HZ BOOKS
华章科技

信息安全
技术丛书

iOS
应用逆向工程

第2版

沙梓社 吴航 著

iOS App Reverse Engineering
Second Edition

- 全球第一本讲解iOS8应用逆向工程的实战手册，作者毫无保留地分享了数年来在iOS逆向工程领域的经验。
- 内容系统深入，逻辑紧密，实战性强，从iOS系统架构等理论出发，以多个实例贯穿全书，阐述class-dump、Theos、Cycript、Reveal、IDA、LLDB等常用工具的使用，通俗易懂。
- 总结提炼出一套从UI观察切入代码分析的iOS应用逆向工程方法论，授人以渔。

机械工业出版社
China Machine Press

信息安全技术丛书

iOS应用逆向工程（第2版）

沙梓社　吴航　著

ISBN：978-7-111-49436-2

本书纸版由机械工业出版社于2015年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

推荐序一

推荐序二

第2版序

第1版序

前言

第一部分 概念篇

第1章 iOS逆向工程简介

1.1 iOS逆向工程的要求

1.2 iOS应用逆向工程的作用

1.3 iOS应用逆向工程的过程

1.4 iOS应用逆向工程的工具

1.5 小结

第2章 越狱iOS平台简介

2.1 iOS系统结构

2.2 iOS二进制文件类型

2.3 小结

第二部分 工具篇

第3章 OSX工具集

3.1 class-dump

3.2 Theos

3.3 Reveal

3.4 IDA

3.5 iFunBox

3.6 dyld_decache

3.7 小结

第4章 iOS工具集

4.1 CydiaSubstrate

4.2 Cycript

4.3 LLDB与debugserver

4.4 dumpdecrypted

4.5 OpenSSH

4.6 usbmuxd

4.7 iFile

4.8 MTerminal

4.9 syslogd to/var/log/syslog

4.10 小结

第三部分 理论篇

第5章 Objective-C相关的iOS逆向理论基础

5.1 tweak在Objective-C中的工作方式

5.2 tweak的编写套路

5.3 实例演示

5.4 小结

第6章 ARM汇编相关的iOS逆向理论基础

6.1 ARM汇编基础

6.2 tweak的编写套路

6.3 LLDB的使用技巧

6.4 小结

第四部分 实战篇

第7章 实战1：Characount for Notes 8

7.1 备忘录

7.2 搭建tweak原型

7.3 逆向结果整理

7.4 编写tweak

7.5 小结

第8章 实战2：自动将指定电子邮件标记为已读

8.1 电子邮件

8.2 搭建tweak原型

8.3 逆向结果整理

8.4 编写tweak

8.5 小结

第9章 实战3：保存与分享微信小视频

9.1 微信

9.2 搭建tweak原型

9.3 逆向结果整理

9.4 编写tweak

9.5 彩蛋放送

9.6 小结

第10章 实战4：检测与发送iMessage

10.1 iMessage

10.2 检测一个号码或邮箱地址是否支持iMessage

10.3 发送iMessage

10.4 逆向结果整理

10.5 编写tweak

10.6 小结

越狱开发一览

沙箱逃脱

编写tweak——新时代的hacking

推荐序一

前一段时间跟吴航在微博上私信聊天，他说正在写一本iOS方面的图书，我让他书出来的时候送一本给我。之后他在私信上跟我说书写完了，让我给写个序，我当即表示“压力山大”，但还是欣然答应了。

我认识吴航是在2011年9月，当时安全管家在找iOS开发高手，吴航作为我们安全管家iOS开发组的第一个工程师进来了，他和我们从零开始搭建iOS团队，并着手安全管家越狱版的开发。到了2012年年中，我认识到iOS自身的安全性非常好，在非越狱的iOS上我们能做的关于安全的事情并不多，而越狱行为本身就是一个最大的安全风险，是用户主动选择的结果，跟我们自身的安全理念不符，因此无须投入太多关注，恰好吴航本人也想出去做自己的事情，于是我支持了他的决定。

在那大半年的接触中，我发现吴航是个难得的技

术人才，在技术的钻研上有股子狠劲，拥有丰富的开发实战经验，又善于利用各种工具解决问题，因此在他带团队的时候，评估出来的开发进度基本上都能达成。我印象深刻的有两件事，第一件是在开发越狱版安全管家时，由于这方面官方公开的资料几乎没有，很多涉及系统底层的开发，需要自己摸着石头过河，并得反复尝试。当时我们制定了一个比较紧的开发周期，希望在较短的时间内开发出越狱版安全管家的原型。吴航的压力不小，他接连几个月都在研究系统底层，向各路高人请教，通过Google找寻国外网站上的资料，没日没夜地想办法，后来终于在既定的时间内完成，这让我看到了吴航不仅不惧困难，而且敢于负责任。另一件事是在开发App Store版安全管家时，有个版本在我的手机上在不同页面间快速切换时会有极小的概率导致安全管家崩溃，我就反馈给了吴航。虽然只是小概率事件，但他亲自反复高强度测试，细致地排查代码，最终揪出了导致这个问题的内存指针

Bug，这足见其严谨的技术态度和对质量高标准的追求。

虽然我不做开发很多年了，但是至今仍忘不了年轻时作为一个工程师，非常渴望与更高水平的人交流，希望聆听高手们实战经验分享的情景。吴航愿意把他的经验总结成书，是广大iOS开发者的福音，这本书能够带给大家实实在在的干货，让大家都能在技术的道路再攀高峰。

赵岗

安全管家创始人

推荐序二

In our lives,we pay very little attention to things that work.Everything we interact with hides a fractal of complexity—hundreds of smaller components,all of which serve a vital role,each disappearing into its destined form and function.Every day,millions of people take to the streets with phones in their hands,and every day hardware,firmware,and software blend into one contiguous mass of games,photographs,phone calls,and text messages.

It holds,then,that each component retains leverage over the others.Hardware owns firmware,firmware loads and reins in software,and software in turn directs hardware.If you could take control of one of them,could you influence a device to enact your own desires?

iOS 8 App Reverse Engineering provides a unique

view inside the software running on iOSTM,the operating system that powers the Apple iPhone®and iPad®.Within,you will learn what makes up application code and how each component fits into the software ecosystem at large.You will explore the hidden second life your phone leads,wherein it is a full-fledged computer and software development platform and there is no practical limit to its functionality.

So,young developer,break free of restricted software and find out exactly what makes your iPhone tick!

（在生活中，我们经常会忽略许多习以为常的事物。事实上，那些我们每天都与之打交道的东西，往往都蕴含了一种“复杂”的美感——它们由成百上千的微小组件构成，各个微小组件各司其职，在各自的岗位上发挥着不可替代的关键作用。现代生活中，智能手机已经成了我们每天必不可少的工具，通过硬件、

软件和固件协同合作，它为我们带来了好玩的游戏、有趣的照片，以及便利的沟通渠道——电话和短信。

在一个巴掌大的手机里，各个组件之间的关系错综复杂，互相影响。硬件为固件的运行提供支撑平台，固件掌管软件，而软件又回过头来调度硬件。如果你能控制它们之中的哪怕一个，不就可以让手机听命于你了吗？但App Store的插手，又为你对它们的控制加上了重重阻力。

本书从独特的角度剖析iOS应用，你会从比App Store App更低一级的深度去了解软件的各个组件在构造整个软件时起到的作用，你会由此发现手机的“里世界”——它的能力远不止App Store所许可的那样有限，确切地说，它是一台功能齐全的计算机，在它的“里世界”里，一切皆有可能。

年轻的开发者，从这里开始打破App Store的限制，重新认识真正的iPhone吧！）

Dustin L.Howett

iPhone Tweak开发者

第2版序

转眼，本书第1版面世已经快1年了，在这一年里，因为有大家的认可与推广，本书得到了广泛关注。与此同时，iOS逆向工程也在国内iOS开发者圈子里“漫延”开来，并达到了前所未有的高度，正是因为大家的努力，才使得该技术得到发展，而我们作者团队也贡献了一点力量，甚感欣慰！

随着盘古、太极等国内越狱团队的横空出世，以及各种第三方市场的蓬勃发展，iOS技术层面的较量已经从App开发转向底层研究；随着WireLurker等病毒的出现，一些深藏不露的安全问题也开始浮出水面，苹果构建的封闭系统正在出现一条条裂缝。不管是进攻还是防守，都离不开iOS逆向工程技术的使用。在可以预见的未来，苹果将进入恶意软件重度防御时代[1]，iOS逆向工程的应用一定会越来越广泛，让我们拭目以待。

自本书第1版上市后，反响一直不错，京东等各大网店的好评率高达95%以上。随着iOS 8的发布，我们清楚地意识到第1版的内容已经不再适合最新的iOS 8。同时根据一年多以来跟大家不断的沟通和交流，也意识到第1版存在缺憾和不足，例如讲解不够细致，术多道少等，影响了书的可读性。因此，在即将推出全新升级的《iOS应用逆向工程》里，不但全面支持iOS 8，还大幅更新了章节内容，涵盖更多细节，配备了更多的例子，增加了“道”的分量，比第1版的逻辑性更强，更易读了。在升级版中，我们尝试从抽象的逆向工程中抽离出一个通用的方法论，试图传递给大家一种逆向工程的思想，而不仅仅是工具的使用。

本书第1版上市之后，我曾把书的目录和内容框架发布到国际iOS越狱社区上，得到了非常正面的反馈，包括Cydia的作者saurik、OSX著名研究员fG!、Theos作者DHowett等国际一线开发者均对本书表示了

浓厚的兴趣，这也让我萌生了让该书走向国际的想法。在整理这一版时，我与编辑沟通了该想法，没想到还真促成了此事。国际版将由美国CRC出版社在全球出版发行，由8位国外知名研究员（5位美国籍、1位加拿大籍、1位阿根廷籍、1位丹麦籍）审核，全球iOS逆向工程社区对国际版寄予了厚望。在第1版的前言里，航哥曾提到我夸下的海口：“弟的目标远大，要玩就朝着国际一线大牛的目标去！”虽然离这个目标还差得很远，但我已经在朝这个目标努力迈进了，不是么？

[1] http://www.feng.com/Story/2015-Apple-will-enter-the-era-of-malware-severe-defense_602029.shtml.

第1版序

我是一个热爱自助旅游的人。在大学的每个寒暑假，我都会抽出7~10天的时间，挑选中国的一个地方当一次背包客。因为是自助游，没有导游帮你安排好一切，所以在出行前，我和同伴需要花费不少时间制订计划、确认路线、购买车票，考虑路上可能出现的状况，并思考对策。都说旅游能够开阔眼界，自助游尤其如此——在路上见到的人和事让我增长见识，更重要的是，在开始这段旅程前，我就需要对旅程中的点点滴滴有所准备——当身体还站在起点时，我的头脑就已经到达终点了，这种思维方式对全局观的培养是有利的，也让我在思考其他问题时形成了从长计议的思维方式。

因此在2009年攻读硕士研究生前，我就曾深入思考过自己想要从事的研究方向。我学习的是计算机专业，而从本科开始，身边绝大部分同学的研究平台是

Windows。作为一名动手能力并不强的普通学生，如果我继续从事Windows的研究，有两点好处：

· 这个方向有海量成熟资料，我的学习之路上不会缺少参照；

· 研究人数众多，碰到问题可以请教讨论的人比比皆是。

但是，从另一个侧面来看，这两点“好处”也并非有百利而无一害：

· 参考的资料越多，意味着我会越多地重蹈前人的“覆辙”；

· 研究人数越多，意味着竞争压力越大。

总的来说，如果我从事Windows相关的工作，起步会很顺利，但后续难保不被淹没在人海里；如果另寻他路，入门会很辛苦，但坚持下去或能独辟蹊径。

幸运的是，我的想法与导师的如出一辙。他推荐我选择当时国内“小荷才露尖尖角”的移动开发方向，而我在这之前一直使用的是一款飞利浦蓝屏手机，对智能手机毫无概念，更别说在上面开发软件了。但是，导师是我仔细分析所有硕士生导师特点，与数名学长交换意见之后谨慎挑选的师从对象，他的判断本身就含有我的判断，因此，我相信这个判断，于是开始搜寻移动开发的相关资料。仅仅是了解了一些移动互联网和智能手机的概念，我就隐隐发现，这个行业顺应了人们对计算机和互联网更小、更快，与生活融合更紧的历史发展趋势，一定大有作为，遂将研究方向定为iOS。

万事开头难，iOS与我熟悉的Windows有着太多太多的不同：类UNIX、完整的生态系统、全封闭、Objective-C语言，还有对我影响最深的“越狱”，这一切的一切在当时几乎找不到完整的参考资料，有半年多近一年的时间，我折腾黑苹果的时间要以星期为单

位。我硬着头皮把《Objective-C基础教程》上宛如天书一般的Objective-C代码敲入Xcode，然后运行模拟器看效果，但代码和画面完全对不上号；对iOS上似UNIX非UNIX的东西（如后台运行）大肆Google，屡败屡战。当同学们都发表了第一篇小论文时，我甚至没明确自己这个月究竟在干什么，我缺乏太多的基础知识；当同学们周末出去K歌、打牙祭时，我一个人闷在宿舍里对着电脑啃代码；当同学们躺在床上睡懒觉时，我一个人一大早爬起来去实验室加班。一个人是孤独的，但这种孤独换来的是学识的积累，从而转化成内心的笃定，到了最后，因为内在的充实，就不再会感受到外在的孤独了。男人因孤独而优秀，付出一定会有回报——经过一年多的磨合，在2011年3月的一天，以前晦涩难懂的代码突然变得平易近人了，每一句的含义、每一行的关系都变得清楚了，零散的知识点在我的脑袋里被连成了线，整个体系的逻辑慢慢清晰了。于是我快马加鞭，在2011年4月初完成了

毕业设计的程序雏形，并得到了当时对此方向并不抱太大希望的导师的高度评价——“从以前自我感觉良好的优越感变成了肚里有货后的真正自信”，这标志着我对iOS研究的正式入门。明白自己在做什么之后，就能有的放矢，研究效率呈几何倍率提高——在这两年里，我知道了Theos从而“勾搭”上了作者DHowett，向Activator的作者rpetrich讨教过问题，跟TheBigBoss源的管理员Optimo发生过争执，他们是我这一路走来帮我解决实际问题最多的朋友；开发SMSNinja的过程中结识了本书另一作者航哥，在不断深入研究的同时认识了一票做人低调、办事高调的高手，意识到自己并不孤单——我们孤胆，我们并肩。

在本书即将出版的时候回望这5年，我不禁庆幸自己当初的选择是正确的。在iOS方向5年的积淀就足够出一本书，这在Windows方向是不可想象的，而Apple、Google和Microsoft三大巨头的不断发力和市场反馈也直接证明了这个行业一定会是下一个互联网十

年的绝对主角，能够亲眼见证并参与其中，我三生有幸。人生苦短，必须果敢，所以，少年，不要犹豫了，快到碗里来吧！

在受到航哥的邀请写作本书时，我是有些犹豫的。中国人口众多，各行各业竞争都很激烈，自己走了那么多弯路，碰了N鼻子灰才总结提炼出的这些知识，一股脑儿全都交代出去了，会不会有意无意地培养出更多“竞争对手”？这么做是不是把自己的优势拱手相让了？但是纵观越狱iOS的发展历史，从基本的Cydia和CydiaSubstrate，到Theos这样的开发利器，再到Activator这样的神级插件，这些对我影响最为深远的软件无一不是开源的，正是因为这些大牛分享了自己的“优势”，我才能博采众长、逐渐成长；rpetrich牵头的tweakweek和posixninja牵头的openjailbreak也都把宝贵的独门秘籍大白天下，让更多的爱好者参与越狱iOS生态环境的建设。他们是这个圈子里的一线开发者，他们的优势完全没有因为“分享”而减少。我是一

个受益于这个分享链条的人，怎么能在小有所成之后就过河拆桥，断掉我这一环节呢？况且，我是打算在这条路上继续求索的，如果我不停下，我的优势就会一直保持——我的竞争对手只有我自己。相信我们的分享会帮到很多和当年的我一样在门外苦苦徘徊的开发者，集大家智慧创造出的作品能够更好地让科技服务于人，而且我也能结交更多志同道合的朋友，精神生活得到更大满足。这也聊可算作是从长计议吧。

啰啰嗦嗦地说了很多，有些严肃，但这也正是我对待科学技术的态度。本书的内容适合国内绝大多数不满足于折腾App Store的iOS爱好者，通篇干货，童叟无欺，比我的硕士毕业论文要实在得多。更多后续的内容，还请关注本书的官方论坛http://bbs.iosre.com和官方微博@iOS应用逆向工程。让我们一起提升中国iOS开发者在国际上的地位！

在这里，我要感谢母亲对我事业的全力支持，使

我在钻研学术之时能尽可能少地因琐事分心。感谢我的爷爷为我的英语启蒙，良好的英语素养是跟国际同行交流的必要条件；感谢我的导师授我以渔，让我在硕士3年经历脱胎换骨的成长；感谢DHowett、rpetrich和Optimo等大牛对我的无私帮助和尖锐批评，让我在快速成长的同时认识到差距巨大，不敢懈怠；感谢念茜、flyingbird、INT80、jerryxjtu、漏网之鱼、Proteas等前辈对本书的审核与建议，和对我这个初学者的点拨；感谢我的家人和朋友们，你们的支持与鼓励是我前进下去的不竭动力；还要感谢我未来的女朋友，你的缺席让能我一心一意地学习知识，本书稿费啊有我的一半也有你的一半。事业、亲情、友情、爱情是我等凡人的毕生追求，但往往只能求二争三，不可四者兼得，因为这个原因而有意、无意冒犯、伤害过的人，我欠你们一声“对不起”，感谢你们对我的成全。

最后跟大家分享一首我喜爱的诗，啊，人生！多

么奇妙。

未选之路

罗伯特·弗罗斯特[1]

黄色的树林里分出两条路，
可惜我不能同时去涉足，
我在那路口久久伫立，
我向着一条路极目望去，
直到它消失在丛林深处。

但我却选了另外一条路，
它荒草萋萋，十分幽寂；
显得更诱人、更美丽，
虽然在这两条小路上，
都很少留下旅人的足迹。

虽然那天清晨落叶满地，
两条路都未见脚印痕迹。

呵，留下一条路等改日再见！

但我知道路径延绵无尽头，

恐怕我难以再回返。

也许多少年后在某个地方，

我将轻声叹息把往事回顾：

一片树林里分出两条路，

而我选了人迹更少的一条，

从此决定了我一生的道路。

（谨以此书纪念我已仙逝的外祖父刘汉民、祖母吴朝玉）

沙梓社（snakeninny）

[1] Robert Frost（1874—1963），20世纪美国最受欢迎的诗人之一、四度普利策奖得主。本篇为其代表诗作，原题为“The Road Not Taken”。——编辑注

前言

为什么要写这本书

两年前我正式从传统网络设备行业转行进入移动互联网行业，当时正是移动应用开发市场最火爆的时候，创业公司如雨后春笋般的成立，尤其社交类App更是大受追捧，只要有一个不错的构想就可能拿到千万级投资，高价挖人组队的信息更是让人眼花缭乱。那时我已经开发了几个颇具难度的企业应用类App，对于那些轻量级的普通社交App不是太看得上，想着要玩点比较酷的技术，机缘巧合进入了安全管家（北京安管佳科技有限公司），从零开始搭建iOS团队，负责包括越狱方向在内的iOS开发。

其实iOS越狱开发的基础就是iOS逆向工程，那个时候我并没有这方面的经验，面向的是一个完全未知的领域，不过好在有Google，国内国外的信息多少还

是能够搜到点，而且对于iOS开发者，越狱开发和逆向工程并不是一个完全隔离的世界，虽然被分享出来的都是零零散散甚至重复度很高的知识，但是只要投入大量精力，把知识归纳总结，慢慢可以整理出一幅完整的图谱。

然而独自一人学习的过程是孤独的，尤其是遇见困难和问题无人交流，让人一筹莫展。每次一个人扛下所有问题的时候，总是感叹：要是有一个水平不错的交流者该是多么幸福？虽然也可以给Ryan Petrich等一线大牛发邮件请教，但很多在我们看来当时解决不了的难题在这类高手眼中很可能就是个低级问题，不苦心钻研一番根本不好意思去问。这个阶段大概持续了有大半年，直到2012年在微博上遇到本书的另一作者snakeninny，那时他还是一个面临毕业的研究生，整天“不务正业”地研究iOS底层，而且研究得还相当有深度。我曾和他提过：“你看，有多少人都投入到App领域捞钱去了，你咋不去呢？”他说：“弟的

目标远大，要玩就朝着国际一线大牛的目标去！”小兄弟，你够狠！

不过，多数时候我们都是自己在折腾，只是偶尔在网上交流一下问题及解决方法，但往往能碰撞出一些有价值的内容。在一起合写本书之前，我们曾经合作逆向分析过陌陌，做了一个插件用于在陌陌iOS版上把美女的位置标注在地图上。当然我们都是善意的开发者，主动将这个漏洞告诉了陌陌，他们很快就修复了。这次，我们再次合作，将iOS逆向工程方向的知识整理出版，呈现给各位读者。

在接触越狱开发、逆向工程的这些年，个人感觉最大的收获就是看待App时，完全以一种庖丁解牛的眼光去审视：App如何构成、性能如何，可以直接反映出开发团队水平高低。这些经验知识不仅可用于越狱开发，也适用于传统的App开发，至于带来的影响，有正有负吧！我们不能因为苹果不提倡越狱就否

定这个领域的存在，盲目地相信本书曝光的安全问题不存在不过是掩耳盗铃罢了。

有经验的开发者都明白，知识掌握得越深，越会接触到底层技术。比如sandbox保护机制具体体现在哪些方面？runtime只用来研究理论知识是不是有点大材小用了？

在Android领域，底层技术已经被扩散开，而在iOS领域，这个方向展现出来的内容还只是冰山一角。虽然国外也有几本iOS安全方向的书籍，比如《Hacking and Securing iOS Applications》、《iOS Hacker’s Handbook》，但是内容太难，绝大多数人根本读不懂，即使我们这些有一定经验的开发者，读这些书也非常吃力，效果不好。

阳春白雪不为我们这些喜欢实践的技术宅所好，那么来点下里巴人的，不必遮遮掩掩，直接全面展开这些知识岂不是更痛快？于是就有了我们这本书，书

中的内容以概念、工具、理论、实战的形式全面、系统地展开知识点，由浅入深，图文并茂，带着读者一步步地探索App的内在。我们不会像一些技术博客那样貌似很高深地独立分析某一片段的代码，也不纠结“茴”字有几种写法，而是尽我们所能将一个完整的知识体系呈现给读者，提供一整套iOS应用逆向工程的方法论，相信读者一定会有所收获。

近些年，国内投入在越狱iOS这个方向的人越来越多，但都比较低调，他们开发出的越狱工具、App助手、Cydia插件影响着整个iOS的发展。他们积累的技术非我们这些散兵游勇所能及，但我们更愿意分享这些知识，希望能够抛砖引玉。

读者对象

本书主要面向以下读者：

· iOS狂热爱好者。

· 中高级iOS开发人员。他们在掌握了App开发之后对iOS有更深的渴求。

· 架构师。在逆向App的整个过程中，架构师能学习那些优秀App的架构设计，以这种方式博采众长，提高自己的架构设计能力。

· 在别的系统上从事逆向工程，想要转向iOS逆向工程的工程师。

如何阅读本书

本书将分为四大部分，分别是概念、工具、理论和实战。前三部分介绍iOS逆向工程这个领域的背景、知识体系，以及相应的工具集、理论知识；第四部分则以4个具体案例将前面的知识以实战的方式展开，让读者可以实践验证前面学到的知识，加深对iOS逆向工程的理解。

如果读者不具备iOS逆向工程经验，建议还是从

头开始按顺序阅读，而不要直接跨越到第四部分去模拟实战。虽然实战的成果很炫，但知其然而不知其所以然也没意思，对不对？

勘误和支持

由于作者的水平有限，编写的时间也很仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正，欢迎访问本书的官方论坛http://bbs.iosre.com，全球的iOS逆向工程师都在这里聚集，你的问题应该会得到满意的解答。如果你有更多的宝贵意见，也欢迎你通过微博@iOS应用逆向工程或官方论坛与我们联系，我们很期待能够听到你们的真挚反馈。

致谢

首先要感谢evad3rs、盘古、太极、saurik等顶级团队与高手，他们奠定了越狱iOS的基石；还要感谢

DHowett，是他提供了Theos这个强大的开发工具使我得以迈进iOS逆向工程的大门。

感谢安全管家，为我进入iOS逆向工程领域提供了一个充分发挥的环境，虽然我早已离开，但希望它发展得更好。

感谢微博上每一位热心的朋友——唐巧_boy、卢明华、你在瓦西里、isdada、Jagie、onevcat、戴铭、费西FISH、xuzhanji、Life无法Debug、移动开发小冉、HorseLuke、网络蝎子、hongjiang_wang、月之舞狼、StayNStay、bluesea哈哈哈、郑州IOS、青年土豆的烦恼、木土吉吉，以及这个仓促写就的名单之外的更多朋友，感谢你们对我的支持和鼓励。特地感谢唐巧_boy的引荐，他的热心帮助促成了本书的出版。

感谢机械工业出版社华章公司的编辑杨绣国老师，感谢她的魄力和远见，在这三个月的时间里始终支持我的写作，她的指点和帮助引导我们顺利完成全

部书稿。

谨以此书献给我最亲爱的家人，以及众多热爱iOS开发的朋友们。

吴航（hangcom2010）

第一部分　概念篇

· 第1章　iOS逆向工程简介

· 第2章　越狱iOS平台简介

软件逆向工程，指的是通过分析一个程序或系统的功能、结构或行为，将它的技术实现或设计细节推导出来的过程。当我们对一个软件的功能很感兴趣，却又拿不到它的源代码时，往往可以通过逆向工程的方式对它进行分析。

对于iOS开发者来说，运行在iOS上的各种软件是我们知晓的最复杂且超奇妙的虚拟物品之一，它们精巧而细致，新颖且创意十足。作为开发者，在看到一些精美的App，惊叹于它的实现之外，也一定会好奇：在优雅的外表下，它们采用了何种技术？我们能否从中学到些什么？

第1章　iOS逆向工程简介

虽然可口可乐的配方是高度机密，但还是有些公司可以调制出跟可乐几乎没有差别的味道。虽然我们拿不到别人App的源码和文档，但仍可以通过逆向工程来一窥究竟。

1.1 iOS逆向工程的要求

iOS逆向工程指的是在软件层面上进行逆向分析的一个过程。读者如果想要具备较强的iOS逆向工程能力，最好能非常熟悉iOS设备的硬件构成、iOS系统的运行原理，还要具备丰富的iOS开发经验。如果你拿到任意一个App之后能够大致推断出它的项目规模和使用的技术，比如它的MVC（Model-View-Controller，请Google“iOS MVC”）模型是怎么建立的，引用了哪些framework和经典的开源代码，说明你的iOS逆向工程能力已经不容小觑了。

这要求高吗？好像确实有点高！不过，这些条件都是充分非必要的。如果你目前还不具备这些充分条件，那么一定要满足两个必要条件：强烈的好奇心和锲而不舍的精神。因为在iOS逆向工程中，好奇心会驱动你去研究经典的App，而在研究的过程中一定会遇到一系列的困难和障碍，但你又不可能对任何问题

都胸有成竹，所以这时就需要有锲而不舍的精神来支撑你克服一个又一个困难。请相信，在投入大量精力去编写代码、调试程序、分析逻辑之后，你会在不断的试验和错误中感受到逆向工程的艺术之美，你的个人能力也会得到质的提升。

1.2 iOS应用逆向工程的作用

打个比喻，iOS逆向工程就像一杆长矛，专门刺破App看似安全的防护盾。有趣的是，很多制作App的公司还没有意识到这样一杆长矛的存在，固步自封地以为自己的盾坚不可摧。

对于微信和WhatsApp之类的IM应用，交流的信息是它们的核心；对于银行、支付、电商类的软件，交易数据和客户信息是它们的核心。所有的核心数据都是需要重点保护的，于是，开发人员通过反调试、数据加密、代码混淆等各种手段重重保护自己App，为的就是增加逆向工程的难度，避免类似的安全问题影响用户体验。

可是目前App防护所用到的技术跟iOS逆向工程所使用的技术根本就不是同一个维度的。一般的App防护，感觉就像是一个城堡，将App的MVC布置在城堡内部，外围圈上厚厚的城墙，看上去易守难攻，就

像图1-1所示的这样。

图1-1　防御良好的城堡（图片来自刺客信条）

但是当我们站到高处，在天空中鸟瞰这个App所在的城堡，它的内部结构就不再是秘密，如图1-2所示。

图1-2　鸟瞰城堡（图片来自刺客信条）

这时，所有的Objective-C函数定义、所有的property、所有的导出函数、所有的全局变量、所有的逻辑完全暴露在我们面前，城墙的防护意义荡然无存。处在这个维度，城墙已经不再是阻碍，我们更应该关注的是如何从偌大的城堡里面找到想要找的那一个人。

此时，基于iOS逆向工程技术，可以在不破坏城墙的前提下，选择任意高维度地点进入低维度城堡，巧取而不强夺，通过监视甚至改变App的运行逻辑，

从而达到获取核心信息，了解软件设计原理等战术目的。

说得似乎很玄乎，可事实就是如此。就笔者数年来对App和iOS系统本身进行逆向工程的经历和成果来看，iOS应用逆向工程可以“透视”绝大多数App，它们的设计理念与实现细节在逆向工程中暴露无遗。

以上比喻只是iOS逆向工程的一隅之见，但也形象地说明了iOS逆向工程的强大威力。概括起来，iOS逆向工程主要有两个作用：

· 分析目标程序，拿到关键信息，可以归类于安全相关的逆向工程；

· 借鉴他人的程序功能来开发自己的软件，可以归类于开发相关的逆向工程。

1.2.1 安全相关的iOS逆向工程

安全相关的IT行业一般会大量运用逆向工程技术。比如：通过逆向一个金融类App，来评定安全等级；通过逆向iOS病毒，来找到查杀的方法；通过逆向iOS系统电话、短信功能，来构建一个手机防火墙，等等。

1.评定安全等级

iOS中那些具有交易功能的App一般会先加密数据，然后将加密过的数据存储在本地或通过网络传输。如果安全意识不够强，就完全有可能将敏感信息（如银行账号和密码）直接用明文保存或传输，安全隐患极大。

假如一家有名望的公司考虑推出一款App，为了让App的质量能够对得起公司的声誉和用户的信任，该公司聘请一家安全机构来评估这个App的安全性。绝大多数情况下，安全机构无法拿到App的源代码，不能通过代码审核的方式正向分析App的安全性，因

此，他们只有利用iOS逆向工程技术，尝试“攻击”这个App，然后依据结果评定其安全等级。

2.逆向恶意软件

iOS是智能移动终端操作系统，它同计算机操作系统没有本质区别。从第一代开始，它就已具备了上网功能，而互联网正是恶意软件传播的最好媒介。2009年暴露的Ikee病毒是iOS上公开的第一款蠕虫病毒，它会感染那些已经越狱并且安装了ssh服务，但是没有更改ssh默认密码“alpine”的iOS，将它们的锁屏背景图改成一个英国歌手的照片。2014年年底出现的WireLurker病毒会窃取用户隐私信息，并且可以通过PC和Mac传播，给iOS用户带来了比较严重的危害。

对于恶意软件的开发者来说，他们通过逆向工程定位系统和软件漏洞，利用漏洞渗透进目标主机，获取数据，为所欲为。

对于杀毒软件的开发者来说，他们通过逆向工程剖析病毒样本，观察病毒行为，尝试查杀被感染主机上的病毒，并总结出可以防范病毒的方法。

3.检查软件后门

开源软件的一大优势是其具有较好的安全性。成千上万的程序员浏览并修改开源软件的代码，代码中几乎不可能存在任何后门，软件的安全问题往往在大白于天下之前就能及时得到解决。而对于闭源软件，逆向工程是检查其是否留有后门的主要方法之一。比如我们常会在越狱iOS中通过AppStore以外的渠道安装各种软件，这些软件未经苹果官方审核，存在一定安全隐患；还有的App会故意留有后门，伺机损害用户的利益。对这一类行为进行检测的过程中通常少不了逆向工程的身影。

4.去除软件使用限制

iOS开发者通过AppStore或Cydia等渠道出售自己的App，作为他们最主要的经济来源之一。在软件的世界里，收费与盗版永远是共存的，不少开发者也在自己的App里加入了防止盗版的功能。但这场矛与盾的战争永远不会停止，再好的防御也会有被攻破的一天，盗版软件的层出不穷把防止盗版变成了一项不可能完成的任务。比如Cydia上最知名的“共享”源xsellize能够在几乎所有收费软件发布后的1天时间内对其完成破解，是业界的一大毒瘤。

1.2.2 开发相关的iOS逆向工程

对于iOS开发者来说，逆向工程是最为实用的技术之一。例如，工程师可以逆向系统API，在自己的App里使用一些文档中没有提及的私有功能，还可以逆向一些经典软件，学习借鉴它们的技术和设计。

1.逆向系统API

工程师编写的软件之所以能够运行在操作系统中，提供各种各样的功能，是因为操作系统本身已经内嵌了这些功能，软件只是对其进行重新组合罢了。众所周知，能在App Store上架的App的功能十分有限，在苹果公司严格的审核制度下，绝大多数App的实现都源于公开的开发文档，而不能使用诸如发短信、打电话等文档中不涉及的功能。如果你的软件面向Cydia，那么不采用非公开功能将会导致软件丧失极大的竞争力。如果你的软件想拥有文档里没有提及的非公开功能，最有效的途径就是逆向iOS系统API，还原系统实现相应功能的代码，并应用到自己的软件中。

2.借鉴别的软件

逆向工程最受欢迎的应用场合就是“借鉴”他人的软件功能。对于App Store上的大多数App来说，其技术实现并不复杂，巧妙的创意和良好的运营才是其成

功的关键。如果只是单纯借鉴其功能，那么采用逆向工程来还原代码，费时费力，性价比低，不如从头开发一个功能类似的软件省时省力，性价比高。但是，当我们不知道App中的某个功能是如何实现的时候，逆向工程就能起到关键性的作用。这种情况在大量使用私有函数的Cydia软件中尤其常见，比如2013年3月面世的，号称iOS上第一款通话录音软件的Audio Recorder，它是闭源软件，但足够有趣，此时使用iOS逆向工程技术就能够对它了解一二。

有些老牌软件的架构设计合理，代码工整规范，实现得非常优雅。我们没有他们那样深厚的技术功底和人才储备，想要借鉴他们使用的高级技术，却又求学无门。在这种情况下，逆向工程就是解决问题的金钥匙。通过逆向那些软件，可以从App中把它们的设计思路抽象出来为我所用，从而提高自己App的精致程度。比如，WhatsApp的稳定性、健壮性出类拔萃，如果我们自己要编写一个IM类App，通过逆向工

程技术学习WhatsApp的整体架构与设计思路将是非常有益的。

1.3 iOS应用逆向工程的过程

要逆向一个App时，应该怎么思考？应该从何入手？这本书的初衷就是引导初学者走进iOS逆向工程的大门，培养读者从逆向的角度思考问题。

一般来说，软件逆向工程可以看作系统分析和代码分析两个阶段的有机结合。在系统分析阶段，要从整体上观察目标程序的行为特征、文件的组织架构，从而找到我们感兴趣的地方，进入代码分析阶段后，则要把软件的核心代码还原出来，最终达到我们的目的。

1.3.1 系统分析

在系统分析阶段，应在不同的条件下运行目标程序，在程序中进行各种各样的操作，观察程序的行为特征，同时寻找我们感兴趣的功能点。比如选择哪个选项会弹框，按下哪个按钮会发声，输入什么内容屏

幕会有什么显示，等等。还可以浏览文件系统，观察程序显示的图片、程序的配置文件存放的位置，数据库文件中存放了哪些信息，有没有加密等特征。

以新浪微博App为例，我们在查看它的Documents目录时，会看到如下一些数据库文件：

```
-rw-r--r-- 1 mobile mobile  210944 Oct 26 11:34
db_46100_1001482703473.dat
-rw-r--r-- 1 mobile mobile  106496 Nov 16 15:31
db_46500_1001607406324.dat
-rw-r--r-- 1 mobile mobile  630784 Nov 28 00:43
db_46500_3414827754.dat
-rw-r--r-- 1 mobile mobile 6078464 Dec  6 12:09
db_46600_1172536511.dat
……
```

用SQLite工具打开它们，可以看到一些微博关注信息，如图1-3所示。

这样的信息给逆向工程提供了很多线索：数据库文件名、微博用户的ID，用户信息对应的URL等，这些都可以作为逆向工程的切入点。寻找和整理这些线索，从中抽丝剥茧找到我们感兴趣的东西，往往是

iOS逆向工程的第一步。

1807621622	天生歌姬A-Lin	http://tp3.sinaimg.cn/1807621622/50/5700356795/0
1497487043	焦波和俺爹俺娘	http://tp4.sinaimg.cn/1497487043/50/1297238551/1
2835121504	Angela侯湘婷	http://tp1.sinaimg.cn/2835121504/50/5664550946/0
1744390777	林凡Freya	http://tp2.sinaimg.cn/1744390777/50/40053380567/0
2875568950	EDC尤原庆	http://tp3.sinaimg.cn/2875568950/50/40001989049/1
1565668374	财上海	http://tp3.sinaimg.cn/1565668374/50/5703348848/1
3962782795	旗绮贞cheerego	http://tp4.sinaimg.cn/3962782795/50/40043044497/0
1283498527	许哲佩PeggyHsu	http://tp4.sinaimg.cn/1283498527/50/40054968787/0
1198922365	曹方Icy	http://tp2.sinaimg.cn/1198922365/50/5635147308/0
2270268414	Dawen王大文	http://tp3.sinaimg.cn/2270268414/50/5705504906/1
1787113000	Mrdadado黄玠	http://tp1.sinaimg.cn/1787113000/50/5602434900/1
1751505334	魏如萱waa	http://tp3.sinaimg.cn/1751505334/50/5711190523/0

图1-3　新浪微博数据库信息

1.3.2　代码分析

完成系统分析之后，就该对App的二进制文件进行代码分析了。

通过逆向工程，可以推导出这个App的设计思路、内部算法和实现细节，但这是一个非常复杂的过程，可以说是一种解构再重组的艺术。想让自己的逆向工程水平达到艺术的高度，需要对软件开发、硬件原理和iOS系统有透彻的理解。一点一点地分析程序

的底层指令绝非易事，也不是一本书能够完全阐述清楚的。

本书的目标仅仅是向初学者讲述iOS逆向工程入门时所用到的工具和一般思路，但技术是不断发展的，书中的内容不可能覆盖所有的知识点。出于这个考虑，笔者搭建了一个iOS逆向工程论坛http://bbs.iosre.com，供大家讨论和交流最新的技术。

1.4 iOS应用逆向工程的工具

了解了一些iOS逆向工程的理论，就要使用各种工具实践这些理论了。相对于正向开发，逆向工程使用的工具并不那么“智能”，很多工作需要我们手工完成。但是对工具的熟练使用能够极大地提高逆向工程的效率。iOS逆向工程的工具可以分为四大类：监测工具、反汇编工具（disassembler）、调试工具（debugger），以及开发工具。

1.4.1 监测工具

在iOS逆向工程中，起到嗅探、监测、记录目标程序行为的工具统称为监测工具。这些工具通常可以记录并显示目标程序的某些操作，如UI变化、网络活动、文件访问等。iOS逆向常用的监测工具有Reveal、snoop-it、introspy等。

图1-4所呈现的是一款监测工具——Reveal，它可

以用来实时监测目标App的UI布局变化。

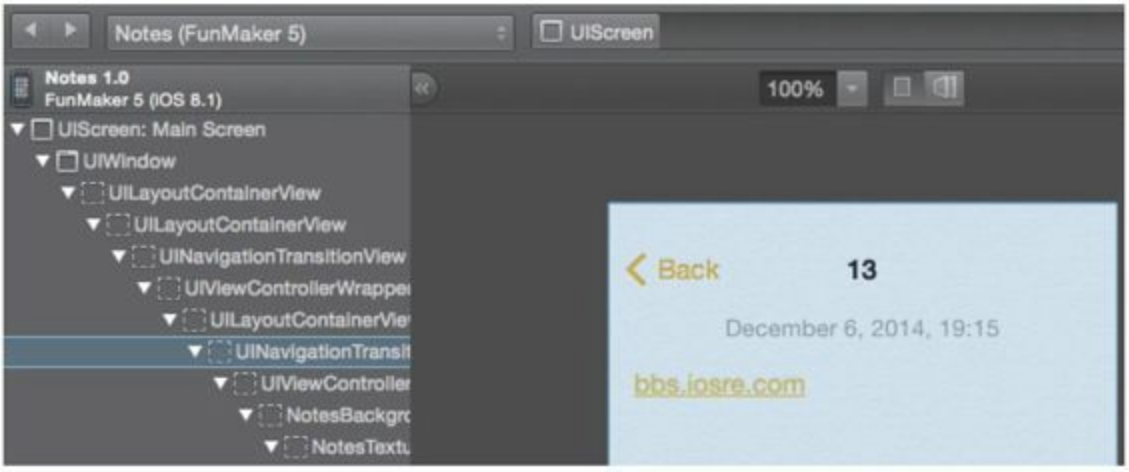


图1-4　Reveal

Reveal能够辅助定位App中我们感兴趣的部分，让我们能够迅速从UI层面切入代码层面。

1.4.2　反汇编工具

从UI层面切入代码层面后，就要用到反汇编工具来梳理代码了。反汇编工具把二进制文件作为输入，经过处理后输出这个文件的汇编代码；在iOS逆向工程中，常用的反汇编工具主要是IDA和Hopper。

作为老牌反汇编工具，IDA是逆向工程中最常用

的利器之一。它支持Windows、Linux、OSX平台和多种处理器架构，如图1-5所示。

Hopper是一款近年面世的反汇编工具，它主要针对的是苹果系操作系统，如图1-6所示。

把二进制文件反汇编之后，就要阅读生成的汇编代码了。这是iOS逆向工程中最具挑战，也是最有意思的部分，具体会在第6章详细讲述。本书主要以IDA作为反汇编工具，但我们会在http://bbs.iosre.com交流Hopper的使用心得。

```
sub_2974
PUSH            {R7,LR}
MOVW            R0, #(:lower16:(selRef_processInfo - 0x298C))
MOV             R7, SP
MOVT.W          R0, #(:upper16:(selRef_processInfo - 0x298C))
MOV             R2, #(classRef_NSProcessInfo - 0x298E) ; classRef_NSProcessInfo
ADD             R0, PC ; selRef_processInfo
ADD             R2, PC ; classRef_NSProcessInfo
LDR             R1, [R0] ; "processInfo"
LDR             R0, [R2] ; _OBJC_CLASS_$_NSProcessInfo
BLX             _objc_msgSend
MOV             R1, #(selRef_processName - 0x29A0) ; selRef_processName
ADD             R1, PC ; selRef_processName
LDR             R1, [R1] ; "processName"
BLX             _objc_msgSend
MOV             R1, #(selRef_isEqualToString_ - 0x29B4) ; selRef_isEqualToString_
MOVW            R2, #(:lower16:(cfstr_Springboard - 0x29BA)) ; "SpringBoard"
ADD             R1, PC ; selRef_isEqualToString_
MOVT.W          R2, #(:upper16:(cfstr_Springboard - 0x29BA)) ; "SpringBoard"
ADD             R2, PC  ; "SpringBoard"
LDR             R1, [R1] ; "isEqualToString:"
BLX             _objc_msgSend
TST.W           R0, #0xFF
BEQ             loc_29CE
```

图1-5　IDA

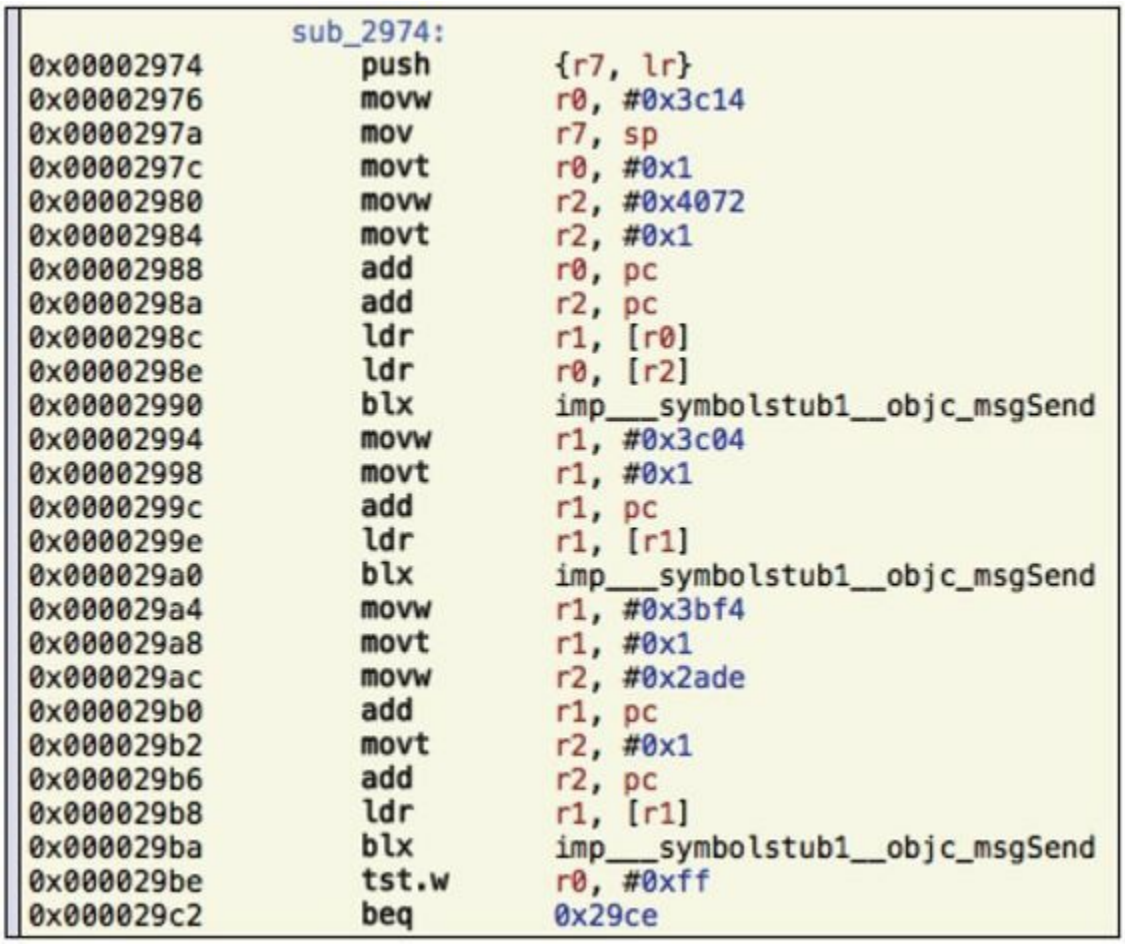


图1-6　Hopper

1.4.3　调试工具

iOS开发者对调试工具应该不陌生，在App开发中，少不了在Xcode中调试代码。我们可以在某一行代码上设置断点，使进程能够停止在那一行代码上，并实时显示进程当前的状态。在iOS逆向工程中，用

到的调试工具主要是LLDB。图1-7是使用LLDB进行调试的示例。

```
snakeninnys-MacBook:~ snakeninny$ lldb
(lldb) attach Finder
Process 303 stopped
Executable module set to "/System/Library/CoreServices/
Finder.app/Contents/MacOS/Finder".
Architecture set to: x86_64-apple-macosx.
(lldb) c
Process 303 resuming
```

图1-7　LLDB

1.4.4　开发工具

从UI层面切入代码层面，用反汇编工具和调试工具分析过二进制文件后，就可以整理分析结果，用开发工具写程序了。对于App开发者来说，Xcode是最常用的开发工具。但是我们一旦将战场从AppStore转移到越狱iOS，开发工具的种类就得到了扩充，不但有基于Xcode的iOSOpenDev，还有偏命令行的Theos。从个人体验来说，Theos是让笔者最为兴奋的开发工具，在知道Theos之前，笔者感觉自己一直都

被限制在AppStore中，直到掌握了Theos的用法，才突破了AppStore，完整地认识了整个iOS系统。本书主要以Theos作为开发工具，关于iOSOpenDev的问题，可以到http://bbs.iosre.com交流讨论。

1.5 小结

本章科普了iOS应用逆向工程的相关概念，旨在让读者对iOS逆向工程有一个概念上的大体了解。详细的技术内容和案例会在后面的章节中逐一讲解，敬请期待。

第2章　越狱iOS平台简介

相较于iOS应用的高层表象，人们对其底层实现更感兴趣，这也是大家进行逆向工程的源动力。但是我们也都知道，未越狱的iOS是个封闭的黑盒子，直到evad3rs、盘古、太极等团队把iOS越狱之后，这个黑盒子才被打开，神秘的iOS得以完整地展现在我们面前。

2.1 iOS系统结构

对于未越狱的iOS，苹果官方开放给第三方直接访问iOS文件系统的接口非常有限，开发者只需要遵循规定，参考文档即可完成工作。因此，纯粹的App Store开发者可能对iOS系统结构一无所知。

因为权限极低，来自App Store的普通App（以下简称StoreApp）不能访问自身目录以外的绝大多数文件。而iOS一旦越狱，来自Cydia的App就可以拥有比StoreApp更高的权限，从而访问全系统文件；来自Cydia的iFile即是iOS上一个老牌的第三方文件管理App，如图2-1所示。

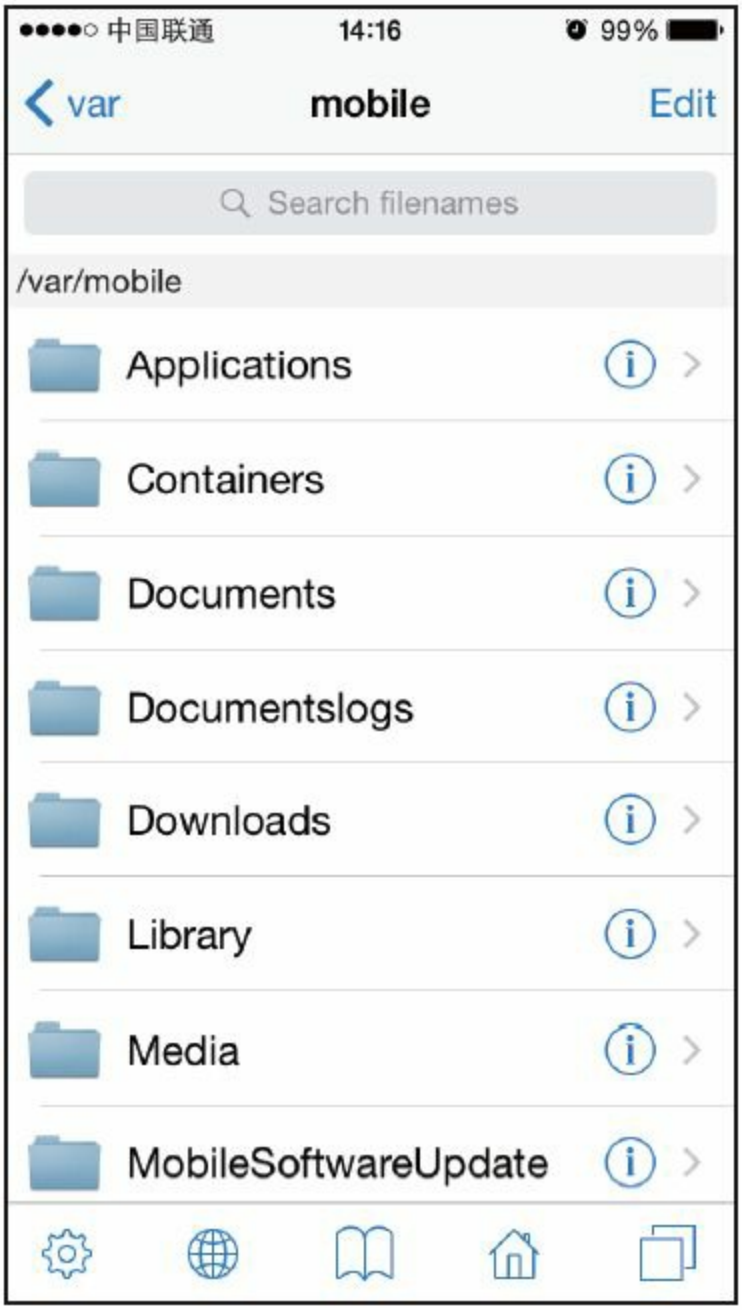


图2-1　iFile

还可以在AFC2服务的帮助下，通过iFunBox等PC端软件访问iOS全系统文件，如图2-2所示。

因为要逆向的对象来自于iOS，所以能够访问iOS全系统文件是开展iOS逆向工程的首要前提。

图2-2　iFunBox

2.1.1　iOS目录结构简介

iOS是由OSX演化而来的，而OSX则是基于UNIX操作系统的。这三者虽然有很大区别，但它们血脉相连。从Filesystem Hierarchy Standard和hier(7)中，可以一窥iOS目录结构的设计标准。

Filesystem Hierarchy Standard（以下简称FHS）

为类UNIX操作系统的文件目录结构制定了一套标准，它的初衷之一是让用户预知文件或目录的存放位置。OSX在此基础上形成了自己的hier(7)框架。类UNIX操作系统的常见目录结构如下所示。

· /：根目录，以斜杠表示，其他所有文件和目录在根目录下展开。

· /bin：“binary”的简写，存放提供用户级基础功能的二进制文件，如ls、ps等。

· /boot：存放能使系统成功启动的所有文件。iOS中此目录为空。

· /dev：“device”的简写，存放BSD设备文件。每个文件代表系统的一个块设备或字符设备，一般来说，“块设备”以块为单位传输数据，如硬盘；而“字符设备”以字符为单位传输数据，如调制解调器。

· /sbin：“system binaries”的简写，存放提供系统级基础功能的二进制文件，如netstat、reboot等。

· /etc：“Et Cetera”的简写，存放系统脚本及配置文件，如passwd、hosts等。在iOS中，/etc是一个符号链接，实际指向/private/etc。

· /lib：存放系统库文件、内核模块及设备驱动等。iOS中此目录为空。

· /mnt：“mount”的简写，存放临时的文件系统挂载点。iOS中此目录为空。

· /private：存放两个目录，分别是/private/etc和/private/var。

· /tmp：临时目录。在iOS中，/tmp是一个符号链接，实际指向/private/var/tmp。

· /usr：包含了大多数用户工具和程序。/usr/bin

包含那些/bin和/sbin中未出现的基础功能，如nm、killall等；/usr/include包含所有的标准C头文件；/usr/lib存放库文件。

· /var：“variable”的简写，存放一些经常更改的文件，比如日志、用户数据、临时文件等。其中/var/mobile和/var/root分别存放了mobile用户和root用户的文件，是重点关注的目录。

上述目录中的内容多用于系统底层，逆向难度较大，作为初学者，暂时不用在其中投入太多精力。建议初学者从学和练的角度出发，循序渐进，由易到难，先从熟悉的内容开刀，这样效率更高。

作为iOS开发者，日常操作所对应的功能模块大多来自iOS的独有目录，如下所示。

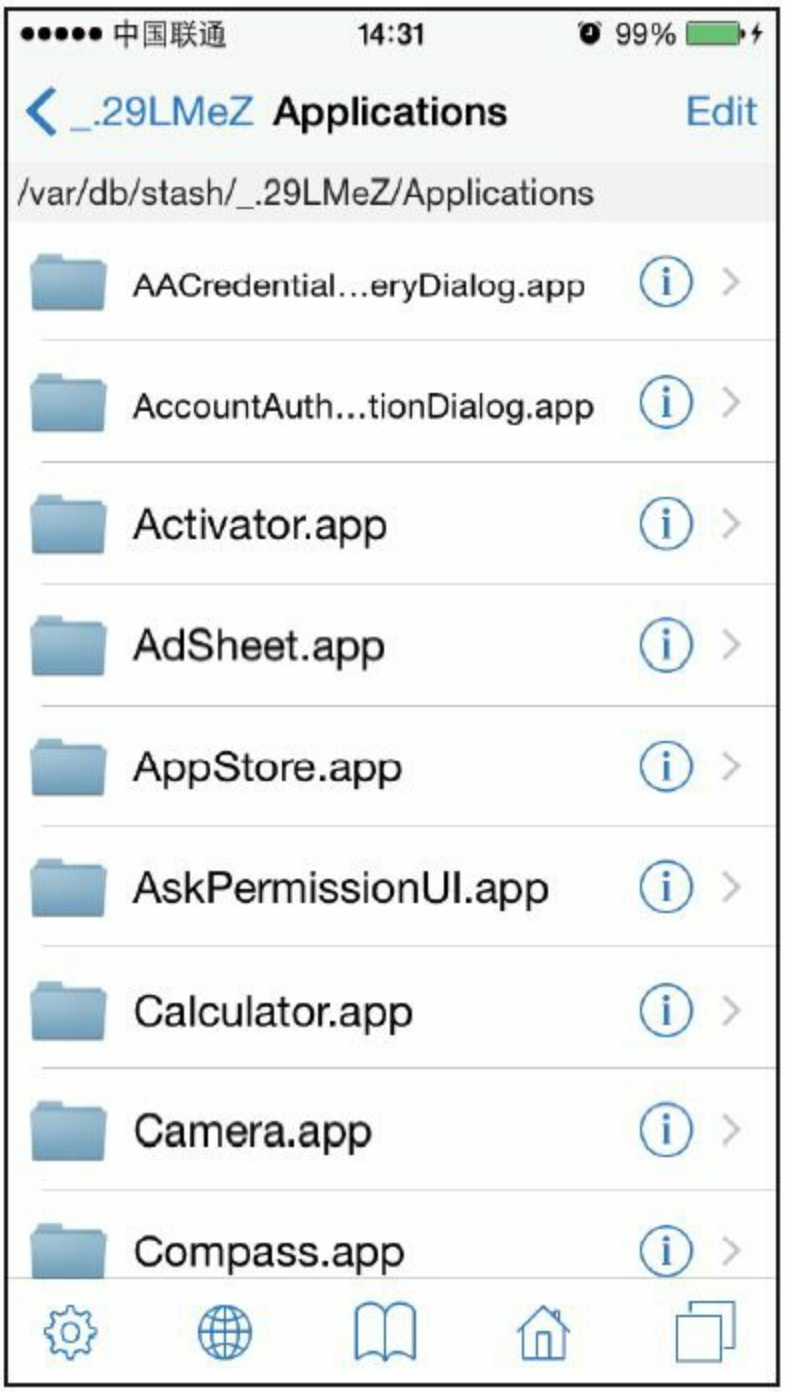


图2-3　/Applications

· /Applications：存放所有的系统App和来自于Cydia的App，不包括StoreApp，如图2-3所示。

· /Developer：如果一台设备连接Xcode后被指定为调试用机（如图2-4所示），Xcode就会在iOS中生成这个目录，其中会含有一些调试需要的工具和数据，它的目录结构如图2-5所示。

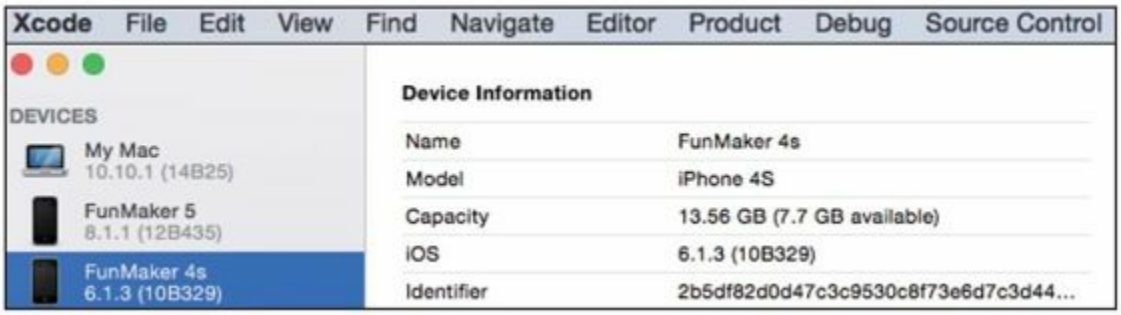


图2-4　指定设备为调试用机

· /Library：存放一些提供系统支持的数据，其结构如图2-6所示。其中/Library/MobileSubstrate下存放了所有基于CydiaSubstrate（原名MobileSubstrate）的插件。

· /System/Library：iOS文件系统中最重要的目录之一，存放大量系统组件，其目录结构如图2-7所示。

对于该目录，在逆向工程的初学阶段，需要重点关注的有：

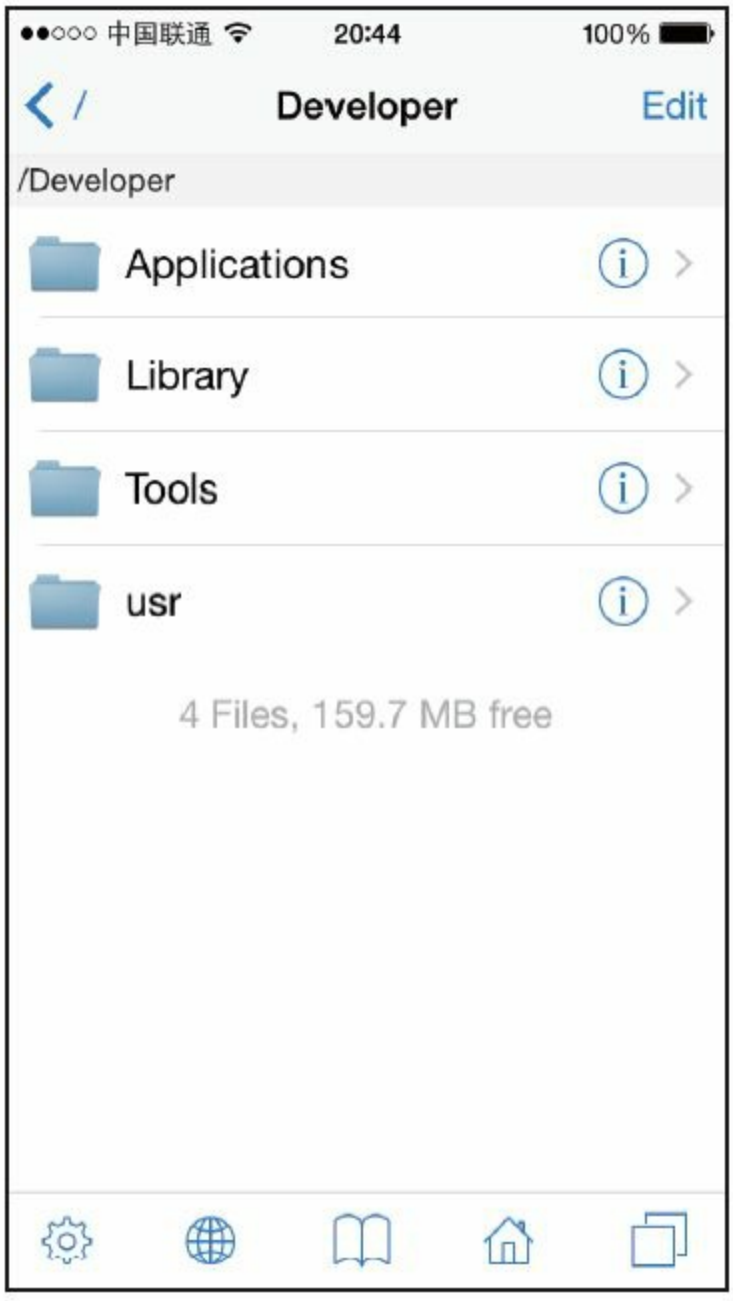


图2-5　/Developer

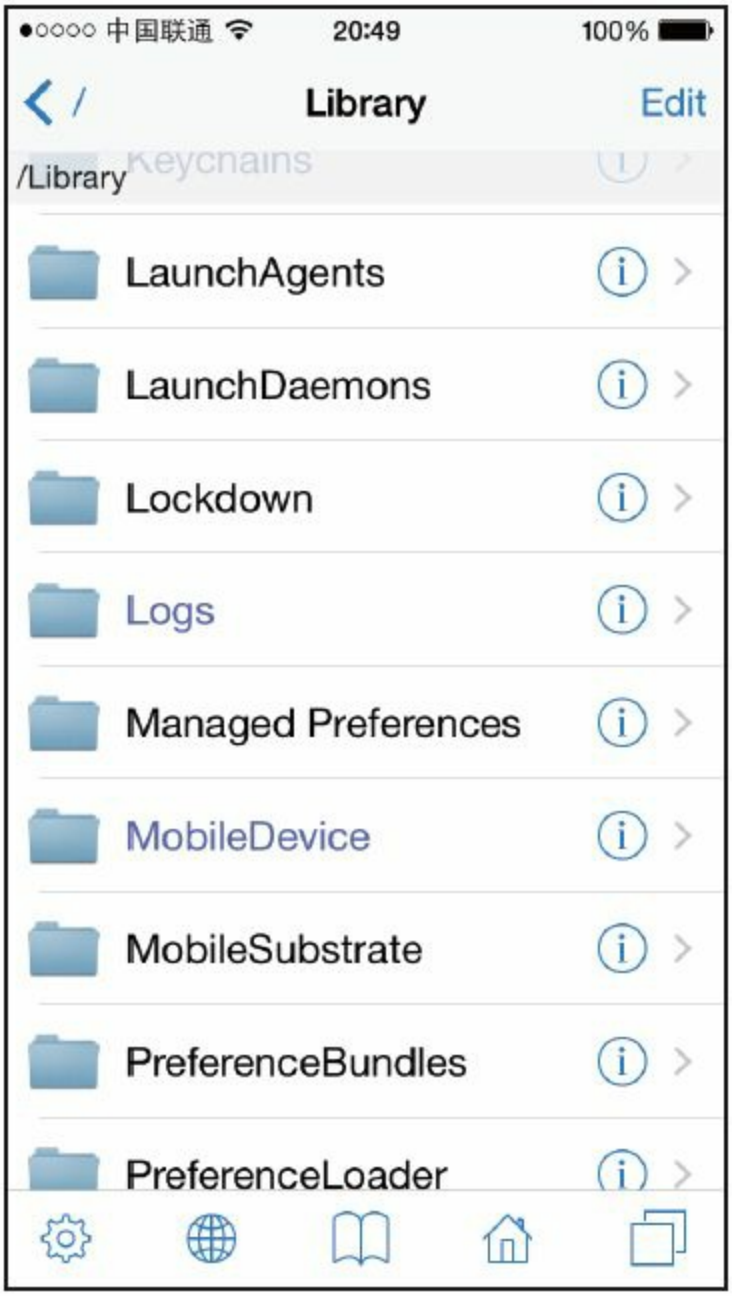

●○○○○ 中国联通
20:49
100%
/
Library
Edit
/Library
Keychains
LaunchAgents
LaunchDaemons
Lockdown
Logs
Managed Preferences
MobileDevice
MobileSubstrate
PreferenceBundles
PreferenceLoader

图2-6　/Library

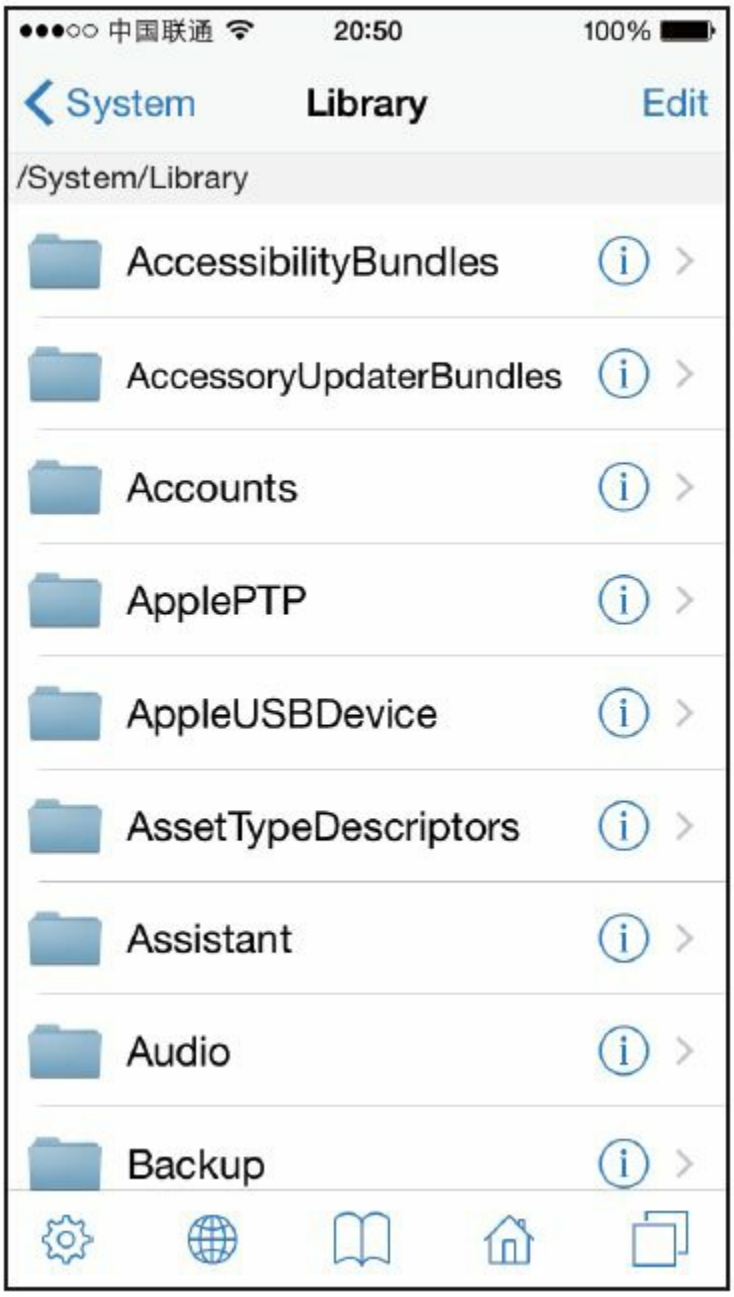


图2-7　/System/Library

· /System/Library/Frameworks

和/System/Library/PrivateFrameworks：存放iOS中的各种framework，其中出现在SDK文档里的只是冰山一

角，还有数不清的未公开功能等待我们去挖掘。

· /System/Library/CoreServices里的SpringBoard.app：iOS桌面管理器（类似于Windows里的explorer），是用户与系统交流的最重要中介。

“/System”目录下的玄机远不止上面提到的3个目录这么简单，更多的进阶内容，会在http://bbs.iosre.com持续讨论。

· /User：用户目录，实际指向/var/mobile，其目录结构如图2-8所示。

这个目录里存放大量用户数据，比如：

· /var/mobile/Media/DCIM下存放照片；

· /var/mobile/Media/Recordings下存放录音文件；

· /var/mobile/Library/SMS下存放短信数据

库；

· /var/mobile/Library/Mail下存放邮件数据。

另外一个非常重要的子目录是/var/mobile/Containers，存放StoreApp。值得注意的是，App的可执行文件在bundle与App中的数据目录被分别存放在/var/mobile/Containers/Bundle和/var/mobile/Containers/Data这两个不同目录下，如图2-9所示。

对iOS目录结构的初步了解有助于在发现感兴趣的功能后，想要定位其对应的文件时有规律可循。上面的介绍只是整个iOS目录结构的九牛一毛，更详细的讨论，尽在http://bbs.iosre.com。

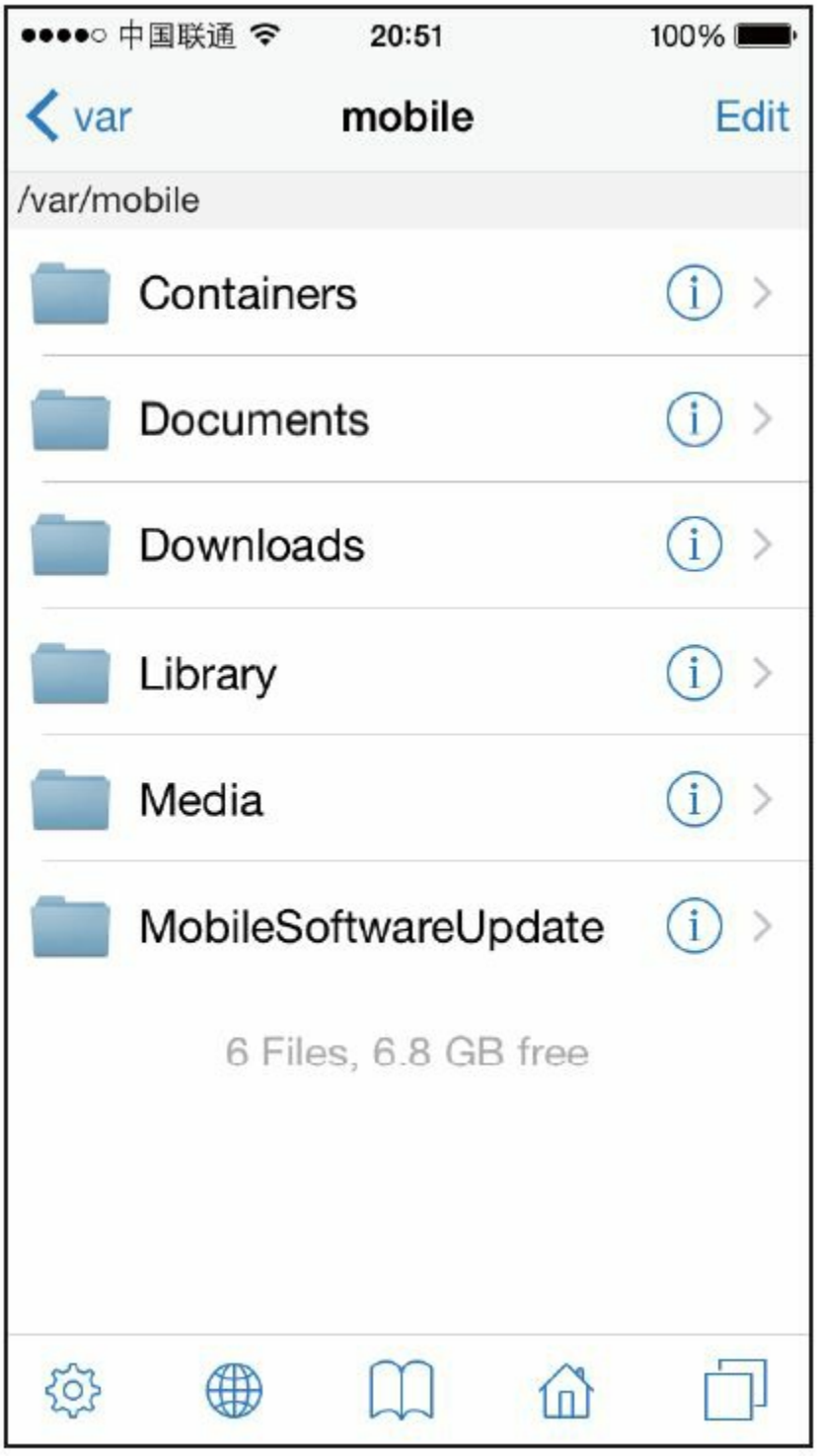

●●●●○ 中国联通
20:51
100%
var
mobile
Edit
/var/mobile
Containers
Documents
Downloads
Library
Media
MobileSoftwareUpdate
6 Files, 6.8 GB free

图2-8　/User

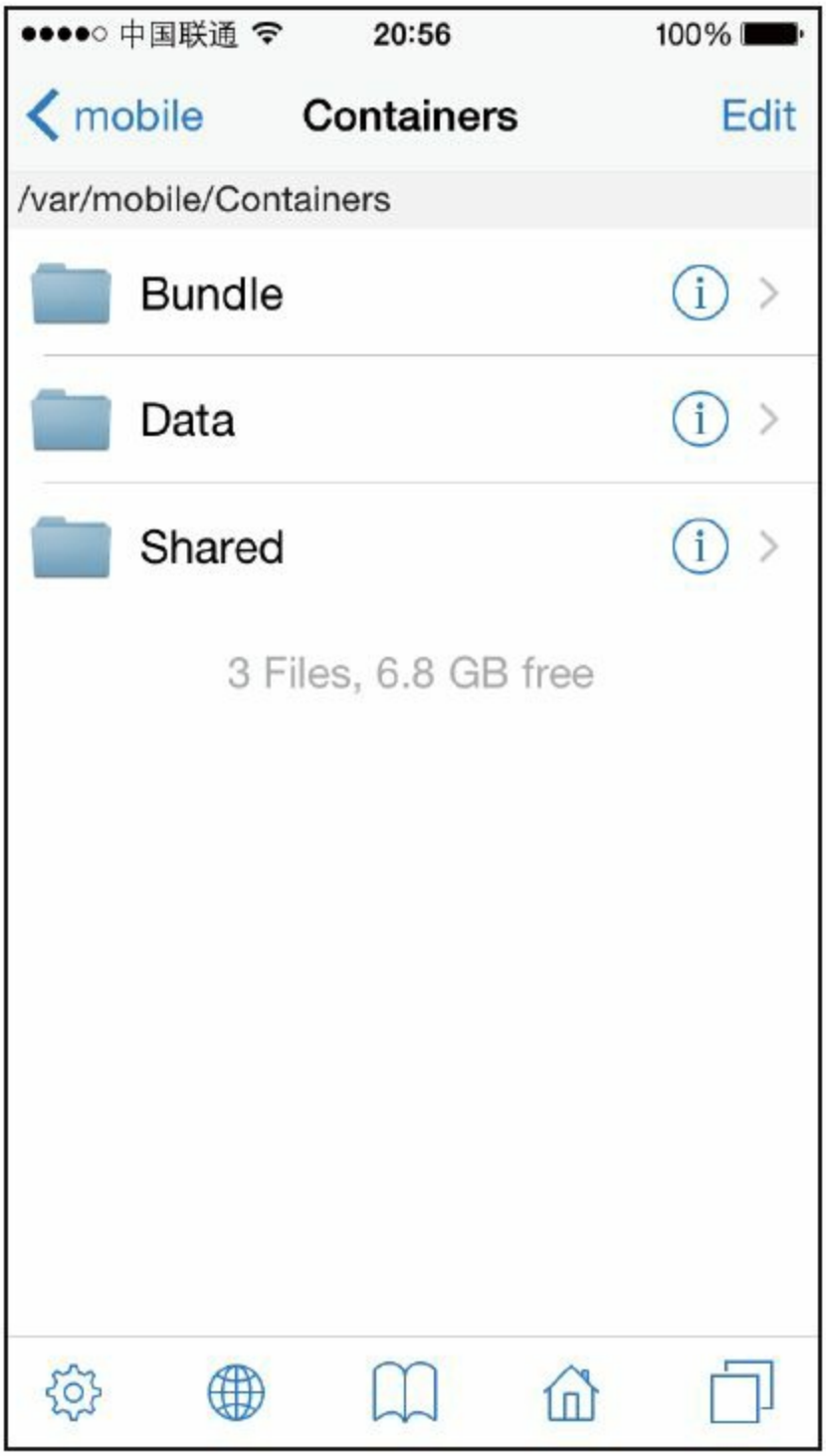


图2-9　/var/mobile/Containers

2.1.2　iOS文件权限简介

iOS是一个多用户操作系统。“用户”是一个抽象的概念，它代表对操作系统的所有权和使用权。比如，mobile用户无法调用reboot命令重启iOS，而root用户却可以；“组”是用户的一种组织方式，一个组可以包含多个用户，一个用户也可以属于多个组。

iOS中的每个文件都有一个属主用户和一个属主组，或者说这个用户和这个组拥有这个文件；每个文件都具有一系列权限，简单地说，权限的作用在于说明文件的属主用户能做什么，属主组能做什么，以及其他所有人能做什么。iOS用3位（bit）来表示文件的权限，从高位到低位分别是r（read）权限、w（write）权限，以及x（execute）权限。文件与用户的关系存在以下三种可能性：

· 此用户是属主用户；

· 此用户不是属主用户，但在属主组里；

· 此用户既不是属主用户，又不在属主组里。

所以需要用3*3位来表示一个文件的权限，如果某一位为1，则这一位代表的权限生效，否则无效。例如，111101101代表rwxr-xr-x，即该文件的属主用户拥有r、w、x权限，而属主组和其他所有人只具有r和x权限；同时，二进制的111101101转换成十六进制是755，也是一种常见的权限表示法。

事实上，除r、w、x权限外，文件还可以拥有SUID、SGID和sticky等特殊权限，它们的应用频率不高，因此不占用单独的权限位，而是以简化形式出现在x权限所在的权限位中。在iOS逆向工程初学阶段，一般接触不到这些特殊权限，仅作简单了解即可。如果你对它们感兴趣，可以阅读http://thegeekdiary.com/what-is-suid-sgid-and-sticky-bit/，也可以来http://bbs.iosre.com参与讨论。

2.2 iOS二进制文件类型

在iOS逆向工程初学阶段，我们的目标主要是Application、Dynamic Library（以下简称dylib）和Daemon这三类二进制文件，对它们的了解越深入，逆向工程就会越顺利。这三类文件分工不同，其目录结构和文件权限也有一些区别。

