branch=z9hG4bK27872D99A6C4E6F680AA86F35A269B94;
    rport=9999;received=113.116.52.68
From: "1001" <sip:1001@106.14.57.231>;tag=19ECCB94C06EED66481598302DD9BAA5
To: "1001" <sip:1001@106.14.57.231>;tag=3c84d3731alac304707aa88dc7c9656b.41ee61d9
Call-ID: 57F0632DA784E4C1CE6EAFF0A4BA9D31@106.14.57.231
CSeq: 2 REGISTER
Contact: <sip:1001@113.116.52.68:9999;transport=udp>;expires=3000;received=
    "sip:113.116.52.68:9999"

```

5.fix_nated_contact

与fix_nated_register类似，fix_nated_contact函数可用于修复Contact头域，其使用SIP请求消息中的源IP地址和端口替换原Contact头域中的私网地址。

fix_nated_contact函数的使用方法如下。

```

if string.find(KSR.pv.get("Sua"), "Cherry Phone") then
    KSR.nathelper.fix_nated_contact()
end

```

6.小结

通过上述几个函数，Kamailio可以探测到对端位于NAT背后的情况，并根据不同情况使用自己获取到的对端的“外网”地址尝试与对端进行通信，而不是使用SIP消息中的Contact头域中的地址。这便是NAT穿透的原理。由于NAT需要特定的环境，故本示例的脚本仅用于说明问题，而不能直接在本地运行。2.3节介绍的路由脚本中有比较完整的NAT相关的处理逻辑，大家可以翻回去看一看，从而加深理解。

最后，还有一个比较有意思的事情，值得提一下。Kamailio最新的版本在nathelper模块中增加了一个参数——alias_name。该参数用于修改alias的名字，也就是我们在上面看到“alias=”，你可以将其替换成任意的字符串，比如“my_alias=”。正常来说，这个参数仅在IP服务器集群的内部使用，不会影响外面。但有时候会遇到一种情况，就是跟你对接的运营商或网关也使用了Kamailio，也用了跟你同样的方法解决NAT穿透问题，但他们忘了在发出SIP消息前要把这个“alias=”参数过滤掉，消息到了你这边就把你的Kamailio搞晕了。遇到这种情况，你可以告诉对方他们错了，但是，说服对方通常不是那么容易的，而且即使对方认可你的理由也不一定能在你期望的时间内改好。求人不如求己，通过alias_name参数将alias名修改成一个有创意的、不容易被别人使用的字符串就能解决上述问题了。

8.14.3 代理媒体

Kamailio仅是一个SIP信令代理，但在有些网络环境下（如NAT环境），不同的SIP终端间的媒体不能直接互通，需要中间服务器来做RTP媒体转发。常用的媒体转发工具有rtpengine、RTPProxy、Mediaproxy等，它们的功能类似，在此仅以rtpengine为例进行讲解。

1.rtpengine简介

rtpengine是一个媒体转发服务器。除了做媒体转发之外，还有录音、录像、转码、带内DTMF到RFC 2833 DTMF的转换、T38传真到G.711透传传真的转换、s RTP到普通RTP的转换，以及WebRTC中DTLS s RTP到普通RTP的转换等功能。

rtpengine参与的媒体转发的拓扑如图8-3所示。客户端使用SIP通过Kamailio呼叫FreeSWITCH，Kamailio控制rtpengine打开RTP转发通道，rtpengine会返回修改后的SDP（Kamailio控制rtpengine打开RTP转发通道，会携带SDP作为参数），Kamailio拿着这个SDP再去呼叫FreeSWITCH，FreeSWITCH应答后，返回的SDP也要经过rtpengine处理，最后返回给客户端。

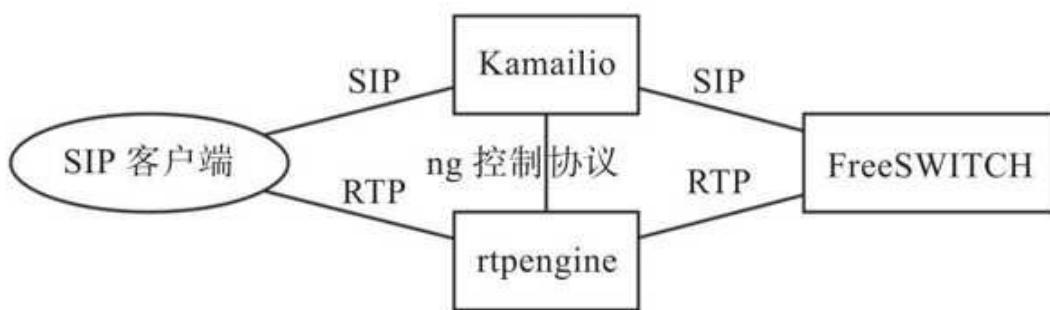


图8-3 通过rtpengine转发媒体的拓扑示意

rtpengine可以工作在Linux操作系统的用户空间，也可以工作在内核空间。当它工作在内核空间时，由于IP包不需要复制到用户空间进行处理，因而转发非常高效。但内核空间也是有限制的，比如无法在内核中进行转码。

rtpengine本身分为两个部分——内核部分和用户部分。内核部分叫xt_RTPENGINE，主要是一个Linux内核模块；用户部分实现在iptables及ip6tables中，用户可以控制内核部分（如打开或关闭端口转发）。

可以这样理解rtpengine：当来了一个呼叫时，Kamailio只转发信令，同时调用iptables打开内核中的IP包转发通道，通话建立后，内核将从一个端口收到的数据包直接从另外一个端口发出去，数据不会被复制到用户空间，因而可以节省相当多的开销。

2.Kamailio的rtpengine模块

Kamailio内部也有一个相对应的rtpengine模块，在该模块中可使用ng控制命令与rtpengine服务器通信，可以命令rtpengine服务器打开和关闭端口转发功能等。常见的ng控制命令有如下几个。

- offer：当呼叫到达时，发送offer命令给rtpengine服务器。

- **answer**: 当呼叫应答后，发送**answer**命令给**rtpengine**服务器。
 - **delete**: 当呼叫失败或者呼叫成功后要结束呼叫时，发送**delete**命令给**rtpengine**服务器。
 - **ping**: 心跳保活。
- 其中**offer**和**answer**会带很多参数，包括**sdp**、**call-id**、**from-tag**等必备参数，以及一些可选参数，具体如下。
- **replace**: 替换媒体属性。如**replace-origin**可以替换**sdp**里面的**origin**地址；**replace-session-connection**可以替换**sdp**里面的**connection**地址等。
 - **SIP-source-address**: 忽略**sdp**正文中给出的任何IP地址，使用接收到的SIP消息的源地址。
 - **trust-address**: 信任**sdp**里面的地址，其作用跟**SIP-source-address**相反。
 - **direction**: 用于控制的方向，比如**direction=priv** **direction=pub**，应用在“1:1 NAT”网络模式下。
 - **transcode**: 转码，比如**transcode-opus**。
 - **ICE**: ICE控制，比如**ICE=remove**（删除ICE）、**ICE=force**（强制ICE）等，WebRTC转SIP时通常要用到。

rtpengine模块配置方法如下。

```
loadmodule "rtpengine.so"
# 配置1个或者多个 rtpengine
modparam("rtpengine", "rtpengine_sock", "udp:192.168.1.100:2223")
modparam("rtpengine", "rtpengine_sock", "udp:192.168.1.101:2223")
```

此外，也可以将上述配置信息放到数据库表里。

```
INSERT INTO rtpengine (setid, url) VALUES (1, 'udp:192.168.1.100:2223');
INSERT INTO rtpengine (setid, url) VALUES (1, 'udp:192.168.1.101:2223');
```

在配置了数据库的URL后，**rtpengine**模块初始化时就可以从数据库中读取上述数据库表中的信息了。参数配置如下。

```
modparam("rtpengine", "db_url", DBURL)
```

3. 使用**rtpengine**进行路由转发

由于涉及Linux内核，**rtpengine**的安装非常复杂。在本书随书附赠的代码中有一个**Dockefile-rtpe**文件，其可以直接在本地编译一个Docker镜像。当然，Docker镜像如果跑在Linux宿主机上是有可能启用内核模式的，在笔者的实验环境中，并未启用内核模式。可以使用如下命令编译和启动**rtpengine**镜像（基于Debian 11）。

```
make build-rtpe      # 编译镜像，镜像名为 kb-rtpe
make up-rtpe         # 启动 rtpengine 容器
make bash-rtpe       # 进入容器内部
```

为便于学习和调试，默认情况下**rtpengine**是不启动的，可以使用如下命令启动。

```
/start-rtpe.sh
```

上述脚本中主要使用了以下命令启动**rtpengine**服务。

```
rtpengine --interface $RTPE_IP_LOCAL\!$RTPE_IP_PUBLIC \
--listen-ng $RTPE_IP_LOCAL:2223 \
--dtls-passive -f -m SRTPE_RTP_START \
-M $RTPE_RTP_END -E -L 7 --log-facility=local1
```

对上述命令中的主要参数说明如下。

□ **--interface**: 监听IP地址, 以! (在Shell脚本中前面的“\”用于转义) 分隔开的IP地址, 前面是本地IP地址, 后面是外网IP地址。

□ **--listen-ng**: 管理端口, Kamailio会连接该端口并对它下达命令。

□ **--dtls-passive**: 在进行DTLS操作时使用被动模式。

□ **-m**: RTP的起始端口。

□ **-M**: RTP的结束端口。

环境变量从.env文件中获取, 其内容如下。 (请记住这个端口范围, 后面我们分析日志时会看到。)

```
KAM_IP_PUBLIC=192.168.3.180 # Kamailio 和 rtpengine 所有的外网 IP 地址
RTPE_RTP_START=29102        # rtpengine 起始 RTP 端口
RTPE_RTP_END=29200          # rtpengine 结束 RTP 端口
```

在book.cfg中添加如下配置, 告诉Kamailio的rtpengine模块如何连接rtpengine服务。其中kb-rtpe是rtpengine容器的名字, 可以直接通过域名引用而无须通过IP地址)。

```
modparam("rtpengine", "rtpengine_sock", "udp:kb-rtpe:2223")
```

启动rtpengine后重启Kamailio, Kamailio就会主动连接rtpengine服务器并对其进行控制。

我们来看如下路由脚本。为了简单, 我们只保留了必备的流程, 没有对SIP消息来源的方向进行检查。

```

function ksr_request_route()
    ksr_register_always_ok()

    if KSR.is_INVITE() then -- 收到客户端的 INVITE 消息，调用 rtpengine 来修改 SDP
        KSR.rtpengine.rtpengine_offer("replace-origin replace-session-connection
            direction=pub direction=priv")
        KSR.tm.t_on_reply("ksr_onreply_manage"); -- 为了能处理 FreeSWITCH 返回的 SDP,
            加一个回调函数
    end

    if KSR.is_BYE() then
        KSR.rtpengine.rtpengine_delete("") -- 呼叫结束后释放 RTP 转发
    end

    -- 按以前的方式正常的转发
    KSR.rr.record_route();
    KSR.pv.sets("Sdu", FS1_URI)
    KSR.tm.t_relay()
end

function ksr_onreply_manage() -- 收到 FreeSWITCH 的响应消息时回调该函数
    local scode = KSR.kx.get_status();
    if scode>100 and scode<299 then -- 如果呼叫成功，且消息中有 SDP，则调用 rtpengine
        修改 SDP
        KSR.rtpengine.rtpengine_answer("replace-origin replace-session-connection
            direction=priv direction=pub")
    end
    return 1;
end

```

当Kamailio收到客户端的INVITE消息时，我们会在rtpengine日志中看到如下内容。（为方便排版，这里加了一些换行符，下同。）

```

[1651037557.873122] INFO: [z-Khh2rSd280vY9kVKzAgPSpxVBFTMIP]:
[control] Received command 'offer' from 172.22.0.3:50242 # 从服务器收到一个 offer 指令
[1651037557.873653] DEBUG: [z-Khh2rSd280vY9kVKzAgPSpxVBFTMIP]:
[control] Dump for 'offer' from 172.22.0.3:50242: # offer 指令内容如下
{
    "supports": [
        "load limit"
    ],
    "sdp": "v=0
o=- 3860026357 3860026357 IN IP4 172.22.0.1
m=audio 4014 RTP/AVP 96 9 8 0 101 102
s=pjmedia

c=IN IP4 172.22.0.1
...
... 此处省略很多内容 ...

```

可以看出，从客户端上来的SDP中，音频端口为4014，IP地址为172.22.0.1（这个地址实际上是笔者本地的Docker网关的地址，被改过一次了，在此不重要，忽略就好）。

然后，rtpengine将里面的端口换成了29194（正好在我们上面设置的环境变量范围内），IP地址换成了rtpengine的外网地址。日志如下。

```

[1651037557.883481] INFO: [z-Khh2rSd280vY9kVKzAgPSpxVBFTMIP]:
[control] Replying to 'offer' from 172.22.0.3:50242 (elapsed time 0.009704 sec)
[1651037557.883568] DEBUG: [z-Khh2rSd280vY9kVKzAgPSpxVBFTMIP]:
[control] Response dump for 'offer' to 172.22.0.3:50242:
{
    "sdp": "v=0
o=- 3860026357 3860026357 IN IP4 192.168.3.180
m=audio 29194 RTP/AVP 96 9 8 0 101 102
s=pjmedia
c=IN IP4 192.168.3.180 # 此处 IP 地址也变成了外网 IP 地址，是笔者电脑上的宿主机地址，从上述
".env" 文件中来

```

当FreeSWITCH回复200 OK时，Kamailio又向rtpengine请求更换SDP，其中，172.22.0.3是FreeSWITCH的IP地址。

```
[1651037706.275866] INFO: [VPxnpEZGWEknJt4dhoBd6BkZNLiZilDx]:
[control] Received command 'answer' from 172.22.0.3:32787
[1651037706.276016] DEBUG: [VPxnpEZGWEknJt4dhoBd6BkZNLiZilDx]:
[control] Dump for 'answer' from 172.22.0.3:32787: { "supports": [ "load limit" ],
  "sdp": "v=0
o=FreeSWITCH 1651008656 1651008657 IN IP4 172.22.0.4
s=FreeSWITCH
c=IN IP4 172.22.0.4
```

从下面的日志中我们看到，rtpengine把FreeSWITCH的200 OK消息中的IP地址和端口号也换成了自己的。

```
[1651037706.281788] INFO: [VPxnpEZGWEknJt4dhoBd6BkZNLiZilDx]:
[control] Replying to 'answer' from 172.22.0.3:32787 (elapsed time 0.005661 sec)
[1651037706.281857] DEBUG: [VPxnpEZGWEknJt4dhoBd6BkZNLiZilDx]:
[control] Response dump for 'answer' to 172.22.0.3:32787: { "sdp": "v=0
o=FreeSWITCH 1651008656 1651008657 IN IP4 192.168.3.180
s=FreeSWITCH
c=IN IP4 192.168.3.180
m=audio 29110 RTP/AVP 8 102
```

至此，rtpengine成了一个中间人，所有的RTP都经过它转发，而客户端和FreeSWITCH都不知道与它们进行RTP通信的实际上是rtpengine，而这是Kamailio在中间“捣的鬼”。

4.关于rtpengine模块的更多解读

为进一步理解及使用rtpengine，下面简单解释一下Kamailio rtpengine模块中的几个主要函数及其使用场景。

rtpengine模块主要提供如下几个函数。

- int KSR.rtpengine.rtpengine_offer(str "flags"): 在收到INVITE消息并且没有to标签的时候调用该函数，替换INVITE消息中的SDP。
- int KSR.rtpengine.rtpengine_answer(str "flags"): 收到响应消息200 OK时调用该函数，替换响应消息中的SDP。
- int KSR.rtpengine.rtpengine_delete(str "flags"): 呼叫失败或呼叫结束时调用该函数，清理现场。
- int KSR.rtpengine.rtpengine_manage(str "flags"): 同时具有上面三个函数的功能，通过不同的参数实现不同的功能。

下面是几种常见的应用场景。

1) 1:1 NAT

1:1 NAT即将一个内部地址映射到一个外部地址的NAT模式。如果外网是UAC，内网是UAS，则使用如下参数（其中Priv和Pub在rtpengine.conf中定义）。

```
KSR.rtpengine.rtpengine_offer("replace-origin replace-session-connection
direction=pub direction=priv")
KSR.rtpengine.rtpengine_answer("replace-origin replace-session-connection
direction=priv direction=pub")
```

如果内网是UAC，外网是UAS，则使用如下参数。

```
KSR.rtpengine.rtpengine_offer("replace-origin replace-session-connection
direction=priv direction=pub")
KSR.rtpengine.rtpengine_answer("replace-origin replace-session-connection
direction=pub direction=priv")
```

2) WebRTC转SIP

WebRTC转SIP主要是完成DTLS与普通RTP间的转换。如果WebRTC侧是UAC，则使用如下方法。

```
KSR.rtpengine.rtpengine_offer("rtcp-mux-demux DTLS=off SDES-off ICE=remove RTP/AVP")
KSR.rtpengine.rtpengine_answer("rtcp-mux-offer generate-mid DTLS=passive SDES-off
ICE=force RTP/SAVPE")
```

如果WebRTC侧是UAS，则使用如下方法。

```
KSR.rtpengine.rtpengine_offer("rtcp-mux-offer generate-mid DTLS=passive SDES-off
ICE=force RTP/SAVPE")
KSR.rtpengine.rtpengine_answer("rtcp-mux-demux DTLS=off SDES-off ICE=remove RTP/AVP")
```

3) 转码

转码的实现方法如下。

```
KSR.rtpengine.rtpengine_offer("codec-mask-PCMA codec-strip-opus transcode-opus")
KSR.rtpengine.rtpengine_answer("") -- answer 时可以不传 codec 参数，因为 rtpengine 已经
知道要做什么了
```

上面代码的意思是跟UAC协商PCMA，跟UAS协商Opus，双方编码不一致就会进行自动转码。其中：

codec-mask-PCMA跟UAC协商PCMA，但不会带到UAS侧。

codec-strip-opus跟UAC不协商Opus。

transcode-opus跟UAS协商Opus。

此外，也可以考虑增加single-codec参数，让rtpengine应答时只支持单个编码。

4) DTMF转码

下面的代码可以把带内DTMF转成RFC2833 DTMF。

```
KSR.rtpengine.rtpengine_offer("always-transcode codec-transcode-telephone-event
codec-offer-telephone-event ptime=20")
KSR.rtpengine.rtpengine_answer("")
```

5) rtpengine安装及配置简介

下面是rtpengine在Debian11上进行安装的相关命令。

```
apt update && apt upgrade -y
echo 'deb https://deb.sipwise.com/spce/mr10.2.1/ bullseye main' > /etc/apt/
sources.list.d/sipwise.list
echo 'deb-src https://deb.sipwise.com/spce/mr10.2.1/ bullseye main' >> /etc/apt/
sources.list.d/sipwise.list
wget -q -O - https://deb.sipwise.com/spce/keyring/sipwise-keyring-bootstrap.gpg | 
apt-key add -
apt-get update
apt-get install -y ngcp-rtpengine
```

编译rtpengine时需要编译内核模块，对环境要求比较严格。如果需要从源码编译，可以参考下面两个链接中的内容，在此就不赘述了。

<https://github.com/sipwise/rtpengine/blob/master/README.md>

<https://nickvsnetworking.com/rtpengine-installation-configuration/>

rtpengine启动时可指定一个配置文件，一般是rtpengine.conf。配置文件类似于Windows上的ini格式

文件。下面是一个配置文件的例子。（笔者在文件内加了部分注释。）

```
[rtpengine]          # 默认设置
table = 0           # table 大于 0，则内核转发 RTP
# table = -1        # 禁止，不需要内核转发 RTP
interface = 192.168.1.100 # 单网卡：
#interface = 192.168.1.100!23.34.45.54 # 内网地址！外网地址

### 1:1 NAT, Priv 和 Pub 分别代表不同 IP 对应的名字，可在 kamailio 脚本中引用
# interface = priv/192.168.1.100;pub/192.168.1.100!23.34.45.54

# ng 端口，Kamailio 的 rtpengine 模块参数要连接这个端口
listen-ng = 192.168.1.100:2223

### 监听的 TCP 端口和 UDP 端口，用来跟 rtpengine 的客户端通信
# listen-tcp = 25060
# listen-udp = 12222

### 监听 HTTP、WebSocket 和 Prometheus 请求
# Prometheus 的 URI 是 /metrics, mr9.5 及更高的版本都支持
listen-http = 9101
timeout = 60          # 媒体超时
silent-timeout = 3600
tos = 184
#control-tos = 184
# delete-delay = 30
```

```

# final-timeout = 10800

# foreground = false      # 是否启动到前台
# pidfile = /run/ngcp-rtpengine-daemon.pid # 启动到后台时, PID 文件
# num-threads = 16        # 线程数
# rtp 端口范围
port-min = 30000
port-max = 40000
# max-sessions = 5000    # 最大 Session 限制
# recording-dir = /var/spool/rtpengine # 录音路径
# 录音方法, proc|pcap, 二选一。如果配置成 proc, 需要使能内核转发 RTP, 并启动进程 rtpengine-
# recording
# recording-method = proc
# recording-format = raw    # 录音格式
# redis 支持相关参数
# redis = 127.0.0.1:6379/5
# redis-write = password@12.23.34.45:6379/42
# redis-num-threads = 8
# no-redis-required = false
# redis-expires = 86400
# redis-allowed-errors = -1
# redis-disable-time = 10
# redis-cmd-timeout = 0
# redis-connect-timeout = 1000

# b2b-url = http://127.0.0.1:8090/
# xmlrpc-format = 0

# 日志相关
# log-level = 6
# log-stderr = false
# log-facility = daemon
# log-facility-cdr = local0
# log-facility-rtcp = local1
# 统计图
# graphite = 127.0.0.1:9006
# graphite-interval = 60
# graphite-prefix = foobar.
# Homer 地址, 用于监控
# homer = 123.234.345.456:65432
# homer-protocol = udp
# homer-id = 2001