2.2.1 Application

Application就是我们最熟悉的App了。虽然对于大多数iOS开发者来说，工作都是在跟App打交道，但在iOS逆向工程中，关注App的侧重点与正向开发还是不尽相同的。了解下面的几个App相关概念，是开始逆向工程前的必备工作。

1.bundle

bundle的概念来源于NeXTSTEP，它不是一个文

件，而是一个按某种标准结构来组织的目录，其中包含了二进制文件及运行所需的资源。正向开发中常见的App和framework都是以bundle的形式存在的；在越狱iOS中常见的PreferenceBundle（如图2-10所示），可以看成是一种依附于Settings的App，结构与App类似，本质也是bundle。

图2-10　PreferenceBundle

Framework也是bundle，但framework的bundle中存放的是一个dylib，而不是可执行文件。相对来说，framework的地位比App更高，因为一个App的绝大多

数功能都是通过调用framework提供的接口来实现的。将某个bundle确立为逆向目标后，绝大多数逆向线索都可以在bundle内找到，这大大降低了逆向工程的复杂度。

2.App目录结构

在iOS逆向工程中，对App目录结构的熟悉程度是决定工作效率的重要因素。App目录的以下三个部分比较重要。

· Info.plist

Info.plist记录了App的基本信息，如bundle identifier、可执行文件名、图标文件名等。其中bundle identifier会在后续章节的CydiaSubstrate中成为tweak的重要配置信息。可以通过Xcode查看它的值，如图2-11所示。

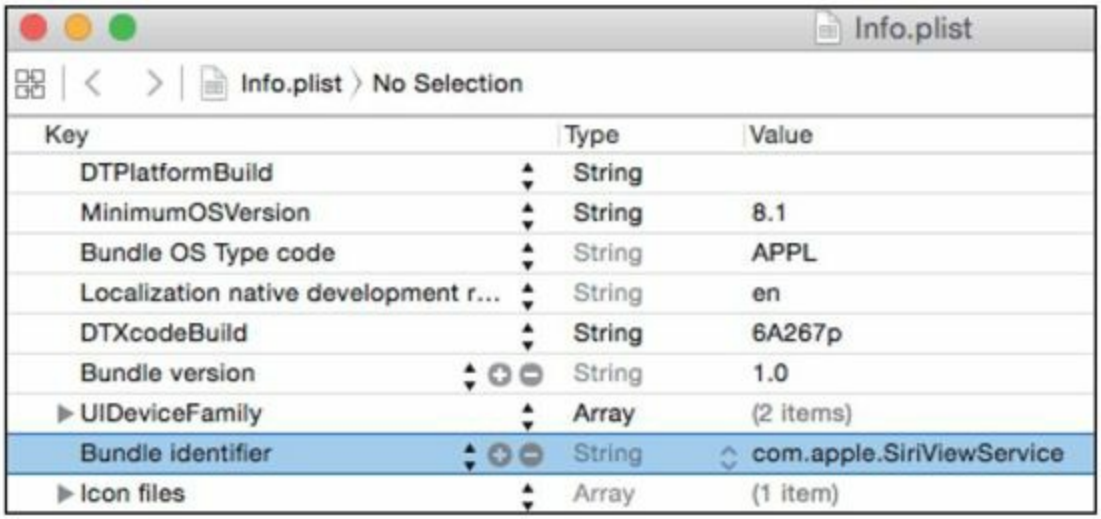


图2-11　用Xcode查看Info.plist

也可以通过Xcode自带的命令行工具plutil查看它的值，如下：

```
snakeninnysiMac:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Siri
 | grep CFBundleIdentifier
"CFBundleIdentifier" => "com.apple.SiriViewService"
```

本书主要采用plutil的方式来查看plist文件。

· 可执行文件

可执行文件的重要性不言而喻，它是App目录下最核心的部分，也是逆向工程最主要的目标。同样可

以通过Xcode和plutil两种方式来查看Info.plist，定位可执行文件。用Xcode查看Info.plist的界面如图2-12所示。

也可以通过Xcode自带的命令行工具plutil查看它的值，如下：

```
snakeninnysiMac:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Siri
 | grep CFBundleExecutable
"CFBundleExecutable" => "SiriViewService"
```

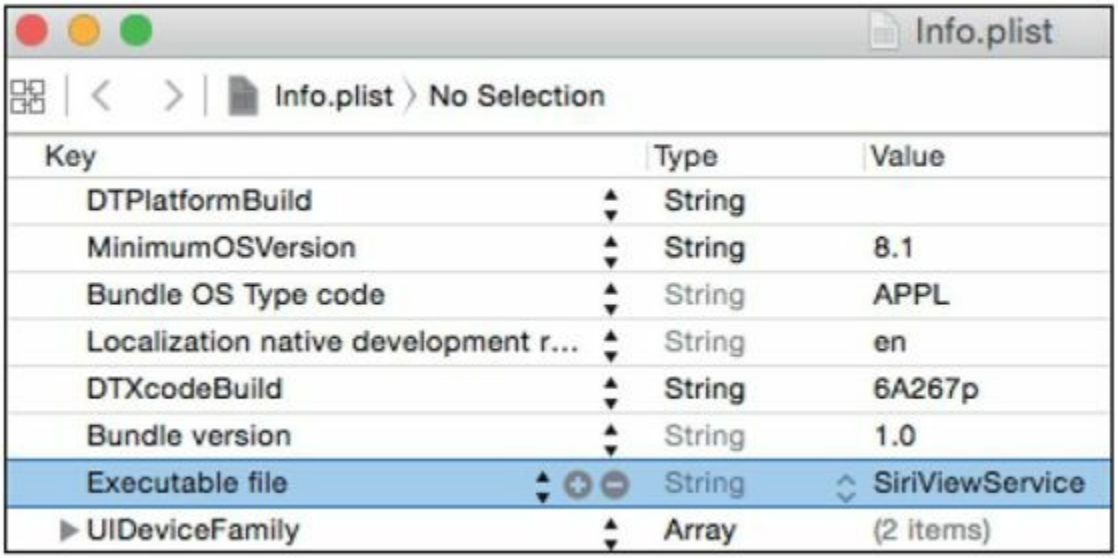


图2-12　用Xcode查看Info.plist

· lproj目录

lproj目录下存放的是各种本地化的字符串（.strings），是iOS逆向工程的重要线索，也可以用plutil查看，如下：

```
snakeninnysiMac:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Siri

{
  "ASSISTANT_INITIAL_QUERY_IPAD" => "What can I help you
with?"
  "ASSISTANT_BOREALIS_EDUCATION_SUBHEADER_IPAD" => "Just
say "Hey Siri" to learn more."
  "ASSISTANT_FIRST_UNLOCK_SUBTITLE_FORMAT" => "Your
passcode is required when %@ restarts"
……
```

这部分内容的应用场景会在第5章出现。

3.系统App VS.StoreApp

/Applications/目录存放系统App和从Cydia下载的App（我们把来自Cydia的App视为系统App），而/var/mobile/Containers/目录存放的则是StoreApp。虽然两者都是App，但它们在如下方面存在着一些差别。

· 目录结构

两种App的bundle内部目录结构区别不大，都含有Info.plist、可执行文件、lproj目录等，但是数据目录的位置不同：StoreApp的数据目录在/var/mobile/Containers/Data/下，以mobile权限运行的系统App的数据目录在/var/mobile/下，而以root权限运行的系统App的数据目录在/var/root/下。

· 安装包格式与权限

Cydia App的安装包格式一般是deb，StoreApp的安装包格式一般是ipa。其中deb是来自Debian的安装包格式，由Cydia作者saurik移植到iOS中，它的属主用户和属主组一般是root和admin，能够以root权限运行；而ipa是苹果为iOS推出的专属App安装包格式，属主用户和属主组都是mobile，只能以mobile权限运行。

· 沙盒（sandbox）

通俗地说，iOS中的沙盒就是一种访问限制机制，我们可以把它看作是权限的一种表现形式，授权文件（entitlements）也是沙盒的一部分。它是iOS最核心的安全组件之一，其实现很复杂，这里不过多讨论其细节。总的来说，沙盒会将App的文件访问范围限制在这个App内部，一个App一般不知道其他App的存在，更别说访问它们了；沙盒还会限制App的功能，例如对iCloud接口的调用就必须经过沙盒的允许。

在初学阶段，我们的目标不是沙盒，知道有这样一个东西存在就够了。在iOS逆向工程中，越狱本身已经破除了iOS的绝大多数安全限制，并对沙盒进行了一定程度的扩充，因此我们往往很容易忽略sandbox的存在，从而碰到一些看似很奇怪的问题。比如某个tweak不能写文件，调用了某个函数却没有

出现应有的效果，在确保自己的代码没有问题的前提下，就要回过头来检查这些问题是不是因为权限不够，或者沙盒限制造成的。App相关的概念不是用三言两语可以描述完的，如果有什么疑问，可以直接来http://bbs.iosre.com交流讨论。

2.2.2 Dynamic Library

大部分iOS开发者的日常工作应该都是写App，估计很少有人写过dylib，因此对dylib的概念很陌生。殊不知，在Xcode工程里导入的各种framework，链接的各种lib，其实本质都是dylib。可以用“file”命令验证一下，如下：

```
snakeninnysiMac:~ snakeninny$ file
/Users/snakeninny/Code/iOSSystemBinaries/8.1.1_iPhone5/Sy

/Users/snakeninny/Code/iOSSystemBinaries/8.1.1_iPhone5/Sy
 Mach-O dynamically linked shared library arm
```

如果把焦点转移到越狱iOS中，Cydia里的各种

tweak无一不是以dylib的形式工作的，正是这些tweak的存在让我们能够随意定制自己的iOS。在逆向工程中，我们会频繁接触各种dylib，因此有必要了解一些相关知识。

在iOS中，lib分为static和dynamic两种，其中static lib在编译阶段成为App可执行文件的一部分，会增加可执行文件的大小。因为App尺寸变大，启动时需要加载的内容变多，所以可能会导致App启动变慢。dylib则相对“智能”一些，它不会改变可执行文件的大小，只有当App需要用到这个dylib时，iOS才会把它加载进内存，成为App进程的一部分。

值得一提的是，dylib虽然充斥在iOS的各个角落，是逆向工程的重要目标类型，但其本身并不是可执行文件，不能独立运行，只能为别的进程服务，而且它们寄生在别的进程里，成为了这个进程的一部分。因此，dylib的权限是由它寄生的那个App决定

的，同一个dylib寄生在系统App和StoreApp里时的权限是不同的。例如，你写了一个Instagram的tweak，用来把喜欢的图片保存在本地，如果保存目录是/var/mobile/Containers/Data/下App对应的Documents目录，那么因为Instagram是一个StoreApp，这样的操作是没有问题的，tweak能够正常工作。而如果保存目录是/var/mobile/Documents，那么在兴高采烈地保存了一大堆美图，准备回头细细品味时，你就会发现/var/mobile/Documents里啥图片也没有——操作都被sandbox给禁掉了。

2.2.3 Daemon

相信本书的绝大部分读者从接触iOS开发的第一天起，就不断被苹果灌输这样一个观念——iOS中没有真正的后台多任务，你的App在后台将被大大限制。如果你是一个纯粹的App Store开发者，坚信并坚守这个观念，那么它将是你的App通过苹果审核的助

推剂；但既然你阅读了这本书，想要在学习逆向工程的同时了解一些官方文档没有阐述的事实，那么你就要保持冷静，理性思考。让我们一起回想一下iPhone上的一些现象。

1）当我们正在用iPhone上网或刷微博时来了一个电话，所有其他操作会立即中断，iOS第一时间将接听电话的界面呈现在我们面前。如果iOS中没有真正的后台多任务，系统是如何实时处理这个来电的呢？

2）对于那些经常收到垃圾短信和骚扰电话的朋友来说，类似于SMSNinja这样的防火墙软件必不可少。如果它不能常驻iOS后台，怎么能够实时地处理并过滤收到的每一条短信呢？

3）Backgrounder是一款iOS 5时代的插件，它能够帮助App实现真正的后台运行。有了它，我们再也不用担心因为push功能的不给力而漏收QQ消息啦！

如果iOS中没有真正的后台多任务，Backgrounder怎么会存在呢？

这些现象无一不说明iOS实际上存在真正的后台多任务。那么难道是苹果说错了？并不是！对于StoreApp来说，当用户按下home键时，进程就进入后台了，大多数功能都会被暂停；也就是说，对于遵纪守法的App Store开发者来说，可以把iOS看作是没有真正后台多任务的系统，因为你唯一能干的事不支持后台多任务。但iOS源于OSX，后者又跟所有类UNIX操作系统一样，有daemon（即守护进程，Windows称Service）的概念。越狱开放了iOS全文件系统，daemon也得以展现在我们面前。

Daemon为后台运行而生，给用户提供了各种“守护”，如imagent保障了iMessage的正确收发，mediaserverd处理了几乎所有的音频、视频，syslogd则用于记录系统日志等。iOS中的daemon主要由一个

可执行文件和一个plist文件构成。iOS的根进程是launchd，它会在开机时检查/System/Library/LaunchDaemons和/Library/LaunchDaemons下所有格式符合规定的plist文件，然后启动对应的daemon。这里的plist文件与App中的Info.plist文件作用类似，即记录daemon的基本信息，如下：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
/Users/snakeninny/Code/iOSSystemBinaries/
8.1.1_iPhone5/System/Library/LaunchDaemons/com.apple.imag

{
  "WorkingDirectory" => "/tmp"
  "Label" => "com.apple.imagent"
  "JetsamProperties" => {
    "JetsamMemoryLimit" => 3000
  }
  "EnvironmentVariables" => {
    "NSRunningFromLaunchd" => "1"
  }
  "POSIXSpawnType" => "Interactive"
  "MachServices" => {
    "com.apple.hsa-authentication-server" => 1
    "com.apple.imagent.embedded.auth" => 1
    "com.apple.incoming-call-filter-server" => 1
  }
  "UserName" => "mobile"
  "RunAtLoad" => 1
  "ProgramArguments" => [
    0 =>
"/System/Library/PrivateFrameworks/IMCore.framework/image

  ]
  "KeepAlive" => {
```

```
    "SuccessfulExit" => 0
  }
}
```

相对于App，daemon提供的功能要底层得多，逆向难度也要大得多，随意改动造成的后果当然也就严重得多，所以白苹果的惨案才会时有发生。在iOS逆向工程初学阶段，请不要把daemon当作练习目标；当你逆向了几个App，有了一定的心得和积累后再挑战这些daemon才是比较明智的选择。相比App，逆向daemon花费的时间和精力会更多，但更多的付出一定会带来更丰厚的回报。例如，“iOS上的第一款电话录音软件”Audio Recorder就是通过逆向mediaserverd这个daemon实现的。

2.3 小结

本章简单介绍了iOS系统结构和常见的二进制文件类型，它们都是App Store开发者不需要了解也接触不到的知识，在学习iOS逆向工程时很容易形成概念盲区。本章旨在科普那些在逆向工程中非常重要但苹果官方闭口不提的iOS系统级知识点，从而为App Store开发者打开iOS逆向工程的这扇窗。

若本章所涉及的每一个知识点深究下去都可以扩充成整整一章，但作为逆向工程初学者，了解这些概念之后知道碰到问题应该往哪个方向寻求解决方案，就达到了本章的目的。如果对上面的内容有任何疑问或者心得，都欢迎来http://bbs.iosre.com跟大家交流。

第二部分　工具篇

· 第3章　OSX工具集

· 第4章　iOS工具集

第一部分介绍了逆向工程的基本概念，从第二部分开始，将介绍在iOS逆向工程中用到的一系列工具。

相对于常规AppStore开发，iOS逆向工程工具的最大特点就是“杂”。在AppStore开发中，一个Xcode就可以完成绝大部分工作，它是苹果的嫡系出身，下载、安装和使用都非常方便。至于其他的一些插件、工具，所提供的只是一些锦上添花的功能，在开发中并不是必需的。

而在不那么常规的iOS逆向工程中，我们却不得不面对一长串叫起来都嫌绕口的工具。这就如同面前摆着两张餐桌，一张上摆着一碗面，碗上只放着一双

筷子，它叫Xcode；而另一张桌子上摆着大闸蟹和牛排，旁边横七竖八地堆满了蟹八件、刀、叉，等等，其中的几个大块头分别叫Theos、Reveal、IDA……

这些工具相互之间并没有紧密耦合、互相依赖的关系，整合度远没有Xcode高，因此在使用过程中得根据需要把它们手动组合起来。第二部分不可能涵盖所有逆向工程工具，不过相信大家在完全吃透本书内容之后，一定会具备举一反三的能力，届时可根据自己的需求寻找对应的工具，也可以来http://bbs.iosre.com上交流你的心得。

另外，因为需要介绍的工具颇多，略显杂乱，所以第二部分的内容分成两章，分别是OSX工具集和iOS工具集。本章所使用的iOS设备是iPhone 5，iOS版本是8.1。

第3章　OSX工具集

iOS逆向工程用到的一系列工具功能不同，角色各异，它们在OSX上完成的主要是开发和调试工作。iOS干这个活儿有些吃力，毕竟屏幕尺寸在这摆着呢。

本章主要介绍的工具有4个：class-dump、Theos、Reveal、IDA，其他的都是配套使用的辅助工具。

3.1 class-dump

class-dump，顾名思义，就是用来dump目标对象的class信息的工具。它利用Objective-C语言的runtime特性，将存储在Mach-O文件中的头文件信息提取出来，并生成对应的.h文件。

class-dump的用法比较简单，首先去http://stevenygard.com/projects/class-dump下载最新版的class-dump，如图3-1所示。

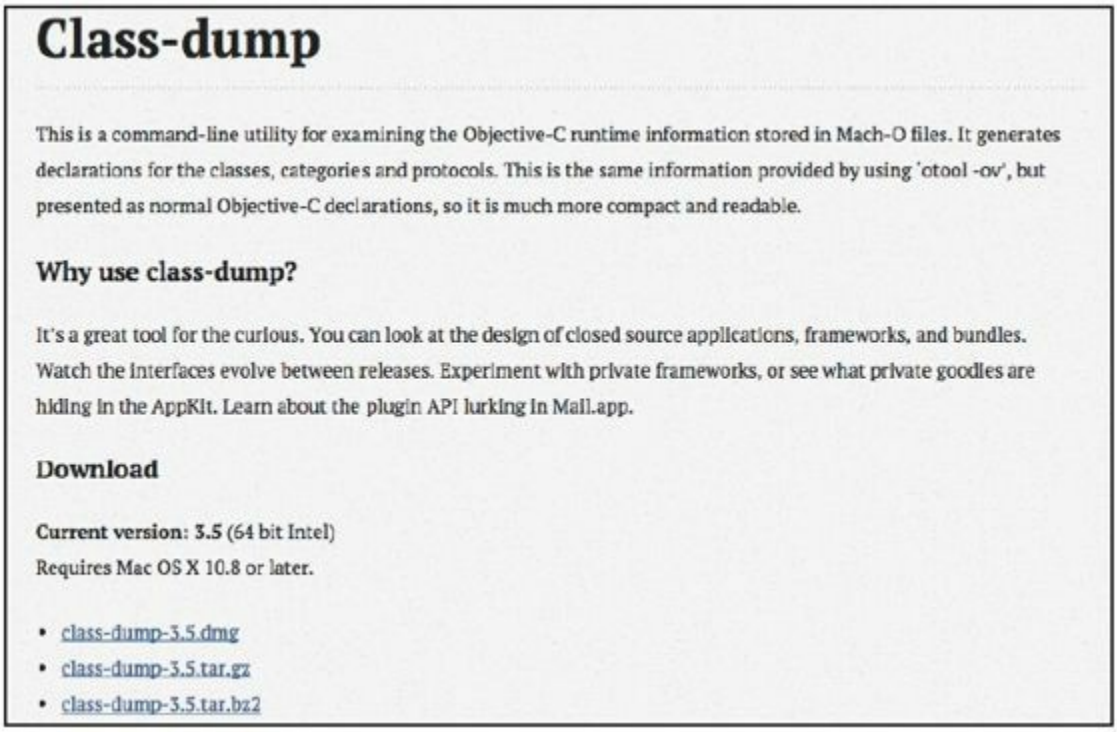


图3-1　class-dump主页

下载class-dump-3.5.dmg后，将dmg文件里的class-dump复制到“/usr/bin”下，然后在Terminal中执行“sudo chmod 777/usr/bin/class-dump”命令赋予其执行权限。运行class-dump，即可看到它的一些基本参数，如下：

```
snakeninnysiMac:~ snakeninny$ class-dump
class-dump 3.5 (64 bit)
Usage: class-dump [options] <mach-o-file>
  where options are:
        -a              show instance variable offsets
        -A              show implementation addresses
        --arch <arch>   choose a specific architecture
from a universal binary (ppc, ppc64, i386, x86_64, armv6,
armv7, armv7s, arm64)
        -C <regex>      only display classes matching
regular expression
        -f <str>        find string in method name
        -H              generate header files in current
directory, or directory specified with -o
        -I              sort classes, categories, and
protocols by inheritance (overrides -s)
        -o <dir>        output directory used for -H
        -r              recursively expand frameworks and
fixed VM shared libraries
        -s              sort classes and categories by
name
        -S              sort methods by name
        -t              suppress header in output, for
testing
        --list-arches   list the arches in the file, then
exit
        --sdk-ios       specify iOS SDK version (will look
in
```

```
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPh

        --sdk-mac       specify Mac OS X version (will
look in /Developer/SDKs/MacOSX<version>.sdk
        --sdk-root      specify the full SDK root path (or
use --sdk-ios/--sdk-mac for a shortcut)
```

class-dump的对象是Mach-O格式的二进制文件，如Framework的库文件和App的可执行文件。下面以笔者的一个App为例，来看看class-dump的完整流程。

1.定位App的可执行文件

首先把带class-dump的App拷贝到OSX中，笔者放在了“/Users/snakeninny”下。然后在Terminal中进入App所在的目录，并用Xcode自带的plutil工具查看Info.plist中的“CFBundleExecutable”字段，如下：

```
snakeninnysiMac:~ snakeninny$ cd
/Users/snakeninny/SMSNinja.app/
snakeninnysiMac:SMSNinja.app snakeninny$
snakeninnysiMac:SMSNinja.app snakeninny$ plutil -p
Info.plist | grep CFBundleExecutable
  "CFBundleExecutable" => "SMSNinja"
```

当前目录下的“SMSNinja”就是App的可执行文件。

2.class-dump可执行文件

把SMSNinja的头文件class-dump到“/path/to/headers/SMSNinja/”下，并将头文件内容按名字排序，命令如下：

```
snakeninnysiMac:SMSNinja.app snakeninny$ class-dump -S -s
-H SMSNinja -o /path/to/headers/SMSNinja/
```

大家可以用自己的App实践一下，然后用class-dump的头文件对比源文件中的头文件，是不是十分相似？除了一些参数类型被改成了id，参数名用arg1、arg2表示之外，几乎就是一模一样的吧？class-dump帮我们排序后，头文件的可读性甚至变高了。

使用class-dump分析自己的App没有太大意义，我们当然不会止步于此：同样是闭源应用，class-

dump既然能提取自己App里的头文件，自然也能提取别人App里的头文件！

透过这些头文件，闭源App的程序架构就能初现端倪了，经验丰富的开发人员可以从中了解非常多的信息，这些信息是iOS逆向工程的基础。不过现在的App工程越来越大，而且还在不停地引用第三方代码，因此经常会发现class-dump出来了成百上千个头文件，虽然靠人工及经验一点点地分析是很好的练习方式，但过程实在太复杂，让人头大。在后面的章节，将通过各种工具逐步缩小目标范围，最后精准地定位目标函数。

值得注意的是，从AppStore下载的App都是经过加密的，可执行文件被加上了一层“壳”，就像是一颗硬硬的核桃，class-dump应付不了这样的文件。你想想，我们试图用class-dump这样一个镊子来取肉，别说解馋，没把镊子夹坏就算不错啦！此时使用class-

dump看上去会“失效”。要想吃核桃，还得先用别的工具把壳砸开才行，具体的内容下一章就会揭晓。关于class-dump的更多用法，请持续关注http://bbs.iosre.com。

3.2 Theos

3.2.1 Theos简介

Theos是一个越狱开发工具包，由iOS越狱界知名人士Dustin Howett（@DHowett）开发并分享到GitHub上。Theos与其他越狱开发工具相比，最大的特点就是简单：下载安装简单、Logos语法简单、编译发布简单，可以让使用者把精力都放在开发工作上去。

值得一提的是，越狱开发中常用的另一工具iOSOpenDev是整合在Xcode里的，熟悉Xcode的朋友可能会对它更感兴趣。但逆向工程接触底层知识较多，很多东西无法自动化，因此推荐使用整合度并不算高的Theos，当你手动完成一个又一个练习时，对逆向工程的理解一定会更深。

这里插播一个关于DHowett的小段子：DHowett

的全名叫Dustin L.Howett，他是个很有个性的少年，出生在美国宾夕法尼亚州的郊区，从小痴迷电脑。大学读了不到一年，觉得老师讲得没意思，就不愿意好好听了，自然也就跟不上。更重要的是，他和一个姑娘展开了疯狂的异地恋，于是就干脆辍学，搬到了那个姑娘的所在地加州，并求职进了Saurik的公司SaurikIT。DHowett的早期作品CyDelete以Cy开头，而这种命名方式是Saurik御用的，说明DHowett的作品得到了Saurik的认可，也足见DHowett与Saurik关系之好。但遗憾的是，在Dustin辍学后，他和女朋友之间开始出现问题，最后分道扬镳了。之后Dustin离开了SaurikIT，进入了另一家创业公司DailyBooth，但这家公司经营不善，没多久就倒闭了，他就又回家待业了。过了没多久，Dustin爱上了另一个女孩，所以他又为了这个姑娘搬回旧金山，并且在当地一家不错的公司Airbnb找到了一份新工作。在我眼里，Dustin敢想敢干、敢爱敢恨、敢作敢当，真是“让我们红尘作

伴活得潇潇洒洒”，可以说是一个风一般的男子，令人十分崇拜。

3.2.2 安装Theos

1.安装Xcode与Command Line Tools

一般来说，iOS开发者都会安装Xcode，其中附带了Command Line Tools。如果还没有安装Xcode，请到Mac AppStore免费下载。如果安装了多个Xcode，需要使用xcode-select命令指定一个活动Xcode，即Theos默认使用的Xcode。假设安装了3个Xcode，并将它们分别命名为Xcode1.app、Xcode2.app和Xcode3.app，若要指定Xcode3为活动Xcode，则运行如下命令：

```
snakeninnys-MacBook:~ snakeninny$ sudo xcode-select -s
/Applications/Xcode3.app/Contents/Developer
```

2.下载Theos

从GitHub上下载Theos，操作如下：

```
snakeninnysiMac:~ snakeninny$ export THEOS=/opt/theos
snakeninnysiMac:~ snakeninny$ sudo git clone
git://github.com/DHowett/theos.git $THEOS
Password:
Cloning into '/opt/theos'...
remote: Counting objects: 4116, done.
remote: Total 4116 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (4116/4116), 913.55 KiB | 15.00
KiB/s, done.
Resolving deltas: 100% (2063/2063), done.
Checking connectivity... done
```

3.配置ldid

ldid是专门用来签名iOS可执行文件的工具，用以在越狱iOS中取代Xcode自带的codesign。从http://joedj.net/ldid下载ldid，把它放在“/opt/theos/bin/”下，然后用以下命令赋予它可执行权限：

```
snakeninnysiMac:~ snakeninny$ sudo chmod 777
/opt/theos/bin/ldid
```

4.配置CydiaSubstrate

首先运行Theos的自动化配置脚本，操作如下：

```
snakeninnysiMac:~ snakeninny$ sudo
/opt/theos/bin/bootstrap.sh substrate
Password:
Bootstrapping CydiaSubstrate...
 Compiling iPhoneOS CydiaSubstrate stub... default
target?
 failed, what?
 Compiling native CydiaSubstrate stub...
 Generating substrate.h header...
```

此处会遇到Theos的一个bug，它无法自动生成一个有效的libsubstrate.dylib文件，需要手动操作。解决方法很简单：首先在Cydia中搜索安装“CydiaSubstrate”（如图3-2所示）。

图3-2　CydiaSubstrate

然后用iFunBox或scp等方式将iOS上的“/Library/Frameworks/CydiaSubstrate.framework/Cydi贝到OSX中，将其重命名为libsubstrate.dylib后放到“/opt/theos/lib/libsubstrate.dylib”中，替换掉无效的文

件即可。

5.配置dpkg-deb

deb是越狱开发安装包的标准格式，dpkg-deb是一个用于操作deb文件的工具，有了这个工具，Theos才能正确地把工程打包成为deb文件。

从https://raw.githubusercontent.com/DHowett/dm.pl/master/下载dm.pl，将其重命名为dpkg-deb后，放到“/opt/theos/bin/”目录下，然后用以下命令赋予其可执行权限：

```
snakeninnysiMac:~ snakeninny$ sudo chmod 777
/opt/theos/bin/dpkg-deb
```

6.配置Theos NIC templates

Theos NIC templates内置了5种Theos工程类型的模板，方便创建多样的Theos工程。除此以外，还可

以从https://github.com/DHowett/theos-nic-templates/archive/master.zip获取额外的5种模板，下载后将解压得到的5个.tar文件复制到“/opt/theos/templates/iphone/”下即可。

3.2.3 Theos用法介绍

1.创建工程

1）更改工作目录至常用的iOS工程目录（如笔者的是“/Users/snakeninny/Code/”），然后输入“/opt/theos/bin/nic.pl”，启动NIC（New Instance Creator），如下：

```
snakeninnysiMac:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
```

可以看到，这里共有10种模板可供选择，其中1、4、6、8、9是Theos的自带模板，2、3、5、7、10是上一小节下载的。在逆向工程初级阶段，所开发程序的主要类型是tweak，其他模板的用法可以来http://bbs.iosre.com讨论交流。

2）选择“9”，即创建一个tweak工程，命令如下：

```
Choose a Template (required): 9
```

3）输入tweak的工程名称，命令如下：

```
Project Name (required): iOSREProject
```

4）输入deb包的名字（类似于bundle identifier），命令如下：

```
Package Name [com.yourcompany.iosreproject]:
com.iosre.iosreproject
```

5）输入tweak作者的名字，命令如下：

```
Author/Maintainer Name [snakeninny]: snakeninny
```

6）输入“MobileSubstrate Bundle filter”，也就是tweak作用对象的bundle identifier，命令如下：

```
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.apple.springboard
```

7）输入tweak安装完成后需要重启的应用，以进程名表示，如下：

```
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]: SpringBoard
Instantiating iphone/tweak in iosreproject/...
Done.
```

简单的7步完成之后，一个名为iosreproject的文件夹就在当前目录生成了，该文件夹里就是刚创建的tweak工程。

2.定制工程文件

用Theos创建tweak工程非常方便，但简洁的工程框架下目前还是些粗糙的内容，需要进一步加工相关的文件。先来看看刚刚生成的工程目录，如下：

```
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 40
-rw-r--r--  1 snakeninny  staff   184 Dec  3 09:05
Makefile
-rw-r--r--  1 snakeninny  staff  1045 Dec  3 09:05
Tweak.xm
-rw-r--r--  1 snakeninny  staff   223 Dec  3 09:05
control
-rw-r--r--  1 snakeninny  staff    57 Dec  3 09:05
iOSREProject.plist
lrwxr-xr-x  1 snakeninny  staff    11 Dec  3 09:05 theos
-> /opt/theos
```

除去一个指向Theos目录的符号链接外，只有4个文件，从工程复杂度来说完全不会吓跑初学者，反而会让我们跃跃欲试，Theos的产品体验做得很好。

古语云：“一粒米中藏世界，半边锅内煮乾坤”。4根顶梁柱就足以撑起tweak的毛坯房，但漂亮的tweak离不开我们的精装修，这4个文件的内容可是大有玄机！

（1）Makefile

Makefile文件指定工程用到的文件、框架、库等信息，将整个过程自动化。iOSREProject的Makefile内容如下：

```
include theos/makefiles/common.mk
TWEAK_NAME = iOSREProject
iOSREProject_FILES = Tweak.xm
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
	install.exec "killall -9 SpringBoard"
```

下面来逐行解读。

```
include theos/makefiles/common.mk
```

固定写法，不要更改。

```
TWEAK_NAME = iOSREProject
```

tweak的名字，即用Theos创建工程时指定的“Project Name”，跟control文件中的“Name”字段对应，不要更改。

```
iOSREProject_FILES = Tweak.xm
```

tweak包含的源文件（不包括头文件），多个文件间以空格分隔，如：

```
iOSREProject_FILES = Tweak.xm Hook.xm New.x ObjC.m
ObjC++.mm
```

可以按需更改。

```
include $(THEOS_MAKE_PATH)/tweak.mk
```

根据不同的Theos工程类型，通过include命令指定不同的.mk文件；在逆向工程初级阶段，我们开发的一般是Application、Tweak和Tool三种类型的程序，它们对应的.mk文件分别是application.mk、tweak.mk和tool.mk，可以按需更改。

```
after-install::
      install.exec "killall -9 SpringBoard"
```

读者应该从字面意思就能对这两行的作用猜个八

九不离十——在tweak安装之后杀掉SpringBoard进程，好让CydiaSubstrate在进程启动时加载对应的dylib。

是不是非常简单？Makefile里的默认内容确实非常简单，但有点简单过头了。如何指定SDK版本？怎么导入framework？lib文件在哪里链接？作为iOS开发者的你一定会提出这些问题。别急别急，面包会有的，牛奶也会有的。

· 指定处理器架构

```
ARCHS = armv7 arm64
```

上面的语句在表示不同的处理器架构时，其间以空格分隔。值得注意的是，采用arm64架构的App不兼容armv7/armv7s架构，必须适配arm64架构的dylib。在绝大多数情况下，这里固定填写“arm7 arm64”就行了。

· 指定SDK版本

```
TARGET = iphone:Base SDK:Deployment Target
```

比如：

```
TARGET = iphone:8.1:8.0
```

上面的语句即指定采用8.1版本的SDK，且发布对象为iOS 8.0及以上版本。也可以把“Base SDK”设置为“latest”，指定以Xcode附带的最新版本SDK编译，如：

```
TARGET = iphone:latest:8.0
```

· 导入framework

```
iOSREProject_FRAMEWORKS = framework name
```

例如：

```
iOSREProject_FRAMEWORKS = UIKit CoreTelephony CoreAudio
```

上面的语句所展示的功能没什么多说的，但既然是tweak开发，很多朋友关注的应该是如何导入private framework吧？很简单，用下面的语句即可：

```
iOSREProject_PRIVATE_FRAMEWORKS = private framework name
```

例如：

```
iOSREProject_PRIVATE_FRAMEWORKS = AppSupport ChatKit
IMCore
```

虽然只是多了个“PRIVATE”，但有一点要注意：private framework是AppStore开发所不允许使用的，它的内容在每个iOS版本之间可能发生变化，在导入之前，一定要确定导入的private framework确实存在。举一个例子，如果你的tweak打算兼容iOS 7和iOS 8两个版本，那么Makefile可写成如下内容：

```
ARCHS = armv7 arm64
TARGET = iphone:latest:7.0
include theos/makefiles/common.mk
TWEAK_NAME = iOSREProject
iOSREProject_FILES = Tweak.xm
```

```
iOSREProject_PRIVATE_FRAMEWORK = BaseBoard
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
      install.exec "killall -9 SpringBoard"
```

上面的语句可以成功编译和链接，并不会报错。但是，因为BaseBoard这个private framework只存在于8.0及以上版本的SDK里，在iOS 7里是没有的，所以这个tweak在iOS 7中会因找不到framework而无法正常工作。这种情况可以通过弱链接（谷歌搜索“makefile weak linking”）或dlopen()、dlsym()和dlclose()系列函数动态调用private framework来解决。

· 链接Mach-O对象（Mach-O object）

```
iOSREProject_LDFLAGS = -lx
```

Theos采用GNU Linker来链接Mach-O对象，包括.dylib、.a和.o。在Terminal中输入“man ld”，定位到“-lx”部分，它是这么写的：

“-lx This option tells the linker to search for

libx.dylib or libx.a in the library search path.If string x is of the form y.o,then that file is searched for in the same places,but without prepending`lib’ or appending`.a’ or`.dylib’ to the filename.”

大致意思是说，-lx代表链接libx.a或libx.dylib，即给“x”加上“lib”的前缀，以及“.a”或“.dylib”的后缀；如果x是“y.o”的形式，则直接链接y.o，不加任何前缀或后缀。由图3-3可知，iOS支持链接的Mach-O对象全是以“libx.dylib”和“y.o”形式命名的，完全兼容GNU Linker。

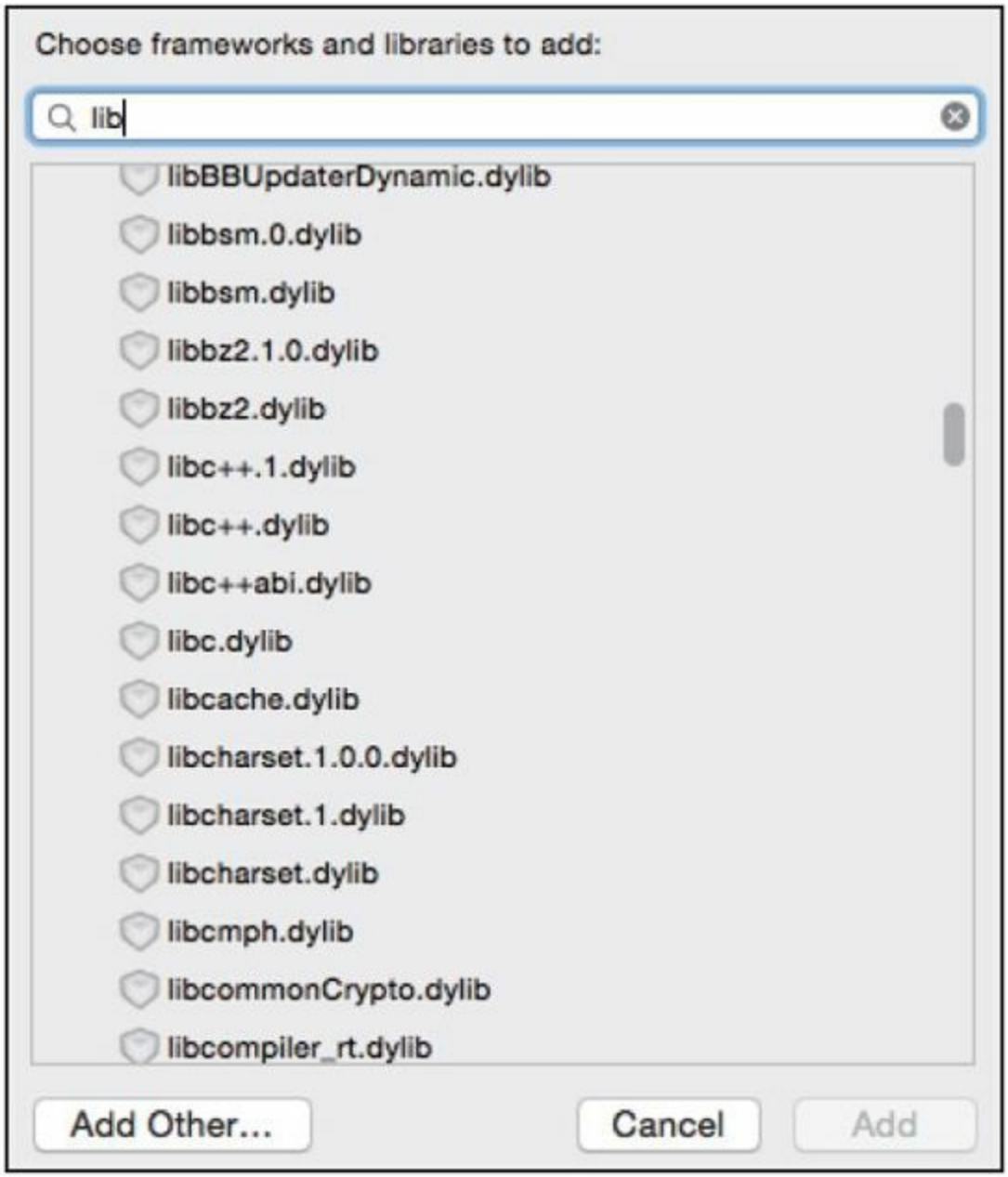


图3-3　链接Mach-O对象

这样，链接Mach-O对象就很方便了。例如，要链接libsqlite3.0.dylib、libz.dylib和dylib1.o，像下面这么写就可以了：

```
iOSREProject_LDFLAGS = -lz -lsqlite3.0 -dylib1.o
```

稍后还有一个字段需要介绍，但一般来说，Makefile中定义了以上字段就已经完全够用了；更详细的Makefile介绍，可以参阅http://www.gnu.org/software/make/manual/html_node/M

（2）Tweak.xm

用Theos创建tweak工程，默认生成的源文件是Tweak.xm。“xm”中的“x”代表这个文件支持Logos语法，如果后缀名是单独一个“x”，说明源文件支持Logos和C语法；如果后缀名是“xm”，说明源文件支持Logos和C/C++语法，与“m”和“mm”的区别类似。Tweak.xm的内容如下：

```
/* How to Hook with Logos
Hooks are written with syntax similar to that of an
Objective-C @implementation.
You don't need to #include <substrate.h>, it will be done
automatically, as will
the generation of a class list and an automatic
constructor.
%hook ClassName
// Hooking a class method
+ (id)sharedInstance {
```

```
        return %orig;
}
// Hooking an instance method with an argument.
- (void)messageName:(int)argument {
      %log; // Write a message about this call, including
its class, name and arguments, to the system log.
      %orig; // Call through to the original function
with its original arguments.
      %orig(nil); // Call through to the original
function with a custom argument.
      // If you use %orig(), you MUST supply all
arguments (except for self and _cmd, the automatically
generated ones.)
}
// Hooking an instance method with no arguments.
- (id)noArguments {
      %log;
      id awesome = %orig;
      [awesome doSomethingElse];
      return awesome;
}
// Always make sure you clean up after yourself; Not
doing so could have grave consequences!
%end
*/
```

这就是最基本的Logos语法，包含%hook、%log、%orig这3个预处理指令，它们的作用如下。

· %hook

指定需要hook的class，必须以%end结尾，如下：

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
```

```
{
        NSLog(@"You've pressed home button.");
        %orig; // call the original _menuButtonDown:
}
%end
```

这段代码的意思是钩住（hook）SpringBoard类里的_menuButtonDown:函数，先将一句话写入syslog，再执行函数的原始操作。

· %log

该指令在%hook内部使用，将函数的类名、参数等信息写入syslog，可以以%log([(<type>)<expr>,...])的格式追加其他打印信息，如下：

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
      %log((NSString *)@"iOSRE", (NSString *)@"Debug");
      %orig; // call the original _menuButtonDown:
}
%end
```

打印结果如下：

```
Dec  3 10:57:44 FunMaker-5 SpringBoard[786]: -
[<SpringBoard: 0x150eb800> _menuBu-
```

```
ttonDown:++++++++++++++++++++++++++++++++++++++++++++++++++

         Timestamp:           75607608282
         Total Latency:       20266 us
         SenderID:            0x0000000100000190
         BuiltIn:             1
         AttributeDataLength: 16
         AttributeData:       01 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00
         ValueType:           Absolute
         EventType:           Keyboard
         UsagePage:           12
         Usage:               64
         Down:                1

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

         ]: iOSRE, Debug
```

· %orig

该指令在%hook内部使用，执行被钩住（hook）的函数的原始代码，如下：

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
      NSLog(@"You've pressed home button.");
      %orig; // call the original _menuButtonDown:
}
%end
```

如果去掉%orig，那么原始函数不会得到执行，例如：

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
      NSLog(@"You've pressed home button but it's not
functioning.");
}
%end
```

还可以利用%orig更改原始函数的参数，例如：

```
%hook SBLockScreenDateViewController
- (void)setCustomSubtitleText:(id)arg1 withColor:(id)arg2
{
      %orig(@"iOS 8 App Reverse Engineering", arg2);
}
%end
```

这样一来，锁屏界面原本显示日期的地方就变成了如图3-4所示的样子。

图3-4　更改锁屏界面

除了%hook、%log、%orig以外，Logos常用的预处理指令还有%group、%init、%ctor、%new、%c，下面继续逐一介绍。

· %group

该指令用于将%hook分组，便于代码管理及按条件初始化分组（含义稍后有详细解释），必须以%end结尾；一个%group可以包含多个%hook，所有不属于某个自定义group的%hook会被隐式归类到%group_ungrouped中。%group的用法如下：

```
%group iOS7Hook
%hook iOS7Class
- (id)iOS7Method
{
        id result = %orig;
        NSLog(@"This class & method only exist in iOS
7.");
        return result;
}
%end
%end // iOS7Hook
%group iOS8Hook
%hook iOS8Class
- (id)iOS8Method
{
        id result = %orig;
        NSLog(@"This class & method only exist in iOS
8.");
        return result;
}
%end
%end // iOS8Hook
%hook SpringBoard
-(void)powerDown
{
        %orig;
}
%end
```

这段代码的意思是在%group iOS7Hook中钩住iOS7Class的iOS7Method，在%group iOS8Hook中钩住iOS8Class的iOS8Method函数，然后在%group_ungrouped中钩住SpringBoard类的powerDown函数。

需要注意的是，%group必须配合下面的%init使用才能生效。

· %init

该指令用于初始化某个%group，必须在%hook或%ctor内调用；如果带参数，则初始化指定的group，如果不带参数，则初始化_ungrouped，如下：

```
#ifndef kCFCoreFoundationVersionNumber_iOS_8_0
#define kCFCoreFoundationVersionNumber_iOS_8_0 1140.10
#endif
%hook SpringBoard
- (void)applicationDidFinishLaunching:(id)application
{
        %orig;
        %init; // Equals to %init(_ungrouped)
        if (kCFCoreFoundationVersionNumber >=
kCFCoreFoundationVersionNumber_iOS_7_0 &&
kCFCoreFoundationVersionNumber <
kCFCoreFoundationVersionNumber_iOS_8_0) %init(iOS7Hook);
```

```
        if (kCFCoreFoundationVersionNumber >=
kCFCoreFoundationVersionNumber_iOS_8_0) init(iOS8Hook);
}
%end
```

只有调用了%init，对应的%group才能起作用，切记切记！

· %ctor

tweak的constructor，完成初始化工作；如果不显式定义，Theos会自动生成一个%ctor，并在其中调用%init(_ungrouped)。因此，

```
%hook SpringBoard
- (void)reboot
{
        NSLog(@"If rebooting doesn't work then I'm
screwed.");
        %orig;
}
%end
```

可以成功生效，因为Theos隐式定义了如下内容：

```
%ctor
{
```

```
        %init(_ungrouped);
}
```

而

```
%hook SpringBoard
- (void)reboot
{
        NSLog(@"If rebooting doesn't work then I'm
screwed.");
        %orig;
}
%end
%ctor
{
        // Need to call %init explicitly!
}
```

里的%hook无法生效，因为这里显式定义了%ctor，却没有显式调用%init，%group(_ungrouped)不起作用。%ctor一般可以用来初始化%group，以及进行MSHookFunction等操作，如下：

```
#ifndef kCFCoreFoundationVersionNumber_iOS_8_0
#define kCFCoreFoundationVersionNumber_iOS_8_0 1140.10
#endif
%ctor
{
        %init;
        if (kCFCoreFoundationVersionNumber >=
kCFCoreFoundationVersionNumber_iOS_7_0 &&
kCFCoreFoundationVersionNumber <
kCFCoreFoundationVersionNumber_iOS_8_0) %init(iOS7Hook);
        if (kCFCoreFoundationVersionNumber >=
```

```
kCFCoreFoundationVersionNumber_iOS_8_0) %init(iOS8Hook);
        MSHookFunction((void
*)&AudioServicesPlaySystemSound,
                (void
*)&replaced_AudioServicesPlaySystemSound,
                (void
**)&original_AudioServicesPlaySystemSound);
}
```

注意，%ctor不需要以%end结尾。

· %new

在%hook内部使用，给一个现有class添加新函数，功能与class_addMethod相同。它的用法如下：

```
%hook SpringBoard
%new
- (void)namespaceNewMethod
{
      NSLog(@"We've added a new method to SpringBoard.");
}
%end
```

有的朋友可能会问，Objective-C的category语法也可以给现有class添加新函数，为什么还需要%new呢？其实原因就在于category与class_addMethod的区别，前者是静态的，而后者是动态的。那么在这种情

况下，静态还是动态，有什么关系呢？当然有关系，尤其是当class来自某个可执行文件的时候。举个例子，上面的代码给SpringBoard类添加了一个新方法，如果使用category，代码应该是下面这样：

```
@interface SpringBoard (iOSRE)
- (void)namespaceNewMethod;
@end
@implementation SpringBoard (iOSRE)
- (void)namespaceNewMethod
{
      NSLog(@"We've added a new method to SpringBoard.");
}
@end
```

如果尝试编译上面的代码，会得到“error：cannot find interface declaration for‘SpringBoard’”的报错信息，即编译器找不到SpringBoard类的定义。可以构造一个SpringBoard的定义，骗过编译器，如下：

```
@interface SpringBoard : NSObject
@end
@interface SpringBoard (iOSRE)
- (void)namespaceNewMethod;
@end
@implementation SpringBoard (iOSRE)
- (void)namespaceNewMethod
{
      NSLog(@"We've added a new method to SpringBoard.");
}
```

```
@end
```

重新编译，仍然会报错，如下：

```
Undefined symbols for architecture armv7:
  "_OBJC_CLASS_$_SpringBoard", referenced from:
      l_OBJC_$_CATEGORY_SpringBoard_$_iOSRE in
Tweak.xm.b1748661.o
ld: symbol(s) not found for architecture armv7
clang: error: linker command failed with exit code 1 (use
-v to see invocation)
```

ld找不到“SpringBoard”的定义。一般来说，iOS程序员在碰到这个错误时的第一反应是：“是不是忘了导入哪个framework？”，但是转念一想，SpringBoard类是SpringBoard这个App里的一个类，而不是一个framework，要怎么导入？现在你是不是觉得%new非常可爱了呢？

· %c

该指令的作用等同于objc_getClass或NSClassFromString，即动态获取一个类的定义，在%hook或%ctor内使用。

Logos的预处理指令还有%subclass和%config，但笔者到现在也没有用过，感兴趣的读者可以移步http://iphonedevwiki.net/index.php/Logos一探究竟，也可以来http://bbs.iosre.com跟大家一起讨论。

（3）control

control文件记录了deb包管理系统所需的基本信息，会被打包进deb包里。iOSREProject里control文件的内容如下：

```
Package: com.iosre.iosreproject
Name: iOSREProject
Depends: mobilesubstrate
Version: 0.0.1
Architecture: iphoneos-arm
Description: An awesome MobileSubstrate tweak!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
```

其中：

· Package字段用于描述这个deb包的名字，采用的命名方式同bundle identifier类似，均为反向DNS格

式，可以按需更改；

· Name字段用于描述这个工程的名字，可以按需更改；

· Depends字段用于描述这个deb包的“依赖”。“依赖”指的是这个程序运行的基本条件，可以填写固件版本或其他程序，如果当前iOS不满足“依赖”中定义的条件，则此tweak无法正常运行。如

```
Depends: mobilesubstrate, firmware (>= 8.0)
```

表示当前iOS版本必须在8.0以上，且必须安装CydiaSubstrate，才能正常运行这个tweak，可以按需更改。

· Version字段用于描述这个deb包的版本号，可以按需更改；

· Architecture字段用于描述deb包安装的目标设备架构，不要更改；

· Description字段是deb包的简单介绍，可以按需更改；

· Maintainter字段用于描述deb包的维护人，例如BigBoss源中所有deb包的维护人均为BigBoss，而非软件作者，可以按需更改；

· Author字段用于描述tweak的作者（注意与Maintainer的区别），可以按需更改；

· Section字段用于描述deb包所属的程序类别，不要更改。

control文件中可以自定义的字段还有很多，但上面这些信息就已经足够了。更全面的说明可以参阅debian的官方网站（http://www.debian.org/doc/debian-policy/ch-controlfields.html）或留意其他deb包里的

control文件。值得注意的是，Theos在打包deb时会对control文件作进一步处理，上面的control文件在得到处理后内容变为：

```
Package: com.iosre.iosreproject
Name: iOSREProject
Depends: mobilesubstrate
Architecture: iphoneos-arm
Description: An awesome MobileSubstrate tweak!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Version: 0.0.1-1
Installed-Size: 104
```

这里Theos更改了Version字段，用以表示Theos的打包次数，方便管理；增加了一个Installed-Size字段，用以描述deb包安装后的估算大小，可能会与实际大小有偏差，但不要更改。

control文件中的很多信息直接体现在Cydia中，如图3-5所示，大家可以对比看看。

图3-5　control信息在Cydia中的体现

（4）iOSREProject.plist

这个plist文件的作用和App中的Info.plist类似，它记录了一些配置信息，描述了tweak的作用范围。我们可以用plutil，也可以用Xcode来编辑它。

iOSREProject.plist的最外层是一个dictionary，只有一个名为“Filter”的键，如图3-6所示。

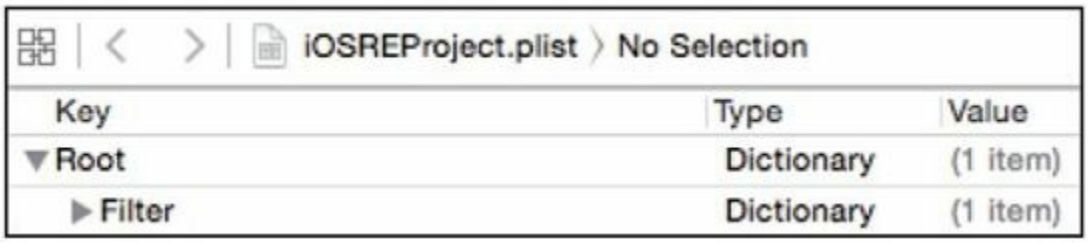


图3-6　iOSREProject.plist

Filter下是一系列array，可以分为三类。

· Bundles，指定若干bundle为tweak的作用对象，如图3-7所示。

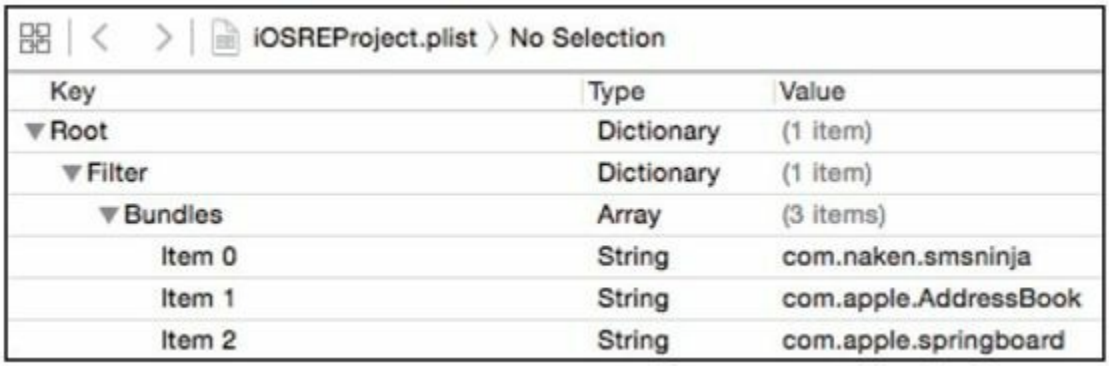


图3-7　Bundles

按照图3-7中的配置，tweak的作用对象是三个

bundle，即SMSNinja、AddressBook.framework和SpringBoard。

· Classes，指定若干class为tweak的作用对象，如图3-8所示。

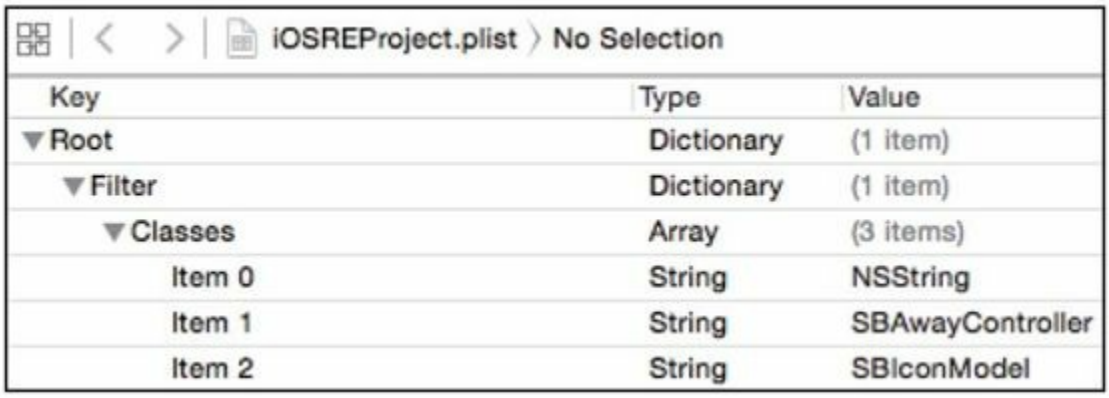


图3-8　Classes

按照图3-8的配置，tweak的作用对象是三个class，即NSString、SBAwayController和SBIconModel。

· Executables，指定若干可执行文件为tweak的作用对象，如图3-9所示。

按照图3-9中的配置，tweak的作用对象是三个可

执行文件，即callservicesd、imagent和mediaserverd。

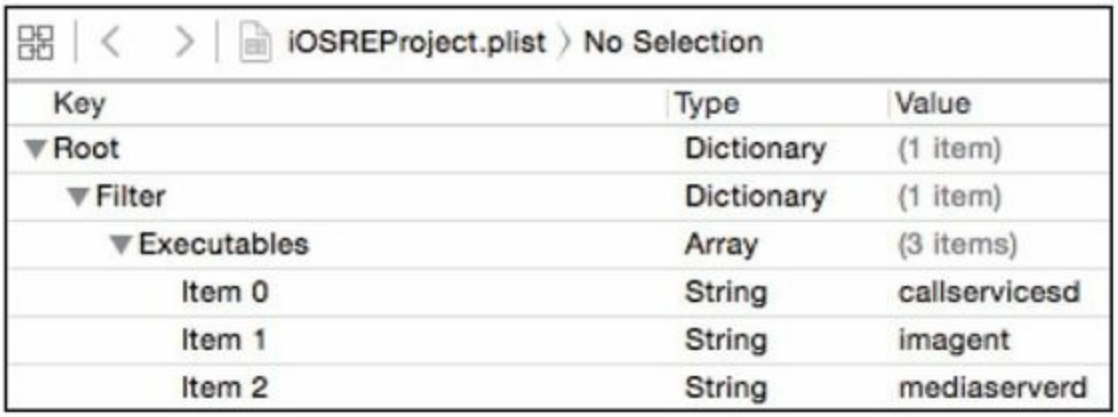


图3-9　Executables

这三类array可以混合使用，如图3-10所示。

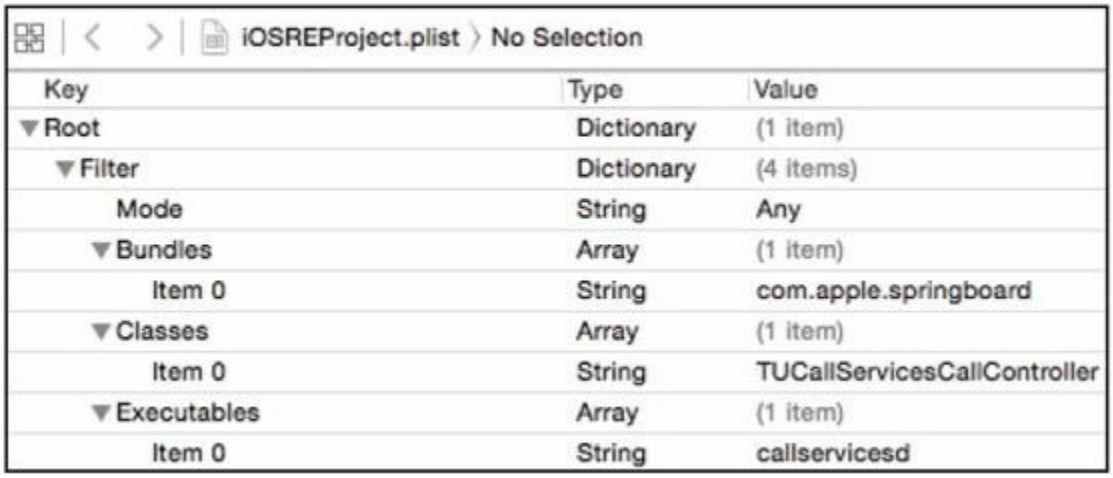


图3-10　混合三类array

注意，当Filter下有不同类的array时，需要添加一个“Mode：Any”键值对。当Filter下的array只有一类

时，不需要添加“Mode：Any”键值对。

3.编译+打包+安装

前面在完成了Theos的安装后，使用NIC创建了第一个tweak工程，还逐一解读了工程的组成文件，那么现在就剩下最后一步——编译了。完成这一步，一个tweak就算正式完成——我们可以把tweak安装到设备上，开始周而复始的“safe mode”之旅了，是不是很期待呢？

（1）编译

Theos采用“make”命令来编译Theos工程。在Theos工程目录下运行make命令，如下：

```
snakeninnysiMac:iosreproject snakeninny$ make
Making all for tweak iOSREProject...
 Preprocessing Tweak.xm...
 Compiling Tweak.xm...
 Linking tweak iOSREProject...
 Stripping iOSREProject...
 Signing iOSREProject...
```

从输出的信息看，Theos完成了预处理、编译、签名等一系列动作，此时会发现当前目录下多了一个新的“obj”文件夹，如下：

```
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 32
-rw-r--r--  1 snakeninny  staff  262 Dec  3 09:20
Makefile
-rw-r--r--  1 snakeninny  staff    0 Dec  3 11:28
Tweak.xm
-rw-r--r--  1 snakeninny  staff  223 Dec  3 09:05 control
-rw-r--r--@ 1 snakeninny  staff  175 Dec  3 09:48
iOSREProject.plist
drwxr-xr-x  5 snakeninny  staff  170 Dec  3 11:28 obj
lrwxr-xr-x  1 snakeninny  staff   11 Dec  3 09:05 theos -
> /opt/theos
```

里面有一个.dylib文件，如下：

```
snakeninnysiMac:iosreproject snakeninny$ ls -l ./obj
total 272
-rw-r--r--  1 snakeninny  staff  33192 Dec  3 11:28
Tweak.xm.b1748661.o
-rwxr-xr-x  1 snakeninny  staff  98784 Dec  3 11:28
iOSREProject.dylib
```

它就是tweak的核心。

（2）打包

打包使用的“make package”命令来自于Theos本身，其实就是先执行“make”命令，然后再执行“dpkg-deb”命令，如下：

```
snakeninnysiMac:iosreproject snakeninny$ make package
Making all for tweak iOSREProject...
 Preprocessing Tweak.xm...
 Compiling Tweak.xm...
 Linking tweak iOSREProject...
 Stripping iOSREProject...
 Signing iOSREProject...
Making stage for tweak iOSREProject...
dm.pl: building package `com.iosre.iosreproject' in
`./com.iosre.iosreproject_0.0.1-7_iphoneos-arm.deb'.
```

上面生成了一个名为“com.iosre.iosreproject_0.0.1-7_iphoneos-arm.deb”的文件，这就是可以最终发布的安装包。

“make package”命令还有一个很重要的功能。在执行完“make package”之后，除了“obj”文件夹外，你会发现tweak工程目录下还生成了一个“_”文件夹，如下：

```
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 40
-rw-r--r--  1 snakeninny  staff   262 Dec  3 09:20
```

```
Makefile
-rw-r--r--  1 snakeninny  staff     0 Dec  3 11:28
Tweak.xm
drwxr-xr-x  4 snakeninny  staff   136 Dec  3 11:35 _
-rw-r--r--  1 snakeninny  staff  2396 Dec  3 11:35
com.iosre.iosreproject_0.0.1-7 _iphoneos-arm.deb
-rw-r--r--  1 snakeninny  staff   223 Dec  3 09:05
control
-rw-r--r--@ 1 snakeninny  staff   175 Dec  3 09:48
iOSREProject.plist
drwxr-xr-x  5 snakeninny  staff   170 Dec  3 11:35 obj
lrwxr-xr-x  1 snakeninny  staff    11 Dec  3 09:05 theos
-> /opt/theos
```

这个文件夹是干什么的？打开它，可以看到2个文件夹，分别是“DEBIAN”和“Library”：

```
snakeninnysiMac:iosreproject snakeninny$ ls -l _
total 0
drwxr-xr-x  3 snakeninny  staff  102 Dec  3 11:35 DEBIAN
drwxr-xr-x  3 snakeninny  staff  102 Dec  3 11:35 Library
```

其中“DEBIAN”里只有tweak工程里的control文件，Theos在编译过程中向control文件里稍稍增加了几个字段而已，如下：

```
snakeninnysiMac:iosreproject snakeninny$ ls -l _/DEBIAN
total 8
-rw-r--r--  1 snakeninny  staff  245 Dec  3 11:35 control
```

“Library”的目录结构如图3-11所示。

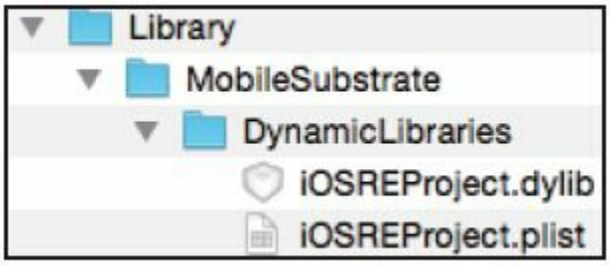


图3-11　Library目录结构

对比生成deb的包内容：

```
snakeninnysiMac:iosreproject snakeninny$ dpkg -c
com.iosre.iosreproject_0.0.1-7_iphoneos-arm.deb
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35 ./
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35
./Library/
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35
./Library/MobileSubstrate/
drwxr-xr-x snakeninny/staff  0 2014-12-03 11:35
./Library/MobileSubstrate/DynamicLibraries/
-rwxr-xr-x snakeninny/staff 98784 2014-12-03 11:35
./Library/MobileSubstrate/DynamicLibraries/iOSREProject.d

-rw-r--r-- snakeninny/staff   175 2014-12-03 11:35
./Library/MobileSubstrate/DynamicLibraries/iOSREProject.p
```

以及在Cydia中iOSREProject的文件系统，如图3-12所示。

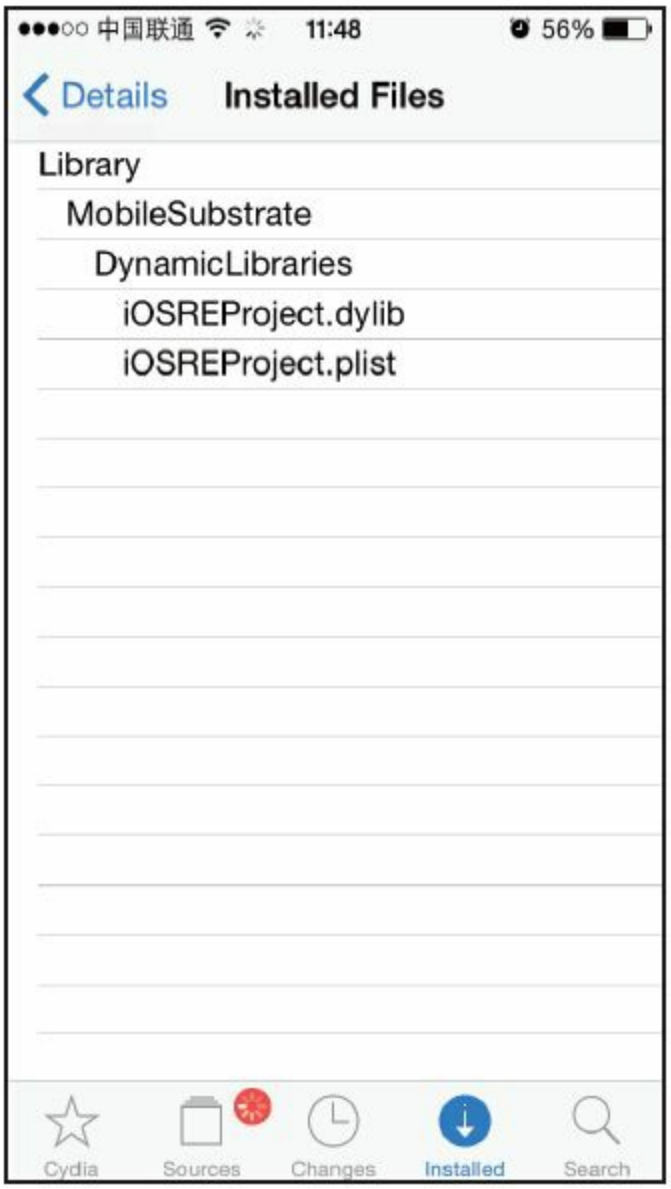


图3-12　iOSREProject文件系统

可以看到，三者是完全相同的。到这里，你可能也猜到了，这个deb包其实就是由“DEBIAN”提供debian信息，“Library”提供实际文件的简单组合。事实上，还可以在工程目录下创建一个名为“layout”的

文件夹，然后把工程打包成deb并安装到iOS中，此时“layout”中的所有文件会被解包到iOS文件系统的相同位置（这里的“layout”相当于iOS中的根目录“/”），这极大扩充了deb包的作用范围。下面用一个小示例佐以说明。

回到刚才的iOSREProject中，在Terminal中输入“make clean”及“rm*.deb”，将工程恢复到最初的状态，如下：

```
snakeninnysiMac:iosreproject snakeninny$ make clean
rm -rf ./obj
rm -rf "/Users/snakeninny/Code/iosreproject/_"
snakeninnysiMac:iosreproject snakeninny$ rm *.deb
snakeninnysiMac:iosreproject snakeninny$ ls -l
total 32
-rw-r--r--  1 snakeninny  staff  262 Dec  3 09:20
Makefile
-rw-r--r--  1 snakeninny  staff    0 Dec  3 11:28
Tweak.xm
-rw-r--r--  1 snakeninny  staff  223 Dec  3 09:05 control
-rw-r--r--@ 1 snakeninny  staff  175 Dec  3 09:48
iOSREProject.plist
lrwxr-xr-x  1 snakeninny  staff   11 Dec  3 09:05 theos -
> /opt/theos
```

然后生成一个空的“layout”目录，如下：

```
snakeninnysiMac:iosreproject snakeninny$ mkdir layout
```

并在“layout”下随便放一些空文件，如下：

```
snakeninnysiMac:iosreproject snakeninny$ touch
./layout/1.test
snakeninnysiMac:iosreproject snakeninny$ mkdir
./layout/Developer
snakeninnysiMac:iosreproject snakeninny$ touch
./layout/Developer/2.test
snakeninnysiMac:iosreproject snakeninny$ mkdir -p
./layout/var/mobile/Library/Preferences
snakeninnysiMac:iosreproject snakeninny$ touch
./layout/var/mobile/Library/Preferences/3.test
```

最后用“make package”打包，并将生成的deb文件拷贝到iOS中，用iFile安装。然后在Cydia中查看iOSREProject的文件系统，如图3-13所示。

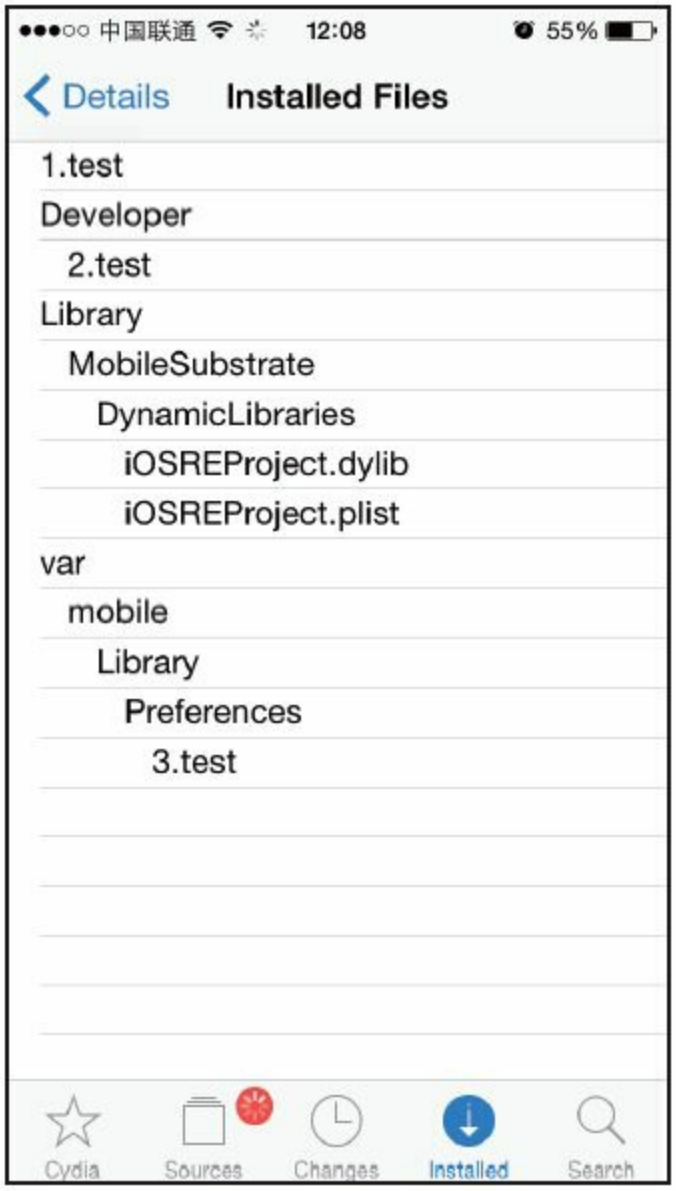


图3-13　iOSREProject文件系统

除“DEBIAN”以外的所有文件都被解包到了iOS文件系统的相同位置，本来不存在的中间文件夹也被自动创建。deb包的玄机还有很多，这里也只是管中窥豹，更全面的介绍请移

步http://www.debian.org/doc/debian-policy，官方文档总是最好的学习资料。

（3）安装

最后，要把这个deb文件安装到iOS中去。安装的方法多种多样，这里介绍两种最具代表性的：图形界面安装法和命令行安装法。大多数人的第一直觉是图形界面一定比命令行简单，那好，咱们先介绍图形界面安装法。

· 图形界面安装法

这个方法确实简单：通过iFunBox等软件把deb拖到iOS里去，然后用iFile安装它，最后重启iOS。虽然全过程都由图形界面操作，但人机交互太多，又要动电脑又要滑手机，一来二去非常繁琐，并不适用于tweak开发。

· 命令行安装法

这个方法要用到简单的ssh命令，故而要求越狱的iOS安装了OpenSSH，如果对这部分知识不了解，请先快速浏览一遍第4章的“OpenSSH”部分。下面具体介绍安装法。

首先，需要在Makefile的最上一行加上本机IP地址，如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
```

然后调用“make package install”命令完成编译打包安装一条龙服务，如下：

```
snakeninnysiMac:iosreproject snakeninny$ make package
install
Making all for tweak iOSREProject...
 Preprocessing Tweak.xm...
 Compiling Tweak.xm...
 Linking tweak iOSREProject...
 Stripping iOSREProject...
 Signing iOSREProject...
Making stage for tweak iOSREProject...
dm.pl: building package `com.iosre.iosreproject:iphoneos-
arm' in `./com.iosre.iosreproject_0.0.1-15_iphoneos-
arm.deb'
install.exec "cat > /tmp/_theos_install.deb; dpkg -i
/tmp/_theos_install.deb && rm /tmp/_theos_install.deb" <
"./com.iosre.iosreproject_0.0.1-15_iphoneos-arm.deb"
```

```
root@iOSIP's password:
Selecting previously deselected package
com.iosre.iosreproject.
(Reading database ... 2864 files and directories
currently installed.)
Unpacking com.iosre.iosreproject (from
/tmp/_theos_install.deb) ...
Setting up com.iosre.iosreproject (0.0.1-15) ...
install.exec "killall -9 SpringBoard"
root@iOSIP's password:
```

从以上信息可以看到，Theos在整个安装过程中要求我们输入两次root密码。虽然多次输入密码给人很安全的感觉，但实在是太麻烦了。好在通过设置iOS的authorized_keys可以省略SSH输密码的步骤，让“make package install”真正地从“一只多脚虫”变成“一条飞天龙”，具体步骤如下：

1）删除“/Users/snakeninny/.ssh/known_hosts”中iOSIP对应的条目。

假设iOS的IP地址是iOSIP。编辑“/Users/snakeninny/.ssh/known_hosts”，找到iOSIP所在的那一行，如下：

```
iOSIP ssh-rsa
```

完整删掉这一行。

2）生成authorized_keys。

在Terminal中执行如下命令：

```
snakeninnysiMac:~ snakeninny$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key
(/Users/snakeninny/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
/Users/snakeninny/.ssh/id_rsa.
Your public key has been saved in
/Users/snakeninny/.ssh/id_rsa.pub.
……
snakeninnysiMac:~ snakeninny$ cp
/Users/snakeninny/.ssh/id_rsa.pub ~/authorized_keys
```

就会在用户目录下生成authorized_keys。

3）配置iOS。

在Terminal中执行如下命令：

```
FunMaker-5:~ root# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key
```

```
(/var/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
/var/root/.ssh/id_rsa.
Your public key has been saved in
/var/root/.ssh/id_rsa.pub.
……
FunMaker-5:~ root# logout
Connection to iOSIP closed.
snakeninnysiMac:iosreproject snakeninny$ scp
~/authorized_keys root@iOSIP:/var/root/.ssh
The authenticity of host 'iOSIP (iOSIP)' can't be
established.
RSA key fingerprint is
75:98:9a:05:a3:27:2d:23:08:d3:ee:f4:d1:28:ba:1a.
Are you sure you want to continue connecting (yes/no)?
yes
Warning: Permanently added 'iOSIP' (RSA) to the list of
known hosts.
root@iOSIP's password:
authorized_keys                               100%  408
0.4KB/s   00:00
```

重新使用ssh命令进入iOS试试看，还需要输密码吗？此时，“make package install”真正变成了一次配置，一键安装，一劳永逸！

（4）清理

Theos提供了方便的工程清理命令“make clean”，其实际作用就是依次执行“rm-rf./obj”和“rm-rf'/Users/snakeninny/Code/iosre/_'"”两个命令，从而删

除“make”和“make package”命令生成的文件夹。也可以用“rm*.deb”，删除“make package”命令生成的所有deb文件。

3.2.4 Theos开发tweak示例

前几节完整地介绍了Theos的安装和使用方法，虽然还没有涵盖Theos的所有功能，但对于逆向工程初学者来说已经完全够用了。讲了这么多内容却还没有涉及一行真实的代码，是不是有些意犹未尽啊？

接下来将以一个最简单的tweak为例来进行讲解。安装了该程序之后，每次重启SpringBoard都将会弹出一个UIAlertView。

1.用Theos新建tweak工程“iOSREGreetings”

新建iOSREGreetings工程的命令如下：

```
snakeninnysiMac:Code snakeninny$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
```

```
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): iOSREGreetings
Package Name [com.yourcompany.iosregreetings]:
com.iosre.iosregreetings
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.apple.springboard
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]:
Instantiating iphone/tweak in iosregreetings/...
Done.
```

2.编辑Tweak.xm

编辑后的Tweak.xm内容如下：

```
%hook SpringBoard
- (void)applicationDidFinishLaunching:(id)application
{
    %orig;
    UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Come to http://bbs.iosre.com for more
fun!" message:nil delegate:self cancelButtonTitle:@"OK"
otherButtonTitles:nil];
    [alert show];
    [alert release];
}
%end
```

3.编辑Makefile及control

编辑后的Makefile内容如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
TWEAK_NAME = iOSREGreetings
iOSREGreetings_FILES = Tweak.xm
iOSREGreetings_FRAMEWORKS = UIKit
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
	install.exec "killall -9 SpringBoard"
```

编辑后的control内容如下：

```
Package: com.iosre.iosregreetings
Name: iOSREGreetings
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Greetings from iOSRE!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

以上代码非常简单，当SpringBoard的applicationDidFinishLaunching:函数得到调用时，代表SpringBoard的启动过程已经结束。钩住（hook）这个