# sip-source = false
# dtls-passive = false # 是否启用 DTLS 被动模式

[rtpengine-testing]          # 另一组设置
table = -1
interface = 10.15.20.121
listen-ng = 2223
foreground = true
log-stderr = true
log-level = 7

```

虽然rtpengine工作在内核模式时效率很高，但它总归要转发媒体，需要比较高的处理能力和带宽。一个Kamailio可以同时连接多个rtpengine，以便将媒体分散转发到不同的服务器。此外，rtpengine也可以跟Redis配合做HA（High Availability，高可用）配置。rtpengine还有很多有用的功能和特性，但这些都超出了本书的范围。关于rtpengine就介绍到这里，更详细的介绍可以参考Sipwise官网（<https://www.sipwise.com/>）以及Github上的相关说明（<https://github.com/sipwise/rtpengine>）。

8.14.4 使用FreeSWITCH做B2BUA模式

Kamailio本质上是一个Proxy（代理服务器），用它做的SBC也只能是代理模式。有时候，我们需要B2BUA模式的SBC，这时候就可以使用FreeSWITCH配合来做。FreeSWITCH本质上就是一个B2BUA，它不仅可以用来做拓扑隐藏，还能基于强大的媒体功能和API功能做各种应用级的二次开

发。

那么能否单纯用FreeSWITCH做SBC呢？能，也不能。FreeSWITCH功能强大，可以监听不同的网卡和IP地址，做各种路由转发。但它也有一个缺点，那就是不能直接代理注册，而有些SBC需要代理用户注册。当然，FreeSWITCH也有一种特殊的模式，支持递进的用户注册，即FreeSWITCH上的用户可以配置网关，当用户向FreeSWITCH注册时，FreeSWITCH可以继续向上游的网关注册，这种模式下的两个注册实际上是完全独立的，有完全独立的密码和用户认证体系。当然，我们这里介绍的重点不是FreeSWITCH，所以就不深入探讨这种方法了。下面我们仅讨论如何让Kamailio和FreeSWITCH配合来做SBC。

对照图8-3所示，使用FreeSWITCH做SBC，会得到图8-4所示架构（其中Kamailio和FreeSWITCH的SBC共同完成SBC功能，然后对接第三方的SIP服务器）。

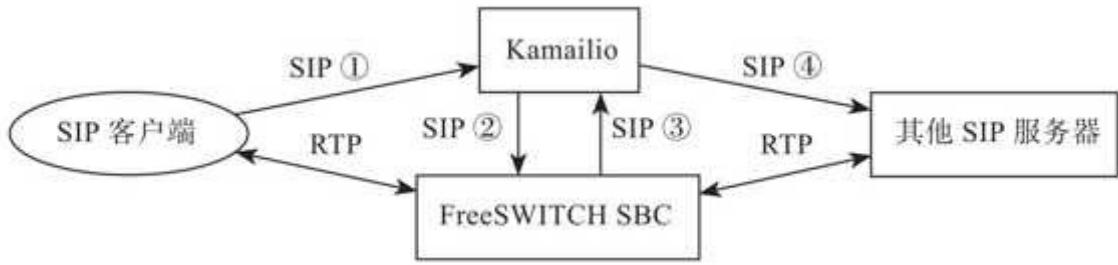


图8-4 Kamailio与FreeSWITCH组合做SBC示意图

一个简单的呼叫流程如下。

- (1) 客户端的呼叫请求到达Kamailio。
- (2) Kamailio通过SIP呼叫FreeSWITCH SBC。
- (3) FreeSWITCH SBC再把呼叫送回Kamailio。
- (4) Kamailio再去呼叫其他SIP服务器。

从上述流程中可以看出，本来正常的SIP呼叫流程是从SIP客户端通过Kamailio直接到达其他SIP服务器，在此，通过让SIP消息到FreeSWITCH上绕了一圈，就完成了让FreeSWITCH更换SDP、打开RTP媒体转发端口之类的工作，这为后续的媒体转发奠定了基础。

在实际使用中，Kamailio可以使用KSR.hdr.append()函数添加SIP头域，用于通知FreeSWITCH开启相应功能特性，如转码（完全可以自定义，如X-SBC-Transcode:true）等。FreeSWITCH可以通过sip_h_X-SBC-Transcode通道变量获取该SIP头域的值进而进行相应处理。返回Kamailio时，FreeSWITCH也可以添加其他SIP头域。

总之，Kamailio和FreeSWITCH组合的SBC与Kamailio和rtpengine组合的SBC相比，相当于使用SIP控制信息代替了ng控制命令，且RTP消息通过FreeSWITCH转发。它们之间的另一个不同是——FreeSWITCH是一个B2BUA，FreeSWITCH侧的SIP消息是“一进一出”的，即从FreeSWITCH出来的SIP消息已经是FreeSWITCH中的另一条腿了，SIP Call-ID与进来的那条腿上的Call-ID没有任何关系，这就做到了拓扑隐藏。

当然，由于使用了SIP，Kamailio中就有了判断4个方向的SIP消息，具体如下。

- 来自SIP客户端。
- 来自FreeSWITCH SBC，客户端呼叫其他FreeSWITCH服务器方向。
- 来自FreeSWITCH SBC，其他FreeSWITCH服务器呼叫SIP客户端方向。
- 来自其他SIP服务器。

不过，相信通过“呼叫从哪里来”（参见6.4节）的例子，加上更多的“if...else”判断，你就可以轻松应对来自各种不同呼叫方向的消息了。具体的实现留给读者自行练习，由于篇幅原因，我们就不讲具体的代码了。

8.14.5 拓扑隐藏

SIP消息可能会泄露SIP网络拓扑信息，这主要是由SIP的设计决定的。SIP消息每经过一级转发，相关的SIP代理服务就会把自己的IP地址以及一些相关的参数放到Via头域中，通过观察这些头域，就可以推断SIP集群的内部结构，甚至在集群外部多打一些电话，也能计算出内部SIP服务器的数量。

没有进行拓扑隐藏操作的消息如下所示（很多IP地址一目了然）。

```
INVITE sip:9196@seven.local;transport=tcp SIP/2.0
Record-Route: <sip:192.168.7.8:35060;r2=on;lr=on;ftag=FLc9x1-plwtHl6Fa7C2V1SpiDVqNybg>
Record-Route: <sip:192.168.7.8:35060;transport=tcp;r2=on;lr=on;ftag=FLc9x1-
plwtHl6Fa7C2V1SpiDVqNybg>
Via: SIP/2.0/UDP 192.168.7.8:35060;branch=z9hG4bKaee3.91f795d5f556cd83c61aaec087
3f028.0;i=1;rport
Via: SIP/2.0/TCP 192.168.7.8:53530;received=172.22.0.1;rport=61024;branch=z9hG4bK
PjUV5Pu0V3n8hd.YYkYTUBw20gWy3EMymb;alias
Call-ID: eDkzvZrfhVUFJMDHPCU1.WmbvXEpBnL6
Contact: "Seven Du" <sip:1000@192.168.7.8:54839;transport=TCP>
```

有时候为了安全，需要将拓扑隐藏，但又要遵循SIP协议，这就需要一些技术。在FreeSWITCH中，SIP很好隐藏，因为FreeSWITCH本身是一个B2BUA，桥接的两条腿无任何直接的关系。在Kamailio中，有一个topoh模块可以进行拓扑隐藏，其原理是将一些没有直接关系的IP地址加密，这样对方在不知道加密密钥的情况下就无法看到相应的IP地址了。

下面是开启topoh模块的方法。

```
loadmodule "topoh.so"
modparam("topoh", "mask_key", "YanTaiXiaoYingTao") # 设置一个加密密钥，要让别人不容易猜到
modparam("topoh", "mask_ip", "10.0.0.1") # 将真实 IP 地址替换成这个假的 IP 地址
modparam("topoh", "mask_callid", 1) # 是否也加密 Call-ID，旧的客户端在 Call-ID 中会包含
# IP 地址等敏感信息
modparam("topoh", "callid_prefix", "***") # 在 Call-ID 头域中增加前缀
```

开启topoh模块后，再看一下SIP消息，有些IP地址已经被隐藏掉了（变成了假IP地址，当然最边缘的代理服务器IP地址无法隐藏，也没有必要），具体如下。

```
INVITE sip:9196@seven.local;transport=tcp SIP/2.0
Record-Route: <sip:192.168.7.8:35060;r2=on;lr=on;ftag=ratGKswB97-t-V43DH8qiLCU9ly4QTlc>
Record-Route: <sip:10.0.0.1;line=sr--feTSOJF.pRxEOhrEvRRSO.K.0DTStXvDVF5-
3Yv10KPD7Nu-O2Y6fRu612Y6fRuCLXUC5KvDBX1V1E7b0w7ZBbmQOb5XJhR-Ve.bK4F61wP4QXyDT**>
Via: SIP/2.0/UDP 192.168.7.8:35060;branch=z9hG4bK0c.8e2a27eacf2662f599f7bf58891e710b.0;
i=1;rport
Via: SIP/2.0/UDP 10.0.0.1;branch=z9hG4bKsr-4PebZ52r.kY4bKNh.cwvZOJfSkR7ZOhEc.FEc-
u-tQOCVefCVbY.c-vZ02vZONr.cm4HOKwjBDu-LoG-LbYEOJTE5huDLWU6tEAnBAFjJ-PDwmjbjOQ2
CQQbCB0eQdCkQ4eZjGEVcPXpXKERjV1Pb7aUVedBSfdyjVd5
Contact: "Seven Du" <sip:10.0.0.1;line=sr--feTSOJT.OoN.cwvZOJfSkR7ZOhEc.FEc-
u11WU6LET67WPnQX040mGDh**>
Call-ID: ***EJhFVJXdSBEQX017Q0JK43K7jfKc.VKKCeE0c0NPCT7A*
```

在集群环境下，多个服务器可以共享相同的mask_key，而不同的服务器都可以解密相关的头域。在实际应用中，也可以定期更换mask_key，减小它因泄露或被破解带来的风险。

使用topoh模块仅需要加载它并配置相应的参数，而无须在路由脚本中写呼叫逻辑。

8.15 WebRTC



WebRTC 的全称是 Web RealTime Communication，即基于 Web 的实时通信。实际上 WebRTC 提供了在浏览器中使用 JavaScript API 访问本地的音频和视频设备的手段，以及点对点流媒体实时传输等功能。目前大部分浏览器都已经支持 WebRTC，包括一些移动端的浏览器。WebRTC 只是媒体层的标准，没有规定信令。从理论上讲，用户可以使用任何信令在通话的双方间交换 SDP，进而建立点对点的媒体连接。由于 SIP 的广泛使用并深入人心，因而将 SIP 移植到浏览器里也成为理所当然的事。由于浏览器没有原生的 UDP 和 TCP 通信协议，但支持 WebSocket，因此，WebSocket 也成了浏览器中信令传输层的协议。基于 WebSocket 实现的 SIP 称为 SIP over WebSocket



，缩写为 SIP/WS 或 SIP/WSS，分别对应非安全连接和安全连接。

Kamailio 支持 SIP over WebSocket。使用它需要加载以下模块，因为 WebSocket 是基于 HTTP 升级而来的。

```
loadmodule "websocket.so"
loadmodule "xhttp.so"
modparam("xhttp", "event_callback", "ksr_xhttp_event")
```

```
function ksr_request_route()
    ksr_register_always_ok()

    KSR.rr.record_route();
    -- 转发到后端的 FreeSWITCH URI。UDP 通常可以工作，但这里我们使用了 TCP
    -- 因为基于 WebRTC 的 SIP 包通常比较大，UDP 传输容易超过网络 MTU 而发生分包，有时无法有效恢复
    KSR.pv.sets("$du", FS1_URI .. ';transport=tcp')
    KSR.tm.t_relay()
    KSR.x.exit()
end

function ksr_xhttp_event(evname)
    KSR.set_reply_close()
    KSR.set_reply_no_connect()

    KSR.info("==== http request:.. evname .. " .. "Ri:" .. KSR.pv.get("SRI") .. "\n")
    local upgrade = KSR.hdr.get("Upgrade")
    -- 检查是否存在 Upgrade 头域，如果为 WebSocket 请求
    if upgrade == "websocket" then
        -- 则进行 WebSocket 握手
        if KSR.websocket.handle_handshake() > 0 then
            -- 握手成功
            KSR.info("handshake ok\n")
        else
            KSR.err("Handshake ERR\n")
        end
        return 1
    end

    KSR xhttp_reply(404, "Not Found", "text/plain", "Not Found")
    return 1
end
```

从上面的脚本中可以看出，SIP 是在 WebSocket 中传输的，除首次连接需要进行一次握手外，其他的信令传输都跟普通的 SIP 没什么区别，转发流程也类似。

为了能进行测试，我们编写了如下 HTML。

```

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <button onclick="makeCall()">呼叫 </button>
    <button onclick="hangup()">挂断 </button>
    <br/>
    <br/>
    <video id="remoteVideo"></video>
    <video id="localVideo" muted="muted"></video>
    <script src="sip-0.20.0.min.js"></script>
    <script src="demo.js"></script>
  </body>
</html>

```

上述代码很直观，只有两个简单的按钮（button），当有呼叫时，远端媒体和本端媒体会显示在两



个video标签上（video标签也支持音频）。在此我们使用的是SIP.js 库。JavaScript脚本（demo.js）如下。

```

var host = '192.168.7.8'; // SIP服务器地址，改成你自己的
var port = '35060'; // SIP端口，改成你自己的
// 将HTTP和HTTPS分别替换为ws和wss
const protocol = window.location.protocol.replace('http', 'ws');
const wsUrl = protocol + "//" + host + ":" + port;

const SimpleUser = SIP.Web.SimpleUser;

const domain      = host;
const callerURI   = 'sip:alice@' + domain; // 主叫URI
const calleeURI   = 'sip:9196@' + domain; // 被叫URI

// 获取HTML中的video标签，在呼叫时将这些标签关联起来
const remoteVideoElement = document.getElementById("remoteVideo");
const localVideoElement = document.getElementById("localVideo");

const configuration = {
  aor: callerURI, // 主叫地址, aor的全称是Address of Record
  delegate: { // 回调函数
    onCallCreated: () => { // 在呼叫创建时回调
    },
    onCallAnswered: () => { // 在应答时回调
      console.log('answered');
    },
    onCallHangup: () => { // 在挂机时回调
    }
  },
  media: { // 媒体参数
    local: { // 本地媒体关联的HTML中的video标签
      video: localVideoElement,
    },
    remote: { // 远端媒体关联的HTML中的video标签
      audio: remoteVideoElement,
      video: remoteVideoElement,
    },
    constraints: {
      audio: true, // 是否启用音频呼叫
      video: true, // 是否启用视频呼叫
    }
  },
  userAgentOptions: {
    displayName: 'Alice',
  },
};

// 创建一个简单的SIP UA并连接
const simpleUser = new SimpleUser(wsUrl, configuration);
simpleUser.connect();

function makeCall() { // 呼叫

```

```
    simpleUser.call(calleeURI);
}

function hangup() { // 挂断
    simpleUser.hangup();
}
```

在上述代码中笔者添加了详细的注释，更多的API可以参考<https://github.com/onsip/SIP.js/blob/master/docs/simple-user.md>中的内容。



可以使用如下命令之一启动一个简易的Web服务器。

```
python -m SimpleHTTPServer # Python 2
python3 -m http.server # Python 3
```

然后就可以使用浏览器（如Chrome）访问<http://localhost:8080>，进行呼叫测试了，如图8-5所示，这里呼叫的9196是FreeSWITCH的环回测试，其中左侧是远端视频，右侧是本端视频（从摄像头采集的视频）。



图8-5 呼叫9196进行环回测试

上面只是简单讲解了WebRTC的配置和演示，在实际的生产环境中逻辑肯定更复杂，可以参考以下链接中的内容获取更多配置：

- <https://github.com/kamailio/kamailio/tree/master/misc/examples/webrtc>。
- <https://www.kamailio.org/docs/modules-devel/modules/websocket.html>。

[1] timerec使用一个RFC标准，《Internet Calendaring and Scheduling Core Object Specification》，即《互联网日历和计划核心对象标准》，该标准定义了周期性日期计划的时间表示方法。详情可参考<https://datatracker.ietf.org/doc/html/rfc2445>。

Chapter 9

第9章 性能

Kamailio装在一般的服务器上每秒都可以处理上万次的注册请求和呼叫请求，包括完整的Challenge验证。虽然现实生产中的场景要比实验室里的场景复杂很多，但依然极少会达到Kamailio的性能瓶颈。

那么，Kamailio性能强悍的秘密是什么？其实，无非就是拥有精心编写的代码，并使用了正确的数

据结构和算法。下面我们先来看一些测试数据，然后再分析一下Kamailio高性能的秘密。

9.1 性能测试

本节给出的测试数据是笔者找到的Kamailio团队所做测试得到的一些性能测试数据，有些测试数据比较旧，但是所涉及的测试步骤和测试方法现在依然适用。一般来说，每个企业在Kamailio实际上线前都要根据自己的场景和业务逻辑制定压力测试方案，所以本节的测试步骤和数据也只是给大家提供一个大概的参考。感兴趣的读者可以用同样的方法在你自己的软硬件上进行测试、对比。

9.1.1 早期的性能测试



下面是一次早期测试的步骤和相关数据。当时的版本是Kamailio（OpenSER）1.2.0。测试的对象主要是与事务（tm）和用户注册相关的模块（registrar及usrloc）。

1. 相关内容和参数

本次测试主要关注事务转发性能，Kamailio在中间转发SIP信令。性能测试拓扑架构如图9-1所示。

测试主要有以下两种。

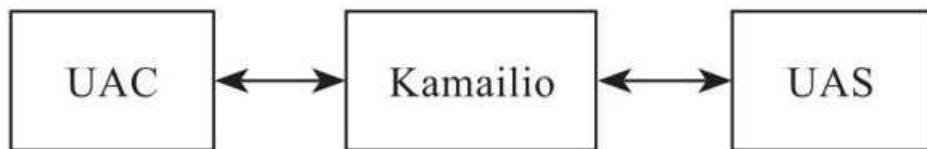


图9-1 性能测试拓扑架构

□ 面向事务转发的测试：通过转发最简单的SIP MESSAGE消息测试基本的性能指标。

□ 面向呼叫的测试：复杂的转发流程，覆盖“INVITE+ACK+BYE”的全呼叫流程。

这里的测试软件使用SIPp，并用SIPp分别作为UAC和UAS，用两台配置相同的服务器作为UAC（32位i386架构、单核AMD处理器），Kamailio以及UAS运行在一台配有双核Intel Core 2处理器的服务器上。

2. SIP MESSAGE测试

Kamailio使用“-m 768”参数启动，即使用768MB共享内存。使用两台SIPp，共产生400000个MESSAGE消息，观察每次处理的事务数和速度（平均响应时间）。

SIP MESSAGE测试的流程如图9-2所示。

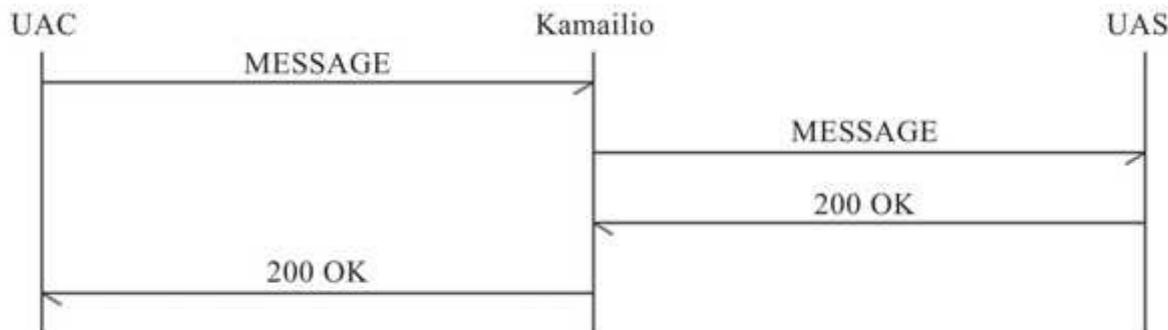


图9-2 SIP MESSAGE测试流程

使用的IP地址如下。

- UAC: 192.168.2.100:5060及192.168.2.101:5060。
- Kamailio: 192.168.2.102:5060。
- UAS: 192.168.2.102:5070。

使用的命令如下。

```
./sipp -sf uac_msg.xml -rsa 192.168.2.102:5060 192.168.2.102:5070 -m 200000 -r  
10000 -d 1 -l 70
```

两台SIPP上的统计结果如表9-1所示。

表9-1 UAC MESSAGE SIPP测试结果统计表

内 容	UAC1	UAC2
IP 地址	192.168.2.100:5060	192.168.2.101:5060
消息总数	200000	200000

(续)

内 容	UAC1	UAC2
最大并发请求数	70	70
最大允许请求速率	10000 个 / 秒	10000 个 / 秒
平均请求速率	8047.966 个 / 秒	7427.765 个 / 秒
失败数	0	0
重传数	0	0
超时	0	0
总时长	00:00:24:851	00:00:26:926

也就是说在双核服务器上Kamailio每秒大约处理了15000（即8047+7427）个事务转发请求。

t_relay()影响时间与请求处理的关系示意如图9-3所示，可以看到，大部分转发的处理时间都小于0.1毫秒。

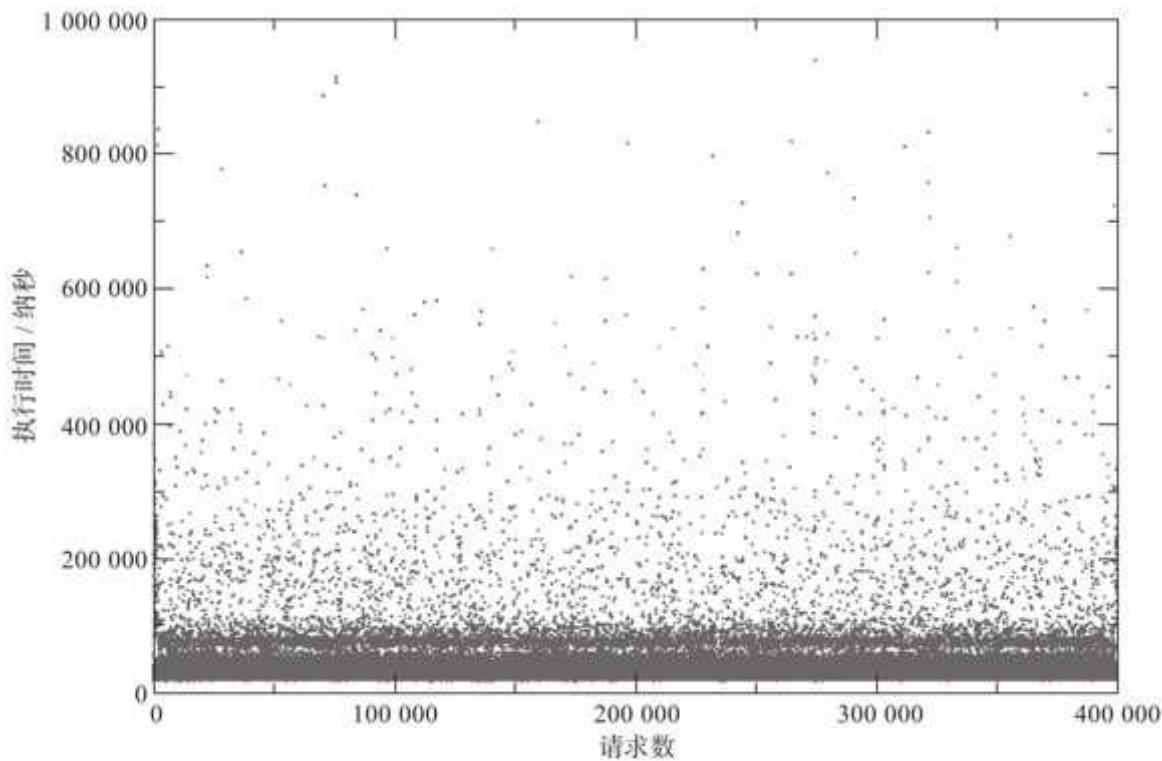


图9-3 t_relay()执行时间

根据SIPp的-trace_rtt参数给出的响应时间画出的示意图，如图9-4和图9-5所示。从图中可以看出，由于UAC侧多了网络层的传输时间，这种方式比Kamailio中的统计结果大很多。一件比较有意思的事情是，那些比较高的竖线反映了Kamailio内的时钟行为，当请求正好赶上时钟锁定（同步）时，会有比较大的延迟。

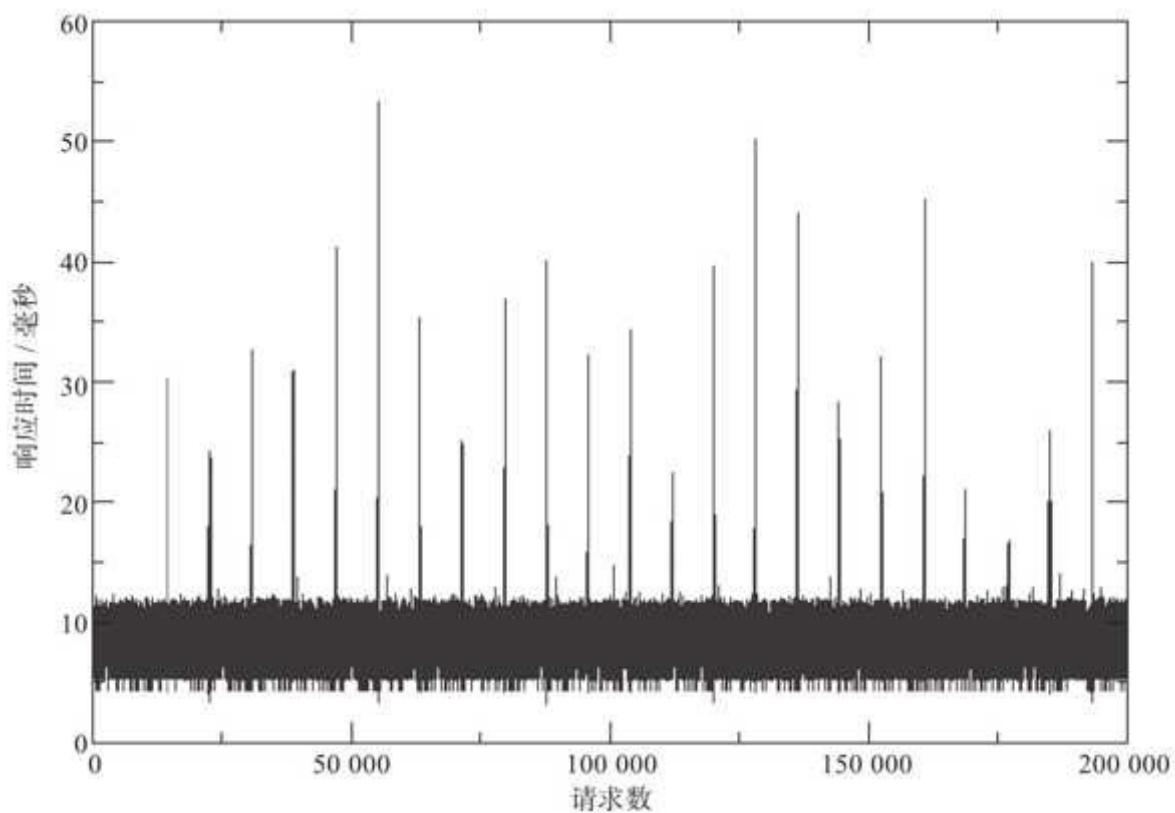


图9-4 SIPp统计的MESSAGE响应时间UAC1

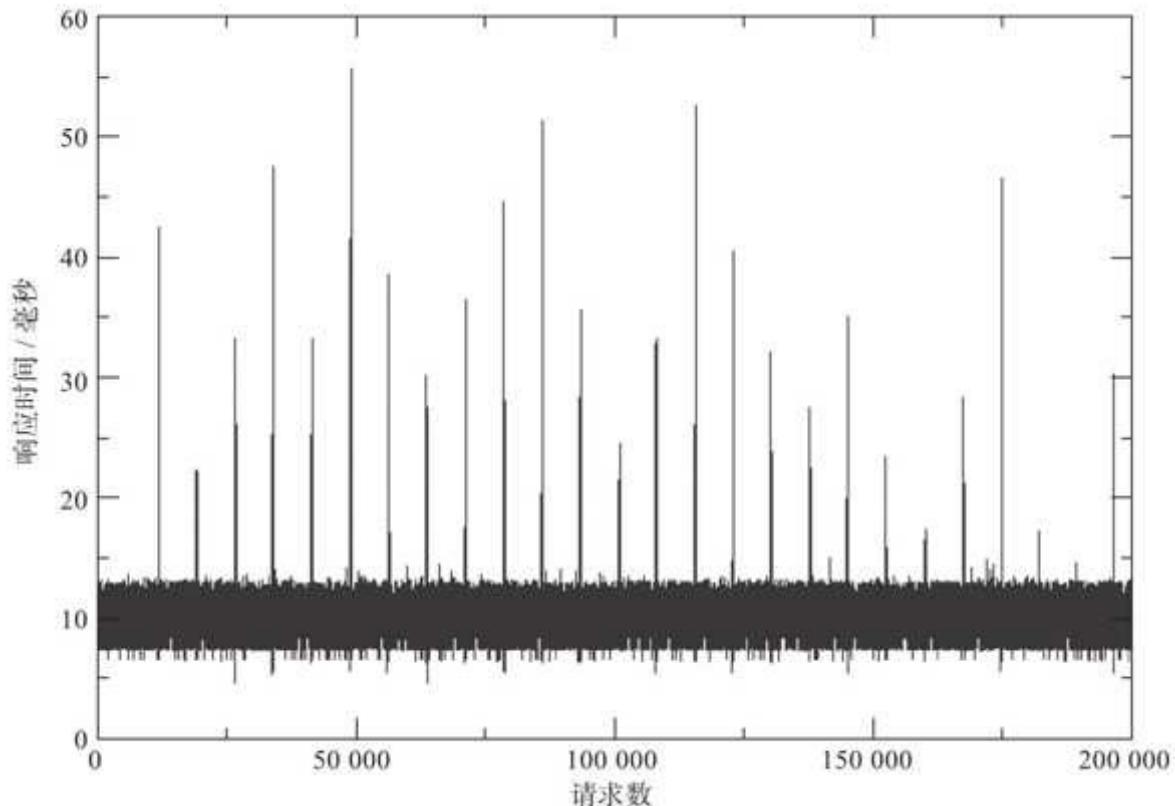


图9-5 SIPp统计的MESSAGE响应时间UAC2

3.呼叫测试

呼叫测试的大部分参数都跟上一小节讲的MESSAGE测试类似，所以这里就不重复介绍了。呼叫测试的主要流程如图9-6所示。

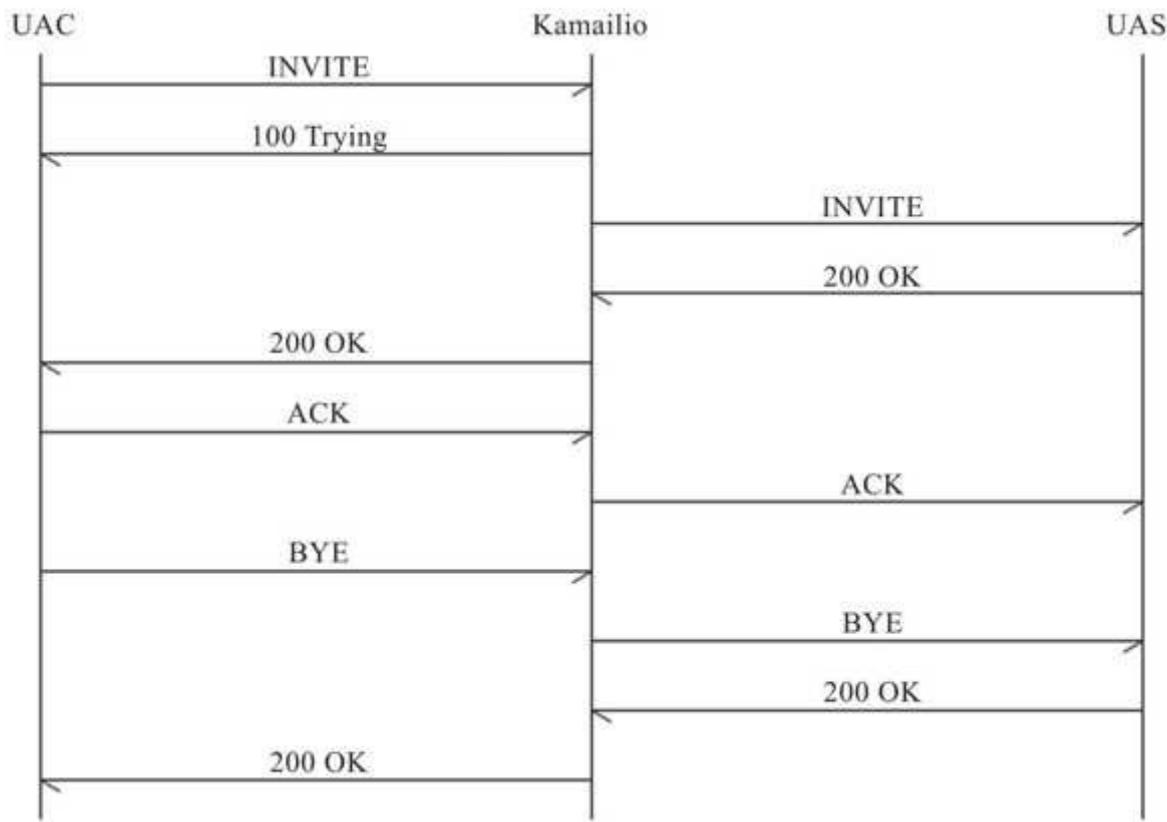


图9-6 呼叫测试流程

呼叫测试使用的SIPp命令如下。

```
./sipp -sf uac_inv.xml -rsa 192.168.2.102:5060 192.168.2.102:5070 -m 200000 -r  
7000 -d 1 -l 27
```

呼叫测试统计结果如表9-2所示。

表9-2 呼叫测试统计结果

内容	UAC1	UAC2
IP 地址	192.168.2.100:5060	192.168.2.101:5060
消息总数	200000	200000
最大并发请求数	27	27
最大允许请求速率	7000 个 / 秒	7000 个 / 秒
平均呼叫请求速率	4164.758 个 / 秒	3896.661 个 / 秒
失败数	0	0
重传数	0	0
超时	0	0
总时长	00:00:48:022	00:00:51:326

从表9-2所示数据中可以看出，Kamailio每秒约处理了8060个呼叫请求。SIPp侧统计的呼叫响应时间如图9-7和图9-8所示。

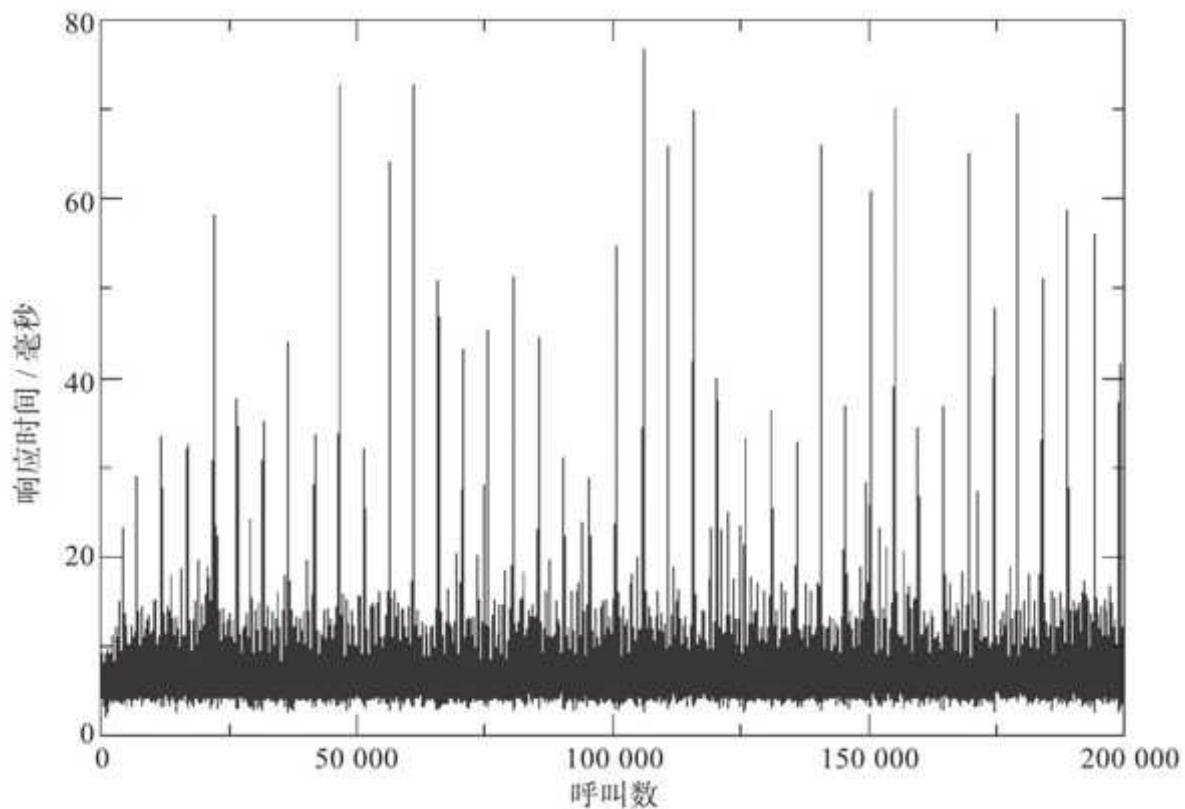


图9-7 UAC1上的呼叫响应时间

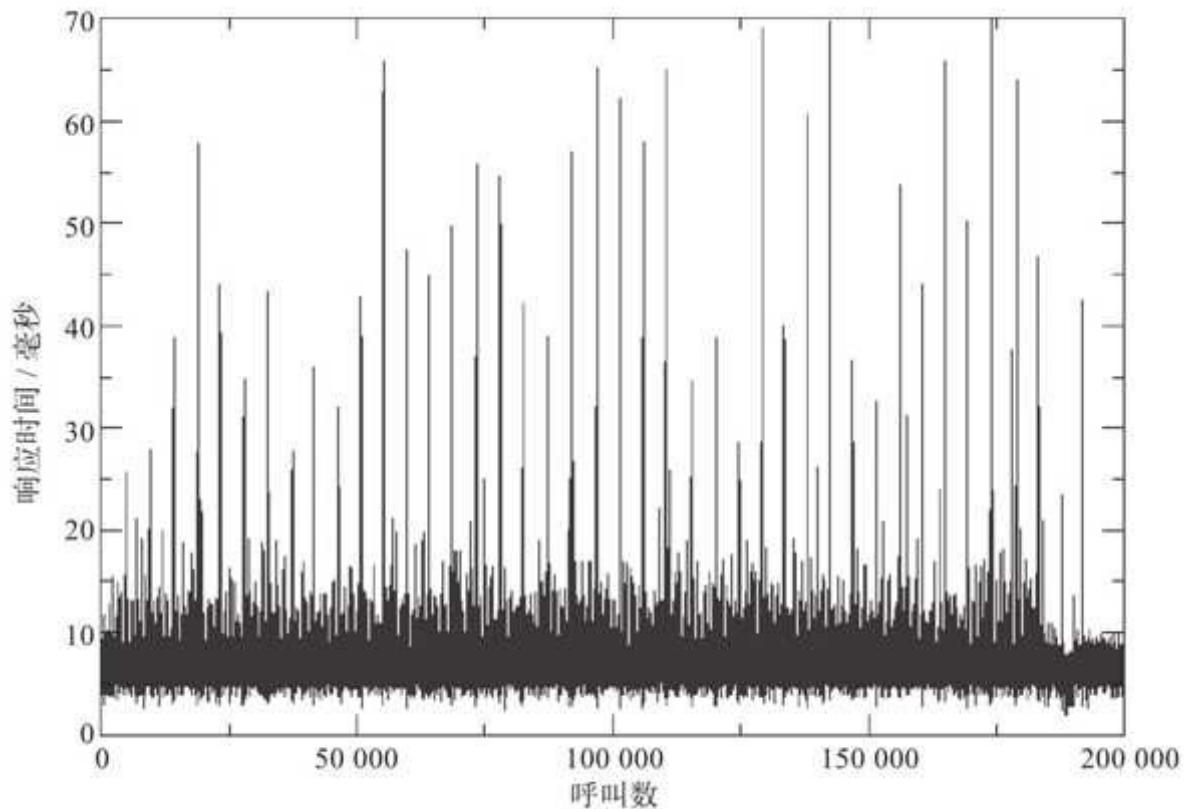


图9-8 UAC2上的呼叫响应时间



从这次测试中我们又发现一件有意思的事情：由于呼叫由两个事务组成，两次一共大约处理了16000（即 8060×2 ）个事务，跟上面的MESSAGE测试的15000个事务差不多。

4. 注册测试

注册测试使用了512MB的共享内存，启动参数为-m 512。本次测试仅使用了一个SIPP作为UAC来发起注册。使用10万个8位号码，通过SIPP以线性顺序发起注册，测试前清空location表（用于存储注册信息的表），该表的存储使用Write-Back模式，即使用modparam("usrloc", "db_mode", 2)参数来实时将数据写入内存，并定期写入数据库。

测试流程如图9-9所示。



图9-9 SIPP注册测试流程

用户号码存储在sip-users-random.txt中，我们可以使用以下命令发起测试。

```
./sipp 192.168.2.102 -sf uac-reg.xml -inf sip-users-random.txt -r 20000 -m 100000
-trace_rtt -trace_screen -l 100
```

测试结果如下。

- 注册次数：100000。
- 最大并发注册数：100。
- 最大允许的注册请求频率：20000个/秒。
- 平均请求频率：7692.899个/秒。
- 失败数：0。
- 重传数：0。
- 超时：0。
- 耗时：00:00:12:999（时：分：秒：毫秒）。

从测试结果来看，Kamailio每秒处理了约7600个注册请求。SIPP侧统计的注册响应时间如图9-10所示。

Kamailio侧统计的save(location)的注册执行时间如图9-11所示。

从图9-11所示可以看出，大部分写入数据库的处理时间都在500微秒以下，那些比较高的线是由于定时器触发了写数据库和检查过期的Contact地址的行为。

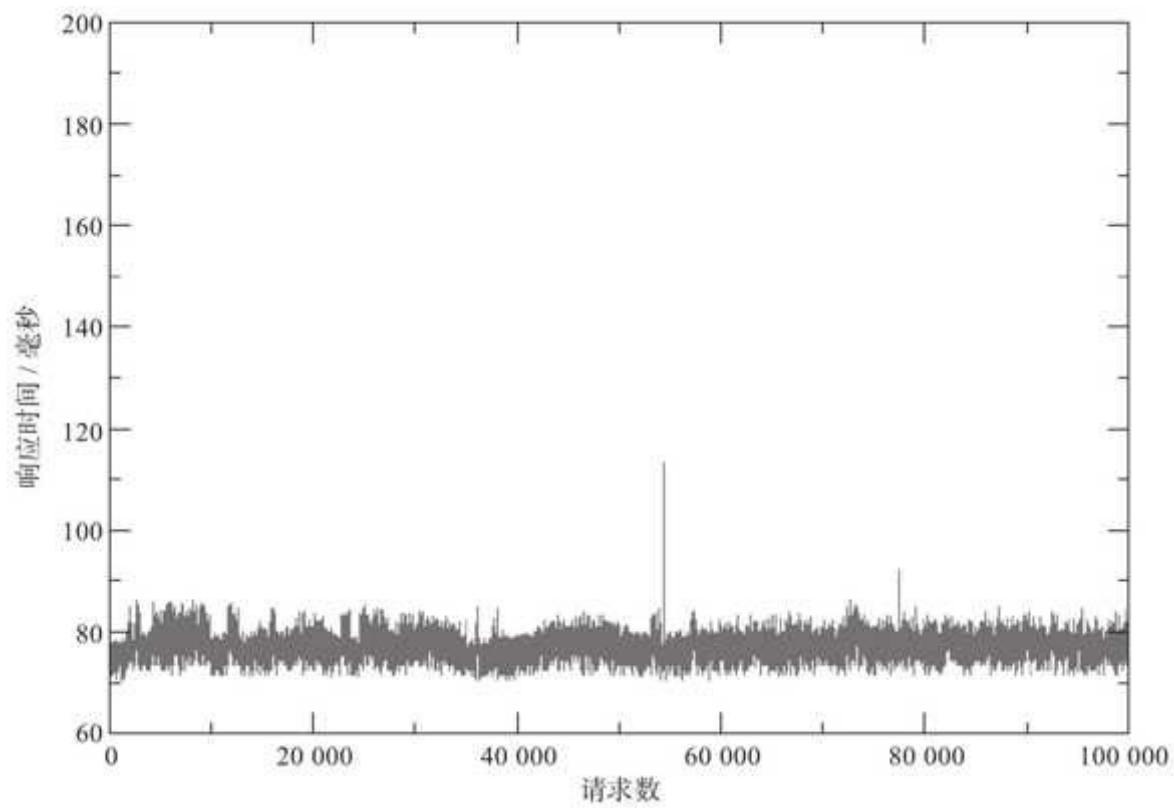


图9-10 注册响应时间

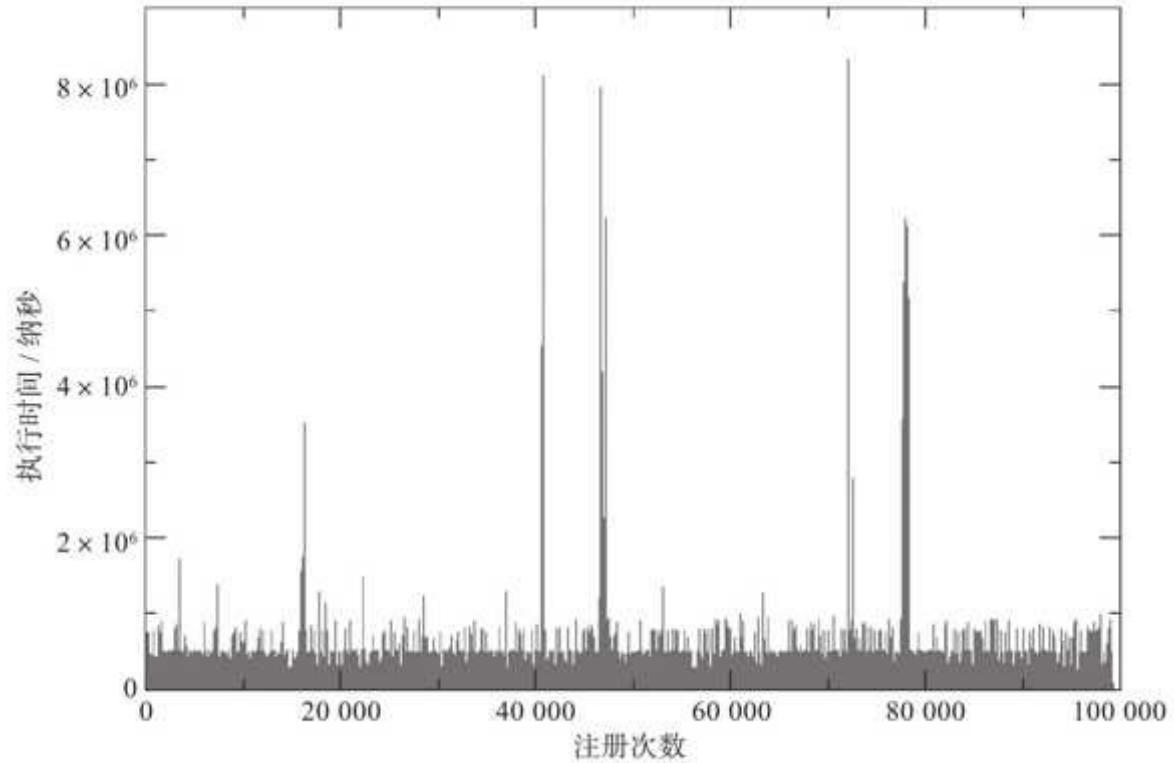


图9-11 注册执行时间

接着做一次刷新注册测试，该测试与上述注册流程一样，区别是10万个注册信息已经保存在内存哈希表里了，不需要插入新数据，仅需要更新相关的注册记录。将sip-users-random.txt复制到sip-

users-linear.txt并把里面的CSeq加1以模拟一个刷新注册的行为。使用的测试命令如下。