函数，调用%orig完成它的原始操作，然后弹出一个自定义的UIAlertView；这样一来，每次重启SpringBoard都会弹出一个对话框。你看懂了吗？

准备就绪，在Terminal中敲入“make package install”，待SpringBoard重启之后会看到如图3-14所示的结果，简单粗暴。

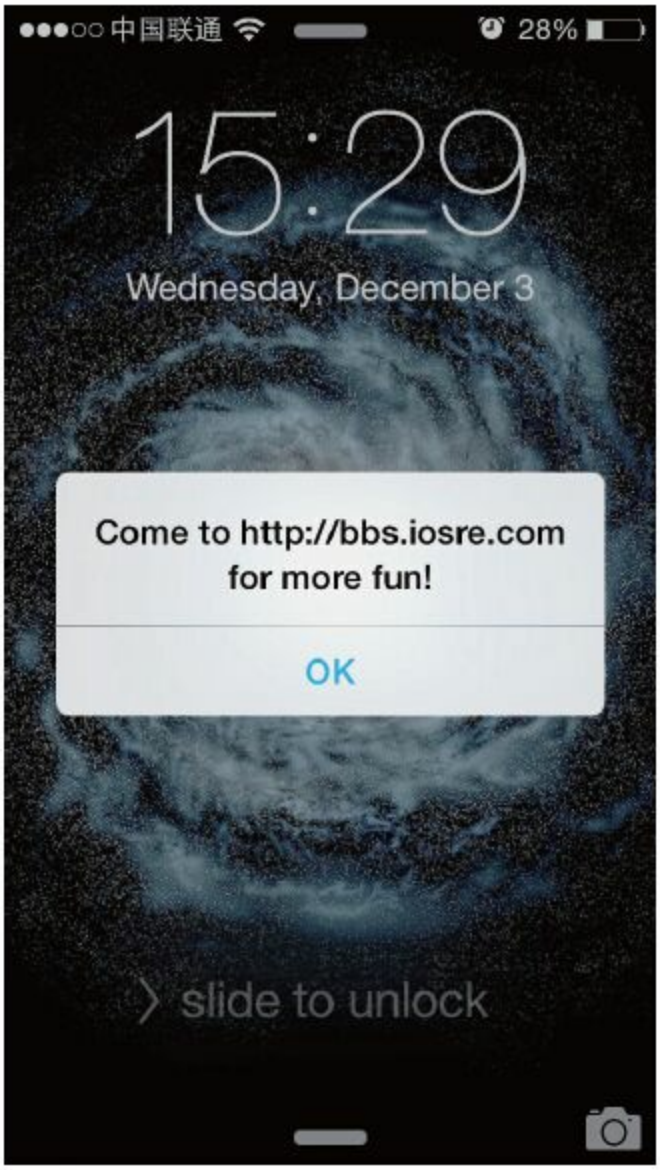


图3-14　第一个tweak

是的，仅仅是这样一些小小的改动，就已经可以改变App的行为了。此时，封闭的iOS已经向我们打开了大门……

因为有Theos这样的开发工具存在，修改闭源的

iOS程序变得前所未有的方便。不过在前面也提到了，现在的App工程量越来越大，class-dump头文件也越来越多，要从浩如烟海的函数名中筛选出我们感兴趣的目标，比确定目标后编写代码还要难得多。面对成千上万行代码，如果没有其他工具辅助分析，逆向工程简直是一场噩梦，让人一筹莫展。那么接下来，就轮到这些辅助分析工具隆重登场了。

3.3 Reveal

Reveal是由ITTY BITTY出品的UI分析工具，可以直观地查看App的UI布局，如图3-15所示。

官方给Reveal的定位是“See your application’s view hierarchy at runtime with advanced 2D and 3D visualisations”，但作为逆向工程师，查看自己App的UI布局显然不能满足我们的需求，能够查看别人App的UI布局才是正经事儿——图3-16就是用Reveal查看AppStore的效果。

图3-15　Reveal

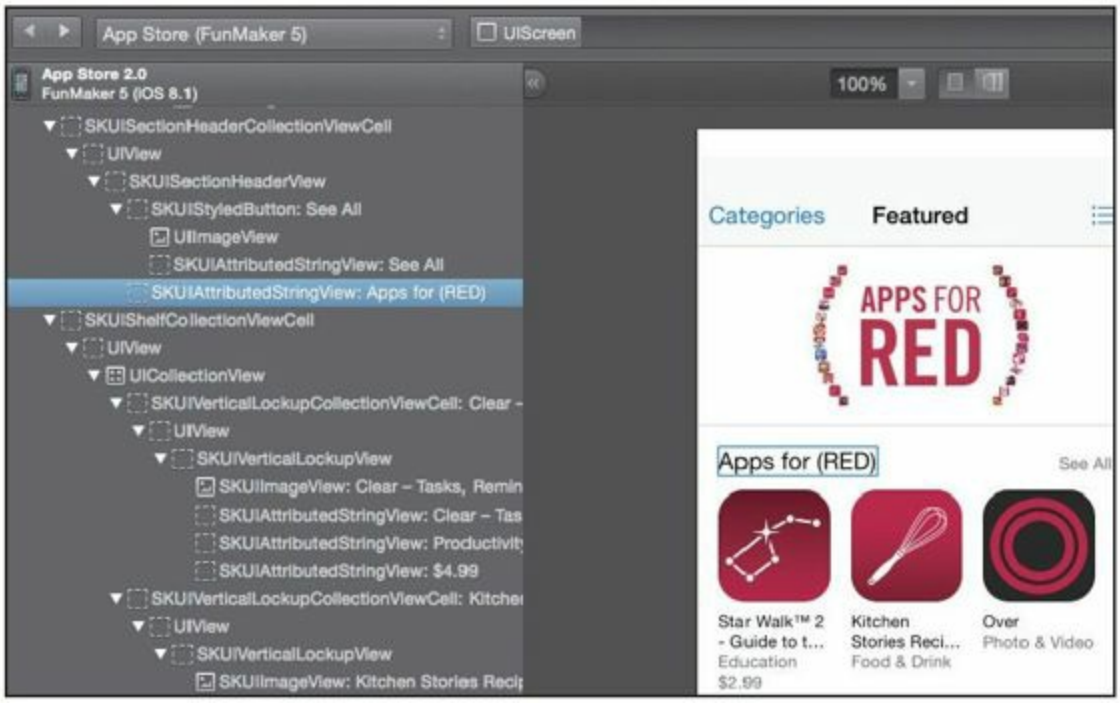


图3-16　用Reveal查看AppStore的UI布局

在Reveal界面的左侧，AppStore的UI布局以树形方式展现出来，当选中一个控件时，右侧的界面截图会实时地标出选中的位置。同时大家也一定注意到了，Reveal也解析出了UI控件对应的类名，如图3-16中所示的SKUIAttributedStringView。要查看别人App的UI布局，还需要对Reveal做简单配置。

1.安装Reveal Loader

在Cydia中搜索并安装Reveal Loader，如图3-17所

示。

值得注意的是，在安装Reveal Loader的时候，它会自动从Reveal的官网下载一个必须的文件libReveal.dylib。如果网络状况不太好，Reveal Loader不一定能够成功下载这个dylib文件，而且它没有针对dylib下载失败的情况做容错处理，可能会在下载界面卡顿很长时间，导致Cydia停止响应。因此，在下载它之前最好连接美国VPN，且在下载完Reveal Loader后，检查iOS上的“/Library/”目录下有没有一个名为“RHRevealLoader”的文件夹，如下：

```
FunMaker-5:~ root# ls -l /Library/ | grep RHRevealLoader
drwxr-xr-x  2 root   admin  102 Dec  6 11:10
RHRevealLoader
```

图3-17　Reveal Loader

如果没有，就手动创建一个，如下：

```
FunMaker-5:~ root# mkdir /Library/RHRevealLoader
```

然后打开Reveal，在它标题栏的“Help”选项下，选中其中的“Show Reveal Library in Finder”子选项，如图3-18所示。

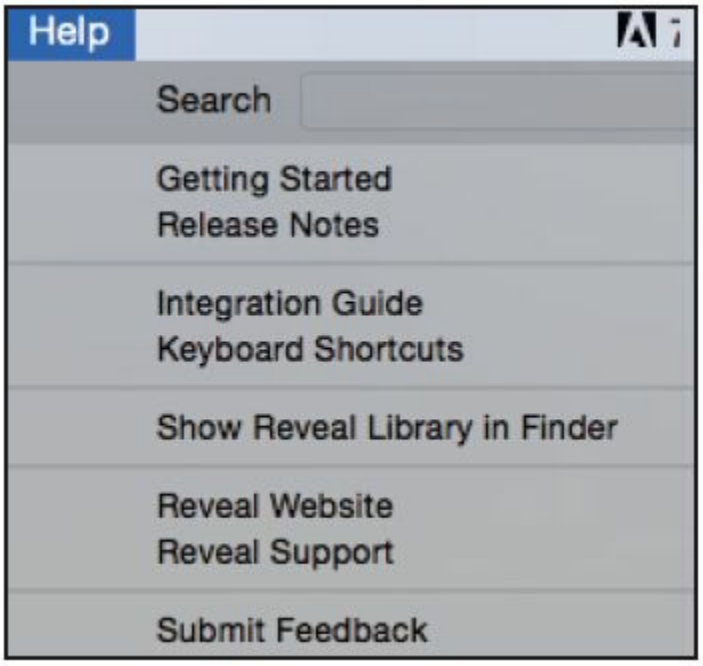


图3-18　Show Reveal Library in Finder

此时就会出现图3-19所示的界面。

把这个libReveal.dylib通过scp或iFunBox等方式拷贝到刚才创建的RHRevealLoader目录下，如下：

```
FunMaker-5:~ root# ls -l /Library/RHRevealLoader
total 3836
-rw-r--r-- 1 root admin 3927408 Dec  6 11:10
libReveal.dylib
```

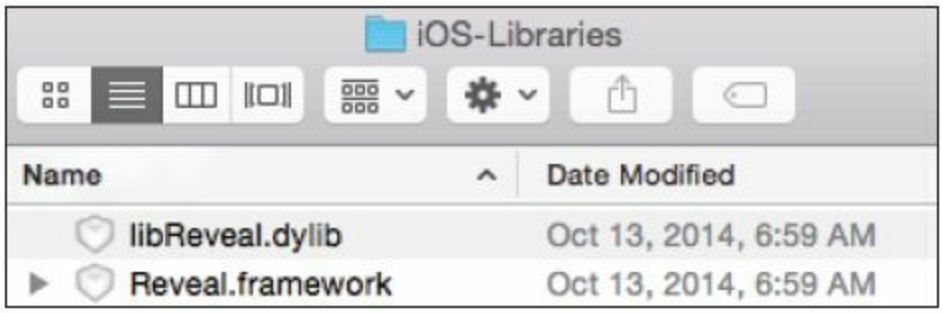


图3-19　libReveal.dylib

至此完成Reveal Loader的安装。

2.配置Reveal Loader

Reveal Loader的配置界面位于Settings应用中，它的名字叫“Reveal”，如图3-20所示。

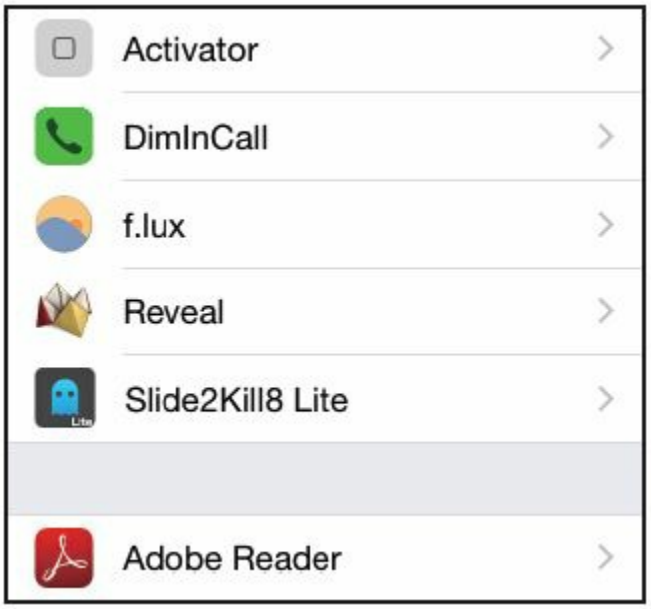


图3-20　Reveal Loader

点击“Reveal”进入其界面，呈现在我们面前的主要是一些使用声明，如图3-21所示。

点击“Enabled Applications”，进入配置界面。要分析哪个App，就打开对应的开关。这里打开了AppStore和Calculator的开关，如图3-22所示。

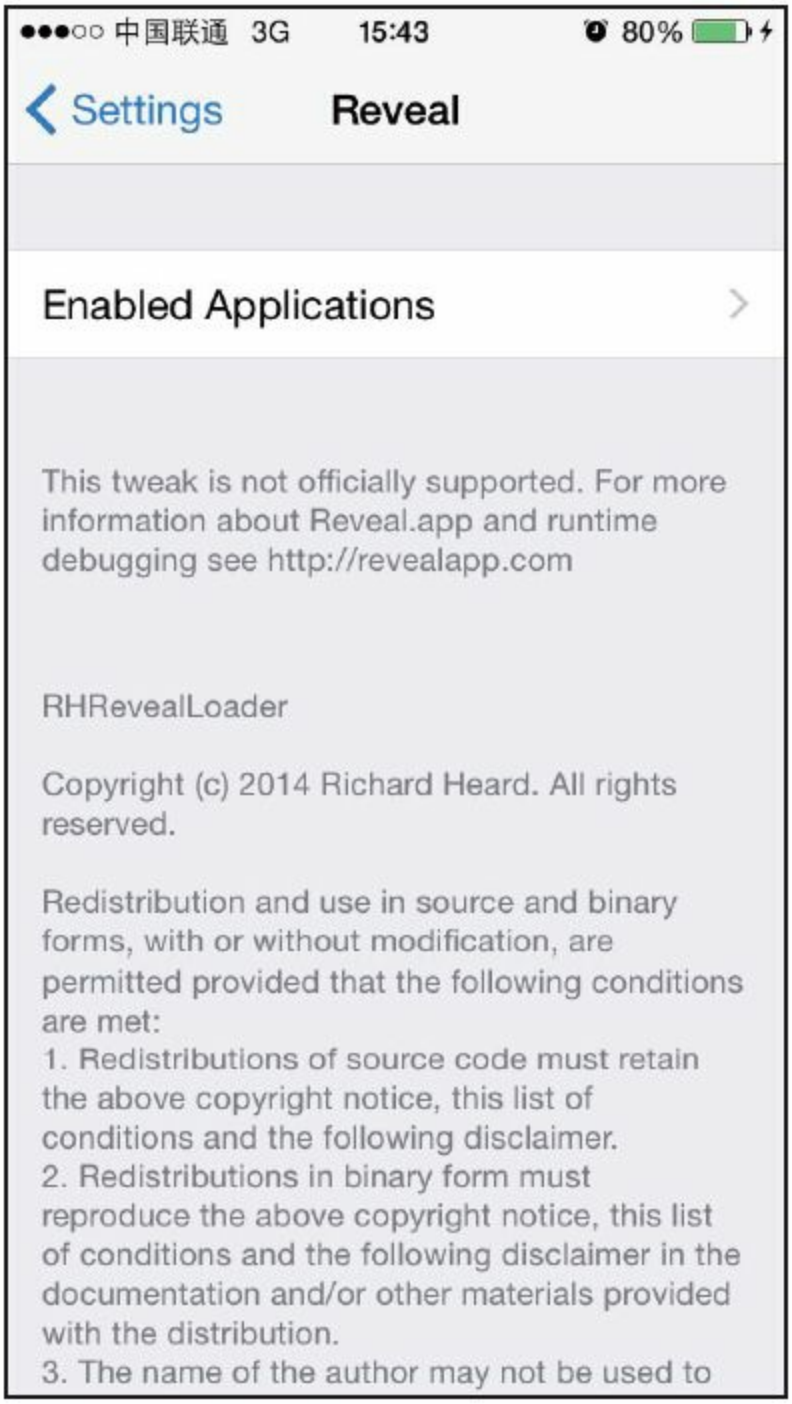


图3-21　Reveal Loader使用声明

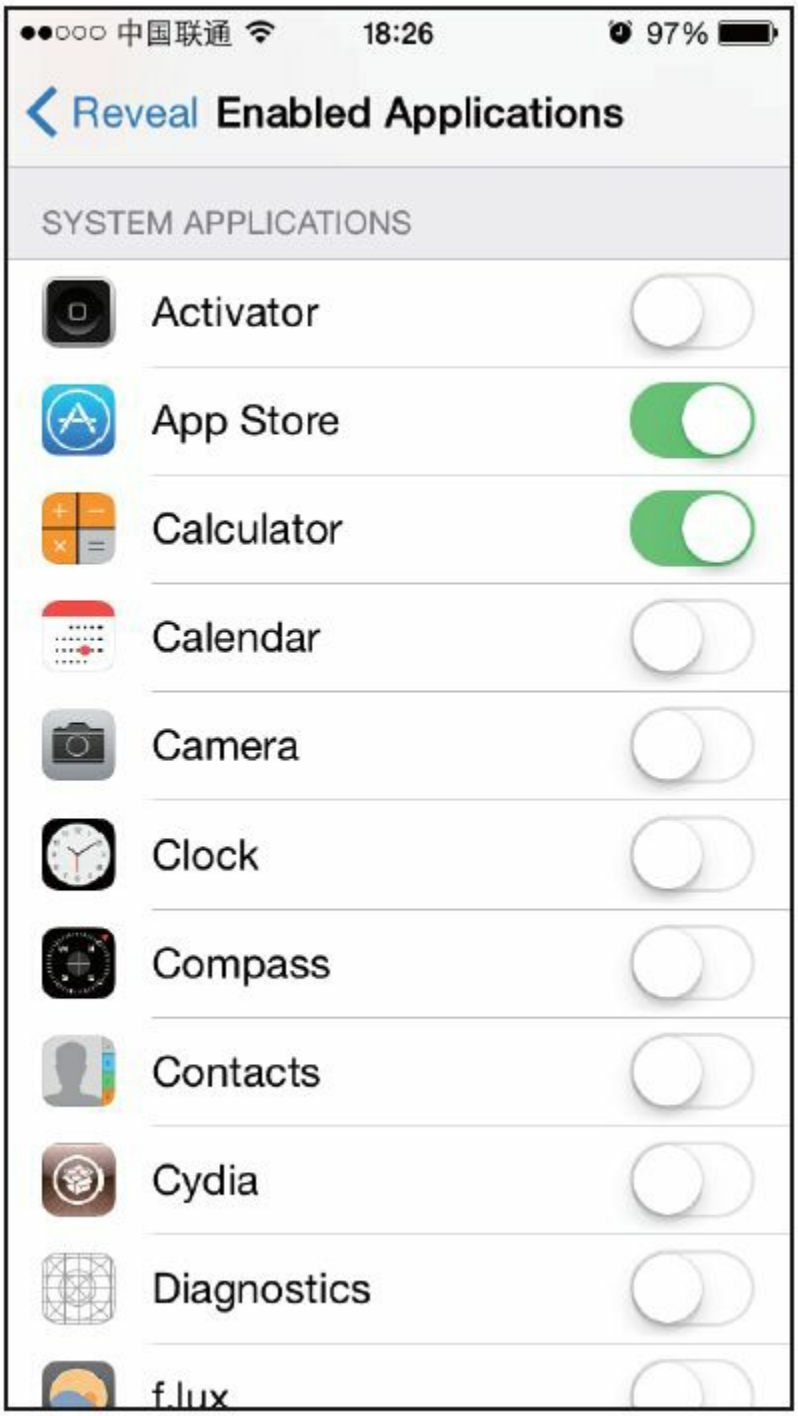


图3-22　配置Reveal Loader

Reveal Loader的配置就是这样了。既直观又方便，不是吗？

3.使用Reveal查看目标App的UI布局

一切准备就绪，轮到主角Reveal出场了。首先确认OSX和iOS位于同一网段内，然后启动Reveal，并重启iOS上的目标App（即如果App开着，需要先关掉，再打开）。从Reveal界面左上角选择目标App，稍等一会儿，Reveal就会把目标App的UI布局展现在我们面前，如图3-23所示。

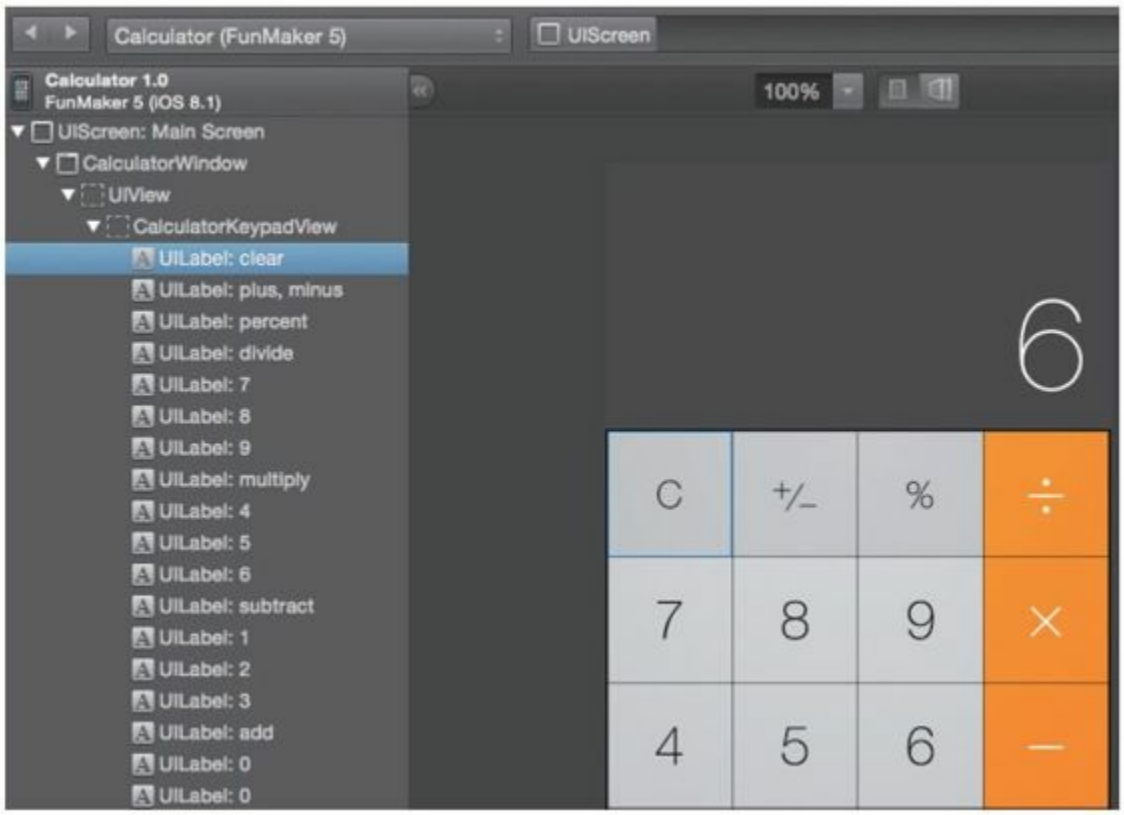


图3-23　Calculator的UI布局

Reveal的使用并不复杂，它是一款用户体验不错的工具。但是在iOS逆向工程中，对App的分析往往不会只是停留在UI层，外在表象下的内在实现才是最终目标。在本书的后半部分，将采用Reveal的“文字版”，即recursiveDescription函数，配合Cycript来挖掘隐藏在UI布局下的代码，届时你就会感知iOS逆向工程的真正威力。

3.4 IDA

3.4.1 IDA简介

即使你以前没有从事过iOS逆向工程相关的工作，也一定听说过IDA（The Interactive Disassembler）的鼎鼎大名。而对于绝大多数接触过逆向工程的人来说，IDA三个字则是如雷贯耳，它乃逆向工程中最负盛名的神器之一（如图3-24所示）。如果说class-dump能够帮我们罗列出要分析的点，那IDA就能进一步帮我们把这些点铺成面。

图3-24　IDA官网

笼统地说，IDA是一个支持Windows、Linux和Mac OS X的多平台反汇编器/调试器，它的功能非常强大，以至于连官方都不能给出一个详尽的功能列表。

IDA的正式版是收费的，但其作者也是程序员出身，深知我们生活不易，所以慷慨地提供了一个免费的试用版，对于逆向工程初学者来说，已完全够用了。IDA的下载和安装十分方便，具体可参考：https://www.hex-rays.com/products/ida/index.shtml，本书不再赘述。

3.4.2 IDA使用说明

IDA启动时会短暂地显示如图3-25所示的窗口。

图3-25　IDA启动界面

这时可以点击“OK”，或等上几秒，它会自动关闭，之后就会看到IDA的主界面，如图3-26所示。

在该界面中，不用繁琐地在菜单里点击“打开文件”，然后一个目录一个目录地去翻找，只需把要分析的文件拖进IDA的灰色区域就行了。打开文件后，还需要做一些基本的配置，如图3-27所示。

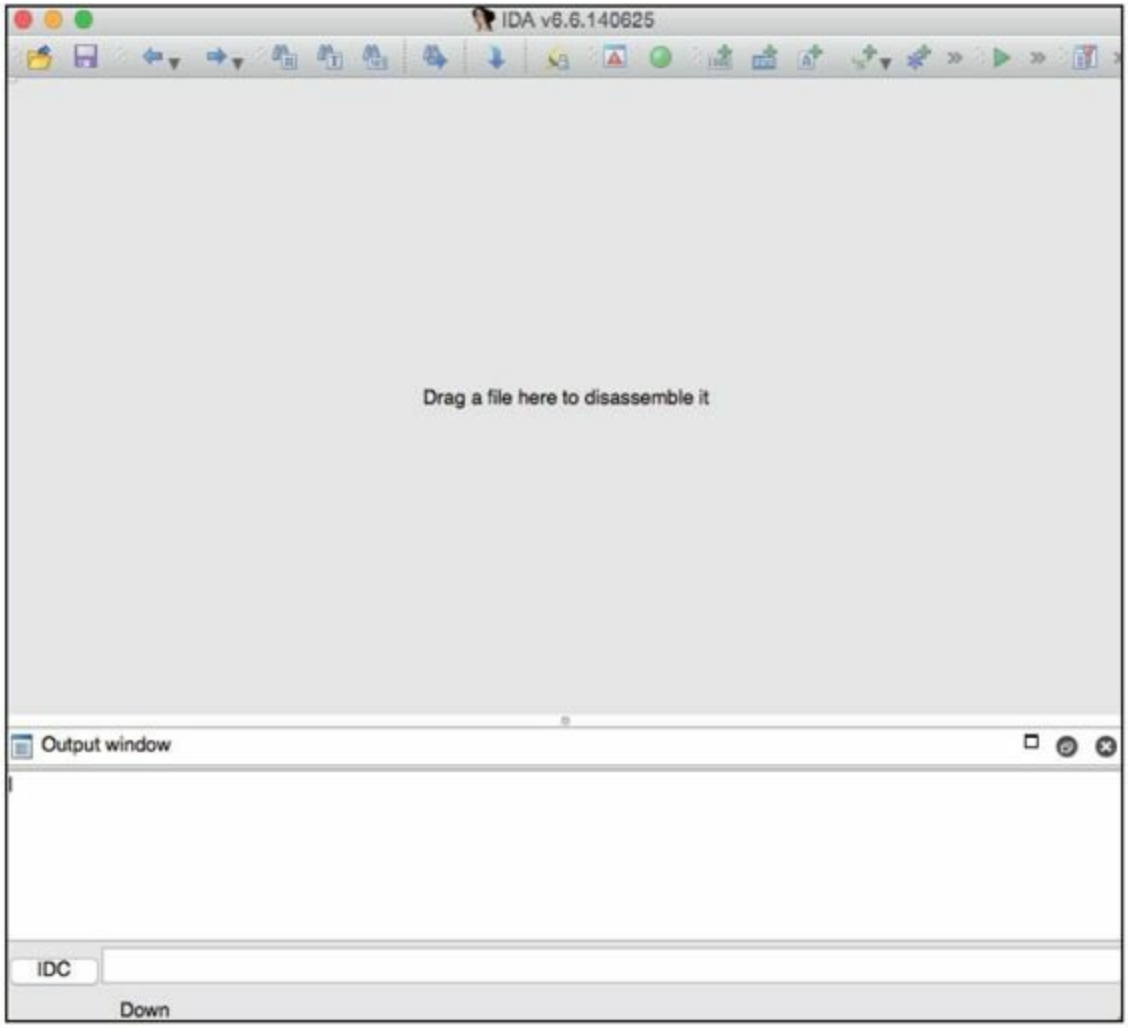

IDA v6.6.140625
Drag a file here to disassemble it
Output window
IDC
Down

图3-26　IDA主界面

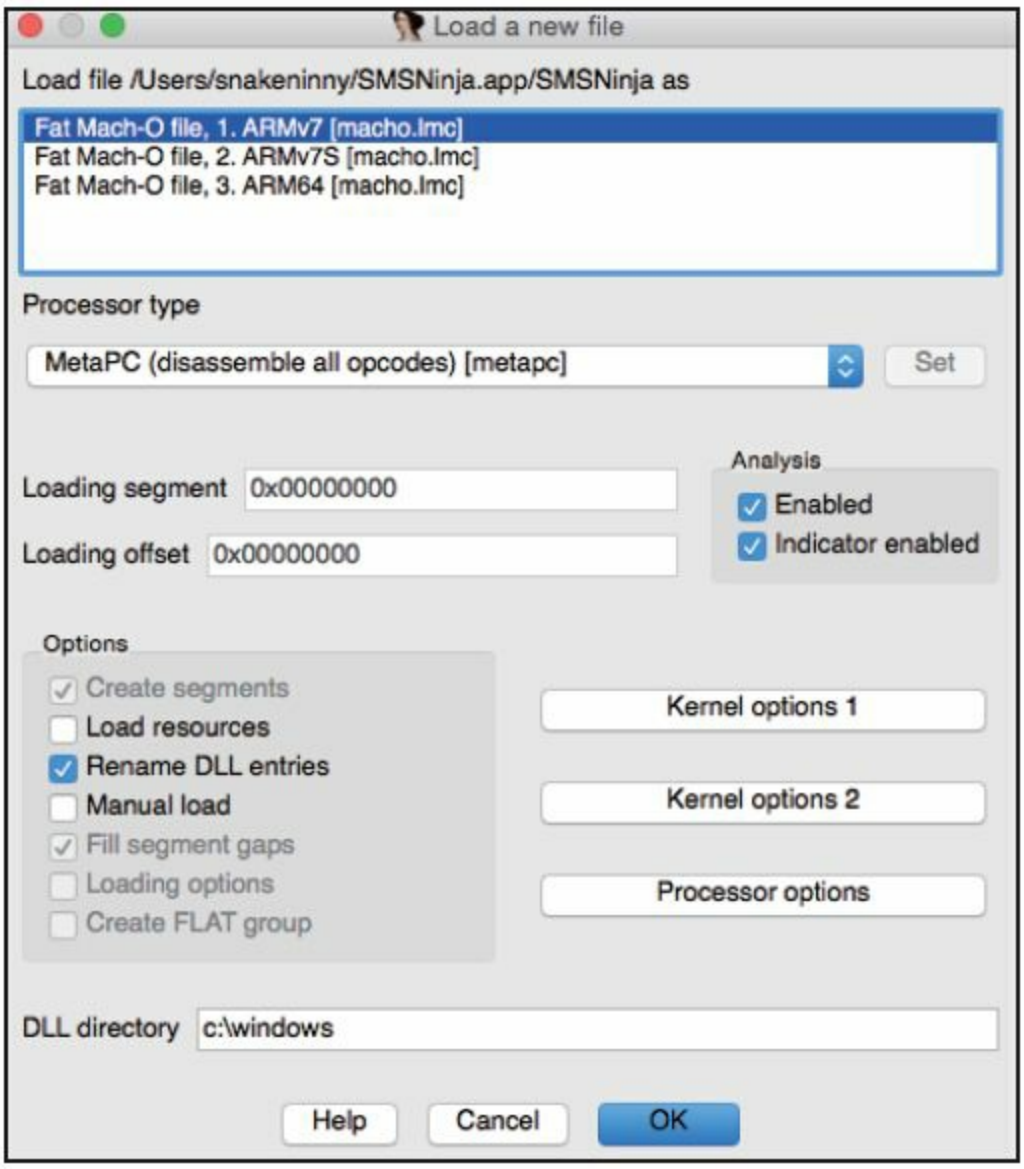


图3-27　IDA初始配置

有一个地方需要注意：对于一些fat binary（指的是为了兼容不同架构的处理器，而把多种指令集糅合到一个binary）来说，图3-27中最上面的白框内会出现多个Mach-O文件供我们选择。建议先迅速查阅第4

章“dumpdecrypted”这一节的ARM对照表，找到设备对应的ARM信息，例如笔者的iPhone 5对应的是ARMv7s。如果设备的ARM没有出现在这些选项中，就选那个向下兼容的选项，即如果选项里有ARMv7S，就选它；否则选ARMv7。这种方法应该可以应对碰到的99%的情况，如果你恰巧是那1%，请来http://bbs.iosre.com分享这种百里挑一的喜悦，我们一起解决问题。

这里笔者选择了ARMv7S，然后点击“OK”，此时，会连续弹出好几个窗口，一路点击“Yes”和“OK”就可以了，如图3-28和图3-29所示。

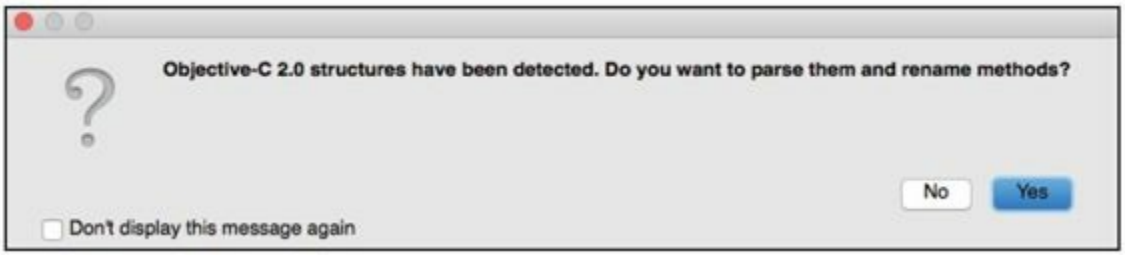


图3-28　IDA启动选项

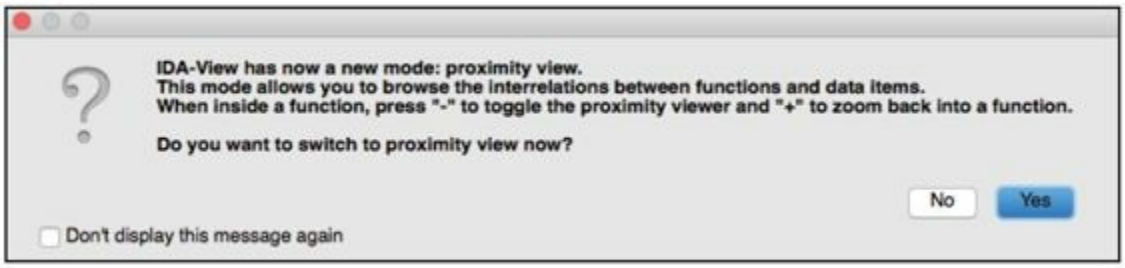


图3-29　IDA启动选项

因为试用版的IDA无法保存，所以即使在这些窗口上勾选“Don’t display this message again”，下次打开IDA还是会出现这些窗口；虽然有些麻烦，但这么强大的工具都让我们免费使用了，麻烦就麻烦点吧。

按钮都点完后，内容丰富的主界面再次进入我们眼帘，如图3-30所示。

在进入图3-30所示的界面时，你会看到上方的进度条不断滚动，下方的Output window也会打印出对文件的分析进度。当进度条的主色调变成蓝色（IDA界面上的颜色在黑白印刷页上看不出来，请谅解），Output window中显示“The initial autoanalysis has been finished.”时，表示IDA的初始分析已完成。

在初学阶段，由于主要用IDA作静态分析，基本用不上Output window，所以在开始分析之前，可以先关掉Output window。

现在看到的两个大窗口分别是左侧小部分的Functions window（如图3-31所示）和右侧大部分的Main window（如图3-32所示）。下面分别介绍一下这2个窗口的用途。

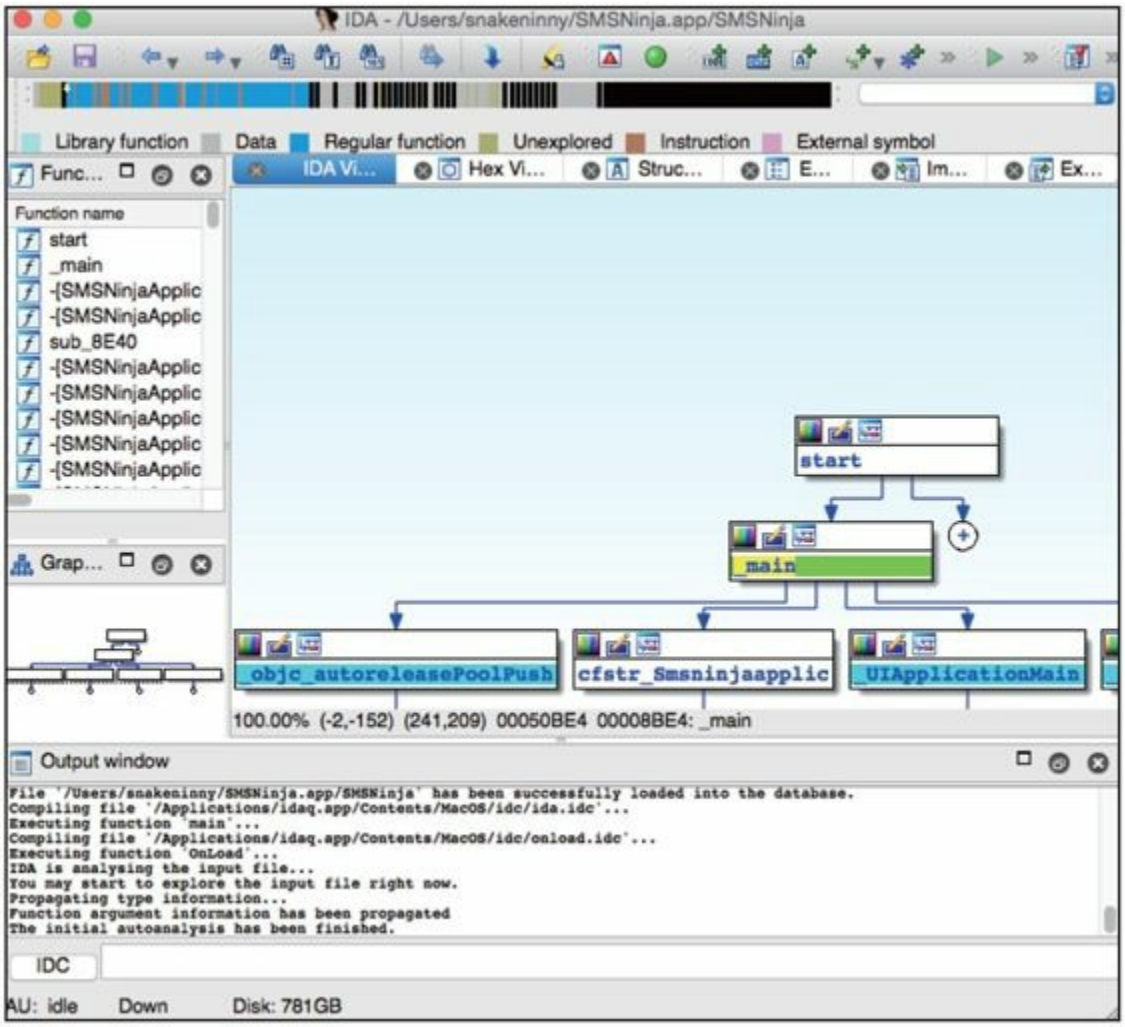


图3-30　IDA主界面

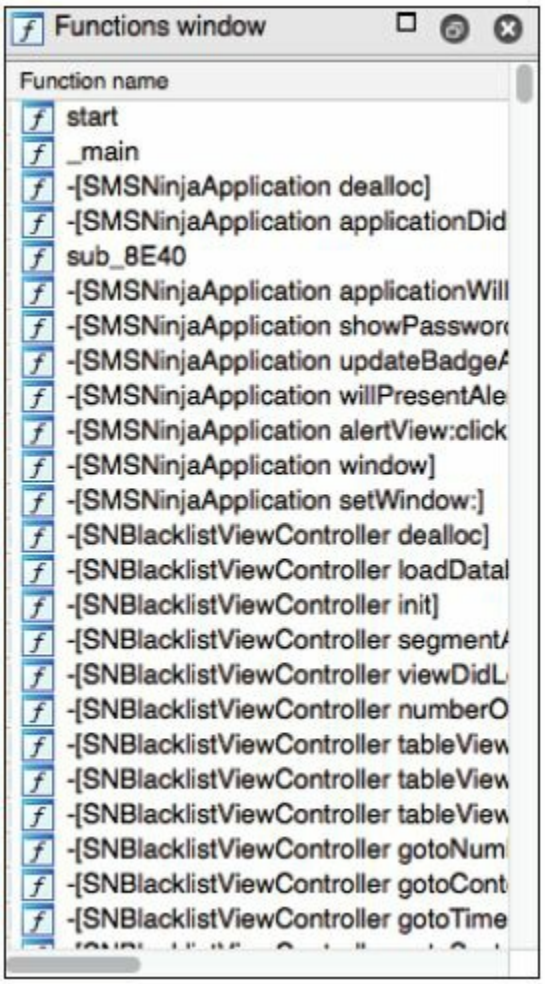


图3-31　Functions window

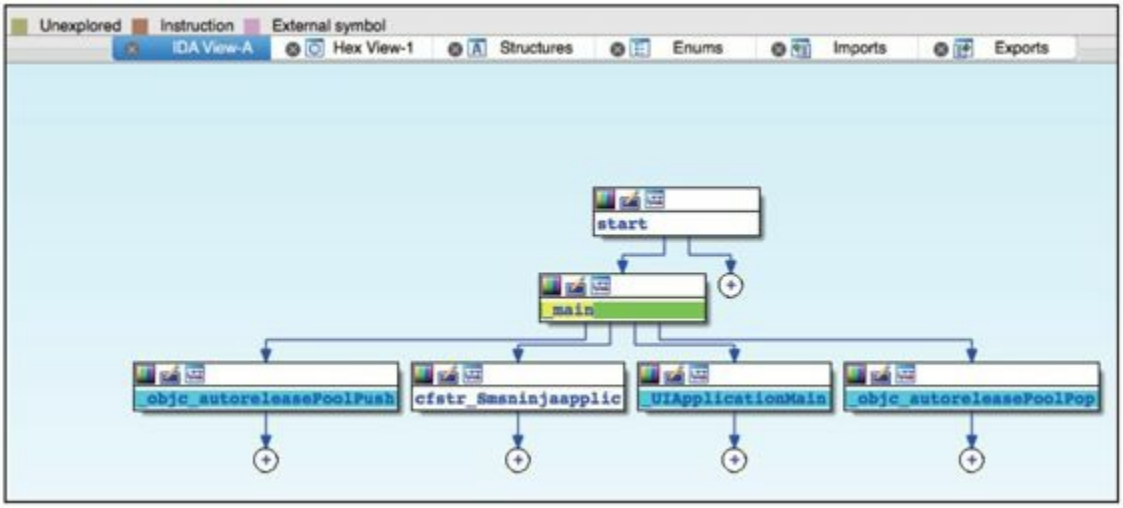


图3-32　Main window

（1）Functions window

顾名思义，这个窗口展示了IDA分析出来的所有函数（规范地讲，Objective-C的function应该称为method，即方法，此处统称为函数，请大家注意），双击一个函数，Main window会显示它的函数体。在选中Function Window时点击菜单栏上的“Search”，会弹出如图3-33所示的子菜单。

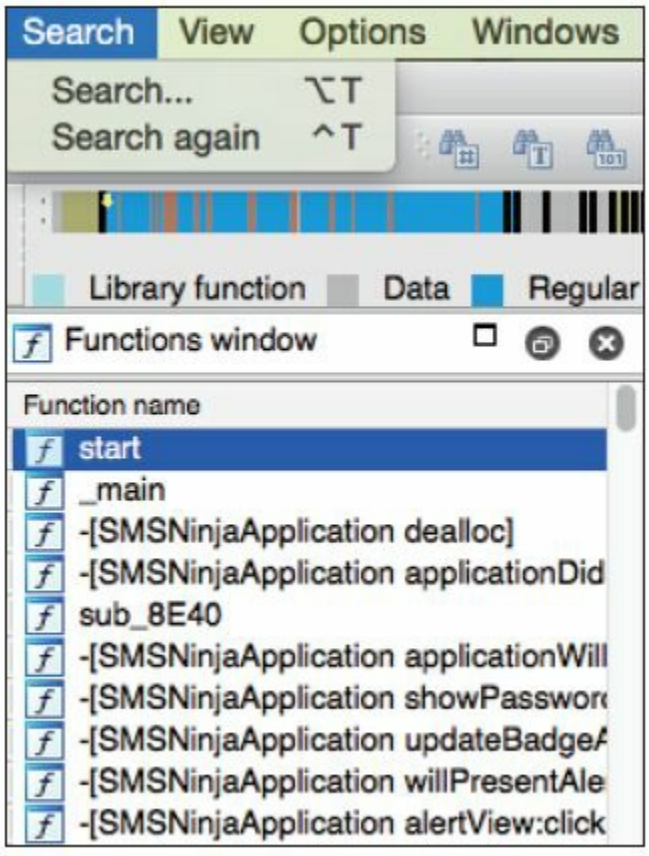


图3-33　查找函数

选择“Search”，并在图3-34的对话框中输入要查找的内容，可以在所有函数名里查找指定的字符串，十分方便；当要查找的内容出现在多个函数名里时，还可以点击“Search again”来遍历这些函数名。当然，上面的操作也都可以用IDA中显示的快捷键完成。

Functions window中的Objective-C函数与class-dump导出的内容吻合。除了Objective-C函数外，IDA还将所有subroutine罗列了出来，这是class-dump做不到的。class-dump导出的内容都是Objective-C函数名，可读性高，容易上手，是iOS逆向工程初学者的乐园；subroutine的名称只是一个代号，没有明显含义，分析难度大，大多数初学者看到这里就打了退堂鼓。但是，iOS的底层是用C和C++实现的，编译之后生成的大都是subroutine，class-dump拿它没辙，只能使用IDA这样的工具。要想深层次挖掘iOS中最有趣

的部分，掌握IDA的用法是必经之路。

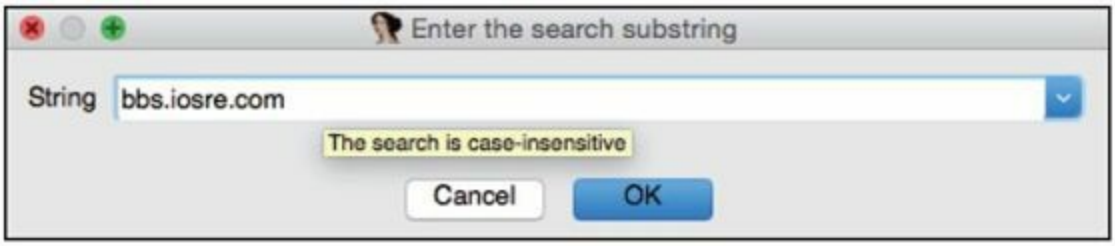
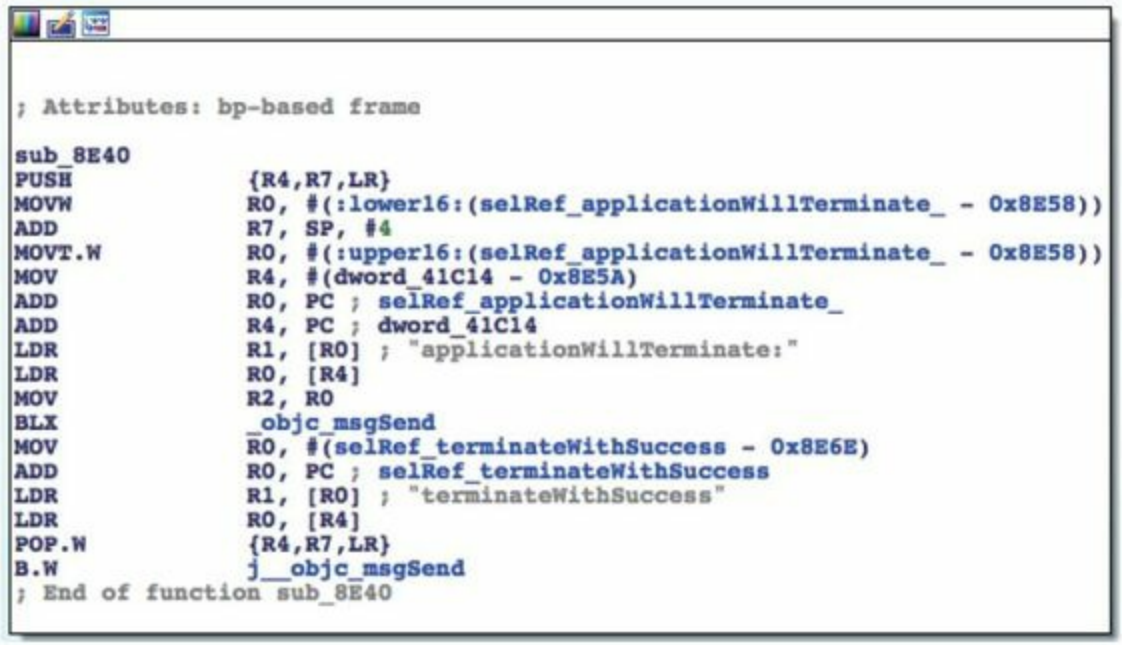


图3-34　查找函数

（2）Main window

绝大多数没有用过IDA的iOS开发者，包括笔者，在第一次看见初始分析完成后的Main window时都懵了——这都是些什么玩意儿？这是人类文字吗？还是把IDA关了，刷会儿微博压压惊吧。这跟很多工程师在写第一行代码时不知如何下手的感觉很类似。其实，跟写代码时需要定义一个入口函数一样，在逆向工程里，也需要找到自己感兴趣的入口函数。在Functions window中双击这个入口函数，使得Main window跳转到函数体，然后用鼠标选中Main window，按一下空格，界面会瞬间变清爽，可读性

一下就强了起来，让人感觉到“我的天空，星星都亮了”（如图3-35所示）。



```
; Attributes: bp-based frame

sub_8E40
PUSH            {R4,R7,LR}
MOVW            R0, #(:lower16:(selRef_applicationWillTerminate_ - 0x8E58))
ADD             R7, SP, #4
MOVT.W          R0, #(:upper16:(selRef_applicationWillTerminate_ - 0x8E58))
MOV             R4, #(dword_41C14 - 0x8E5A)
ADD             R0, PC ; selRef_applicationWillTerminate_
ADD             R4, PC ; dword_41C14
LDR             R1, [R0] ; "applicationWillTerminate:"
LDR             R0, [R4]
MOV             R2, R0
BLX             _objc_msgSend
MOV             R0, #(selRef_terminateWithSuccess - 0x8E6E)
ADD             R0, PC ; selRef_terminateWithSuccess
LDR             R1, [R0] ; "terminateWithSuccess"
LDR             R0, [R4]
POP.W           {R4,R7,LR}
B.W             j__objc_msgSend
; End of function sub_8E40
```

图3-35　Graph view

Main window有两种显示模式，分别是Graph view和Text view，它们之间可以通过空格键切换。Graph view把被分析的程序逻辑用方块的形式表现出来，方便我们分析程序各个分支之间的关系。各个方块之间的执行顺序用带箭头的线表示，当执行出现分支时，满足判断条件分支的线是绿色的，否则是红色的；当执行没有分支时，线是蓝色的。比如，在图3-36中，

当最上面的方块执行完后，会判断R0是否为0，如果R0!=0，则满足判断条件（BNE，即Branch if Not Equal to zero），走绿色的那条路接着执行右边的方块；否则走红色的路执行左边的方块。此处是IDA的难点之一，后面的实例中会再次讲解。

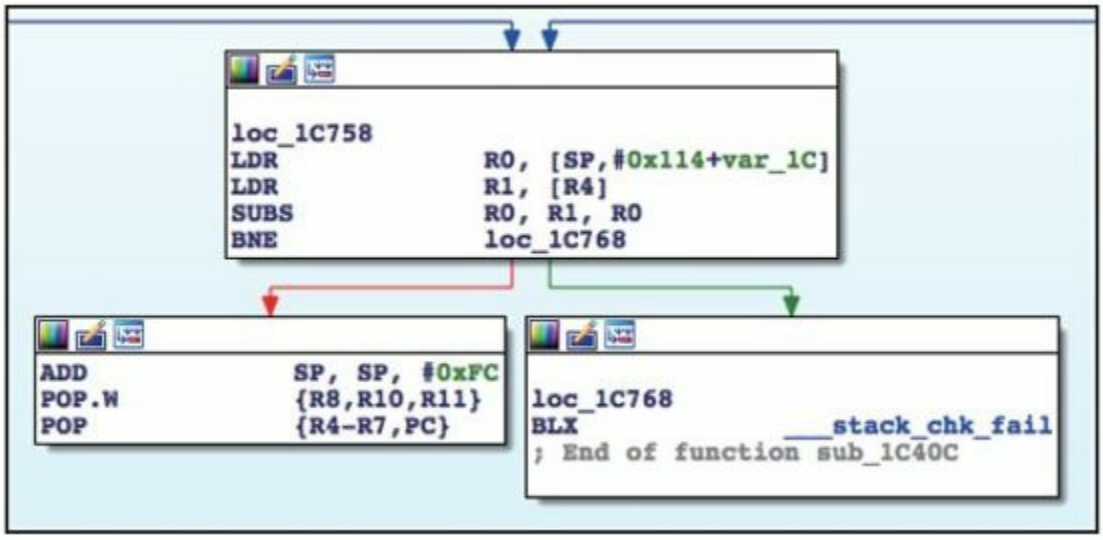


图3-36　IDA的分支

细心的读者也一定注意到了，IDA中的字体色彩缤纷，事实上，不同颜色的字体表示的含义也各不相同，如图3-37所示。

图3-37　颜色指示栏

当选中一个符号时，相同的符号都会用黄色高亮显示，方便跟踪这个符号的轨迹，如图3-38所示。

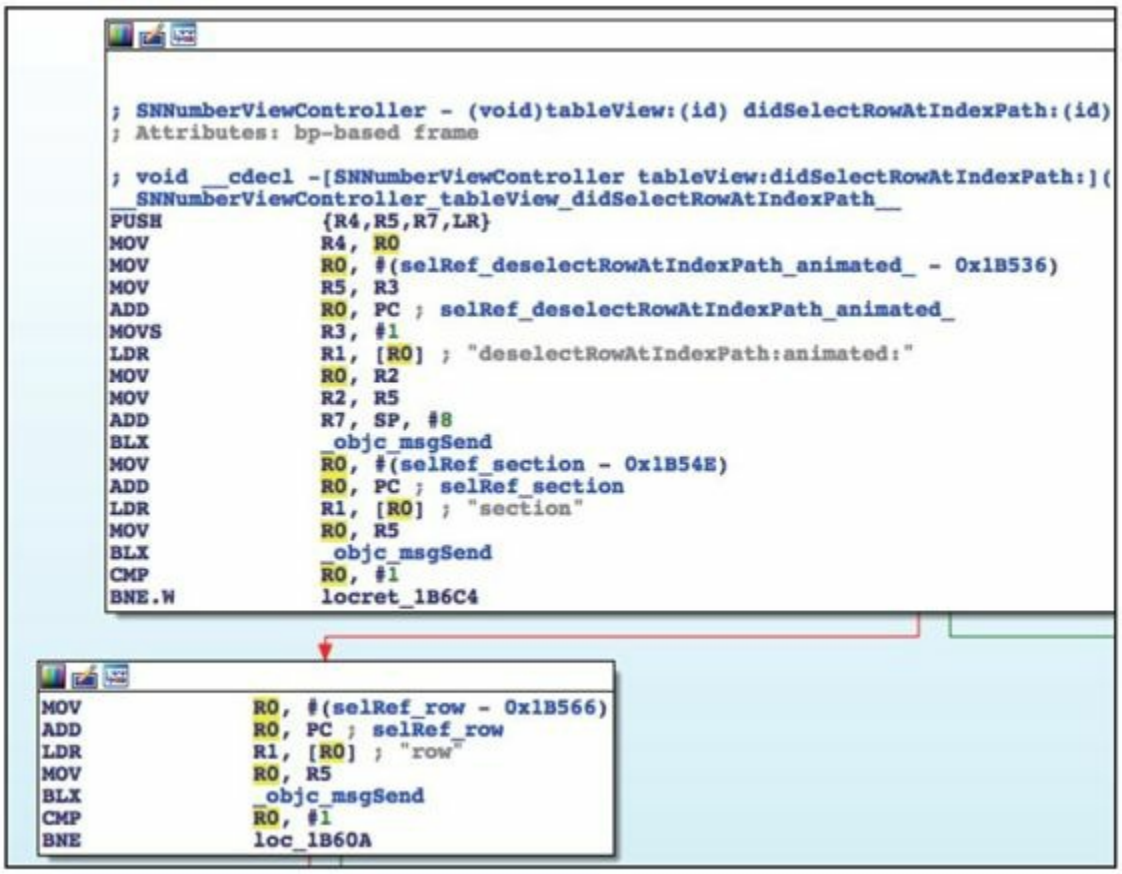


图3-38　符号高亮

双击符号，可以查看它的实现，效果与图3-35类似。在任意符号上点击鼠标右键，会弹出如图3-39所示的菜单。

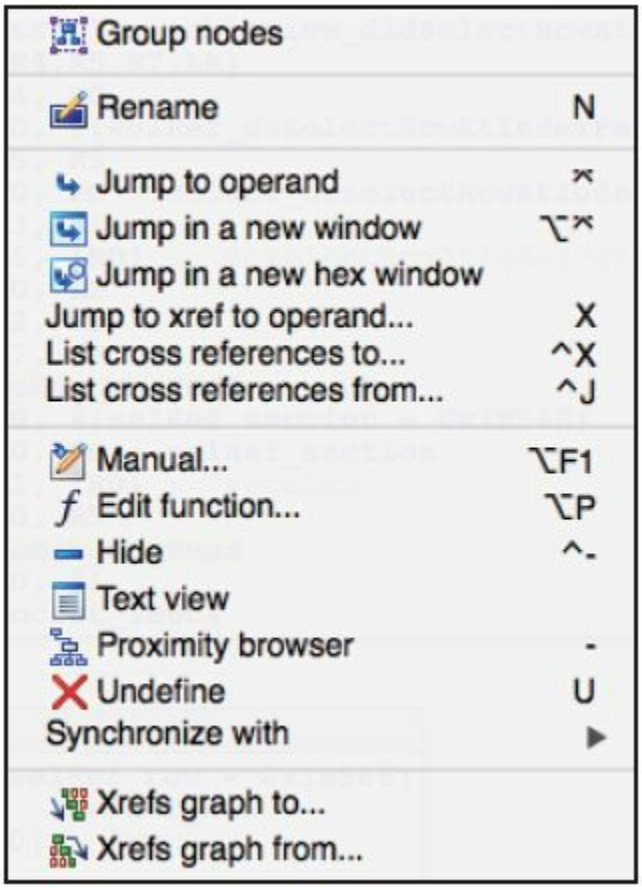


图3-39　在符号上点击右键

其中，常用的功能有“Jump to xref to operand...”（快捷键“x”），点击后出现的窗口罗列了这个文件中显式引用这个符号的所有信息，如图3-40所示。

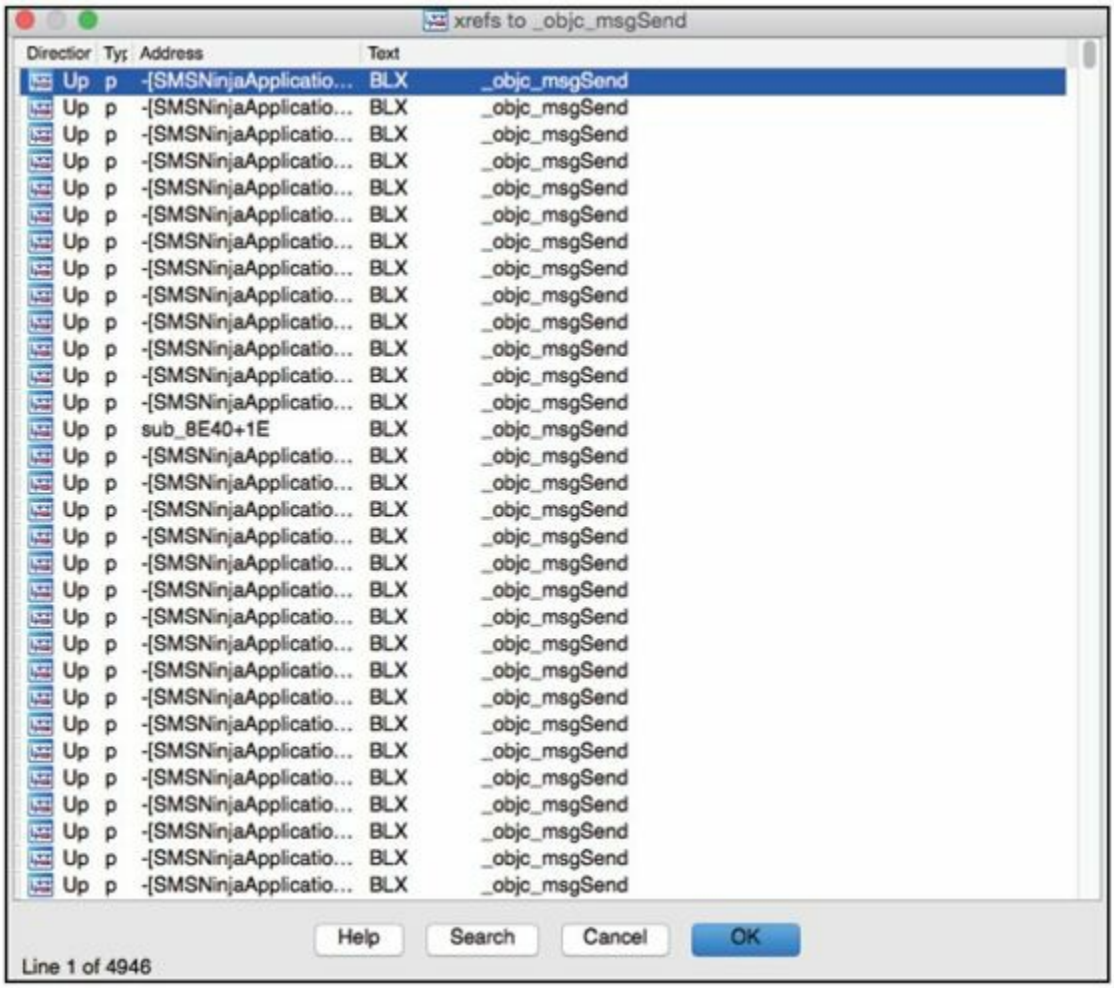


图3-40　Jump to xref to operand...

如果觉得这种表示方法不够直观，更喜欢Main window里的Graph view形式，可以选择菜单下面的“Xrefs graph to...”，但如果运气不好，这个符号被引用过多的话，就会看到类似图3-41这样的一团乱麻的界面，真是剪不断，理还乱。

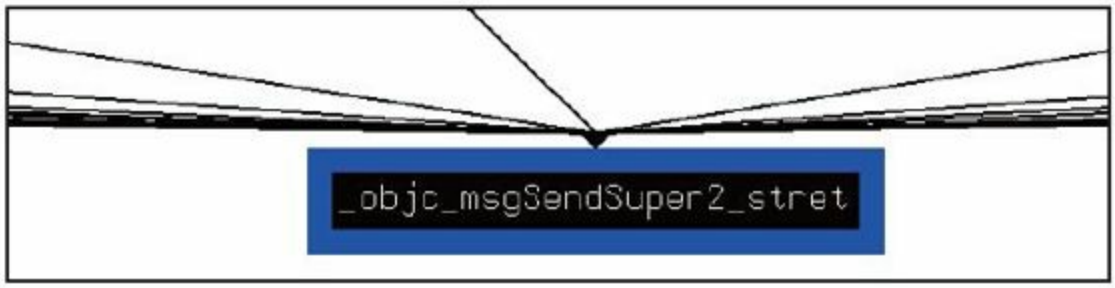


图3-41　符号引用的Graph view

在图3-41中，黑色的部分都是由一根根直线构成的，两侧基本已经黑成了一片，可以看出_objc_msgSendSuper2_stret这个符号被大量引用。

相对地，“Xrefs graph from...”则会显示这个符号显式引用的所有符号，如图3-42所示。

从图3-42可以看到，sub_1DC1C是个subroutine，它显式引用了j__objc_msgSend、_OBJC_CLASS_$_UIApplication和_objc_msgSend，而_objc_msgSend又显式引用了__imp__objc_msgSend。双击Main window里的_objc_msgSend，再双击__imp__objc_msgSend，可以看到它来自libobjc.A.dylib，如图3-43所示。

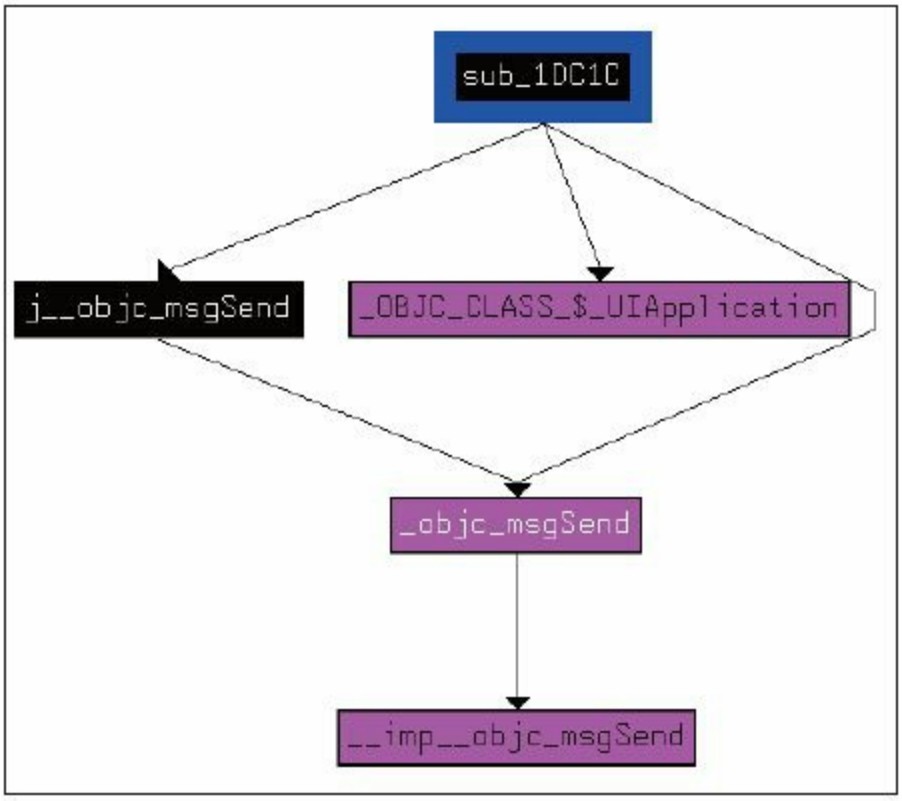


图3-42　查看sub_1DC1C显式引用的符号

在多数情况下，找到一个感兴趣的符号时，会想进一步查找与这个符号相关的所有线索。一种笨拙但有效的方法是鼠标选中Main Window时点击菜单栏上的“Search”，此时会弹出如图3-44所示的子菜单。

然后选择“text...”，会弹出如图3-45所示的窗口。

图3-43　查看外部符号来源

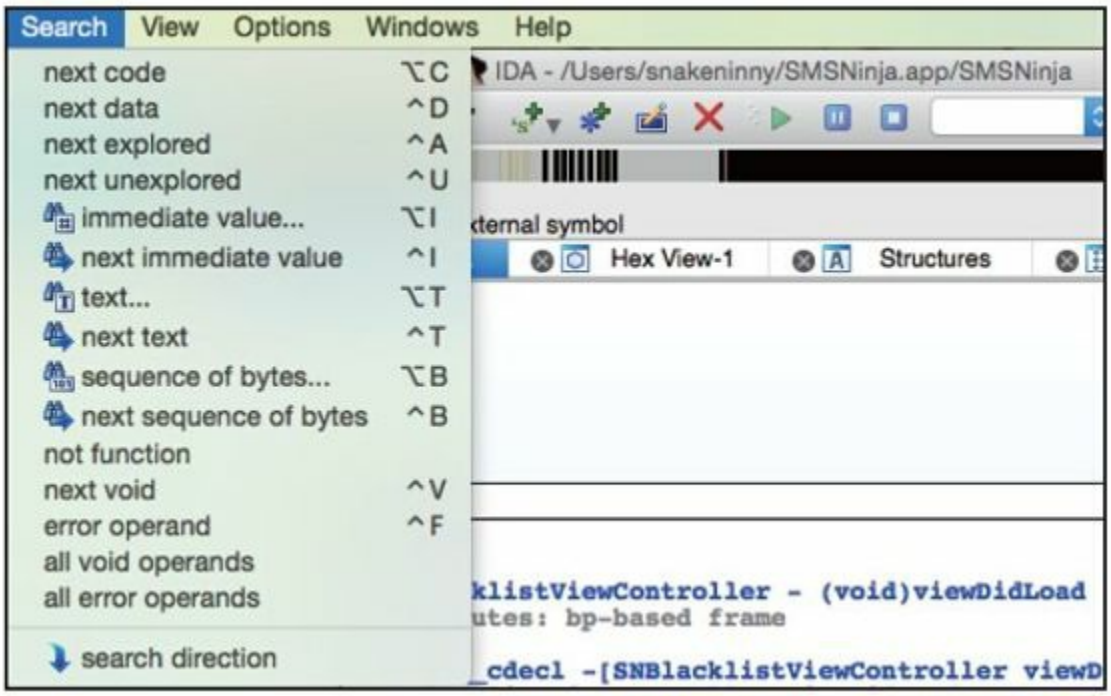


图3-44　在Main window里查找

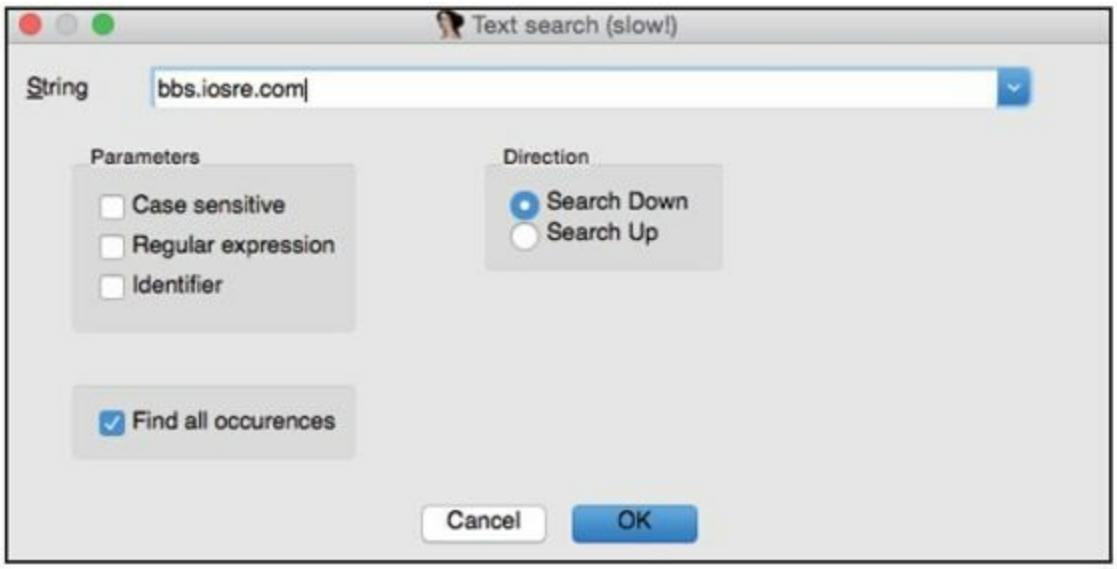


图3-45　搜索文本

这时，可以根据自己的情况选择搜索是否对大小写敏感，搜索格式是不是正则表达式等，然后勾选“Find all occurences”，点击“OK”，IDA会将文件中所有满足搜索条件的符号列出，供我们一一查看。

Graph view提供的功能非常多，以上只是简单介绍了几个常用的功能，熟练地使用它们是进行更深入研究的保障。Graph view的界面比较简洁，各代码块间逻辑清晰，适合肉眼观看。相对来说，现阶段切换到Text view的机会比较少，一般是在配合LLDB进行动态调试时，才会在Text view中对界面左侧罗列的符

号地址特别关注，如图3-46所示。

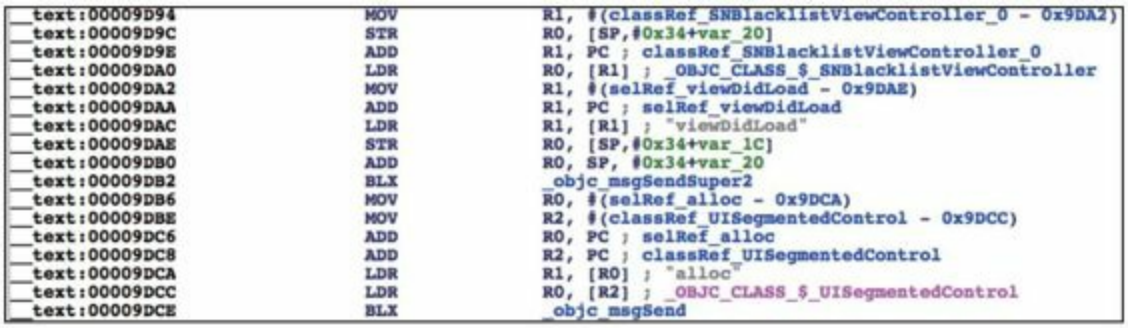


图3-46　Text view

需要注意的是，IDA的某些Bug会导致Graph view的末端显示不全（例如一个subroutine本来有100行指令，但只显示了80行），当你对某一个Graph view块中的指令产生明显怀疑时，可切换到Text view看看Graph view是不是漏显示了某些代码。这类Bug出现的概率不大，如果你不幸中彩，欢迎来http://bbs.iosre.com交流解决方案。

3.4.3　IDA分析示例

说了IDA的这么多使用方法，下面用一个简单的例子向大家演示IDA的威力。越狱iOS的用户都知

道，在Cydia中安装完一个tweak后，Cydia会建议我们“Restart SpringBoard”，那么这个respring的操作是如何实现的呢？请大家快速浏览3.5节，用iFunBox将iOS中
的“/System/Library/CoreServices/SpringBoard.app/Spring
贝到OSX中，并用IDA打开它，等待初始分析结束后在Function window里搜索“relaunch SpringBoard”，定位到这个函数，双击跳转到它的实现代码中，如图3-47所示。

```
; SpringBoard - (void)relaunchSpringBoard
; Attributes: bp-based frame

; void __cdecl -[SpringBoard relaunchSpringBoard](struct SpringBoard *self, SEL)
__SpringBoard_relaunchSpringBoard_

var_8= -8

PUSH            {R4,R7,LR}
ADD             R7, SP, #4
STR.W           R8, [SP,#4+var_8]!
SUB             SP, SP, #8
MOV             R0, #(_UIApp_ptr - 0x1990A)
MOVW            R1, #(:lower16:(selRef_beginIgnoringInteractionEvents - 0x19912))
ADD             R0, PC ; _UIApp_ptr
MOVT.W          R1, #(:upper16:(selRef_beginIgnoringInteractionEvents - 0x19912))
LDR             R0, [R0] ; _UIApp
ADD             R1, PC ; selRef_beginIgnoringInteractionEvents
LDR             R1, [R1] ; "beginIgnoringInteractionEvents"
LDR             R0, [R0]
BLX             _objc_msgSend
MOV             R0, #(off_40802C - 0x19928)
MOVW            R1, #(:lower16:(selRef_hideSpringBoardStatusBar - 0x19930))
ADD             R0, PC ; off_40802C
MOVT.W          R1, #(:upper16:(selRef_hideSpringBoardStatusBar - 0x19930))
LDR             R4, [R0] ; dword_4DD8B4
ADD             R1, PC ; selRef_hideSpringBoardStatusBar
LDR             R1, [R1] ; "hideSpringBoardStatusBar"
LDR             R0, [R4]
BLX             _objc_msgSend
MOVS            R0, #1
BL              sub_35D2C
MOVW            R2, #(:lower16:(cfstr_SpringboardRel - 0x1994C)) ; "SpringBoard relaunch"
MOVS            R0, #5
MOVT.W          R2, #(:upper16:(cfstr_SpringboardRel - 0x1994C)) ; "SpringBoard relaunch"
MOVS            R1, #0
ADD             R2, PC  ; "SpringBoard relaunch"
MOV.W           R8, #0
BL              sub_350B8
MOVW            R0, #(:lower16:(selRef_performSelector_withObject_afterDelay_ - 0x1996A))
MOVW            R9, #0
MOVT.W          R0, #(:upper16:(selRef_performSelector_withObject_afterDelay_ - 0x1996A))
MOV             R2, #(selRef__relaunchSpringBoardNow - 0x1996C)
ADD             R0, PC ; selRef_performSelector_withObject_afterDelay_
ADD             R2, PC ; selRef__relaunchSpringBoardNow
LDR             R1, [R0] ; "performSelector:withObject:afterDelay:"
MOVT.W          R9, #0x4010
LDR             R0, [R4]
MOVS            R3, #0
LDR             R2, [R2] ; "_relaunchSpringBoardNow"
STRD.W          R8, R9, [SP]
BLX             _objc_msgSend
```

图3-47　[SpringBoard relaunchSpringBoard]

可以看到，这个函数的实现流程既简单又清晰。根据从上到下的执行顺序，首先调用beginIgnoringInteractionEvents，开始忽略所有用户交互事件；然后调用hideSpringBoardStatusBar来隐藏状态栏；接着连续执行2个subroutine，分别是

sub_35D2C和sub_350B8。接下来，双击sub_35D2C，跳转到它的实现，看看它做了什么，如图3-48所示。

在图3-48中，一眼就能看到好多含有“log”字样的关键词，先“initialize”，然后判断是否“enabled”，最后“log”。稍微懂一点英语的朋友都能猜到，sub_35D2C的作用是将respring的一些操作记录下来，与respring的主体功能无关。点击IDA菜单栏上的蓝色后退按钮（如图3-49所示），或直接按ESC，回退到relaunchSpringBoard函数体中，继续往下分析。

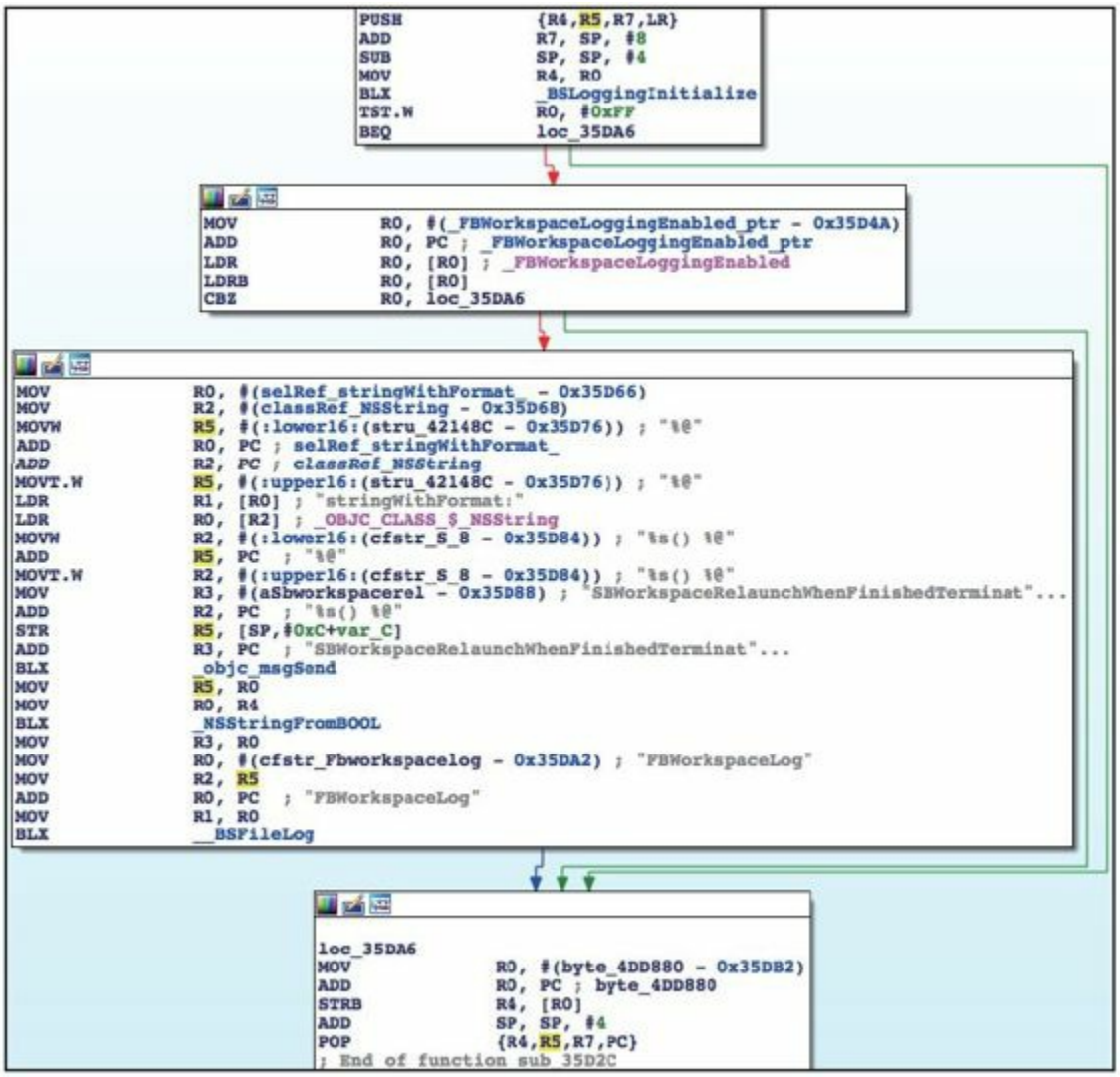


图3-48　sub_35D2C

图3-49　后退按钮

双击sub_350B8，跳转到如图3-50所示的界面。

从图3-50可以看到，这个subroutine只是在为调用

sub_350C4作一些准备工作而已。双击sub_350C4，跳转到它的实现，你会发现sub_350C4的上半部分与图3-48所示的sub_35D2C非常类似，都只是做了一些操作记录。但与sub_35D2C不同的是，sub_350C4还做了一些实际工作，如图3-51所示。