```
./sipp 192.168.2.102 -sf uac-reg.xml -inf sip-users-linear.txt -r 20000 -m 100000  
-trace_rtt -trace_screen -l 100
```

测试结果如下。

- 注册次数：100000。
- 最大并发注册数：100。
- 最大允许的注册请求频率：20000个/秒。
- 平均请求频率：10082.678个/秒。
- 失败数：0。
- 重传数：0。
- 超时：0。
- 耗时：00:00:09:918（时：分：秒：毫秒）。

从测试结果来看，Kamailio大约每秒处理了10000个刷新注册请求。响应时间如图9-12所示。

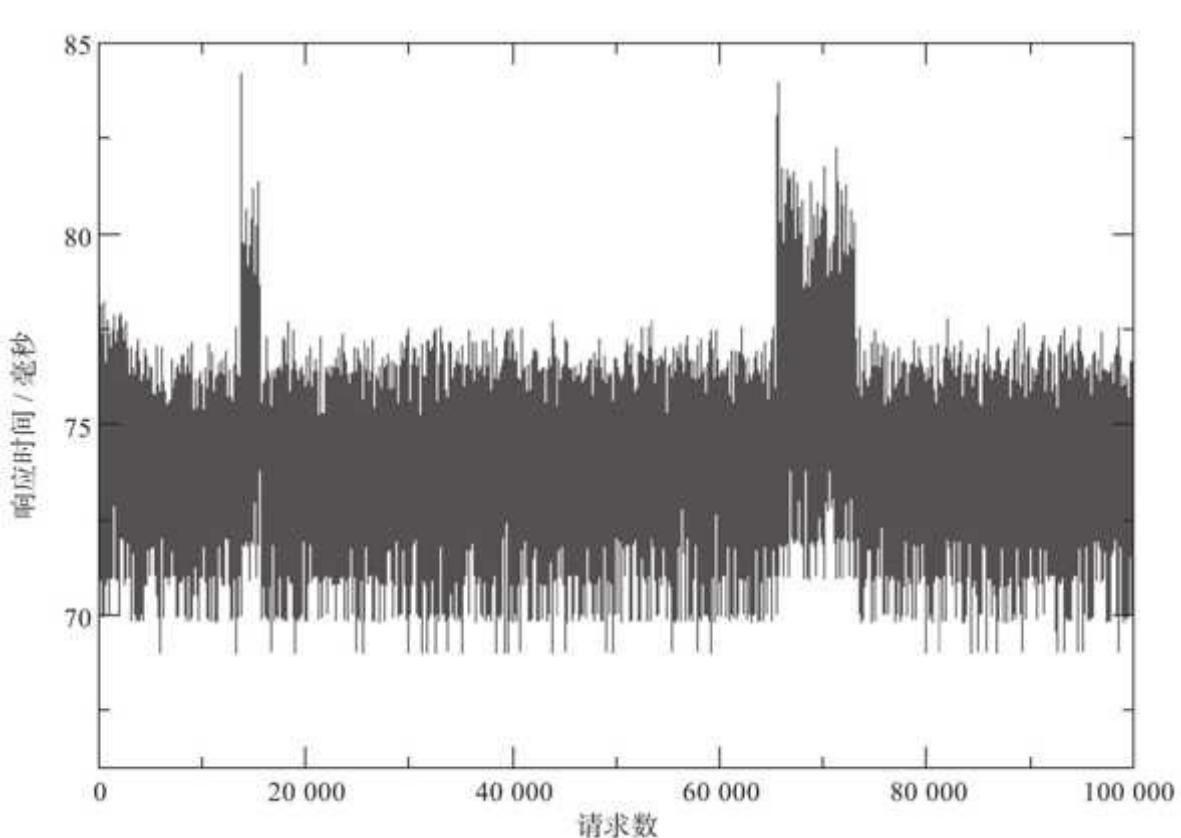


图9-12 刷新注册响应时间

对比图9-11与图9-12可以看到，在这两种情况下有比较大的性能提升，这主要是因为处理刷新注册比处理新注册需要更少的内存。

5.查找注册信息测试

在完成上述注册测试的基础上，10万个用户信息已经注册到了系统上并写入了location表，然后用SIPp发送10万个MESSAGE消息以触发查表（lookup）动作。

测试流程如图9-13所示。

测试时使用的命令如下。

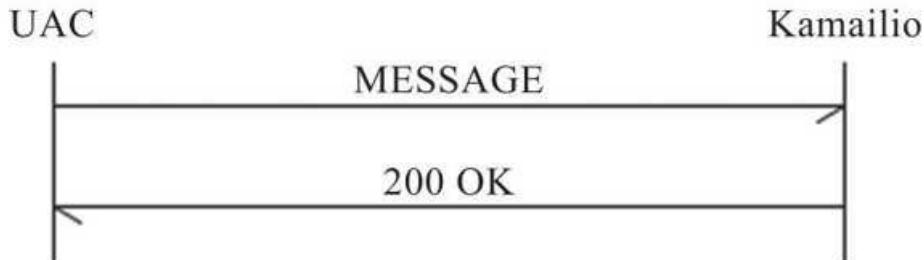


图9-13 查表测试流程

```
./sipp 192.168.2.102 -sf uac-msg.xml  
-inf sip-users-random.txt -r 20000 -m 100000 -trace_rtt -trace_screen -l 100
```

测试结果如下。

- 注册次数：100000。
- 最大并发注册数：100。
- 最大允许的注册请求频率：20000个/秒。
- 平均请求频率：10488.777个/秒。
- 失败数：0。
- 重传数：0。
- 超时：0。
- 耗时：00:00:09:534（时：分：秒：毫秒）。

从结果来看，Kamailio每秒处理了10500次查表。

响应时间如图9-14所示。

Kamailio侧统计的lookup(location)的执行时间如图9-15所示。

图9-15中所示比较高的竖线发生在哈希表碰撞比较多以及定时器触发检查过期的注册信息时，属于正常现象。

6.小结

上面的测试比较简单，仅涵盖了tm模块，在实际应用中肯定还需要进行更多的处理，但tm模块应该是这里面开销最大的，因而对其进行测试也比较有代表性。从测试结果来看，Kamailio在本例的硬件条件下每秒可以处理8000个呼叫请求，这意味着每分钟处理48万个呼叫请求，即每小时处理2880万个呼叫请求。从测试结果来看，还是非常理想的，当然，如果涉及实时数据库的访问，以及



路由逻辑处理等，处理速度肯定是比较慢的。不过，退一步讲，即使这些因素会让整个系统处理能力降低两个数量级，每秒仅能处理80个呼叫请求，也能支撑相当多的应用正常使用。

关于用户注册的处理，在实际应用中一般来说平均的注册过期时间是10分钟（600秒）。系统大约可以支持400万个（600秒×7000个/秒）注册请求，这个数字已经很可观了。

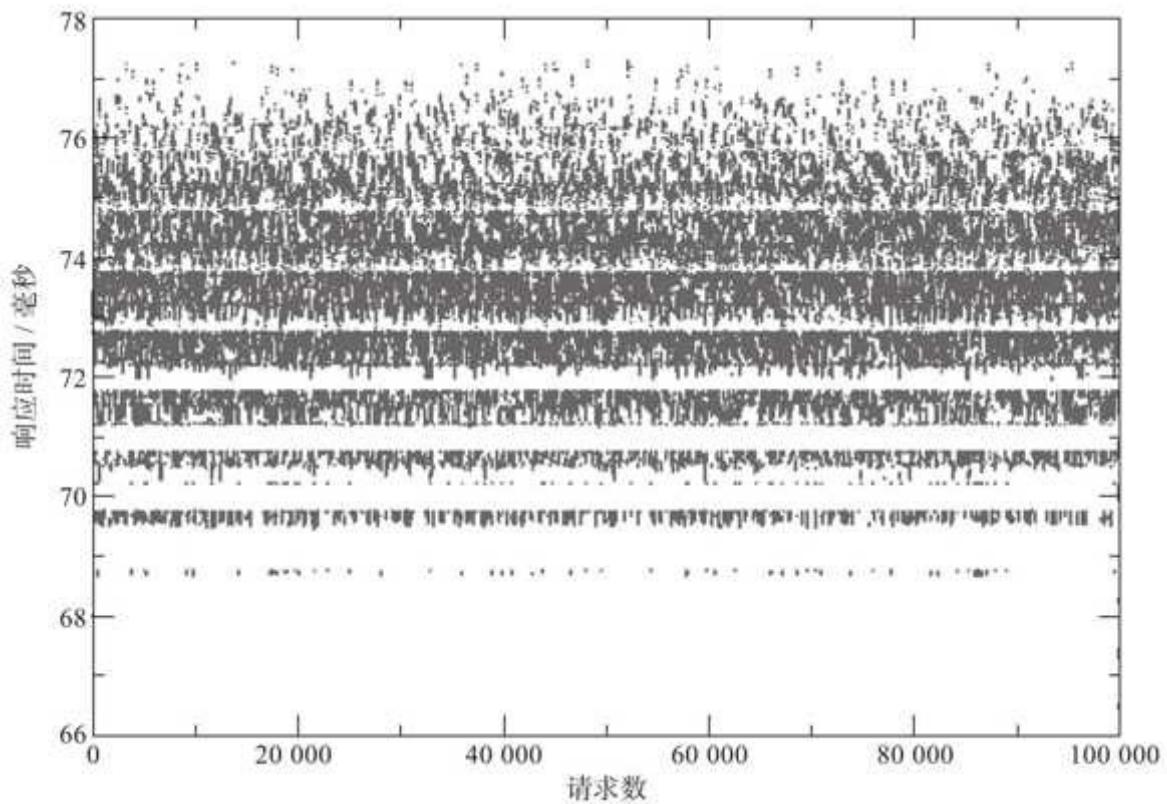


图9-14 查表响应时间

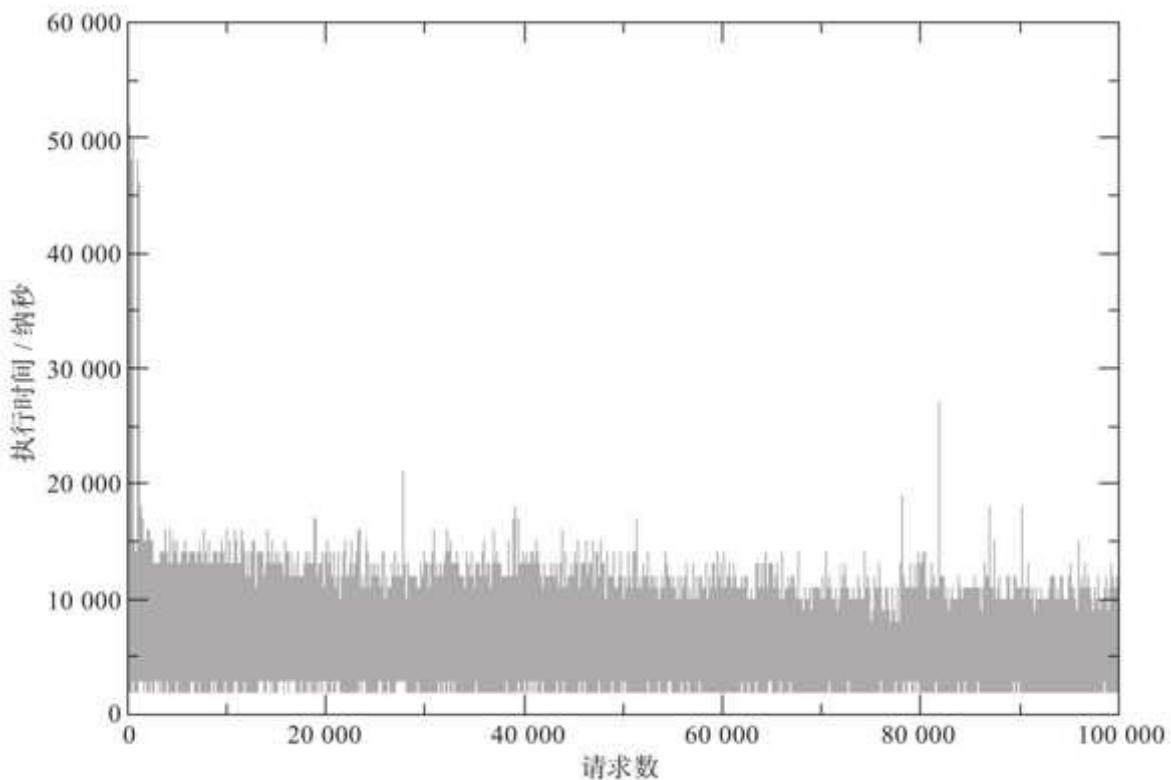


图9-15 注册执行时间

最后需要注意的是，本次测试仅是在实验室环境中的测试，系统的整体性能往往跟很多因素相关，比如下面这些因素。

- 硬件性能：系统的性能跟硬件性能成正比，Kamailio是一个多进程的系统，可以平均用到所有的处理器，所以，CPU核数越多，性能越好。
- 网络性能：不同硬件网卡的性能也有很大差异。
- 测试工具：测试工具有自己的局限性，不一定能反映真实情况。
- 数据采样：测试中的数据埋点和统计采样也会带来额外的开销。

9.1.2 KEMI性能测试



Kamailio自5.0版本起引入了KEMI，官方在2018年11月专门做了KEMI的性能测试，这次测试的重点不是获取Kamailio的最大处理能力，而是比较原生路由脚本request_route与Lua路由脚本ksr_request_route的执行时间，以了解原生路由和KEMI路由脚本间的性能差异。

测试服务器的规格如下。

- 服务器：Intel NUC 7i7DNHE。
- CPU：i7-8650U @ 1.90GHz，4核8线程。
- 内存：16GB。
- 操作系统：Debian 10。

使用SIPp进行测试，场景为标准的注册（REGISTER）场景，包含Challenge验证。使用原生、Lua、Python这三种路由脚本。其中，原生路由脚本、Lua路由脚本与我们在2.3节讲过的类似，Python版路由脚本的逻辑也一致，只是所用语言不同。在同样的硬件条件下，分别启动这三种路由脚本，把Kamailio的日志输出到日志文件，再用一个awk脚本分析日志文件，得到下面的指标。

- cnt：处理的SIP消息数（计数器）。
- sum：request_route或者ksr_request_route()总的执行时间（微秒）。
- min：request_route或者ksr_request_route()的最小执行时间（微秒）。
- max：request_route或者ksr_request_route()的最大执行时间（微秒）。
- avg：request_route或者ksr_request_route()的平均执行时间（微秒）。

官方在进行测试时，对每种路由脚本都做了多次测试，限于篇幅，在此每种路由脚本仅列举一组指标，如表9-3所示。

注意表中的值只是其中一次的结果，实际的情况是，三种脚本的执行效率大致相当，有时原生路由的执行快一点，有时Lua路由的执行快一点，注册请求的平均处理时间为60~80微秒，Python路由脚本跟其他路由脚本的速度差不多，这有些出乎意料。

表9-3 KEMI路由脚本测试指标

指标	原生路由	Lua 路由	Python 路由
总消息数	63157	61080	61031
总时长 / 毫秒	4748289	4203512	4353501
最小时长 / 毫秒	21	26	23
最大时长 / 毫秒	2028	1316	3005
平均时长 / 毫秒	75.1823	68.8198	71.3326

9.1.3 使用VoIPPerf进行性能测试

SIPp是一个经典的压测工具。网上关于SIPp的资料比较多，本书就不多介绍了。在此，我们来看另一个压测工具。



VoIPPerf 是一个新的SIP测试工具，使用起来比较简单。它提供了一个测试用SIP服务器和一个客户端，可以通过JSON配置文件方便地随机产生电话号码。下面来看一看它的使用方法。

VoIPPerf提供了一个Docker镜像，因此测试者无须自己从头编译代码。本书的代码示例中也集成了这个Docker镜像，你只需要像其他示例一样使用如下命令启动一个容器即可。

```
make up-perf
```

容器启动后，默认会启动一个SIP服务器，可以使用如下命令查看相关信息。

```
# docker logs kb-voip-perf

使用如下参数启动: /voip_perf/voip_perf --local-port 5060 --trying --ringing
--delay=250
12:38:30.354      os_core_unix.c !pjlib 2.9-svn for POSIX 初始化完成
12:38:30.372      voip_perf.c 本地端口号 [5060]
Log 级别设置为 :3
12:38:30.389      voip_perf.c 延迟模块注册完毕 id[12]
voip_perf 0.6.2 使用服务器方式启动
在以下URI上接收 SIP 消息 URIs:
sip:0@172.22.0.7:5060 无状态处理
sip:1@172.22.0.7:5060 有状态处理
sip:2@172.22.0.7:5060 呼叫处理
收到的 INVITE 消息如果没有对应的号码将使用有状态处理
```

也可以使用如下命令持续跟踪日志输出。

```
# docker logs -f kb-voip-perf
```

使用如下命令进入容器。

```
make bash-perf
```

进入容器后，可以执行客户端命令进行测试。从上面的Docker日志输出中我们可以看到，服务端的IP地址是172.22.0.7，因此下面的命令中使用该IP地址，即将voip_perf客户端连接到voip_perf服务端并进行呼叫测试。相关命令和输出如下。

```

# ./voip_perf \
"sip:+1206??????@172.22.0.7" \
--method="INVITE" \
--local-port=5072 \
--caller-id="+1?????????" \
--count=1 \
--proxy=172.22.0.7:5060 \
--duration=5 \
--call-per-second=500 \
--window=100000 \
--thread-count=1 \
--interval=1 \
--timeout 7200
13:29:35.560      os_core_unix.c !pjlib 2.9-svn for POSIX 初始化完成
13:29:35.571      voip_perf.c 方法 :[INVITE]
13:29:35.571      voip_perf.c 本地端口号 [5072]
13:29:35.571      voip_perf.c 主叫号码 :[+1?????????] [12 位]
添加代理服务器 : [sip:172.22.0.7:5060;lr|22]
日志级别设为 :3
13:29:35.589      voip_perf.c 延迟模块注册完毕 id[12]
发送一个 INVITE 呼叫到 'sip:+1206??????@172.22.0.7', 最大 100000 个并发任务, 请等待...
1 个任务已启动, 0 个已完成, 等待中...
13:29:42.997 voip_perf.c INVITE-100 count[1] 平均 [3.0ms] 标准差 [0.0ms] 最大 [3ms]
13:29:42.997 voip_perf.c INVITE-180 count[1] 平均 [6.0ms] 标准差 [0.0ms] 最大 [6ms]
13:29:42.997 voip_perf.c INVITE-200 count[1] 平均 [252.0ms] 标准差 [0.0ms] 最大 [252ms]

```

完成

```

共发送 1 个 INVITE 呼叫, 用时 18 毫秒, 速率 55 个 / 秒
共收到 1 个响应, 用时 5265 毫秒, 速率 0 个 / 秒 :
>> 收到的连接响应消息详细信息 :
- 200 连接响应 :           1      (OK)
>> 收到的断开连接响应消息详细信息 :
- 200 断开连接响应 :       1      (OK)
-----  

总响应数 :           1      (速率 =0 个 / 秒 )

```

最大并发任务数 : 0

从上面的输出我们可以看到，在同一个Docker容器中，voip_perf作为客户端连接了voip_perf服务器并成功完成了一次测试呼叫，但由于数量太少所以统计出来的速率为0。其中，客户端使用的参数如下。

- "sip:+1206??????@172.22.0.7": 这是Request URI，即请求URI，?部分会随机生成号码。
- method="INVITE": 发送INVITE消息。
- local-port=5072: 本地端口。
- caller-id="+1?????????": 主叫号码。
- count=1: 一共发送1个消息。
- proxy=172.22.0.7:5060: 代理服务器地址，SIP消息将发送到这个地址上。
- duration=5: 持续时长5秒。
- call-per-second=500: 每秒最多发送500个呼叫请求。
- window=100000: 最大并发呼叫数。
- thread-count=1: 使用一个线程。
- interval=1: 统计采样间隔，此处每秒向voip_perf_stats.log文件写入一次数据。

□--timeout 7200：超时（秒），如果到了这个时间还有没发送完消息，则退出并打印相关统计。

执行完上述命令后，可以看到voip_perf服务端的Docker日志中有类似下面的输出（call变成了1）。

```
总数(速率)：无状态:0 (0/秒), 有状态:0 (0/秒), 呼叫:1 (0/秒)
```

接下来，如果读者在同步做实验，可以修改其中的参数，如--count和--thread-count等，观察对比测试数据。

初步了解了该工具以后，我们再把Kamailio加上。使用如下脚本，把从voip_perf客户端来的呼叫转发到voip_perf服务端。

```
-- voip-perf.lua:  
function ksr_request_route()  
    KSR.rr.record_route()  
    KSR.tm.t_relay() -- 保证客户端发来的R-URI中是voip_perf的服务端地址，在此直接路由即可  
end
```

由于容器启动后，kb-voip-perf容器跟Kamailio容器一样都连接到同一个网络上，因此可以直接使用内网IP地址访问Kamailio容器。找到Kamailio的IP地址（在笔者的环境中是172.22.0.3），修改命令行上的参数，其他不变，只修改如下一行代码。

```
--proxy=172.22.0.3:35060 \
```

当然，不要忘了，其实在Docker中也可以直接使用容器的名字进行DNS访问，如上面命令可以换成以下命令，这样就不需要知道Kamailio容器内部的IP地址了。

```
--proxy=kb-kam:35060 \
```

总之，上述命令可以将SIP消息发送到Kamailio服务器，由于请求的R-RUI还是“sip:+1206??????@172.22.0.7”，因而，呼叫在Kamailio的路由脚本中又转发给了voip_perf服务器。

在只有一个并发（--count=1）的情况下确认一切正常后，慢慢将并发数加大以进行压测。更多的参数和使用方法可以参阅VoIPPerf工具的相关说明手册。掌握好VoIPPerf工具后，读者也可以尝试修改Lua路由脚本，让它指向FreeSWITCH，或者对本书中其他的Lua脚本（如无状态转发和有状态转发脚本等）进行压测。

9.2 拆解Kamailio高性能信令服务设计



Kamailio的共同创始人Daniel-Constantin Mierla在2016年1月FOSDEM上的演讲中详细讲解了Kamailio内部的一些设计和权衡，该演讲的标题是“Designing High Performance RTC Signaling Servers”。这一节，我们就对这次演讲中提到的要点进行简要分析。

9.2.1 懒解析

懒解析就是每次只解析重要的部分，不重要的部分用到时再解析。具体表现为以下几点。

- 只解析需要解析的部分。比如只解析必备的SIP消息头域，那些用不到的头域无须解析，如果后续在路由脚本中用到，再进行解析。
- 将解析出来的部分缓存起来，这样以后再用到的时候就无须重复解析了。
- 不复制，只保留解析的指针。在解析时不会复制整个消息，也不会破坏当前SIP消息的内存。仅需要很少的额外内存开销来保存解析出来的消息的指针，在用的时候就可以快速定位了。
- 使用私有内存。SIP消息仅跟当前的事务有关，不会跟其他进程共享，因而解析SIP消息仅使用私有内存，无须加锁，这样可保证实时高效。
- 只有在需要的时候才将相关数据移到共享内存。当然，如果需要跟踪整个对话，或需要跟异步Worker进行互操作，就需要将私有的SIP数据移到共享内存。
- 保留对消息修改的diff列表。在处理SIP消息时，如果对SIP有修改，不是直接修改当前的SIP消息，而是保存一个变更列表，这样速度最快。
- 只有在发送的时候才根据修改的内容生成完整的消息（按需生成）。如果对SIP消息有变更（对应上文提到的diff列表），则在消息发出前把这个列表应用到整个SIP消息上，生成新的SIP消息。这样在对SIP消息有很多修改时，可以一次性集中操作，以节省资源。

当然，任何事情都有两面性，上面的处理方式也会带来如下一些问题。

- 缺少对消息的合法性验证。由于消息只在发出时才产生，在中间无法进行有效的合法性验证。但由于消息生成是可控的（毕竟路由脚本是我们自己写的），因而该问题也不是太严重。
- 新手可能对消息懒处理比较困惑。比如，当你修改了一个SIP头域，再次查询这个SIP头域的时候，发现该头域并没有变，但在SIP消息发出后，又发现它变了。随着对Kamailio的深入理解，就会知道这是正常现象。

下面来举一个例子。如图9-16所示，Kamailio在收到SIP消息后，解析该消息，取得头域和正文部分，然后删除Subject头域（第7行），并在第11行加上My-Subject头域。在这个过程中，SIP消息内容没有任何变化，但是保存了修改列表（diff列表），该列表记录了删去了多少个字符、在哪个位置添加了哪些内容等，在消息向外发送时将这些修改应用到消息上，并形成新的消息然后发送出去。

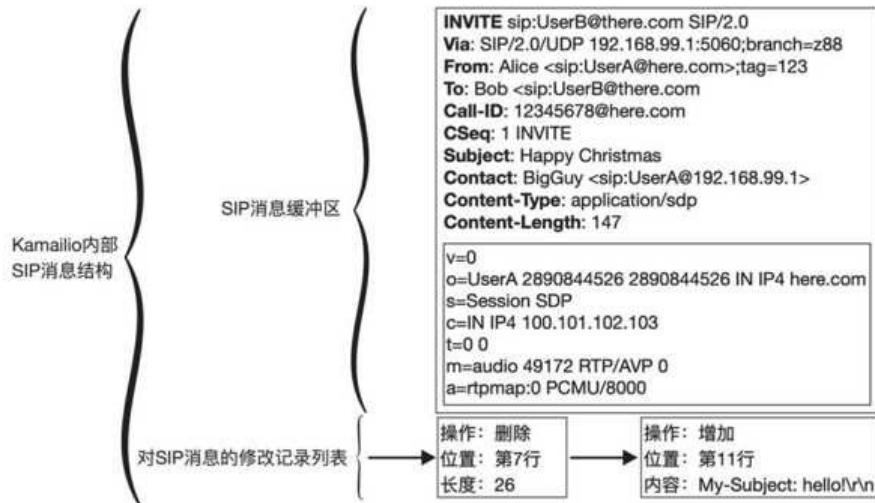


图9-16 懒解析示意

Kamailio有自己的str类型，它是一个典型的字符串类型的结构体，保存了字符串指针和长度，具体如下。