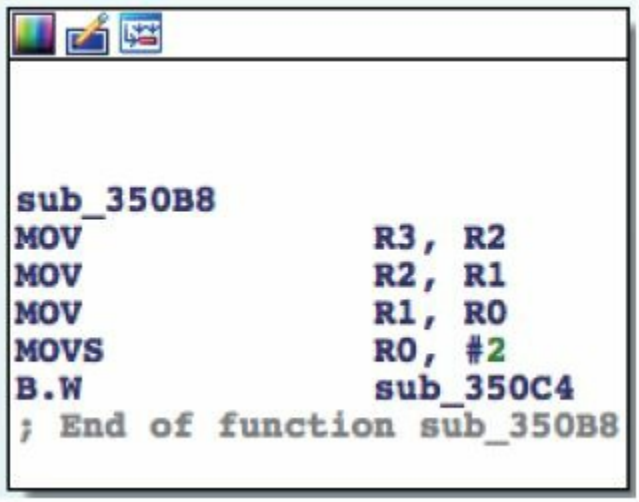


图3-50　sub_350B8

我们现在还不了解汇编语言，只能大致浏览一下这些关键词，不过，可以猜测出这个subroutine的作用是生成一个名为“TerminateApplicationGroup”的事件，指定其处理方法为sub_351F8，然后把生成的事件加入一个处理队列里，并依次处理队列里的事件，

从而关掉所有的App——商城关门之前，要关闭里面的所有店铺；respring之前，自然也要关掉所有的App。现在去看看sub_351F8的实现，如图3-52所示。

```
loc_35156
MOVW            R0, #(:lower16:(classRef_FBWorkspaceEvent - 0x3517E))
ADD.W           R10, SP, #0x44+var_34
MOVT.W          R0, #(:upper16:(classRef_FBWorkspaceEvent - 0x3517E))
MOV             R1, #(__NSConcreteStackBlock_ptr - 0x35172)
MOVW            R9, #(:lower16:(selRef_eventWithName_handler_ - 0x3519E))
ADD             R1, PC ; __NSConcreteStackBlock_ptr
MOVT.W          R9, #(:upper16:(selRef_eventWithName_handler_ - 0x3519E))
MOVW            R4, #(:lower16:(unk_40B640 - 0x351A2))
LDR             R1, [R1] ; __NSConcreteStackBlock
ADD             R0, PC ; classRef_FBWorkspaceEvent
MOVT.W          R4, #(:upper16:(unk_40B640 - 0x351A2))
MOV             R2, #(cfstr_Terminateappli - 0x351AA) ; "TerminateApplicationGroup"
LDR             R0, [R0] ; _OBJC_CLASS_$_FBWorkspaceEvent
MOV             R3, #(sub_351F8+1 - 0x351A4)
STR             R1, [SP,#0x44+var_3C]
MOV.W           R1, #0xC2000000
STR             R1, [SP,#0x44+var_38]
ADD             R9, PC ; selRef_eventWithName_handler_
MOVS            R1, #0
ADD             R4, PC ; unk_40B640
ADD             R3, PC ; sub_351F8
STMIA.W         R10, {R1,R3,R4}
ADD             R2, PC  ; "TerminateApplicationGroup"
ADD             R3, SP, #0x44+var_3C
LDR.W           R1, [R9] ; "eventWithName:handler:"
STR             R6, [SP,#0x44+var_24]
STR             R5, [SP,#0x44+var_20]
STRB.W          R8, [SP,#0x44+var_1C]
STR.W           R11, [SP,#0x44+var_28]
BLX             _objc_msgSend
MOV             R4, R0
MOV             R0, #(selRef_sharedInstance - 0x351D4)
MOV             R2, #(classRef_FBWorkspaceEventQueue - 0x351D6)
ADD             R0, PC ; selRef_sharedInstance
ADD             R2, PC ; classRef_FBWorkspaceEventQueue
LDR             R1, [R0] ; "sharedInstance"
LDR             R0, [R2] ; _OBJC_CLASS_$_FBWorkspaceEventQueue
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(selRef_executeOrAppendEvent_ - 0x351EA))
MOV             R2, R4
MOVT.W          R1, #(:upper16:(selRef_executeOrAppendEvent_ - 0x351EA))
ADD             R1, PC ; selRef_executeOrAppendEvent_
LDR             R1, [R1] ; "executeOrAppendEvent:"
BLX             _objc_msgSend
ADD             SP, SP, #0x2C
POP.W           {R8,R10,R11}
POP             {R4-R7,PC}
; End of function sub_350C4
```

图3-51　sub_350C4

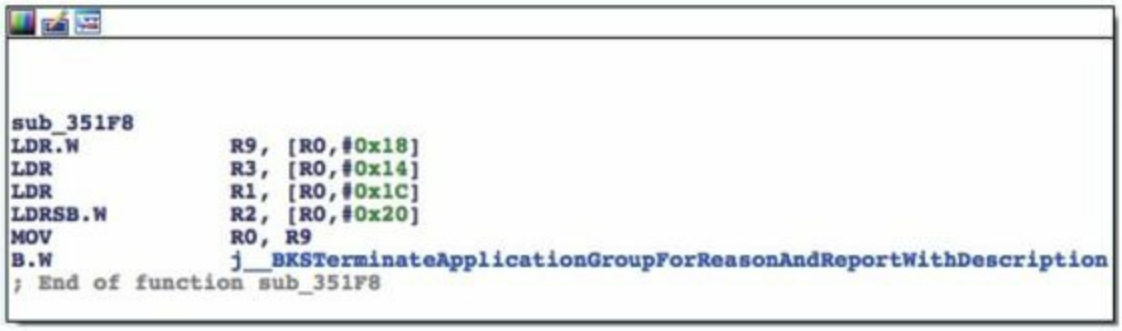


图3-52　sub_351F8

从BKSTerminateApplicationGroupForReasonAndReportWit的名字来看，其作用已经很明显了，它印证了刚刚对sub_350C4的分析。再次回退到relaunchSpringBoard函数体里，到这里，分析已经接近尾声了：函数调用_relaunchSpringBoardNow，完成respring操作。

不需要了解汇编代码，也不用熟悉调用规则，我们以几乎零基础的水平成功完成了这次逆向工程，不是吗？当然，这不能说明我们的水平有多么高深，而是提醒我们应该为拥有IDA这样强大且免费的神器感到幸运，从而激励我们加倍努力。在绝大多数情况下，对IDA的使用跟上面的示例没有本质区别，你只

需要耐着性子，仔细咀嚼IDA呈现出的每一行代码，要不了多久，你就会深切感受到逆向工程的艺术之美。

IDA的用法远不止本节所示的这么简单，如果你在使用过程中有任何疑问，都欢迎来http://bbs.iosre.com讨论交流。

3.5 iFunBox

iFunBox（如图3-53所示）是一款老牌iOS文件管理工具，可以非常方便地操作iOS中的文件。

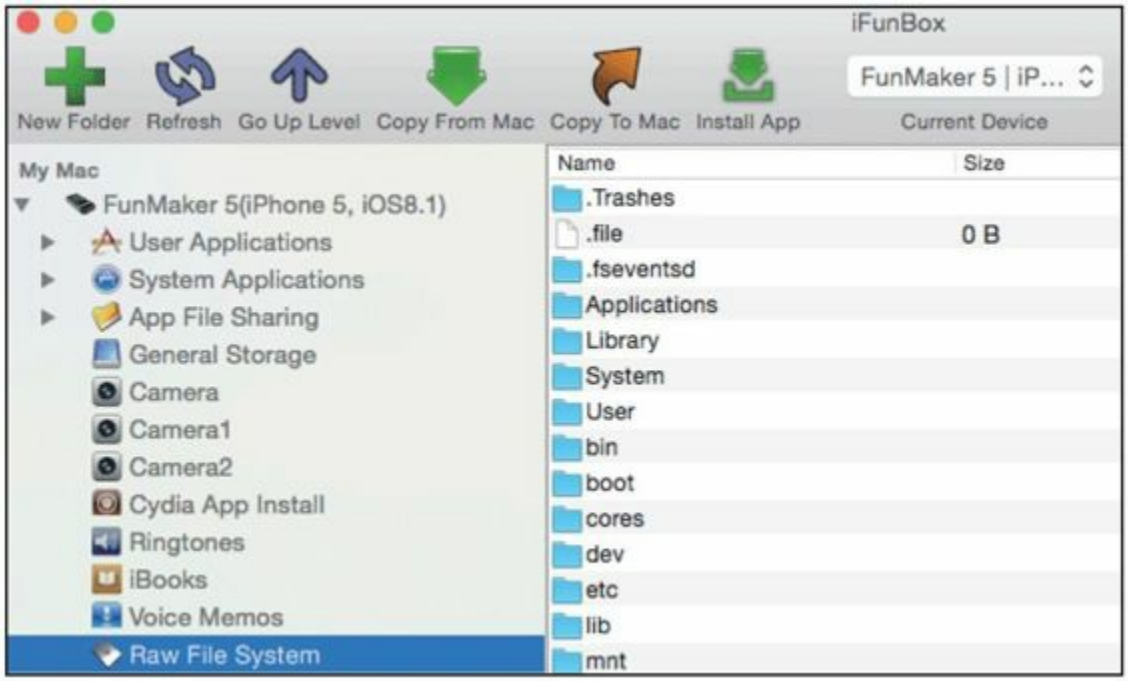


图3-53 iFunBox

iFunBox的使用并不复杂，我们主要用到的是它的文件传输功能。有一点需要注意的是，越狱iOS必须安装“Apple File Conduit 2”（如图3-54所示），简称AFC2，这样iFunBox才能够浏览iOS全系统文件，而且这也是接下来大部分操作的先决条件。

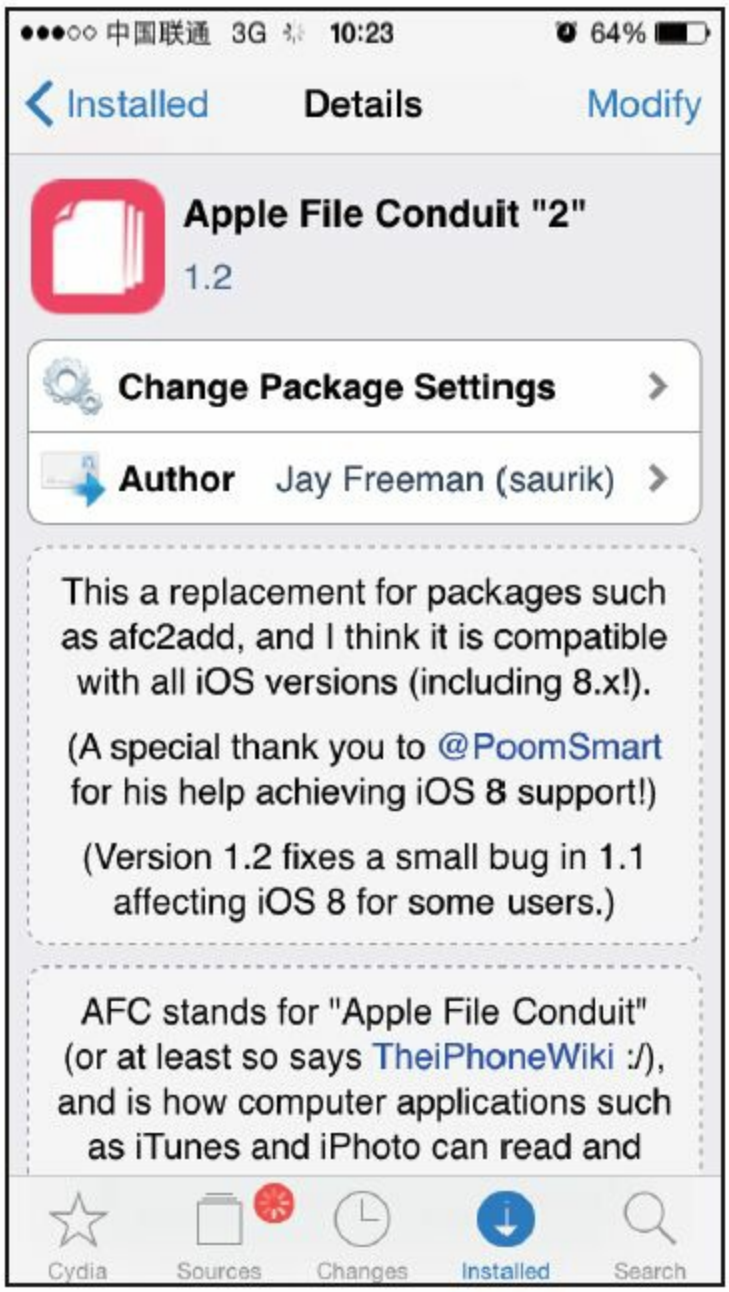


图3-54　Apple File Conduit 2

3.6 dyld_decache

安装了iFunBox和AFC2之后，不少读者会迫不及待地开始浏览iOS文件系统，看看这个封闭平台的表面下到底暗藏了多少玄机。相信大家很快就会发现一个问题：“/System/Library/Frameworks/”、“/System/Library/目录下，怎么没有库文件？

从iOS 3.1开始，包括frameworks在内的许多库文件被存进了一个大cache里，这个cache文件位于“/System/Library/Caches/com.apple.dyld/dyld_shared_为dyld_shared_cache_armv7、dyld_shared_cache_armv7s或dyld_shared_cache_arm64），可以使用KennyTM开发的dyld_decache将其中的二进制文件提取出来。这样做的好处是确保分析的文件来自本机，在使用Mac工具集与iOS工具集分析同一目标时，OSX与iOS上分析

出的指令和地址等数据是完全吻合的，避免了出现驴唇不对马嘴的低级错误。有关这个cache的进一步介绍，可以参阅DHowett的博客（http://blog.howett.net/2009/09/cache-or-check/）。

使用dyld_decache之前，要将“/System/Library/Caches/com.apple.dyld/dyld_shared_ iFunBox（不能用scp）从iOS拷贝到OSX中。然后从https://github.com/downloads/kennytm/Miscellaneous/dyl下载dyld_decache。解压之后赋予其可执行权限，如下：

```
snakeninnysiMac:~ snakeninny$ chmod +x
/path/to/dyld_decache\[v0.1c\]
```

然后开始提取二进制文件，如下：

```
snakeninnysiMac:~ snakeninny$ /path/to/dyld_decache\
[v0.1c\] -o /where/to/store/decached/binaries/
/path/to/dyld_shared_cache_armx
  0/877: Dumping
'/System/Library/AccessibilityBundles/AXSpeechImplementat

  1/877: Dumping
'/System/Library/AccessibilityBundles/AccessibilitySettin
```

```
  2/877: Dumping
'/System/Library/AccessibilityBundles/AccountsUI.axbundle

……
```

提取出的所有二进制文件都存放在了“/where/to/store/decached/binaries/”下。值得一提的是，逆向工程需要分析的二进制文件现在散落在OSX和iOS两个系统中，不方便查找，建议利用下一章提到的scp工具把iOS文件系统拷贝一份存在OSX里。

3.7 小结

本章重点介绍了class-dump、Theos、Reveal、IDA这4个工具，熟悉它们的使用方法，是我们一步步掌握iOS逆向工程的前提。

第4章　iOS工具集

第3章介绍了iOS逆向工程的OSX工具集，为了完成iOS逆向工程，还需要在iOS上安装、配置一系列工具，将两个平台联动起来。以下的操作均在iPhone 5，iOS 8.1中完成，如果你在使用中碰到了问题，请到http://bbs.iosre.com上交流反馈。

4.1 CydiaSubstrate

CydiaSubstrate（如图4-1所示）是绝大部分tweak正常工作的基础，它由MobileHooker、MobileLoader和Safe mode组成。

图4-1 CydiaSubstrate的logo

4.1.1 MobileHooker

MobileHooker的作用是替换系统函数，也就是所谓的hook，它主要包含以下两个函数：

```
void MSHookMessageEx(Class class, SEL selector, IMP
replacement, IMP *result);
void MSHookFunction(void* function, void* replacement,
void** p_original);
```

其中MSHookMessageEx作用于Objective-C函数，通过调用method_setImplementation函数将[class selector]的实现改为replacement，达到hook的目的。这是什么意思呢？举个简单的例子，向一个NSString对象发送hasSuffix:消息（即调用[NSString hasSuffix:]），正常情况下它的实现是判断NSString对象有没有某个后缀；如果把这个实现替换成hasPrefix:的实现，那么NSString对象在收到hasSuffix:消息后，实际进行的操作是“口是心非”地判断这个NSString对象有没有某个前缀（prefix）。是不是容易理解一些了？

第3章提到的Logos语法主要是对此函数作了一层封装，让编写针对Objective-C函数的hook代码变得更简单直观了，但其底层实现仍完全基于MSHookMessageEx。对于Objective-C函数的hook，推荐使用更一目了然的Logos语法。如果对MSHookMessageEx本身的使用感兴趣，可以去看它

的官方文档，或者Google搜索“cydiasubstrate fuchsiaexample”，以“http://www.cydiasubstrate.com”开头的那个链接就是了。

MSHookFunction作用于C和C++函数，通过编写汇编指令，在进程执行到function时转而执行replacement，同时保存function的指令及其返回地址，使得用户可以选择性地执行function，并保证进程能够在执行完replacement后继续正常运行。

上面这段话可能有些晦涩难懂，这里用两张图来解释一下，先看图4-2。

在图4-2中，进程先执行一些指令，再执行函数A，接着执行剩下的指令。如果钩住（hook）了函数A（即上面说的function），想用函数B（即replacement）替换它，那么进程的执行流程就变成了图4-3的样子。

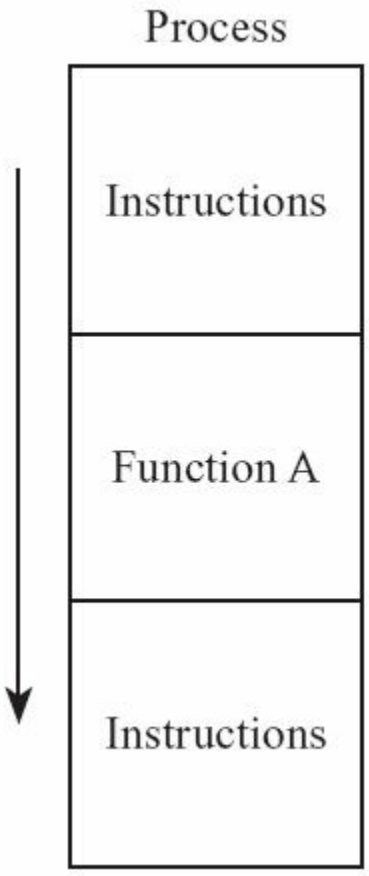


图4-2　进程正常执行流程

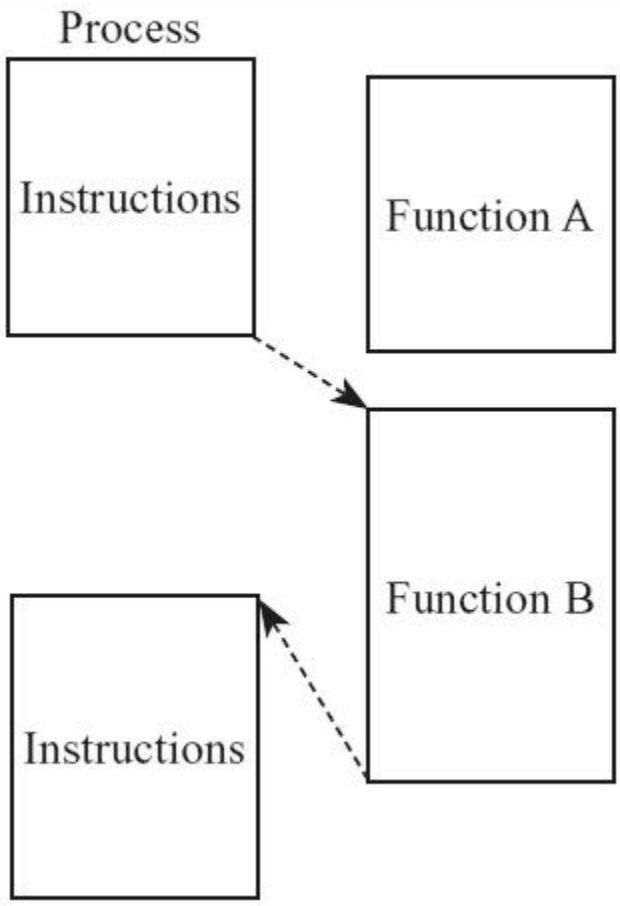


图4-3　用B替换A

在图4-3中，进程先执行一些指令，在原本应该执行函数A的地方跳转到函数B的位置执行函数B，同时函数A的代码被MobileHooker保存了下来。在函数B中，可以选择是否执行函数A，以及何时执行函数A，在函数B执行完成后，则会继续执行剩下的指令。

值得注意的是，MSHookFunction对function的指令总长度是有要求的，即function里所有指令加起来的长度不能太短（saurik曾在非正式的场合里说过大约是至少8字节，但此数字未经严格验证，请不要以此为准）。那么你可能会问了，如果想钩住（hook）那些短函数，该怎么办呢？

一种变通的方法是钩住（hook）短函数内部调用的其他函数——短函数之所以短，是因为内部一般都调用了其他函数，由其他函数来做出实际操作。因此把长度满足要求的其他函数作为MSHookFunction的目标，然后在replacement里做一些逻辑判断，将它与短函数关联上，再把对短函数的修改写在这里就好了。

如果看了这两段话仍有不解之处，没关系，在了解了MSHookFunction大概的工作原理之后，会以一个简单的例子，解释上面提到的这些内容。需要说明的是，这个例子里会涉及比较多的底层知识，对于新手

来说理解会有一定的困难，如果接触逆向工程的时间不长，看不懂下面的例子也没关系，直接跳到4.1.2节吧。当你在实际操作中碰到了类似的情况，再来回看这一小节，相信那时会对这些内容有更好的理解。此外，欢迎来http://bbs.iosre.com参与讨论。

下面一起来完成如下操作。

1）用Theos新建iOSRETargetApp，命令如下：

```
snakeninnys-MacBook:Code snakeninny$
/opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/library
  [3.] iphone/preference_bundle
  [4.] iphone/tool
  [5.] iphone/tweak
Choose a Template (required): 1
Project Name (required): iOSRETargetApp
Package Name [com.yourcompany.iosretargetapp]:
com.iosre.iosretargetapp
Author/Maintainer Name [snakeninny]: snakeninny
Instantiating iphone/application in iosretargetapp/...
Done.
```

2）修改RootViewController.mm，命令如下：

```
#import "RootViewController.h"
```

```
class CPPClass
{
      public:
            void CPPFunction(const char *);
};
void CPPClass::CPPFunction(const char *arg0)
{
      for (int i = 0; i < 66; i++) // This for loop makes
this function long enough to validate MSHookFunction
      {
            u_int32_t randomNumber;
            if (i % 3 == 0) randomNumber =
arc4random_uniform(i);
            NSProcessInfo *processInfo = [NSProcessInfo
processInfo];
            NSString *hostName = processInfo.hostName;
            int pid = processInfo.processIdentifier;
            NSString *globallyUniqueString =
processInfo.globallyUniqueString;
            NSString *processName =
processInfo.processName;
            NSArray *junks = @[hostName,
globallyUniqueString, processName];
            NSString *junk = @"";
            for (int j = 0; j < pid; j++)
            {
                  if (pid % 6 == 0) junk = junks[j % 3];
            }
            if (i % 68 == 1) NSLog(@"Junk: %@", junk);
      }
      NSLog(@"iOSRE: CPPFunction: %s", arg0);
}
extern "C" void CFunction(const char *arg0)
{
      for (int i = 0; i < 66; i++) // This for loop makes
this function long enough to validate MSHookFunction
      {
            u_int32_t randomNumber;
            if (i % 3 == 0) randomNumber =
arc4random_uniform(i);
            NSProcessInfo *processInfo = [NSProcessInfo
processInfo];
            NSString *hostName = processInfo.hostName;
            int pid = processInfo.processIdentifier;
            NSString *globallyUniqueString =
processInfo.globallyUniqueString;
            NSString *processName =
processInfo.processName;
            NSArray *junks = @[hostName,
```

```
globallyUniqueString, processName];
            NSString *junk = @"";
            for (int j = 0; j < pid; j++)
            {
                  if (pid % 6 == 0) junk = junks[j % 3];
            }
            if (i % 68 == 1) NSLog(@"Junk: %@", junk);
      }
      NSLog(@"iOSRE: CFunction: %s", arg0);
}
extern "C" void ShortCFunction(const char *arg0) //
ShortCFunction is too short to be hooked
{
      CPPClass cppClass;
      cppClass.CPPFunction(arg0);
}
@implementation RootViewController
- (void)loadView {
      self.view = [[[UIView alloc] initWithFrame:
[[UIScreen mainScreen] applicationFrame]] autorelease];
      self.view.backgroundColor = [UIColor redColor];
}
- (void)viewDidLoad
{
      [super viewDidLoad];
      CPPClass cppClass;
      cppClass.CPPFunction("This is a C++ function!");
      CFunction("This is a C function!");
      ShortCFunction("This is a short C function!");
}
@end
```

上面简单地写了一个CPPClass::CPPFunction、一个CFunction和一个ShortCFunction，作为hook的对象。这里特意在CPPClass::CPPFunction和CFuntion里添加了一些无用代码，目的仅仅是增加这两个函数的长度，使得针对它们俩的MSHookFunction生效。而

ShortCFunction会因长度太短，导致针对它的MSHookFunction失效。但是在接下来的tweak中会巧妙地回避这个问题。

3）修改iOSRETargetApp的Makefile并安装，命令如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
APPLICATION_NAME = iOSRETargetApp
iOSRETargetApp_FILES = main.m
iOSRETargetAppApplication.mm RootViewController.mm
iOSRETargetApp_FRAMEWORKS = UIKit CoreGraphics
include $(THEOS_MAKE_PATH)/application.mk
after-install::
      install.exec "su mobile -c uicache"
```

在上面这段代码中，最后那句“su mobile–c uicache”用来刷新桌面UI缓存，显示出iOSRETargetApp图标。在Terminal里运行“make package install”命令将其安装到设备上。运行iOSRETargetApp，待红色背景显示之后ssh到iOS上，看看产生的输出与期待的结果是否相符，如下：

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 18 11:13:34 FunMaker-5 iOSRETargetApp[5072]: iOSRE:
CPPFunction: This is a C++ function!
Nov 18 11:13:34 FunMaker-5 iOSRETargetApp[5072]: iOSRE:
CFunction: This is a C function!
Nov 18 11:13:35 FunMaker-5 iOSRETargetApp[5072]: iOSRE:
CPPFunction: This is a short C function!
```

4）用Theos新建iOSREHookerTweak，命令如下：

```
snakeninnys-MacBook:Code snakeninny$
/opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/library
  [3.] iphone/preference_bundle
  [4.] iphone/tool
  [5.] iphone/tweak
Choose a Template (required): 5
Project Name (required): iOSREHookerTweak
Package Name [com.yourcompany.iosrehookertweak]:
com.iosre.iosrehookertweak
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.iosre.iosretargetapp
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]: iOSRETargetApp
Instantiating iphone/tweak in iosrehookertweak/...
Done.
```

5）修改Tweak.xm，命令如下：

```
#import <substrate.h>
void (*old__ZN8CPPClass11CPPFunctionEPKc)(void *, const
char *);
```

```
void new__ZN8CPPClass11CPPFunctionEPKc(void *hiddenThis,
const char *arg0)
{
      if (strcmp(arg0, "This is a short C function!") ==
0) old__ZN8CPPClass11CPPFunctionEPKc(hiddenThis, "This is
a hijacked short C function from
new__ZN8CPPClass11CPPFunctionEPKc!");
      else old__ZN8CPPClass11CPPFunctionEPKc(hiddenThis,
"This is a hijacked C++ function!");
}
void (*old_CFunction)(const char *);
void new_CFunction(const char *arg0)
{
      old_CFunction("This is a hijacked C function!"); //
Call the original CFunction
}
void (*old_ShortCFunction)(const char *);
void new_ShortCFunction(const char *arg0)
{
      old_CFunction("This is a hijacked short C function
from new_ShortCFunction!"); // Call the original
ShortCFunction
}
%ctor
{
      @autoreleasepool
      {
            MSImageRef image =
MSGetImageByName("/Applications/iOSRETargetApp.app/iOSRET

            void *__ZN8CPPClass11CPPFunctionEPKc =
MSFindSymbol(image, "__ZN8CPPClass11CPPFunctionEPKc");
            if (__ZN8CPPClass11CPPFunctionEPKc)
NSLog(@"iOSRE: Found CPPFunction!");
            MSHookFunction((void
*)__ZN8CPPClass11CPPFunctionEPKc, (void
*)&new__ZN8CPPClass11CPPFunctionEPKc, (void
**)&old__ZN8CPPClass11CPPFunctionEPKc);
            void *_CFunction = MSFindSymbol(image,
"_CFunction");
            if (_CFunction) NSLog(@"iOSRE: Found
CFunction!");
            MSHookFunction((void *)_CFunction, (void
*)&new_CFunction, (void **)&old_CFunction);
            void *_ShortCFunction = MSFindSymbol(image,
"_ShortCFunction");
            if (_ShortCFunction) NSLog(@"iOSRE: Found
ShortCFunction!");
            MSHookFunction((void *)_ShortCFunction, (void
```

```
*)&new_ShortCFunction, (void **)&old_ShortCFunction); //
This MSHookFuntion will fail because ShortCFunction is
too short to be hooked
      }
}
```

在这段代码中，有很多需要注意的地方，如下所示。

· MSFindSymbol的作用

简单地说，MSFindSymbol的作用是查找待钩住（hook）的symbol。那symbol又是什么呢？

在计算机中，一个函数的指令被存放在一段内存中，当进程需要执行这个函数时，它必须知道要去内存的哪个地方找到这个函数，然后执行它的指令。也就是说，进程要根据这个函数的名称，找到它在内存中的地址，而这个名称与地址的映射关系，是存储在“symbol table”中的——“symbol table”中的symbol就是这个函数的名称，进程会根据这个symbol找到它在内存中的地址，然后跳转过去执行。

试想这样一个场景：你的软件调用了一个库，这个库里有一个lookup函数，用于到你的服务器上查询信息。另一个软件如果知道了这个函数的symbol，那它岂不是可以导入这个库，然后随意调用lookup，消耗你的服务器资源，为它自己提供便利？

为了避免这种情况，symbol被分为2类，即public symbol与private symbol（其实还有一类stripped symbol，但跟本章关系不大，这里就不介绍了，感兴趣的朋友可以浏览下面提供的参考链接，或自行查阅相关资料）。别人的private symbol不是你想用，想用就能用。也就是说，MSHookFunction直接作用在private symbol上是无效的。所以saurik提供了MSFindSymbol这个API来访问private symbol。如果你仍然不清楚什么是symbol，只需要记住下面的写法就好：

```
MSImageRef image =
MSGetImageByName("/path/to/binary/who/contains/the/implem

void *symbol = MSFindSymbol(image, "symbol");
```

其中MSGetImageByName的参数是“symbol代表的函数其实现（implementation）所在的二进制文件的全路径”。不说绕口令，举个例子，NSLog函数的实现位于Foundation库，所以对于NSLog这个symbol来说，MSGetImageByName的参数就应该是“/System/Library/Frameworks/Foundation.framework/I简单吧？

对MSFindSymbol函数更详细的解释可以参考其官方文档http://www.cydiasubstrate.com/api/c/MSFindSymbol/；关于symbol的种类及定义，请阅读http://msdn.microsoft.com/en-us/library/windows/hardware/ff553493(v=vs.85).aspx，以及http://en.wikibooks.org/wiki/Reverse_Engineering/Mac_O或者自行查阅相关资料。

· symbol的来源

你可能已经注意到了，我们在iOSRETargetApp的RootViewController.mm中定义的3个函数名分别是CPPClass::CPPFunction、CFunction和ShortCFunction，怎么到了iOSREHookerTweak的tweak.xm里，它们却变成了__ZN8CPPClass11CPPFunctionEPKc、_CFunction和_ShortCFunction？简单地说，这是因为编译器对函数名做了进一步的处理。处理过程是什么样的不需要关心，我们关心的是处理结果，这3个以下划线开头的symbol是怎么来的？因为在实战中，我们拿不到被hook函数的源代码，所以一般情况下，这些symbol都是从IDA对二进制文件的分析结果中提取的。下面来看一个简单的例子。

把iOSRETargetApp二进制文件丢到IDA里，初始分析完成后的Functions Window如图4-4所示。

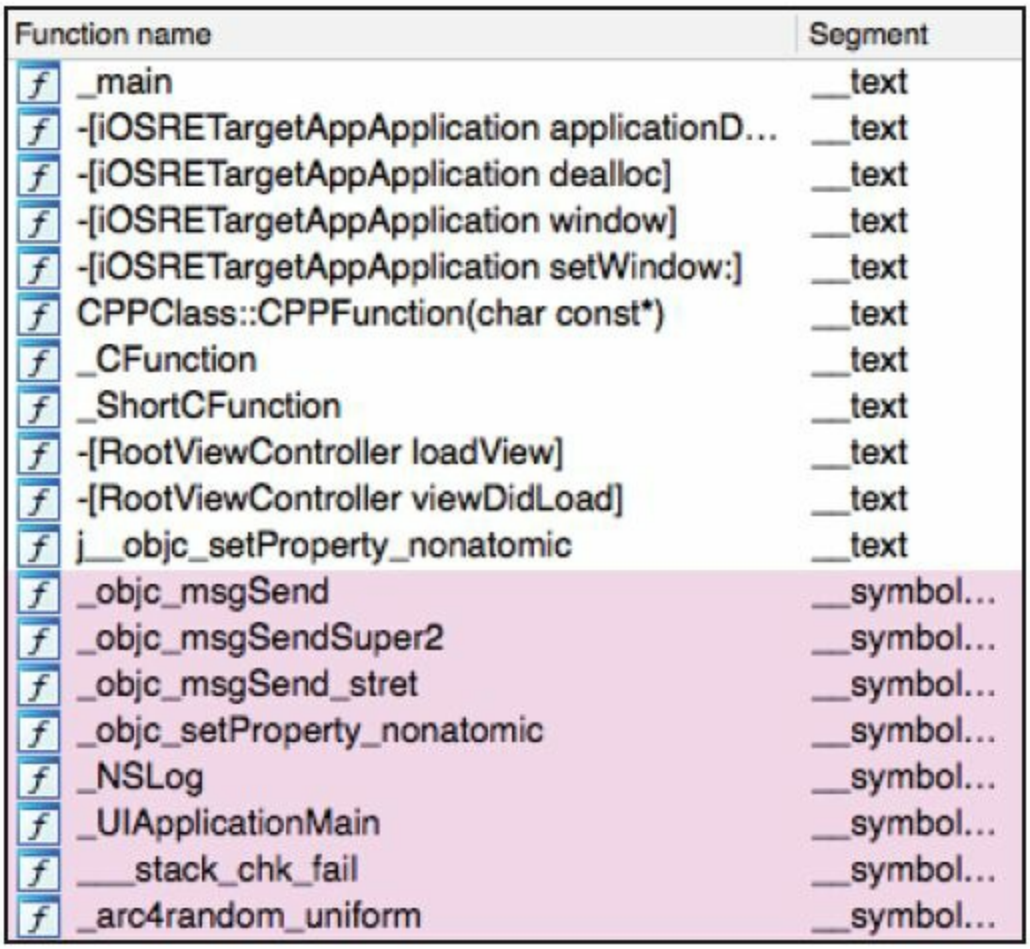
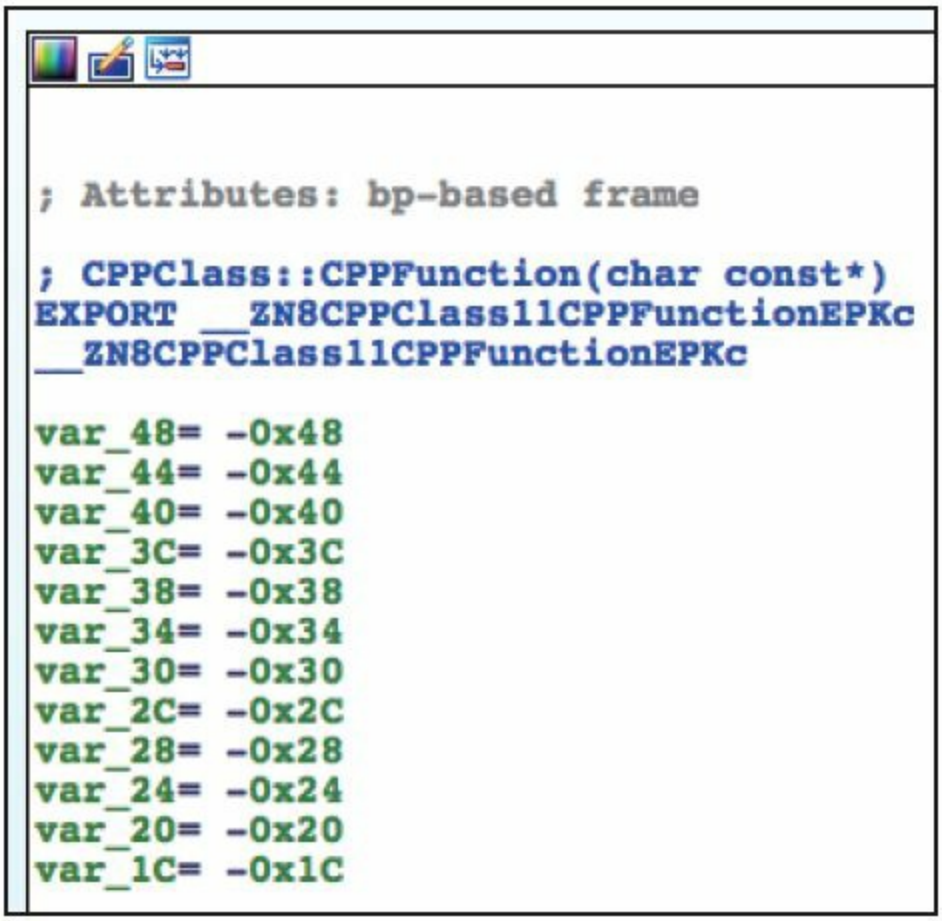


图4-4　Function window

可以看到，CPPClass::CPPFunction(char const*)、_CFunction和_ShortCFunction位列其中。双击“CPPClass::CPPFunction(char const*)”，跳转到其实现上，如图4-5所示。

第4行下划线开头的这个字符串，就是我们要找的symbol。同理，_CFunction和_ShortCFunction的来

源也显而易见了，如图4-6和图4-7所示。



```
; Attributes: bp-based frame

; CPPClass::CPPFunction(char const*)
EXPORT __ZN8CPPClass11CPPFunctionEPKc
__ZN8CPPClass11CPPFunctionEPKc

var_48= -0x48
var_44= -0x44
var_40= -0x40
var_3C= -0x3C
var_38= -0x38
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C
```

图4-5　CPPClass::CPPFunction(char const*)

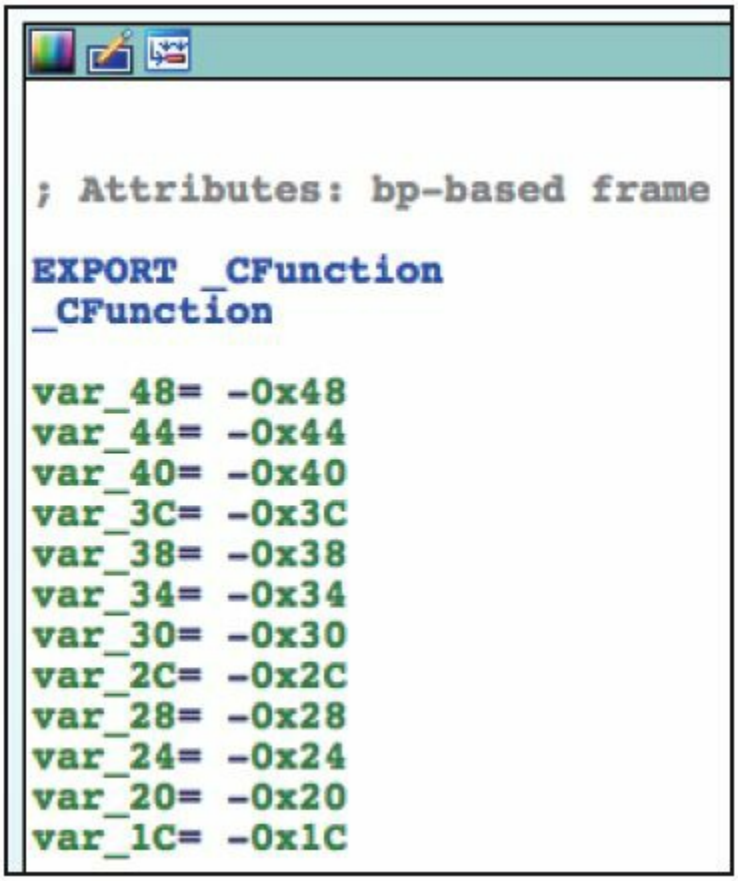


图4-6　CFunction

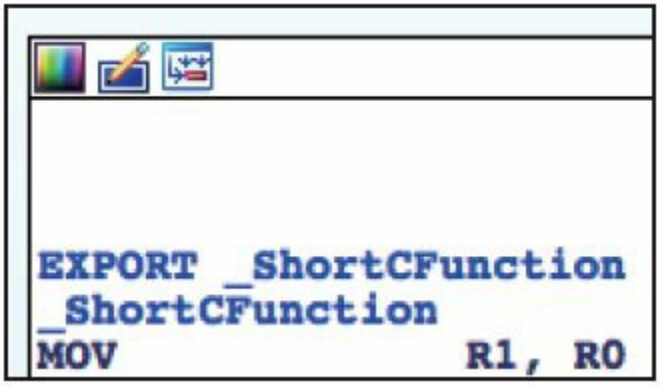


图4-7　ShortCFunction

此方法适用于查找任何symbol，在初学阶段，建议不要纠结symbol是怎么生成的，对这个知识点

的“不知其所以然”无伤大雅，只需要记住symbol跟函数名不同就够了。在学习逆向工程的整个过程中，symbol的概念一定会潜移默化地融入你的知识体系，无须刻意去强化。

· MSHookFunction的写法

MSHookFunction的三个参数的作用分别是：替换的原函数、替换函数，以及被MobileHooker保存的原函数。红花还需绿叶衬，单独的一个MSHookFunction函数是没有意义的，需要有一套固定的体系来承载它，这个体系的写法如下：

```
#import <substrate.h>
returnType (*old_symbol)(args);
returnType new_symbol(args)
{
      // Whatever
}
void InitializeMSHookFunction(void) // This function is
often called in %ctor i.e. constructor
{
      MSImageRef image =
MSGetImageByName("/path/to/binary/who/contains/the/implem

      void *symbol = MSFindSymbol(image, "symbol");
      if (symbol) MSHookFunction((void *)symbol, (void
*)&new_ symbol, (void **)&old_ symbol);
      else NSLog(@"Symbol not found!");
```

```
}
```

相信对比了上面的Tweak.xm，你很快就能理解这套体系的含义了。与symbol的情况类似，在实战中，我们拿不到被钩住（hook）的函数的源代码，函数的原型我们是不知道的，因此returnType究竟是什么，args一共有几个，各是什么类型，我们一无所知。这时，就需要借助更高级的逆向工程技术来还原出被钩住（hook）的函数原型了。这部分知识会在第6章重点讲述，现在不理解完全是正常的。建议读者在看完第6章后复习本节的内容，一定会有新的体会。

6）修改iOSREHookerTweak的Makefile并安装，命令如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
TWEAK_NAME = iOSREHookerTweak
iOSREHookerTweak_FILES = Tweak.xm
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
```

```
install.exec "killall -9 iOSRETargetApp"
```

到这里，请再次运行iOSRETargetApp，看看产生的输出，确定结果是否符合预期，如下：

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE:
Found CPPFunction!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE:
Found CFunction!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE:
Found ShortCFunction!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE:
CPPFunction: This is a hijacked C++ function!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE:
CFunction: This is a hijacked C function!
Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE:
CPPFunction: This is a hijacked short C function from
new__ZN8CPPClass11CPPFunctionEPKc!
```

值得一提的是，对短函数（即ShortCFunction）的直接hook失效了（否则会输出“This is a hijacked short C function from new_ShortCFunction!”），而对短函数内部调用的其他函数（即CPPClass::CPPFunction）的hook却是有效的，可通过判断它的参数，推测出它的调用者是ShortCFunction，这样一来，即可间接hook短函数，

从而达到围魏救赵的效果。以上介绍的MSHookFunction体系基本涵盖了初学者可能碰到的所有问题，由于Theos仅提供了MSHookMessageEx的封装，掌握这套体系的用法就显得尤为重要了。如果还有什么不明白的地方，可到http://bbs.iosre.com上讨论。

4.1.2 MobileLoader

MobileLoader的作用是加载第三方dylib。在iOS启动时，会由launchd将MobileLoader载入内存，然后MobileLoader会根据dylib的同名plist文件指定的作用范围，有选择地在不同进程里通过dlopen函数打开目录/Library/MobileSubstrate/DynamicLibraries/下的所有dylib。这个plist文件的格式已在Theos部分详细讲解，此处不再赘述。对于大多数初级iOS逆向工程师来说，MobileLoader的工作过程是完全透明的，此处仅作简单了解即可。

4.1.3 Safe mode

应用的质量良莠不齐，程序崩溃在所难免。因为tweak的本质是dylib，寄生在别的进程里，一旦出错，可能会导致整个进程崩溃，而一旦崩溃的是SpringBoard等系统进程，则会造成iOS瘫痪，所以CydiaSubstrate引入了Safe mode，它会捕获SIGTRAP、SIGABRT、SIGILL、SIGBUS、SIGSEGV、SIGSYS这6种信号，然后进入安全模式，如图4-8所示。

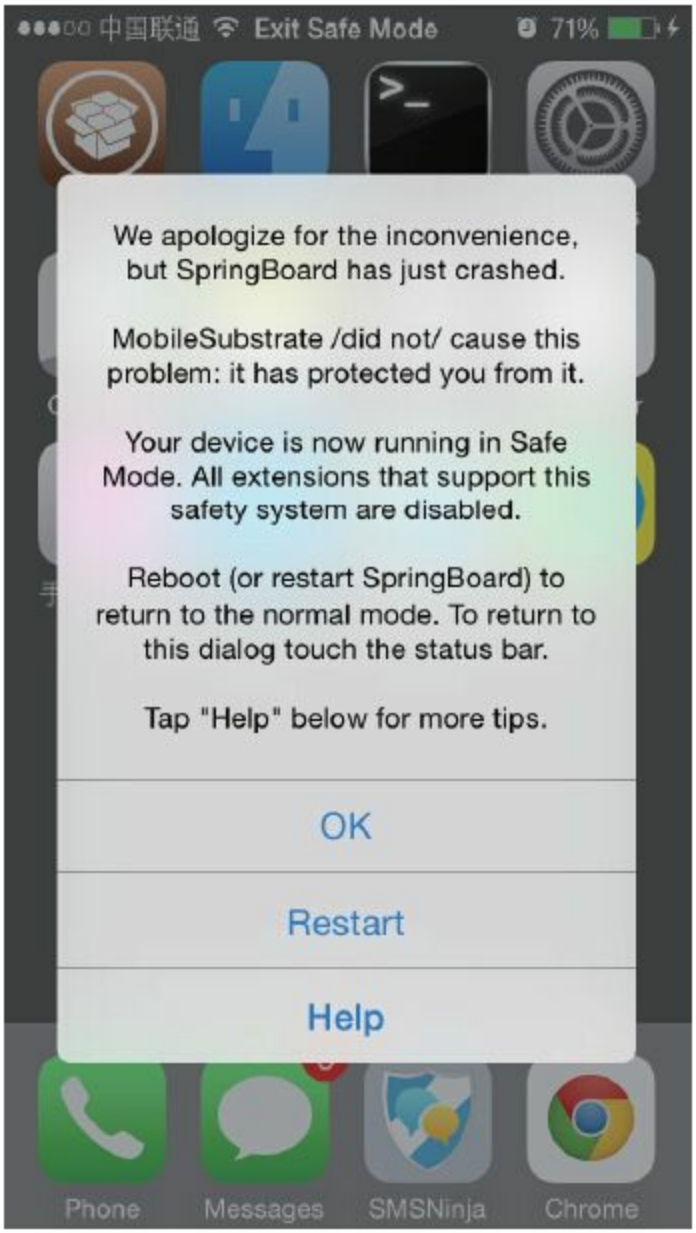


图4-8　安全模式

在安全模式里，所有基于CydiaSubstrate的第三方dylib均会被禁用，便于查错与修复。但是，并不是拥有了Safe mode就能高枕无忧，在很多时候，设备还是会因为第三方dylib的原因而无法进入系统，症状主

要有：开机时卡在白苹果上，或者进度圈不停地转。在出现这种情况时，可以同时按住home和lock键硬重启，然后按住音量“+”键来完全禁用CydiaSubstrate，待系统重启完毕后，再来查错与修复。当问题被成功修复后，再重启一次iOS，就能重新启用CydiaSubstrate了，非常方便。

4.2 Cycript

Cycript是由saurik推出的一款脚本语言（如图4-9所示），可以看作是Objective-JavaScript。

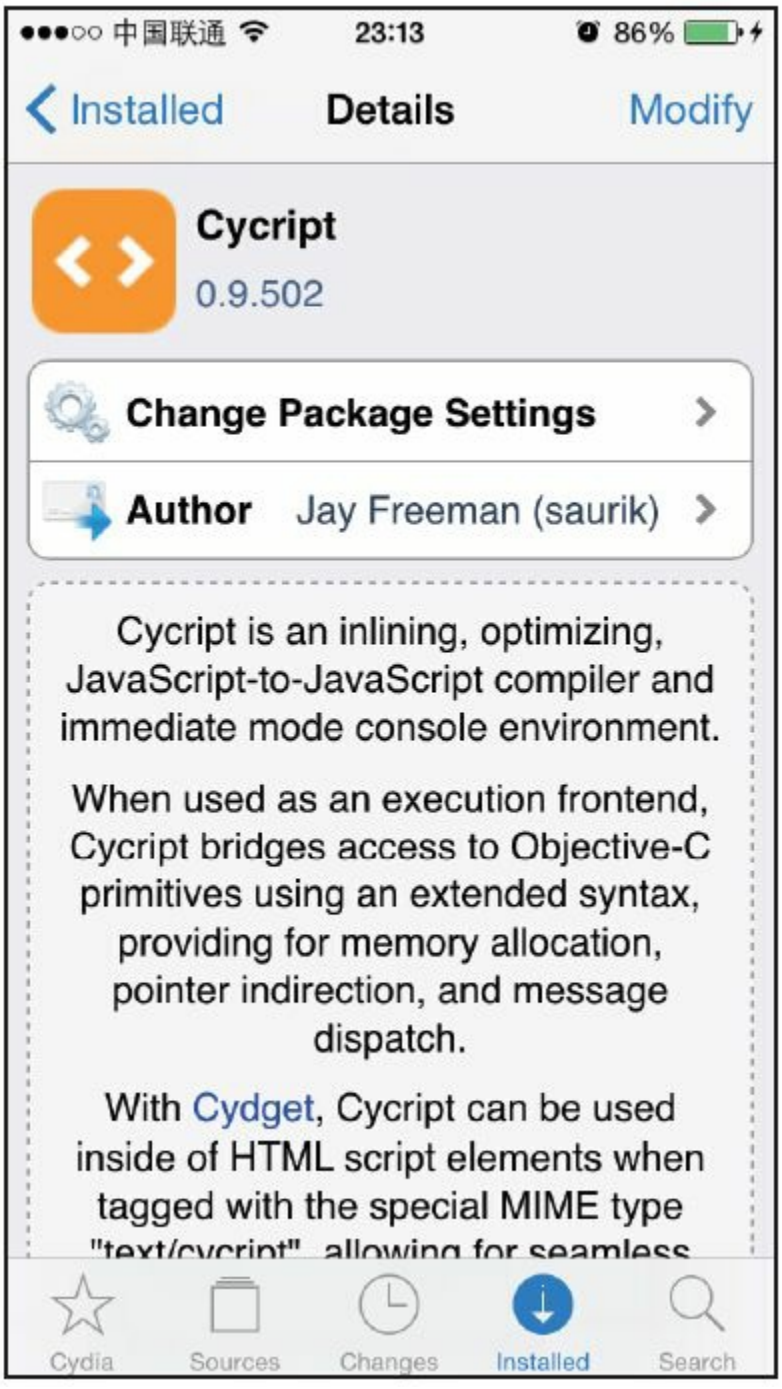


图4-9　Cycript

很多朋友可能会因为不了解JavaScript，所以在潜意识里认为Cycript很晦涩。事实上，笔者也不懂JavaScript，因为懒得学习新知识，所以在知道Cycript的很长一段时间里故意无视它，直到有一次在公司的无聊会议中把玩MTerminal，在Cycript中完成了几个函数的测试，节省了不少时间，才重新认识这门语法简单而功能强大的语言。其实对于熟悉Objective-C的朋友们来说，脚本语言不难上手，只要克服自己的畏难情绪，就一定能快速掌握它，Cycript当然也不例外。Cycript具备脚本语言的便利，可以直接用来写App，但saurik自己都说：“This isn't quite‘ready for primetime’”；笔者认为，Cycript最为贴心和实用的功能是它可以帮助我们轻松测试函数效果，整个过程安全无副作用，效果十分显著，实乃业界良心！因此，本书只对此功能作简单介绍，更多详细资料可参阅它的官网http://www.cycript.org。

可以从MTerminal中执行Cycript，也可以ssh到iOS中执行Cycript。输入“cycript”，出现“cy#”提示符，说明已成功启动Cycript，如下：

```
FunMaker-5:~ root# cycript
cy#
```

在启动Cycript之后，就可以开始编写App了。因为这里主要用到它测试函数的功能，而不是用它来写App，所以需要把代码注入一个现成的进程中，让代码运行起来。按下“control+D”，先退出Cycript。一般来说，选择注入哪个进程，要依测试的具体函数而定，这个函数所属的类存在于哪些进程，则注入这些进程，从而保证这个类是存在的。这句话的含义有些难以理解，举例说明如下。

假如现在要测试PhoneApplication类的+sharedNumberFormatter函数功能及其返回值，则必须注入MobilePhone这个进程，因为PhoneApplication

类只存在于MobilePhone进程中；同理，如果要测试SBUIController类的-lockFromSource:函数功能，则必须注入SpringBoard这个进程；当然，如果要测试NSString类的-length函数功能及其返回，则可注入任意链接了Foundation库的进程。因为需要用Cycript测试的一般都是私有函数，所以一个总的准则是从哪个进程逆向出的函数，就注入这个进程来测试；从哪个库逆向出的函数，就注入链接这个库的进程来测试。

通过进程注入方式调用Cycript测试函数的步骤很简单，以SpringBoard为例，首先找到进程名或PID，如下：

```
FunMaker-5:~ root# ps -e | grep SpringBoard
 4567 ??         0:27.45
/System/Library/CoreServices/SpringBoard.app/SpringBoard
 4634 ttys000    0:00.01 grep SpringBoard
```

SpringBoard进程的PID是4634。接下来输入“cycript-p 4634”或“cycript-p SpringBoard”，把Cycript注入SpringBoard，这时，Cycript就已经运行在

SpringBoard进程里，可以开始测试了。

我们都知道，UIAlertView是iOS中使用最多的弹框类。使用Objective-C语言，弹出一个对话框只需要3行代码，如下：

```
UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"iOSRE" message:@"snakeninny" delegate:nil
cancelButtonTitle:@"OK" otherButtonTitles:nil];
[alertView show];
[alertView release];
```

上面的Objective-C语言转化成Cycript非常简单，如下：

```
FunMaker-5:~ root# cycript -p SpringBoard
cy# alertView = [[UIAlertView alloc]
initWithTitle:@"iOSRE" message:@"snakeninny" delegate:nil
cancelButtonTitle:@"OK" otherButtonTitles:nil]
#"<UIAlertView: 0x1700e580; frame = (0 0; 0 0); layer =
<CALayer: 0x164146c0>>"
cy# [alertView show]
cy# [alertView release]
```

不需要声明对象类型，也不需要结尾的分号，就是这么简单。如果函数有返回值，Cycrip会把它在内存中的地址及一些基本信息实时打印出来，非常直

观。执行上面的语句后，SpringBoard会弹出对话框，如图4-10所示。

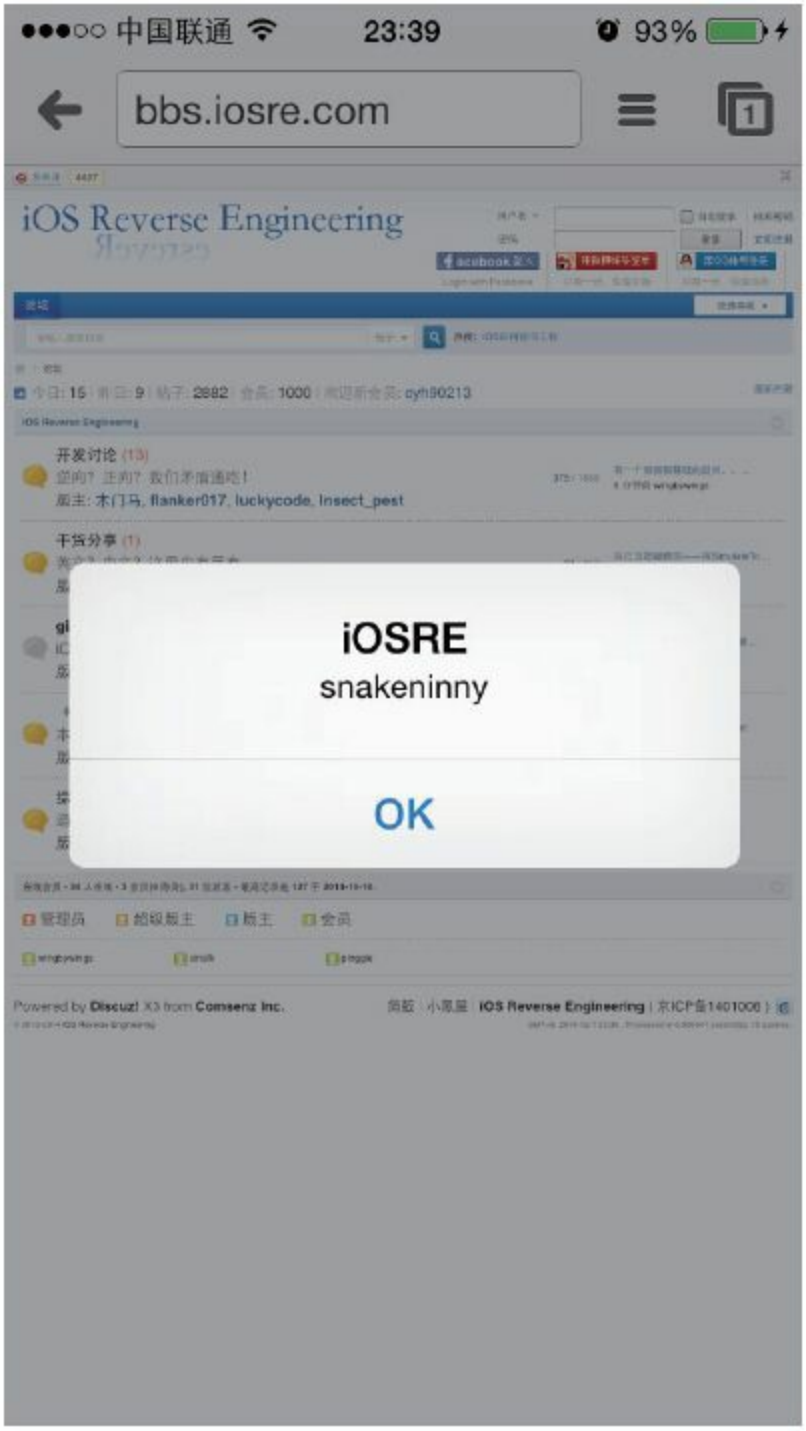


图4-10　用Cycript执行代码

如果知道一个对象在内存中的地址，可以通过“#”操作符来获取这个对象，例如：

```
cy# [[UIAlertView alloc] initWithTitle:@"iOSRE"
message:@"snakeninny" delegate:nil
cancelButtonTitle:@"OK" otherButtonTitles:nil]
#"<UIAlertView: 0x166b4fb0; frame = (0 0; 0 0); layer =
<CALayer: 0x16615890>>"
cy# [#0x166b4fb0 show]
cy# [#0x166b4fb0 release]
```

如果知道一个类对象存在于当前的进程中，却不知道它的地址，不能通过“#”操作符来获取它，此时，不妨试试choose命令，它的用法如下：

```
cy# choose(SBScreenShotter)
[#"<SBScreenShotter: 0x166e0e20>"]
cy# choose(SBUIController)
[#"<SBUIController: 0x16184bf0>"]
```

只需要choose一个类，Cycript就能帮你在内存中找出一个它的对象，供你使用。太方便了是不是？不过，choose命令并不是百发百中的，当它不能返回给你一个可用对象时，就必须手动寻找了。这部分内容会在第6章详细介绍。

测试私有函数的方法用到的一般也就是上面几个命令，下面以登录iMessage的Apple ID为例，用Cycript测试并实现这个功能。先拿到iMessage的登录管理器，命令如下：

```
FunMaker-5:~ root# cycript -p SpringBoard
cy# controller = [CNFRegController
controllerForServiceType:1]
#"<CNFRegController: 0x166401e0>"
```

然后登录自己的iMessage，命令如下：

```
cy# [controller
beginAccountSetupWithLogin:@"snakeninny@gmail.com"
password:@"bbs.iosre.com" foundExisting:NO]
#"IMAccount: 0x166e7b30 [ID: 5A8E19BE-1BC9-476F-AD3B-
729997FAA3BC Service: IMService[iMessage] Login:
E:snakeninny@gmail.com Active: YES LoginStatus:
Connected]"
```

这一步相当于是在图4-11所示的界面上做了登录iMessage的操作。

函数返回了一个登录成功的IMAccount，也就是iMessage账号。接着选择用于收发iMessage的地址，

命令如下：

```
cy# [controller setAliases:@[@"snakeninny@gmail.com"]
onAccount:#0x166e7b30]
1
```

这一步相当于是在图4-12所示的界面上做了选择iMessage地址的操作。

返回值表明操作成功。最后检查一下此账号是否完成了登录过程，如下：

```
cy# [#0x166e7b30 CNFRegSignInComplete]
1
```

返回值表明已完成了iMessage账号的登录。

很简单吧？不用我再解释什么了吧？作为本节的练习，下面请自行把刚才登录iMessage的Cycript语言翻译成Objective-C语言，并编写一个tweak来验证你的翻译是否正确，好好体会一下Cycript的用法。注意，Apple ID的用户名和密码要改成你自己的！

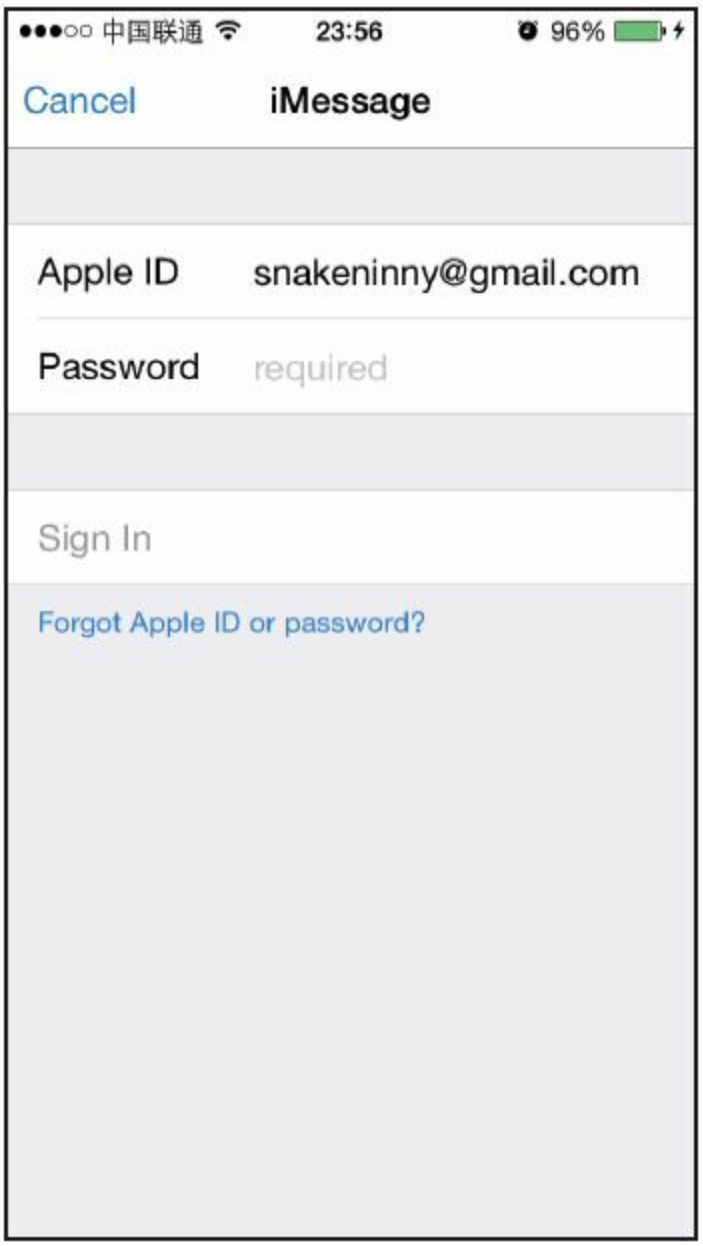


图4-11　登录iMessage

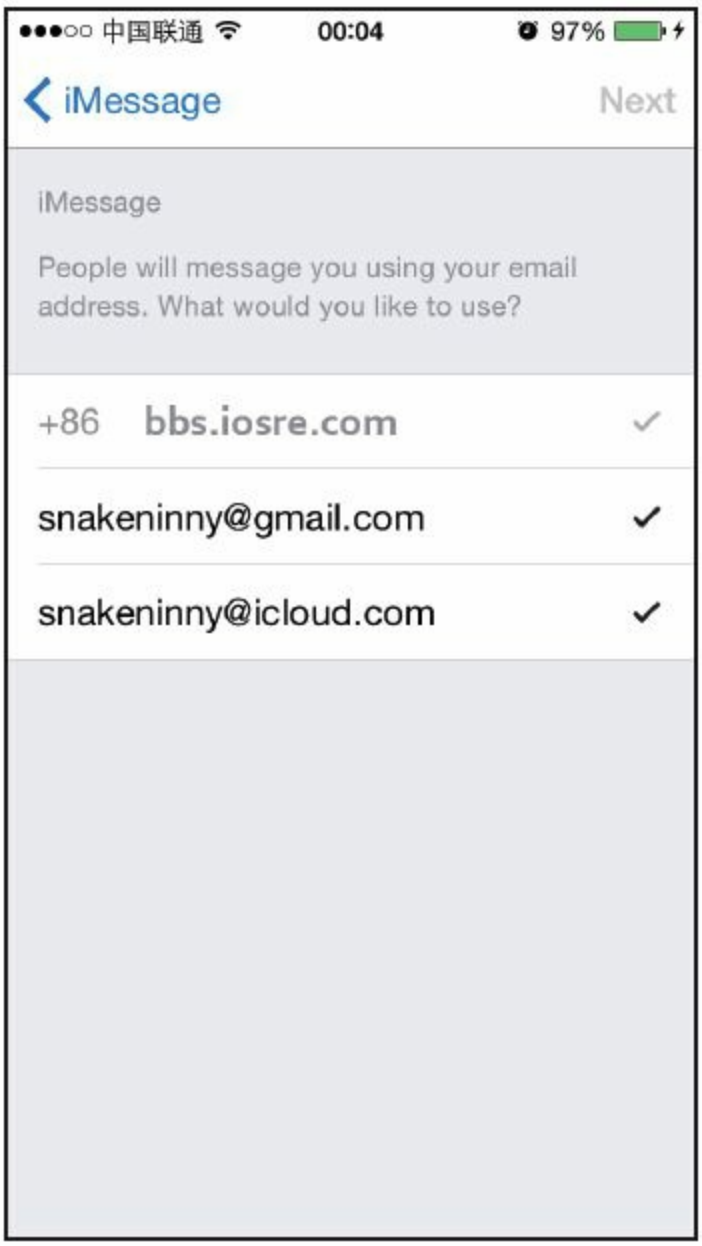


图4-12　选择iMessage地址

4.3 LLDB与debugserver

4.3.1 LLDB简介

如果说IDA是倚天剑，那么LLDB就是屠龙刀，两者在iOS逆向工程中的地位不相上下，难分伯仲。LLDB全称为“Low Level Debugger”，是由苹果出品，内置于Xcode中的动态调试工具，不但通吃C、C++、Objective-C，还全盘支持OSX、iOS，以及iOS模拟器。LLDB的功能可以概括为以下四点：

· 在指定的条件下启动程序；

· 在指定的条件下停止程序；

· 在程序停止的时候检查程序内部发生的事；

· 在程序停止的时候对程序进行改动，观察程序的执行过程有什么变化。

LLDB没有图形界面，使用时看着Terminal中黑压压的一片文字，初学者很容易被吓跑，但是一旦掌握其基本用法，配合IDA双管齐下，就能解决很多难倒大片初学者的问题，投资回报极高。LLDB是运行在OSX中的，要想调试iOS，还需要另一个工具的配合，它就是debugserver。

4.3.2 debugserver简介

debugserver运行在iOS上，顾名思义，它作为服务端，实际执行LLDB（作为客户端）传过来的命令，再把执行结果反馈给LLDB，显示给用户，即所谓的“远程调试”。在默认情况下，iOS上并没有安装debugserver，只有在设备连接过一次Xcode，并在Window→Devices菜单中添加此设备后，debugserver才会被Xcode安装到iOS的“/Developer/usr/bin/”目录下。

但是，因为缺少task_for_pid权限，通过Xcode安

装的debugserver只能调试我们自己的App——调试自己的App是正向开发的事儿，而我们是想搞逆向工程，我们有自己App的源代码，还需要逆哪门子向？要能debug别人的App才够给力啊！别担心，下面就以笔者的操作为例，看看怎么配置debugserver+LLDB，动态调试别人的App，发挥它们在逆向工程中的真正威力。

4.3.3 配置debugserver

1.帮debugserver减肥

对照表4-1，记下设备的ARM信息。

表4-1 支持iOS 8的设备一览

Name	ARM	Name	ARM
iPhone 4s	armv7	The New iPad	armv7
iPhone 5	armv7s	iPad with Retina display	armv7s
iPhone 5c	armv7s	iPad Air	arm64
iPhone 5s	arm64	iPad Air 2	arm64
iPhone 6 Plus	arm64	iPad mini with Retina display	arm64
iPhone 6	arm64	iPad mini 3	arm64
iPad 2	armv7	iPod touch 5	armv7
iPad mini	armv7		

笔者的设备是iPhone 5，对应的ARM是armv7s。将未经处理的debugserver从iOS拷贝到OSX中的“/Users/snakeninny/”目录下，命令如下：

```
snakeninnysiMac:~ snakeninny$ scp
root@iOSIP:/Developer/usr/bin/debugserver ~/debugserver
```

然后帮它减肥，命令如下：

```
snakeninnysiMac:~ snakeninny$ lipo -thin armv7s
~/debugserver -output ~/debugserver
```

注意把这里的“armv7s”换成你的设备所对应的ARM。

2.给debugserver添加task_for_pid权限

下载http://iosre.com/ent.xml到OSX的“/Users/snakeninny/”目录，然后运行如下命令：

```
snakeninnysiMac:~ snakeninny$ /opt/theos/bin/ldid -
Sent.xml debugserver
```

注意，“-S”选项与“ent.xml”之间是没有空格的。

正常情况下，上面这条命令会在5秒内执行完毕。如果ldid卡住了，执行超时，就换一种方案：下载http://iosre.com/ent.plist到“/Users/snakeninny/”，然后运行如下命令：

```
snakeninnysiMac:~ snakeninny$ codesign -s - --
entitlements ent.plist -f debugserver
```

3.将经过处理的debugserver拷回iOS

将经过处理的debugserver拷回iOS，并添加执行权限，命令如下：

```
snakeninnysiMac:~ snakeninny$ scp ~/debugserver
root@iOSIP:/usr/bin/debugserver
snakeninnysiMac:~ snakeninny$ ssh root@iOSIP
FunMaker-5:~ root# chmod +x /usr/bin/debugserver
```

这里之所以把处理过的debugserver存放在iOS的“/usr/bin/”下，而没有覆盖“/Developer/usr/bin/”下的原版debugserver，一是因为原版debugserver是不可写

的，无法覆盖；二是因为“/usr/bin/”下的命令无须输入全路径就可以执行，即在任意目录下运行“debugserver”都可启动处理过的debugserver。

4.3.4 用debugserver启动或附加进程

debugserver最常用的2种场景，就是启动和附加进程，它们的命令都很简单，分别是：

```
debugserver -x backboard IP:port /path/to/executable
```

debugserver会启动executable，并开启port端口，等待来自IP的LLDB接入。

```
debugserver IP:port -a "ProcessName"
```

debugserver会附加ProcessName，并开启port端口，等待来自IP的LLDB接入。

例如：

```
FunMaker-5:~ root# debugserver -x backboard *:1234
/Applications/MobileSMS.app/MobileSMS
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-
320.2.89
 for armv7.
Listening to port 1234 for a connection from *...
```

上面的代码会启动MobileSMS，并开启1234端口，等待任意IP地址的LLDB接入。而：

```
FunMaker-5:~ root# debugserver 192.168.1.6:1234 -a
"MobileSMS"
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-
320.2.89
 for armv7.
Attaching to process MobileNotes...
Listening to port 1234 for a connection from
192.168.1.6...
```

会附加MobileSMS，并开启1234端口，等待来自192.168.1.6的LLDB接入。

如果上面的命令在执行时报错，如下：

```
FunMaker-5:~ root# debugserver *:1234 -a "MobileSMS"
dyld: Library not loaded:
/Developer/Library/PrivateFrameworks/ARMDisassembler.fram

  Referenced from: /usr/bin/debugserver
  Reason: image not found
Trace/BPT trap: 5
```

说明iOS上的“/Developer/”目录下缺少必要的调试数据。这种情况一般是因为没有在Xcode的Window→Devices菜单中添加此设备，重新添加设备就可以解决问题。

当退出debugserver时，当前调试的进程也会一并退出。debugserver的配置到此结束，接下来的所有操作都是在LLDB上完成的。

4.3.5 LLDB的使用说明

在了解LLDB的用法之前，需要对LLDB的一个大Bug有所了解：Xcode 6所附带的LLDB（版本号320.x.xx）在armv7和armv7s设备上有时会混淆ARM和THUMB指令，根本无法调试，且在本书截稿之时，此Bug仍未得到修复。一个暂时的解决方案是从https://developer.apple.com/downloads/index.action下载安装Xcode 5.0.1或Xcode 5.0.2，它们所附带的LLDB（版本号300.x.xx）可以正常调试armv7和

armv7s设备。在安装旧版Xcode的时候，注意将其安装在与当前Xcode不同的路径下，如/Applications/OldXcode.app，这样就不会影响当前的Xcode了。在启用LLDB时，在Terminal中输入如下命令：

```
snakeninnysiMac:~ snakeninny$
/Applications/OldXcode.app/Contents/Developer/usr/bin/lld
```

即可启动旧版LLDB，然后用LLDB连接正在等待的debugserver，命令如下：

```
(lldb) process connect connect://iOSIP:1234
Process 790987 stopped
* thread #1: tid = 0xc11cb, 0x3995b4f0
libsystem_kernel.dylib`mach_msg_trap + 20, queue =
'com.apple.main-thread, stop reason = signal SIGSTOP
    frame #0: 0x3995b4f0
libsystem_kernel.dylib`mach_msg_trap + 20
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x3995b4f0:  pop    {r4, r5, r6, r8}
   0x3995b4f4:  bx     lr
libsystem_kernel.dylib`mach_msg_overwrite_trap:
   0x3995b4f8:  mov    r12, sp
   0x3995b4fc:  push   {r4, r5, r6, r8}
```

注意，“process connect connect://iOSIP:1234”的

执行耗时较长，在WiFi条件下一般需要3分钟以上时间，请耐心等待。在4.6节里，会有通过USB连接调试的介绍，届时速度会大幅增加。当进程停下来时，就可以正式开始调试了。接下来看看常用的LLDB命令有哪些。

1.image list

“image list”与GDB中的“info shared”类似，用于列举当前进程中的所有模块（image）。因为ASLR（Address Space Layout Randomization，详见http://theiphonewiki.com/wiki/ASLR）的关系，每次进程启动时，同一进程的所有模块在虚拟内存中的起始地址都会产生随机偏移。

举个简单的例子，进程A中有一个模块B，B模块的大小是100字节。进程A第一次启动时，模块B可能会被加载到虚拟内存的0x00到0x64，第二次启动被加载到0x10到0x74，第三次被加载到0x60到0xC4，也就

是说它的大小虽然未变，但起始地址每次都在变，然而这个起始地址恰恰是接下来会频繁用到的一个关键数据。那么问题来了，如何获得这个数据呢？

答案就是使用“image list-o-f”命令。待LLDB连接debugserver后，输入“image list-o-f”命令，输出如下：

```
(lldb) image list -o -f
[  0] 0x000cf000
/private/var/db/stash/_.29LMeZ/Applications/SMSNinja.app/

[  1] 0x0021a000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000

[  2] 0x01645000
/usr/lib/libobjc.A.dylib(0x00000000307b5000)
[  3] 0x01645000
/System/Library/Frameworks/Foundation.framework/Foundatio
 (0x0000000023c4f000)
[  4] 0x01645000
/System/Library/Frameworks/CoreFoundation.framework/CoreF

[  5] 0x01645000
/System/Library/Frameworks/UIKit.framework/UIKit
(0x00000000264c1000)
[  6] 0x01645000
/System/Library/Frameworks/CoreGraphics.framework/CoreGra
 (0x0000000023238000)
……
 [235] 0x01645000
/System/Library/Frameworks/CoreGraphics.framework/Resourc

[236] 0x0008a000 /usr/lib/dyld(0x000000001fe8a000)
```

在上面的输出内容中，第一列[X]是模块的序

号；第二列是模块在虚拟内存中的起始地址因ASLR而产生的随机偏移（以下简称ASLR偏移）；第三列是模块的全路径，括号里是偏移之后的起始地址。各种偏移，各种地址，是不是把你绕晕了？没关系，看一个简单的示例你就全明白了。

假设虚拟内存是一个靶场，有1000个靶位。进程的模块是靶子，一共有600个靶子。1000个靶位只摆了600个靶子，这些靶子均匀地排成一横排，靶位1放着靶子1，靶位2放着靶子2，依此类推，靶位600放着靶子600，而靶位601到1000是空着的，如图4-13所示（上面是靶位号，下面是靶子号）。

图4-13　靶场（1）

模块在内存中的起始地址，就是靶子所在的靶位，术语叫模块基地址（image base address）。现在

靶场觉得这样的靶子排列过于简单，打靶的人在适应靶子排列规律后，很容易百发百中，因此将每块靶子往后随机移动了若干靶位，移动之后靶位5放着靶子1，靶位6放着靶子2，靶位8放着靶子3，靶位13放着靶子4，靶位15放着靶子5……靶位886放着靶子600，如图4-14所示。

图4-14　靶场（2）

也就是靶子1偏移了4个靶位，靶子2偏移了4个靶位，靶子3偏移了5个靶位，靶子4偏移了9个靶位，靶子5偏移了10个靶位，靶子600偏移了286个靶位——这种随机偏移（ASLR）大大增加了打靶的难度。对于靶子1来说，偏移前的基地址是靶位1，偏移后的基地址是靶位5，而偏移的值是4个靶位，即

```
偏移后模块基地址 = 偏移前模块基地址 + ASLR偏移
```