```
struct _str{
    char* s; /*< Pointer to the first character of the string */
    int len; /*< Length of the string */
};

typedef struct _str str;
```

使用这种字符串结构的好处是，在解析时无须破坏被解析的消息本身（一般来说，由于C语言的字符串要求必须以“0”结束，因此仅用C语言原生的字符串指针必然回破坏原始内存空间，如果要保存原来的信息就不得不复制一份）。

图9-17所示是对From头域的解析示例，其中from.uri、from.username等都是str类型的结构体。我们可以看到，虽然解析出了各种结构，但是并没有破坏From头域的内存内容，而只是在解析出结构体的同时保存了字符串的指针起始位置和长度。

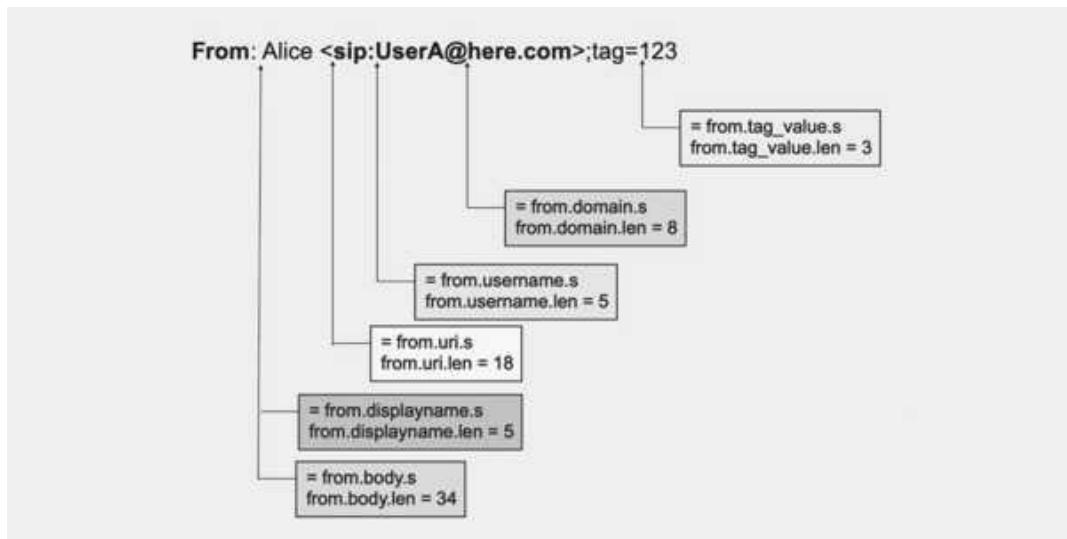


图9-17 对From头域的解析示例

9.2.2 内存管理

Kamailio有自己的内存管理器和内存池。内存管理器首先向操作系统申请一大块内存作为自己的内存池，然后自己管理内部的内存申请和释放。Kamailio的内存分为共享内存和私有内存。由于Kamailio是一个多进程的系统，各进程使用私有内存，可以自由存取而无须加锁，这保证了使用效率。而Kamailio中通用的数据结构，如路由、事务等，则存放在共享内存中。内存管理器提供一个抽象层用于共享内存的存取，并可便于开发者调用。另外，内存管理器也有一些专门针对SIP操作的优化功能，通过这些功能可使SIP操作非常快。Kamailio的内存结构如图9-18所示。

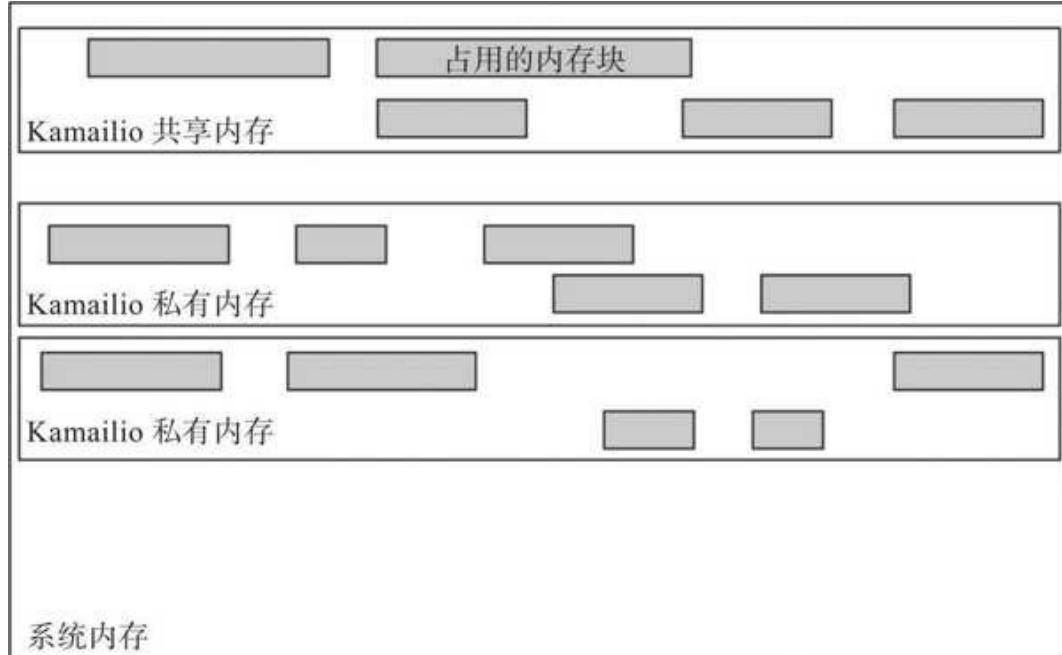


图9-18 Kamailio内存结构

Kamailio的这种内存管理方式有以下好处。

- 可以对常用的内存块大小做优化。自己管理的内存更方便进行内部优化，如将小内存块放到一起或将大块的内存放到一起等。
- 启用或禁用连接（join）操作。可以按需将空闲的小内存块合成一个大的内存块，避免出现内存碎片等。
- 可以选择不同的内存申请算法（在启动时）。其中主要的可选算法如下。
 - fast malloc（即f_malloc）是Kamailio中默认的算法。
 - quick malloc（即q_malloc）中有一些额外的用于辅助调试的信息和功能，适用于在开发时进行内存相关的调试。



○ tlf malloc 是2015年Kamailio在4.3版本中引入的，它的内存申请和释放（malloc和free）都是 $O(1)$ 的时间复杂度，并且没有最坏的情况。



○ doug lea malloc是由Doug Lea 写的内存申请算法，该算法已进入公有领域。



□ 避免非必要的锁定。使用私有内存，能不加锁就不加锁，以保证存取效率。

□ 方便排错，可按需调整。有的内存算法（如quick malloc）内部有相应的调试手段，方便检查内存问题，发现问题时也可以根据需要调整算法。

当然这样做也有如下一些缺点。

□ 在同一个项目中使用不同的内存管理工具不是一件很容易的事。开发者需要小心地使用这些工具。

□ 需要**Kamailio**核心开发者自己维护内存管理代码。这是很大的工作量。

Kamailio默认使用32MB共享内存和4MB私有内存，可以在Kamailio启动时使用如下命令调整这两种内存。

```
kamailio -m 512 -M 8
```

在Kamailio脚本中，下列伪变量会在私有内存中。

□ \$ru、\$rU、\$rd: 请求URI相关的变量。

□ \$fu、\$tu: From URI、To URI相关的变量。

□ \$hdr(name): SIP消息头。

□ \$var(name): 私有变量。

□ \$dbr(key): 数据库查询结构。

下列伪变量会在共享内存中。

□ \$avp(key): 跟每一个SIP事务相关的键值对。

□ \$xavp(key): 键值对的扩展实现。

□ \$shv(key): 共享变量。

□ \$sht(key): 共享哈希表。

私有内存适用于当前SIP消息中相关属性的引用和处理，如果你想在处理SIP请求时保存一个值并在处理SIP响应时引用，则需要使用共享内存中的变量。

9.2.3 并发和同步

当多个进程或线程对共享内存同时访问时，必须有相应的互斥手段，这类手段包括如下几个。

□ 尽量将数据存到私有内存，只有在有必要的情况下才放到共享内存。

□ 可使用标准的Posix锁，也可使用自己实现的基于忙等待（Busy Loop）的锁。

□ 可使用消息队列。

□ 可使用内存围栏。

Kamailio通过在不同情况下灵活使用级别和互斥、锁和数据结构，可以非常好地支持并发访问。

9.2.4 定时器和异步操作

定时器可以为很多模块实现“懒”操作，如保活（Keep Alive）、清除过期的数据等对时间要求精确度不高的操作。这些定时器触发的回调会在独立的Worker进程中异步执行，因而不会响应当前的SIP消息解析。用户也可以根据情况自行调整定时器的时间间隔，示例如下。具体含义可参阅这些模块的说明文档。

```

modparam("usrloc", "timer_procs", 4)          # 启动多少个时钟 Worker 进程
modparam("nathelper", "natping_processes", 6)  # 启动多少个 NAT Ping 进程
modparam("dialog", "timer_procs", 4)            # 启动多少个时钟 Worker 进程
modparam("dialog", "ka_timer", 10)              # 保活时钟数
modparam("dialog", "ka_interval", 300)          # 保活时间间隔

```

9.2.5 缓存

Kamailio内部使用大量哈希表做缓存，哈希表的大小是可以调整的。usrloc、dialog等模块都用到了哈希表，示例如下。

```

modparam("usrloc", "hash_size", 12)
modparam("htable", "htable", "a=>size=4;autoexpire=7200;")
modparam("htable", "htable", "b=>size=8;")
modparam("dispatcher", "ds_hash_size", 9)

```

除哈希表外，还可以用树。比如，字冠路由就比较适合用树的方式查找。pdt、mtree、userblacklist使用的都是树。图9-19所示是从号码分析树中查找010和021等，该图仅是示意图，不代表Kamailio中真实的结构。

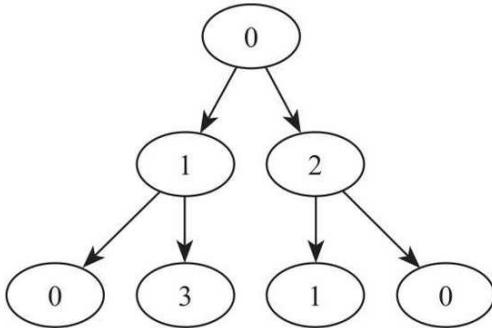


图9-19 号码分析二叉树示意图

mtree模块就使用了树，下面是该模块的一些示例配置。

```

loadmodule "mtree.so"

modparam("mtree", "db_url", DBURL)
modparam("mtree", "mtree", "name=didmap;dbtable=didmap;type=0")
modparam("mtree", "char_list", "0123456789*+")
modparam("mtree", "pv_value", "$var(mtval)")

```

示例路由脚本如下（关于mtree更详细的用法参见8.1节）。

```

if KSR.mtree.mt_match("didmap", KSR.pv.get("SrU"), 0) then
    local dsid = KSR.pv.get("$var(mtval){s.int}")
    KSR.info("Routing to " .. dsid .. "\n")
end

```

9.2.6 异步处理

异步处理可以在等待外部IO或API时防止阻塞SIP路由进程。主要处理方式如下。

□将耗时的操作转移到专门的Worker进程中处理，这可以通过async模块实现，具体的执行过程为

tmx（暂停）→mqueue（分配）→rtimer（处理）。

□采用异步方式进行数据库读写（目前仅MySQL支持异步读写），可以防止数据库读写阻塞当前进程。

□采用异步方式进行HTTP/JSON-RPC交互，可以防止阻塞当前进程。

与数据库异步处理相关的配置参数示例如下。

```
async_workers=4                                # 异步 Worker 进程数量
modparam("sqlops", "sqlcon", "ca=>dbdriver://username:password@dbhost/dbname")
sql_query_async("ca", "delete from domain"); # 使用异步函数执行 SQL 语句
modparam("acc", "db_insert_mode", 2)           # 使用异步方式插入数据
```

与定时器触发的异步处理不同，本节所讲的异步处理是由路由脚本主动发起的。

9.2.7 其他

与性能相关的条件和因素，除了上面讲过的以外，还有下列值得注意的地方。

□children参数可以控制启动多少个Worker进程，可以根据实际需要配置。

□TCP/TLS协议有最大并发限制（Max Connections），有文件描述符限制。

□通过内部的DNS缓存可以提高性能，在有些情况下也可以禁用某些缓存。

□在关键的字段上添加索引可以大大优化数据库查询时间（但要注意过犹不及）。

□Syslog也有异步模式，可以按需使用。

□保证DNS能及时响应（在使用域名路由时，如DNS Failover），DNS查询是同步操作，是阻塞的。

□使用API路由时，API服务要能及时响应，否则会拖慢整个应用，影响系统的吞吐量。

最后，简单做一下总结。本章的内容在前文中大部分都有涉及，因此在这里仅就性能相关的要点进行了集中罗列，并没有深入展开。Kamailio是一个历史很长的项目，各种性能优化也都经历了真实项目和时间的考验，但深入研究Kamailio内部的秘密并不是本书的重点。如果读者通过对本章的学习，能领会到Kamailio功能的强大、性能的强悍，那就足够了，剩下的就看你怎么用好它了。当然，在学会Kamailio基本的使用方法之后，再去追求性能也是理所当然的事。笔者希望本章的内容能作为你下一个追求的起点。

Chapter 10

第10章

安全

安全生产高于一切，没有安全一切都免谈。在实际的生产环境中，经常会遇到各种各样的攻击，典型的攻击有疯狂注册（又称呼叫“撞”密码）、洪水攻击、DDoS攻击等。其中，DDoS是一个世界性的难题，这需要云主机厂商跟你一起对抗，其他的攻击Kamailio都能轻松应对。

10.1 基本安全手段和策略

在深入讨论安全问题之前，我们先说明一点——安全问题并没有万能的解决方案，你必须能 7×24 小时进行监控并能及时启用新的安全策略。下面是一些常用的基本安全手段。

- 监控、探测并屏蔽来自同一IP地址的高频率呼叫。
- 监控、探测同一时间段内大量鉴权失败信息。
- 只允许白名单中的IP地址段访问你的系统。
- 呼叫某目的地要消耗过多资源时，提供警告、屏蔽功能，或启用二次验证。
- 并发超限时告警。
- 超长通话告警。
- 计费超限时告警。
- 检查且仅允许使用安全的密码。
- 启用TLS等。
- 只允许成功注册的用户呼叫。
- 带To tag的INVITE请求必须与已有的Dialog匹配。
- 限制并发注册的数量。
- 限制允许的User-Agent头域。
- 当用户在允许的区域外注册时，限制一些呼叫权限。
- 不同时段启用不同的策略。
- 如果有可能，换用非“众所周知”的端口，如不要用5060端口。

在实际的生产环境中，要根据具体情况采用不同的安全策略。下面我们来看一些典型的例子



。注意与前文相比，本章的例子更关注安全逻辑和算法，例子中用到的模块和函数大部分在前文都有提到，即使是前文没有出现过的新函数，限于篇幅，我们也只是大致解释其用法和作用，不再详细讲解所有参数的含义了，有需求的读者可以自行查阅相关文档。

10.2 限呼

典型的安全手段是动态探测可能的恶意呼叫并进行限呼。下面是一些限呼实例。

10.2.1 限制User-Agent头域



通过限制User-Agent头域可以限制一些已知的扫描者。比如，SIPvicious¹ 是一个著名的SIP安全扫描工具，它通过扫描服务器来检测其有没有漏洞，进而在黑客攻破系统之前就已修补漏洞。但网上有些人却直接拿它来扫描别人的系统。如果你的系统经常受到这种扫描，可以先从User-Agent头域下手，屏蔽一部分扫描。示例代码如下。

```
bad-ua.lua:  
BAD_USER_AGENTS = { -- 黑名单  
    "sipcli",  
    "sipvicious",  
    "sip-scan",  
    "sipsak",  
    "sundayddr",  
    "friendly",  
    "iwar",  
    "sivus",  
    "gulp",  
    "sipv",  
    "smap",  
    "siparmyknife",  
    "test agent",  
    "xcvl23",  
    "pplsip",  
    "sipscan",  
    "custom",  
    "sipptk",  
    "vaxsip",  
    "加入更多 ...",  
  
    function ksr_request_route()  
        if KSR.corex.has_user_agent() > 0 then  
            local ua = string.lower(KSR.pv.get("Sua")); -- 获取 User-Agent 头域  
            for idx, val in pairs(BAD_USER_AGENTS) do -- 循环检查是否在我们的黑名单中  
                if string.find(ua, val) then  
                    KSR.warn("Dropping " .. KSR.pv.get("Srm") .. " UA [".. val .."]" ..  
                        " from " .. KSR.pv.get("Sfu") .. " IP:" ..  
                        KSR.pv.get("Ssi") .. ":" .. KSR.pv.get("Ssp") .. "\n");  
                    KSR.x.drop(); -- 如果 User-Agent 在黑名单中，直接丢弃，不给对方任何消息提示，  
                    -- 沉默是金  
                end  
            end  
        end  
    end  
    -- 下面是正常的业务代码 --  
end
```

上面的函数循环判断SIP请求中的User-Agent头域是否在已知的列表中，如果是，则不做任何响应，让对方不知我们的虚实。如果用KSR.sl.sl_send_reply(200,"OK")直接返回200 OK，那就是一



个“密罐”了。

据说以上方法可以屏蔽90%以上的扫描攻击，但需注意的是，以上方法治标不治本，它只能防止那些漫无目的的攻击，如果黑客确实盯上了你的服务器，他们很容易改变自己的User-Agent头域让你认不出来。

10.2.2 限呼某些目的地

可以通过呼叫字冠或正则表达式限呼。本示例使用mtree模块，该模块的加载和配置方法如下。

```
loadmodule "mtree.so"
modparam("mtree", "db_url", DBURL)
modparam("mtree", "char_list", "+0123456789") # 允许出现在字冠中的字符，默认为
0123456789
modparam("mtree", "mtree", "name=pblock;dbtable=pblock") # 定义树的名字
modparam("mtree", "pv_value", "$var(mtval)")
```

下面是路由脚本（pblock.lua）。

```
function ksr_request_route()
    ksr_register_always_ok()
    local dest = KSR.pv.gete("SrU");
    if KSR.mtree.mt_match("pblock", dest, 0) == 1 then -- 如果从表中找到匹配的数据
        KSR.tm.t_send_reply("503", "Destination Blocked") -- 则返回 503
        KSR.x.exit()
    end
    -- 一切正常，来话不在黑名单中，可以继续路由
    KSR.tm.t_send_reply("404", "Not Found") -- 在此，我们无事可做，简单返回 404 只是为了
    脚本完整
end
```

pblock建表语句如下。

```
CREATE TABLE pblock (
    id SERIAL PRIMARY KEY NOT NULL,
    tprefix VARCHAR(32) DEFAULT '' NOT NULL,
    tvalue VARCHAR(128) DEFAULT '' NOT NULL,
    CONSTRAINT pblock_tprefix_idx UNIQUE (tprefix)
);
```

插入一条黑名单数据，具体如下。

```
INSERT INTO pblock (tprefix, tvalue) VALUES ('1234', 'bad number');
```

运行上述脚本，如果拨打1234开头的号码，就会返回503，否则返回404。关于mtree的用法详见8.1节。

10.2.3 限制高频呼叫

使用pike、pipelimit、htable等模块可以限制呼叫频次，htable也可以用fail2ban代替。我们在2.3节讲

过使用pike模块限制呼叫频次的例子，在此，我们再看一下使用pipelimit模块  的例子。

对pipelimit模块配置如下。

```
loadmodule "pipelimit.so"
modparam("pipelimit", "db_url", DBURL) # 数据库 URL
```

Lua代码如下（pipelimit.lua）。

```

function ksr_request_route()
    ksr_register_always_ok()
    if KSR.pipelimit.pl_check("test") < 0 then -- 检查是否超过限制
        KSR.pipelimit.pl_drop();           -- 如果超限则默认返回 503，响应码也可以配置
        KSR.x.exit();
    end
    KSR.info("Good to go\n")
end

```

pipelimit模块相关的数据表结构如下。

```

CREATE TABLE pl_pipes (
    id SERIAL PRIMARY KEY NOT NULL,
    pipeid VARCHAR(64) DEFAULT '' NOT NULL,      -- 表名
    algorithm VARCHAR(32) DEFAULT '' NOT NULL,    -- 使用的算法
    plimit INTEGER DEFAULT 0 NOT NULL            -- 限制
);

```

pipelimit模块使用ratelimit模块提供的TAILDROP、RED、NETWORK、FEEDBACK算法，具体算法可以参见模块说明文档。在数据库表中插入以下数据。

```
INSERT INTO pl_pipes VALUES (default, 'test', 'TAILDROP', 10);
```

重启Kamailio，打两个电话，日志中会打印“Good to go”，这证明没有超限。使用kamcmd pl.list命令查询会发现last_counter变成了2。

```

{
    name: test
    algorithm: TAILDROP
    limit: 10
    counter: 0
    last_counter: 2
    unused_intervals: 0
}

```

10.2.4 限制太多的错误鉴权

下面是一个常用的场景：如果在一段时间内鉴权失败且超过一定次数，则将对应客户端屏蔽。

要实现上述功能就需要用到哈希表存储鉴权次数。在配置文件中加入以下配置（在此我们设置自动过期时间为300秒）。

```
modparam("htable", "htable", "userban=>size=8;autoexpire=300;")
```

具体的逻辑参见下面的脚本及其中的注释（userban.lua）。

```

function ksr_request_route()
    local au
    local auth_count
    au = KSR.pv.get("$au") -- 获取鉴权用户名
    if au then -- 如果有的话，说明用户需要鉴权
        -- 从哈希表中获取鉴权总数，其中，哈希表的键是当前鉴权用户名与 `:::auth_count` 拼接的字符串
        auth_count = KSRhtable.sht_get("userban", au .. "::auth_count")
        if auth_count and auth_count >= 10 then
            local exp = KSR.pv.get("$Ts") - 300 -- 将过期时间设为距当前时间 300 秒内
            -- 如果用户最后一次不成功则鉴权在 300 秒以内，表示该用户应该被屏蔽
            if KSRhtable.sht_get("userban", au .. "::last_auth") > exp then
                KSR.err("auth - User blocked\n") -- 打印错误消息
                KSR.sl.send_reply(403, "Try later") -- 可以返回错误，以便给用户一个友好的提示
            KSR.x.exit() -- 结束该消息处理
        else
            -- 否则打印当前的鉴权累计数
            KSRhtable.sht_seti("userban", au .. "::auth_count", 0)
        end
    end
    if KSR.hdr.is_present("Authorization") < 0 and KSR.hdr.is_present("Proxy-Authorization") < 0 then
        -- 如果以上两个头域不存在，则发送 Challenge 鉴权验证（401 或 407）消息
        KSR.auth.auth_challenge(KSR.kx.get_e_fhost(), 0)
        KSR.x.exit()
    end

    -- 根据 `subscriber` 表对用户进行鉴权，如果小于 0 则表示鉴权不通过
    if KSR.auth_db.auth_check(KSR.kx.get_e_fhost(), "subscriber", 1) < 0 then
        if not auth_count then -- 如果哈希表不存在，则初始化并生产一个哈希表
            KSRhtable.sht_seti("userban", au .. "::auth_count", 0)
        end
        -- 将计数器加 1

        auth_count = KSRhtable.sht_inc("userban", au .. "::auth_count")
        KSR.info("auth_count = " .. auth_count .. "\n") -- 打印当前的计数器值
        if auth_count >= 10 then -- 如果超过 10 次就不再继续鉴权了
            KSR.err("many failed auth in a row\n")
            KSR.x.exit()
        end
        -- 记住最后一次失败的时间
        KSRhtable.sht_seti("userban", au .. "::last_auth", KSR.pv.get("$Ts"))
        KSR.auth.auth_challenge(KSR.kx.get_e_fhost(), 0) -- 再次发起 Challenge 鉴权
        KSR.x.exit()
    end
    -- 鉴权通过，可以清空原来的鉴权失败计数
    KSRhtable.sht_rm("userban", au .. "::auth_count")
    -- 删除鉴权相关的头域，以免转发到下一跳时对下一跳造成困扰
    if not KSR.is_method_in("RP") then
        KSR.auth.consume_credentials()
    end
    -- 一切正常，下面可以继续后续的流程，如转发给下一跳等
    return 1
end

```

通过上述代码可以看到，我们使用了哈希表存储鉴权失败的计数，如果在一定时间内失败超过一定次数则拒绝对应的客户端，如果认证成功则清空计数。

此外，本例子也是一个Challenge验证的例子，根据请求消息中是否有Authorization或Proxy-Authorization头域决定是否发起Challenge验证。

10.2.5 限制并发呼叫

有时候为了服务器安全，也需要限制每个账号的并发呼叫数。Kamailio对SIP的处理是事务级别

的，如果要跟踪整个会话或对话，就需要用到dialog模块。

首先加载dialog模块，并设置相应的参数，具体如下。

```
loadmodule "dialog.so"
modparam("dialog", "db_mode", 0)          # 不使用数据库
modparam("dialog", "hash_size", 1024)      # 哈希表的大小
modparam("dialog", "enable_stats", 1)       # 启用统计
modparam("dialog", "profiles_with_value", "caller") # 使用 caller 做 profile 的名字
```

其中，dialog中的profile用于对话跟踪。在本例中，我们对该用户的最大呼叫数使用了硬编码的值，在实际使用时，可以从数据库或其他地方获取该值。赋值语句如下（在此最大并发数设为1）。

```
KSR.pv.seti("$xavp(caller=>active_calls)", 1)
```

详细的流程见下面脚本内的注释（active-calls.lua）。

```
FLT_ACALLS = 10                                -- 定义一个Flag

function ksr_request_route()
    ksr_register_always_ok()
    KSR.pv.seti("$xavp(caller=>active_calls)", 1) -- 设置最大并发呼叫数
    ksr_route_dialog()                            -- 调用该函数进行对话处理
    KSR.rr.record_route();
```

```

    KSR.pv.sets("$du", FS1_URI)
    KSR.tm.t_relay()                                -- 转发到下一跳
end

function ksr_route_dialog()                      -- 对话处理函数
    if KSR.is_CANCEL() or
        (KSR.siputils.has_totag() > 0 and KSR.is_method_in("IBA")) then --
            INVITE|BYE|ACK
            KSR.dialog.dlg_manage()                         -- 自动管理对话
            return
    end

    if KSR.is_method("INVITE") and KSR.siputils.has_totag() < 0 and (not
        (KSR.isflagset(FLT_ACALLS))) then
        -- 仅处理第一个 INVITE 消息 (不包含 reINVITE)
        -- 从 XAVP 中获取 active_calls 计数, 默认值为 0
        local active_calls = KSR.pv.getvn("$xavp(caller[0]>active_calls)", 0)
        if active_calls > 0 then -- 如果启用 active_calls 且其值大于 0, 则继续检查
            -- 从 dialog profile 中获取当前一共有多少对话, 存放在 "$var(acsize)" 变量中
            local rc = KSR.dialog.get_profile_size("caller", KSR.pv.get("$fU") ..
                "@" .. KSR.pv.get("$fd"), "$var(acsize)")
            if rc < 0 then -- 获取失败, 返回 500 消息并退出
                KSR.sl.sl_send_reply(500, "Exceeded Max Allowed Active Calls")
                KSR.x.exit()
            end
            local acsize = KSR.pv.getvn("$var(acsize)", 0) -- 从变量中获取值并将其赋给 Lua
            变量
            KSR.info(KSR.pv.get("$fU") .. "8" .. KSR.pv.get("$fd") .. " acsize =
                " .. acsize .. "\n")
            if acsize >= active_calls then -- 如果当前并发的对话数大于预设的值
                -- 则返回 500 消息并退出处理
                KSR.sl.sl_send_reply(503, "Exceeded Max Allowed Active Calls")
                KSR.x.exit()
            end
            -- 将当前的对话记到 caller 这个 profile 里
            KSR.dialog.set_dlg_profile("caller", KSR.pv.get("$fU") .. "@" .. KSR.
                pv.get("$fd"))
        end
        KSR.setflag(FLT_ACALLS)           -- 设置处理标志, 表示我们已经处理过了, 防止重复处理
        KSR.dialog.dlg_manage()          -- 自动对话管理
    end
end

```

通过上述代码可以看出, 对并发呼叫的处理其实依靠的就是一个计数器。在此使用了dialog模块的profile功能进行跟踪计数。当收到首个INVITE时将计数器加1, 失败或收到BYE消息时将计数器减1。除此之外, 也可以使用共享内存、Redis之类的工具进行计数, 这些内容留给读者自行练习。

10.3 TLS

到目前为止，UDP还是最流行的SIP承载协议，但随着技术的发展、时代的进步，以及人们对安全性要求的提高，TLS也渐渐成了SIP的标配。



TLS（Transport Layer Security，安全传输层）协议是HTTPS底层使用的协议。通过使用TLS协议，工作在应用层的程序（如HTTP和SIP）就可以使用同样的安全加密机制了。

10.3.1 理解TLS证书及密钥

使用TLS协议需要有安全证书，而使用安全证书需要有加密密钥。在理解TLS协议之前，我们需要了解一下加密技术及一些相关术语。

TLS协议所使用的加密方法主要是非对称加密^[11]，每个人（或服务器）都有一对密钥——私钥（Private Key）和公钥（Public Key），前者自己拿着，后者通过公开信息发布出去，如果用私钥加密，就可以用公钥解密，反之亦然。私钥主要用于签名，比如我想向外发布一则消息，我就用自己的私钥对这个消息进行加密。消息发布出去以后，由于所有人都知道我的公钥，他们可以使用公钥解密，这样任何人都可以确认这个消息确实是我发出的，而不可能是有人伪造的，因为只有我自己拿着我的私钥。在这个例子中，私钥用于签名，公钥用于解密和验证。但公钥同样也可以用于加密。比如有人想给我发一条机密消息，他先用我的公钥将消息加密，然后将密文发给我，我用私钥对密文进行解密，就看到了消息内容，而其他任何人即使盗取了这条加密消息，由于他们不知道我的私钥，也无法解密。当然，这里面的重点是：私钥一定自己拿着且放到安全的位置。

我们来看标准的HTTPS网站。网站上的内容都用网站的私钥加密，而所有浏览器都有网站的公钥，若是能通过公钥解密并看到网页上的信息，就可以确信该网站不是伪造的。但这里有一个问题，那就是浏览器如何知道它拿到的网站公钥不是伪造的呢？这就涉及公钥的分发问题了。当然，实际情况比这个要复杂得多，网站分发的并不是公钥，而是网站证书（Certificate），在网站证书中包含服务器的公钥及服务器的域名等信息。

浏览器客户端为了能验证网站上的证书是真实有效的，就需要找一个可信任的第三方去问，而这个可信任的第三方就叫CA（Certification Authority，证书认证机构）。所有人大都会对这个CA无条件信任，CA把自己签名的证书装到所有人的电脑或手机里了（装操作系统的时候证书就已经安装好了，不管你用的是macOS、Windows还是Android）。显而易见，如果有人动过你的电脑，把你的电脑中的CA证书替换了，你的电脑也就不安全了。

有了CA以后，CA给网站发证，并且用它自己的私钥加密，然后拍胸脯说这个证书是安全的，你相信它就行了。这就是整个信任的原理。当然，一个CA忙不过来，它可以授权二级CA，二级CA又可以授权更多的CA来做这件事。最顶端的那个CA就称为根CA（Root CA），它要保证私钥的绝对安全，要把私钥存到安全等级很高的保险柜里。当然根CA也不止一个，据统计，全球有数百个公共根CA。

那网站如何申请证书呢？你需要先写一个申请书，申请书中包含你自己机构的一些信息及你的公钥。CA收到你的申请书后，会给你签发证书，并用它的私钥进行签名。然后你就可以将证书放到自己的网站，供浏览器进行验证了。

10.3.2 自签名证书

一般来说，证书是需要向证书颁发机构购买的，而且得先有域名。但在很多测试场景下或在开发过程中，若仅为了测试证书的可行性以及验证完全流程，则就可以使用自签名证书。使用如下命令可以生成自签名证书。

```
openssl req -x509 -nodes -newkey rsa:1024 -keyout kamailio-book.key -out kamailio-book.crt -addext "subjectAltName=DNS:seven.local,IP:192.168.3.180"
```

在生成证书的过程中，它会问你一些问题，按提示填入即可，示例如下。

```
Country Name (2 letter code) [AU]:CN          # 国家代码, CN 代表中国
State or Province Name (full name) [Some-State]:Shan Dong    # 省
Locality Name (eg, city) []:Yan Tai           # 市
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Kamailio Book # 组织或公司名
Organizational Unit Name (eg, section) []:SIP      # 部门名称
Common Name (e.g. server FQDN or YOUR name) []:seven.local # 公用名称, 一般是域名
Email Address []:dujinfang@gmail.com           # Email 地址
```

在上述命令中，我们创建了一个私钥（kamailio-book.key）和一个证书（kamailio-book.crt），在检查证书时，有的客户端会针对其中的公用名称做检查，但更多的会根据别名中的域名或IP地址做检查。所以，这里我们通过-addext参数添加了域名和IP地址作为别名。其中的域名和IP地址是笔者本地的，读者在使用时可以换成你自己的。

10.3.3 在Kamailio中配置TLS

在Kamailio中配置TLS支持也很简单，首先修改kamailio.cfg，增加如下配置。

```
#!define WITH_TLS
```

实际上，上述代码解锁了以下配置。

```
enable_tls=yes
listen=tls:KAM_IP_LOCAL:KAM_SIP_TLS_PORT advertise KAM_IP_PUBLIC:KAM_SIP_TLS_PORT
```

配置tls.cfg，内容如下。

```
[server"default]
method = TLSv1.1+                                # 服务端默认配置
verify_certificate = no                            # 使用 TLSv1.1 以上版本
require_certificate = no                          # 不验证证书
                                                # 不需要客户端提供证书

private_key = /etc/kamailio/tls/kamailio-book.key # 私钥
certificate = /etc/kamailio/tls/kamailio-book.crt # 证书

[client"default]                                    # Kamailio 作为客户端时的默认配置
method = TLSv1.1+
verify_certificate = yes
require_certificate = yes
```

10.3.4 TLS连接测试

配置好Kamailio后即可重启Kamailio，之后就可以使用OpenSSL做连通性测试了，命令和输出如下。

```

# openssl s_client -connect seven.local:35061

CONNECTED(00000005)                                # 连接成功
depth=0 C = CN, ST = Shan Dong, L = Yan Tai, O = Kamailio Book, OU = SIP, CN =
    seven.local, emailAddress = dujinfang@gmail.com
verify error:num=18:self-signed certificate          # 自签名证书
verify return:1
depth=0 C = CN, ST = Shan Dong, L = Yan Tai, O = Kamailio Book, OU = SIP, CN =
    seven.local, emailAddress = dujinfang@gmail.com
verify return:1
---
Certificate chain                                         # 证书信任链
0 s:C = CN, ST = Shan Dong, L = Yan Tai, O = Kamailio Book, OU = SIP, CN = seven.
    local, emailAddress = dujinfang@gmail.com
1:C = CN, ST = Shan Dong, L = Yan Tai, O = Kamailio Book, OU = SIP, CN = seven.
    local, emailAddress = dujinfang@gmail.com
    a:PKEY: rsaEncryption, 1024 (bit); sigalg: RSA-SHA256
    v:NotBefore: May 13 06:37:22 2022 GMT; NotAfter: Jun 12 06:37:22 2022 GMT
---
Server certificate                                       # 证书内容
-----BEGIN CERTIFICATE-----
MIIDIjCCAougAwIBAgIUhQGQHqTuF07Y02rb1qpD3isYPAUwDQYJKoZIhvcNAQEL
BQAwg2MxCzAJBgNVBAYTAKNOMRIwEAYDVQQIDALTaGFuIERvbmcxE DAOBgNVBAcM
... 省略很多行 ...

```

另外，也可以使用Chrome浏览器打开相关域名和端口，如打开https://seven.local:35061，Chrome提示连接失败，并在地址栏中显示“非安全连接”，点击后可以显示证书，显示内容与上述内容类似。

确保TLS连通后，就可以使用sipexer发送SIP消息进行测试了，示例如下。

```

sipexer -tls-insecure "sip:seven.local:35061;transport=tls"
sipexer -tls-insecure -ruri sip:seven.local "sip:192.168.3.180:35061;transport=tls"

```

在上述命令中，我们使用了-tls-insecure参数，这可让客户端不验证证书，因为我们的证书是自签名的，验证也无法通过。在实际使用时，如果验证证书，可能会出现如下错误。

(1) 证书由未知的机构颁发，具体如下。

```
error: x509: certificate signed by unknown authority
```



(2) 无法验证证书，因为证书中不包含任何关于IP地址的SAN，具体如下。

```
error: x509: cannot validate certificate for 192.168.3.180 because it doesn't
contain any IP SANs
```

10.3.5 自制CA根证书

在10.3.4节中，由于我们使用了自签名证书，但其不能被操作系统信任，因此，我们使用了不安全的选项运行sipexer。在此，我们尝试自己做一个CA根证书，这样就可以自己给自己签名了。

由于我们需要自己充当CA，所以需要先产生一个CA私钥（ca.key），实现方法如下。

```
openssl genrsa -out ca.key 1024
```

生成CA机构自己的证书申请文件——ca.csr，其中csr是Certificate Secure Request的缩写，具体如下。

```
openssl req -new -key ca.key -out ca.csr
```

CA用自己的私钥和证书申请文件生成自己签名的证书，这样便得到了CA根证书（ca.crt）。

```
openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt
```

生成的几个文件可以使用如下命令查看和验证自己的证书。

```
openssl rsa -noout -text -in ca.key  
openssl req -noout -text -in ca.csr  
openssl x509 -noout -text -in ca.crt  
openssl verify ca.crt # 验证证书，结果会显示是自签名证书
```

为我们的Kamailio服务器生成一个新的私钥，实现方法如下。

```
openssl genrsa -out server.key 1024
```

生成一个证书申请文件，实现方法如下。

```
openssl req -new -key server.key -out server.csr
```

使用CA证书对我们的申请文件进行签名，生成我们Kamailio服务器所需的证书，实现方法如下。

```
openssl x509 -req -CA ca.crt -CAkey ca.key -CAcreateserial -in server.csr -out server.crt -extfile extfile
```

上述命令中我们使用一个-extfile参数指定了一个文件extfile。该参数相当于前面讲过的-addext参数，但此处不能使用-addext，只能使用-extfile指定从一个文件中读取相应的文件。在笔者电脑上extfile文件的内容如下。

```
subjectAltName=DNS:seven.local,IP:192.168.3.180
```

有了CA签名的证书，就可以用server.key和server.crt替换掉原来的kamailio-book.key和kamailio-book.crt了，之后就可以重启Kamailio进行测试。

当然，由于我们自己的CA是不受操作系统信任的，在测试时客户端还是会报错。为了让操作系统信任我们的CA，可以将CA根证书导入系统中。在笔者的macOS系统中，直接在终端命令行上通过open ca.crt命令打开系统的钥匙串按提示导入证书并信任该证书即可；在Windows上，可以从“控制面板”→“系统和安全”处导入证书。

在本例中，我们成功充当CA，生成了自签名的根证书，并用它签发了证书。如果测试完毕，可以将根证书从系统中删除；如果不删除，则一定要注意保护好这个CA的私钥。

10.3.6 其他

上面使用自签名证书（通过自制的CA根证书签名的证书）演示了TLS的配置。在实际使用时，如果在生产环境的公网上使用，则需要向证书颁发机构购买真正的证书。国内腾讯云、阿里云等都提



供限量版的免费证书，国外的Let's Encrypt机构也提供免费证书。无论是免费的证书还是收费的证书，如果是由权威机构签发的，一般在申请时仅需要提供域名，而不需要IP地址。事实上，很多机构不支持针对IP地址签发证书。当然，证书的安全级别也有很多种，高安全级别的证书（如金融机构使用的证书）签发前需要提供更多的企业信息以供验证，费用也比较高。

在此我们主要讨论SIP服务器端的证书。实际上，客户端也可以有证书，并且可以被服务端验证，

注

但通常没有人这么做，这主要是由于为每个客户端安全地分发证书是非常困难的。所以，上面在Kamailio的tls.cfg中设置了不验证客户端证书。这其实问题不大，因为我们后续还会在SIP消息中对用户的身份进行Challenge验证。

如果是在企业内网上使用TLS，有人也会将域名解析到内网IP地址，但这样做是不允许的，主要原因是会影响反向地址解析。如果内网与外网完全隔离，则需要在企业内部架设DNS服务器。有些证书颁发机构也会给企业颁发适用于内网的证书。当然，也可以在公司内部使用自签名证书，但那样的话，需要将自己当作CA，并把自己的CA根证书装到每一个需要连接TLS的电脑和设备上。更简单的办法是，在SIP客户端或话机上直接关掉TLS证书验证。但这些做法都是很不安全的，在实际使用时请与企业内部的安全主管讨论，以选择适当的安全策略。

注

此外，值得一提的是，RFC 5922明确说明了SIP协议必须不支持泛域名证书。这种证书存在一个典型问题：假设你的泛域名证书支持很多域名（如*.example.com）和主机，并且大部分都非常安全（如secure.example.com），但是只要有一台主机安全级别不高（如weak.example.com）并被黑客攻破，黑客就可以想办法冒充其他安全的主机，从而导致整个泛域名下的主机变得不安全。

10.4 iptables

纵然Kamailio有很多安全手段，我们也不要因为学了Kamailio而忘记iptables。iptables是Linux上的一个实用程序，用于在用户空间（User Space）管理内核（Kernel）中的防火墙IP包过滤规则。iptables不仅功能强大，而且由于其实际的过滤规则工作在内核中，所以运行非常高效。

iptables规则一般由一些链（Chain）组成，常用的有INPUT（入链）、OUTPUT（出链）及 FORWARD（转发链）等。下面是一些基本的iptables规则。

```
iptables -A INPUT -i lo -j ACCEPT          # 允许所有 loopback 网络包
# 允许所有已经成功连接的数据包(用于TCP)
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j ACCEPT      # SSH 端口
iptables -A INPUT -p tcp --dport 80 -j ACCEPT      # Web 端口, HTTP
iptables -A INPUT -p tcp --dport 443 -j ACCEPT     # 安全 Web 端口, HTTPS
iptables -A INPUT -p tcp --dport 5060:5069 -j ACCEPT # 允许这些 SIP TCP 端口
iptables -A INPUT -p udp --dport 5060:5069 -j ACCEPT # 允许这些 SIP UDP 端口
iptables -A INPUT -p udp --dport 16384:32768 -j ACCEPT # 允许这些 RTP 端口
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT # 允许 ping
iptables -P INPUT DROP                      # 其他端口都不允许，丢弃数据包，黑洞
iptables -P FORWARD DROP                   # 不允许端口转发
iptables -P OUTPUT ACCEPT                  # 允许所有向外发的包
```

除了这些基本规则外，我们还可以拒绝一些常见的扫描包，这与10.1节所讲的内容类似。下列规则屏蔽了包含friendly-scanner的IP包。

```
iptables -I INPUT -j DROP -p tcp --dport 5060 -m string --string "friendly-
scanner" --algo bm
iptables -I INPUT -j DROP -p udp --dport 5060 -m string --string "friendly-
scanner" --algo bm
```

上述规则仅供参考，可以根据需要添加。下面是一些iptables常用命令，仅供参考。

```
iptables -L -v          # 列出所有规则
iptables -L -v -n --line-numbers # 列出所有规则，并显示行号
iptables -D INPUT 2      # 删除 INPUT 链的第二行
iptables -P INPUT ACCEPT # 允许 INPUT 链
iptables -P OUTPUT ACCEPT # 允许 OUTPUT 链
iptables -P FORWARD ACCEPT # 允许 FORWARD 链
iptables -F              # 清空所有规则
# 开 IP 地址白名单，允许 1.2.3.4 访问本机的 5060 端口
iptables -A INPUT -j ACCEPT -p tcp --dport 5060 -s 1.2.3.4/32
```

rfc/rfc5922.html#section-7.2。