回到逆向工程的场景里来，以刚才“image list-o-f”输出中的第4个模块（即Foundation）为例，它的ASLR偏移是0x1645000，偏移后模块基地址是0x23c4f000，所以它的偏移前模块基地址是0x23c4f000-0x1645000=0x2260A000。

0x2260A000是哪里来的呢？把Foundation二进制文件拖到IDA里，初始分析完成后的界面如图4-15所示。

图4-15　在IDA中分析Foundation

把IDA View-A拉到最上面，看到第一行的“HEADER:2260A000”了吗？这就是0x2260A000的

来源。

既然明白了“基地址”的意思是“起始地址”，那么趁热打铁，了解一下与“模块基地址”相似的另一概念：“符号基地址（symbol base address）”。回到IDA，在Functions window里搜索“NSLog”，然后跳转到它的实现，如图4-16所示。

```
__text:2261AB94
__text:2261AB94                 EXPORT _NSLog
__text:2261AB94 _NSLog                                  ; CODE XREF:
__text:2261AB94                                         ; -[NSLock lo
__text:2261AB94
__text:2261AB94 var_18          = -0x18
__text:2261AB94
__text:2261AB94                 SUB             SP, SP, #0xC
__text:2261AB96                 PUSH            {R7,LR}
__text:2261AB98                 MOV             R7, SP
__text:2261AB9A                 SUB             SP, SP, #4
__text:2261AB9C                 ADD.W           R9, R7, #8
__text:2261ABA0                 STMIA.W         R9, {R1-R3}
__text:2261ABA4                 ADD.W           R1, R7, #8
__text:2261ABA8                 STR             R1, [SP,#0x18+var_18]
__text:2261ABAA                 BL              _NSLogv
__text:2261ABAE                 ADD             SP, SP, #4
__text:2261ABB0                 POP.W           {R7,LR}
__text:2261ABB4                 ADD             SP, SP, #0xC
__text:2261ABB6                 BX              LR
__text:2261ABB6 ; End of function _NSLog
```

图4-16　NSLog

因为Foundation的基地址是已知的，而NSLog函数在Foundation中的位置是固定的，所以可以根据下面的公式，得出NSLog的基地址：

```
NSLog的基地址 = NSLog在Foundation中的相对位置 + Foundation的基地址
```

那“NSLog函数在Foundation中的相对位置”又是什么呢？回到图4-16，NSLog函数第一条指令“SUB SP,SP,#0xC”左边的那个数0x2261AB94，代表NSLog在Foundation中的位置，减去Foundation第一行“HEADER:2260A000”中提取出来的0x2260A000，就是NSLog函数在Foundation中的相对位置，即0x10B94。

因此，NSLog的基地址=0x10B94+0x23c4f000=0x23C5FB94。细心的朋友一定已经发现了，公式

```
偏移后模块基地址 = 偏移前模块基地址 + ASLR偏移
```

稍作修改，就可以用到符号基地址的计算中：

```
偏移后符号基地址 = 偏移前符号基地址 + 符号所在模块的ASLR偏移
```

下面来验证一下。

NSLog的偏移前符号基地址是0x2261AB94，Foundation的ASLR偏移是0x1645000，两者相加正是0x23C5FB94。

举一反三，指令基地址的计算也可以套用上面的公式：

偏移后指令基地址 = 偏移前指令基地址 + 指令所在模块的ASLR偏移

自然，符号基地址=符号对应函数第一条指令的基地址。

在接下来的内容中，会大量用到偏移后基地址，因此必须把这一节的几个概念弄懂，然后记住：偏移前基地址从IDA里看，ASLR偏移从LLDB里看，两者相加就是偏移后基地址。至于看哪里，怎么看，文中也已解释清楚，现在要靠你自己完全掌握了。

2.breakpoint

“breakpoint”与GDB中的“break”类似，用于设置断点。在逆向工程中一般用到的是：

```
b function
```

或

```
br s -a address
```

以及

```
br s -a 'ASLROffset+address'
```

前者在函数的起始位置设置断点，如下面的命令：

```
(lldb) b NSLog
Breakpoint 2: where = Foundation`NSLog, address =
0x23c5fb94
```

后两者在地址处设置断点，如下面的命令：

```
(lldb) br s -a 0xCCCCC
 Breakpoint 5: where =
SpringBoard`___lldb_unnamed_function303$$SpringBoard,
address = 0x000ccccc
(lldb) br s -a '0x6+0x9'
Breakpoint 6: address = 0x0000000f
```

注意，在输出的“Breakpoint X:”中，这个X是断点的序号，稍后就会用到。当进程停在断点上时，断点所在的那一行代码并未得到执行。

因为逆向工程中的调试涉及的多是汇编代码，所以大多数情况下都是在某一条汇编指令上下断点，在函数上下断点的情况很少。要在汇编指令上下断点，就要知道它的偏移后基地址，前面已经详细讲解过了。我们以在“-[SpringBoard_menuButtonDown:]”函数的第一条指令设置断点为例，演示一下操作流程。

（1）用IDA查看偏移前基地址

在IDA中打开SpringBoard二进制文件，待初始分析结束后切换到Text view，定位到“-

[SpringBoard_menuButtonDown:]”，如图4-17所示。

可以看到，第一条指令“PUSH{R4-R7,LR}”的偏移前基地址是0x17730。

（2）用LLDB查看ASLR偏移

先ssh到iOS中配置debugserver，命令如下：

```
snakeninnysiMac:~ snakeninny$ ssh root@iOSIP
FunMaker-5:~ root# debugserver *:1234 -a "SpringBoard"
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-
320.2.89
 for armv7.
Attaching to process SpringBoard...
Listening to port 1234 for a connection from *...
```

```
__text:00017730 ; SpringBoard - (void)_menuButtonDown:(struct __IOHIDEvent *)
__text:00017730 ; Attributes: bp-based frame
__text:00017730
__text:00017730 ; void __cdecl -[SpringBoard _menuButtonDown:](struct SpringB
__text:00017730 __SpringBoard__menuButtonDown__         ; DATA XREF: __objc_c
__text:00017730
__text:00017730 var_68          = -0x68
__text:00017730 var_64          = -0x64
__text:00017730 var_60          = -0x60
__text:00017730 var_5C          = -0x5C
__text:00017730 var_58          = -0x58
__text:00017730 var_54          = -0x54
__text:00017730 var_50          = -0x50
__text:00017730 var_4C          = -0x4C
__text:00017730 var_48          = -0x48
__text:00017730 var_44          = -0x44
__text:00017730 var_40          = -0x40
__text:00017730 var_3C          = -0x3C
__text:00017730 var_38          = -0x38
__text:00017730 var_34          = -0x34
__text:00017730 var_30          = -0x30
__text:00017730 var_2C          = -0x2C
__text:00017730 var_28          = -0x28
__text:00017730 var_24          = -0x24
__text:00017730 var_20          = -0x20
__text:00017730 var_1C          = -0x1C
__text:00017730
__text:00017730                 PUSH            {R4-R7,LR}
__text:00017732                 ADD             R7, SP, #0xC
```

图4-17　[SpringBoard_menuButtonDown:]

然后在OSX中用LLDB远程连接，并查看ASLR偏移，命令如下：

```
snakeninnysiMac:~ snakeninny$
/Applications/OldXcode.app/Contents/Developer/usr/bin/lld

(lldb) process connect connect://iOSIP:1234
Process 93770 stopped
* thread #1: tid = 0x16e4a, 0x30dee4f0
libsystem_kernel.dylib`mach_msg_trap + 20, queue =
'com.apple.main-thread, stop reason = signal SIGSTOP
    frame #0: 0x30dee4f0
libsystem_kernel.dylib`mach_msg_trap + 20
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x30dee4f0:  pop    {r4, r5, r6, r8}
   0x30dee4f4:  bx     lr
libsystem_kernel.dylib`mach_msg_overwrite_trap:
   0x30dee4f8:  mov    r12, sp
   0x30dee4fc:  push   {r4, r5, r6, r8}
(lldb) image list -o -f
[0] 0x000b5000
/System/Library/CoreServices/SpringBoard.app/SpringBoard
(0x00000000000b9000)
[1] 0x006ea000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000

[2] 0x01645000
/System/Library/PrivateFrameworks/StoreServices.framework
 (0x000000002ca70000)
[3] 0x01645000
/System/Library/PrivateFrameworks/AirTraffic.framework/Ai
 (0x0000000027783000)
……
[419] 0x00041000 /usr/lib/dyld(0x000000001fe41000)
 (lldb) c
Process 93770 resuming
```

SpringBoard模块的ASLR偏移是0xb5000。

（3）设置并触发断点

综上，第一条指令的偏移后基地址是0x17730+0xb5000=0xCC730。在LLDB中输入“br s-a 0xCC730”即可在第一条指令处设下断点，如下：

```
(lldb) br s -a 0xCC730
Breakpoint 1: where =
SpringBoard`___lldb_unnamed_function299$$SpringBoard,
address = 0x000cc730
```

按下设备上的home键，触发断点，如下：

```
(lldb) br s -a 0xCC730
Breakpoint 1: where =
SpringBoard`___lldb_unnamed_function299$$SpringBoard,
address = 0x000cc730
Process 93770 stopped
* thread #1: tid = 0x16e4a, 0x000cc730
SpringBoard`___lldb_unnamed_function299$$SpringBoard,
queue = 'com.apple.main-thread, stop reason = breakpoint
1.1
    frame #0: 0x000cc730
SpringBoard`___lldb_unnamed_function299$$SpringBoard
SpringBoard`___lldb_unnamed_function299$$SpringBoard:
-> 0xcc730:  push   {r4, r5, r6, r7, lr}
   0xcc732:  add    r7, sp, #12
   0xcc734:  push.w {r8, r10, r11}
   0xcc738:  sub    sp, #80
(lldb) p (char *)$r1
(char *) $0 = 0x0042f774 "_menuButtonDown:"
```

当进程停下来之后，可以用“c”命令让进程继续运行。LLDB相较于GDB的一个重大改进，是可以在进程运行的过程中输入LLDB命令。需要注意的是，部分进程（如SpringBoard）在停止一段时间后会因响应超时而自动重启，对于这类进程，要尽量让它维持在运行状态，避免因自动重启而导致调试信息丢失的悲剧发生。

还可以通过“br dis”、“br en”和“br del”系列命令来禁用、启用和删除断点。如果要禁用所有断点（“dis”代表“disable”），命令如下：

```
(lldb) br dis
All breakpoints disabled. (2 breakpoints)
```

禁用某个断点的命令如下：

```
(lldb) br dis 6
1 breakpoints disabled.
```

启用所有断点（“en”代表“enable”）的命令如下：

```
(lldb) br en
All breakpoints enabled. (2 breakpoints)
```

启用某个断点的命令如下：

```
 (lldb) br en 6
1 breakpoints enabled.
```

删除所有断点（“del”代表“delete”）的命令如下：

```
(lldb) br del
About to delete all breakpoints, do you want to do that?:
[Y/n] Y
```

删除某个断点的命令如下：

```
(lldb) br del 8
1 breakpoints deleted; 0 breakpoint locations disabled.
```

另一个非常有用的命令，是指定在某个断点得到

触发的时候，执行预先设置的指令，它的用法如下（假设1号断点位于某个objc_msgSend函数上）：

```
(lldb) br com add 1
```

执行这条命令后，LLDB会要求我们设置一系列指令，以“DONE”结束，如下：

```
Enter your debugger command(s).  Type 'DONE' to end.
> po [$r0 class]
> p (char *)$r1
> c
> DONE
```

这里输入了3条指令，1号断点一旦触发，就会顺序执行它们，如下：

```
(lldb) c
Process 97048 resuming
__NSArrayM
(char *) $11 = 0x26c6bbc3 "count"
Process 97048 resuming
Command #3 'c' continued the target.
```

“br com add”命令一般用于自动观察某个断点被触发时其上下文的变化，找到进一步分析的线索，在

本书的后半部分，将会看到它的使用场景。

3.print

LLDB的主要功能之一是“在程序停止的时候检查程序内部发生的事”，而这个功能正是通过“print”命令完成的，它可以打印某处的值。仍然以“-[SpringBoard_menuButtonDown:]”里的指令为例，演示它的一系列用法，如图4-18所示。

已知“MOVS R6,#0”的偏移后基地址为0xE37DE，在这条指令上下一个断点，待断点被触发后，看看当前R6的值，如下：

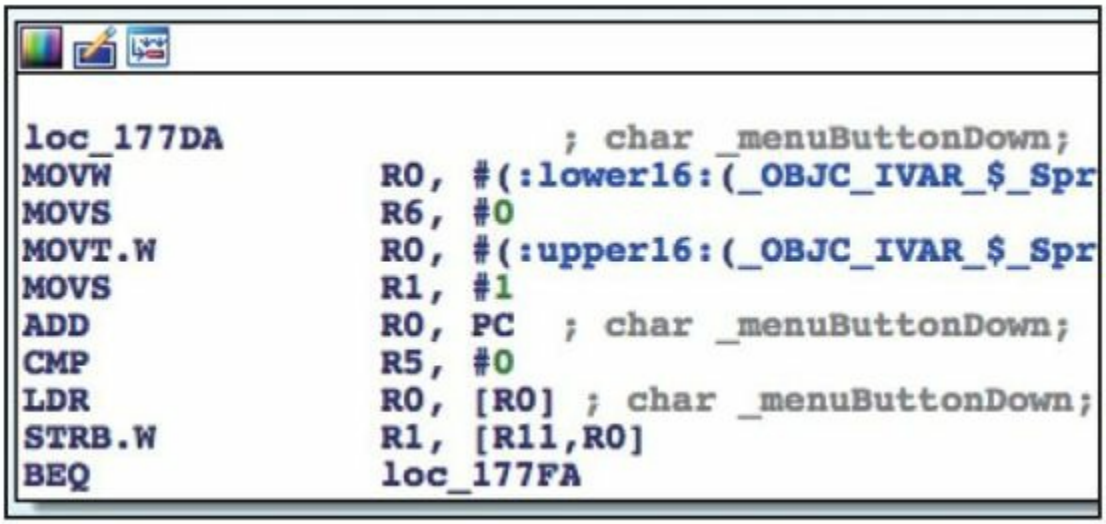


图4-18　[SpringBoard_menuButtonDown:]

```
(lldb) br s -a 0xE37DE
Breakpoint 2: where =
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
174, address = 0x000e37de
Process 99787 stopped
* thread #1: tid = 0x185cb, 0x000e37de
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
174, queue = 'com.apple.main-thread, stop reason =
breakpoint 2.1
    frame #0: 0x000e37de
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
174
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
174:
-> 0xe37de:  movs   r6, #0
   0xe37e0:  movt   r0, #75
   0xe37e4:  movs   r1, #1
   0xe37e6:  add    r0, pc
(lldb) p $r6
(unsigned int) $1 = 364526080
```

此条指令执行之后，R6应该被置0。输入“ni”执行此条指令，再次查看R6的值，如下：

```
(lldb) ni
Process 99787 stopped
* thread #1: tid = 0x185cb, 0x000e37e0
SpringBoard`___lldb_unnamed_function299 $$SpringBoard +
176, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x000e37e0
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
176
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
176:
-> 0xe37e0:  movt   r0, #75
   0xe37e4:  movs   r1, #1
   0xe37e6:  add    r0, pc
```

```
   0xe37e8:  cmp    r5, #0
(lldb) p $r6
(unsigned int) $2 = 0
(lldb) c
Process 99787 resuming
```

可以看到，“p”命令将R6的值正确打印了出来。

在Objective-C中，[someObject someMethod]的底层实现，实际是objc_msgSend(someObject,someMethod)，其中，前者是一个Objective-C对象，后者则可以强制转换成一个字符串（第6章将详细讲解这些内容）。在图4-19中，“BLX_objc_msgSend”执行了[SBTelephonyManager sharedTelephonyManaer]。

```
MOV     R2, #(classRef_SBTelephonyManager - 0x178A0)
ADD     R0, PC ; selRef_sharedTelephonyManager
ADD     R2, PC ; classRef_SBTelephonyManager
LDR     R1, [R0] ; "sharedTelephonyManager"
LDR     R0, [R2] ; _OBJC_CLASS_$_SBTelephonyManager
BLX     _objc_msgSend
```

图4-19　还原objc_msgSend

已知“BLX_objc_msgSend”的偏移后地址是0xCC8A2，在上面下一个断点，待触发后打印

出“objc_msgSend”的参数，如下：

```
(lldb) br s -a 0xCC8A2
Breakpoint 1: where =
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
370, address = 0x000cc8a2
Process 103706 stopped
* thread #1: tid = 0x1951a, 0x000cc8a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
370, queue = 'com.apple.main-thread, stop reason =
breakpoint 1.1
    frame #0: 0x000cc8a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
370
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
370:
-> 0xcc8a2:  blx    0x3e3798            ; symbol stub
for: objc_msgSend
   0xcc8a6:  mov    r6, r0
   0xcc8a8:  movw   r0, #31088
   0xcc8ac:  movt   r0, #74
(lldb) po [$r0 class]
SBTelephonyManager
(lldb) po $r0
SBTelephonyManager
(lldb) p (char *)$r1
(char *) $2 = 0x0042eee6 "sharedTelephonyManager"
(lldb) c
Process 103706 resuming
```

可以看到，用“po”命令打印了Objective-C对象，用“p(char*)”通过强制转换的方式打印了C语言基本数据类型对象，简单明了。需要注意的是，当进程停在某一条“BL”指令上时，LLDB会自动解析这条指令，把指令中地址对应的符号注释出来，如上例中的

```
-> 0xcc8a2:  blx    0x3e3798                  ; symbol
stub for: objc_msgSend
```

但是，LLDB的解析有时会出错，注释出的符号不对。这种情况下，请以IDA静态解析出的符号为准。

最后，可以用“x”命令打印一个地址处存放的值，如下：

```
(lldb) p/x $sp
(unsigned int) $4 = 0x006e838c
(lldb) x/10 $sp
0x006e838c: 0x00000000 0x22f2c975 0x00000000 0x00000000
0x006e839c: 0x26c6bf8c 0x0000000c 0x17a753c0 0x17a753c8
0x006e83ac: 0x000001c8 0x17a75200
(lldb) x/10 0x006e838c
0x006e838c: 0x00000000 0x22f2c975 0x00000000 0x00000000
0x006e839c: 0x26c6bf8c 0x0000000c 0x17a753c0 0x17a753c8
0x006e83ac: 0x000001c8 0x17a75200
```

上面用“p/x”以十六进制方式打印了SP，它是一个指针，值为0x6e838c。而“x/10”则打印出了这个指针指向的连续10个字（word）的数据。

4.nexti与stepi

“nexti”与“stepi”的作用都是执行下一条机器指令，它们最大的区别是前者不进入函数体，而后者会进入函数体。它们可分别简写为“ni”与“si”，是调试时使用最多的指令之一。你可能会问，“进入或者不进入函数体”，是什么意思？这里举个“-[SpringBoard_menuButtonDown:]”里的例子来说明，如图4-20所示。

“BL__SpringBoard__accessibilityObjectWithinProxi偏移后基地址是0xEE92E，它调用了_SpringBoard__accessibilityObjectWithinProximity__0函数。在它上面下断点，然后使用“ni”命令，如下：

图4-20　[SpringBoard_menuButtonDown:]

```
(lldb) br s -a 0xEE92E
Breakpoint 2: where =
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510, address = 0x000ee92e
Process 731 stopped
```

```
* thread #1: tid = 0x02db, 0x000ee92e
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510, queue = 'com.apple.main-thread, stop reason =
breakpoint 2.1
    frame #0: 0x000ee92e
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510:
-> 0xee92e:  bl     0x2fd654                  ;
___lldb_unnamed_function16405$$SpringBoard
   0xee932:  tst.w  r0, #255
   0xee936:  beq    0xee942                   ;
___lldb_unnamed_function299$$SpringBoard + 530
   0xee938:  blx    0x403f08                  ; symbol
stub for: BKSHIDServicesResetProximityCalibration
(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee932
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
514, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x000ee932
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
514
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
514:
-> 0xee932:  tst.w  r0, #255
   0xee936:  beq    0xee942                   ;
___lldb_unnamed_function299$$SpringBoard + 530
   0xee938:  blx    0x403f08                  ; symbol
stub for: BKSHIDServicesResetProximityCalibration
   0xee93c:  movs   r0, #0
 (lldb) c
Process 731 resuming
```

可见，“ni”没有进入_SpringBoard__accessibilityObjectWithinProximity__0函数体。再来看看“si”，如下：

```
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee92e
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510, queue = 'com.apple.main-thread, stop reason =
breakpoint 2.1
    frame #0: 0x000ee92e
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
510:
-> 0xee92e:  bl     0x2fd654                  ;
___lldb_unnamed_function16405$$SpringBoard
   0xee932:  tst.w  r0, #255
   0xee936:  beq    0xee942                   ;
___lldb_unnamed_function299$$SpringBoard + 530
   0xee938:  blx    0x403f08                  ; symbol
stub for: BKSHIDServicesResetProximityCalibration
(lldb) si
Process 731 stopped
* thread #1: tid = 0x02db, 0x002fd654
SpringBoard`___lldb_unnamed_function16405$$SpringBoard,
queue = 'com.apple.main-thread, stop reason = instruction
step into
    frame #0: 0x002fd654
SpringBoard`___lldb_unnamed_function16405$$SpringBoard
SpringBoard`___lldb_unnamed_function16405$$SpringBoard:
-> 0x2fd654:  movw   r0, #33920
   0x2fd658:  movt   r0, #43
   0x2fd65c:  add    r0, pc
   0x2fd65e:  ldrsb.w r0, [r0]
(lldb) c
Process 731 resuming
```

“movw r0,#33920”的偏移前基地址是0x226654，在IDA中如图4-21所示。

```
__text:00226654
__text:00226654 ; -[SpringBoard _accessibilityObjectWithinProximity]_0
__text:00226654 __SpringBoard__accessibilityObjectWithinProximity__0
__text:00226654                                 ; CODE XREF: -[SpringBoard _menuButtonDown:]:loc_1792E↑p
__text:00226654                                 ; -[SpringBoard _menuButtonDown:]+2AC↑p ...
__text:00226654                 MOV             R0, #(byte_4DEAE0 - 0x226660)
__text:0022665C                 ADD             R0, PC ; byte_4DEAE0
__text:0022665E                 LDRSB.W         R0, [R0]
__text:00226662                 BX              LR
__text:00226662 ; End of function -[SpringBoard _accessibilityObjectWithinProximity]_0
```

图4-21

SpringBoard__accessibilityObjectWithinProximity__0

其位于_SpringBoard__accessibilityObjectWithinProximity__0函数内部，即“si”命令进入了函数体，这就是“进入或者不进入函数体”的意思。

5.register write

“register write”命令用于给指定的寄存器赋值，从而“对程序进行改动，观察程序的执行过程有什么变化”。在图4-22所示的代码中，已知“TST.W R0,#0xFF”的偏移后基地址是0xEE7A2，如果R0的值是0，进程会走左边的分支，否则会走右边的分支。

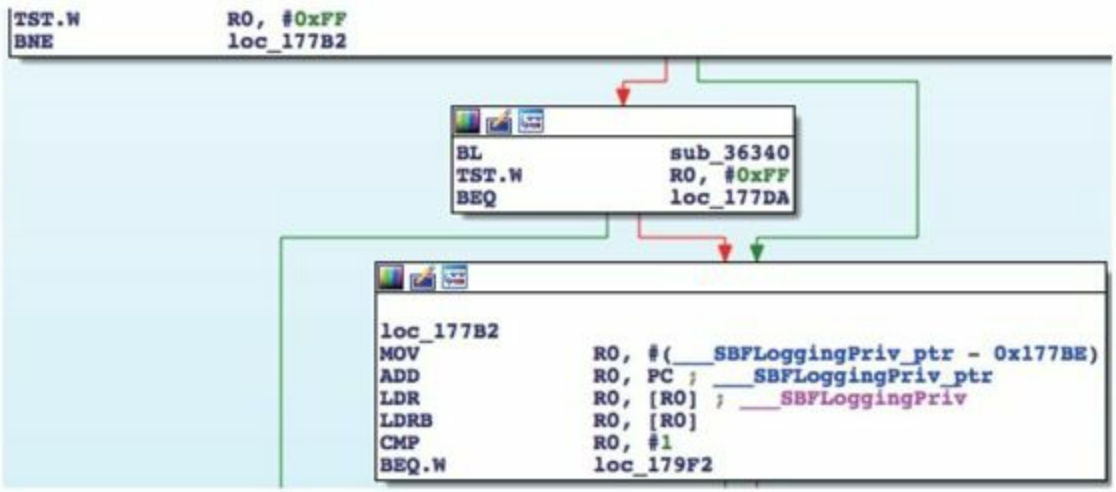


图4-22　分支

这里下个断点看看这里R0的值是多少，如下：

```
(lldb) br s -a 0xEE7A2
Breakpoint 3: where =
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114, address = 0x000ee7a2
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114, queue = 'com.apple.main-thread, stop reason =
breakpoint 3.1
    frame #0: 0x000ee7a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114:
-> 0xee7a2:  tst.w  r0, #255
   0xee7a6:  bne    0xee7b2                    ;
___lldb_unnamed_function299$$SpringBoard + 130
   0xee7a8:  bl     0x10d340                   ;
___lldb_unnamed_function1110$$SpringBoard
   0xee7ac:  tst.w  r0, #255
(lldb) p $r0
(unsigned int) $0 = 0
```

由于R0的值是0，因此在BNE的作用下，它会走左边的分支，如下：

```
(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a6
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
118, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x000ee7a6
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
118
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
118:
-> 0xee7a6:  bne    0xee7b2                  ;
___lldb_unnamed_function299$$SpringBoard + 130
   0xee7a8:  bl     0x10d340                 ;
___lldb_unnamed_function1110$$SpringBoard
   0xee7ac:  tst.w  r0, #255
   0xee7b0:  beq    0xee7da                  ;
___lldb_unnamed_function299$$SpringBoard + 170
(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a8
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
120, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x000ee7a8
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
120
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
120:
-> 0xee7a8:  bl     0x10d340                 ;
___lldb_unnamed_function1110$$SpringBoard
   0xee7ac:  tst.w  r0, #255
   0xee7b0:  beq    0xee7da                  ;
___lldb_unnamed_function299$$SpringBoard + 170
   0xee7b2:  movw   r0, #2174
```

再次触发断点，通过“register write”命令更改R0

的值为1，看看它会走哪个分支，如下：

```
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114, queue = 'com.apple.main-thread, stop reason =
breakpoint 3.1
    frame #0: 0x000ee7a2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
114:
-> 0xee7a2:  tst.w  r0, #255
   0xee7a6:  bne    0xee7b2                  ;
___lldb_unnamed_function299$$SpringBoard + 130
   0xee7a8:  bl     0x10d340                 ;
___lldb_unnamed_function1110$$SpringBoard
   0xee7ac:  tst.w  r0, #255
(lldb) p $r0
(unsigned int) $5 = 0
(lldb) register write r0 1
(lldb) p $r0
(unsigned int) $6 = 1
(lldb) ni
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7a6
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
118, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x000ee7a6
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
118
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
118:
-> 0xee7a6:  bne    0xee7b2                  ;
___lldb_unnamed_function299$$SpringBoard + 130
   0xee7a8:  bl     0x10d340                 ;
___lldb_unnamed_function1110$$SpringBoard
   0xee7ac:  tst.w  r0, #255
   0xee7b0:  beq    0xee7da                  ;
___lldb_unnamed_function299$$SpringBoard + 170
(lldb)
Process 731 stopped
* thread #1: tid = 0x02db, 0x000ee7b2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
```

```
130, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x000ee7b2
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
130
SpringBoard`___lldb_unnamed_function299$$SpringBoard +
130:
-> 0xee7b2:  movw   r0, #2174
   0xee7b6:  movt   r0, #63
   0xee7ba:  add    r0, pc
   0xee7bc:  ldr    r0, [r0]
```

此时，进程改道右边的分支了。

LLDB的命令还有很多种，这里只列举了iOS逆向工程初期最常用的五种，希望读者能够窥一斑而见全豹，感受到LLDB的强大威力。LLDB仍处在开发阶段，除了几个官方网站，还未见成熟的教程；LLDB脱胎于GDB，虽然两者的命令有差别，但用法和思路是一脉相承的。要想完整地熟悉LLDB的使用，推荐阅读“Peter’s GDB tutorial”和“RMS’s gdb Debugger Tutorial”（Google一下）。IDA宜静，LLDB宜动，熟练地使用这两个工具是成为逆向高手的必经之路。

4.3.6 LLDB使用小提示

1.调试的二进制文件必须从iOS中提取

IDA分析的二进制文件必须与LLDB调试的二进制文件相同，这样偏移前基地址、ASLR偏移、偏移后基地址才能对应得上。IDA分析的二进制文件可以通过第3章介绍的dyld_decache工具从本机获取；从其他渠道（如SDK、模拟器等）提取的文件一般不能用作动态调试。

2.LLDB中的简化输入

在使用LLDB时，如果想重复执行上一条指令，直接按回车键就可以了；如果想查看以前执行过的指令，按方向键的向上和向下键就可以了。

LLDB的命令都很简单，但怎么用简单的命令去解决复杂的问题，却不简单。在第6章还会列举一些LLDB的常用场景，但在那之前，请大家务必掌握本节的知识。

4.4 dumpdecrypted

前面在介绍class-dump时提到过，从AppStore下载的App（以下简称StoreApp）是被苹果加密过的（从其他渠道下载的一般没有加密），可执行文件被套上了一层保护壳，而class-dump无法作用于加密过的App。在这种情况下，想要获取头文件，需要先解密App的可执行文件，俗称“砸壳”。dumpdecrypted就是由越狱社区的知名人士Stefan Esser（@i0n1c）出品的一款砸壳工具，被越狱社区广泛运用在iOS逆向工程研究中。

dumpdecrypted在GitHub上开源了，得自行编译才能使用。下面就从零开始，以一个虚构的TargetApp.app为例，引导大家进行一次完整的App砸壳，请大家对着电脑，跟着笔者一起操作。

1）从GitHub下载dumpdecrypted源码，命令如下：

```
snakeninnysiMac:~ snakeninny$ cd /Users/snakeninny/Code/
snakeninnysiMac:Code snakeninny$ git clone
git://github.com/stefanesser/dumpdecrypted/
Cloning into 'dumpdecrypted'...
remote: Counting objects: 31, done.
remote: Total 31 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (31/31), 6.50 KiB | 0 bytes/s,
done.
Resolving deltas: 100% (15/15), done.
Checking connectivity... done
```

2）编译dumpdecrypted.dylib，命令如下：

```
snakeninnysiMac:~ snakeninny$ cd
/Users/snakeninny/Code/dumpdecrypted/
snakeninnysiMac:dumpdecrypted snakeninny$ make
`xcrun --sdk iphoneos --find gcc` -Os  -Wimplicit -
isysroot `xcrun --sdk iphoneos --show-sdk-path` -F`xcrun
--sdk iphoneos --show-sdk-path`/System/Library/Frameworks
-F`xcrun --sdk iphoneos --show-sdk-
path`/System/Library/PrivateFrameworks -arch armv7 -arch
armv7s -arch arm64 -c -o dumpdecrypted.o dumpdecrypted.c
`xcrun --sdk iphoneos --find gcc` -Os  -Wimplicit -
isysroot `xcrun --sdk iphoneos --show-sdk-path` -F`xcrun
--sdk iphoneos --show-sdk-path`/System/Library/Frameworks
-F`xcrun --sdk iphoneos --show-sdk-
path`/System/Library/PrivateFrameworks -arch armv7 -arch
armv7s -arch arm64 -dynamiclib -o dumpdecrypted.dylib
dumpdecrypted.o
```

上面的make命令执行完毕后，会在当前目录下生成一个dumpdecrypted.dylib文件，这就是等下砸壳所要用到的榔头。此文件生成一次即可，以后可以重复使用，下次砸壳时无须再重新编译。

3）用ps命令定位待砸壳的可执行文件。在iOS 8中，StoreApp全部位于/var/mobile/Containers/下，其中可执行文件位于/var/mobile/Containers/Bundle/Application/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/TargetApp.app/下。我们不知道X是什么，肉眼定位需要手工遍历所有目录，劳民伤财，但一个简单的小技巧就可以省时省力：首先在iOS中关掉所有StoreApp，然后打开Target，接着ssh到iOS上，打印出所有进程，如下：

```
snakeninnysiMac:~ snakeninny$ ssh root@iOSIP
FunMaker-5:~ root# ps -e
  PID TTY           TIME CMD
    1 ??         3:28.32 /sbin/launchd
……
5717 ??         0:00.21
/System/Library/PrivateFrameworks/MediaServices.framework

 5905 ??         0:00.20 sshd: root@ttys000
 5909 ??         0:01.86
/var/mobile/Containers/Bundle/Application/03B61840-2349-
4559-B28E-0E2C6541F879/TargetApp.app/TargetApp
 5911 ??         0:00.07
/System/Library/Frameworks/UIKit.framework/Support/pasteb

 5907 ttys000    0:00.03 -sh
 5913 ttys000    0:00.01 ps -e
```

因为iOS上只打开了一个StoreApp，所以唯一的那个含有“/var/mobile/Containers/Bundle/Application/”字样的结果就是TargetApp可执行文件的全路径。

4）用Cycript找出TargetApp的Documents目录路径。StoreApp的Documents目录位于/var/mobile/Containers/Data/Application/YYYYYYYY-YYYY-YYYY-YYYY–YYYYYYYYYYYY/下，Y与之前的X值不同，而且这次PS也帮不上忙了。因此，需要借助强大的Cycript，让App告诉我们Documents的路径。命令如下：

```
FunMaker-5:~ root# cycript -p TargetApp
cy# [[NSFileManager defaultManager]
URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask][0]
#"file:///var/mobile/Containers/Data/Application/D41C4343
63AA-4BFF-904B-2146128611EE/Documents/"
```

5）将dumpdecrypted.dylib拷贝到Documents目录下。拷贝命令如下：

```
snakeninnysiMac:~ snakeninny$ scp
/Users/snakeninny/Code/dumpdecrypted/dumpdecrypted.dylib
root@iOSIP:/var/mobile/Containers/Data/Application/D41C43
63AA-4BFF-904B-2146128611EE/Documents/
dumpdecrypted.dylib
100%  193KB 192.9KB/s   00:00
```

这里采用的是scp方式，也可以使用iFunBox等工具来操作。

6）开始砸壳。dumpdecrypted.dylib的用法是：

```
DYLD_INSERT_LIBRARIES=/path/to/dumpdecrypted.dylib
/path/to/executable
```

实际操作起来就是：

```
FunMaker-5:~ root# cd
/var/mobile/Containers/Data/Application/D41C4343-63AA-
4BFF-904B-2146128611EE/Documents/
FunMaker-
5:/var/mobile/Containers/Data/Application/D41C4343-63AA-
4BFF-904B-2146128611EE/Documents root#
DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/var/mobile/Containers/Bundle/Application/03B61840-2349-
4559-B28E-0E2C6541F879/TargetApp.app/TargetApp
mach-o decryption dumper
DISCLAIMER: This tool is only meant for security research
purposes, not for application crackers.
[+] detected 32bit ARM binary in memory.
[+] offset to cryptid found: @0x81a78(from 0x81000) = a78
[+] Found encrypted data at address 00004000 of length
6569984 bytes - type 1.
[+] Opening
```

```
/private/var/mobile/Containers/Bundle/Application/03B6184
2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp for
reading.
[+] Reading header
[+] Detecting header type
[+] Executable is a plain MACH-O image
[+] Opening TargetApp.decrypted for writing.
[+] Copying the not encrypted start of the file
[+] Dumping the decrypted data into the file
[+] Copying the not encrypted remainder of the file
[+] Setting the LC_ENCRYPTION_INFO->cryptid to 0 at
offset a78
[+] Closing original file
[+] Closing dump file
```

当前目录下会生成TargetApp.decrypted，即砸壳后的文件，如下：

```
FunMaker-
5:/var/mobile/Containers/Data/Application/D41C4343-63AA-
4BFF-904B-2146128611EE/Documents root# ls
TargetApp.decrypted  dumpdecrypted.dylib OtherFiles
```

赶紧把砸壳后的文件拷贝到OSX上备用吧，class-dump、IDA等工具已经迫不及待啦。

以上6步还算简洁明了，但可能会有朋友问，为什么要把dumpdecrypted.dylib拷贝到Documents目录下操作？

问得好。我们都知道，StoreApp对沙盒以外的绝大多数目录没有写权限。dumpdecrypted.dylib要写一个decrypted文件，但它是运行在StoreApp中的，与StoreApp的权限相同，那么它的写操作就必须发生在StoreApp拥有写权限的路径下才能成功。StoreApp一定是能写入其Documents目录的，因此在Documents目录下使用dumpdecrypted.dylib时，保证它能在当前目录下写一个decrypted文件，这就是把dumpdecrypted.dylib拷贝到Documents目录下操作的原因。

最后来看看如果不放在Documents目录下，可能会出现什么问题，如下：

```
FunMaker-5:
/var/mobile/Containers/Data/Application/D41C4343-63AA-
4BFF-904B-2146128611EE/Documents root# mv
dumpdecrypted.dylib /var/tmp/
FunMaker-5:
/var/mobile/Containers/Data/Application/D41C4343-63AA-
4BFF-904B-2146128611EE/Documents root# cd /var/tmp
FunMaker-5:/var/tmp root#
DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/private/var/mobile/Containers/Bundle/Application/03B6184
2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp
dyld: could not load inserted library
```

```
'dumpdecrypted.dylib' because no suitable image found.
Did find:
      dumpdecrypted.dylib: stat() failed with errno=1
Trace/BPT trap: 5
```

这里errno的值是1，即“Operation not permitted”，砸壳失败。如果你在使用dumpdecrypted的过程中碰到任何问题，或对它有进一步的研究，都欢迎来http://bbs.iosre.com参与讨论。

4.5 OpenSSH

OpenSSH会在iOS上安装SSH服务（如图4-23所示），以给外界提供一个通过ssh接入iOS的途径。

图4-23 OpenSSH

这里用得最多的一般只有2个命令：ssh和scp，前者用于远程登录，后者用于远程拷贝文件。ssh的用法如下：

```
ssh user@iOSIP
```

如

```
snakeninnysiMac:~ snakeninny$ ssh mobile@192.168.1.6
```

scp的用法如下。

1）把文件从本地拷贝到iOS上，命令如下：

```
scp /path/to/localFile user@iOSIP:/path/to/remoteFile
```

如

```
snakeninnysiMac:~ snakeninny$ scp ~/1.png
root@192.168.1.6:/var/tmp/
```

2）把文件从iOS拷贝到本地，命令如下：

```
scp user@iOSIP:/path/to/remoteFile /path/to/localFile
```

如

```
snakeninnysiMac:~ snakeninny$ scp
root@192.168.1.6:/var/log/syslog ~/iOSlog
```

两种命令的用法都比较简单直观。在安装OpenSSH后需要注意修改默认登录密码"alpine"。iOS上的用户有2个，分别是root和mobile，修改密码的命令如下：

```
FunMaker-5:~ root# passwd root
Changing password for root.
New password:
Retype new password:
FunMaker-5:~ root# passwd mobile
Changing password for mobile.
New password:
Retype new password:
```

如果没有修改默认密码，Ikee等病毒就有可能通过ssh以root用户身份登录iOS，拿到最高权限。这个后果是非常严重的：iOS中的所有数据，包括短信、电话本、AppleID的账号密码等敏感信息泄露的风险

将大大增加，你的设备可能会被入侵者玩弄于股掌之间，为所欲为。因此，在安装OpenSSH之后一定要记得修改默认密码！

4.6 usbmuxd

很多朋友是通过WiFi连接使用SSH服务的，因为无线网络的不稳定性及传输速度的限制，在复制文件或用LLDB远程调试时，iOS的响应很慢，效率不高。iOS越狱社区的知名人士Nikias Bassen（@pimskeks）开发了一款可以把本地OSX/Windows端口转发到远程iOS端口的工具usbmuxd，使我们能够通过USB连接线ssh到iOS中，大大增加了ssh连接的速度，也方便了那些没有WiFi的朋友。它的用法如下，比较简单。

（1）下载并配置usbmuxd

从

http://cgit.sukimashita.com/usbmuxd.git/snapshot/usbmux

1.0.8.tar.gz下载usbmuxd，解压到本地。我们用到的只有python-client目录下的tcprelay.py和usbmux.py两个文件，把它们放到同一个目录下，比如，笔者的是：

```
/Users/snakeninny/Code/USBSSH/
```

（2）使用usbmuxd转发端口

usbmuxd的用法比较简单，在Terminal中输入如下命令：

```
/Users/snakeninny/Code/USBSSH/tcprelay.py -t 远程iOS上的端口:本地OSX/Windows上的端口
```

即可把本地OSX/Windows上的端口转发到远程iOS上的端口。

下面是使用场景举例。

在没有WiFi的情况下，使用USB连接到iOS，用lldb调试SpringBoard。

1）把本地2222端口转发到iOS的22端口，命令如下：

```
snakeninnysiMac:~ snakeninny$
/Users/snakeninny/Code/USBSSH/tcprelay.py -t 22:2222
Forwarding local port 2222 to remote port 22
```

2）ssh到iOS中，并用debugserver附加SpringBoard，命令如下：

```
snakeninnysiMac:~ snakeninny$ ssh root@localhost -p 2222
FunMaker-5:~ root# debugserver *:1234 -a "SpringBoard"
```

3）把本地1234端口转发到iOS的1234端口，命令如下：

```
snakeninnysiMac:~ snakeninny$
/Users/snakeninny/Code/USBSSH/tcprelay.py -t 1234:1234
Forwarding local port 1234 to remote port 1234
```

4）用lldb开始调试，命令如下：

```
snakeninnysiMac:~ snakeninny$
/Applications/OldXcode.app/Contents/Developer/usr/bin/lld

(lldb) process connect connect://localhost:1234
```

使用usbmuxd能极大提升ssh的速度，用LLDB远程连接debugserver的时间被缩短至15秒以内，强烈建议大家把usbmuxd作为ssh连接的首选方案。

4.7 iFile

iFile是iOS上一款非常强大的文件管理App，可以看作是iOS版的Finder，如图4-24所示。它能进行各种文件操作，从最简单的浏览，到编辑、剪贴、复制，还可以安装deb文件，十分方便。

iFile的界面十分直观，无需过多说明。如果要安装deb文件，就要先关掉Cydia，然后在点击deb文件后弹出的选项中选择“Installer”即可，如图4-25所示。

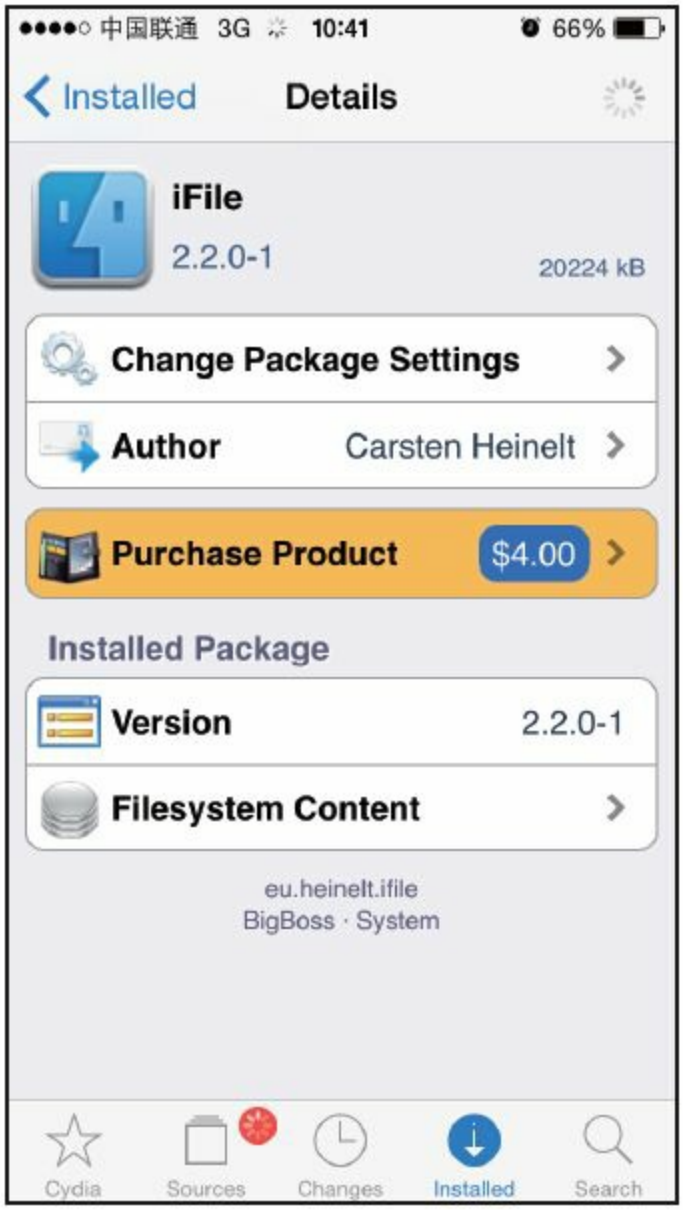

●●●●○ 中国联通 3G 10:41 66%
Installed
Details
iFile
2.2.0-1
20224 kB
Change Package Settings
Author
Carsten Heinelt
Purchase Product
$4.00
Installed Package
Version
2.2.0-1
Filesystem Content
eu.heinelt.ifile
BigBoss · System
Cydia
Sources
Changes
Installed
Search

图4-24　iFile

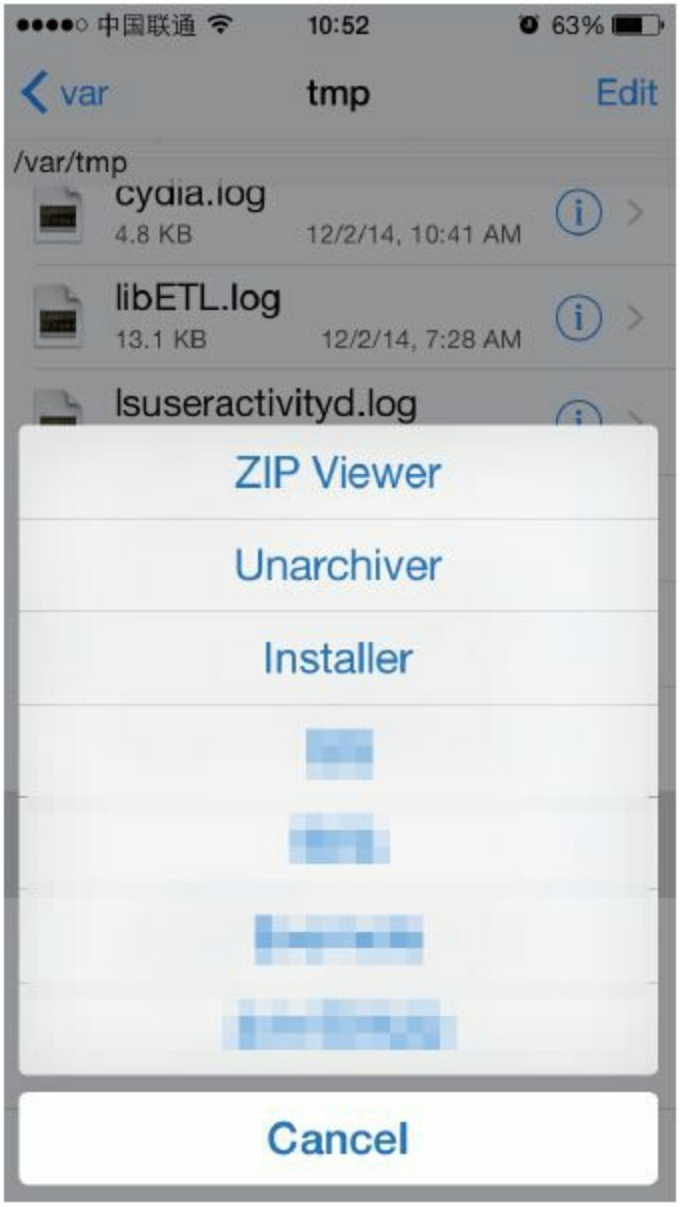


图4-25　安装deb文件

4.8 MTerminal

MTerminal是开源的iOS版Terminal，基本功能一应俱全，如图4-26所示。其用法与Terminal区别不大，只是屏幕和键盘小了点。笔者认为MTerminal最实用的场景是在没有电脑的环境下利用碎片时间，结合Cycript进行代码测试。

图4-26　MTerminal

4.9 syslogd to/var/log/syslog

syslogd是iOS中记录系统日志的守护进程，“syslogd to/var/log/syslog”的作用是把日志给写入“/var/log/syslog”文件中，如图4-27所示。

图4-27　syslogd to/var/log/syslog

在安装完这个插件后要重启（reboot）一次iOS，才会生成“/var/log/syslog”文件。在iOS运行的全过程中

这个文件会变得越来越大，可以通过

```
FunMaker-5:~ root# cat /dev/null > /var/log/syslog
```

命令来将它清空，节省系统容量。

4.10 小结

本章重点介绍了9个工具，其中CydiaSubstrate、LLDB、Cycript是核心中的核心。正是由于这些iOS工具的存在，配合OSX工具集，才构成了一个相对完整的iOS逆向工程环境。“既要知其然，还要知其所以然”，要对工具有所了解，从下一章开始，我们就该进一步学习一些理论了。

第三部分　理论篇

· 第5章　Objective-C相关的iOS逆向理论基础

· 第6章　ARM汇编相关的iOS逆向理论基础

当你从第一部分了解了iOS应用逆向工程的基本概念，并跟着第二部分把玩过一些逆向工具之后，就已经具备了iOS应用逆向工程的基本知识。当你完成了书上的例子之后，接下来可能会有一种无从下手的感觉，不知道下一步该做些什么。逆向工程是一门需要动手的学问，而从哪里动手、该怎么动手，其实是有套路可循的。第5章和第6章分别尝试从Objective-C和ARM的角度出发，用iOS应用逆向工程独有的理论知识把介绍过的工具串联起来，总结出一套通用的逆向工程方法论。这就开始吧！

第5章　Objective-C相关的iOS逆向理论基础

Objective-C语言是一门面向对象的高级语言，想必大家都能较为熟练地掌握它的基本用法，在逆向工程的入门阶段采用Objective-C语言有助于大家更平稳地从App开发进阶到逆向工程。幸运地是，iOS采用的文件格式Mach-O中包含了足够多的原始数据，让我们能够用class-dump等工具还原出二进制文件的头文件；有了这些信息，就可以开始Objective-C级别的逆向工程了，而撰写tweak无疑是这个阶段最受欢迎的项目，下面就从它开始吧！

5.1 tweak在Objective-C中的工作方式

在第3章中介绍Theos时，已经介绍了tweak的概念。依据维基百科的定义，tweak指的是对电子系统进行轻微调整来增强其功能的工具；在iOS中，tweak特指那些能够增强其他进程功能的dylib，是越狱iOS的最重要组成部分。

正是因为tweak的存在，越狱iOS用户才能依照自己的喜好打造独一无二的个性化系统，iOS开发者才有机会站在优秀软件的肩膀上为它们添砖加瓦，丰富它们的功能，而这些便利都是原版iOS和AppStore无法提供的。Cydia中最受欢迎的软件几乎全是创意各异的tweak（图5-1是Cydia中的tweak图标），如Activator、Barrel、SwipeSelection等。一般来说，一个tweak的核心是各种“hook”，而绝大部分的hook是针对Objective-C方法的，那么tweak是如何工作的呢？

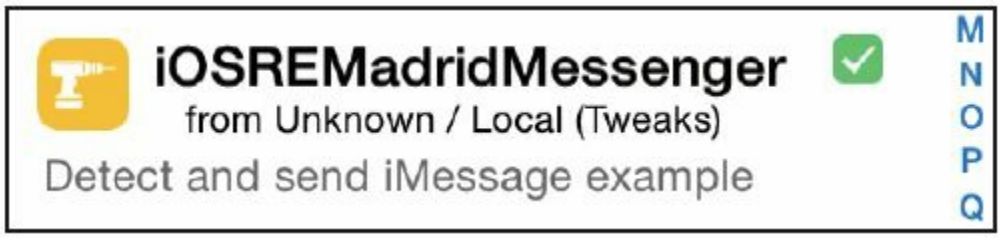


图5-1　tweak图标

Objective-C是典型的面向对象语言。iOS是由一个个小的组件构成的，这些组件其实就是一个个对象。举个例子，iOS里的每个图标、每条信息和每张照片都是对象；除了这些用户能够看到的对象以外，还有很多对象一直在后台工作，为前台对象提供各种支持。例如有些对象负责与苹果的服务器通信，有的对象负责读写文件。一个对象可以拥有其他对象，例如图标对象就拥有一个标签对象，用来显示这个图标代表的App名称。一般来说，每个对象都有自己存在的意义，工程师通过对不同对象的组合排序，实现不同的功能；在Objective-C里，我们称对象的功能为“方法”，“方法”的具体行为则称为“实现”。对象、方法和实现的关系，就是tweak大做文章的地方。

对象具备了某种功能，代码就可以发出指令“[object method]”，让一个对象去执行它的功能，也就是“调用对象的方法”。看到这里，可能有朋友会说，指令里的“对象”和“方法”都是名词，而执行一个功能需要的不应该是一个动词吗？说得没错，我们还缺少一个动词，需要去“实现”这个方法。需要的动词已经出现了——“实现”，它指的是当某个方法得到调用时，iOS实际干了些什么，也就是执行了什么代码。在Objective-C里，方法和实现的关系不是在编译时决定的，而是在运行时决定的。

在实际使用中，“[object method]”中的method不一定是一个名词，它也可能是一个动词。但仅凭简短的[object method]，还是不知道要怎么实现这个方法，比如：“妈妈，接一下电话”，翻译成Objective-C语言是“[妈妈接电话]”，这里的对象是妈妈，方法是“接电话”，实现是“放下手里的炒菜铲子，把炉火关小一点，然后走到客厅去接电话”；“snakeninny，

过来搬个东西”，翻译成Objective-C语言是“[snakeninny搬东西]”，这里的对象是snakeninny，方法是“搬东西”，实现是“停下手里的工作，从椅子上起来，走到老板的办公室里把一个箱子抬到楼下”。上面的两个例子如果没有“实现”的具体描述，即使调用了“方法”，“对象”也不知道具体该干嘛。“实现”是“方法”的释义，“方法”是词语，“实现”是词语的意义——这不就是词典吗？

随着时代的进步，词典的内容产生了变化，一些旧词语被赋予了新解释，“灌水”跟液体已经没有太大关系，“粉丝”也从一种食物变成了一类人。这些现象在iOS中也有体现，我们可以通过改变“实现”和“方法”的对应关系，赋予一个方法新的意义，从而达到更改对象功能的目的；只要别人在查询一个词语意义的时候参考了你修改过的词典，那么他的方法就有了新的实现，例如，笔者开发的LowPowerBanner（如图5-2所示）会在低电量时以横幅代替弹窗，提醒用

户没电了——哈哈，那正是因为笔者更改了低电量提醒的实现，善意地欺骗系统“弹窗”的意思是“横幅”。

笔者的另一个短信防火墙SMSNinja（如图5-3所示）能在收到垃圾短信时将其自动放进垃圾箱，这是通过更改iOS收到短信的动作实现的，在原有基础上增加了检测垃圾短信的功能。这种“更改词典内容”的方式就是通过CydiaSubstrate进行hook操作来实现的。CydiaSubstrate的用法已经在前两章详细介绍过了，想必大家都还记得。

图5-2　LowPowerBanner

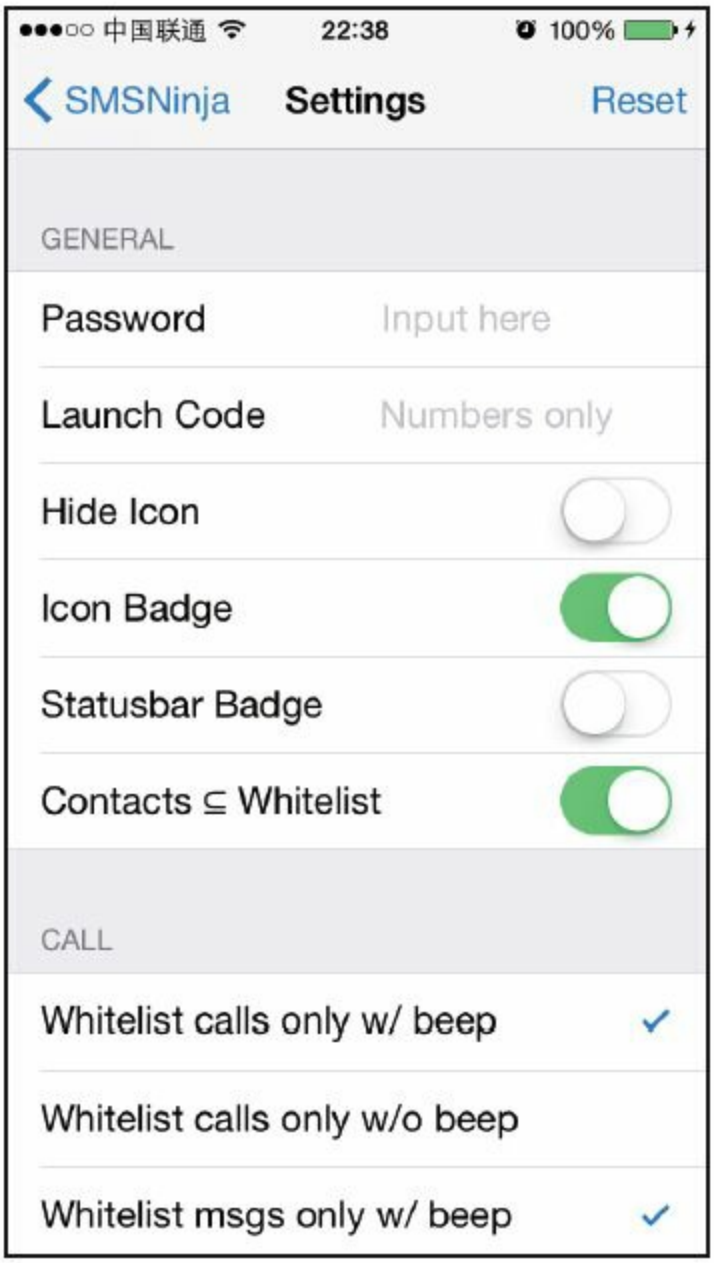


图5-3　SMSNinja

5.2 tweak的编写套路

只有理解了tweak的工作方式，才能在编写tweak时清楚地知道自己想干什么、在干什么。一般来说，编写tweak会用到C、C++和Objective-C三种语言，有了一个灵感时，该如何自如地运用这三种语言把灵感变成一个好用的tweak呢？事实上，编写tweak的思路是有规律可循的，而且随着你对iOS的了解愈加深入，对编程语言的掌握愈加熟练，这种规律会变得越来越明显。下面将围绕一个简单的tweak例子，从iOS工程师使用最多的Objective-C语言开始分析，总结归纳Objective-C级别的逆向工程理论。

5.2.1 寻找灵感

可能有部分iOS工程师读到这里时，已经能够结合前几章的知识开始开发tweak了，但可能也还有部分人感到无从下手，不知道该写些什么东西。这种有

劲儿没处使的感觉确实挺难受，面对这种情况，该怎么办呢？一般情况下，可以从这几个方面找灵感。

1.多使用，多观察

没事就把你的手机拿出来把玩把玩，把系统的每个角落都扫一遍，别光顾着刷朋友圈。虽然iOS的功能已足够多，但也不可能符合每个用户的要求，所以，用得越多，你对iOS的了解就越多，哪些地方用着不爽的感觉就会更强烈。上网看看吧，iOS的用户基数巨大，他们中一定有跟你想法相同的人——你碰到需要解决的实际问题了，这不就是灵感吗？笔者在iOS 6时代开发的Characount for Notes（如图5-4所示）就是这样得来的。当时，笔者经常把微博的内容存成记事本，但微博是有140字限制的，于是就做了一个这样的tweak，用来统计记事本每页的字数，从而控制微博的长度。曾有一位阿拉伯用户还专门给笔者发邮件说很喜欢这个插件，希望加入更多功能把记

事本改造成一个Word，但笔者对这个想法不大感兴趣，所以只好对他说声抱歉了。

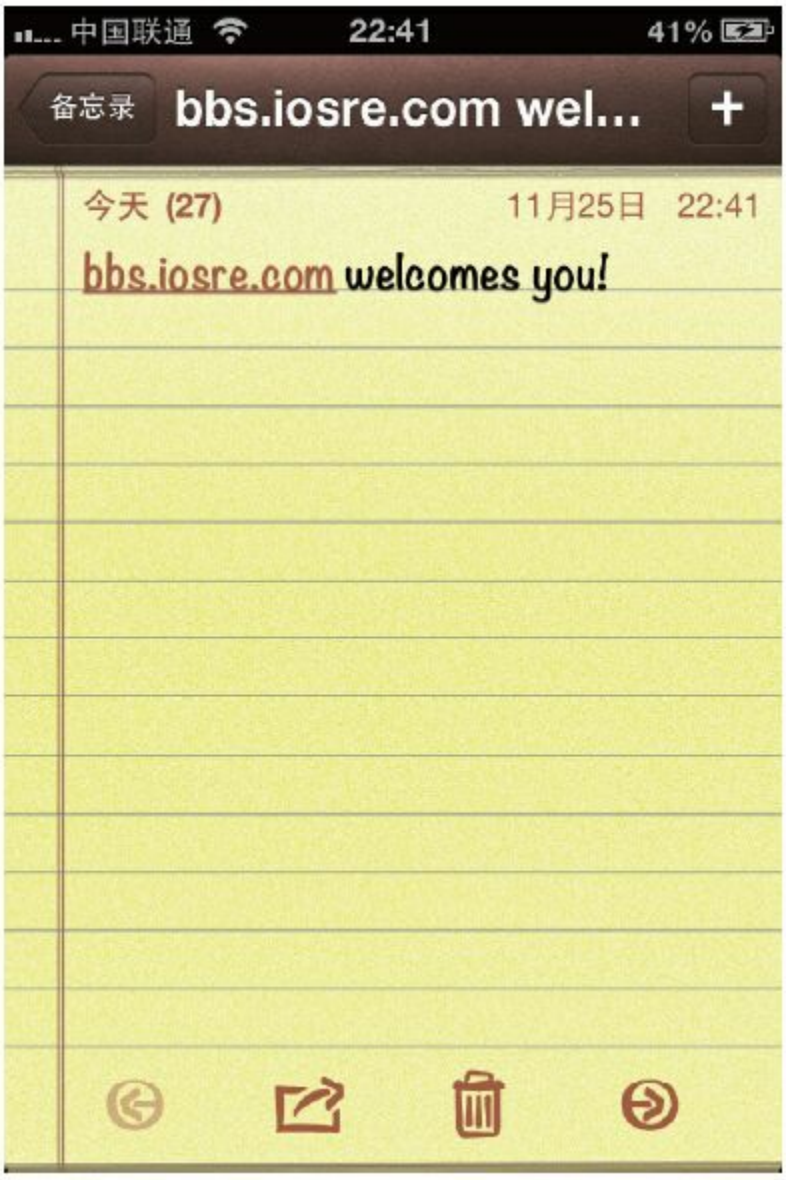


图5-4　Characount for Notes

2.倾听用户的声音

每个人使用iOS的方式不同，他们的需求各异。如果你自己没有太多灵感，那就多听听果粉们的需求；只要有需求，tweak就有用户。大的项目已经有人做了，我们就针对少数人群定制tweak；水平不足做不了底层的复杂功能，就从高层的简单功能做起；每一版发布后，虚心听取用户的意见和建议，及时改进，快速迭代，你的付出不会没有回报。

LowPowerBanner这个iOS 6插件就是笔者听取用户PrimeCode的建议编写的，完成第1版仅用了约5小时，写代码不到50行，但发布后8小时下载量即突破3万次（如图5-5所示），受欢迎程度大大超出笔者的预期。同志们，群众的眼睛是雪亮的，群众的智慧也是无穷的，如果你没有什么灵感，就走到群众中去吧！

Downloads for **snakeninny**

App	Total Count	Installer	Cydia
SMSNinja (v1.2)	22934	0	22934
LowPowerBanner (v0.0.1-42)	35493	0	35493

图5-5　LowPowerBanner的第一版下载量

3.解剖iOS

当你的能力越大时，能做的事情也就越多。千里之行始于足下，从小程序做起，经过层层磨炼，你对iOS的理解会不断加深；iOS是个封闭的系统，它暴露给我们的只是冰山一角，有太多太多的功能还有待我们进一步挖掘。每次越狱发布后，都会有人把最新的头文件发布出来，Google一下“iOS private headers”即可轻松找到下载链接，省去了自己class-dump的麻烦。Objective-C语言的函数命名很规律，大多数函数都可以望名生义，如SpringBoard.h里的，如下函数：

```
- (void)reboot;
- (void)relaunchSpringBoard;
```

和UIViewController.h里的，如下函数：

```
- (void)attentionClassDumpUser:(id)arg1
yesItsUsAgain:(id)arg2
althoughSwizzlingAndOverridingPrivateMethodsIsFun:
(id)arg3
```

```
itWasntMuchFunWhenYourAppStoppedWorking:(id)arg4
pleaseRefrainFromDoingSoInTheFutureOkayThanksBye:
(id)arg5;
```

通览这些函数名，是灵感的重要来源之一，也是了解iOS底层的便捷渠道。掌握越多的iOS实现细节，意味着手里握有的零件就越多，因此你就能组装出与别人不同的设备。limneos开发的“Audio Recorder”就是最好的例子，iOS早在2007年就面世了，但通话录音的功能直到7年后才由这位希腊开发者实现。有这个想法的人很多很多，已经动手的人也肯定不在少数，但为什么只有limneos成功了？因为他对iOS的解剖比别人更彻底！说起来很简单，做起来不简单。

5.2.2 定位目标文件

知道自己想要实现什么功能后，就要开始寻找实现这个功能的二进制文件，方法一般有以下几种。

1.固定位置

现阶段我们的逆向目标一般是dylib、bundle或daemon，它们在系统中的位置几乎是固定的：

· 基于CydiaSubstrate的dylib全部位于“/Library/MobileSubstrate/DynamicLibraries/”下，几乎不费吹灰之力就可以轻松定位。

· bundle主要分为App和framework两类，其中AppStore App全部位于“/var/mobile/Containers/Bundle/Application/”下，系统App全部位于“/Applications/”下，framework全部位于“/System/Library/Frameworks”或“/System/Librar
关于其他类型App的定位，可以来http://bbs.iosre.com一起讨论。

· daemon的配置文件均位于“/System/Library/LaunchDaemons/”、“/Library/Daemons”或“/Library/LaunchAgents/”下，是一个

plist格式的文件。其中的“ProgramArguments”字段，即是daemon可执行文件的绝对路径，如下：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
/Users/snakeninny/Desktop/com.apple.backboardd.plist
{
……
  "ProgramArguments" => [
    0 => "/usr/libexec/backboardd"
  ]
……
}
```

2.Cydia定位

通过“dpkg-i”命令安装的deb包，其内容会被Cydia如实记录，若要查看，在Cydia的“Installed”项中选择“Expert”，如图5-6所示。

然后选择目标软件，进入“Details”界面，如图5-7所示。

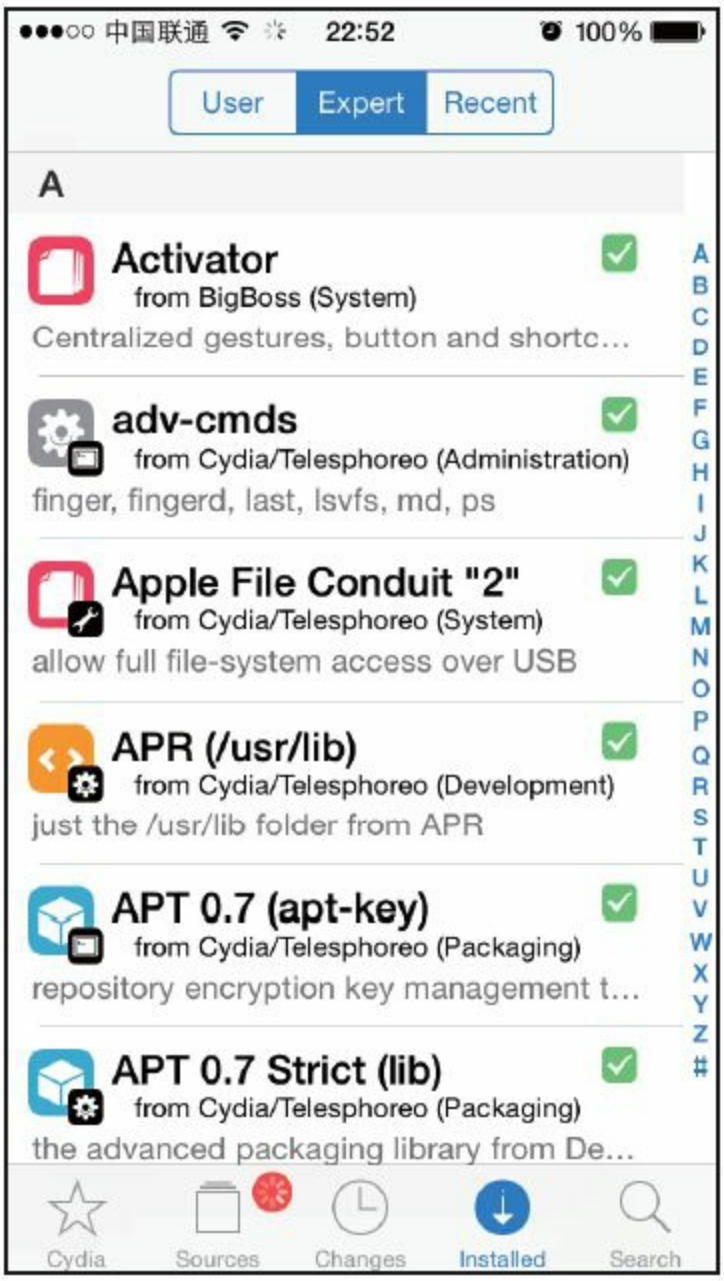


图5-6　Cydia的Expert界面

图5-7　Details界面

之后选择“Filesystem Content”，即可浏览软件包里的所有文件，如图5-8所示。

deb包中的每个文件被放在了iOS的哪个路径下，

一目了然。

3.PreferenceBundle

PreferenceBundle是寄生在Settings应用里的App，它的功能界定有些模糊，既可以作为单纯的配置文件，由别的进程读取后执行，如图5-9所示的“DimInCall”界面。

也可以含有实际功能，自己来执行一些操作，如图5-10所示的“WLAN”界面。

我们关注的重点是应用的实际功能，因此如何定位PreferenceBundle执行实际功能的二进制文件，就是需要研究的课题之一。来自AppStore的第三方PreferenceBundle仅可作为配置文件存在，不会含有实际功能；来自Cydia的也不是问题，刚才介绍的Cydia定位方式已经完全够用了；但对于iOS自带的PreferenceBundle来说，定位的过程就要复杂一些。

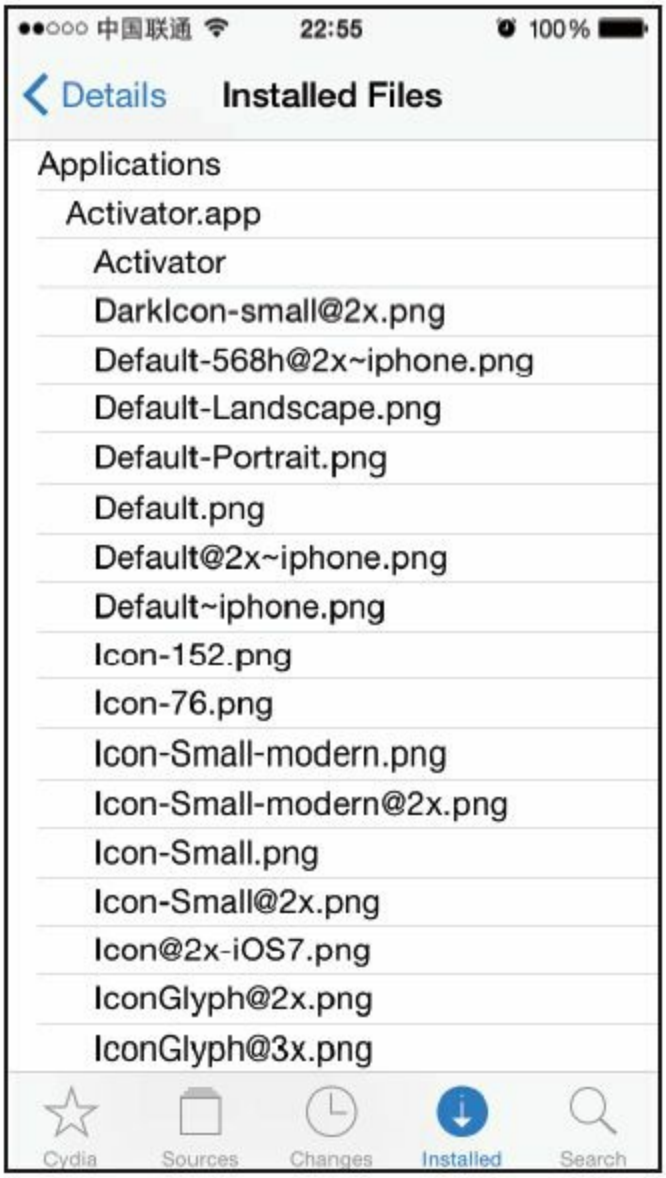


图5-8　Filesystem Content界面

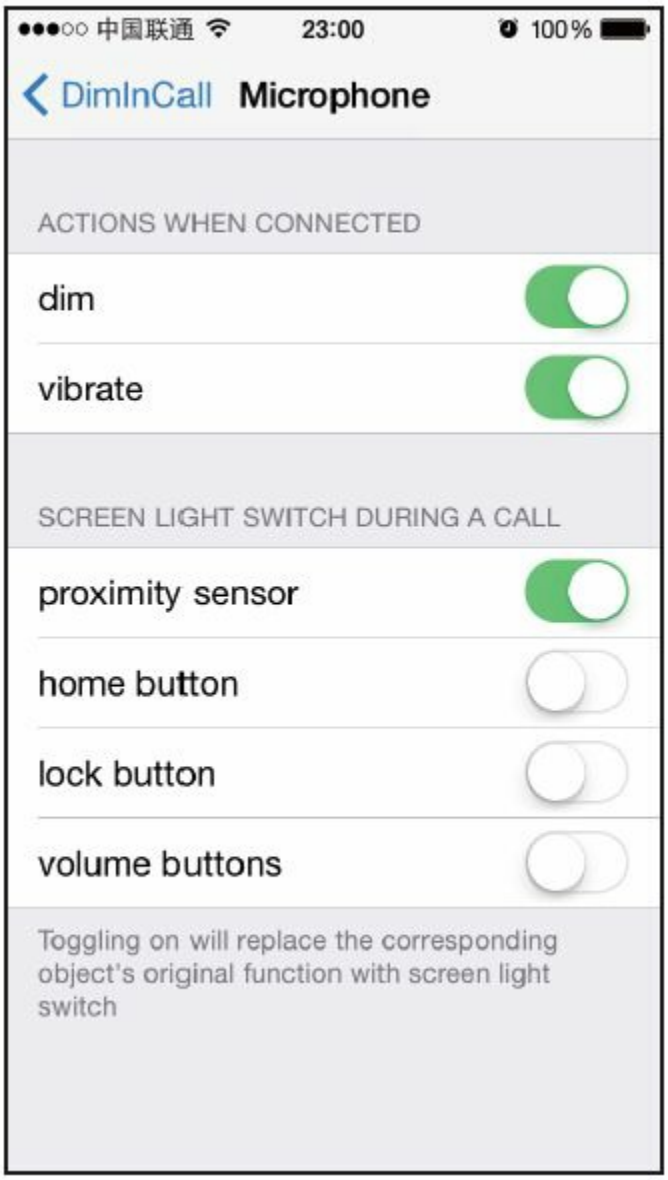


图5-9　DimInCall界面

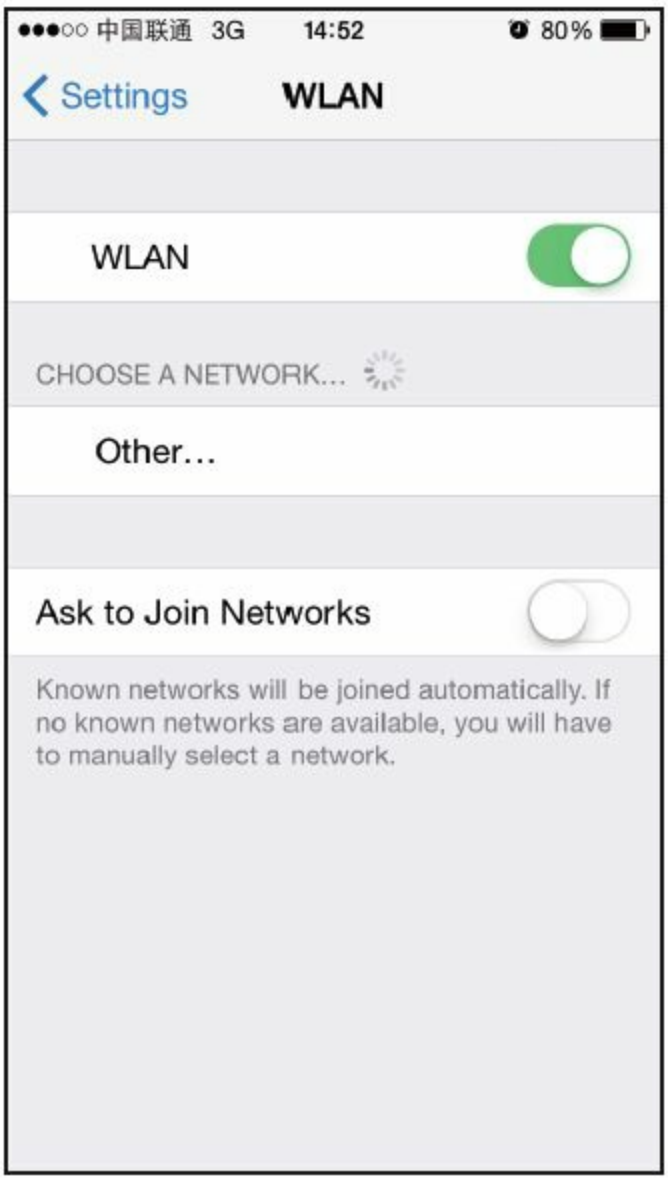


图5-10　WLAN界面

PreferenceBundle的界面可以用代码编写，也可以用具有固定格式的plist文件构造（格式请参考http://iphonedevwiki.net/index.php/Preferences_specifie在逆向此类程序时，如果发现界面中的控件类型全部

来自preference specifier plist罗列的标准控件类型，如“About”界面（如图5-11所示），则需注意分辨此界面是用代码编写的，还是用plist构造的。对于iOS自带的PreferenceBundle来说，如果是用代码编写的，一般情况下实际功能就已经包含在二进制文件里了，它们位于“/System/Library/PreferenceBundles/”下；如果是用plist构造的，就需要分析plist和实际功能间的关系，从中找到切入点，定位含有实际功能的二进制文件。总之，PreferenceBundle的情况相对复杂，并不适合作为新手练习。如果对上面的内容一知半解，不要紧，稍后本章会以一个实例来提供参考。更多关于PreferenceBundle的讨论，尽在http://bbs.iosre.com。

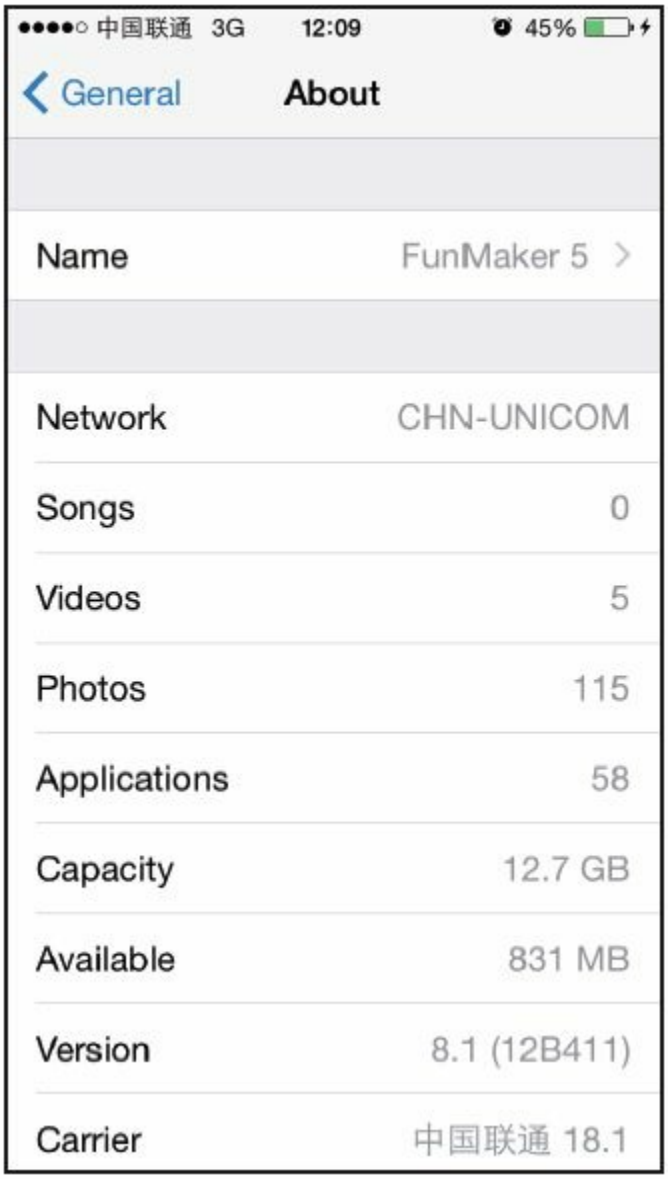


图5-11　About界面

4.grep命令

grep是一个来自UNIX系统的命令行工具，能够搜索文件中是否含有给定的正则表达式。OSX自带grep命令，iOS上的grep命令则是由Saurik移植过来

的，随着Cydia默认安装。在寻找一个字符串的出处时，grep能够快速缩小查找范围。例如，想知道都有哪些地方调用了[IMDAccount initWithAccountID:defaults:service:]，可以ssh到iOS后使用grep命令查看一下，如下：