```
# 黑名单。从 1.2.3.4 收到的包全部丢弃，黑洞
iptables -I INPUT -s 1.2.3.4 -j DROP
```

上面这些规则都是通过手工或使用脚本执行iptables命令的方式设置的IP包过滤规则。iptables本身

没有稳定的C语言API，所以不方便直接在Kamailio中调用，不过，有一个iptables-api项目支持通过HTTP REST API的方式设置iptables IP包过滤规则。有了它，再配合我们在6.5.1节讲过的HTTP相关模块，就可以在Kamailio中动态调整iptables规则了。

使用iptables最大的风险就是在操作远程主机的防火墙时自己把自己隔离在外面，所以在学习和设置iptables时一定要慎重。



10.5 其他安全建议和相关链接

关于安全我们暂时就讲这么多。正如本章开头所说，安全问题并没有万能的解决方案。在实际应用中还需要多监控，遇到问题做到早发现、早解决。幸运的是，网络上大部分扫描攻击都是漫无目的的，如果你不幸遇到大规模的有组织的攻击，那么事先准备好的服务降级（如只保留最重要客户的IP地址段）策略也许能救命，而且你应该在第一时间报警。此外，应用程序的缺陷、运维管理的疏漏、来自内部人员有意或无意的破坏也是不可忽视的影响安全的因素，有时这些甚至比外来的攻击更难以发现和定位，破坏性更强。

除了本章列出的方法和示例，下面的链接中还有一些有用的内容，大家可自行学习。

- APIBAN是一个SIP安全网站，提供免费API，用于检查恶意呼叫。其网址为<https://apiban.org/>。
- Fail2Ban让Kamailio在鉴权失败时打印客户端的IP地址，通过监控Kamailio日志并调用iptables屏蔽这些IP地址，并可以在指定的时间后自动解封，具体参见https://www.fail2ban.org/wiki/index.php/Main_Page。
- Kamailio安全的相关内容可以参见<http://www.kamailio.org/wiki/tutorials/security/kamailio-security>。
- 关于18小时SIP扫描攻击的相关介绍可以参考<https://kb.asipto.com/kamailio:usage:k31-sip-scanning-attack>。
- 关于SIP安全的内容可以参见<https://www.wiley.com/en-gb/SIP+Security-p-9780470516362>。

[1] 非对称加密主要是基于数学上的素数特性实现的——设有两个非常大的素数 p 和 q ，它们的乘积为 N ，通过 N 很难推导出 p 和 q 。

附录A

安装Kamailio

为了方便大家学习和使用，本书的Kamailio示例都是基于Docker镜像和Docker容器的方式讲解的。如果你不会使用Docker或者你的环境中无法使用Docker，就会影响对本书内容的理解和使用。为了避免这种情况，我们在此简单介绍通用的非Docker环境安装Kamailio的方法。

A.1 在Debian和Ubuntu上安装Kamailio

使用APT软件仓库安装软件是Debian和Ubuntu上标准的软件安装方法，Kamailio官方维护着自己的APT软件仓库。

使用Kamailio的APT仓库前需要先安装其GPG Key，如果你的机器上还没有GPG工具，可以使用如下命令安装。

```
apt-get install gnupg2 wget
```

使用如下命令安装GPG Key。

```
wget -O https://deb.kamailio.org/kamailiodebkey.gpg | sudo apt-key add -
```

将如下内容放到/etc/apt/sources.list中。

```
deb http://deb.kamailio.org/kamailio55 buster main
deb-src http://deb.kamailio.org/kamailio55 buster main
```

上述代码中，buster代表Debian 10的发行代号；如果是Debian 11，则应换成bullseye；如果是Ubuntu，则应换成bionic、focal等。更多不同的版本和仓库对应关系参见<https://deb.kamailio.org/>。

接下来就可以使用如下命令安装Kamailio了。

```
apt-get update
apt-get install kamailio
```

如果你想安装其他模块，可以使用如下命令查询Kamailio中所有的模块。

```
apt-cache search kamailio
```

选择你需要的模块进行安装，如安装mysql和websocket模块，命令如下。

```
apt-get install kamailio-mysql-modules
apt-get install kamailio-websocket-modules
```

按照上述方法安装的Kamailio，配置文件在/etc/kamailio/目录下。安装完成后你就可以使用本书中介绍的方法创建数据库，以及启动和关闭Kamailio了，具体如下。

```
kamdbctl create
/etc/init.d/kamailio start # 用 System V 方式启动并控制脚本
/etc/init.d/kamailio stop
systemctl status kamailio # 用 Systemd 方式启动并控制脚本
systemctl start kamailio
systemctl stop kamailio
```

在Debian及Ubuntu上安装Kamailio的更多内容请参考<https://kamailio.org/docs/tutorials-devel/kamailio-install-guide-deb/>。

此外，Kamailio也维护着RPM包，RPM包适用于RedHat和CentOS系列的Linux系统。不过CentOS 8



已经在2021年12月31日失去支持，CentOS 7也已于2020年8月失去完整支持，并将于2024年完全失去支持。至于新的CentOS Stream，那是另一个版本体系了。如果你在使用CentOS或CentOS Stream，可以参考<http://www.kamailio.org/wiki/packages/rpms>。

A.2 从源代码安装

Kamailio的开发者大都使用Ubuntu和Debian系列的Linux系统。如果你想在其他系统（如CentOS、Alpine Linux等）上使用Kamailio，或者你想使用Kamailio最新的版本，甚至自己修改Kamailio的代码，则建议你直接从源代码安装，这样更便捷。

在此我们还是以Debian为例进行说明。笔者使用的电脑是Apple Mac M1，CPU是ARM芯片，目前还没有找到相应的Kamailio预编译版本，所以只能自己安装了。笔者使用安装了ARM版（aarch64）的Debian Bullseye（11）Docker镜像。

A.2.1 前期准备

安装前需要准备编译环境，即安装C语言编译器及相关的编译工具等，相关命令如下。

```
apt-get install git gcc g++ flex bison make autoconf pkg-config
```

如果你需要加密支持以及正则表达式等，那么以下依赖库可能会被用到。

```
apt-get install libssl-dev libcurl libcurl4-openssl-dev libxml2 libxml2-dev  
libpcre3 libpcre3-dev
```

如果你想安装其他附加的模块，如postgresql、mysql等，也需要安装相应客户端依赖库，具体如下。

```
apt-get install libpq-dev libmysqlclient-dev
```

创建一个目录并拉取源代码，具体命令如下。

```
mkdir kamailio # 创建一个目录来存放源代码  
cd kamailio # 进入该目录  
# 从Github上拉取源代码  
git clone --depth 1 --no-single-branch https://github.com/kamailio/kamailio  
cd kamailio # 进入源代码目录
```

A.2.2 编译安装

Kamailio源代码也是使用核心+模块的方式组织的，使用了类似Linux内核的模块配置方式。先执行以下命令生成模块配置文件。

```
make cfg
```

然后修改src/modules.lst，添加需要编译的模块，具体如下。

```
include_modules= db_postgres db_mysql
```

当然，这里的步骤也可以在执行make cfg时一起完成，一起完成的命令如下。

```
make include_modules="db_postgres db_mysql" cfg
```

如果你想指定安装的目标路径，可以使用如下命令。

```
make PREFIX="/usr/local/kamailio-devel" include_modules="db_postgres db_mysql" cfg
```

总之，准备好src/modules.lst以后，就可以进行Kamailio的编译安装了，相关命令如下。

```
make all          # 标准编译安装，也可以使用“make Q=0 all”查看更多编译输出信息  
make install    # 安装
```

A.2.3 安装路径



系统默认将Kamailio安装到/usr/local下，目录结构采用Linux标准的FHS 方式。比如，下列命令的可执行文件会被安装到/usr/local/sbin中。

- kamailio**: Kamailio服务器。
- kamdbctl**: 数据库操作脚本。
- kamctl**: Kamailio控制客户端脚本。
- kamcmd**: Kamailio控制命令行工具。

Kamailio模块默认安装到/usr/local/lib/kamailio/modules/或/usr/local/lib64/modules/中。Kamailio相关的文档可以在/usr/local/share/doc/kamailio/中找到，创建数据库的原始SQL代码在/usr/local/share/kamailio/目录中。当然，最重要的是，默认的配置文件是/usr/local/etc/kamailio/kamailio.cfg。

如果你在上面进行make cfg操作时指定了PREFIX，则所有内容都会安装到PREFIX指定的路径下，这样更方便进行查找和一键删除。

关于编译安装Kamailio更多的信息可以参考<https://kamailio.org/docs/tutorials-devel/kamailio-install-guide-git/>。

附录B

FreeSWITCH快速入门

Kamailio主要是一个SIP转发服务器，本身不能发起通话，不应答通话，也不处理媒体，因而一般与媒体服务器（如FreeSWITCH）配合使用。与Kamailio专注于处理信令相比，FreeSWITCH更侧重于处理媒体，执行各种不同的呼叫流程，如应答、放音、桥接通话、会议等。本书很多例子中都使用FreeSWITCH作为后端的媒体引擎。为了照顾不熟悉FreeSWITCH的读者，这里对FreeSWITCH进行简单介绍。

B.1 FreeSWITCH简介

FreeSWITCH既是一个SIP服务器，又是一个B2BUA（背靠背用户代理）。当有呼叫到达FreeSWITCH时，这路呼叫在FreeSWITCH中称为一条腿（Leg），也叫一个通道（Channel）。FreeSWITCH会根据主被叫号码等呼叫相关的参数查找拨号计划（Dialplan）。Dialplan相当于一个路由表，表中有一些Action，表明要执行的动作。典型动作有如下几个。

- **answer:** 应答。
- **echo:** 回声测试，自动应答，然后将收到的声音原样回放。
- **playback:** 放音，可以播放一个声音文件。
- **record:** 录音。
- **ivr:** 进入一个交互式语音菜单，需要事先配置，可以通过按键执行一些逻辑，如“查询余额请按1.....”等。
- **bridge:** 桥接，呼叫另一个终端或通过网关呼叫外网的电话。
- **conference:** 会议。

上述这些动作又叫Application，Dialplan的作用在于找到这些Application。不同的Application有不同的行为（上面已经看到了），也能控制不同数量的腿，如上面前5个都是单腿呼叫，bridge则可以控制两条腿（又发起一路呼叫），而conference则可以控制多条腿（在多人电话会议中实现混音、视频融屏等）。

需要特别说明的一点是，bridge会产生一条新腿。如A -> FreeSWITCH -> B的呼叫场景。FreeSWITCH执行bridge后，A -> FreeSWITCH与FreeSWITCH -> B是两条不同的腿，是背靠背的，



这也是B2BUA 的由来。而在Kamailio中，呼叫场景为A -> Kamailio -> B，Kamailio只负责信号转发，全程传递的都是一个SIP消息（当然Kamailio可以在中间改一些消息头域，但不改变本质），因而我们说它是一个Proxy（代理服务器）。

B.2 运行FreeSWITCH

从头安装和运行FreeSWITCH比较复杂，因此，笔者做了一个Docker镜像，可以方便大家使用和学习。在此假设你已经熟悉Docker，如果不熟悉的话也可以翻到附录D进行学习。

下面假设你在macOS或Linux环境中，并且已经安装好Docker、docker-compose和make工具，我们先看以下命令。

```
git clone https://github.com/rts-cn/xswitch-free.git
cd xswitch-free
make setup # 可选，生成.env，修改生成的.env 里的环境变量
make start # 启动 Docker 容器
```

首先，克隆本项目，然后进入xswitch-free目录，make setup会生成.env，.env里面是相关的环境变量，可以根据情况修改这些变量（一般至少要将EXT_IP改为你自己的宿主机的IP地址）。最后make start会以NAT方式启动容器。

启动后，你就可以将你称手的软电话注册到FreeSWITCH的IP地址上（默认端口为5060），用户名和密码任意。此时，你打电话就可以看到相应的日志，我们注册的两个不同号码可以互拨。

如果想进入控制台，可以打开另一个终端，这可以通过执行如下命令实现。

```
make cli
```

B.3 环境变量

该Docker容器涉及的环境变量及其默认值如下。

- SIP_PORT:** 默认SIP端口。
- SIP_TLS_PORT:** SIP TLS端口。
- SIP_PUBLIC_PORT:** SIP public Profile端口。
- SIP_PUBLIC_TLS_PORT:** SIP public Profile TLS端口。
- RTP_START:** 起始RTP端口。
- RTP_END:** 结束RTP端口。
- EXT_IP:** 宿主机IP地址或公网IP地址， SIP Profile中的ext-sip-ip及ext-rtp-ip默认会用到它。
- FREESWITCH_DOMAIN:** 默认的FreeSWITCH域。
- LOCAL_NETWORK_ACL:** 默认为none，在host网络模式下可以关闭。

B.4 配置

本镜像没有使用FreeSWITCH的默认配置。FreeSWITCH的默认配置为了展示FreeSWITCH各种强大的功能，设计得过于复杂，初学者难以理解，所以，我们使用了最小化的配置，目标是让使用者快速上手，并进一步打造自己的镜像和容器。

以下配置可以接受任何注册，也可以打电话。也就是说，你可以通过软电话使用任意的用户名和密码向FreeSWITCH注册。对于初学者而言，能打通电话是最重要的。

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

如果没有配置EXT_IP环境变量，需要将配置中如下内容注释掉，然后在fs_cli控制台上执行reload mod_sofia使配置生效。

```
<param name="ext-rtp-ip" value="${ext_rtp_ip}"/>
<param name="ext-sip-ip" value="${ext_sip_ip}"/>
```

B.5 常用命令

FreeSWITCH常用命令都在Makefile中，看起来也很直观。如果你的环境中没有make，也可以直接运行Makefile中的相关命令。

- make setup: 初始化环境，如果.env不存在，会从env.example中复制。
- make start: 启动镜像。
- make run: 启动镜像并进入后台模式。
- make cli: 进入容器和fs_cli。fs_cli是一个FreeSWITCH的客户端，可以连接到FreeSWITCH上实时查看日志和控制FreeSWITCH的目的。
- make bash: 进入容器和bash Shell环境。可以进一步执行fs_cli等。
- make stop: 停止容器。
- make pull: 更新镜像，更新后可以用make start重启容器。
- make get-sounds: 下载声音文件到本地，需要有wget工具。

如果没有安装Docker Compose，也可以直接使用Docker命令启动容器，具体方法如下。

```
docker run --rm --name xswitch-free \
-p 5060:5060/udp \
-p 2000-2020:2000-2020/udp \
-e ext_ip=192.168.7.7 \
-e sip_port=5060 \
-e sip_public_port=5080 \
-e rtp_start=2000 \
-e rtp_end=2010 \
ccr.ccs.tencentyun.com/xswitch/xswitch-free
```

由上述内容可以看出，采用上述方法需要输入很多参数，所以还是使用Docker Compose比较方便。

Windows用户可以使用build.cmd中相关的命令，如可以在命令行环境中执行build.cmd start来启动Docker容器。

B.6 修改配置

可以直接进入容器修改配置，并在fs_cli控制终端上执行reloadxml命令或重载相关模块使之生效，但在容器重启后修改过的数据将会被丢弃。

如果想保持自己的修改，那就需要把配置文件放到宿主机上。通过以下命令可以生成默认的配置文件。

```
make eject
```

完成上述操作后就可以修改docker-compose.yml了，这里我们取消掉以下行的注释。

```
volumes:  
- ./conf:/usr/local/freeswitch/conf:cached
```

修改后需要重启镜像，具体命令如下。

```
make stop  
make start
```

B.7 增加声音文件

为了压缩空间，这里我们没有将声音文件打包到镜像内。如果需要挂载声音文件，可以先执行 make get-sounds 命令来下载声音文件，然后修改 docker-compose.yml 的 volumes 配置，再按如下方法增加挂载。

```
volumes:  
- ./sounds/:/usr/local/freeswitch/sounds:cached
```

B.8 host模式网络

典型的Docker容器是通过NAT模式来运行的，但是，如果在Linux宿主机上，有时候使用host模式会比较方便（因为少了一层NAT）。本镜像不需要特殊的配置就可以使用host模式，只需要在 docker-compose.yml 中启用host模式。

如果环境变量中没有EXT_IP，则可能无法启动Sofia Profile，此时应禁掉default.xml和public.xml中的ext-sip-ip和ext-rtp-ip参数。

默认配置的是NAT模式，我们在Profile中启动如下配置。

```
<param name="local-network-acl" value="${local_network_acl}">
```

注意，local_network_acl的值是从LOCAL_NETWORK_ACL环境变量来的，默认值为none，它实际上是一个不存在的ACL，所以FreeSWITCH会认为任何来源的IP地址都会在NAT后面，因而对外总是使用EXT_IP环境变量里面的IP地址。

如果在host网络模式下，则可以在.env中注释掉LOCAL_NETWORK_ACL这个环境变量，让它使用默认值localnet.auto。

B.9 测试号码

在默认配置下，可以拨打表B-1所示测试号码。

表B-1 测试号码

号 码	说 明
9196	回音测试 Echo
888	XSwitch 技术服务电话
3000	进入会议
其他号码	查找本地注册用户并桥接，即你可以注册两个不同的用户互拨

FreeSWITCH相关的配置文件都在conf目录下。这里介绍的镜像采用的是一种最精简的配置，读者可以根据需要添加相应的配置来测试不同的功能。在本书的例子中，大部分使用了这些默认的配置，不过，为了能方便与Kamailio配合使用，相应的运行环境已经集成到了一起，可以在随书附赠的代码中看到相应的docker-compose编排文件。

更多信息可以查看<https://github.com/rts-cn/xswitch-free>。

附录C

Lua快速入门

本书介绍的Kamailio路由脚本绝大部分是使用Lua语言编写的，不熟悉Lua的读者可以通过阅读本附录快速入门。

Lua是一门小众语言，它可能不像其他语言（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[\[1\]](#)。

Lua非常简洁又非常强大，经常作为“胶水”语言嵌入各种软件中，比如FreeSWITCH、Kamailio、VLC、PostgreSQL、Wireshark中都有它，本书也主要以Lua语言做路由配置。当然，正因为它非常简单，所以即使你以前不熟悉Lua，在读完本章后，也可以轻松阅读本书。

Lua的语法非常简洁易懂，以致有人说：“如果你会其他编程语言，在30分钟内就能学会Lua。”

大家都知道，如果具有某种语言的编程经验，那么可以对比着学另一种新的语言。在这时里，我们对比JavaScript语言来学习Lua。

C.1 Lua与JavaScript的相似性

Lua与JS（JavaScript的缩写，下同）有很多相似的地方，简述如下。

(1) 变量无须声明：Lua与JS都是弱类型的语言（不像C），所以它们不需要事先声明变量的类型。

(2) 区分大小写：Lua和JS都是区分大小写的。true和false分别代表布尔类型的真和假，true与True、TRUE是完全不同的。

(3) 函数可以接受不定个数的参数：与JS类似，在Lua中，与已经声明的函数参数个数相比，实际传递的参数个数可多可少。举例如下。

```
function showem( a, b, c )
    print( a, b, c )
end

showem( 'first' )                                --> first    nil      nil
showem( 'first', 'second' )                      --> first    second   nil
showem( 'first', 'second', 'third' )              --> first    second   third
showem( 'first', 'second', 'third', 'fourth' )    --> first    second   third
```

在Lua中，不定长的参数列表使用“...”（称为vararg expressions）来表示，举例如下。

```
function showem(a, b, ...)
    local output = tostring(a) .. "\t" .. tostring(b)
    local theArgs = { ... }
    for i,v in ipairs(theArgs) do
        output = output .. "\t#" .. i .. ":" .. v
    end
    print(output)
end

showem('first')                                    --> first    nil
showem('first', 'second')                        --> first    second
showem('first', 'second', 'third')                --> first    second   #1:third
showem('first', 'second', 'third', 'fourth')     --> first    second   #1:third #2:fourth
```

(4) 哈希表可以用方括号或点方式引用：哈希表是编程语言中一种重要的数据结构。在Lua中，哈希表用Table来实现，在JS中用稀疏数组实现。无论如何，在两者中的哈希表都可以使用如下语法进行引用。

```
theage = gavin['age']
theage = gavin.age
```

(5) 数字：在JS和Lua中，整数和浮点数是没有区别的。它们在内部都是以浮点数表示。在Lua中，所有的数字类型都是number类型。

(6) 分号是可选的：JS和Lua类似，在不产生歧义的情况下，行尾的分号可以有，也可以没有，不同的是对待分号的方式。在JS中，按惯例是包含分号的，而在Lua中，按惯例是不包含分号的。

(7) 默认全局变量：在JS中，如果用var声明一个变量并赋值，则它是本地变量；如果不var声明，默认就是全局的。举例如下。

```

function foo()
{
    var jim = "This variable is local to the foo function";
    jam = "This variable is in global scope";
}

```

而在Lua中也类似，local声明一个本地变量，省略local则默认为全局变量。举例如下。

```

function foo()
    local jim = "This variable is local to the foo function";
    jam = "This variable is in global scope";
end

```

(8) 使用双引号和单引号表示字符串：在JS和Lua中，字符串是用引号引起来的，并且单引号和双



引号的作用没有任何不同。引号要配对使用，但这两种引号可以混合以避免使用转义符，必要时可以使用\来转义。举例如下。

```

local book = "Seven's Book";           --> Seven's Book
local book = 'Seven\'s Book';          --> Seven's Book
local book = '"Awsome" book of Seven'; --> "Awsome" book of Seven
local book = "\\" Awsome \\ book of Seven"; --> "Awsome" book of Seven

```

上述代码中，箭头后面为实际变量的值。

(9) 函数是一等公民：在JS和Lua中，函数是一等公民，这意味着，你可以将它赋值给一个变量，将它作为参数进行传递，或者直接加上括号进行调用。比如，在Lua中有如下情况。

```

1 mytable = { }
2 mytable.squareit = function( x )
3     return x * x
4 end
5 thefunc = mytable.squareit
6 print( thefunc( 7 ) ) --> 49

```

其中，第1行声明一个Table类型的变量mytable；第2~4行定义一个匿名函数，并将它赋值给mytable的squareit成员变量；第5行将上述成员变量的值又赋给了一个变量thefunc。至此，thefunc代表第2~4行定义的匿名函数。最后，在第6行就可以通过thefunc引用该匿名函数，该匿名函数对输入的参数7进行平方计算($x * x$)，最后得到的结果是49。

(10) 闭包：在JS和Lua中，函数都是闭包。简单来说，这意味着函数可以访问其在定义时可以访问的局部变量，尽管在以后调用时这些局部变量看起来已经“失效”了。比如，下面的Lua例子中，local n是一个局部变量，在执行makeFunction函数后，n的值本应失效，但却能在后面调用时照样使用它，详见代码内注释。

```

local a
function makeFunction()
    local n = 100
    return function (x)
        return n + x
    end
end
a = makeFunction() -- 执行上面定义的函数，这时变量a的值就是一个函数，即上面定义的匿名函数

```

```
-- 执行完上述函数后，变量 n 由于失去作用域应该失效，但由于“闭包”的特性，会在 a 这个变量引用该匿名函数期间有效
print(a(1)) # 执行 a 函数并打印结果，输入参数为 1，输出结果为 101
print(a(2)) # 输出结果为 102
```

C.2 区别

Lua与JS又有很多区别，简述如下。

1. 单行和多行注释

JS使用//做单行注释，而Lua中使用--。

JS使用“/*... */”来做多行注释，而Lua中使用“--[[...]]”。（注意，这里的“...”表示实际被注释掉的内容。）

JS中多行注释不能嵌套，解析器将在遇到第一个“*/”时终止该注释，而在Lua中可以使用类似“--[===[...]==]”这样的方式进行注释，并通过在方括号中加多个等号（前后等号的数量要匹配）来改变最外层的注释。

参考以下Lua注释。

```
-- 本行是单行注释
local jim = "This is not commented"

--[[[
local foo = "本行代码被注释掉了"
local bar = "本行代码也被注释掉了"
--]]

local jam = "本行是有效的"
---[[ 本行相当于一个单行注释，只有前面两个 -- 有效
local foo = "本行也是有效的，为什么？因为上一行是一个单行注释，管不到本行"
local bar = "本行也是有效的，原因同上"
--]] 本行也相当于一个单行注释，只有前面两个 -- 有效