```
FunMaker-5:~ root# grep -r
initWithAccountID:defaults:service: /System/Library/
Binary file
/System/Library/Caches/com.apple.dyld/dyld_shared_cache_a
 matches
grep: /System/Library/Caches/com.apple.dyld/enable-
dylibs-to-override-cache: No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep: /System/Library/Frameworks/System.framework/System:
No such file or directory
```

从运行结果得知，要查找的函数在dyld_shared_cache_armv7s中出现了。再次对decache过的dyld_shared_cache_armv7s使用grep命令，如下：

```
snakeninnysiMac:~ snakeninny$ grep -r
```

```
initWithAccountID:defaults:service:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/dyld
 matches
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Syst
 Too many levels of symbolic links
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Syst
 Too many levels of symbolic links
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Syst
 matches
```

可以看到，[IMDAccount initWithAccountID:defaults:service:]出现在了IMDaemonCore中，可以就从它着手开始分析。

5.2.3 定位目标函数

在找到含有目标功能的二进制文件之后，可以通过class-dump导出头文件，在里面寻找自己感兴趣的函数。具体做法比较简单，可分为以下两种。

1.OSX自带的搜索功能

不得不承认，OSX自带的搜索功能在笔者用过的

操作系统中是最强大的，强大到既能搜索文件名，又能搜索文件内容，而且不论是搜目录还是搜全盘，速度都非常快。利用这一便利工具，可以在大量文件中快速定位目标文件，例如，对iPhone自带的距离感应器（Proximity Sensor）很感兴趣，想看看相关的函数可能会提供哪些功能，可以在Finder中打开保存所有class-dump头文件的文件夹，然后在右上角的搜索栏中输入proximity（大小写不敏感），如图5-12所示。

默认情况下Finder会把本机所有内容中含有proximity关键词的文本文件罗列出来，如图5-13所示。

也可以缩小搜索范围，选择在当前目录下递归搜索文件名。剩下的工作就是找出你感兴趣的文件，然后开始分析喽！

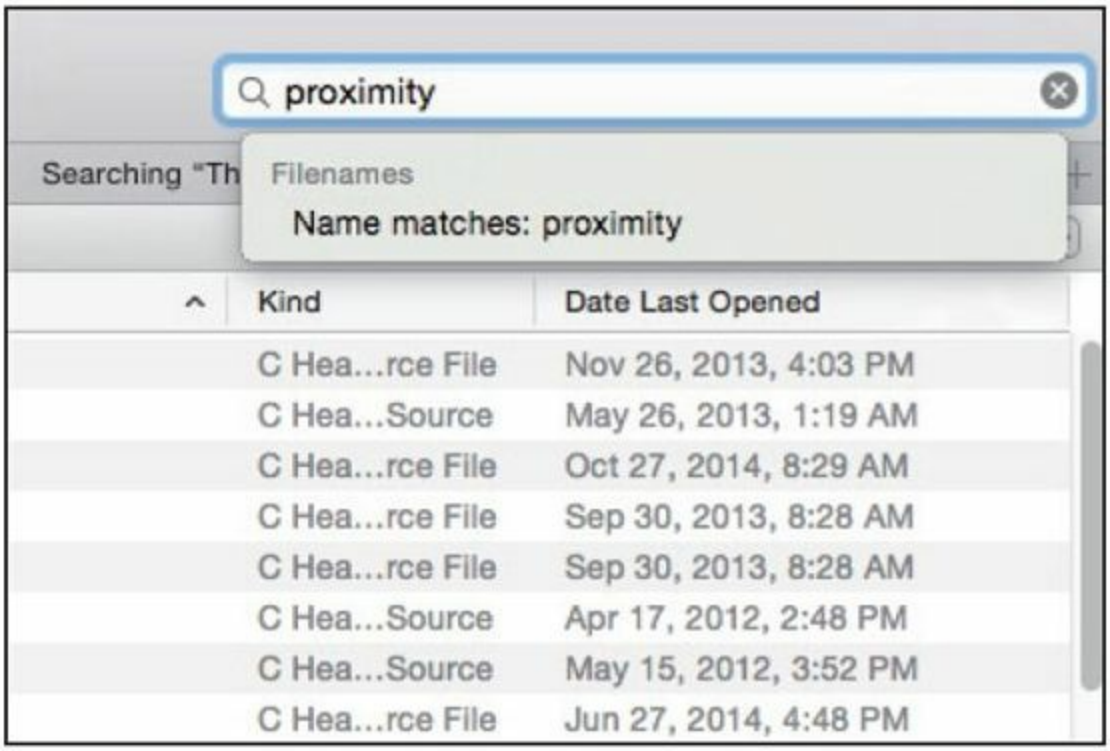


图5-12　在搜索栏输入关键词

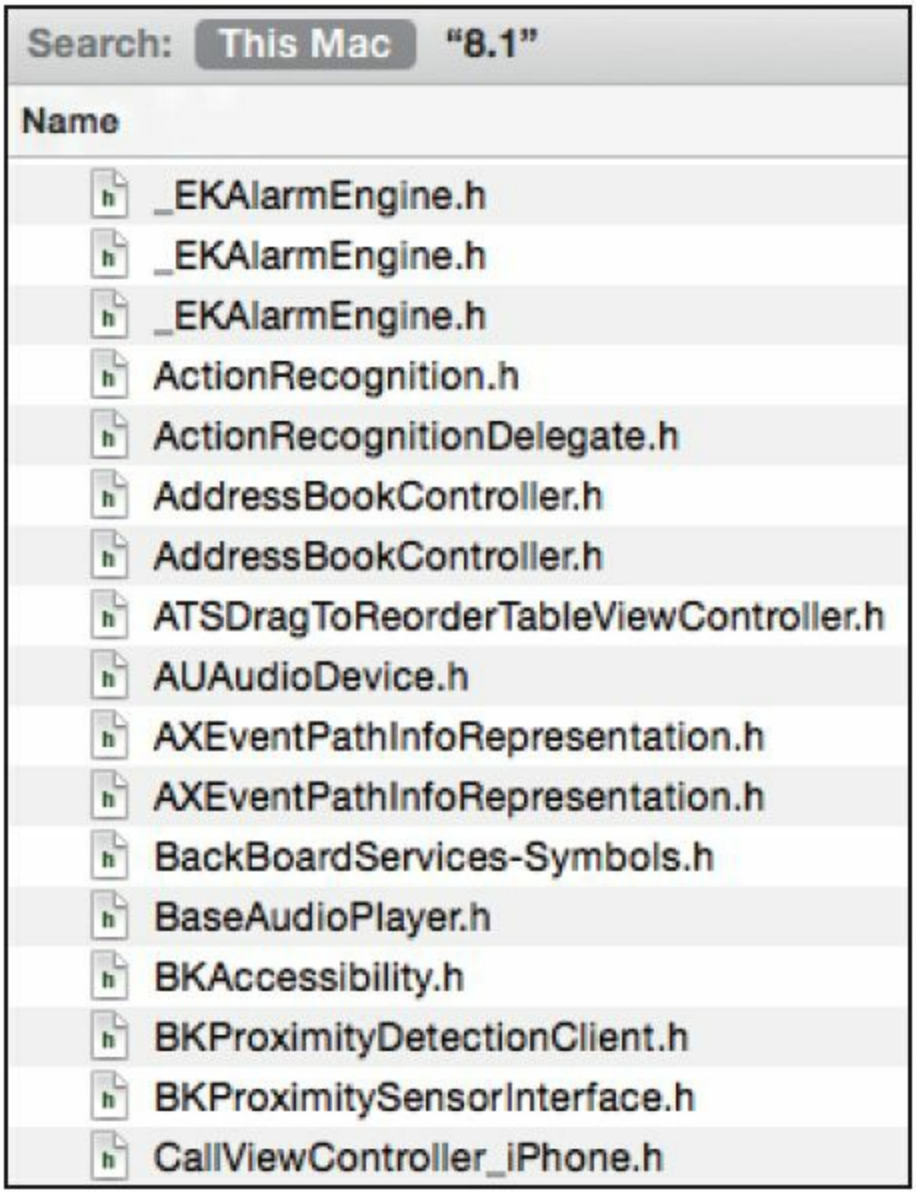


图5-13　搜索结果

2.grep命令

是的，你没看错，强大的grep再一次出现了。既然grep能搜索出二进制文件里的字符串，对付文本文件就更不在话下了。对于刚才的例子，使用grep来试

试，如下：

```
snakeninnysiMac:~ snakeninny$ grep -r -i proximity
/Users/snakeninny/Code/iOSPrivateHeaders/8.1
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/Frameworks/C
            char proximityUUID[512];
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/Frameworks/C
    NSUUID *_proximityUUID;
……
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/SpringBoard/
 (_Bool)proximityEventsEnabled;
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/SpringBoard/
 (void)_proximityChanged:(id)arg1;
```

虽然grep显示出的结果大而全，但看起来有些乱。推荐使用Finder的搜索功能，毕竟在便捷程度相差无几的情况下，图形界面比命令行界面用起来更方便。

5.2.4 测试函数功能

在逆向工程中，我们感兴趣的绝大多数函数都是私有的，没有文档可供参考，如果运气足够好，谷歌可能会帮上你的忙，但也可能说明你想做的东西别人已经做过了；如果搜索不到，那么恭喜，你可能发现

了一块新大陆，但是，函数的用法和功能需要你自己测试。

Objective-C函数的功能测试相对于C/C++函数来说要简单得多，有CydiaSubstrate和Cycript两种方法可供选择。

1.CydiaSubstrate

在测试函数功能时，主要利用CydiaSubstrate来钩住（hook）住一个函数，从而判断这个函数的调用时机。假设怀疑SBScreenShotter.h中的saveScreenshot:在截屏时得到了调用，就可以撰写以下代码来验证：

```
%hook SBScreenShotter
- (void)saveScreenshot:(BOOL)screenshot
{
        %orig;
        NSLog(@"iOSRE: saveScreenshot: is called");
}
%end
```

将filter设置成“com.apple.springboard”，并用Theos制作成deb，安装在iOS中，然后注销

（respring）一次。如果感觉有些生疏，不要着急，这是正常的，不求快，但求稳。等锁屏界面完全出现后，同时按下home键和lock键截屏，然后ssh到iOS查看syslog，如下：

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 24 16:22:06 FunMaker-5 SpringBoard[2765]: iOSRE:
saveScreenshot: is called
```

可以看到，syslog中出现了我们的自定义信息，说明在截屏时，saveScreenshot:得到了调用。此时，你一定会跟笔者一样好奇：这个函数名的含义太明显了，调用这个函数，是不是真就能实现截屏的功能呢？

在iOS的世界中，好奇不会害死猫，就怕你失去好奇心。要满足好奇心，就用Cycript！

2.Cycript

在知道Cycript之前，笔者测试函数功能的工具是

Theos。比如，针对上面的例子，笔者会编写下面这样一个tweak。

```
%hook SpringBoard
- (void)_menuButtonDown:(id)down
{
    %orig;
    SBScreenShotter *shotter = [%c(SBScreenShotter) sharedInstance];
    [shotter saveScreenshot:YES]; // 这里参数传YES是我猜的，等会我们试验一下传NO是什么效果
}
%end
```

在tweak生效后，按下home键，就会调用saveScreenshot：函数，然后观察屏幕是不是白光一闪，相册里是不是多了一张截屏。再进入Cydia把tweak删掉，把home键的单纯还给它……

其实如果没有对比，这种方法看起来还算简单，但是当笔者用Cycript达到了相同目的时，才后知后觉地发现，以前浪费了多少“井猜”的“绳命”！

Cycript的用法前面已经介绍过了，因为SBScreenShotter是SpringBoard里的类，所以这里将

Cycript注入SpringBoard进程，然后直接调用待测试函数观察实际效果即可，整个编译过程对我们是透明的，测试后无须任何清理工作，简直会让人忍不住哼唱：“测一个简单函数，让我的心情快乐，逆向就像一条河，难免会碰到波折。”

ssh到iOS后输入如下命令：

```
FunMaker-5:~ root# cycript -p SpringBoard
cy# [[SBScreenShotter sharedInstance] saveScreenshot:YES]
```

你的屏幕是不是也白光一闪，“咔嚓”一声，截屏一张，与同时按下2个键截屏的效果如出一辙？好了，现在可以确认这个函数能完成截屏操作了。为了进一步满足好奇心和求知欲，在Cycript提示符下按“↑”键，重复上一次输入的命令，然后把“YES”改成“NO”，看看是什么效果。下一节将会继续说明。

5.2.5 解析函数参数

在上面的例子中，函数的参数明确，函数名的含义明显，但我们还是拿不准在调用时到底是传YES还是NO，只能靠猜。浏览通过class-dump导出的头文件时，会发现绝大多数函数的参数类型是id，也就是Objective-C里的泛型，它们是在运行时动态决定的，猜都没法猜。我们从感兴趣的功能开始，一路分析到了对应的函数，只差一步就能闯过第一关了，难道要就此放弃？“不要放弃！”CydiaSubstrate和Theos异口同声地说。

还记得我们是怎样判断函数调用时机的吧？既然能打印一个自定义字符串，就完全能打印出函数参数的信息——description函数能够把对象的内容表示成一个NSString，object_getClassName函数能够把对象的类名表示成一个char*，两者可分别用%@和%s打印出来，这就为解析参数提供了足够参考。对于刚才完成的截屏操作，saveScreenshot：的参数是YES还是NO，唯一的区别好像在于屏幕是否闪现白光。依据

这个线索，很快就能定位到可疑的SBScreenFlash类，其中有一个有意思的函数flashColor:withCompletion:——是否闪光可以选择，难道闪光的颜色也可以改变？而且，参数类型似乎就是UIColor吧？编写下面的代码，来满足一下自己的好奇心。

```
%hook SBScreenFlash
- (void)flashColor:(id)arg1 withCompletion:(id)arg2
{
        %orig;
        NSLog(@"iOSRE: flashColor: %s, %@",
object_getClassName(arg1), arg1); // [arg1 description]
可以直接写成arg1
}
%end
```

作为练习，请读者把上面的代码变成一个可用的tweak。

安装完成后，注销（respring）一次，截个屏，再通过ssh命令连接到iOS上看看syslog，你所看到的内容应该如下所示：

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 24 16:40:33 FunMaker-5 SpringBoard[2926]: iOSRE:
flashColor: UICached DeviceWhiteColor,
UIDeviceWhiteColorSpace 1 1
```

可以看出，color是一个UICachedDeviceWhiteColor类型的对象，它的description是“UIDevice WhiteColorSpace 11”。根据命名规则，UICachedDeviceWhiteColor是UIKit中的一个类，但在文档中搜索不到这个类，因此可以断定它是个私有类。在class-dump出的UIKit头文件中找到UICachedDeviceWhiteColor.h，打开看看，如下：

```
@interface UICachedDeviceWhiteColor : UIDeviceWhiteColor
{
}
- (void)_forceDealloc;
- (void)dealloc;
- (id)copy;
- (id)copyWithZone:(struct _NSZone *)arg1;
- (id)autorelease;
- (BOOL)retainWeakReference;
- (BOOL)allowsWeakReference;
- (unsigned int)retainCount;
- (id)retain;
- (oneway void)release;
@end
```

它继承自UIDeviceWhiteColor，于是继续找到UIDeviceWhiteColor.h，如下：

```
@interface UIDeviceWhiteColor : UIColor
```

```
{
    float whiteComponent;
    float alphaComponent;
    struct CGColor *cachedColor;
    long cachedColorOnceToken;
}
- (BOOL)getHue:(float *)arg1 saturation:(float *)arg2
brightness:(float *)arg3 alpha:(float *)arg4;
- (BOOL)getRed:(float *)arg1 green:(float *)arg2 blue:
(float *)arg3 alpha:(float *)arg4;
- (BOOL)getWhite:(float *)arg1 alpha:(float *)arg2;
- (float)alphaComponent;
- (struct CGColor *)CGColor;
- (unsigned int)hash;
- (BOOL)isEqual:(id)arg1;
- (id)description;
- (id)colorSpaceName;
- (void)setStroke;
- (void)setFill;
- (void)set;
- (id)colorWithAlphaComponent:(float)arg1;
- (struct CGColor *)_createCGColorWithAlpha:(float)arg1;
- (id)copyWithZone:(struct _NSZone *)arg1;
- (void)dealloc;
- (id)initWithCGColor:(struct CGColor *)arg1;
- (id)initWithWhite:(float)arg1 alpha:(float)arg2;
@end
```

UIDeviceWhiteColor继承自UIColor，因为UIColor是一个公开类，所以对参数类型的解析到这个程度就可以了。对其他id类型参数的解析均可重复上述思路。

知道了一个函数的调用效果，解析了它的参数，它的使用文档就可以由我们自行撰写了，建议大家对

自己分析的函数作简单记录，这样在下次使用时就能迅速回想起它的用法。

接下来要用Cycript来测试这个函数，看看传进去一个[UIColor magentaColor]是什么效果，如下：

```
FunMaker-5:~ root# cycript -p SpringBoard
cy# [[SBScreenFlash mainScreenFlasher] flashColor:
[UIColor magentaColor] withCompletion:nil]
```

一抹紫红色的光在屏幕上散开，比白色的闪光有个性多了。检查相册，并没有看到新截屏，因此自然地猜测，这个函数仅仅负责截屏时的闪光功能，而不进行实际截屏操作——一个新的tweak灵感就此产生：我们可以钩住（hook）这个flashColor:with Completion：函数，把自定义的颜色作为参数传递给它，从而使截屏闪光变得丰富多彩起来。这个tweak作为练习，请读者独立完成。

以上的套路是笔者5年多以来的总结，因为iOS逆向工程没有任何官方资料可供参考，笔者个人经验难

免有失偏颇，不可能面面俱到，所以，http://bbs.iosre.com的大门向任何讨论开放，欢迎提问！

5.2.6 class-dump的局限性

分析通过class-dump导出的头文件，我们找到了感兴趣的东西，并在5.2.4节的Cycript试验中看到了对SBScreenShotter类中saveScreenshot：函数传YES和NO两种参数时函数的不同执行效果。

在5.2.5节里，解析了SBScreenFlash类的flashColor:withCompletion:函数参数。从flashColor:withCompletion:的效果来看，我们猜测它应该发生在saveScreenshot：的内部，而如果仅根据class-dump的头文件，结合CydiaSubstrate，最多也只能判断出saveScreenshot：和flashColor:withCompletion:的先后调用顺序，至于两者的实现细节和调用关系则不得而知。

完成了一个tweak，应该小小庆祝一下。从灵感，到文件，到函数，最后到成型的tweak，所有Objective-C级别的逆向工程都遵循这个套路，只是实现细节不同而已。即使完全不懂越狱开发，相信你也能掌握这个套路，它一点都不难。难度低，门槛就低，竞争就多，压力就大，当你掌握了Objective-C级别的逆向工程思路，想要进阶更高的级别，就会发现class-dump不够用了。

在完成一个小tweak之后，我们还应清楚地意识到，与这个tweak相关的很多知识点还没有弄清楚，而通过class-dump得到的信息并不足以支撑我们弄清这些未知的东西，就好像我们身处逆向工程这片茂密的原始森林中，class-dump提供了可以落脚的小屋，但要走出这片森林，还需要一张地图和一个指南针——它们就是IDA和LLDB。这两款工具就像两座挡在我们面前的大山，绝大多数逆向工程初学者都没能成功翻越它们，爬到半山腰就打道回府了，而翻越大

山的人们顺利跨过逆向工程的门槛，欣赏到了别样的风景。梦想还是要有的，万一实现了呢？我们鼓起勇气，试试看能不能征服它们。

5.3 实例演示

在翻山越岭之前，本节将针对刚才讲过的理论作一次全面的实战演练，让大家更牢固地掌握所学的知识，以便更平稳地过渡到第6章。本次实战演练的内容是一个真实的示例，它按照5.2节所示的套路，完整地讲述了笔者iOS 6插件“Speaker SBSettings Toggle”（如图5-14所示）的开发过程。当时笔者还不会使用IDA和LLDB，所有的线索几乎都来自class-dump和误打误撞，能够比较好地代表iOS逆向工程初学者的状态。

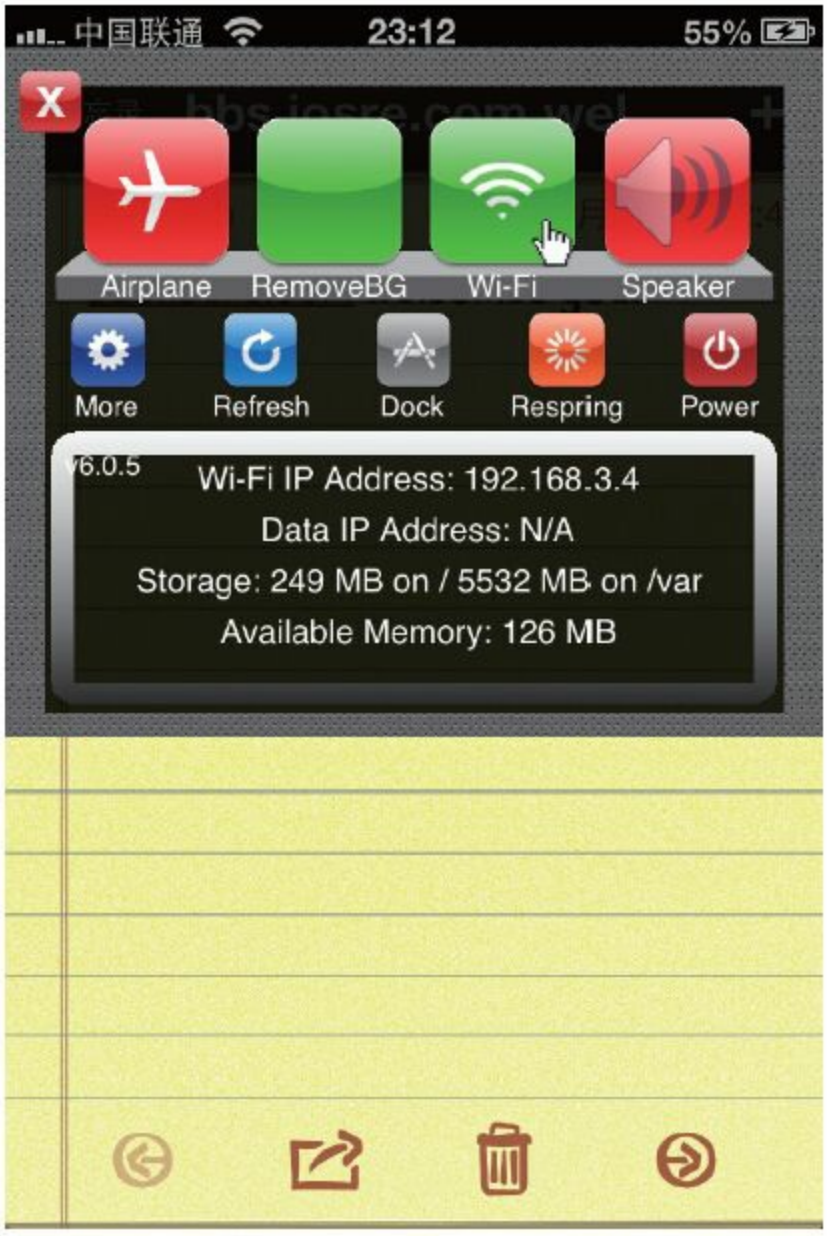


图5-14　Speaker SBSettings Toggle

注意　下面的具体步骤已不适用于iOS 8，请大家当做案例，了解思路，作为参考就好。

5.3.1　得到灵感

2012年3月底，笔者收到一个伊朗裔加拿大人Shoghian发来的邮件，邮件中分享了一个创意：iOS通话时用户可以从听筒切换到免提，但很少有人知道，接听来电时是可以默认打开免提的，这个功能对那些开车、做饭或工作时双手不方便接电话的人非常有用。但是这么有用的功能却被iOS藏在了“设置”→“通用”→“辅助功能”→“来电使用”的四级目录里（如图5-15所示），设置起来非常繁琐。SBSettings上各种各样的开关就是为解决这类问题而存在的，因此笔者打算把这个功能做成一个toggle，帮这家酒香不怕巷子深的饭店找一个临街的门面，把好的事物呈现在更多人的面前。

图5-15　Incoming Calls界面

5.3.2　定位文件

因为这个功能位于“设置”中，所以笔者的第一反应自然是

在“/Applications/Preferences.app”和“/System/Library/Pre
寻找可疑文件，大致步骤如下。

1.将iOS系统语言换成英文

因为iOS的文件系统是全英文的，所以在开始分析之前，笔者先把iOS的系统语言设置成了英文，这样在浏览文件系统时看到的关键词与UI上显示的关键词就更有可能产生对应关系。

2.发现“Accessibility”关键词

切换语言后，“设置”→“通用”→“辅助功能”→“来电使用”翻译成了“Settings”→“General”→“Accessibility”→“Incoming Calls”，其中Accessibility关键词引起了笔者的注意，因为不结合语境的话，Accessibility是不会直译成“辅助功能”的。于是笔者ssh到iOS中，以Accessibility为关键词进行了一次grep操作，如下：

```
FunMaker-4s:~ root# grep -r Accessibility /
grep: /Applications/Activator.app/Default-
568h@2x~iphone.png: No such file or directory
grep: /Applications/Activator.app/Default.png: No such
file or directory
grep: /Applications/Activator.app/Default~iphone.png: No
such file or directory
grep: /Applications/Activator.app/LaunchImage-700-
568h@2x.png: No such file or directory
Binary file
/Applications/Activator.app/en.lproj/Localizable.strings
matches
grep: /Applications/Activator.app/iOS7-Default-
Landscape@2x.png: No such file or directory
grep: /Applications/Activator.app/iOS7-Default-
Portrait@2x.png: No such file or directory
Binary file /Applications/AdSheet.app/AdSheet matches
Binary file /Applications/Compass.app/Compass matches
……
```

得到的结果很多，但最吸引笔者的是下面这几个以strings为后缀的文件：

```
Binary file
/Applications/Preferences.app/English.lproj/General-
Simulator.strings matches
Binary file
/Applications/Preferences.app/English.lproj/General~iphon
 matches
Binary file /Applications/Preferences.app/General-
Simulator.plist matches
Binary file /Applications/Preferences.app/General.plist
matches
Binary file /Applications/Preferences.app/Preferences
matches
Binary file
/Applications/Preferences.app/en_GB.lproj/General-
Simulator.strings matches
Binary file
/Applications/Preferences.app/en_GB.lproj/General~iphone.
 matches
```

如果不出意外，它们是App字符串本地化的配置文件，里面应该含有Accessibility在代码中的符号名。用Xcode自带的plutil工具查看strings文件非常方便，先来看

看“/Applications/Preferences.app/English.lproj/General~ip如下：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
~/General\~iphone.strings
{
  "Videos..." => "? Videos..."
  "Wallpaper" => "Wallpaper"
  "TV_OUT" => "TV Out"
  "SOUND_EFFECTS" => "Sound Effects"
  "d_MINUTES" => "%@ Minutes"
……
  "ACCESSIBILITY" => "Accessibility"
  "Multitasking_Gestures" => "Multitasking Gestures"
……
}
```

由“ACCESSIBILITY”=>“Accessibility”基本可以断定，“ACCESSIBILITY”就是代码中使用的符号名。

3.发现General.plist

有了新的线索后，以大写的ACCESSIBILITY为

关键词，又grep了一遍，如下：

```
FunMaker-4s:~ root# grep -r ACCESSIBILITY /
grep: /Applications/Activator.app/Default-
568h@2x~iphone.png: No such file or directory
grep: /Applications/Activator.app/Default.png: No such
file or directory
grep: /Applications/Activator.app/Default~iphone.png: No
such file or directory
grep: /Applications/Activator.app/LaunchImage-700-
568h@2x.png: No such file or directory
grep: /Applications/Activator.app/iOS7-Default-
Landscape@2x.png: No such file or directory
grep: /Applications/Activator.app/iOS7-Default-
Portrait@2x.png: No such file or directory
Binary file
/Applications/Preferences.app/Dutch.lproj/General-
Simulator.strings matches
Binary file
/Applications/Preferences.app/Dutch.lproj/General~iphone.
 matches
Binary file
/Applications/Preferences.app/English.lproj/General-
Simulator.strings matches
Binary file
/Applications/Preferences.app/English.lproj/General~iphon
 matches
Binary file
/Applications/Preferences.app/French.lproj/General-
Simulator.strings matches
Binary file
/Applications/Preferences.app/French.lproj/General~iphone
 matches
Binary file /Applications/Preferences.app/General-
Simulator.plist matches
Binary file /Applications/Preferences.app/General.plist
matches
Binary file
/Applications/Preferences.app/German.lproj/General-
Simulator.strings matches
Binary file
/Applications/Preferences.app/German.lproj/General~iphone
 matches
……
```

得到的结果与刚才grep结果的重合度很高，其中，刚才没有留意的“/Applications/Preferences.app/General.plist”显得格外醒目。在5.2.2节中，特意提到了PreferenceBundle的概念，此处General.plist既是plist格式的文件，又包含关键词，我们看看它里面有什么：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
~/General.plist
{
  "title" => "General"
  "items" => [
    0 => {
      "cell" => "PSGroupCell"
    }
    1 => {
      "detail" => "AboutController"
      "cell" => "PSLinkCell"
      "label" => "About"
    }
    2 => {
      "cell" => "PSLinkCell"
      "id" => "SOFTWARE_UPDATE_LINK"
      "detail" => "SoftwareUpdatePrefController"
      "label" => "SOFTWARE_UPDATE"
      "cellClass" => "PSBadgedTableCell"
    }
        ……
    24 => {
      "detail" => "PSInternationalController"
      "cell" => "PSLinkCell"
      "label" => "INTERNATIONAL"
    }
    25 => {
      "cell" => "PSLinkCell"
      "bundle" => "AccessibilitySettings"
      "label" => "ACCESSIBILITY"
```

```
        "requiredCapabilities" => [
          0 => "accessibility"
        ]
        "isController" => 1
      }
      26 => {
        "cell" => "PSGroupCell"
      }
          ……
    ]
}
```

4.发现AccessibilitySetting.bundle

果不其然，这个文件就是一个标准的preference specifier plist，大写的“ACCESSIB-ILITY”来自25号单元。对比preferences specifier plist格式，将目标锁定在Accessibility-Settings这个bundle中；由AccessibilitySettings的名字，自然地猜测这个bundle可能负责Accessibility下的所有功能。根据5.2.2节中的文件固定位置理论，AccessibilitySettings.bundle一定安安静静地躺在“/System/Library/PreferenceBundles/”中，对将要发生在自己身上的事情浑然不知。

看

看“/System/Library/PreferenceBundles/AccessibilitySettir

面有什么：

```
FunMaker-4s:~ root# ls -la
/System/Library/PreferenceBundles/Accessibility
Settings.bundle
total 240
drwxr-xr-x 37 root wheel   2414 Mar 10  2013 .
drwxr-xr-x 40 root wheel   1360 Jan 14  2014 ..
-rw-r--r--  1 root wheel   2146 Mar 10  2013
Accessibility.plist
-rwxr-xr-x  1 root wheel 438800 Mar 10  2013
AccessibilitySettings
-rw-r--r--  1 root wheel    238 Dec 22  2012
BluetoothDeviceConfig.plist
-rw-r--r--  1 root wheel    252 Mar 10  2013
BrailleStatusCellSettings.plist
-rw-r--r--  1 root wheel   4484 Dec 22  2012
ColorWellRound@2x.png
-rw-r--r--  1 root wheel    916 Dec 22  2012
ColorWellSquare@2x.png
drwxr-xr-x  2 root wheel    646 Feb  7  2013 Dutch.lproj
drwxr-xr-x  2 root wheel    646 Dec 22  2012
English.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 French.lproj
drwxr-xr-x  2 root wheel    646 Dec 22  2012 German.lproj
-rw-r--r--  1 root wheel    703 Mar 10  2013
GuidedAccessSettings.plist
-rw-r--r--  1 root wheel    807 Mar 10  2013
HandSettings.plist
-rw-r--r--  1 root wheel    652 Mar 10  2013
HearingAidDetailSettings.plist
-rw-r--r--  1 root wheel    507 Mar 10  2013
HearingAidSettings.plist
-rw-r--r--  1 root wheel    383 Dec 22  2012
HomeClickSettings.plist
-rw-r--r--  1 root wheel    447 Dec 22  2012
IconPlay@2x.png
-rw-r--r--  1 root wheel   1113 Dec 22  2012
IconRecord@2x.png
-rw-r--r--  1 root wheel    170 Dec 22  2012
IconStop@2x.png
-rw-r--r--  1 root wheel    907 Mar 10  2013 Info.plist
drwxr-xr-x  2 root wheel    646 Feb  7  2013
Italian.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013
```

```
Japanese.lproj
-rw-r--r--  1 root wheel    364 Dec 22  2012
LargeFontsSettings.plist
-rw-r--r--  1 root wheel    217 Mar 10  2013
NavigateImagesSettings.plist
-rw-r--r--  1 root wheel   1030 Dec 22  2012
QuickSpeakSettings.plist
-rw-r--r--  1 root wheel    346 Dec 22  2012
RegionNamesNonLocalized.strings
drwxr-xr-x  2 root wheel    646 Feb  7  2013
Spanish.lproj
-rw-r--r--  1 root wheel    394 Dec 22  2012
SpeakerLoad1@2x.png
-rw-r--r--  1 root wheel    622 Mar 10  2013
TripleClickSettings.plist
-rw-r--r--  1 root wheel    467 Dec 22  2012
VoiceOverBrailleOptions.plist
-rw-r--r--  1 root wheel   2477 Mar 10  2013
VoiceOverSettings.plist
-rw-r--r--  1 root wheel    540 Mar 10  2013
VoiceOverTypingFeedback.plist
-rw-r--r--  1 root wheel    480 Dec 22  2012
ZoomSettings.plist
drwxr-xr-x  2 root wheel    102 Dec 22  2012
_CodeSignature
drwxr-xr-x  2 root wheel    646 Feb  7  2013 ar.lproj
-rw-r--r--  1 root wheel   8371 Dec 22  2012
bottombar@2x~iphone.png
-rw-r--r--  1 root wheel   2701 Dec 22  2012
bottombarblue@2x~iphone.png
-rw-r--r--  1 root wheel   2487 Dec 22  2012
bottombarblue_pressed@2x~iphone.png
-rw-r--r--  1 root wheel   2618 Dec 22  2012
bottombarred@2x~iphone.png
-rw-r--r--  1 root wheel   2426 Dec 22  2012
bottombarred_pressed@2x~iphone.png
-rw-r--r--  1 root wheel   2191 Dec 22  2012
bottombarwhite@2x~iphone.png
-rw-r--r--  1 root wheel   2357 Dec 22  2012
bottombarwhite_pressed@2x~iphone.png
drwxr-xr-x  2 root wheel    646 Feb  7  2013 ca.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 cs.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 da.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 el.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 en_GB.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 fi.lproj
-rw-r--r--  1 root wheel    955 Dec 22  2012 hare@2x.png
drwxr-xr-x  2 root wheel    646 Feb  7  2013 he.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 hr.lproj
```

```
drwxr-xr-x  2 root wheel    646 Feb  7  2013 hu.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 id.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 ko.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 ms.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 no.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 pl.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 pt.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 pt_PT.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 ro.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 ru.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 sk.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 sv.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 th.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 tr.lproj
-rw-r--r--  1 root wheel    998 Dec 22  2012
turtle@2x.png
drwxr-xr-x  2 root wheel    646 Feb  7  2013 uk.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 vi.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 zh_CN.lproj
drwxr-xr-x  2 root wheel    646 Feb  7  2013 zh_TW.lproj
```

这里的GuidedAccess、HearingAid和HomeClick等字眼和我们在“Accessibility”中看到的内容吻合（如图5-16所示），它们印证了笔者的猜测。

5.发现“ACCESSIBILITY_DEFAULT_HEADSET”关键词

借助强大的grep，以“Incoming”为关键词搜索一下这个bundle，如下：

图5-16　关键词重合度高

```
FunMaker-4s:~ root# grep -r Incoming
/System/Library/PreferenceBundles/AccessibilitySettings.b

Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.b
 matches
Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.b
 matches
```

搜索的结果同本小节开始时的场景如出一辙。打开“/System/Library/PreferenceBundles/AccessibilitySettir看：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
~/Accessibility\~iphone.strings
{
  "HAC_MODE_POWER_REDUCTION_N90" => "Hearing Aid Mode
improves performance with some hearing aids, but may
reduce cellular reception."
  "LEFT_RIGHT_BALANCE_SPOKEN" => "Left-Right Stereo
Balance"
  "QUICKSPEAK_TITLE" => "Speak Selection"
  "LeftStereoBalanceIdentifier" => "L"
  "ACCESSIBILITY_DEFAULT_HEADSET" => "Incoming Calls"
  "HEADSET" => "Headset"
  "CANCEL" => "Cancel"
  "ON" => "On"
  "CUSTOM_VIBRATIONS" => "Custom Vibrations"
  "CONFIRM_INVERT_COLORS_REMOVAL" => "Are you sure you
want to disable inverted colors?"
  "SPEAK_AUTOCORRECTIONS" => "Speak Auto-text"
  "DEFAULT_HEADSET_FOOTER" => "Choose route for incoming
calls."
  "HEARING_AID_COMPLIANCE_INSTRUCTIONS" => "Improves
compatibility with hearing aids in some circumstances.
May reduce 2G cellular coverage."
  "DEFAULT_HEADSET" => "Default to headset"
  "ROOT_LEVEL_TITLE" => "Accessibility"
  "HEARING_AID_COMPLIANCE" => "Hearing Aid Mode"
  "CUSTOM_VIBES_INSTRUCTIONS" => "Assign unique vibration
patterns to people in Contacts. Change the default
pattern for everyone in Sounds settings."
  "VOICEOVERTOUCH_TEXT" => "VoiceOver is for users with
blindness or vision disabilities."
  "IMPORTANT" => "Important"
  "COGNITIVE_HEADING" => "Learning"
  "HAC_MODE_EQUALIZATION_N94" => "Hearing Aid Mode
improves audio quality with some hearing aids."
  "SAVE" => "Save"
  "HOME_CLICK_TITLE" => "Home-click Speed"
  "AIR_TOUCH_TITLE" => "AssistiveTouch"
```

```
  "CONFIRM_ZOT_REMOVAL" => "Are you sure you want to
disable Zoom?"
  "VOICEOVER_TITLE" => "VoiceOver"
  "OFF" => "Off"
  "GUIDED_ACCESS_TITLE" => "Guided Access"
  "ZOOMTOUCH_TEXT" => "Zoom is for users with low-vision
acuity."
  "INVERT_COLORS" => "Invert Colors"
  "ACCESSIBILITY_SPEAK_AUTOCORRECTIONS" => "Speak Auto-
text"
  "LEFT_RIGHT_BALANCE_DETAILS" => "Adjust the audio
volume balance between left and right channels."
  "MONO_AUDIO" => "Mono Audio"
  "CONTRAST" => "Contrast"
  "ZOOM_TITLE" => "Zoom"
  "TRIPLE_CLICK_HEADING" => "Triple-click"
  "OK" => "OK"
  "SPEAKER" => "Speaker"
  "AUTO_CORRECT_TEXT" => "Automatically speak auto-
corrections
and auto-capitalizations."
  "HEARING" => "Hearing"
  "LARGE_FONT" => "Large Text"
  "CONFIRM_VOT_USAGE" => "VoiceOver"
  "CONFIRM_VOT_REMOVAL" => "Are you sure you want to
disable VoiceOver?"
  "HEARING_AID_TITLE" => "Hearing Aids"
  "FLASH_LED" => "LED Flash for Alerts"
  "VISION" => "Vision"
  "CONFIRM_ZOOM_USAGE" => "Zoom"
  "DEFAULT" => "Default"
  "MOBILITY_HEADING" => "Physical & Motor"
  "TRIPLE_CLICK_TITLE" => "Triple-click Home"
  "RightStereoBalanceIdentifier" => "R"
}
```

"ACCESSIBILITY_DEFAULT_HEADSET"=>"Inco

Calls"给了我们非常明显的提示，以它为线索继续查找。

6.定位Accessibility.plist

```
FunMaker-4s:~ root# grep -r ACCESSIBILITY_DEFAULT_HEADSET
/System/Library/PreferenceBundles/AccessibilitySettings.b

Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.b
 matches
Binary file
/System/Library/PreferenceBundles/AccessibilitySettings.b
 matches
...
```

除了一个plist文件外，其他都是strings文件，那就是它了。看看它里面有什么：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
~/Accessibility.plist
{
  "title" => "ROOT_LEVEL_TITLE"
  "items" => [
    0 => {
      "label" => "VISION"
      "cell" => "PSGroupCell"
      "footerText" => "AUTO_CORRECT_TEXT"
    }
    1 => {
      "cell" => "PSLinkListCell"
      "label" => "VOICEOVER_TITLE"
      "detail" => "VoiceOverController"
      "get" => "voiceOverTouchEnabled:"
    }
    2 => {
      "cell" => "PSLinkListCell"
      "label" => "ZOOM_TITLE"
      "detail" => "ZoomController"
      "get" => "zoomTouchEnabled:"
    }
    ……
```

```
    18 => {
      "cell" => "PSLinkListCell"
      "label" => "HOME_CLICK_TITLE"
      "detail" => "HomeClickController"
      "get" => "homeClickSpeed:"
    }
    19 => {
      "detail" => "PSListItemsController"
      "set" => "accessibilitySetPreference:specifier:"
      "validValues" => [
        0 => 0
        1 => 1
        2 => 2
      ]
      "get" => "accessibilityPreferenceForSpecifier:"
      "validTitles" => [
        0 => "DEFAULT"
        1 => "HEADSET"
        2 => "SPEAKER"
      ]
      "requiredCapabilities" => [
        0 => "telephony"
      ]
      "cell" => "PSLinkListCell"
      "label" => "ACCESSIBILITY_DEFAULT_HEADSET"
      "key" => "DefaultRouteForCall"
    }
  ]
}
```

又是一个标准的preference specifier plist，而且我们知道了这个配置的setter和getter分别是accessibilitySetPreference:specifier:和accessibilityPreferenceForSpecifier:，可以进入下一环节了。

5.3.3 定位函数

根据preference specifier plist标准，在选择“Incoming Calls”中的某一行时，其setter，即accessibilitySetPreference:specifier：函数得到调用。但问题随之而来，这个函数存在于AccessibilitySettings.bundle里，笔者当时不知道怎么将这个bundle加载进内存，因此没法调用这个函数；也不会用IDA和LLDB，在class-dump的函数里找了又找，仍没有发现任何线索，感觉这个问题的难度已经超出笔者的能力范围，一时解决不了，还沮丧地给Shoghian发了封邮件，如图5-17所示。

图5-17　我和Shoghian之间的交流

这个问题卡了笔者近两个星期，期间笔者一直在想，iOS能在这个函数里干些什么呢？因为preferences specifier plist中提供了PostNotification这一方式来通知别的进程配置文件发生了变动，而AccessibilitySettings的配置与电话相关，正好也是进程间通信的模式，那么，accessibilitySetPreference:specifier：的作用会不会是改

动配置文件，然后发出一个通知？于是笔者利用limneos开发的LibNotifyWatch，在手动改变“来电使用”配置时观察系统中是否出现了相关的通知，没想到，还真让笔者歪打正着了，如下：

```
FunMaker-4s:~ root# grep LibNotifyWatch: /var/log/syslog
Nov 26 00:09:20 FunMaker-4s Preferences[6488]:
LibNotifyWatch: <CFNotification Center 0x1e875600
[0x39b4b100]>
postNotificationName:UIViewAnimationDidCommitNotification
object:UIViewAnimationState userInfo:{
Nov 26 00:09:20 FunMaker-4s Preferences[6488]:
LibNotifyWatch: <CFNotificationCenter 0x1e875600
[0x39b4b100]>
postNotificationName:UIViewAnimationDidStopNotification
object:<UIViewAnimationState: 0x1ea74f20> userInfo:{
……
Nov 26 00:09:21 FunMaker-4s Preferences[6488]:
LibNotifyWatch: CFNotificationCenterPostNotification
center=<CFNotificationCenter 0x1dd86bd0 [0x39b4b100]>
name=com.apple.accessibility.defaultrouteforcall
userInfo=(null) deliverImmediately=1
Nov 26 00:09:21 FunMaker-4s Preferences[6488]:
LibNotifyWatch: notify_post
com.apple.accessibility.defaultrouteforcall
……
```

笔者发现了2条名为“com.apple.accessibility.defaultrouteforcall”的通知！结合前面的一系列推导，想来没有必要再多作解释了。发现了最可疑的通知后，面对的就是另一个同样

重要的问题：配置文件在哪里？

第2章说过，“/var/mobile/”中存放了大量用户数据。“/var/mobile/Containers/”中全是App相关数据，“/var/mobile/Media/”中全是媒体文件，而在“/var/mobile/Library/”中稍加浏览就很容易发现“/var/mobile/Library/Preferences/”目录，进而找到“com.apple.Accessibility.plist”，其内容如下：

```
snakeninnys-MacBook:~ snakeninny$ plutil -p
~/com.apple.Accessibility.plist
{
      ……
  "DefaultRouteForCallPreference" => 2
  "VOTQuickNavEnabled" => 1
  "CurrentRotorTypeWeb" => 3
  "PunctuationKey" => 2
        ……
  "ScreenCurtain" => 0
  "VoiceOverTouchEnabled" => 0
  "AssistiveTouchEnabled" => 0
}
```

在iOS中改变“来电使用”的配置，观察DefaultRouteForCallPreference值的变化规律，很容易得出结论：0对应default，1对应headset，2对应

speaker，与Accessibility.plist的内容吻合。

5.3.4 测试函数

在经过漫长的推理之后，笔者总算得出了一个可能的解决方案，仅需极少代码即可修改配置文件，然后发出一个通知，就这么简单。这个方案可行吗？怀揣一颗惴惴不安又蠢蠢欲动的心，用激动的双手敲出了下面为数不多的几行代码（那时候还不会用Cycript，所以用tweak测试）：

```
%hook SpringBoard
- (void)menuButtonDown:(id)down
{
    %orig;
    NSMutableDictionary *dictionary =
[NSMutableDictionary dictionary WithContents
OfFile:@"/var/mobile/Library/Preferences/com.apple.
Accessibility.plist"];
    [dictionary setObject:[NSNumber numberWithInt:2]
forKey:@"DefaultRouteForCallPreference"];
   [dictionary
writeToFile:@"/var/mobile/Library/Preferences/com.apple.
Accessibility. plist" atomically:YES];

notify_post("com.apple.accessibility.defaultrouteforcall"

}
%end
```

编译、运行、安装、respring，闭着眼睛按下home键，然后伴着极快的心跳依次打开“Settings”→“General”→“Accessibility”→“Incoming Calls”——已选项变成了“Speaker”，成功啦！

5.3.5 编写实例代码

程序的核心功能已经验证完毕，写代码就不用费脑子了。按照SBSettings toggle的编写规范完成代码（http://thebigboss.org/guides-iphone-ipod-ipad/sbsettings-toggle-spec），完整代码如下：

```
#import <notify.h>
#define ACCESSBILITY
@"/var/mobile/Library/Preferences/com.apple.Accessibility
 plist"
// Required
extern "C" BOOL isCapable() {
      if (kCFCoreFoundationVersionNumber >=
kCFCoreFoundationVersionNumber_iOS_5_0 && [[[UIDevice
currentDevice] model] isEqualToString:@"iPhone"])
            return YES;
      return NO;
}
// Required
extern "C" BOOL isEnabled() {
      NSMutableDictionary *dictionary =
[[NSMutableDictionary alloc] initWithCont
entsOfFile:ACCESSBILITY];
      BOOL result = [[dictionary
```

```
objectForKey:@"DefaultRouteForCallPreference"] intValue]
== 0 ? NO : YES;
      [dictionary release];
      return result;
}
// Optional
// Faster isEnabled. Remove this if it's not necessary.
Keep it if isEnabled() is expensive and you can make it
faster here.
extern "C" BOOL getStateFast() {
      return isEnabled();
}
// Required
extern "C" void setState(BOOL enabled) {
      NSMutableDictionary *dictionary =
[[NSMutableDictionary alloc] initWithCont
entsOfFile:ACCESSBILITY];
      [dictionary setObject:[NSNumber numberWithInt:
(enabled ? 2 : 0)] forKey:@"D
efaultRouteForCallPreference"];
      [dictionary writeToFile:ACCESSBILITY
atomically:YES]; [dictionary release];

notify_post("com.apple.accessibility.defaultrouteforcall"

}
// Required
// How long the toggle takes to toggle, in seconds.
extern "C" float getDelayTime() {
      return 0.6f;
}
```

因为程序的创意来自Shoghian，所以笔者在发布这个程序时也标注了他的名字（如图5-18所示）。他很高兴，我们还成了朋友，偶尔也天南地北地扯上一会儿。Speaker SBSettings Toggle是笔者发布在Cydia上的第三个程序，虽然功能简单，也没有作什么宣传，

但还是累积了近10000的下载量（如图5-19所示），对此笔者已经很满意了。更重要的是，这个tweak的制作历经坎坷，看似简单的功能却让笔者花费了九牛二虎之力，无疑给了当时刚刚上路，有些轻飘飘的笔者当头一棒！类似的情况出现过若干次后，笔者才意识到仅仅使用class-dump来做逆向工程是不靠谱的，也间接促使笔者下定决心学习IDA和LLDB，从而迈入了iOS逆向工程的新阶段。

图5-18　第二作者是Shoghian

SMSNinja (v1.3.1)	102947	0	102947
Speaker SBSettings Toggle (v0.0.1-3)	9522	0	9522
Characount for Notes (v1.0)	14447	0	14447

图5-19　积累了近10000下载量

5.4 小结

本章较为完整地介绍了tweak的作用原理及编写简单tweak的思路和流程，佐以真实案例，能够较好地为初学者提供参考。Objective-C级别的逆向工程是iOS逆向工程的第一关，在没有上手IDA和LLDB之前，对iOS的逆向工程不可能深入到什么地步，也没有什么逻辑可言，相信你从案例里也看出来了，我们对二进制文件的逆向非常力不从心，当问题的关键集中在代码上时，解决问题的主要方式就是猜！虽然刚才编写的代码跟iOS自身的实现差了十万八千里，但因为Objective-C函数名的可读性高，所以即使是猜，也还是能利用class-dump出的函数达到预期效果，给自己带来跟App开发完全不同的感觉，让人耳目一新。

在逆向工程初学阶段，我们的主要目的是熟悉越狱iOS环境，了解前几章讲到的各种逆向知识点，在

掌握各种工具用法的同时有意识地培养自己的逆向思维。如果时间比较充裕，强烈建议大家通览class-dump出的头文件，把那些语义明显、自己感兴趣的函数放到iOS上实测一下，这个过程能极大地增加你对iOS底层的熟悉程度，配合后续的IDA与LLDB学习，能达到事半功倍的效果。只要我们多思考、勤练习，就能早日提炼出更适合自己的方法，进一步达到更高的水平。

第6章　ARM汇编相关的iOS逆向理论基础

前面的章节中介绍了iOS逆向工程的基础知识，包括一些常见工具的组合使用，在掌握了这些知识之后，简单地把玩一下Objective-C私有函数，满足一下自己的好奇心已经没问题了，可以针对App开发tweak了。但是，既然看到了这里，相信大家都具有比较强的钻研精神，如果想要真正提高自己的能力，就要尝试一些更有挑战性的内容。那么，从这一章开始，iOS逆向工程就将进入“极昼”，我们将零距离接触编程世界里最让人头大的知识。请先深呼吸一分钟，然后问问自己：“我是否真的适合深入学习iOS逆向工程？”在完成本章之后，相信你会得到答案。

下面即将面对iOS逆向工程中的第一个进阶难点：阅读ARM汇编语言。经过前几章的学习，相信大家已经知道，Objective-C代码在经过编译后形成机

器码，它们由设备的CPU直接执行。别说编写，阅读机器码都已经是一个非常恼人的工作；好在Objective-C和机器码之间有汇编语言这座桥，它的可读性虽然远不如Objective-C，但比机器码要强多了——如果你能够啃下这块硬骨头，那么恭喜你，你有着成为逆向工程师的天赋；如果你在啃骨头的时候牙被崩掉了，或许AppStore开发才是你更好的归宿……

6.1 ARM汇编基础

对于很多iOS开发者来说，ARM汇编是一门全新的语言；如果你是计算机专业科班出身，应该已经对汇编语言有了初步的印象，只是对于很多人来说，大学期间的汇编语言课简直跟天书一样深奥，它在我们心里埋下了恐惧的种子，仿佛一提到汇编语言，它就会像紧箍咒一样勒紧我们的头，让我们疼痛不已。汇编语言真的有这么难？是，因为汇编的语法晦涩难懂；但另一方面，毕竟它只是一门语言，跟英语一样，熟能生巧。

我们一般的工作中与汇编打交道的机会并不多，如果不刻意练习，陡然面对时必然掌握不了，所以会觉得它很难。不过归根到底还是投入的时间和精力是否足够的问题——好了，iOS逆向工程给学习ARM汇编提供了一个绝佳的条件——在逆向一个功能时，往往需要分析大量ARM汇编代码，并把它们翻译成高

级语言，试图重新实现这个功能；虽然暂时还不需要写汇编代码，但大量的阅读必然能加深我们对这门语言的理解。如果想在iOS逆向工程这条路上走下去，ARM汇编是必须掌握的语言，也是一定能够掌握的语言；跟英语类似，ARM汇编的基本概念相当于26个字母和音标；指令相当于单词，它们的变种相当于单词的各种形态；调用规则相当于语法，定义句子之间的联系。接下来，让我们一步步地深入。

6.1.1 基本概念

如果要完整地介绍ARM汇编，ARM公司的用户手册已经做得足够好了。笔者对ARM汇编也只是略知一二，肯定没有用户手册那么全面，但对于iOS逆向工程初学者来说，这些知识足以应对，适度就好。随着iPhone 5s的推出，苹果引入了性能强大的64位处理器，但本书前半部分介绍的大多数工具对64位处理器的支持都不太好，因此后半部分的内容仍以32位处

理器为准，但思路是通用的。

1.寄存器、内存和栈

在高级语言，如Objective-C、C和C++里，操作对象是变量；在ARM汇编里，操作对象是寄存器（register）、内存和栈（stack）。其中，寄存器可以看成CPU自带的变量，它们的数量一般是很有限的；当需要更多变量时，就可以把它们存放在内存中；不过，数量上去了，质量也下来了，对内存的操作比对寄存器的操作要慢得多。

栈其实也是一片内存区域，但它具有栈的特点：先进后出。ARM的栈是满递减（Full Descending）的，向下增长，也就是开口朝下，新的变量被存放到栈底的位置；越靠近栈底，内存地址越小，如图6-1所示。

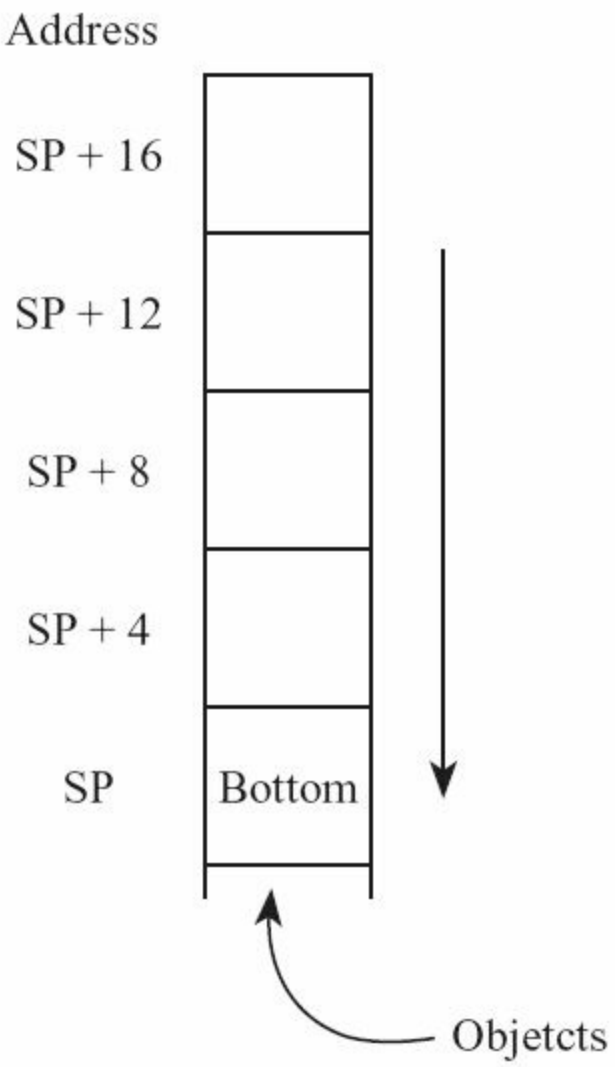


图6-1　ARM的栈

一个名为“stack pointer”（简称SP）的寄存器保存栈的栈底地址，称为栈地址；可以把一个变量给入（push）栈以保存它的值，也可以让它出（pop）栈，恢复变量的原始值。在实际操作中，栈地址会不断变化；但是在执行一块代码的前后，栈地址应该是

不变的，不然程序就要出问题了。为什么？举例说明如下：

```
static int global_var0;
static int global_var1;
…
void foo(void)
{
    bar();
    // 其他操作;
}
```

在上面4行代码中，假设函数foo()用到了A、B、C、D四个寄存器；foo()内部调用了bar()，假设bar()用到了A、B、C三个寄存器。因为2个不同的函数用到了3个相同的寄存器，所以bar()在开始执行前需要将3个寄存器中原来的值入栈以保存其原始值，在结束执行前将它们出栈以恢复其原始值，保证foo()能够正常执行。用伪汇编代码表示如下：

```
// foo()函数
foo:
    // 将A、B、C、D入栈，保存它们的原始值
    入栈    {A, B, C, D}
    // 使用A ~ D
    移动    A, #1          // A = 1
    移动    B, #2          // B = 2
    移动    C, #3          // 你猜猜这行是什么意思？
    调用    bar
```

```
        移动    D, global_var0
        // global_var1 = A + B + C + D
        相加    A, B            // A = A + B，注意此处A的值
        相加    A, C            // A = A + C，还要注意此处A的值
        相加    A, D            // 你再猜猜这行是什么意思？
        移动    global_var1, A
        // 将A、B、C、D出栈，恢复它们的原始值
        出栈    {A-D}
        返回
// bar()函数
bar:
        // 将A、B、C入栈，保存它们的原始值A == 1，B == 2，C == 3
        入栈    {A-C}
        // 使用A ~ C
        移动    A, #2           // 还需要注释吗？
        移动    B, #5
        移动    C, A
        相加    C, B            // C = 7
        // global_var0 = A + B + C (== 2 * C)
        相加    C, C
        移动    global_var0, C          // A = 2，B = 5，C = 14
        // 现在你知道入栈和出栈的重要意义了吗？
        出栈    {A-C}
        返回
```

简单解释一下这段伪代码：foo()先将A、B、C分别设置为1、2、3，然后调用bar()，bar()改变了A、B、C的值，并将全局变量global_var0的值设置为ABC三者之和。如果把此时的A、B、C直接用于foo()，计算出的另一个全局变量global_var1的值就是错的，因此在bar()执行前先要让A、B、C入栈，保存它们的值，执行完成后再出栈，使得foo()能够得到正确的global_var1。注意一点，出于同样的目的，foo()在执

行前后也对A、B、C、D执行了入栈和出栈操作，所以foo()的调用者也能够正常工作。

2.特殊用途的寄存器

ARM处理器中的部分寄存器有特殊用途，如下所示：

R0-R3	传递参数与返回值
R7	帧指针，指向母函数与被调用子函数在栈中的交界
R9	在iOS 3.0以前被系统保留
R12	内部过程调用寄存器，dynamic linker会用到它
R13	SP寄存器
R14	LR寄存器，保存函数返回地址
R15	PC寄存器

因为现在还没有开始自己写汇编代码，所以对上述知识有简单了解就足够了。

3.分支跳转与条件判断

处理器中名为“program counter”（简称PC）的寄存器用于存放下一条指令的地址。一般情况下，计算机一条接一条地顺序执行指令，处理器执行完一条指

令后将PC加1，让它指向下一条指令，如图6-2所示。

处理器顺序执行指令1到指令5，稀松平常、沉闷无聊。但是如果把PC的值变一变，指令的执行顺序就完全不同了，如图6-3所示。

指令的执行顺序被打乱，变成指令1、指令5、指令4、指令2、指令3、指令6，光怪陆离、百花齐放。这种“乱序”的学名叫“分支”（branch），或者“跳转”（jump），它使循环和subroutine成为可能，例如：

```
// endless()函数
endless:
    操作     操作数1, 操作数2
    分支     endless
    返回     // 死循环，执行不到这里啦！
```

在实际情况中，满足一定条件才得以触发的分支是最实用的，这种分支称为条件分支。if else和while都是基于条件分支实现的。在ARM汇编中，分支的条件一般有4种：

· 操作结果为0（或不为0）；

· 操作结果为负数；

· 操作结果有进位；

· 运算溢出（比如两个正数相加得到的数超过了寄存器位数）。

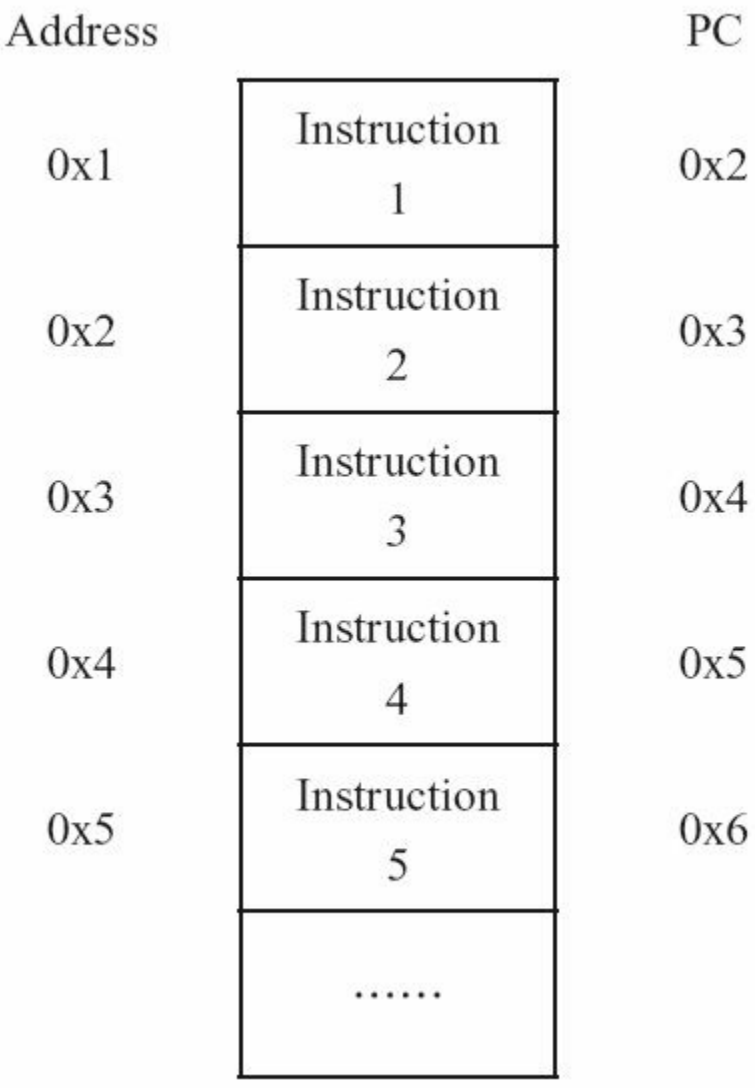


图6-2　顺序执行指令

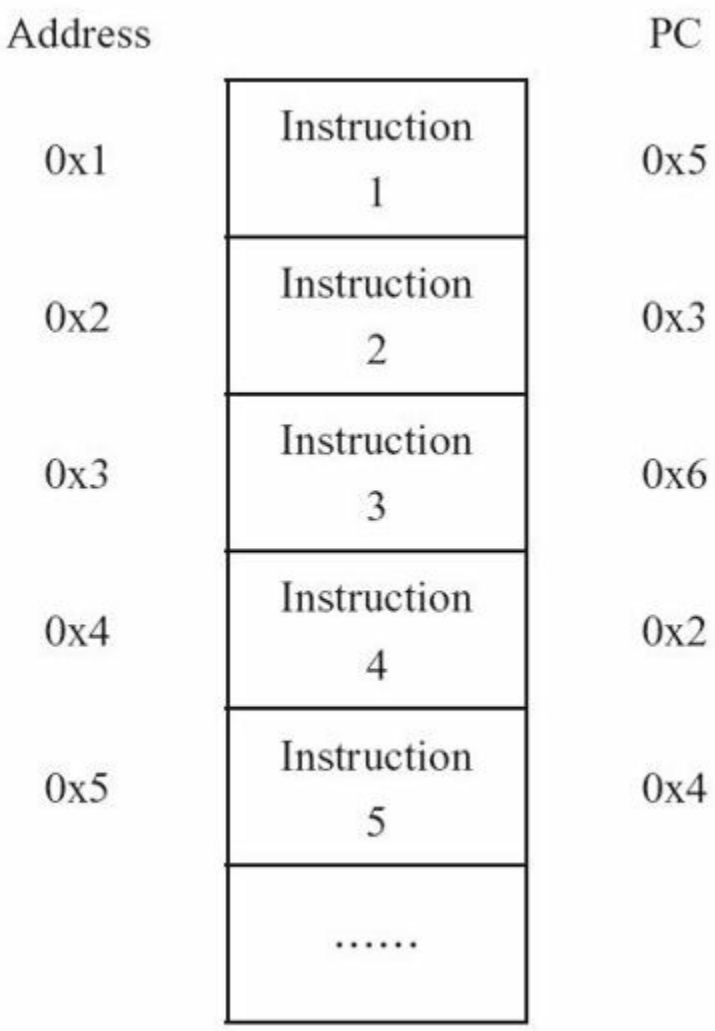


图6-3　乱序执行指令

这些条件的判断准则（flag）存放在程序状态寄存器（Program Status Register，PSR）中，数据处理相关指令会改变这些flag，分支指令再根据这些flag决定是否跳转。下面的伪代码展示了一个for循环：

```
for:
    相加     A, #1
    比较     A, #16
    不为0则跳转到for
```

此循环将A和#16作比较，如果两者不相等，则将A加1，继续比较。如果两者相等，则不再循环，继续往下执行。

6.1.2 ARM/THUMB指令解读

ARM处理器用到的指令集分为ARM和THUMB两种；ARM指令长度均为32bit，THUMB指令长度均为16bit。所有指令可大致分为3类，分别是数据操作指令、内存操作指令和分支指令。

1.数据操作指令

数据操作指令有以下2条规则：

1）所有操作数均为32bit；

2）所有结果均为32bit，且只能存放在寄存器

中。

总的来说，数据操作指令的基本格式是：

```
op{cond}{s} Rd, Rn, Op2
```

其中，“cond”和“s”是两个可选后缀；“cond”的作用是指定指令“op”在什么条件下执行，共有下面17种条件：

```
EQ              结果为0（EQual to 0）
NE              结果不为0（Not Equal to 0）
CS              有进位或借位（Carry Set）
HS              同CS（unsigned Higher or Same）
CC              没有进位或借位（Carry clear）
LO              同CC（unsigned LOwer）
MI              结果小于0（MInus）
PL              结果大于等于0（PLus）
VS              溢出（oVerflow Set）
VC              无溢出（oVerflow Clear）
HI              无符号比较大于（unsigned HIgher）
LS              无符号比较小于等于（unsigned Lower or Same）
GE              有符号比较大于等于（signed Greater than or
Equal）
LT              有符号比较小于（signed Less Than）
GT              有符号比较大于（signed Greater Than）
LE              无符号比较小于等于（signed Less than or
Equal）
AL              无条件（ALways，默认）
```

“cond”的用法很简单，例如：

```
比较 R0, R1
移动 GE R2, R0
移动 LT R2, R1
```

比较R0和R1的值，如果R0大于等于R1，则R2=R0；否则R2=R1。

“s”的作用是指定指令“op”是否设置flag，共有下面4种flag：

```
N（Negative）
如果结果小于0则置1，否则置0；
Z（Zero）
如果结果是0则置1，否则置0；
C（Carry）
对于加操作（包括CMN）来说，如果产生进位则置1，否则置0；对于减操作（包括CMP）来说，Carry相当于Not-Borrow，如果产生借位则置0，否则置1；对于有移位操作的非加/减操作来说，C置移出值的最后一位；对于其他的非加/减操作来说，C的值一般不变；
V（oVerflow）
如果操作导致溢出，则置1，否则置0。
```

需要注意一点，C flag表示无符号数运算结果是否溢出；V flag表示有符号数运算结果是否溢出。

数据操作指令可以大致分为以下4类：

· 算术操作

```
ADD R0, R1, R2          ; R0 = R1 + R2
ADC R0, R1, R2          ; R0 = R1 + R2 + C(arry)
SUB R0, R1, R2          ; R0 = R1 - R2
SBC R0, R1, R2          ; R0 = R1 - R2 - !C
RSB R0, R1, R2          ; R0 = R2 - R1
RSC R0, R1, R2          ; R0 = R2 - R1 - !C
```

算术操作中，ADD和SUB为基础操作，其他均为两者的变种。RSB是“Reverse SuB”的缩写，仅仅是把SUB的两个操作数调换了位置而已；以“C”（即Carry）结尾的变种代表有进位和借位的加减法，当产生进位或没有借位时，将Carry flag置1。

· 逻辑操作

```
AND R0, R1, R2          ; R0 = R1 & R2
ORR R0, R1, R2          ; R0 = R1 | R2
EOR R0, R1, R2          ; R0 = R1 ^ R2
BIC R0, R1, R2          ; R0 = R1 &~ R2
MOV R0, R2              ; R0 = R2
MVN R0, R2              ; R0 = ~R2
```

逻辑操作指令没什么多说的，它们的作用都已经用C操作符表示出来了，大家应该很熟悉；但是C操作符里的移位操作并没有对应的逻辑操作指令，因为ARM采用了桶式移位，共有以下4种指令：

LSL	逻辑左移，见图6-4

图6-4　逻辑左移

LSR	逻辑右移，见图6-5

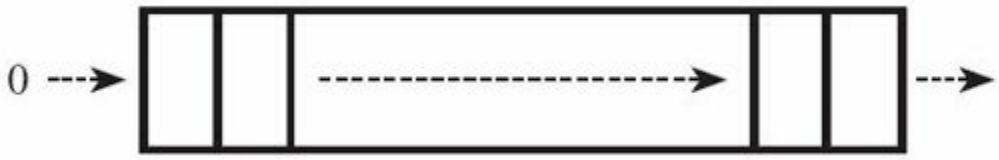


图6-5　逻辑右移

ASR	算术右移，见图6-6

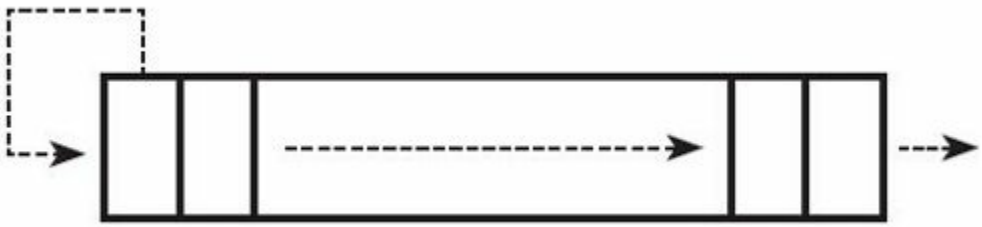


图6-6　算术右移

ROR	循环右移，见图6-7

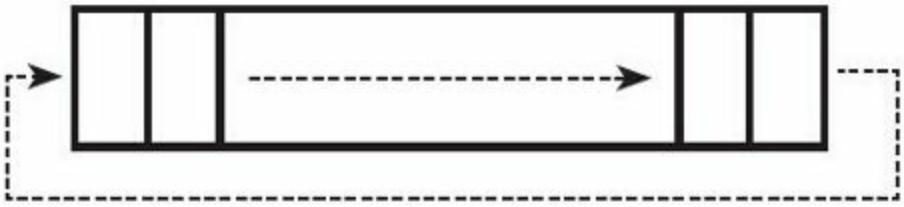


图6-7　循环右移

· 比较操作

```
CMP R1, R2              ; 执行R1 - R2并依结果设置flag
CMN R1, R2              ; 执行R1 + R2并依结果设置flag
TST R1, R2              ; 执行R1 & R2并依结果设置flag
TEQ R1, R2              ; 执行R1 ^ R2并依结果设置flag
```

比较操作其实就是改变flag的算术操作或逻辑操作，只是操作结果不保留在寄存器里而已。

· 乘法操作

```
MUL R4, R3, R2                ; R4 = R3 * R2
MLA R4, R3, R2, R1            ; R4 = R3 * R2 + R1
```

乘法操作的操作数必须来自寄存器。

2.内存操作指令

内存操作指令的基本格式是：

```
op{cond}{type} Rd, [Rn,?Op2]
```

其中Rn是基址寄存器，用于存放基地址；“cond”的作用与数据操作指令相同；“type”指定指令“op”操作的数据类型，共有4种：

```
B（unsigned Byte）
无符号byte（执行时扩展到32bit，以0填充）；
SB（Signed Byte）
有符号byte（仅用于LDR指令；执行时扩展到32bit，以符号位填充）；
H（unsigned Halfword）
无符号halfword（执行时扩展到32bit，以0填充）；
SH（Signed Halfword）
有符号halfword（仅用于LDR指令；执行时扩展到32bit，以符号位填充）。
```

如果不指定“type”，则默认数据类型是word。

ARM内存操作基础指令只有两个：LDR（LoaD Register）将数据从内存中读出来，存到寄存器中；STR（STore Register）将数据从寄存器中读出来，存到内存中。两个指令的使用情况如下：

· LDR

```
LDR Rt, [Rn {, #offset}]          ; Rt = *(Rn {+ offset}),
{}代表可选
LDR Rt, [Rn, #offset]!            ; Rt = *(Rn + offset); Rn
= Rn + offset
LDR Rt, [Rn], #offset             ; Rt = *Rn; Rn = Rn +
offset
```

· STR

```
STR Rt, [Rn {, #offset}]          ; *(Rn {+ offset}) = Rt
STR Rt, [Rn, #offset]!            ; *(Rn {+ offset}) = Rt; Rn
= Rn + offset
STR Rt, [Rn], #offset             ; *Rn = Rt; Rn = Rn +
offset
```

此外，LDR和STR的变种LDRD和STRD还可以操作双字（Doubleword），即一次性操作2个寄存器，其基本格式如下：

```
op{cond} Rt, Rt2, [Rn {, #offset}]
```

其用法与原型类似，如下：

· STRD

```
STRD R4, R5, [R9,#offset]          ;  *(R9 + offset) = R4;
*(R9 + offset + 4) = R5
```

· LDRD

```
LDRD R4, R5, [R9,#offset]          ;  R4 = *(R9 + offset);
R5 = *(R9 + offset + 4)
```

除了LDR和STR外，还可以通过LDM（LoaD Multiple）和STM（STore Multiple）进行块传输，一次性操作多个寄存器。块传输指令的基本格式是：

```
op{cond}{mode} Rd{!}, reglist
```

其中Rd是基址寄存器，可选的“!”指定Rd变化后的值是否写回Rd；reglist是一系列寄存器，用大括号括起来，它们之间可以用“,”分隔，也可以用“-”表示一个范围，比如，{R4–R6,R8}表示寄存器R4、R5、R6、R8；这些寄存器的顺序是按照自身的编号由小到大排列的，与大括号内的排列顺序无关。

需要特别注意的是，LDM和STM的操作方向与

LDR和STR完全相反：LDM是把从Rd开始，地址连续的内存数据存入reglist中，STM是把reglist中的值存入从Rd开始，地址连续的内存中。此处特别容易混淆，大家一定要注意！

“cond”的作用与数据操作指令相同。“mode”指定Rd值的4种变化规律，如下所示：

```
IA（Increment After）
每次传输后增加Rd的值；
IB（Increment Before）
每次传输前增加Rd的值；
DA（Decrement After）
每次传输后减少Rd的值；
DB（Decrement Before）
每次传输前减少Rd的值。
```

这是什么意思呢？下面以LDM为代表，举一个简单的例子，相信大家一看就明白了。在图6-8中，R0指向的值是5。

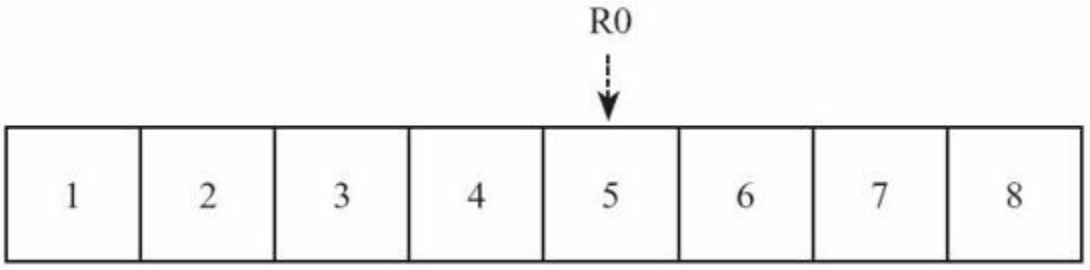


图6-8　块传输指令模拟环境

执行以下命令后，R4、R5、R6的值分别变成：

```
foo():
    LDMIA R0, {R4 - R6}                    ; R4 = 5, R5 = 6,
R6 = 7
    LDMIB R0, {R4 - R6}                    ; R4 = 6, R5 = 7,
R6 = 8
    LDMDA R0, {R4 - R6}                    ; R4 = 5, R5 = 4,
R6 = 3
    LDMDB R0, {R4 - R6}                    ; R4 = 4, R5 = 3,
R6 = 2
```

STM指令的作用方式与此类似，不再赘述。再次提醒，LDM和STM的操作方向与LDR和STR完全相反，切记切记！

3.分支指令

分支指令可以分为无条件分支和条件分支两种。

· 无条件分支

```
B Label          ; PC = Label
BL Label         ; LR = PC - 4; PC = Label
BX Rd            ; PC = Rd并切换指令集
```

无条件分支很简单，举下面一个小例子就会了解：

```
foo():
        B Label                         ; 跳转到Label处往下执行
        ......                          ; 得不到执行
Label:
        ......
```

· 条件分支

条件分支的cond是依照6.2.1节提到的4种flag来判断的，它们的对应关系如下：

```
cond            flag
EQ              Z = 1
NE              Z = 0
CS              C = 1
HS              C = 1
CC              C = 0
LO              C = 0
MI              N = 1
PL              N = 0
VS              V = 1
VC              V = 0
HI              C = 1 & Z = 0
LS              C = 0 | Z = 1
GE              N = V
LT              N !=  V
GT              Z = 0 & N = V
LE              Z = 1 | N != V
```

在条件分支指令前会有一条数据操作指令来设置flag，分支指令根据flag的值来决定代码走向，举例如下：

```
Label:
    LDR R0, [R1], #4
    CMP R0, 0              ; 如果R0 == 0，Z = 1；否则Z = 0
    BNE  Label             ; Z ==  0则跳转
```

4.THUMB指令

THUMB指令集是ARM指令集的一个子集，每条THUMB指令均为16bit；因此THUMB指令比ARM指令更节省空间，且在16位数据总线上的传输效率更高。有得必有失，除了“b”之外，所有THUMB指令均无法条件执行；桶式移位无法结合其他指令执行；大多数THUMB指令只能使用R0~R7这8个寄存器等。相对于ARM指令，THUMB指令的特点如下：

· 指令数量减少

既然THUMB只是一个子集，指令数量必然会减

少。例如，乘法指令中只有MUL保留了下来，其他的都被精简了。

· 没有条件执行

除分支指令外，其他指令无法条件执行。

· 所有指令默认附带“s”

即所有THUMB指令都会设置flag。

· 桶式移位无法结合其他指令执行

移位指令只能单独执行，无法与其他指令结合执行。即，可以：