--[=={ 把下面这些内容全都注释掉，直到最后一行（找到相匹配的同样个数的等号）
--[[
local foo = "foo"
local bar = "bar"
--]]
--]==]
```

2. 用end终止程序块

Lua与Ruby类似，使用end来代替JS中的大括号来终止程序块。下面是Lua中终止程序块的语法。

```
function foo( )
    --my code here
end

if foo( ) then
    --my code here

end

for i=0,1000 do
    --my code here
end
```

3. 使用nil代表空值

类似Ruby，在Lua中，使用nil代表空值，在JS及C语言中则使用null或NULL（在Kamailio原生脚本中使用\$null）。

在JS中，空字符串（""）和0在条件测试中都为假（false）。而在Lua中，nil和false是仅有的

非“真”值，其他所有测试结果都为“真”，如if(0)会返回真。

4.Lua中任何值都可以作为**Table**的键（Key）

在JS中，对象（Object）的所有键都是字符串（如myObj[11]与myObj["11"]是相同的），而在Lua中，字符串、数字，甚至另一个Table都可以是键。参见如下Lua代码。

```
a = {}  
b = {}  
mytable = {}  
mytable[1] = "The number one"  
mytable["1"] = "The string one"  
mytable[a] = "The empty table 'a'"  
mytable[b] = "The empty table 'b'"  
  
print( mytable["1"] ) --> The string one  
print( mytable[1] ) --> The number one  
print( mytable[b] ) --> The empty table 'b'  
print( mytable[a] ) --> The empty table 'a'
```

5.Lua中没有数组，任何复杂的数据类型都是**Table**

在JS里有明确的数组对象，并且有对应的操作数组的方法。举例如下。

```
var myArray = new Array( 10 ); // 声明一个新数组，有 10 个空元素  
var myArray1 = [ 1, 2, 3 ]; // 声明一个新数组，有 3 个元素  
myArray1.pop(); // 从数组中弹出最后一个元素
```

而在Lua中，对象是Table，prototype是Table，哈希表是Table，数组是Table，Table是Table，总之什么都是Table。

Lua中的所谓数组，本身就是一个Table，它相当于JS里的稀疏数组，只是它的第一个值是从1开始的，而不是0。可以使用Table的语法来创建数组。下面Lua代码中的两种方法是等价的。

```
people = { "Gavin", "Stephen", "Harold" }  
people = { [1]="Gavin", [2]="Stephen", [3]="Harold" }
```

如果拿Table当数组来用的话，可以使用两种方法来获取和设置数组的大小，并允许数组中有空值存在。举例如下。

```
people = { "Gavin", "Stephen", "Harold" }  
print( table.getn( people ) ) --> 3
```

```

people[ 10 ] = "Some Dude"

print( table.getn( people ) )           --> 3
print( people[ 10 ] )                  --> "Some Dude"

for i=1,table.getn( people ) do
    print( people[ i ] )
end
--> Gavin
--> Stephen
--> Harold

table.setn( people, 10 )
print( table.getn( people ) )           --> 10

for i=1,table.getn( people ) do
    print( people[ i ] )
end
--> Gavin
--> Stephen
--> Harold
--> nil
--> nil
--> nil
--> nil
--> nil
--> Some Dude

```

6.数字、字符串以及Table都不是对象

与面向对象概念里面的对象不同，数字、字符串以及Table本质上都不是对象。

Lua仍然是面向过程的语言，大多数操作可以通过库函数实现。举例如下。

```

print(string.len(mystring) ) --> 11
print(string.lower(mystring)) --> hello world

```

其中，`string.len`是一个函数，实际上`string`本身是一个Table。从5.1版本起，Lua也支持一些类似面向对象的语法，如上面的例子等价于如下代码。

```

mystring = "Hello World"
print(mystring:len() ) --> 11
print(mystring:lower()) --> hello world

```

在上面的例子中，通过使用“`:`”把`mystring`看成一个对象，`len`和`lower`就类似于这个对象的方法。实际上，Lua是在内部对`mystring`这个假的“对象”做了特殊处理。找到`mystring`所对应的数据类型(`string`)后，调用实际的`string.len()`，并把`mystring`作为该函数的第一个参数，所以`mystring:len()`与`string.len(mystring)`是等价的（这种方法通常称为语法糖）。

7.没有`++`，没有`+=`

在Lua中没有`++`和`+=`这样的缩写形式，所以变量自加必须用以下方式。

```

local i = 0;

i = i + 1;

```

字符串的拼接是使用“`..`”操作符实现的，举例如下。

```

local themessage = "Hello"
themessage = themessage .. " World"

```

如果把一个字符串和一个数字相加，Lua会试图将字符串转换成数字。举例如下。

```
print(10 + "2")      --> 12
print(10 + "a")      --> 出错: attempt to perform arithmetic on a string value
```

8.没有三目运算符

JS或C中的“a ? b : c”是很贴心的，但在Lua中没有这样的语法，不过其有一个短路语法与该语法类似。举例如下。

```
local foo = (math.random() > 0.5) and "It's big!" or "It's small!"

local numusers = 1
print( numusers .. " user" ..
  (numusers == 1 and " is" or "s are") .. " online.")
--> 1 user is online.

numusers = 2
print( numusers .. " user" ..
  (numusers == 1 and " is" or "s are") .. " online.")
--> 2 users are online.
```

9.模式匹配

正则表达式是很方便的字符串匹配工具。与JS以及其他语言不同，为了保持Lua的小巧且减少Lua对其他库的依赖，Lua没有使用常用的POSIX正则表达式（regexp），也没有使用PCRE（Perl兼容的正则表达式），而是使用自己实现的模式匹配算法，其相关实现语法也与JS有很大不同。

模式（Pattern）实际上就是Lua中的正则表达式。它与普通正则表达式最大的不同就是使用%而不是使用\来进行转义。Lua模式及其说明如表C-1所示。

表C-1 Lua模式及其说明

模 式	说 明	模 式	说 明
.	所有字符	%s	空白字符
%a	字母	%u	大写字母
%c	控制字符	%w	字母和数字
%d	数字	%x	十六进制数字
%l	小写字母	%z	内部表示为 0 的字符
%p	标点符号		

除此之外，它还有一些具有魔法含义的特殊字符，这些字符有()、..、%、+、-、*、?、[]、^、\$。它们的含义都与PCRE里相应的字符差不多，其中-的含义与*类似，都是重复前一个字符0次或多次，不同的是，*为最大匹配，它会尽量匹配更长的字符串，而-为最小匹配，它会尽量匹配更短的字符串。

C.3 其他

在本书中，我们大部分都在用Lua来写Kamailio路由脚本。作为对比，我们可以看一个在FreeSWITCH中执行Lua脚本的例子。

在FreeSWITCH中，当有呼入并执行到Lua脚本时会自动生成一个session对象，因而可以在Lua脚本中使用类似面向对象的语法特性进行编程，比如以下脚本可以用于播放欢迎音。

```
session:answer()      -- 应答
session:sleep(1000)    -- 等一会媒体（毫秒）。在 PSTN 环境中媒体建立可能比较慢
session:streamFile("/tmp/hello-lua.wav") -- 播放声音文件
session:hangup()      -- 挂机
```

最后，向大家推荐一本学习Lua的书——《Lua程序设计（第四版）》（原名 *Programming Lua*）。另外，英文好的读者可以参考另一本书 *Programming in Lua*，该书第一版是免费的，见 <http://www.lua.org/pil/>。

[1] 我相信有很多人知道它是缘于2010年一则新闻，新闻中说一个14岁的少年用Lua编出了iOS版的名为 *Bubble Ball* 的游戏，该游戏的下载量曾一度超过《愤怒的小鸟》。

附录D

Docker简介及常用命令

随着云计算及云原生的发展，Docker基本上成了事实上的部署方式。对于运维来讲，使用Docker镜像非常简单（当然制作Docker镜像还是有些门槛的），因而在本书的例子中主要基于Docker镜像和Docker容器给大家讲解相关知识。为了照顾不熟悉Docker的读者，下面我们将对Docker进行简单介绍。

D.1 Docker简介

为了帮助大家理解Docker，我们需要先来介绍一下虚拟机。大家都知道，计算机有CPU、内存、硬盘、网卡等基本硬件，而为了高效使用这些硬件，需要一个操作系统来管理和维护它们，典型的操作系统有Windows、Linux、macOS等（手机其实也是一台计算机，有Android和iOS等操作系统）。一般来说，一台计算机上只能运行一个操作系统，为了能同时运行多个操作系统，人们发明了虚拟机。虚拟机就是使用软件模拟CPU、内存、硬盘和网卡等，这样就可以在操作系统中套操作系统（类似于俄罗斯套娃）。相对于虚拟机，运行原来的操作系统的主机就称为宿主机。常见的虚拟机软件有VMWare、Virtual Box、Xen及KVM等。

虚拟所有硬件会有些慢，也会有些重。在Linux内核中，有一个轻量级的东西叫control groups（即Cgroups），它可以做资源控制、进程控制和隔离。使用Cgroups做出来的虚拟化技术叫LXC（Linux Container），由于LXC只是使用了资源隔离，而不需要像虚拟机那样将虚拟机里全部的CPU指令“翻译”成宿主机的指令，因而更轻。

但是LXC用起来比较麻烦，因而其虽然出现了很多年但一直流行不起来，直到Docker出现。

Docker其实并不是什么虚拟化技术，它只是提供了一组工具，可以方便地生成和管理镜像、启动虚拟化容器等。所以，通过Docker实现的虚拟化系统也不再叫虚拟机，而叫容器。也就是说，在一个Linux操作系统上，可以跑很多不同的容器，不同容器之间的资源（如CPU、进程、内存、网络、硬盘空间等）都是隔离的，不同容器里的内容可以使用不同的资源，包括不同版本的应用程序或依赖库等，它们彼此独立运行，但共用操作系统内核。Docker只适用于Linux，也就是说，宿主机和服务器必须都是Linux系统。

虽然在Linux宿主机上运行Docker的开销很小，但人们还想在macOS及Windows上运行Docker。为了实现这个目标，最早人们都是以虚拟机的方式实现的，如基于Virtual Box实现。但后来，macOS上有了性能更高的Hypervisor.Framework（xhyve是它的具体实现），Windows上也出现了WSL2



，这些技术可以更好地支持虚拟化。

还有一个比较有用的工具叫Docker Compose，它使用一组YAML格式的编排文件，通过它可以更方便地管理很多容器。最初该工具是单独提供的，不过最新版本的Docker已经整合了该工具。

D.2 Docker安装

如果你想使用Ubuntu或者Debian Linux操作系统，可以使用如下命令一键安装Docker。（通过--mirror参数使用阿里云的镜像进行安装速度会快一些。）

```
curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
```

或使用DaoCloud提供的一键安装方式，具体如下。

```
curl -sSL https://get.daocloud.io/docker | sh
```

在其他Linux系统（如CentOS、RHEL等）上以及macOS、Windows上进行安装的方法也有所不同，限于篇幅我们就不一一介绍了，<https://docs.docker.com/engine/install/>上列出了在各种操作系统上安装Docker的方法，大家可以自行学习。

下面几个链接上也有中文的安装说明，大家可以参考。

- <https://www.runoob.com/docker/windows-docker-install.html>。
- <https://www.runoob.com/docker/ubuntu-docker-install.html>。
- <https://www.runoob.com/docker/macos-docker-install.html>。

D.3 基本概念

这里我们简单介绍与Docker相关的一些基本概念。

- 镜像：即Docker Image，里面是一些文件，相当于一个硬盘。镜像是分层的，便于传输和分享。如果日后镜像有更新，可以只下载更新过的层，没动过的层不需要重新下载。
- Tag：镜像有一个Tag属性，相当于给镜像打一个记号。Tag是可选的，它就是一个字符串，如果没有Tag，则默认为latest。Tag通常用于标志镜像的版本。
- 容器：通过一个镜像可以启动一个容器，它是一个隔离的运行环境（可以认为是个轻量级的虚拟机），可以进入一个容器的内部执行命令。Docker容器在运行期间会保存一个临时的镜像，用于存储变动过的文件。容器重启临时镜像的内容还会保留，直到容器被彻底删除。
- 网络：网络最典型的使用方法是使用NAT模式，在Linux宿主机上也可以使用host模式，但后者实际上是破坏了网络隔离。
- Docker Hub：与GitHub（众所周知的代码仓库聚集地）类似，Docker hub是Docker镜像的聚集地。除Docker Hub外，其他云厂商也提供镜像服务，如switch-free镜像就存储在腾讯云上。

D.4 常用命令

以下命令经常用到。

启动一个容器的命令如下。

```
docker run hello-world  
Hello from Docker!
```

在首次需要一个镜像时，Docker会自行从Docker Hub（或指定的其他镜像服务器）上下载。其中Hello from Docker是镜像启动后打印的内容，打印后即退出，容器也会退出。如果想让镜像启动后不退出，可以运行一个永远不退出的程序。比如，以下命令用于运行alpine（它是一个小的Linux发行版，非常小）镜像的Shell。

```
docker run -it alpine sh
```

其中，-it参数表示开启交互式终端。进入Shell环境后就可以在容器内部执行一些命令了。在宿主机上换一个终端容器，可以使用docker ps命令列出所有正在运行的镜像。举例如下。

```
# docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
873ea4a68dc0 alpine "sh" About a minute ago Up
```

有了Container ID，就可以在宿主机上进入容器了。

```
docker exec -it 873ea4a68dc0 sh
```

当然，Container ID是自动生成的，在启动Docker镜像时可以用--name指定一个好记的名字，如以下代码在容器中启动时指定容器的名字为alpine-test。

```
docker run --name alpine-test --rm alpine sh
```

上述代码中，--name用于指定一个名字（alpine-test）；--rm表示容器停止运行后，自动删除相关资源。如果在容器运行期间创建了一个文件，且当初窗口启动时使用了--rm参数，则重启后刚才创建的文件就不存在了；相反，如果启动镜像时并没有使用--rm参数，那么重启镜像后上述文件内容还在。

容器有了名字，以后进入容器就可以把Container ID换成对应的名字了，如在下面的命令中使用了容器的名字alpine-test作为参数，而不是随机生成的Container ID。

```
docker exec -it alpine-test sh
```

在容器中可以使用exit退出Shell，如果退出的是最初启动的那个Shell，那么容器就自动退出了。

其他常用的参数还有-p（做NAT端口映射）、-e（设置环境变量）等，我们在此就不详细讲了，在5.1.2节和5.1.3节有相关的例子。

在宿主机上可以查看标准输出（STDOUT）的日志，命令如下。

```
docker logs alpine-test
```

通过-f参数可以进入Follow模式，即跟踪日志输出永不退出，具体命令如下。

```
docker logs -f alpine-test
```

停止容器的命令如下。

```
docker stop alpine-test
```

彻底删除容器所有缓存的命令如下。

```
docker rm alpine-test
```

在Docker运行期间，会产生大量缓存，如自动下载的镜像，如果我们没有使用`-rm`参数来自动删除容器产生的缓存，也没有使用上面的`docker rm`命令有针对性地进行清除，时间长了这些缓存会占用大量硬盘空间。可以使用如下命令删除缓存。（注意，因为这个命令会删除一些东西，所以在使用前确保知道你自己在做什么，如果不确定的话最好多看看Docker手册或找个同事帮助你一块看着。）

```
docker system prune -a
```

D.5 Docker Compose

此处以随书附增的示例代码中的Docker Compose文件为例（docker/kam.yml）。

```
version: "3.3"          # 配置文件的版本号

services:
  kb-kam:             # 服务，可以有多个
    container_name: kb-kam      # 定义一个 Kamailio 服务，名字任意
    image: kamailio/kamailio-ci:5.5.2-alpine # 使用的镜像，来自 Docker Hub
    # restart: always           # 崩溃后是否自动重启，在生产环境中经常使用
    env_file: .env            # 从该文件中自动导入一些环境变量
    stdin_open: true           # 是否启用标准输入
    tty: true                 # 是否启用控制台，该参数和 stdin_open 参数同时开启可以在控制台使用键盘输入
    # command: ["/bin/sh"] # 容器启动后执行的命令，如果没有则执行构建时 Dockerfile 中指定的命令
    privileged: true          # 特权模式，如是否可以在容器中运行 tcpdump 抓包或 gdb 调试
    entrypoint:              # 启动后自动执行的命令
      - /bin/sh
      # - /start-kam.sh
    volumes:
      - ./etc:/usr/local/etc/kamailio:cached
      - ./etc:/etc/kamailio:cached
      - ./start-kam.sh:/start-kam.sh
    networks:
      - kamailio-example        # 指定网络，这里我们指定了一个外部网络，方便跟其他容器共享
    ports:
      - "${KAM_SIP_PORT}:${KAM_SIP_PORT}" # 映射端口，这里引用了环境变量，在 .env 中设置
      - "${KAM_SIP_PORT}:${KAM_SIP_PORT}/udp"
    networks:
      kamailio-example:
        external: true
```

其中，外部网络使用 docker network create kamailio-example 命令创建，这样可以将 Kamailio 容器与 FreeSWITCH 容器启动到同一个网络上（使用相同的网络地址段）。

可以通过以下命令启动容器。（如果你使用新版本的 Docker，也可以将 docker-compose换成 docker compose，即把-换成空格。）

```
docker-compose -f docker/db.yml up      # 启动到前台，可以在当前 Shell 中进行查看日志、输入命令等操作
docker-compose -f docker/db.yml up -d    # 启动，进入后台模式，不占用当前 Shell
```

上述命令中，如果把 up换成 down 则可以停止服务。从上面可以看出，有了 YAML 编排文件，可以比原始的 Docker 少输入很多命令行参数。另外也可以将多个服务写到同一个 YAML 文件中，同时启停多个服务、管理它们的依赖关系等。

这里只是简单介绍了 Docker 和 Docker Compose，了解了这些内容你就可以顺利阅读本书了。更多的关于 Docker 使用方法的介绍，还需要读者自行去找相关资料。

附录E 模块索引表

Kamailio是一个模块化结构的系统，由核心和外围模块构成。Kamailio的核心很紧凑，大部分功能，甚至连SIP事务处理(tm)这样的基本功能，都是在模块中实现的。Kamailio模块数量很多，其5.6版的官方文档中列出了近250个。肯定不会有人同时用到所有的模块，事实上，常用的模块也就几十个。但即使只是几十个，如果全部详细讲明白所有相关的参数和函数，也需要好几本书才行。

本书并没有直接罗列所有的模块和参数，而是把模块的使用融入实际的案例中，这样不仅节省篇幅，读起来也能更生动。当然在本书中，对于各个模块，我们只是讲了一些重点的、比较有代表性



的参数和函数的使用方法，更多的内容还需要读者自行查阅相关模块的说明文档。

为便于读者根据模块名字找到书中相关的案例，我们在此做了一个简单的索引表格，供读者查阅。

模 块	说 明	所在位置
pike	并发数跟踪模块	2.3 节
core	核心函数	3.1 节
htable	哈希表存取模块	3.2 节
KEMI	脚本语言嵌入式模块	第 4 章
sqlops	SQL 处理模块	4.4.2 节, 7.2 节
sipdump	SIP 日志模块	5.6.1 节

(续)

模 块	说 明	所在位置
tm	事务处理模块	6.2.3 节
dispatcher	负载均衡模块	6.4.2 节
permissions	权限许可模块	6.4.3 节
geoip2	IP 地理位置查询模块	6.4.4 节
xhttp	HTTP 服务端模块	6.5.1 节
http_client	HTTP 客户端模块	6.5.1 节
http_async_client	HTTP 异步客户端模块	6.5.1 节
rtjson	基于 JSON 的路由模块	6.5.2 节
evapi	事件 API 模块（自定义协议）	6.5.3 节
mmtree	动态树模块	8.1 节
dialplan	拨号计划模块	8.2 节
lcr	低成本路由模块	8.3 节
prefix_route	字冠路由模块	8.4 节
drouting	动态路由模块	8.5 节
speeddial	缩位拨号模块	8.6 节
alias_db	用户别名模块	8.7 节
carrieroute	运营商路由模块	8.8 节
pdt	字冠 – 域名翻译模块	8.9 节
usrloc	用户位置模块	8.10 节
db_redis	Redis 数据库连接模块	8.10 节
uac	用户代理客户端模块	8.11 节
acc	记账模块	8.13 节
path	路径模块	8.14.1 节
nathelper	NAT 穿透模块	8.14.2 节
rtpengine	RTP 引擎模块	8.14.3 节
topoh	拓扑隐藏模块	8.14.5 节
websocket	WebSocket 模块	8.15 节
pipelimit	流量限制模块	10.2.3 节
dialog	对话模块	10.2.5 节

后记

至此，本书要跟大家说再见了。作为本书的作者，我其实还有很多的话要跟大家说，有更多实例想跟大家分享，但由于时间以及篇幅的关系，我不能太任性地写下去。作为创业者，我在很多真实的项目中使用了Kamailio，基于此支撑了大量的并发呼叫；作为技术爱好者和程序员，我编写了大量的Kamailio路由脚本，也阅读了一些Kamailio的源代码，对Kamailio项目的源代码及文档也有一点点贡献。本书中使用的例子，有的来自网上公开的资料，有的来自我及我的团队的实践。我花了大量的时间精选和测试这些实例，就是希望给读者一个全方位的参考，同时保证相关内容不至于太复杂。

使用Lua写路由脚本是本书的一大特色，在国际上这也是首创。当然，为了帮助大家阅读原生语言编写的路由脚本，我们在本书中也对原生的概念和函数做了比较详尽的说明。总之，通过对本书的学习，希望大家都能充分理解Kamailio的特点和本质，写出更有创造力的路由脚本，做出更完美的解决方案。

Kamailio历经20年而不衰，且与时俱进、不断进步，足以说明其具有强大的生命力。祝愿Kamailio开源社区明天会更好，再辉煌20年。也希望大家多向行业里的朋友们推荐Kamailio，即使他们不用Kamailio，但相信阅读本书对于他们深入理解SIP、VoIP及RTC通信都有很大的帮助。

最后，也希望有实践经验、有开发能力的读者，能深度参与Kamailio技术社区的讨论和交流，为开源项目做贡献，与开源社区共同发展。同时也希望本书能真正为大家带来帮助，帮大家为行业、为社会提供更多、更好、更稳定的SIP服务。未来是属于我们大家的。

作者简介

杜金房
(网名: seven)



资深网络通信技术专家，腾讯云最具价值专家（TVP）。

在网络通信领域耕耘20多年，精通VoIP、SIP、Kamailio、OpenSIPS、FreeSWITCH及WebRTC等各种网络协议和技术。有10多年的Kamailio和FreeSWITCH应用及开发经验，服务过多家国内通信服务厂商，以及美洲、欧洲、东南亚国家的通信服务厂商。

FreeSWITCH-CN中文社区创始人兼执行主席，国内FreeSWITCH布道者。2021年创办实时解决方案(RTS.cn)社区，专注于SIP、RTC、AI等各种通信技术，通过对话和交流，探讨开源与商业的最佳结合方案。

10余年来，在国内不同城市举办了数十场FreeSWITCH及Kamailio技术培训，很多学员来自运营商、通信设备厂商、大型互联网公司等。

自2011年起每年应邀参加在美国芝加哥举办的ClueCon全球VoIP开发者大会，并发表主题演讲。

著有《FreeSWITCH权威指南》（2014年出版），该书被誉为学习FreeSWITCH的必读物，从业者几乎人手一本。

精通C、Go、Lua、Erlang、Ruby、Javascript等语言。

吕佳婷

烟台大学物理化学硕士，烟台台海玛努尔核电设备有限公司工程师。在化学分析、数据分析程序设计、实验数据维护与数据挖掘等方面有丰富的实践经验，对程序设计和电子通信也有较多研究。