```
LSL R0 #2
```

而不可：

```
ADD R0, R1, LSL #2
```

· 寄存器使用受限

除非显式声明，否则THUMB指令只能使用R0~R7寄存器；但也有例外：ADD、MOV和CMP指令可以将R8~R15作为操作数使用；LDR和STR可以使用PC或SP寄存器；PUSH可以使用LR，POP可以使用PC；BX可以使用所有寄存器。

· 立即数和第二操作数使用受限

大多数THUMB数据操作指令的形式是“op Rd,Rm”，只有移位指令、ADD、SUB、MOV和CMP是例外。

· 不支持数据写回

除了LDMIA和STMIA外，其他THUMB指令均不支持数据写回，即“!”不可用。

我们在iOS逆向工程初级阶段经常会碰到以上指

令，如果对前两节的内容还是一知半解，没关系，自己动手分析两个程序就熟悉了。这一节的内容只是一个引子，在实际操作中如果对指令作用不清楚，ARM的官方文档http://infocenter.arm.com永远是最好的教科书，http://bbs.iosre.com上的讨论也很有参考价值。

6.1.3 ARM调用规则

了解了常用的ARM指令后，相信大家已经能够基本读懂一个函数的汇编代码了。当一个函数调用另一个函数时，常常需要传递参数和返回值；如何传递这些数据，称为ARM汇编的调用规则。

1.前言与后记

在6.1.1节提到，“在执行一块代码时，其前后栈地址应该是不变的”，这个操作是通过被执行代码块的前言（prologs）和后记（epilogs）完成的。前言所

做的工作主要有：

· 将LR入栈；

· 将R7入栈；

· R7=SP；

· 将需要保留的寄存器原始值入栈；

· 为本地变量开辟空间。

后记所做的主要工作跟前言正好相反：

· 释放本地变量占用的空间；

· 将需要保留的寄存器原始值出栈；

· 将R7出栈；

· 将LR出栈，PC=LR。

前言和后记中的这些工作并不是必须的，如果这

块代码压根儿就没有用到栈，就不需要“保留寄存器原始值”这一步了。在逆向工程中，前言与后记的影响主要体现在SP的变化上，此处稍作了解即可，第10章的例子中会有详细的解答。

2.传递参数与返回值

如果想详细了解参数传递规则，可以通读http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042一般情况下，记住最重要的一个金句就好：

“函数的前4个参数存放在R0到R3中，其他参数存放在栈中；返回值放在R0中。”

这句话的意思很好理解，为了加深印象，下面看一个例子：

```
// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --
show-
sdk-path` -o MainBinary main.m
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("%d, %d, %d, %d, %d", 1, 2, 3, 4, 5);
    return 6;
```

```
}
```

把这段代码存成名为main.m的文件，用注释里的那句话编译它，然后把MainBinary拖进IDA，生成的main汇编代码如图6-9所示。

“BLX_printf”执行printf函数，它的6个参数分别存放在R0、R1、R2、R3、[SP,#0x20+var_20]和[SP,#0x20+var_1C]中，返回值存放在R0里，其中var_20=-0x20，var_1C=-0x1C，因此栈上的2个参数分别位于[SP]和[SP,#0x4]。

还需要更多解释吗？

“函数的前4个参数存放在R0到R3中，其他参数存放在栈中；返回值放在R0中。”

一定要牢记上面这句话！

本节只是把iOS逆向工程用到的最基本的ARM汇编知识过了一遍，难免有遗漏，但说白了，只要记住

刚才的“金句”，配合ARM官方网站，就已经可以开始分析程序了。接下来，就来实际动手，看看如何把刚刚学到的知识运用到iOS逆向工程中。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT _main
_main

var_20= -0x20
var_1C= -0x1C
var_18= -0x18
var_14= -0x14
var_10= -0x10
var_C= -0xC

PUSH            {R4,R5,R7,LR}
ADD             R7, SP, #8
SUB             SP, SP, #0x18
MOV             R2, #(aDDDDD - 0xBF6A) ; "%d, %d, %d, %d, %d"
ADD             R2, PC  ; "%d, %d, %d, %d, %d"
MOVS            R3, #1
MOV             R9, #2
MOV             R12, #3
MOV             LR, #4
MOVS            R4, #5
MOVS            R5, #0
STR             R5, [SP,#0x20+var_C]
STR             R0, [SP,#0x20+var_10]
STR             R1, [SP,#0x20+var_14]
MOV             R0, R2  ; char *
MOV             R1, R3
MOV             R2, R9
MOV             R3, R12
STR.W           LR, [SP,#0x20+var_20]
STR             R4, [SP,#0x20+var_1C]
BLX             _printf
MOVS            R1, #6
STR             R0, [SP,#0x20+var_18]
MOV             R0, R1
ADD             SP, SP, #0x18
POP             {R4,R5,R7,PC}
; End of function _main

; __text ends
```

图6-9　main的汇编代码

6.2 tweak的编写套路

在第5章的“tweak的编写套路”一节里，归纳总结了5个步骤，分别是寻找灵感、定位目标文件、定位目标函数、测试函数功能，以及解析函数参数。这些步骤没问题，但“定位目标函数”这个关键环节的水分太大——在class-dump的头文件里搜索自己感兴趣的关键词，可以称为“定位目标函数”吗？非也。

一般情况下，一个软件之所以能引起我们的兴趣，无非是2个元素：功能和数据。如果发现了自己感兴趣的功能，但class-dump的头文件里找不到可疑的关键词，怎么办？如果看到了自己感兴趣的数据，我们该怎么去寻找它的生成算法？对此，class-dump一点辙都没有。因此，通过class-dump及关键词搜索的方式只是“定位目标函数”中的一种情况，不能以偏概全。那么针对更普遍的情况，该怎么定位目标函数呢？

我们感兴趣的功能和数据，都是以软件中产生的某种现象为形式，直观地呈现在我们面前的，我们能看到、感受到。例如，图6-10所示的是邮件应用（以下简称Mail），右下角的那个书写图标代表了“编写邮件”功能；图6-11所示的是设置应用中的电话设置（以下简称MobilePhoneSettings），第一个cell中的内容代表了“本机号码”数据。功能是由函数提供的，数据是由函数生成的，也就是说，外在现象的内在本质，其实是函数。所以，“定位目标函数”实际上是如何从我们感兴趣的外在现象，定位到其内在函数的过程。

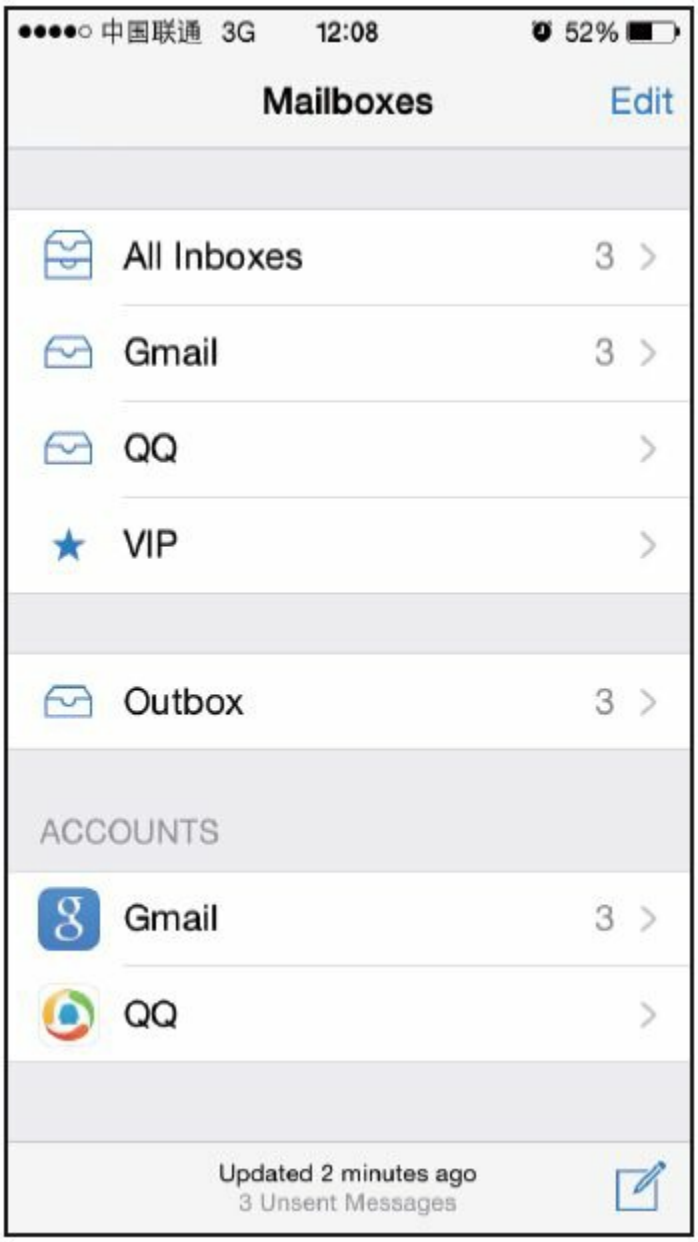

●●●●○ 中国联通 3G
12:08
52%
Mailboxes
Edit
All Inboxes
3
Gmail
3
QQ
VIP
Outbox
3
ACCOUNTS
Gmail
3
QQ
Updated 2 minutes ago
3 Unsent Messages

图6-10　Mail

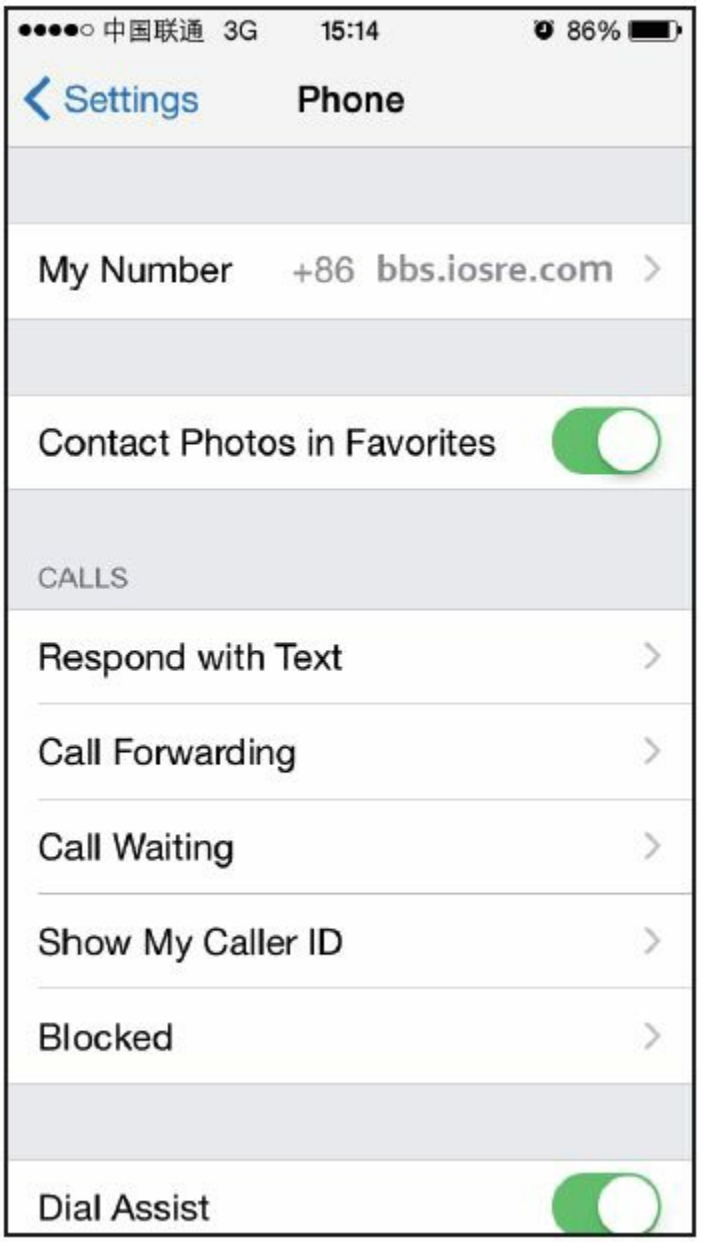


图6-11　MobilePhoneSettings

面对这样的需求，class-dump明显已经不够用了。好在我们现在了解了Cycript、IDA、LLDB的基本用法，对ARM汇编也有了初步印象，有了它们的辅助，“定位目标函数”变得有规律可循了。iOS上最

常见的是一个个App，我们对这种类型的文件也最熟悉，把它们作为初学阶段的练习对象再合适不过了。接下来，就以App为目标，用ARM汇编级别的逆向工程完善“定位目标函数”环节，强化tweak的编写套路。

6.2.1 从现象切入App，找出UI函数

对于App来说，我们感兴趣的现象往往体现在UI上，UI展示了函数的执行过程和结果。函数和UI之间的关联非常紧密，如果能拿到感兴趣的UI对象，就可以找到它所对应的函数，我们称该函数为UI函数。这个过程，一般是利用Cycript，结合UIView中的神奇私有函数recursiveDescription和UIResponder中的nextResponder来实现的。下面先以Mail为例讲解过程，然后把总结出来的方法用在MobilePhoneSettings上加深印象。这部分内容是在iPhone 5，iOS 8.1中完成的。

1.用Cycript注入Mail

先用dumpdecrypted小节中提及的技巧，定位Mail的进程名并注入，命令如下：

```
FunMaker-5:~ root# ps -e | grep /Applications
  363 ??         0:06.94
/Applications/MobileMail.app/MobileMail
  596 ??         0:01.50
/Applications/MessagesNotificationViewService.app/Message

  623 ??         0:08.50
/Applications/InCallService.app/InCallService
  713 ttys000    0:00.01 grep /Applications
FunMaker-5:~ root# cycript -p MobileMail
```

2.查看当前界面的UI层次结构，定位“编写邮件”按钮

UIView中的私有函数recursiveDescription可以返回这个view的UI层次结构。一般来说，当前界面是由至少一个UIWindow构成的，而UIWindow继承自UIView，因此可以利用这个私有函数来查看当前界面的UI层次结构。它的用法如下：

```
cy# ?expand
expand == true
```

首先执行Cycript的?expand命令开启expand功能，Cycript会把格式符号翻译成相应的格式，如“\n”会被翻译成一个换行，让输出的可读性更高。接着输入如下命令：

```
cy# [[UIApp keyWindow] recursiveDescription]
```

UIApp是[UIApplication sharedApplication]的简写，两者等价。调用上面的方法即可打印keyWindow的视图结构，输出类似下面的信息：

```
@"<UIWindow: 0x14587a70; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x147166b0>; layer =
<UIWindowLayer: 0x14587e30>>
   | <UIView: 0x146e6180; frame = (0 0; 320 568);
autoresize = W+H; gestureRecognizers = <NSArray:
0x146e98d0>; layer = <CALayer: 0x146e61f0>>
   |    | <UIView: 0x146e5f60; frame = (0 0; 320 568);
layer = <CALayer: 0x1460ec40>>
   |    |    | <_MFActorItemView: 0x14506a30; frame = (0
0; 320 568); layer = <CALayer: 0x14506c10>>
   |    |    |    | <UIView: 0x145074b0; frame = (-0.5
-0.5; 321 569); alpha = 0; layer = <CALayer: 0x14507520>>
   |    |    |    | <_MFActorSnapshotView: 0x14506f70;
baseClass = UISnapshotView; frame = (0 0; 320 568);
clipsToBounds = YES; hidden = YES; layer = <CALayer:
0x145071c0>>
……
   |    | <MFTiltedTabView: 0x146e1af0; frame = (0 0; 320
568); userInteractionEnabled = NO; gestureRecognizers =
<NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>
   |    |    | <UIScrollView: 0x146bfa90; frame = (0 0;
```

```
320 568); gestureRecognizers = <NSArray: 0x146e1e90>;
layer = <CALayer: 0x146c8740>; contentOffset: {0, 0};
contentSize: {320, 77.5}>
   |    |    | <_TabGradientView: 0x146e7010; frame =
(-320 -508; 960 568); alpha = 0; userInteractionEnabled =
NO; layer = <CAGradientLayer: 0x146e7d80>>
   |    |    | <UIView: 0x146e29c0; frame = (-10000 568;
10320 10000); layer = <CALayer: 0x146e2a30>>"
```

keyWindow的每个subview及二级subview的description会被完整展示在<……>里，包括每个view对象在内存中的地址，以及它的坐标、尺寸等基本信息。其中，缩进的多少体现了视图间的关系，同一缩进量的视图是平级的，如最下面的UIScrollView、_TabGradientView及UIView；缩进少的视图是缩进多的视图的superview，如UIScrollView、_TabGradientView和UIView都是MFTiltedTabView的subview。通过Cycript的“#”操作符，就可以拿到这个window上的任意view，如：

```
cy# tabView = #0x146e1af0
#"<MFTiltedTabView: 0x146e1af0; frame = (0 0; 320 568);
userInteractionEnabled = NO; gestureRecognizers =
<NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>"
```

当然，也可以通过UIApplication和UIView的其他方法，获取我们感兴趣的其他view，如：

```
cy# [UIApp windows]
@[#"<UIWindow: 0x14587a70; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x147166b0>; layer =
<UIWindowLayer: 0x14587e30>>",#"<UITextEffectsWindow:
0x15850570; frame = (0 0; 320 568); opaque = NO;
gestureRecognizers = <NSArray: 0x147503e0>; layer =
<UIWindowLayer: 0x1474ff10>>"]
```

上面的代码可以拿到这个App的所有window；

```
cy# [#0x146e1af0 subviews]
@[#"<UIScrollView: 0x146bfa90; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x146e1e90>; layer =
<CALayer: 0x146c8740>; contentOffset: {0, 0};
contentSize: {320, 77.5}>",#"<_TabGradientView:
0x146e7010; frame = (-320 -508; 960 568); alpha = 0;
userInteractionEnabled = NO; layer = <CAGradientLayer:
0x146e7d80>>",#"<UIView: 0x146e29c0; frame = (-10000 568;
10320 10000); layer = <CALayer: 0x146e2a30>>"]
cy# [#0x146e29c0 superview]
#"<MFTiltedTabView: 0x146e1af0; frame = (0 0; 320 568);
userInteractionEnabled = NO; gestureRecognizers =
<NSArray: 0x146f2dd0>; layer = <CALayer: 0x146e1d50>>"
```

上面的代码可以拿到subview和superview。总之，综合利用这几个函数，就可以拿到UI上的任意view，为下一步操作奠定基础。

要定位“编写邮件”按钮，就要寻找与这个按钮相关的控件。对此，一般采用的方法是排查法，对于形如<UIView：viewAddress;…>的view来说，对其逐个调用[#viewAddress setHidden:YES]函数，UI上消失的那个控件就可以跟它对应起来。当然，一些小技巧可以加快排查的速度——因为这个按钮的左边是上下两排字，所以可以猜测，这个按钮跟两排字是共用一个superview的，如果找到这个superview，那么只排查这个superview的subview就好了，减少了工作量。因为文字一般是会出现在description里的，所以可在recursiveDescription里搜索“3 Unsent Messages”，如下：

```
   |    |    |    |    |    |    |
<MailStatusUpdateView: 0x146e6060; frame = (0 0; 182 44);
opaque = NO; autoresize = W+H; layer = <CALayer:
0x146c8840>>
   |    |    |    |    |    |    |    | <UILabel:
0x14609610; frame = (40 21.5; 102 13.5); text = '3 Unsent
Messages'; opaque = NO; userInteractionEnabled = NO;
layer = <_UILabelLayer: 0x146097f0>>
```

从而获取到它的superview，即

MailStatusUpdateView。如果按钮是MailStatusUpdateView的一个subview，那么通过调用setHidden:函数隐藏MailStatusUpdateView，按钮也会被隐藏。下面试试看：

```
cy# [#0x146e6060 setHidden:YES]
```

执行之后，发现两排字被隐藏了，而按钮没有被隐藏，如图6-12所示。

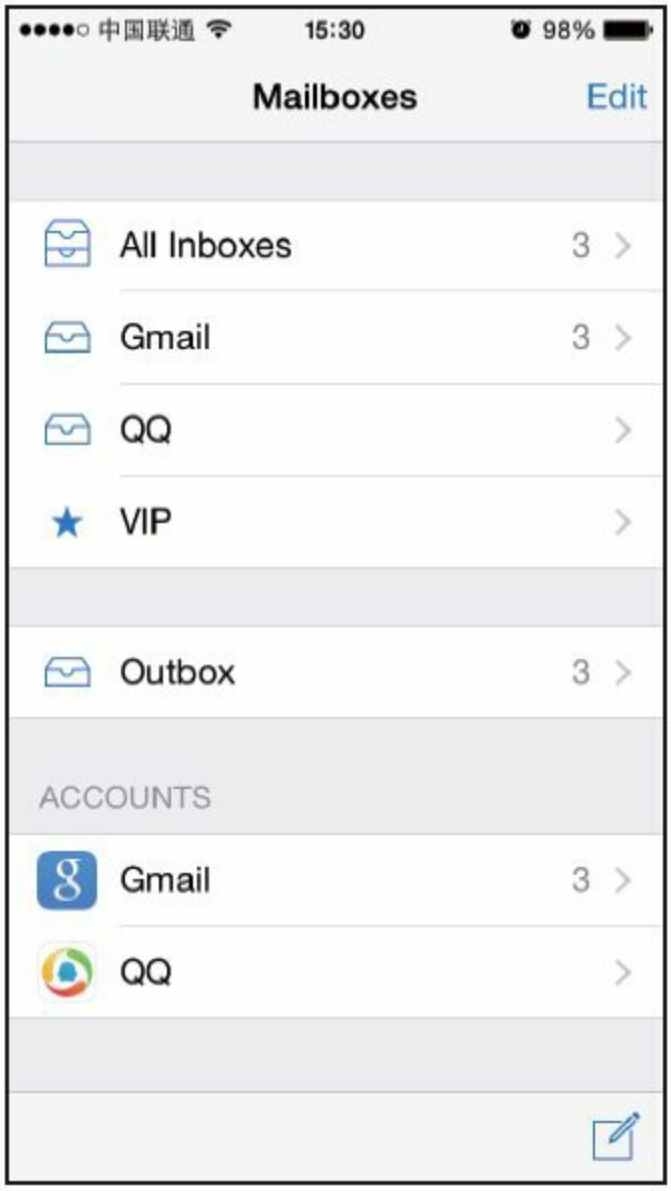


图6-12　两排字被隐藏

这说明MailStatusUpdateView的级别低于或者等于按钮所在的view，对吧？因此，接下来要做的就是排查MailStatus UpdateView的superview。从recursiveDescription可知，它的superview是

MailStatusBarView，如下：

```
   |    |    |    |    |    | <MailStatusBarView:
0x146c4110; frame = (69 0; 182 44); opaque = NO;
autoresize = BM; layer = <CALayer: 0x146f9f90>>
   |    |    |    |    |    |    |
<MailStatusUpdateView: 0x146e6060; frame = (0 0; 182 44);
opaque = NO; autoresize = W+H; layer = <CALayer:
0x146c8840>>
```

试着隐藏它，看看按钮受不受影响，如下：

```
cy# [#0x146e6060 setHidden:NO]
cy# [#0x146c4110 setHidden:YES]
```

效果跟刚才一样，两排字被隐藏，按钮还是没有被隐藏，说明MailStatusBarView的级别仍然不够高，继续找它的superview，即UIToolBar，如下：

```
   |    |    |    |    |    | <UIToolbar: 0x146f62a0;
frame = (0 524; 320 44); opaque = NO; autoresize = W+TM;
layer = <CALayer: 0x146f6420>>
   |    |    |    |    |    |    | <_UIToolbarBackground:
0x14607ed0; frame = (0 0; 320 44); autoresize = W;
userInteractionEnabled = NO; layer = <CALayer:
0x14607d40>>
   |    |    |    |    |    |    |    | <_UIBackdropView:
0x15829590; frame = (0 0; 320 44); opaque = NO;
autoresize = W+H; userInteractionEnabled = NO; layer =
<_UIBackdropViewLayer: 0x158297e0>>
   |    |    |    |    |    |    |    |    |
<_UIBackdropEffectView: 0x14509020; frame = (0 0; 320
44); clipsToBounds = YES; opaque = NO; autoresize = W+H;
```

```
userInteractionEnabled = NO; layer = <CABackdropLayer:
0x145a68d0>>
   |    |    |    |    |    |    |    | <UIView:
0x147335c0; frame = (0 0; 320 44); hidden = YES; opaque =
NO; autoresize = W+H; userInteractionEnabled = NO; layer
= <CALayer: 0x145f3ab0>>
   |    |    |    |    |    |    | <UIImageView:
0x14725730; frame = (0 -0.5; 320 0.5); autoresize = W+BM;
userInteractionEnabled = NO; layer = <CALayer:
0x1472be40>>
   |    |    |    |    |    |    | <MailStatusBarView:
0x146c4110; frame = (69 0; 182 44); opaque = NO;
autoresize = BM; layer = <CALayer: 0x146f9f90>>
```

模仿之前的操作，隐藏UIToolBar，命令如下：

```
cy# [#0x146c4110 setHidden:NO]
cy# [#0x146f62a0 setHidden:YES]
```

效果如图6-13所示。

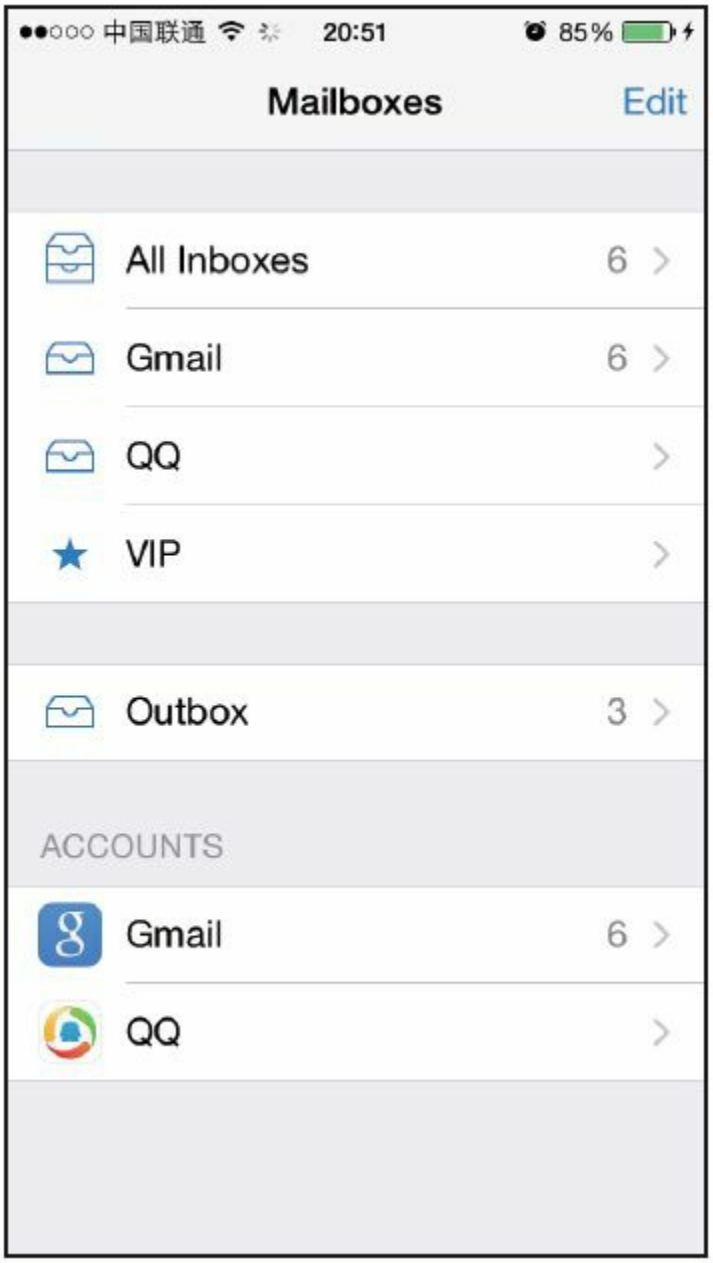


图6-13　UIToolBar被隐藏

此时，按钮被隐藏了，说明按钮是这个UIToolBar的一个subview。在这个UIToolBar的subview里面寻找带有“button”字样的view，很容易就定位到

了UIToolbarButton，如下：

```
   |    |    |    |    |    | <MailStatusBarView:
0x146c4110; frame = (69 0; 182 44); opaque = NO;
autoresize = BM; layer = <CALayer: 0x146f9f90>>
   |    |    |    |    |    |    |
<MailStatusUpdateView: 0x146e6060; frame = (0 0; 182 44);
opaque = NO; autoresize = W+H; layer = <CALayer:
0x146c8840>>
   |    |    |    |    |    |    |    | <UILabel:
0x14609610; frame = (40 21.5; 102 13.5); text = '3 Unsent
Messages'; opaque = NO; userInteractionEnabled = NO;
layer = <_UILabelLayer: 0x146097f0>>
   |    |    |    |    |    |    |    | <UILabel:
0x145f3020; frame = (43 8; 96.5 13.5); text = 'Updated
Just Now'; opaque = NO; userInteractionEnabled = NO;
layer = <_UILabelLayer: 0x145f2e50>>
   |    |    |    |    |    | <UIToolbarButton:
0x14798410; frame = (285 0; 23 44); opaque = NO;
gestureRecognizers = <NSArray: 0x14799510>; layer =
<CALayer: 0x14798510>>
```

下面看看它是不是“编写邮件”按钮，命令如下：

```
cy# [#0x146f62a0 setHidden:NO]
cy# [#0x14798410 setHidden:YES]
```

按钮被成功隐藏，如图6-14所示。

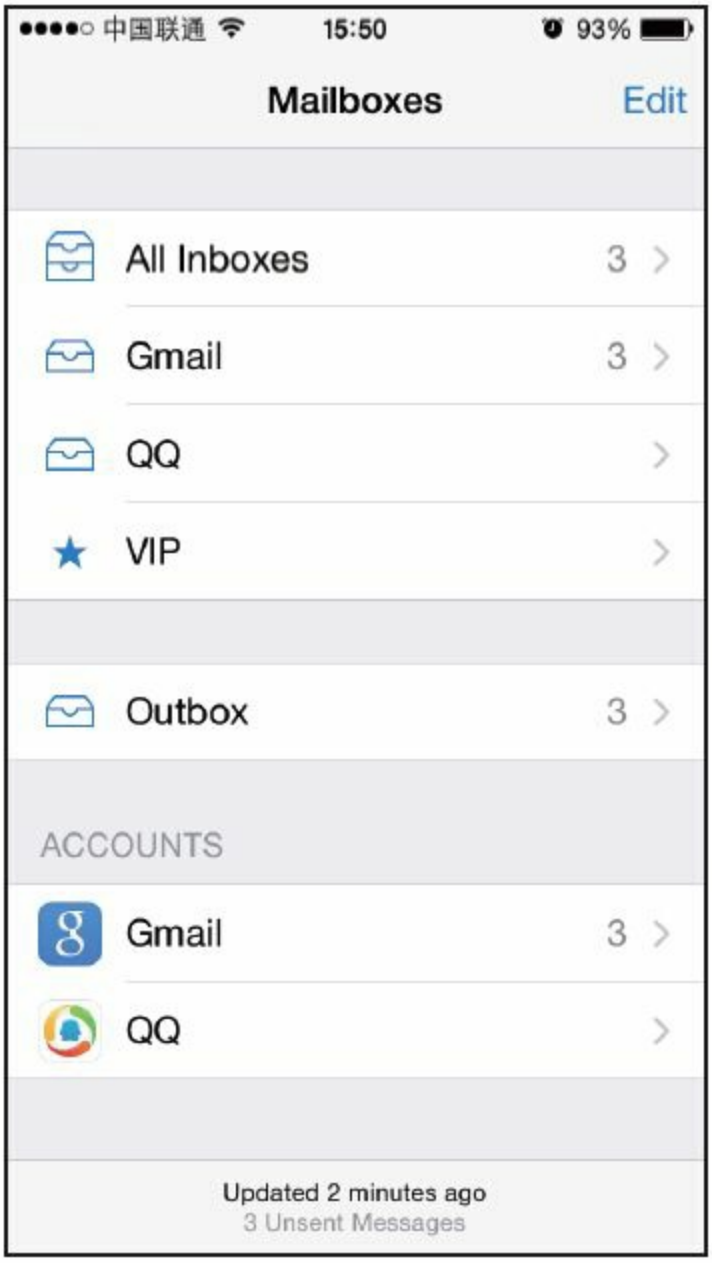


图6-14　按钮被隐藏

至此，我们成功定位到了“编写邮件”按钮，它的description是<UIToolbarButton：0x14798410;frame=(2850;2344);opaque=NO;gestureRecognizers=

<NSArray：0x14799510>;layer=<CALayer：0x14798510>>。接下来要找出它的UI函数。

3.找出“编写邮件”按钮的UI函数

按钮的UI函数，就是点击它之后的响应函数。给UIView对象加上响应函数，一般是通过[UIControl addTarget:action:forControlEvents:]实现的（笔者还没有碰到过例外）；而UIControl提供了一个actionsForTarget:forControlEvent:方法，来获得这个UIControl的响应函数。基于这个条件，只要第2步里定位到的view是UIControl的子类（笔者也还没有碰到过例外），就可以通过这种方式找到它的响应函数。具体到书中的例子，是这样操作的：

```
cy# button = #0x14798410
#"<UIToolbarButton: 0x14798410; frame = (285 0; 23 44);
hidden = YES; opaque = NO; gestureRecognizers = <NSArray:
0x14799510>; layer = <CALayer: 0x14798510>>"
cy# [button allTargets]
[NSSet setWithArray:@[#"<ComposeButtonItem:
0x14609d00>"]]]
cy# [button allControlEvents]
64
cy# [button actionsForTarget:#0x14609d00
forControlEvent:64]
```

```
@["_sendAction:withEvent:"]
```

因此，按下“编写邮件”按钮，Mail会调用[ComposeButtonItem_sendAction:withEvent:]，我们成功找到了它的响应函数。用Cycript注入，定位UI控件，找出UI函数，就这么简单。如果你还不理解，下面会用类似的套路分析MobilePhoneSettings，请注意总结。

4.用Cycript注入MobilePhoneSettings

下面的操作大家应该都很熟悉了：

```
FunMaker-5:~ root# ps -e | grep /Applications
  596 ??         0:01.50
/Applications/MessagesNotificationViewService.app/Message

  623 ??         0:08.55
/Applications/InCallService.app/InCallService
  748 ??         0:01.36
/Applications/MobileMail.app/MobileMail
  750 ??         0:01.82
/Applications/Preferences.app/Preferences
  755 ttys000    0:00.01 grep /ApplicationsFunMaker-5:~
root# cycript -p Preferences
```

注意，桌面上Settings的应用名叫Preferences，下

面会频繁出现，请大家留意。

5.查看当前界面的UI层次结构，定位第一个cell

打印出当前界面的UI层次结构如下：

```
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
@"<UIWindow: 0x17d62e00; frame = (0 0; 320 568);
autoresize = H; gestureRecognizers = <NSArray:
0x17d589b0>; layer = <UIWindowLayer: 0x17d21c60>>
   | <UILayoutContainerView: 0x17d86620; frame = (0 0;
320
568); autoresize = W+H; layer = <CALayer: 0x17d863b0>>
   |    | <UIView: 0x17ef2430; frame = (0 0; 320 0);
layer = <CALayer: 0x17ef24a0>>
   |    | <UILayoutContainerView: 0x17d7eb80; frame = (0
0; 320 568); clipsToBounds = YES; gestureRecognizers =
<NSArray: 0x17eb6400>; layer = <CALayer: 0x17d7ed60>>
……
   |    |    |    |    |    |    |    |    |    |
<PSTableCell: 0x17f92890; baseClass = UITableViewCell;
frame = (0 35; 320 44); text = 'My Number'; autoresize =
W; tag = 2; layer = <CALayer: 0x17f92a60>>
   |    |    |    |    |    |    |    |    |    |
| <UITableViewCellContentView: 0x17f92ad0; frame = (0 0;
287 43.5); gestureRecognizers = <NSArray: 0x17f92ce0>;
layer = <CALayer: 0x17f92b40>>
   |    |    |    |    |    |    |    |    |    |
|    | |<UITableViewLabel: 0x17f92d30; frame = (15 12; 90
20.5); text = 'My Number'; userInteractionEnabled = NO;
layer = <_UILabelLayer: 0x17f92df0>>
   |    |    |    |    |    |    |    |    |    |
|    | <UITableViewLabel: 0x17f93060; frame = (132.5 12;
152.5 20.5); text = '+86PhoneNumber';
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x17f93120>>
```

很容易就可以定位到显示“+86PhoneNumber”的地方，而且几乎不需要测试，就可以知道它所在的cell是PSTableCell。尝试隐藏这个cell，验证一下猜测，命令如下：

```
cy# [#0x17f92890 setHidden:YES]
```

此时，MobilePhoneSettings变成了如图6-15所示的这个样子。

所以第一个cell的description是<PSTableCell: 0x17f92890;baseClass=UITableView-Cell;frame=(035;32044);text='My Number';autoresize=W;tag=2;layer=<CALayer: 0x17f92a60>>。与刚才“编写邮件”按钮不同的是，这次的目标不是这个cell的响应函数（功能），而是它上面显示的内容（数据），actionsForTarget:forControlEvent:不再适用。面对这种情况，该怎么办呢？

在绝大多数情况下，我们感兴趣的数据不会是一个常量。如果这个数据永远显示1，笔者相信你看都不会多看它一眼。当目标是一个变量时，则要思考一个问题：这个变量来自哪里？

图6-15　隐藏第一个cell

任何变量都不是凭空出现的，它是由数据源，经过一定的算法生成的，而我们感兴趣的一般是这个算法，也就是数据源生成变量的这个过程，这个过程往往是由一个或多个函数串联而成的，它们形成了一个调用链，类似于下面的伪代码：

```
id dataSource = ?; // head
id a = function(dataSource);
id b = function(a);
id c = function(b);
…
id z = function(y);
NSString *myPhoneNumber = function(z); // tail
```

变量是已知的，也就是说，我们位于链条的尾部。逆向工程，自然就能够让我们从尾部顺着链条回溯到头部，找出这个调用链上的一个个函数，从而还原一整套算法。总的来说，还原变量的生成算法，就要在回溯的过程中记录其数据源（的数据源的数据源……，以下简称N重数据源）和函数的调用轨迹，当它的N重数据源是一个你可以决定的数据时（比如本例的数据源是——SIM卡），从N重数据源到变量

之间这段链条上的函数，就是变量的生成算法。有点不知所云？看完下面的内容，你就明白了。

6.找出第一个cell的UI函数

按照MVC设计标准（如图6-16所示），M代表model，即数据源，是未知的；V代表view，即第一个cell，是已知的；C代表controller，是未知的。M和V之间没有直接联系，而C既可以访问M又可以访问V，是三者的交流中枢。如果能够利用已知的V，获得C，不就可以访问M，找到自己的数据源了吗？这种方式从逻辑上是说得通的，在实际操作中可行吗？

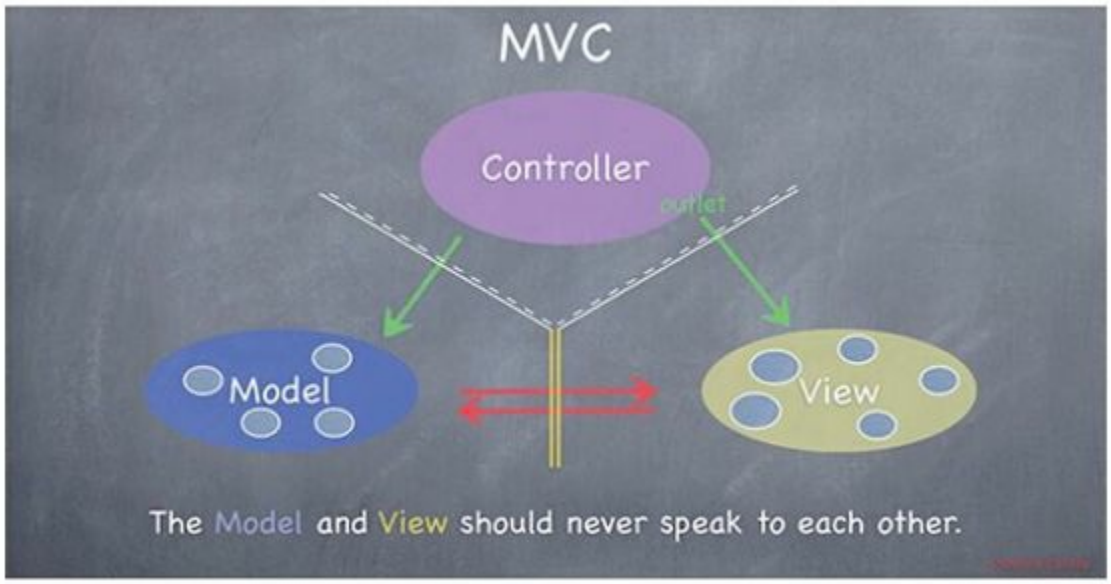


图6-16　MVC设计标准（来自Stanford CS 193P）

从笔者目前的职业经历来看，从V得到C，是100%可行的，用到的关键函数，就是在笔者心目中与recursiveDescription具有同等地位的公开函数[UIResponder nextResponder]，它的描述是这样的：

“The UIResponder class does not store or set the next responder automatically,instead returning nil by default.Subclasses must override this method to set the next responder.UIView implements this method by returning the UIViewController object that manages it(if it has one）or its superview(if it doesn’t);UIViewController implements the method by returning its view’s superview;UIWindow returns the application object,and UIApplication returns nil.”

也就是说，对于一个V，调用nextResponder，要么返回它对应的C，要么返回它的superview。因为

MVC三者缺一不可，所以C是一定存在的，也就是说，一定有一个V的nextResponder是C；又因为通过recursiveDescription可以拿到所有的V，所以从已知的V获得C是可行的，进一步就可以访问M了。

因此，我们现在的目标是拿到cell的C，操作起来很简单——从cell处开始调用nextResponder，一直到返回一个C为止，命令如下：

```
cy# [#0x17f92890 nextResponder]
#"<UITableViewWrapperView: 0x17eb4fc0; frame = (0 0; 320
504); gestureRecognizers = <NSArray: 0x17ee5230>; layer =
<CALayer: 0x17ee5170>; contentOffset: {0, 0};
contentSize: {320, 504}>"
cy# [#0x17eb4fc0 nextResponder]
#"<UITableView: 0x16c69e00; frame = (0 0; 320 568);
autoresize = W+H; gestureRecognizers = <NSArray:
0x17f4ace0>; layer = <CALayer: 0x17f4ac20>;
contentOffset: {0, -64}; contentSize: {320, 717.5}>"
cy# [#0x16c69e00 nextResponder]
#"<UIView: 0x17ebf2b0; frame = (0 0; 320 568); autoresize
= W+H; layer = <CALayer: 0x17ebf320>>"
cy# [#0x17ebf2b0 nextResponder]
#"<PhoneSettingsController 0x17f411e0: navItem
<UINavigationItem: 0x17dae890>, view <UITableView:
0x16c69e00; frame = (0 0; 320 568); autoresize = W+H;
gestureRecognizers = <NSArray: 0x17f4ace0>; layer =
<CALayer: 0x17f4ac20>; contentOffset: {0, -64};
contentSize: {320, 717.5}>>"
```

拿到了C，就可以从C所在的头文件出发，踏上

寻找M的旅途了。对于本例的情况，首先要定位PhoneSettingsController所在的目标文件，我们不确定它是来自Preferences.app本身，还是来自一个PreferenceBundle。对于这种情况，简单验证一下就好了，命令如下：

```
FunMaker-5:~ root# grep -r PhoneSettingsController
/Applications/Preferences.app/
FunMaker-5:~ root# grep -r PhoneSettingsController
/System/Library/
Binary file
/System/Library/Caches/com.apple.dyld/dyld_shared_cache_a
 matches
grep: /System/Library/Caches/com.apple.dyld/enable-
dylibs-to-override-cache: No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep: /System/Library/Frameworks/System.framework/System:
No such file or directory
Binary file
/System/Library/PreferenceBundles/MobilePhoneSettings.bun
 matches
```

看来这个类来自MobilePhoneSettings.bundle。下面class-dump它的二进制文件，然后打开

PhoneSettingsController.h，命令如下：

```
@interface PhoneSettingsController
:PhoneSettingsListController <TPSetPINView-
ControllerDelegate>
……
- (id)myNumber:(id)arg1;
- (void)setMyNumber:(id)arg1 specifier:(id)arg2;
……
- (id)tableView:(id)arg1 cellForRowAtIndexPath:(id)arg2;
@end
```

从上面的代码可以看到，前两个方法明显跟本机号码相关，而第3个方法是用来初始化所有cell的数据源函数，每个cell显示的数据一般也都与这个方法有着千丝万缕的联系。从这3个方法入手，一定可以找到第一个cell的数据源。我们用LLDB在[PhoneSettingsController tableView:cellForRowAtIndexPath:]的末尾下个断点，打印出返回值，也就是cell，看看有没有本机号码的踪迹。下面用debugserver附加Preferences，然后用LLDB连接，查看MobilePhoneSettings的ASLR偏移，如下：

```
(lldb) image list -o -f
[0] 0x00078000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.a

[1] 0x00231000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000

[2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/Bulleti

[3] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreFoundation

……
[322] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PreferenceBundles/MobileP

……
```

可以看到，MobilePhoneSettings的ASLR偏移是0x6db3000。然后在IDA中看看[Phone-SettingsController tableView:cellForRowAtIndexPath:]末尾指令的地址，如图6-17所示。

```
__text:25BB2C2A loc_25BB2C2A                            ; CODE XREF: -[PhoneSettingsController
__text:25BB2C2A                 MOV             R0, R4
__text:25BB2C2C                 ADD             SP, SP, #8
__text:25BB2C2E                 POP             {R4-R7,PC}
__text:25BB2C2E ; End of function -[PhoneSettingsController tableView:cellForRowAtIndexPath:]
```

图6-17　[PhoneSettingsController tableView:cellForRowAtIndexPath:]

因为返回值存放在R0中，所以把断点下在“ADD SP,SP,#8”上，然后返回上一级目录，再重新进入MobilePhoneSettings，待断点触发后打印R0，其中应该存放了已经初始化的cell，如下：

```
(lldb) br s -a 0x2c965c2c
Breakpoint 2: where = MobilePhoneSettings`-
[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236, address =
0x2c965c2c
Process 115525 stopped
* thread #1: tid = 0x1c345, 0x2c965c2c
MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236, queue =
'com.apple.main-thread, stop reason = breakpoint 2.1
    frame #0: 0x2c965c2c MobilePhoneSettings`-
[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236
MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 236:
-> 0x2c965c2c:  add    sp, #8
   0x2c965c2e:  pop    {r4, r5, r6, r7, pc}
MobilePhoneSettings`-[PhoneSettingsController
applicationWillSuspend]:
   0x2c965c30:  push   {r7, lr}
   0x2c965c32:  mov    r7, sp
(lldb) po $r0
<PSTableCell: 0x15f41440; baseClass = UITableViewCell;
frame = (0 0; 320 44); text = 'My Number'; tag = 2; layer
= <CALayer: 0x15f4c930>>
(lldb) po [$r0 subviews]
<__NSArrayM 0x17060e50>(
<UITableViewCellContentView: 0x15ed0660; frame = (0 0;
320 44); gestureRecognizers = <NSArray: 0x15f491e0>;
layer = <CALayer: 0x15ed06d0>>,
<UIButton: 0x15f26f50; frame = (302 16; 8 13); opaque =
NO; userInteractionEnabled = NO; layer = <CALayer:
0x15f27050>>
)
(lldb) po [$r0 detailTextLabel]
```

```
<UITableViewLabel: 0x15eb3480; frame = (0 0; 0 0); text =
'+86PhoneNumber'; userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15eb3540>>
```

可以看到，第一个cell的UI函数确实是[PhoneSettingsController tableView:cellForRow-AtIndexPath:]，我们成功完成了本节的任务。我们有信心，通过PhoneSettingsController类一定可以拿到访问M的方法，在tableView:cellForRowAtIndexPath:内部也一定有M的线索，在下一小节中就会见证。

注意，游戏一般不是采用UIKit来构建UI的，recursiveDescription和nextResponder不适用于游戏。在逆向工程初期，不建议把游戏作为练习目标。如果你在熟悉了本书的内容后想要逆向游戏，可以来http://bbs.iosre.com参与讨论。

6.2.2 以UI函数为起点，寻找目标函数

拿到UI函数，预示着首战告捷。但是，UI函数与UI是密切相关的，也就是说，要想调用

[ComposeButtonItem_sendAction:withEvent:]来编写邮件，或者调用[PhoneSettingsController tableView:cellForRowAtIndexPath:]来获取本机号码，会关联很多UI操作，比如刷新界面、尺寸布局等，有一种牵一发而动全身的感觉。在绝大多数情况下，我们不想搞得这么大张旗鼓，希望只是安静地牵一发，而不会动全身。面对这种挑战，我们该何去何从呢？

作为工程师，一定要具备基本的代码常识：最底层的函数通常是直接用汇编代码编写的，我们还接触不到；而这层以上的函数全都是嵌套调用的。UI函数也不例外——它嵌套调用了我们的目标函数。用伪代码表示如下：

```
drink GetRegular(water arg)
{
      Functions();
      return MakeRegular(arg);
}
drink GetDiet(void)
{
      Functions();
      return MakeDiet();
}
drink GetZero(void)
{
```

```
        Functions();
        return MakeZero();
}
drink GetCoke(sugar arg1, water arg2, color arg3)
{
        if (arg1 > 0 && arg1 < 3) return GetDiet();
        else if (arg1 == 0) return GetZero();
        return GetRegular(arg2);
}
drink Get7Up(void)
{
        Functions();
        return Make7Up();
}
drink GetMirinda(void)
{
        Functions();
        return MakeMirinda();
}
drink GetPepsi(sugar arg1, water arg2, color arg3)
{
        if (arg3 == clear) Get7Up();
        else if (arg3 == orange) GetMirinda();
        return GetRegular(arg2);
}
array GetDrinks(sugar arg1, color arg2) // UIFunction
{
        drink coke = GetCoke(arg1, 100, arg3);
        drink pepsi = GetPepsi(arg1, 105, arg3);
        return ArrayWithComponents(coke, pepsi)
}
```

我们不想每次都喝两种饮料（UI函数），如果只想喝七喜（数据），就要找到Get7Up（生成数据的目标函数）；如果想知道零度是怎么制作的（功能），就要找到MakeZero（提供功能的目标函数）。嵌套调用的函数之间其实也是一个链条，只要已知链条上

的一个环节，就可用通过逆向工程还原整个链条。这个过程主要用到的工具是IDA和LLDB，我们接着上面2个App例子，看看如何以[ComposeButtonItem_sendAction:withEvent:]和[PhoneSettingsController tableView:cellForRowAtIndexPath:]这2个UI函数为线索，寻找“编写邮件”和“获取本机号码”的目标函数。

1.寻找“编写邮件”的目标函数

把MobileMail丢进IDA开始分析，然后在Functions window里搜索[ComposeButtonItem_sendAction:with Event:]，如图6-18所示。

图6-18　找不到

[ComposeButtonItem_sendAction:withEvent:]

说好的

[ComposeButtonItem_sendAction:withEvent:]呢？既然ComposeButtonItem没有实现这个方法，那么去它的父类里看看。打开ComposeButtonItem.h，看看它继承自哪个类，如下：

```
@interface ComposeButtonItem : LongPressableButtonItem
+(id)composeButtonItem;
@end
```

然后打开LongPressableButtonItem.h，看看它有没有实现_sendAction:withEvent:方法，如下：

```
@interface LongPressableButtonItem : UIBarButtonItem
{
    id _longPressTarget;
    SEL _longPressAction;
}
- (void)_attachGestureRecognizerToView:(id)arg1;
- (id)createViewForNavigationItem:(id)arg1;
- (id)createViewForToolbar:(id)arg1;
- (void)longPressGestureRecognized:(id)arg1;
- (void)setLongPressTarget:(id)arg1 action:(SEL)arg2;
@end
```

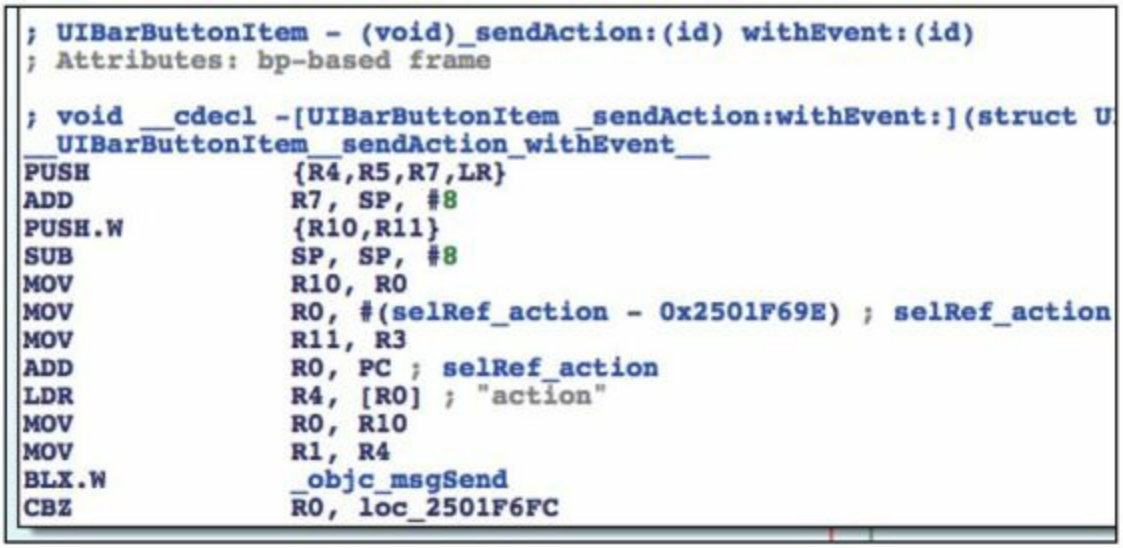
它也没有实现这个方法，那就再到它的父类里去看看。打开UIBarButtonItem.h，如下：

```
@interface UIBarButtonItem : UIBarItem <NSCoding>
……
- (void)_sendAction:(id)arg1 withEvent:(id)arg2;
……
@end
```

原来这个函数是在UIBarButtonItem类中实现的，那么把UIKit的二进制文件拖到IDA里开始分析。UIKit二进制文件较大，IDA分析耗时较长，在等待的间隙，来http://bbs.iosre.com跟大家聊聊吧！

UIKit初始分析结束后，定位到[UIBarButtonItem_sendAction:withEvent:]，如图6-19所

示。



```
; UIBarButtonItem - (void)_sendAction:(id) withEvent:(id)
; Attributes: bp-based frame

; void __cdecl -[UIBarButtonItem _sendAction:withEvent:](struct U
__UIBarButtonItem__sendAction_withEvent__
PUSH            {R4,R5,R7,LR}
ADD             R7, SP, #8
PUSH.W          {R10,R11}
SUB             SP, SP, #8
MOV             R10, R0
MOV             R0, #(selRef_action - 0x2501F69E) ; selRef_action
MOV             R11, R3
ADD             R0, PC ; selRef_action
LDR             R4, [R0] ; "action"
MOV             R0, R10
MOV             R1, R4
BLX.W           _objc_msgSend
CBZ             R0, loc_2501F6FC
```

图6-19　[UIBarButtonItem_sendAction:withEvent:]

第一个调用的函数是objc_msgSend。官方文档的注释是这样的：

“When it encounters a method call,the compiler generates a call to one of the functions objc_msgSend,objc_msgSend_stret,objc_msgSendSuper,o objc_msgSendSuper_stret.Messages sent to an object’s superclass(using the super keyword）are sent using objc_msgSendSuper;other messages are sent using

objc_msgSend.Methods that have data structures as return values are sent using objc_msgSendSuper_stret and objc_msgSend_stret.”

依据第5章中“对象”、“方法”和“实现”的关系来进一步探索，[receiver message]在编译后变成了objc_msgSend(receiver,@selector(message))；当方法有参数时，则由[receiver message:arg1 foo:arg2 bar:arg3]变成objc_msgSend(receiver,@selector(message),arg1,arg2,arg依此类推。因此，第一个objc_msgSend其实是执行了一个Objective-C方法。那么它具体执行的是什么方法呢？调用者是谁，参数又是什么呢？

还记得我们的金句吗？

“函数的前4个参数存放在R0到R3中，其他参数存放在栈中；返回值放在R0中。”

依照金句来看，objc_msgSend调用时的参数应该是objc_msgSend(R0,R1,R2,R3,*SP,*(SP+sizeOfLastArg),...)的形式，还原成等价的Objective-C方法，就是[R0 R1:R2 foo:R3 bar:*SP baz:*(SP+sizeOfLastArg）qux:...]。把这个套路运用在第一个objc_msgSend上，想知道它的等价Objective-C方法，就要看在“BLX.W_objc_msgSend”之前，R0~R3及SP都是什么。这是个从下往上倒推的分析过程，是名副其实的逆向工程。一起来看一下。

在“BLX.W_objc_msgSend”之前，R0最近的一次赋值来自“MOV R0,R10”，即R0来自R10；R10的最近一次赋值来自“MOV R10,R0”，即R10来自R0。在“MOV R10,R0”之前，R0没有被赋值就直接取值了；这显然是不合逻辑的，汇编语言不可能出现这么严重的设计漏洞。那么R0肯定还是在某个地方被赋值了——问题来了，“某个地方”是哪个地方呢？

既然在[UIBarButtonItem_sendAction:withEvent:]的内部，R0没有被赋值，那么唯一的可能就是它在[UIBarButtonItem_sendAction:withEvent:]的调用者中被赋值。[UIBarButtonItem_sendAction:withEvent:]在编译后变成了objc_msgSend(UIBarButtonItem,@selector(_sendAction:w四个参数分别放在了R0~R3中。因此，[UIBarButtonItem_sendAction:withEvent:]得到调用时，R0的值就是UIBarButtonItem，进而调用“MOV R10,R0”时的R0也是UIBarButtonItem，即调用“BLX.W_objc_msgSend”时的R0是UIBarButtonItem。有点迷糊？对照着图6-20再想一想就明白了。

同理，在“BLX.W_objc_msgSend”之前，R1最近的一次赋值来自“MOV R1,R4”，即R1来自R4；R4最近的一次赋值来自“LDR R4,[R0]”，R4来自*R0，即IDA已经标出的“action”。R1的演变过程如图6-21所

示。

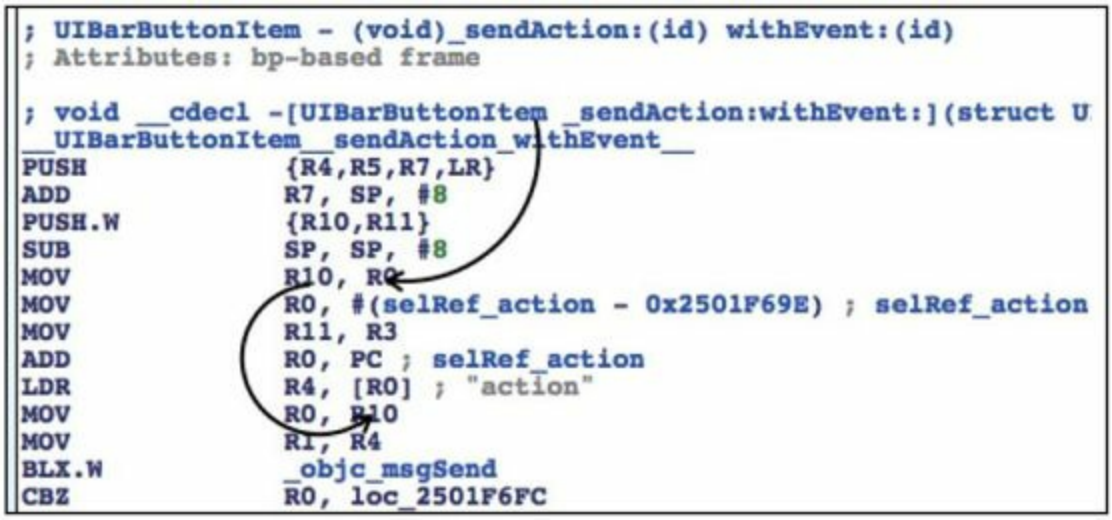
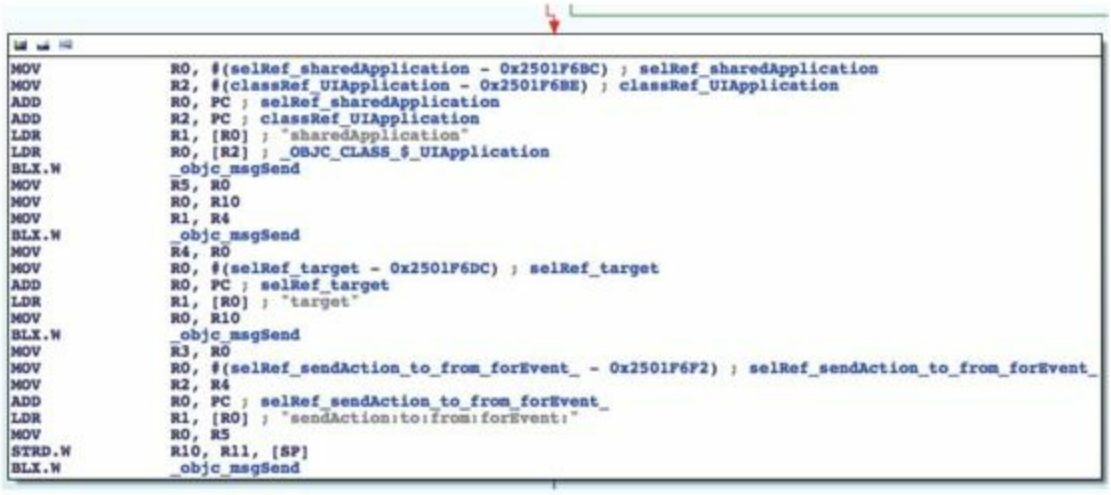


图6-20　R0的演变过程

图6-21　R1的演变过程

因此，第一个objc_msgSend还原成Objective-C方法后，是[self action]，返回值存放在接下来的R0中。没问题吧？接着进程判断[self action]是否为0，如果

是0，则不执行任何操作；否则到达图6-22。



```
MOV             R0, #(selRef_sharedApplication - 0x2501F6BC) ; selRef_sharedApplication
MOV             R2, #(classRef_UIApplication - 0x2501F6BE) ; classRef_UIApplication
ADD             R0, PC ; selRef_sharedApplication
ADD             R2, PC ; classRef_UIApplication
LDR             R1, [R0] ; "sharedApplication"
LDR             R0, [R2] ; _OBJC_CLASS_$_UIApplication
BLX.W           _objc_msgSend
MOV             R5, R0
MOV             R0, R10
MOV             R1, R4
BLX.W           _objc_msgSend
MOV             R4, R0
MOV             R0, #(selRef_target - 0x2501F6DC) ; selRef_target
ADD             R0, PC ; selRef_target
LDR             R1, [R0] ; "target"
MOV             R0, R10
BLX.W           _objc_msgSend
MOV             R3, R0
MOV             R0, #(selRef_sendAction_to_from_forEvent_ - 0x2501F6F2) ; selRef_sendAction_to_from_forEvent_
MOV             R2, R4
ADD             R0, PC ; selRef_sendAction_to_from_forEvent_
LDR             R1, [R0] ; "sendAction:to:from:forEvent:"
MOV             R0, R5
STRD.W          R10, R11, [SP]
BLX.W           _objc_msgSend
```

图6-22　[UIBarButtonItem_sendAction:withEvent:]

又是4个objc_msgSend，从上到下逐个分析：

第一个objc_msgSend的R0来自“LDR R0,[R2]”，IDA已经分析出[R2]是UIApplication类；R1来自“LDR R1,[R0]”，即“sharedApplication”，因此第一个objc_msgSend还原成Objective-C方法就是[UIApplication sharedApplication]，且返回值放入R0。

第二个objc_msgSend的R0来自“MOV R0,R10”，即R10；在图6-20中，我们知道R10的值是

UIBarButtonItem；R1来自“MOV R1,R4”，即R4；在图6-21中，R4的值是“action”。因此第二个objc_msgSend还原成Objective-C方法就是[UIBarButtonItem action]，并将返回值存放在R0中。

第三个objc_msgSend的R0仍来自“MOV R0,R10”，即UIBarButtonItem；R1来自“LDR R1, [R0]”，即“target”。因此第三个objc_msgSend还原成Objective-C方法就是[UIBarButtonItem target]，并将返回值保存在R0中。

第四个objc_msgSend的R0来自“MOV R0,R5”，即R5；R5来自第一个objc_msgSend下方的“MOV R5,R0”，即R0；R0是什么呢？因为第一个objc_msgSend执行之后，把返回值存放在了R0里，所以这个R0就是[UIApplication sharedApplication]的返回值，它是objc_msgSend的第一个参数。R1来自“LDR R1,[R0]”，即“sendAction:to:from:forEvent:”，这是一

个有4个参数的方法，加上objc_msgSend的前2个参数，一共6个参数，因此R0~R3寄存器不够用了，有2个参数要放在栈上。R2来自“MOV R2,R4”，即R4；R4来自第二个objc_msgSend下方的“MOV R4,R0”，即R0；R0来自第二个objc_msgSend执行之后的返回值，即[UIBarButtonItem action]，这是第3个参数。R3来自第三个objc_msgSend下方的“MOV R3,R0”，即R0；R0来自第三个objc_msgSend执行之后的返回值，[UIBarButtonItem target]，这是第4个参数。接下来的2个参数来自栈，而在第四个objc_msgSend以前，栈的最近一次改动来自“STRD.W R10,R11,[SP]”，即先后把R10和R11入栈，因此接下来的2个参数就是R10和R11。R10是刚才已经分析了好几遍的UIBarButtonItem，而R11来自图6-21的“MOV R11,R3”，即R3；R3又是一个没有被赋值就直接取值的寄存器，因此它也是来自[UIBarButtonItem_sendAction:withEvent:]的调用者。根

据之前的分析，R11就是_sendAction:withEvent:的第二个参数，即event。这4个objc_msgSend的参数关系可以用图6-23和图6-24表示。

这样看来，[UIBarButtonItem_sendAction:withEvent:]内最关键的就是[[UIApplication sharedApplication]sendAction:[self action]to:[self target]from:self forEvent:event]这个方法了。因为已经知道[UIBarButtonItem_sendAction:withEvent:]会执行“编写邮件”操作，所以[[UIApplication sharedApplication]sendAction:[self action]to:[self target]from:self forEvent:event]肯定会得到调用。虽然上面用IDA厘清了每个参数的来源，但是这些参数在运行时的值是什么，用IDA仍看不出来；是时候借助LLDB的威力了，一起来看看在运行时这段代码都做了些什么。

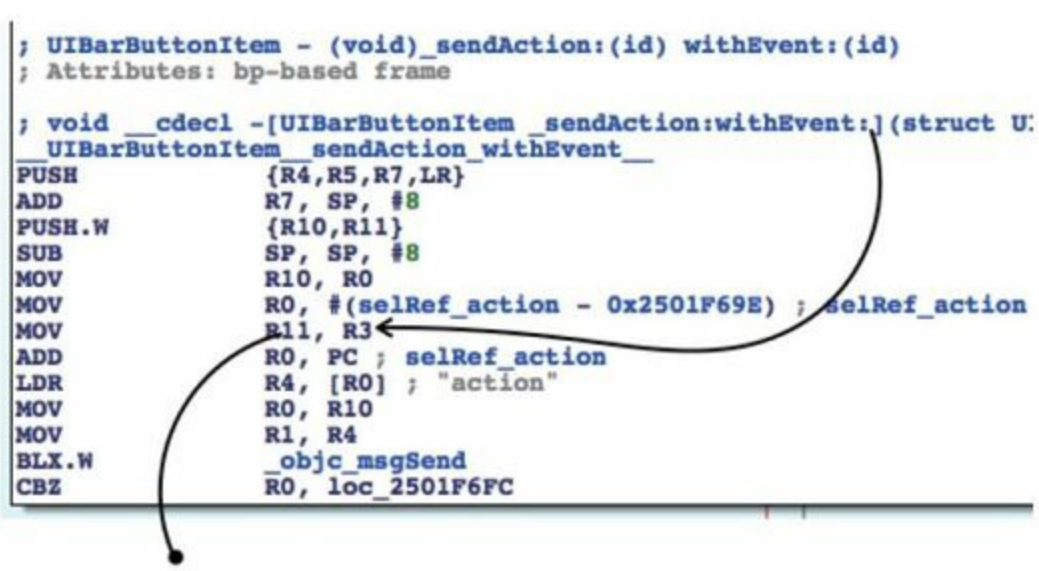


图6-23　objc_msgSend的参数关系（1）

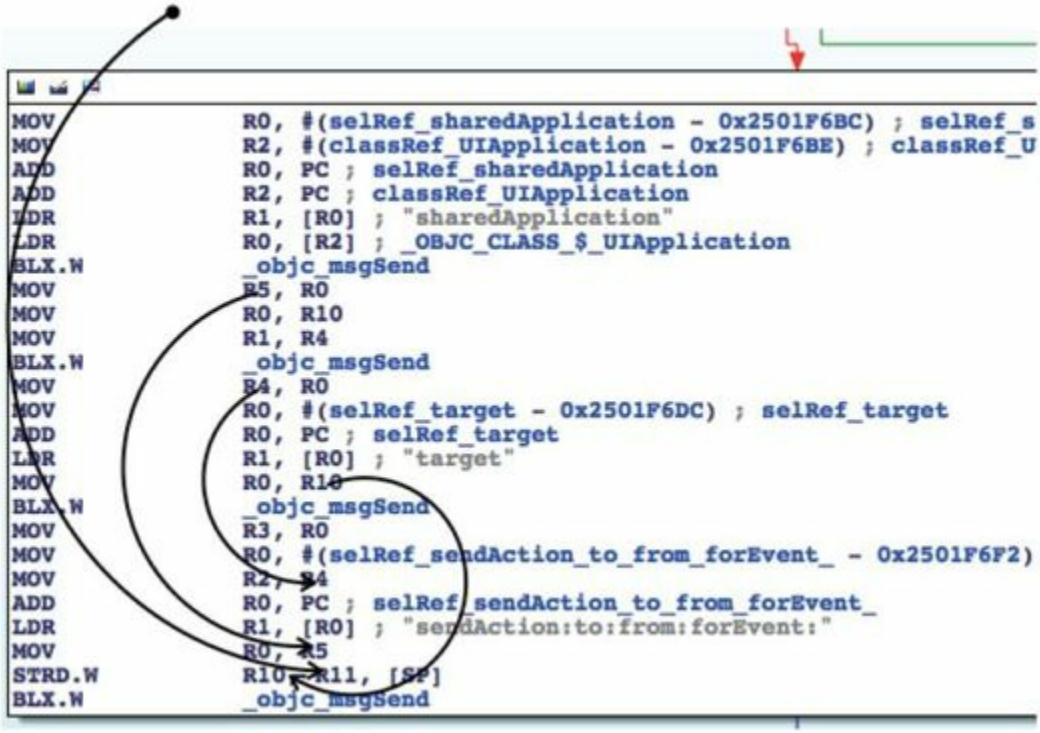


图6-24　objc_msgSend的参数关系（2）

用debugserver附加MobileMail，然后用LLDB连过

去，打印出UIKit的ASLR偏移，如下：

```
(lldb) image list -o -f
[0]
0x0008e000/private/var/db/stash/_.29LMeZ/Applications/Mob

[1]
0x00393000/Library/MobileSubstrate/MobileSubstrate.dylib(

[2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/usr/lib/libarchive.2.dylib
……
[45] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOSDeviceSuppor

……
```

UIKit的ASLR偏移是0x6db3000。再看看第四个objc_msgSend地址是多少，如图6-25所示。

```
__text:2501F6F0                 LDR             R1, [R0] ; "sendAction:to:from:forEvent:"
__text:2501F6F2                 MOV             R0, R5
__text:2501F6F4                 STRD.W          R10, R11, [SP]
__text:2501F6F8                 BLX.W           _objc_msgSend
__text:2501F6FC
__text:2501F6FC loc_2501F6FC                    ; CODE XREF: -[UIBarButtonItem _s
__text:2501F6FC                 ADD             SP, SP, #8
__text:2501F6FE                 POP.W           {R10,R11}
__text:2501F702                 POP             {R4,R5,R7,PC}
__text:2501F702 ; End of function -[UIBarButtonItem _sendAction:withEvent:]
```

图6-25　查看objc_msgSend的地址

在0x6db3000+0x2501F6F8=0x2BDD26F8上下个断点，然后按下“编写邮件”按钮触发断点，看看

[[UIApplication sharedApplication]sendAction:[self action]to:[self target]from:self forEvent:eventFromArg2]的几个参数都是什么，如下：

```
(lldb) br s -a 0x2BDD26F8
Breakpoint 4: where = UIKit`-[UIBarButtonItem(UIInternal)
_sendAction:withEvent:] + 116, address = 0x2bdd26f8
Process 44785 stopped
* thread #1: tid = 0xaef1, 0x2bdd26f8 UIKit`-
[UIBarButtonItem(UIInternal) _sendAction:withEvent:] +
116, queue = 'com.apple.main-thread, stop reason =
breakpoint 4.1
    frame #0: 0x2bdd26f8 UIKit`-
[UIBarButtonItem(UIInternal) _sendAction:withEvent:] +
116
UIKit`-[UIBarButtonItem(UIInternal)
_sendAction:withEvent:] + 116:
-> 0x2bdd26f8:  blx    0x2c3539f8                ; symbol
stub for: roundf$shim
   0x2bdd26fc:  add    sp, #8
   0x2bdd26fe:  pop.w  {r10, r11}
   0x2bdd2702:  pop    {r4, r5, r7, pc}
(lldb) p (char *)$r1
(char *) $48 = 0x2c3de501 "sendAction:to:from:forEvent:"
(lldb) po $r0
<MailAppController: 0x176a8820>
(lldb) po $r2
[no Objective-C description available]
(lldb) p (char *)$r2
(char *) $51 = 0x2d763308 "composeButtonClicked:"
(lldb) po $r3
<nil>
(lldb) x/10 $sp
0x00391198: 0x1776d640 0x176a8ce0 0x1760f5e0 0x00000000
0x003911a8: 0x2c4140f2 0x1776ff50 0x003911cc 0x2bc6ec2b
0x003911b8: 0x176a8ce0 0x00000001
(lldb) po 0x1776d640
<ComposeButtonItem: 0x1776d640>
(lldb) po 0x176a8ce0
<UITouchesEvent: 0x176a8ce0> timestamp: 58147.4 touches:
{(
    <UITouch: 0x1895e2b0> phase: Ended tap count: 1
```

```
window: <UIWindow: 0x17759c30; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x1775c7a0>; layer =
<UIWindowLayer: 0x1752e190>> view: <UIToolbarButton:
0x1776ff50; frame = (285 0; 23 44); opaque = NO;
gestureRecognizers = <NSArray: 0x17758670>; layer =
<CALayer: 0x17770160>> location in window: {308, 534}
previous location in window: {304.5, 534} location in
view: {23, 10} previous location in view: {19.5, 10}
)}
```

其中，objc_msgSend的参数R0~R3很容易理解，分别是self、@selector(sendAction:to:from:forEvent:)、sendAction:的参数和to:的参数，直接打印寄存器就可以了。注意，在执行“po$r2”的时候，LLDB提示“no Objective-C description available”，即R2不是一个Objective-C对象，结合“action”的含义，笔者猜测它是一个SEL，就用“p(char*)$r2”打印了它。如何解析栈中的参数呢？因为SP是指向栈底的指针，而我们知道余下的2个参数都在栈中，且大小均为1个字，所以，可用“x/10$sp”打印从栈底开始的连续10个字，前2个字就是from:和forEvent:的参数。Objective-C方法的大多数参数都是1个字长度的指针，指向一个Objective-C对象，因此我们“po”了前2个字，把参数打印了出

来。为了更便于理解，这里SP、栈上存储的值和参数的关系，可以参考图6-26。

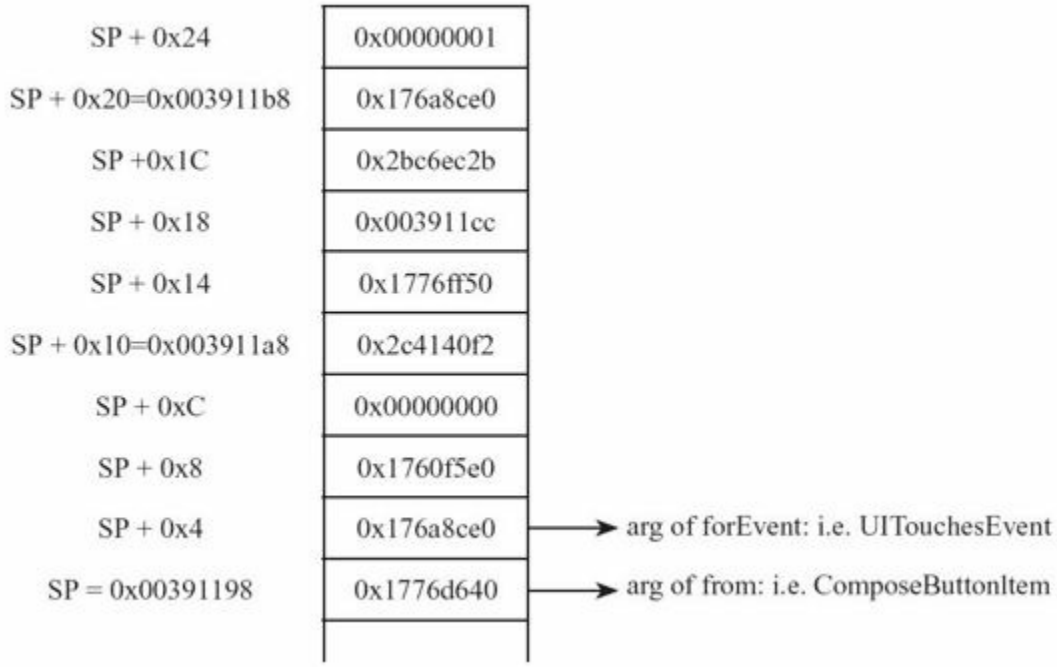


图6-26　SP、栈值和参数的关系

一般情况下，Objective-C方法在栈中的参数不会超过10个，“x/10$sp”就足够了，挨个打印，就能找到栈上的所有参数。

结合IDA和LLDB，我们知道[UIBarButtonItem_sendAction:withEvent:]的核心在于[MailAppController

sendAction:@selector(composeButtonClicked:）to:nil from:ComposeButtonItem forEvent:UITouchesEvent]，离“编写邮件”的目标函数又近了一层。下面在IDA里看看[UIApplication sendAction:to:from:forEvent:]的内部做了些什么，如图6-27所示。

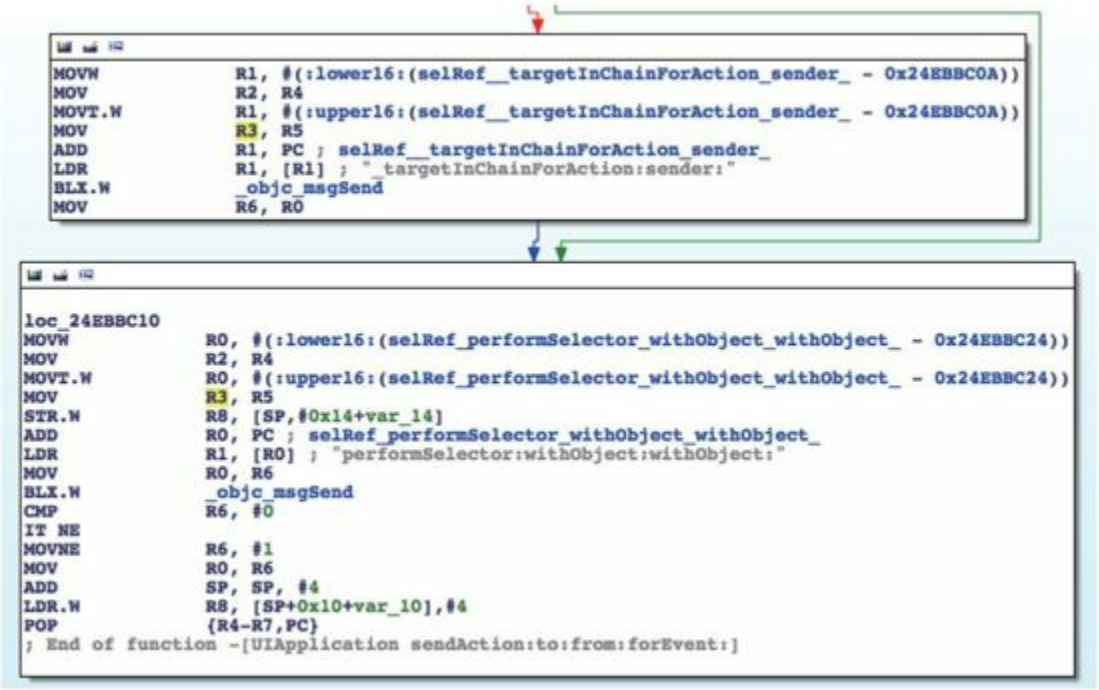


图6-27　[UIApplication sendAction:to:from:forEvent:]

无论如何，loc_24ebbc10中的“performSelector:withObject:withObject:”都会得到执行，我们自然猜测它就是做出实际操作的地方。跟刚

才一样，用LLDB看看这个方法到底执行了什么操作。UIKit的ASLR偏移是0x6db3000，最下面的那个objc_msgSend地址是0x24EBBC26，故而在0x6db3000+0x24EBBC26=0x2BC6EC26上下断点，然后按下“编写邮件”按钮触发断点，再看看这个方法的参数，如下：

```
(lldb) br s -a 0x2BC6EC26
Breakpoint 1: where = UIKit`-[UIApplication
sendAction:to:from:forEvent:] + 66, address = 0x2bc6ec26
Process 226191 stopped
* thread #1: tid = 0x3738f, 0x2bc6ec26 UIKit`-
[UIApplication sendAction: to:from:forEvent:] + 66, queue
= 'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x2bc6ec26 UIKit`-[UIApplication
sendAction:to:from:forEvent:] + 66
UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66:
-> 0x2bc6ec26:  blx    0x2c3539f8                ; symbol
stub for: roundf$shim
   0x2bc6ec2a:  cmp    r6, #0
   0x2bc6ec2c:  it     ne
   0x2bc6ec2e:  movne  r6, #1
(lldb) p (char *)$r1
(char *) $0 = 0x2c3dac95
"performSelector:withObject:withObject:"
(lldb) po $r0
<ComposeButtonItem: 0x14ddf5f0>
(lldb) p (char *)$r2
(char *) $2 = 0x2c4140f2 "_sendAction:withEvent:"
(lldb) po $r3
<UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44);
opaque = NO; gesture Recognizers = <NSArray: 0x14d22ec0>;
layer = <CALayer: 0x14d73ea0>>
(lldb) x/10 $sp
0x003735a8: 0x160a6120 0x00000001 0x14d73c90 0x160a6120
0x003735b8: 0x2c3d9be5 0x003735d4 0x2bc6ebd1 0x14d73c90
0x003735c8: 0x160a6120 0x00000040
```

```
(lldb) po 0x160a6120
<UITouchesEvent: 0x160a6120> timestamp: 73509.2 touches:
{(
    <UITouch: 0x14ff2f20> phase: Ended tap count: 1
window: <UIWindow: 0x14d878b0; frame = (0 0; 320 568);
autoresize = W+H; gestureRecognizers = <NSArray:
0x14dba890>; layer = <UIWindowLayer: 0x14d87a30>> view:
<UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44);
opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>;
layer = <CALayer: 0x14d73ea0>> location in window: {308,
545} previous location in window: {308, 545} location in
view: {23, 21} previous location in view: {23, 21}
)}
```

这是怎么回事？

performSelector:withObject:withObject:调用了[ComposeButtonItem_sendAction:withEvent:]，而[ComposeButtonItem_sendAction:withEvent:]又会调用performSelector:withObject:withObject:，如果它再次调用[ComposeButtonItem_sendAction:withEvent:]，那这段代码就出现循环调用了，与观察到的现象不符，也是不合常理的。那我们执行一下“c”命令，断点一定会被再次触发，看看performSelector:withObject:withObject:有没有发生变化，如下：

```
(lldb) c
```

```
Process 226191 resuming
Process 226191 stopped
* thread #1: tid = 0x3738f, 0x2bc6ec26 UIKit`-
[UIApplication sendAction:to:from:forEvent:] + 66, queue
= 'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x2bc6ec26 UIKit`-[UIApplication
sendAction:to:from:forEvent:] + 66
UIKit`-[UIApplication sendAction:to:from:forEvent:] + 66:
-> 0x2bc6ec26:  blx    0x2c3539f8                ; symbol
stub for: roundf$shim
   0x2bc6ec2a:  cmp    r6, #0
   0x2bc6ec2c:  it     ne
   0x2bc6ec2e:  movne  r6, #1
(lldb) p (char *)$r1
(char *) $6 = 0x2c3dac95
"performSelector:withObject:withObject:"
(lldb) po $r0
<MailAppController: 0x14e7a7a0>
(lldb) p (char *)$r2
(char *) $7 = 0x2d763308 "composeButtonClicked:"
(lldb) po $r3
<ComposeButtonItem: 0x14ddf5f0>
(lldb) x/10 $sp
0x0037356c: 0x160a6120 0x160a6120 0x2d763308 0x14e7a7a0
0x0037357c: 0x14ddf5f0 0x003735a0 0x2bdd26fd 0x14ddf5f0
0x0037358c: 0x160a6120 0x160fbdf0
(lldb) po 0x160a6120
<UITouchesEvent: 0x160a6120> timestamp: 73509.2 touches:
{(
    <UITouch: 0x14ff2f20> phase: Ended tap count: 1
window: <UIWindow: 0x14d878b0; frame = (0 0; 320 568);
autoresize = W+H; gestureRecognizers = <NSArray:
0x14dba890>; layer = <UIWindowLayer: 0x14d87a30>> view:
<UIToolbarButton: 0x14d73c90; frame = (285 0; 23 44);
opaque = NO; gestureRecognizers = <NSArray: 0x14d22ec0>;
layer = <CALayer: 0x14d73ea0>> location in window: {308,
545} previous location in window: {308, 545} location in
view: {23, 21} previous location in view: {23, 21}
)}
```

可以看到，performSelector:withObject:withObject:的参数发生了变化，[MailAppController

composeButtonClicked:ComposeButtonItem]得到了调用，如果再“c”一下，发现断点不再触发，所以可以确定执行实际操作的是composeButtonClicked:。因为在MobileMail内部，调用[UIApplication sharedApplication]可以拿到MailAppController对象；而在本小节开始的时候，我们在ComposeButtonItem.h里看到了可以通过一个类方法+composeButtonItem来拿到ComposeButtonItem对象；所以我们可以拿到调用[MailAppController composeButtonClicked:ComposeButtonItem]所需的全部对象，且在MobileMail的内部任何地方都可以调用这个方法，它可以算作是“编写邮件”的目标函数了。

在Cycript里做最后测试，看看这个目标函数是否好用，命令如下：

```
FunMaker-5:~ root# cycript -p MobileMail
cy# [UIApp composeButtonClicked:[ComposeButtonItem
composeButtonItem]]
```

执行后成功调出“编写邮件”界面。在本例中，我们用IDA追踪函数的调用链，找到目标函数，然后用LLDB解析出了它的参数，虽然有点复杂，但其实不难，不是吗？接下来，将用类似的套路来找出“获取本机号码”的目标函数，请大家注意总结。

2.寻找“获取本机号码”的目标函数

接着上面的内容，根据找到的UI函数[PhoneSettingsController tableView:cellForRowAtIndexPath:]继续往下分析。因为UI函数的返回值存放在R0中，而从图6-17的“MOV R0,R4”可知，R0来自R4。在[PhoneSettingsController tableView:cellForRowAtIndex Path:]里，R4只在图6-28里的“MOV R4,R0”处被赋值了一次，这里的R0来自objc_msgSendSuper2执行后的返回值。

objc_msgSendSuper2没有出现在文档中，由图6-29可知，它来自“/usr/lib/libobjc.A.dylib”。

按字面意思理解，objc_msgSendSuper2的作用应该跟objc_msgSendSuper类似，即向调用者的父类发送消息。不用做过多猜测，在这个objc_msgSendSuper2下个断点，看看它的参数和返回值就知道了。用debugserver附加Preferences，用LLDB连接，然后打印出MobilePhoneSettings的ASLR偏移，如下：

```
; id __cdecl -[PhoneSettingsController tableView:cellForRowAtIndexPath:]
__PhoneSettingsController_tableView_cellForRowAtIndexPath__

var_14= -0x14
var_10= -0x10

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
SUB             SP, SP, #8
MOV             R6, R0
MOV             R0, #(classRef_PhoneSettingsController - 0x25BB2B58) ; c
STR             R6, [SP,#0x14+var_14]
MOV             R5, R3
ADD             R0, PC ; classRef_PhoneSettingsController
LDR             R0, [R0] ; _OBJC_CLASS_$_PhoneSettingsController
MOV             R1, #(selRef_tableView_cellForRowAtIndexPath_ - 0x25BB2B
ADD             R1, PC ; selRef_tableView_cellForRowAtIndexPath_
LDR             R1, [R1] ; "tableView:cellForRowAtIndexPath:"
STR             R0, [SP,#0x14+var_10]
MOV             R0, SP
BLX             _objc_msgSendSuper2
MOV             R4, R0
```

图6-28　R4的来源

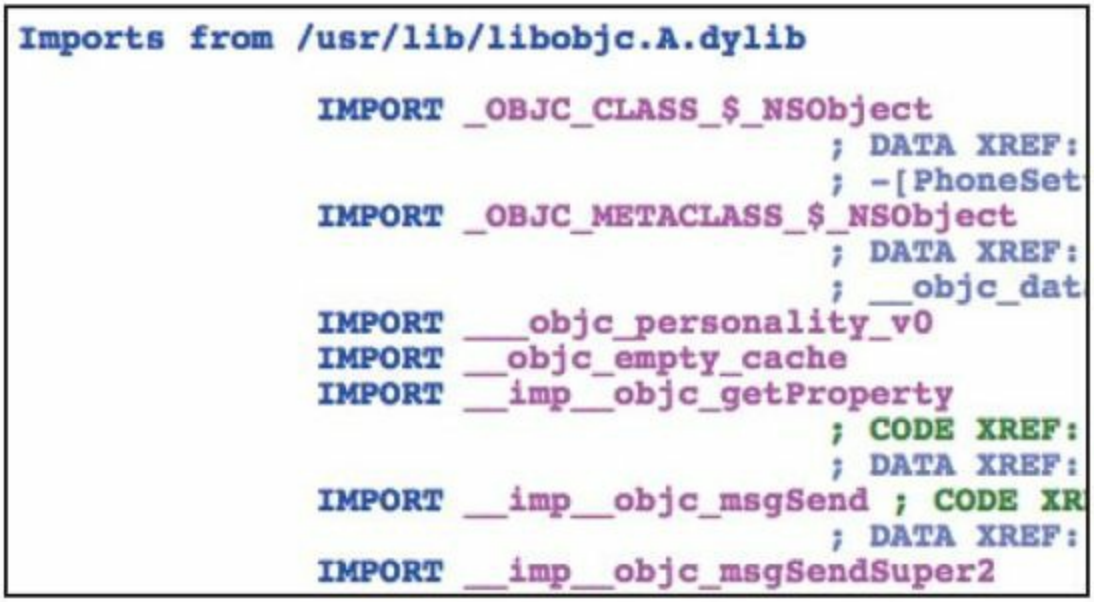


图6-29　objc_msgSendSuper2的来源

```
(lldb) image list -o -f
[  0] 0x00079000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.a

[  1] 0x00232000
/Library/MobileSubstrate/MobileSubstrate.dylib
(0x0000000000232000)
[  2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/Bulleti

[  3] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreFoundation

……
[330] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PreferenceBundles/MobileP

……
```

MobilePhoneSettings的ASLR偏移是0x6db3000。然后看看objc_msgSendSuper2的地址，如图6-30所示。

断点的地址应该是0x6db3000+0x25BB2B68=0x2C965B68。返回上一级目录，再进入MobilePhoneSettings触发断点，如下：

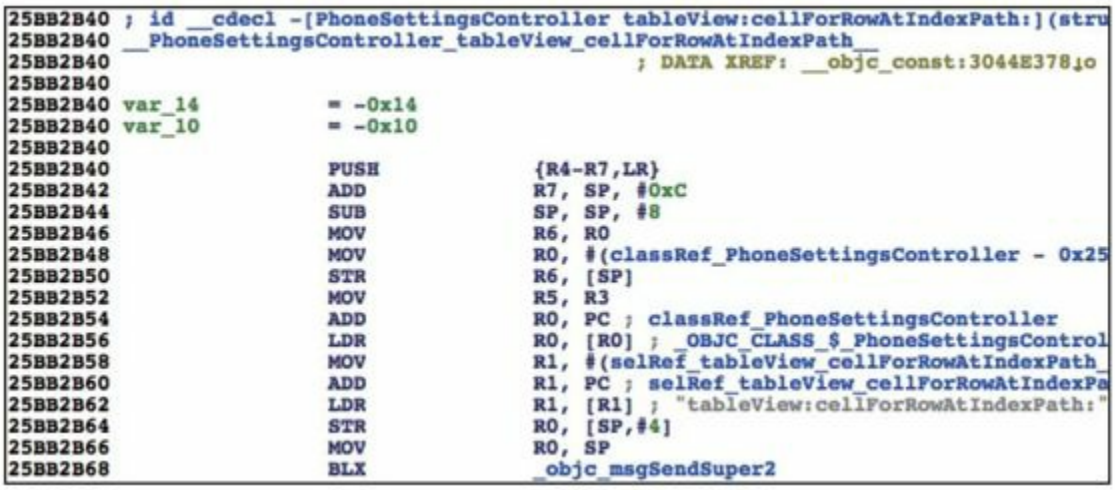


图6-30　查看objc_msgSendSuper2的地址

```
(lldb) br s -a 0x2C965B68
Breakpoint 1: where = MobilePhoneSettings`-
[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 40, address =
0x2c965b68
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x2c965b68
MobilePhoneSettings`-[PhoneSettings Controller
tableView:cellForRowAtIndexPath:] + 40, queue =
'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x2c965b68 MobilePhoneSettings`-
```

```
[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 40
MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 40:
-> 0x2c965b68:  blx    0x2c975fb8                ; symbol
stub for: CTSettingRequest$shim
   0x2c965b6c:  mov    r4, r0
   0x2c965b6e:  movw   r0, #54708
   0x2c965b72:  movt   r0, #2697
(lldb) p (char *)$r1
(char *) $0 = 0x2c3daf33
"tableView:cellForRowAtIndexPath:"
(lldb) po $r0
[no Objective-C description available]
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x2c965b6c
MobilePhoneSettings`-[PhoneSettings Controller
tableView:cellForRowAtIndexPath:] + 44, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2c965b6c MobilePhoneSettings`-
[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 44
MobilePhoneSettings`-[PhoneSettingsController
tableView:cellForRowAtIndexPath:] + 44:
-> 0x2c965b6c:  mov    r4, r0
   0x2c965b6e:  movw   r0, #54708
   0x2c965b72:  movt   r0, #2697
   0x2c965b76:  mov    r2, r5
(lldb) po $r0
<PSTableCell: 0x15fc6b00; baseClass = UITableViewCell;
frame = (0 0; 320 44); text = 'My Number'; tag = 2; layer
= <CALayer: 0x15fbbe40>>
(lldb) po [$r0 detailTextLabel]
<UITableViewLabel: 0x15fb5590; frame = (0 0; 0 0); text =
'+86PhoneNumber'; userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd87e0>>
```

值得一提的是，objc_msgSendSuper2的第一个参数并不是一个Objective-C对象，我不清楚这到底是LLDB的bug，还是情况确实如此，但这不影响本节的

分析，忽略这个细节就好。感兴趣的朋友可以继续研究，然后在http://bbs.iosre.com分享你的发现。

话说回来，LLDB的输出结果预示着objc_msgSendSuper2的返回结果就是初始化好的cell，里面已经含有了本机号码信息。跟上一节类似，到PhoneSettingsController的父类里看看tableView:cellForRowAtIndexPath:的实现。首先打开PhoneSettingsController.h，看看它的父类是谁，如下：

```
@interface PhoneSettingsController :
PhoneSettingsListController
<TPSetPINViewControllerDelegate>
……
@end
```

可以看到，PhoneSettingsController继承自PhoneSettingsListController，再打开Phone-SettingsListController.h，看看它有没有实现tableView:cellForRowAtIndexPath:方法，如下：

```
@interface PhoneSettingsListController : PSListController
{
}
- (id)bundle;
- (void)dealloc;
- (id)init;
- (void)pushController:(Class)arg1 specifier:(id)arg2;
- (id)setCellEnabled:(BOOL)arg1 atIndex:(unsigned
int)arg2;
- (id)setCellLoading:(BOOL)arg1 atIndex:(unsigned
int)arg2;
- (id)setControlEnabled:(BOOL)arg1 atIndex:(unsigned int)
arg2;
- (id)sheetSpecifierWithTitle:(id)arg1 controller:
(Class)arg2
detail:(Class)arg3;
- (void)simRemoved:(id)arg1;
- (id)specifiers;
- (void)updateCellStates;
- (void)viewWillAppear:(BOOL)arg1;
@end
```

可见，PhoneSettingsListController没有实现tableView:cellForRowAtIndexPath:，继续去它的父类PSListController里看看。PSListController已经不在MobilePhoneSettings.bundle里了，用上一章介绍的搜索方法，很容易就可以在所有class-dump头文件里定位PSListController.h，如图6-31所示。

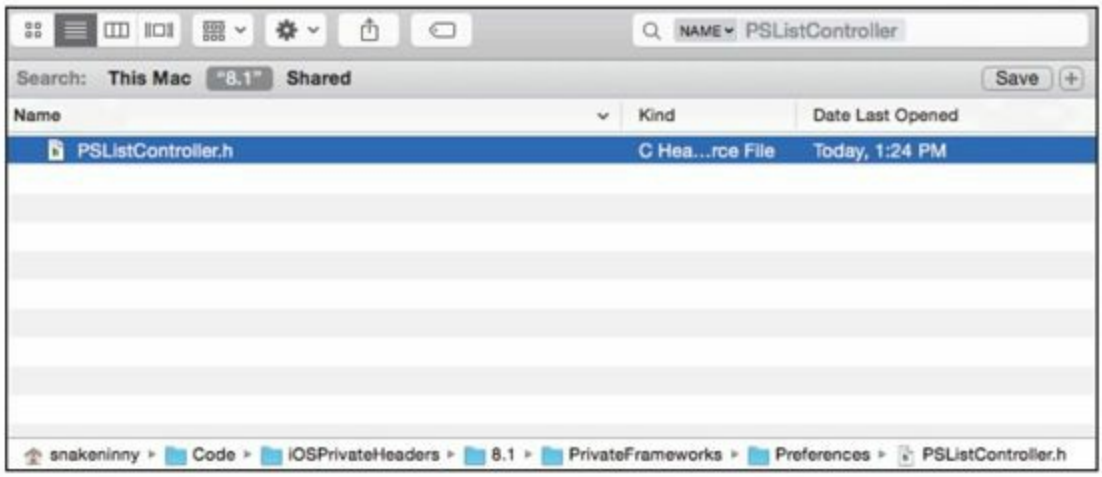


图6-31　定位PSListController.h

注意，PSListController.h来自与Preferences.app同名的Preferences.framework，请大家注意分辨。打开它，看看有没有实现tableView:cellForRowAtIndexPath:方法，如下：

```
@interface PSListController : PSViewController
<UITableViewDelegate, UITableView DataSource,
UIActionSheetDelegate, UIAlertViewDelegate,
UIPopoverControllerDelegate, PSSpecifierObserver,
PSViewControllerOffsetProtocol>
……
- (id)tableView:(id)arg1 cellForRowAtIndexPath:(id)arg2;
……
@end
```

可以看到，它确实实现了这个方法，在IDA中打开Preferences.framework里的二进制文件，定位到

tableView:cellForRowAtIndexPath:，如图6-32所示。

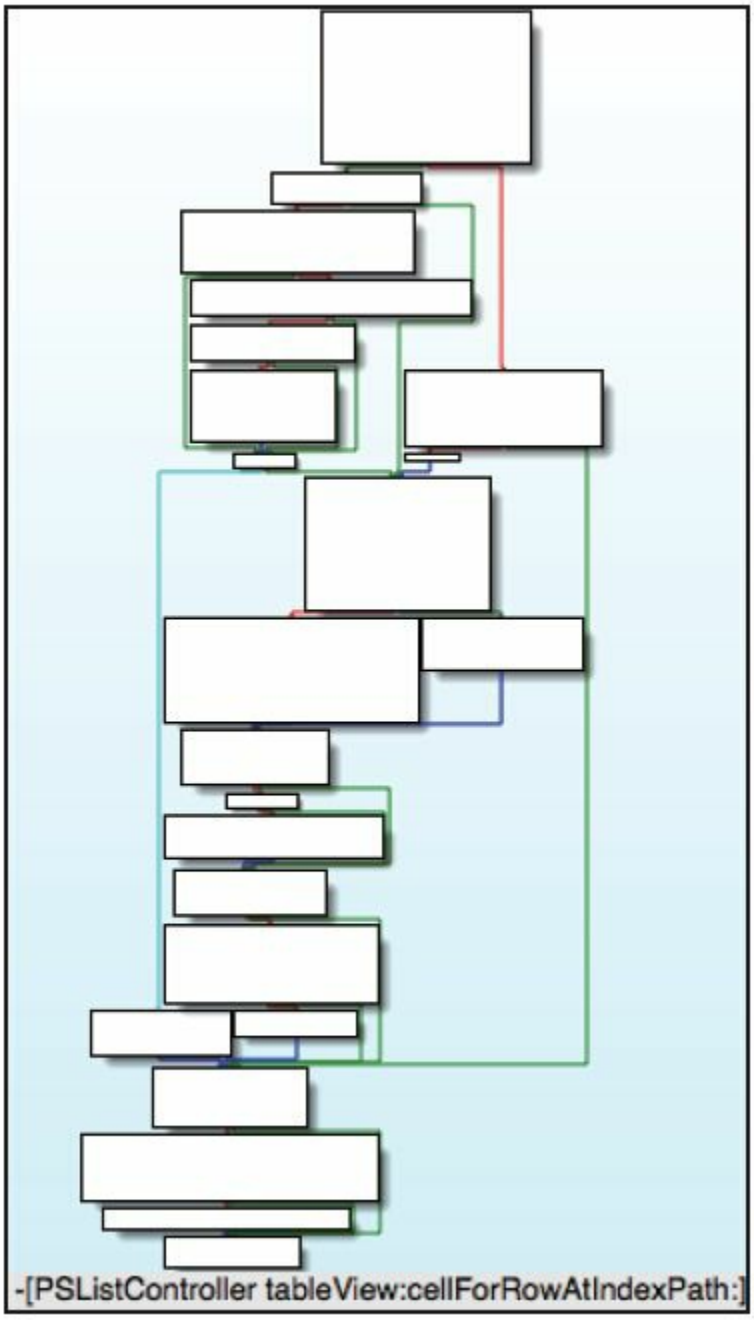


图6-32　[PSListController tableView:cellForRowAtIndexPath:]

它的实现逻辑有些复杂，为了保险起见，先在它的尾部下一个断点，看看返回值里是否含有“本机号码”信息，确认objc_msgSendSuper2是否调用了[PSListController tableView:cellForRowAtIndexPath:]。先看看Preferences.framework的ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x00079000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.a

[  1] 0x00232000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000

[  2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/Bulleti

[  3] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreFoundation

……
[ 42] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/Prefere

……
```

它的ASLR偏移是0x6db3000。然后看看[PSListController tableView:cellForRowAtIndexPath:]尾

部指令的地址，如图6-33所示。

图6-33 [PSListController tableView:cellForRowAtIndexPath:]

因为返回值存放在R0中，而R0来自“MOV R0,R6”，即R6，所以在这条指令上下一个断点，然后打印R6。这条指令的地址是0x2A9F79E6，因此断点的地址是0x6db3000+0x2A9F79E6=0x317AA9E6。返回上一页再重新进入MobilePhoneSettings，触发断点，如下：

```
(lldb) br s -a 0x317AA9E6
Breakpoint 5: where = Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 1026, address =
0x317aa9e6
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9e6 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
1026, queue = 'com.apple.main-thread, stop reason =
breakpoint 5.1
    frame #0: 0x317aa9e6 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 1026
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 1026:
-> 0x317aa9e6:  mov    r0, r6
```

```
    0x317aa9e8:  add    sp, #28
    0x317aa9ea:  pop.w  {r8, r10, r11}
    0x317aa9ee:  pop    {r4, r5, r6, r7, pc}
(lldb) po $r6
<PSTableCell: 0x15f8c6a0; baseClass = UITableViewCell;
frame = (0 0; 320 44); text = 'My Number'; tag = 2; layer
= <CALayer: 0x15f7c0b0>>
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7b8d0; frame = (0 0; 0 0); text =
'+86PhoneNumber'; userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15f7b990>>
```

从LLDB的输出可以确认objc_msgSendSuper2调用了[PSListController tableView:cellForRowAtIndexPath:]，且它的返回值来自于R6。那R6来自于哪里呢？当我们往上回溯，查找R6来源的时候，可以看到R6作为objc_msgSend的第一个参数，多次出现在了这个方法内部，如图6-34所示。

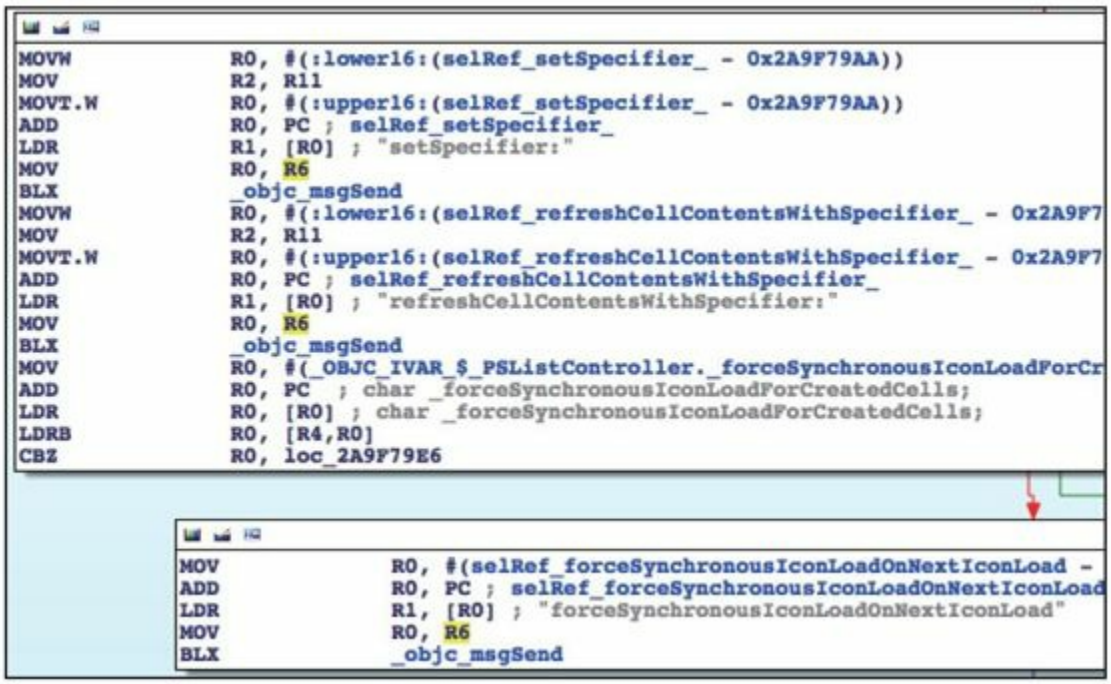


图6-34　R6出现频率很高

再往上一点，会发现往R6里写入的，都是刚刚初始化的各种对象，如图6-35、图6-36、图6-37所示。

```
STR.W R11, [SP,#0x34+var_34]
ADD R0, PC ; selRef_initWithStyle_reuseIdentifier_specifier_
LDR R1, [R0] ; "initWithStyle:reuseIdentifier:specifier"...
MOV R0, R6
BLX _objc_msgSend
MOV R1, #(selRef_autorelease - 0x2A9F784C) ; selRef_autorelease
ADD R1, PC ; selRef_autorelease
LDR R1, [R1] ; "autorelease"
BLX _objc_msgSend
MOV R6, R0
```

图6-35　R6被赋值（1）

这个现象很好理解，tableView:cellForRowAtIndexPath:的作用本来就是返回

一个可用的cell。因此，常规的做法是在方法内部先创建一个空的cell，然后调用别的函数来配置它。那么，从一个空的PSTableCell到含有“本机号码”信息的这个配置过程发生在哪里呢？现在已知头部的PSTableCell不含有本机号码，尾部的PSTableCell含有本机号码，所以这个设置过程一定是发生在tableView:cellForRowAtIndexPath:内部的，且是通过一个objc_msgSend函数完成的。因此，现在的问题变成了，在一堆objc_msgSend函数中，怎么去定位那个设置“本机号码”的objc_msgSend？

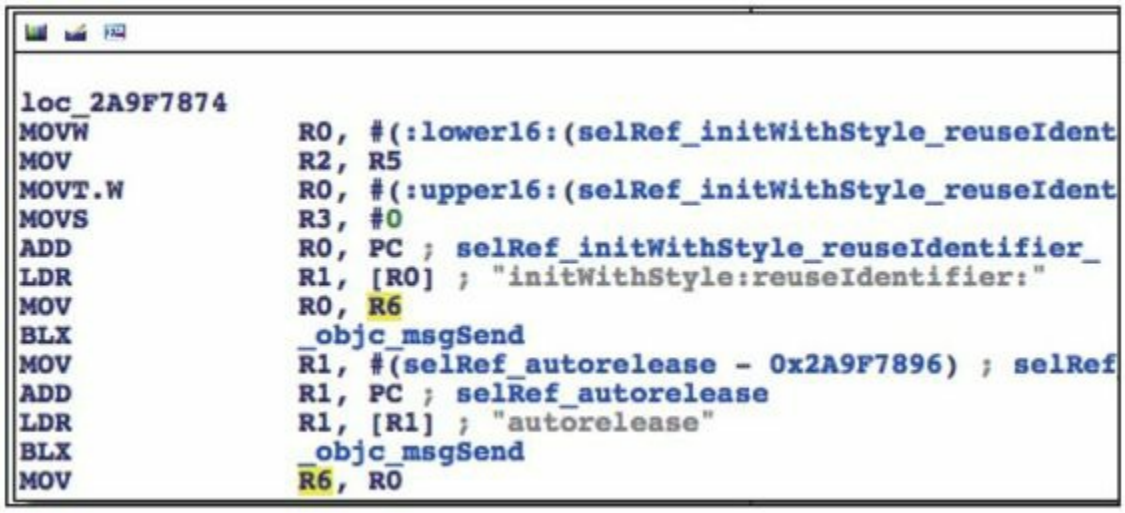


图6-36　R6被赋值（2）

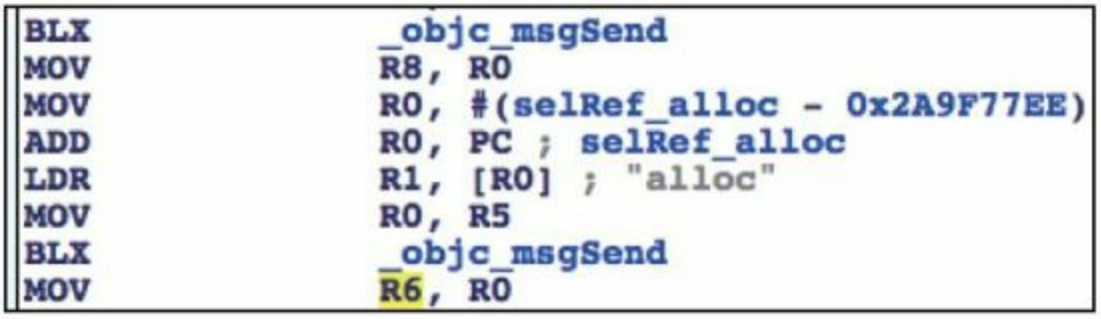


图6-37　R6被赋值（3）

如果不考虑效率，可以从头开始一个个排查。table-View:cellForRowAtIndexPath:内部的objc_msgSend个数毕竟有限，在执行objc_msgSend之前和之后各打印一次[$r6 detailText-Label]，对比两者的异同，就一定可以找到这个objc_msgSend；数学比较好的朋友可能用二分法，从tableView:cellForRow-AtIndexPath:中间部分的某个objc_msgSend开始找，不断缩小排查范围。这就是见仁见智的问题了，大家选择一种自己喜欢的方式就好。在这里，笔者采取了折中的二分法，如图6-38所示。

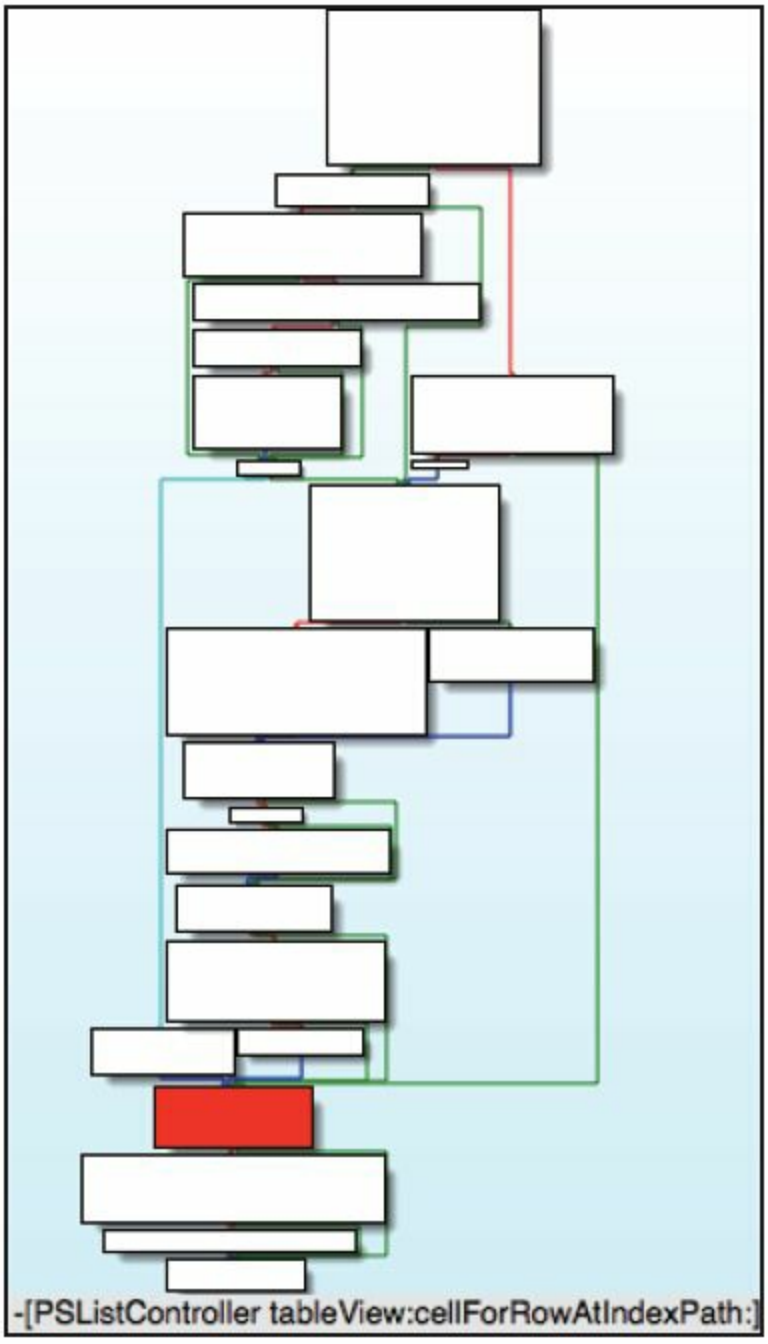


图6-38　[PSListControllertableView:

cellForRowAtIndexPath:]

采用二分查找法固然效率高，但[PSListController tableV-iew:cellForRowAtIndexPath:]的分支很多，从哪

个地方分，可以保证不遗漏每一个分支呢？因为[PSListController tableVi-ew:cellForRowAtIndexPath:]的执行一定会通过图6-38所示的深色方块，所以以这个地方为二分点肯定不会遗漏任何分支，然后从它的第一个objc_msgSend开始排查，如果[$r6 detailTextLabel]含有本机号码信息，那么就往上找，否则往下找。我们去看看这个深色方块包含的汇编指令，如图6-39所示。

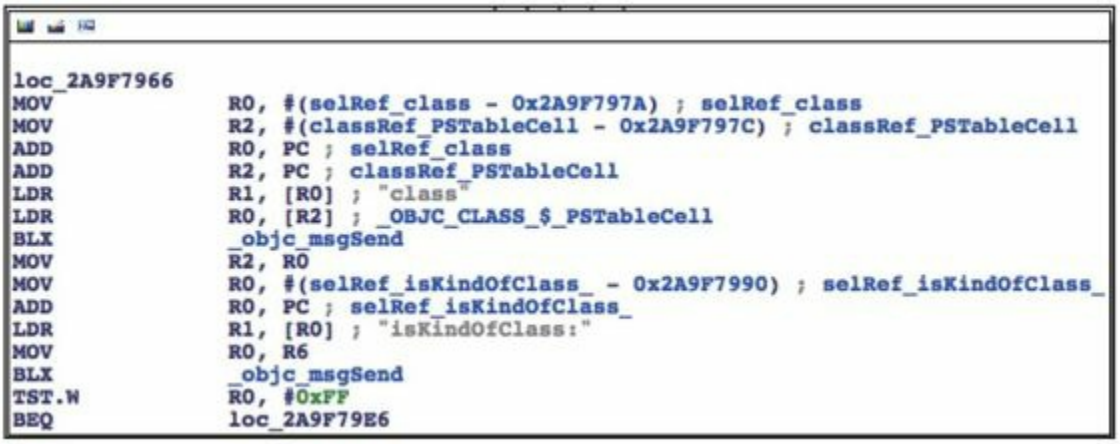


图6-39　深色方块所在的loc_2a9f7966

这里有2个objc_msgSend，就从最上面这一个开始吧，看看它的地址，如图6-40所示。

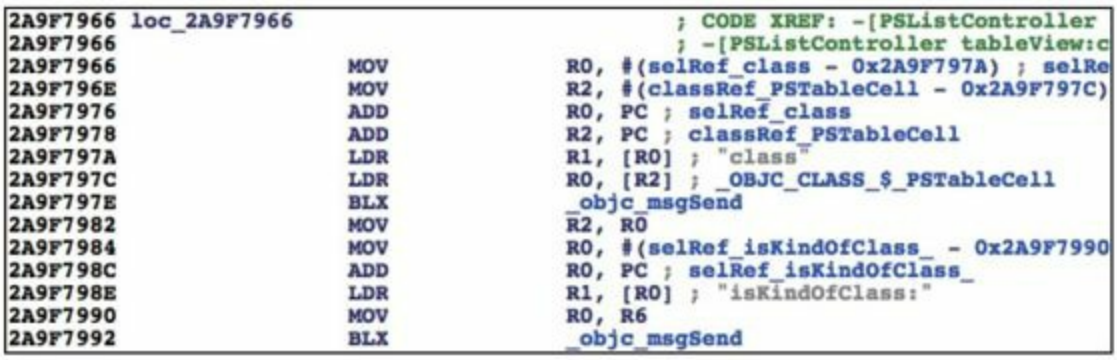


图6-40　查看objc_msgSend的地址

Preferences的ASLR偏移是0x6db3000，刚才已经用到了，所以断点的地址是0x6db3000+0x2A9F797E=0x317AA97E。触发它，看看此时PSTableCell是否含有本机号码信息，如下：

```
(lldb) br s -a 0x317AA97E
Breakpoint 10: where = Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 922, address =
0x317aa97e
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa97e Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
922, queue = 'com.apple.main-thread, stop reason =
breakpoint 10.1
    frame #0: 0x317aa97e Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 922
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 922:
-> 0x317aa97e:  blx    0x31825f04                ; symbol
stub for: ____NETRBClientResponseHandler_block_invoke
   0x317aa982:  mov    r2, r0
   0x317aa984:  movw   r0, #59804
   0x317aa988:  movt   r0, #1736
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
```

```
0x15fd1c90>>
```

它还不含有本机号码信息，说明本机号码信息一定是在图6-38深色方块下方的3个方块里生成的。接着执行“ni”命令，在每个objc_msgSend的前后各“po[$r6 detailTextLabel]”一次，如下：

```
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa982 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
926, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa982 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 926
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 926:
-> 0x317aa982:  mov    r2, r0
   0x317aa984:  movw   r0, #59804
   0x317aa988:  movt   r0, #1736
   0x317aa98c:  add    r0, pc
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x15fd1c90>>
(lldb) ni
……
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa992 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
942, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa992 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 942
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 942:
-> 0x317aa992:  blx    0x31825f04                ; symbol
stub for: ____NETRBClientResponseHandler_block_invoke
   0x317aa996:  tst.w  r0, #255
```

```
   0x317aa99a:  beq    0x317aa9e6                ; -
[PSListController tableView:cellForRowAtIndexPath:] +
1026
   0x317aa99c:  movw   r0, #60302
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x15fd1c90>>
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa996 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
946, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa996 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 946
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 946:
-> 0x317aa996:  tst.w  r0, #255
   0x317aa99a:  beq    0x317aa9e6                ; -
[PSListController tableView:cellForRowAtIndexPath:] +
1026
   0x317aa99c:  movw   r0, #60302
   0x317aa9a0:  mov    r2, r11
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x15fd1c90>>
(lldb) ni
……
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9ac Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
968, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9ac Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 968
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 968:
-> 0x317aa9ac:  blx    0x31825f04                ; symbol
stub for: ____NETRBClientResponseHandler_block_invoke
   0x317aa9b0:  movw   r0, #60822
   0x317aa9b4:  mov    r2, r11
   0x317aa9b6:  movt   r0, #1736
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x15fd1c90>>
(lldb) ni
```

```
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9b0 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
972, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9b0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 972
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 972:
-> 0x317aa9b0:  movw   r0, #60822
   0x317aa9b4:  mov    r2, r11
   0x317aa9b6:  movt   r0, #1736
   0x317aa9ba:  add    r0, pc
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x15fd1c90>>
(lldb) ni
……
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9c0 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
988, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9c0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988:
-> 0x317aa9c0:  blx    0x31825f04                ; symbol
stub for: ____NETRBClientResponseHandler_block_invoke
   0x317aa9c4:  movw   r0, #4312
   0x317aa9c8:  movt   r0, #1737
   0x317aa9cc:  add    r0, pc
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0);
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x15fd1c90>>
(lldb) ni
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9c4 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
992, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x317aa9c4 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 992
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 992:
-> 0x317aa9c4:  movw   r0, #4312
```

```
    0x317aa9c8:  movt   r0, #1737
    0x317aa9cc:  add    r0, pc
    0x317aa9ce:  ldr    r0, [r0]
(lldb) po [$r6 detailTextLabel]
<UITableViewLabel: 0x15f7e490; frame = (0 0; 0 0); text =
'+86PhoneNumber'; userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x15fd1c90>>
```

在0x317aa9c0处的objc_msgSend前后PSTableCell的本机号码信息发生了变化，0x317aa9c0-0x6db3000=0x2A9F79C0，在IDA中定位到这个objc_msgSend，如图6-41所示。

```
__text:2A9F79BC    LDR    R1, [R0] ; "refreshCellContentsWithSpecifier:"
__text:2A9F79BE    MOV    R0, R6
__text:2A9F79C0    BLX    _objc_msgSend
```

图6-41　设置本机号码的objc_msgSend

“用specifier刷新cell的内容”，这个方法的作用显而易见，我们看看这个specifier是什么。在这个objc_msgSend上下个断点，触发后，打印它的参数，如下：

```
(lldb) br s -a 0x317AA9C0
Breakpoint 11: where = Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988, address =
0x317aa9c0
Process 268587 stopped
* thread #1: tid = 0x4192b, 0x317aa9c0 Preferences`-
[PSListController tableView:cellForRowAtIndexPath:] +
```

```
988, queue = 'com.apple.main-thread, stop reason =
breakpoint 11.1
    frame #0: 0x317aa9c0 Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988
Preferences`-[PSListController
tableView:cellForRowAtIndexPath:] + 988:
-> 0x317aa9c0:  blx    0x31825f04                ; symbol
stub for: ____NETRBClientResponseHandler_block_invoke
   0x317aa9c4:  movw   r0, #4312
   0x317aa9c8:  movt   r0, #1737
   0x317aa9cc:  add    r0, pc
(lldb) p (char *)$r1
(char *) $97 = 0x318362d2
"refreshCellContentsWithSpecifier:"
(lldb) po $r2
My Numbe        ID:myNumberCell 0x170ece60      target:
<PhoneSettingsController 0x170ed760: navItem
<UINavigationItem: 0x170d0b40>, view <UITableView:
0x16acb200; frame = (0 0; 320 568); autoresize = W+H;
gestureRecognizers = <NSArray: 0x15d232d0>; layer =
<CALayer: 0x15fc9110>; contentOffset: {0, -64};
contentSize: {320, 717.5}>>
(lldb) po [$r2 class]
PSSpecifier
```

可以看到，specifier是一个PSSpecifier对象，而且与本机号码相关。如果你在第5章的PreferenceBundle部分仔细阅读过preferences specifier plist标准，就知道PSTableCell的内容是由PSSpecifier指定的，因此可以通过[PSSpecifier propertyForKey:@”set”]和[PSSpecifier propertyForKey:@”get”]拿到PSSpecifier的setter和getter，如下：

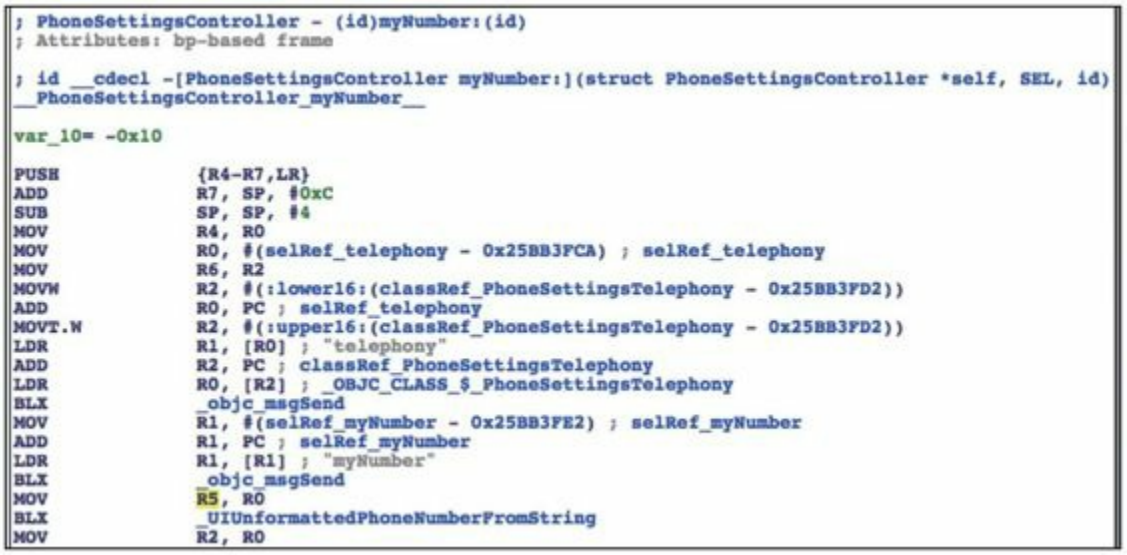
```
(lldb) po [$r2 propertyForKey:@"set"]
```

```
setMyNumber:specifier:
(lldb) po [$r2 propertyForKey:@"get"]
myNumber:
```

还可以通过[PSSpecifier target]拿到它们的target，如下：

```
(lldb) po [$r2 target]
<PhoneSettingsController 0x170ed760: navItem
<UINavigationItem: 0x170d0b40>, view <UITableView:
0x16acb200; frame = (0 0; 320 568); autoresize = W+H;
gestureRecognizers = <NSArray: 0x15d232d0>; layer =
<CALayer: 0x15fc9110>; contentOffset: {0, -64};
contentSize: {320, 717.5}>>
```

非常好，现在我们知道PSTableCell的本机号码是通过[PhoneSettingsController setMy Number:specifier:]方法设置的，通过[PhoneSettingsController myNumber:]读取的（你对它俩还有印象吗？），那么，在myNumber:内部，就一定有获取本机号码的方法，如图6-42所示。



```
; PhoneSettingsController - (id)myNumber:(id)
; Attributes: bp-based frame

; id __cdecl -[PhoneSettingsController myNumber:](struct PhoneSettingsController *self, SEL, id)
__PhoneSettingsController_myNumber__

var_10= -0x10

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
SUB             SP, SP, #4
MOV             R4, R0
MOV             R0, #(selRef_telephony - 0x25BB3FCA) ; selRef_telephony
MOV             R6, R2
MOVW            R2, #(:lower16:(classRef_PhoneSettingsTelephony - 0x25BB3FD2))
ADD             R0, PC ; selRef_telephony
MOVT.W          R2, #(:upper16:(classRef_PhoneSettingsTelephony - 0x25BB3FD2))
LDR             R1, [R0] ; "telephony"
ADD             R2, PC ; classRef_PhoneSettingsTelephony
LDR             R0, [R2] ; _OBJC_CLASS_$_PhoneSettingsTelephony
BLX             _objc_msgSend
MOV             R1, #(selRef_myNumber - 0x25BB3FE2) ; selRef_myNumber
ADD             R1, PC ; selRef_myNumber
LDR             R1, [R1] ; "myNumber"
BLX             _objc_msgSend
MOV             R5, R0
BLX             _UIUnformattedPhoneNumberFromString
MOV             R2, R0
```

图6-42　[PhoneSettingsController　myNumber:]

[PhoneSettingsController myNumber:]的逻辑比较简单，就是看[[PhoneSettingsTelephony telephony]myNumber]的长度是否为0，如果不为0，它就是本机号码，否则返回一个“未知号码”，告诉用户无法读取本机号码。用Cycript测试一下这个方法，如下：

```
FunMaker-5:~ root# cycript -p Preferences
cy# [[PhoneSettingsTelephony telephony] myNumber]
@"+86PhoneNumber"
```

现在，退出Preferences，把它从后台彻底关掉后重新打开，不要进入MobilePhoneSettings界面，再测

试一次这个方法，如下：

```
FunMaker-5:~ root# cycript -p Preferences
cy# [[PhoneSettingsTelephony telephony] myNumber]
ReferenceError: Can't find variable:
PhoneSettingsTelephony
```

出现了错误，这是怎么回事？那是因为PhoneSettingsTelephony是MobilePhoneSettings.bundle中的一个类，如果不进入MobilePhoneSettings界面，这个bundle是不会加载的，所以这个类也是不存在的。也就是说，要调用这个方法，需要先加载MobilePhoneSettings.bundle。Preference.app加载MobilePhoneSettings.bundle的方式被称为延迟加载（lazy load），在iOS逆向工程中出现类似状况的时候很多，当你碰到时，欢迎来http://bbs.iosre.com跟大家交流心得。

其实到此为止，可以认为我们已经找到了目标函数，因为我们拿到了这个方法的调用者和参数，而且这个方法不涉及UI操作，调用起来干净利落。但有一

点让人不爽的是，调用这个方法前必须加载MobilePhoneSettings.bundle。有没有办法去掉这个硬指标，让我们不需要加载这个bundle就能拿到本机号码呢？应该存在这么一个方法。因为本机号码是存储在SIM卡上的，所以[PhoneSettingsTelephony myNumber]的原始数据源应该来自SIM卡，而能够访问SIM卡的显然不止MobilePhoneSetting.bundle，因此底层一定存在更通用的访问SIM卡的库，如果能定位到这个库，估计就可以直接读取本机号码了。既然是一个更底层的库，那么自然要从[PhoneSettingsTelephony myNumber]入手，看看它的内部是如何读取本机号码的，如图6-43所示。

它的逻辑也比较简单，先取出实例变量_myNumber，如果它不是nil，则走左边并记录“My Number requested,returning cached value：%@”，即返回一个缓存中的数据；否则走右边，先调用PhoneSettingsCopyMyNumber函数取得本机号码，再

记录“My Number requested,no cached value,fetched：%@”，即没有在缓存中找到本机号码，返回一个现取的数据。因此，调用PhoneSettingsCopyMyNumber可以取得本机号码，但从名字来看，它仍然是MobilePhoneSettings.bundle里的一个函数，在这个bundle外不能调用，看来我们挖得还不够深。继续看看这个函数内部做了些什么，如图6-44所示。

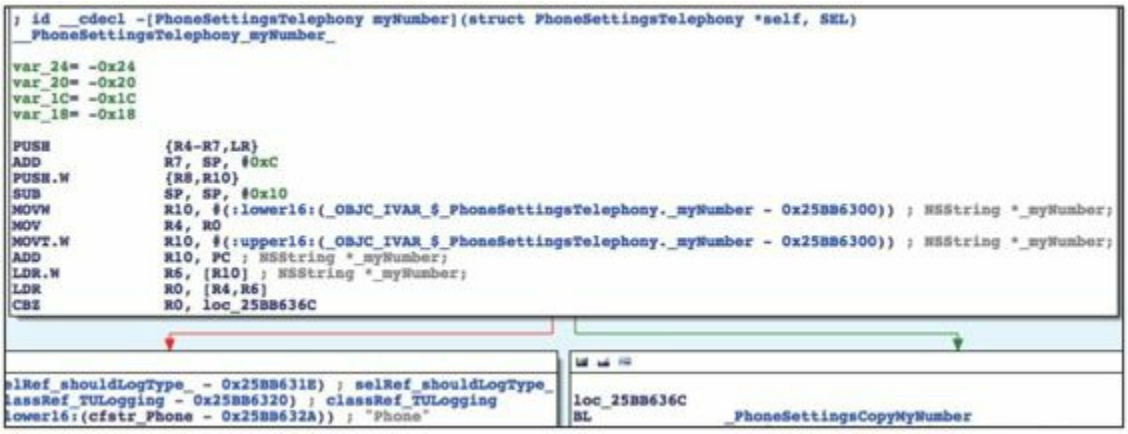


图6-43　[PhoneSettingsTelephony myNumber]

图6-44　PhoneSettingsCopyMyNumber

这段代码先调用CTSettingCopyMyPhoneNumber函数，把返回值给autorelease掉，然后再调用PhoneSettingsCopyFormattedNumberBySIMCountry，看其函数名好像是根据SIM卡所在的国家把号码给格式化了。那么CTSettingCopyMyPhoneNumber函数无论是从名字还是上下文来看，都非常疑似获取本机号码的函数，而且CT前缀说明它来自CoreTelephony，而不是MobilePhoneSettings。双击这个函数，看看它的内部实现，如图6-45所示。

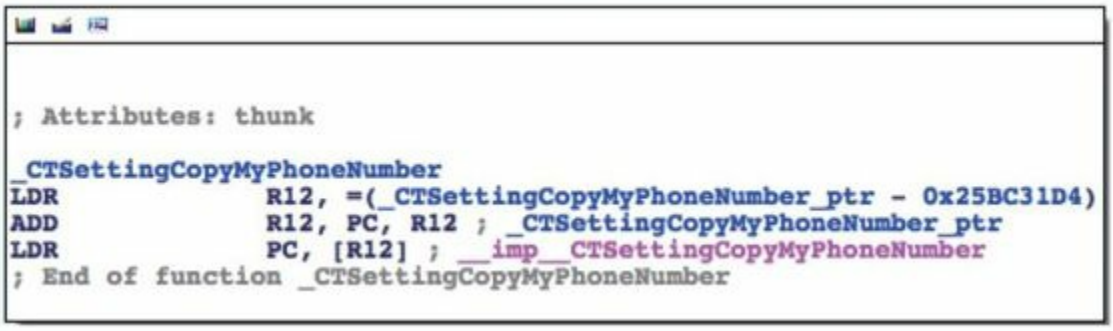


图6-45　CTSettingCopyMyPhoneNumber

果然是一个外部函数，再次双

击“__imp__CTSettingCopyMyPhoneNumber”，看看它来自哪个库——正是CoreTelephony。退出Preferences，把它从后台彻底关掉后重新打开，不要进入MobilePhoneSettings界面，然后用debugserver附加，用LLDB打印出image list，你会发现CoreTelephony赫然名列其中。这意味着，我们不需要加载MobilePhoneSettings.bundle就可以调用CTSettingCopyMyPhoneNumber获取未经格式化的本机号码，它就是我们要找的目标函数。那么还剩最后一个问题——它的参数和返回值是什么？

从图6-44看来，CTSettingCopyMyPhoneNumber不像是有参数——它的前面甚至没有出现R0~R3寄存器。如果它有参数，那么R0~R3也是来自它的调用者，即PhoneSettingsCopyMyNumber。但从图6-43看来，PhoneSettingsCopyMyNumber之前也只出现了R0，且如果进程走右边，R0一定是0，PhoneSettingsCopyMyNumber看起来也没有参数。为

了保险起见，还是去CoreTelephony里看看CTSettingCopyMyPhoneNumber的实现，如图6-46所示。

根据Objective-C函数的命名惯例，CTTelephonyCenterGetDefault是有返回值的；在“BL_CTTelephonyCenterGetDefault”下面，R0被CTTelephonyCenterGetDefault的返回值覆盖掉了；而在图6-46的最下面，R1也被“MOV R1,R4”中的R4覆盖掉了。如果R0和R1是参数，那么这2个参数就没有起任何作用，不合常理，因此说明CTSettingCopyMyPhoneNumber没有参数。那么它的返回值呢？我们会很自然地猜测它的返回值是一个字符串，但为了保险起见，还是在CTSettingCopyMyPhoneNumber的尾部下个断点，把R0打印出来看看吧。先在IDA中看看它的地址，如图6-47所示。

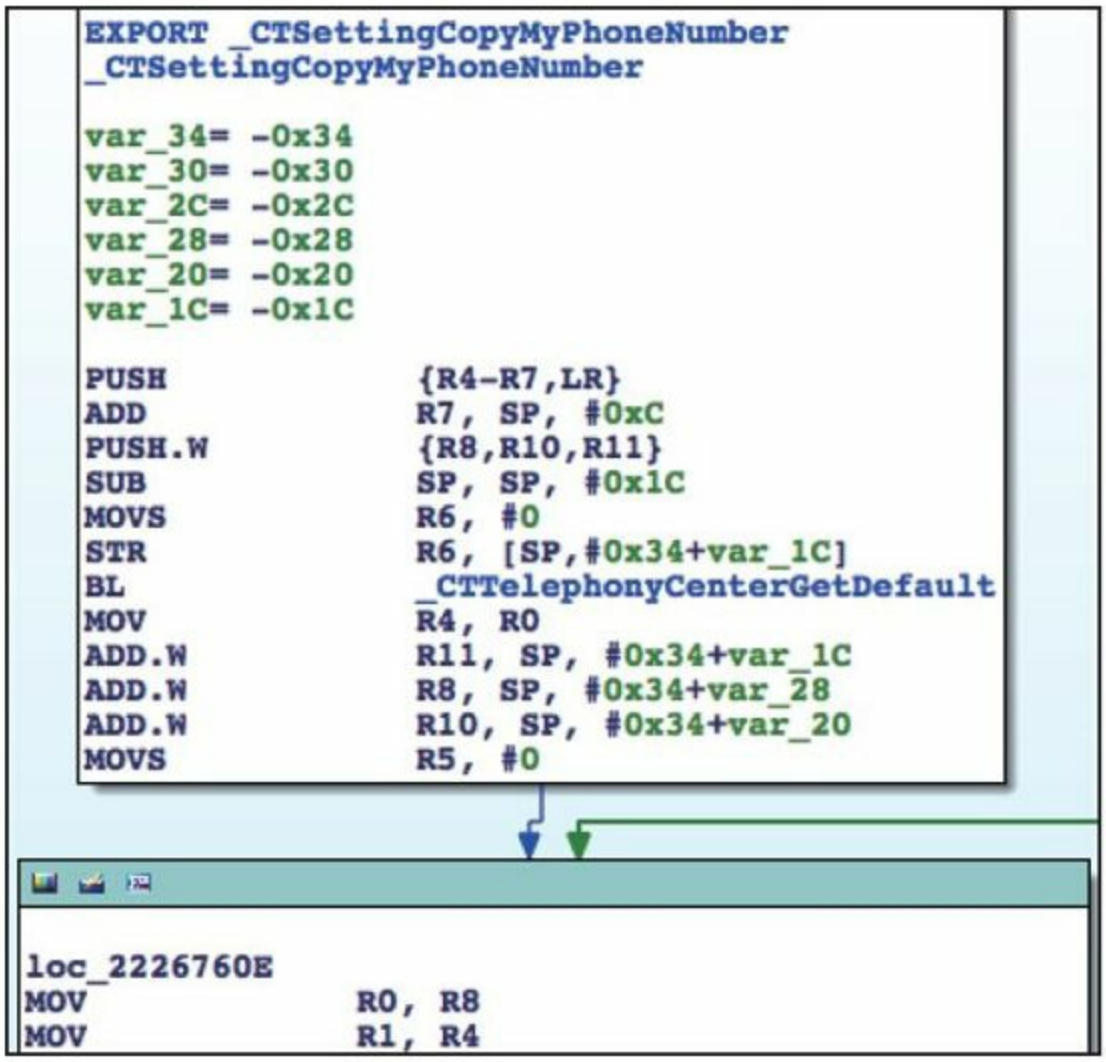


图6-46　CTSettingCopyMyPhoneNumber

然后退出Preferences，把它从后台彻底关掉后重新打开，不要进入MobilePhoneSettings界面，然后用debugserver附加，用LLDB查看CoreTelephony的ASLR偏移，如下：

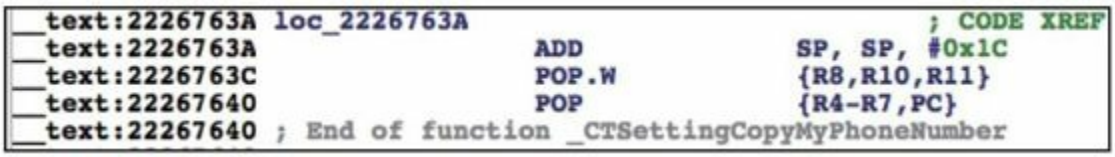


图6-47　CTSettingCopyMyPhoneNumber

```
(lldb) image list -o -f
[  0] 0x000b3000
/private/var/db/stash/_.29LMeZ/Applications/Preferences.a

[  1] 0x0026c000
/Library/MobileSubstrate/MobileSubstrate.dylib
(0x000000000026c000)
[  2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/Bulleti
 [ 51] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/CoreTelephony.

……
```

我们就把断点下在0x6db3000+0x2226763A=0x2901A63A上吧。然后进入MobilePhone-Settings界面，触发断点，如下：

```
(lldb) br s -a 0x2901A63A
Breakpoint 1: where =
CoreTelephony`CTSettingCopyMyPhoneNumber + 78, address =
0x2901a63a
Process 330210 stopped
* thread #1: tid = 0x509e2, 0x2901a63a
CoreTelephony`CTSettingCopyMyPhoneNumber + 78, queue =
'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x2901a63a
```

```
CoreTelephony`CTSettingCopyMyPhoneNumber + 78
CoreTelephony`CTSettingCopyMyPhoneNumber + 78:
-> 0x2901a63a:  add    sp, #28
   0x2901a63c:  pop.w  {r8, r10, r11}
   0x2901a640:  pop    {r4, r5, r6, r7, pc}
   0x2901a642:  nop
(lldb) po $r0
+86PhoneNumber
(lldb) po [$r0 class]
__NSCFString
```

它就是一个NSString，这样就可以还原这个函数的原型啦——

```
NSString *CTSettingCopyMyPhoneNumber(void);
```

它就是我们的目标函数，也就是PSTableCell的数据源，我们通过分析[PhoneSettings Controller tableView:cellForRowAtIndexPath:]所在的函数调用链找到了它。在调用它的时候，注意释放返回值就好了。写一个小tweak测测这个函数，确保它是正确的。

（1）用Theos新建tweak工程“iOSREGetMyNumber”，命令如下：

```
snakeninnys-MacBook:Code snakeninny$
/opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): iOSREGetMyNumber
Package Name [com.yourcompany.iosregetmynumber]:
com.iosre.iosregetmynumber
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.apple.Preferences
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]: Preferences
Instantiating iphone/tweak in iosregetmynumber/...
Done.
```

（2）编辑Tweak.xm，代码如下：

```
extern "C" NSString *CTSettingCopyMyPhoneNumber(void); //
来自CoreTelephony
%hook PreferencesAppController
- (BOOL)application:(id)arg1
didFinishLaunchingWithOptions:
(id)arg2
{
        BOOL result = %orig;
        NSLog(@"iOSRE: my number = %@",
[CTSettingCopyMyPhoneNumber() autorelease]);
        return result;
}
%end
```

（3）编辑Makefile及control

编辑后的Makefile内容如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
TWEAK_NAME = iOSREGetMyNumber
iOSREGetMyNumber_FILES = Tweak.xm
iOSREGetMyNumber_FRAMEWORKS = CoreTelephony #
CTSettingCopyMyPhoneNumber来自这里
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
      install.exec "killall -9 Preferences"
```

编辑后的control内容如下：

```
Package: com.iosre.iosregetmynumber
Name: iOSREGetMyNumber
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Get my number just like MobilePhoneSettings!
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

（4）测试

将写好的tweak编译打包安装到iOS后，打开

Preferences，不要进入MobilePhoneSettings界面。然后ssh到iOS上看看syslog，如下：

```
FunMaker-5:~ root# grep iOSRE: /var/log/syslog
Nov 29 23:23:01 FunMaker-5 Preferences[2078]: iOSRE: my
number = +86PhoneNumber
```

（5）补充

因为笔者的iPhone 5将地区设置为了美国，所以格式化之前的本机号码是“+86PhoneNumber”，被PhoneSettingsCopyFormattedNumberBySIMCountry格式化之后变成了“+86 Pho-neNu-mber”，即美国电话号码格式。

在逆向其他目标碰到CTSettingCopyMyPhoneNumber时，随着iOS逆向工程熟练度的增加，你就会慢慢发现，它的正确原型其实是：

```
CFStringRef CTSettingCopyMyPhoneNumber();
```

因为NSString*和CFStringRef是等价的，所以我们的写法也没问题。

因为CTSettingCopyMyPhoneNumber的函数名中含有“copy”字样，且它返回了一个CoreData对象，所以根据苹果的“Ownership Policy”（Google搜索“apple ownership policy”），我们要负责释放这个函数的返回值。

本节用大量篇幅，用ARM汇编完善了“定位目标函数”环节，并将其细分为“从现象切入App，找出UI函数”和“以UI函数为起点，寻找目标函数”两步，结合Cycript、IDA和LLDB，既定位了目标函数，又解析了一些不够直观的函数参数。两个例子中演示的套路基本可以应付现在95%的App，如果你有幸碰到了那5%搞不定的，欢迎来http://bbs.iosre.com提供案例，我们一起来寻求解决方案。

6.3 LLDB的使用技巧

上一节是不是为你开启了iOS逆向工程的另一扇门？IDA和LLDB的配合简直是无坚不摧，再配合ARM指令集文档，似乎已经达成了“它俩在手，天下我有”的境界。你是不是已经迫不及待，想要废寝忘食地实践刚学到的新知识了呢？

先别急。6.2节的2个例子虽然已经综合运用了IDA和LLDB，但仍没有涵盖LLDB的常用场景。因此下面以几个短例示范一下LLDB的使用技巧，它们在实战中的合理运用能够大大减少我们的工作量。

6.3.1 寻找函数调用者

在上一节的2个例子里，在还原函数调用链时，主要分析的是一个函数调用了哪些函数，也就是还原了函数调用链的下游。当需要追溯函数调用链上游的时候，那就需要分析一个函数的调用者是谁了。看下

面这样一段代码：

```
// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --
show-
sdk-path` -framework Foundation -o MainBinary main.m
#include <stdio.h>
#include <dlfcn.h>
#import <Foundation/Foundation.h>
extern void TestFunction0(void)
{
        NSLog(@"iOSRE: %u", arc4random_uniform(0));
}
extern void TestFunction1(void)
{
        NSLog(@"iOSRE: %u", arc4random_uniform(1));
}
extern void TestFunction2(void)
{
        NSLog(@"iOSRE: %u", arc4random_uniform(2));
}
extern void TestFunction3(void)
{
        NSLog(@"iOSRE: %u", arc4random_uniform(3));
}
int main(int argc, char **argv)
{
        TestFunction3();
        return 0;
}
```

把这段代码存成名为main.m的文件，用注释里的那句话编译它，然后把MainBinary拖进IDA，并查看NSLog的交叉引用，如图6-48所示。

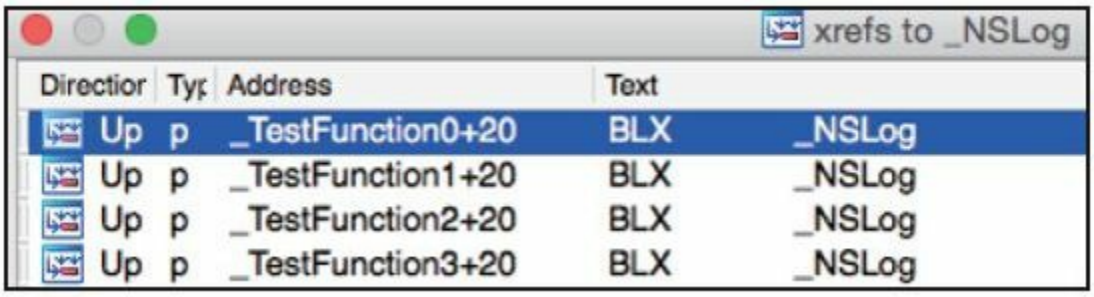


图6-48　查看NSLog的交叉引用

可以看到，在这段代码中，NSLog出现在了4个函数里，如果在逆向时发现syslog中出现了“iOSRE：0”，那么这个输出到底是来自哪个NSLog呢？当代码的逻辑比较简单时，靠人工就可以指出只有TestFunction3得到了调用，它进而又调用了NSLog。可如果这里有20个TestFunction，分别被8个不同的函数调用呢？逻辑变得复杂，人工分析就很吃力了。在这种情况下要寻找NSLog的调用者，LLDB就能起到很大的作用；用LLDB寻找函数调用者，主要有2种方法。

1.查看LR

还记得6.1.3节介绍的LR寄存器吗？它的作用是

保存返回地址。什么是返回地址？举例如下：

```
void FunctionA()
{
……
FunctionB();
……
}
```

在上面的伪代码中，FunctionA调用FunctionB，而A和B一般位于内存中的2块不同区域，它们的地址没有直接关联。B执行结束后，需要回到A里继续执行接下来的指令，如图6-49所示。

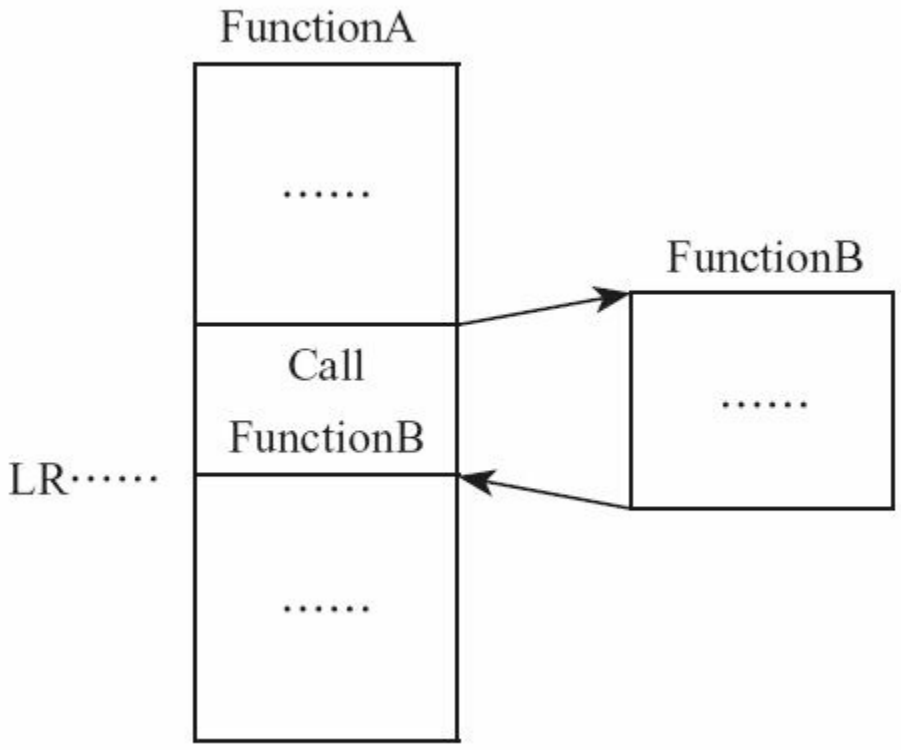


图6-49　返回地址示意图

B执行结束后返回的那个地方，就是返回地址。因为它位于调用者的内部，所以如果能知道LR的值，就可以知道调用者是谁；概念不好懂，操作一遍你就全明白了。先把Foundation.framework的二进制文件拖进IDA，初始分析结束后定位到NSLog，查看其基地址，如图6-50所示。

它的基地址是0x2261ab94，等会我们要在它上下断点，打印LR的值。接着用debugserver启动MainBinary，如下：

```
__text:2261AB94                 EXPORT _NSLog
__text:2261AB94 _NSLog                                  ; CODE XREF: -
__text:2261AB94                                         ; -[NSLock loc
__text:2261AB94
__text:2261AB94 var_18          = -0x18
__text:2261AB94
__text:2261AB94                 SUB             SP, SP, #0xC
__text:2261AB96                 PUSH            {R7,LR}
__text:2261AB98                 MOV             R7, SP
__text:2261AB9A                 SUB             SP, SP, #4
__text:2261AB9C                 ADD.W           R9, R7, #8
__text:2261ABA0                 STMIA.W         R9, {R1-R3}
__text:2261ABA4                 ADD.W           R1, R7, #8
__text:2261ABA8                 STR             R1, [SP,#0x18+var_18]
__text:2261ABAA                 BL              _NSLogv
__text:2261ABAE                 ADD             SP, SP, #4
__text:2261ABB0                 POP.W           {R7,LR}
__text:2261ABB4                 ADD             SP, SP, #0xC
__text:2261ABB6                 BX              LR
__text:2261ABB6 ; End of function _NSLog
```

图6-50　查看NSLog基地址

```
FunMaker-5:~ root# debugserver -x backboard *:1234
/var/tmp/MainBinary
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-
320.2.89
 for armv7.
Listening to port 1234 for a connection from *...
```

再用LLDB连过去，如下：

```
(lldb) process connect connect://localhost:1234
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7000 dyld`_dyld_start,
stop reason = signal SIGSTOP
    frame #0: 0x1fec7000 dyld`_dyld_start
dyld`_dyld_start:
-> 0x1fec7000:  mov    r8, sp
   0x1fec7004:  sub    sp, sp, #16
   0x1fec7008:  bic    sp, sp, #7
   0x1fec700c:  ldr    r3, [pc, #112]          ;
_dyld_start + 132
(lldb) image list -f
[  0]
/Users/snakeninny/Library/Developer/Xcode/iOSDeviceSuppor
 (12B411)/Symbols/usr/lib/dyld
```

此时MainBinary还未启动，我们位于dyld内部。接下来，一直执行“ni”命令，直到出现“error：invalid thread”的提示，如下：

```
(lldb) ni
Process 450336 stopped
```

```
* thread #1: tid = 0x6df20, 0x1fec7004 dyld`_dyld_start +
4, stop reason = instruction step over
    frame #0: 0x1fec7004 dyld`_dyld_start + 4
dyld`_dyld_start + 4:
-> 0x1fec7004:  sub    sp, sp, #16
   0x1fec7008:  bic    sp, sp, #7
   0x1fec700c:  ldr    r3, [pc, #112]            ;
_dyld_start + 132
   0x1fec7010:  sub    r0, pc, #8
(lldb)
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7008 dyld`_dyld_start +
8, stop reason = instruction step over
    frame #0: 0x1fec7008 dyld`_dyld_start + 8
dyld`_dyld_start + 8:
-> 0x1fec7008:  bic    sp, sp, #7
   0x1fec700c:  ldr    r3, [pc, #112]            ;
_dyld_start + 132
   0x1fec7010:  sub    r0, pc, #8
   0x1fec7014:  ldr    r3, [r0, r3]
……
(lldb)
error: invalid thread
```

到这里，不要再执行“ni”命令了，此时dyld开始加载MainBinary，等待一会，进程又会停下来，这时我们已经在MainBinary内部，可以开始调试了，如下：

```
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x1fec7040 dyld`_dyld_start +
64, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x1fec7040 dyld`_dyld_start + 64
dyld`_dyld_start + 64:
-> 0x1fec7040:  ldr    r5, [sp, #12]
   0x1fec7044:  cmp    r5, #0
   0x1fec7048:  bne    0x1fec7054                ;
_dyld_start + 84
```

```
    0x1fec704c:  add    sp, r8, #4
```

下面看看Foundation.framework的ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x000fc000
/private/var/tmp/MainBinary(0x0000000000100000)
[  1] 0x000c6000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1 (12B411)/Symbols/usr/lib/dyld
[  2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/Foundation.fra

……
```

断点下在0x6db3000+0x2261ab94=0x293CDB94。然后执行“c”命令，触发断点，如下：

```
(lldb) br s -a 0x293CDB94
Breakpoint 1: where = Foundation`NSLog, address =
0x293cdb94
(lldb) c
Process 450336 resuming
Process 450336 stopped
* thread #1: tid = 0x6df20, 0x293cdb94 Foundation`NSLog,
queue = 'com.apple.main-thread, stop reason = breakpoint
1.1
    frame #0: 0x293cdb94 Foundation`NSLog
Foundation`NSLog:
-> 0x293cdb94:  sub    sp, #12
   0x293cdb96:  push   {r7, lr}
   0x293cdb98:  mov    r7, sp
   0x293cdb9a:  sub    sp, #4
```

最后打印LR的值，如下：

```
(lldb) p/x $lr
(unsigned int) $0 = 0x00107f8d
```

因为MainBinary的基地址是0x000fc000，所以在IDA里打开MainBinary，然后跳转到0x107f8d-0xfc000=0xBF8D，如图6-51所示。

```
__text:0000BF68                 EXPORT _TestFunction3
__text:0000BF68 _TestFunction3                          ; CODE XREF: _main+1
__text:0000BF68
__text:0000BF68 var_C           = -0xC
__text:0000BF68
__text:0000BF68                 PUSH            {R7,LR}
__text:0000BF6A                 MOV             R7, SP
__text:0000BF6C                 SUB             SP, SP, #4
__text:0000BF6E                 MOVS            R0, #3
__text:0000BF74                 BLX             _arc4random_uniform
__text:0000BF78                 MOV             R1, #(cfstr_IosreU - 0xBF84)
__text:0000BF80                 ADD             R1, PC  ; "IOSRE: %u"
__text:0000BF82                 STR             R0, [SP,#0xC+var_C]
__text:0000BF84                 MOV             R0, R1
__text:0000BF86                 LDR             R1, [SP,#0xC+var_C]
__text:0000BF88                 BLX             _NSLog
__text:0000BF8C                 ADD             SP, SP, #4
__text:0000BF8E                 POP             {R7,PC}
__text:0000BF8E ; End of function _TestFunction3
```

图6-51　TestFunction3

它位于TestFunction3中“BLX_NSLog”的正下方，我们找到了NSLog的调用者。有一点需要强调的是，因为LR在被调用者内部可能会产生变化，所以断点一定要下在基地址上。很简单吧？

2.执行“ni”命令到调用者内部

虽然“查看LR”的方法很简单，但在上面的例子里，我们耍了个小花样：因为事先知道MainBinary调用了NSLog，所以才用LR减去MainBinary的ASLR偏移得到地址，然后在IDA中跳过去。而一般情况下，我们不知道哪个函数调用了NSLog，更不知道哪个模块调用了NSLog，因此也就不知道该用LR减去谁的ASLR偏移了。要解决这个问题，我们的理论依据仍是“B执行结束后，需要回到A里，继续执行接下来的指令”——只要在被调用者的末尾下个断点，然后一直执行“ni”命令，就会回到调用者内部，从而发现调用者。还是来操作一遍：重复上面的步骤，用debugserver重新启动MainBinary，用LLDB挂接过去，直到进入MainBinary内部，然后查看Foundation.framework的ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x0000c000
/private/var/tmp/MainBinary(0x0000000000010000)
[  1] 0x000c5000
```

```
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1 (12B411)/Symbols/usr/lib/dyld
[  2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOSDeviceSuppor

(12B411)/Symbols/System/Library/Frameworks/Foundation.fra

……
```

它的ASLR偏移是0x6db3000。依图6-50，NSLog最后一条指令的地址是0x2261ABB6，因此，在0x6db3000+0x2261ABB6=0x293CDBB6上下一个断点，然后执行“c”命令，触发断点，如下：

```
(lldb) br s -a 0x293CDBB6
Breakpoint 1: where = Foundation`NSLog + 34, address =
0x293cdbb6
(lldb) c
Process 452269 resuming
(lldb) 2014-11-30 23:45:37.070 MainBinary[3454:452269]
iOSRE: 1
Process 452269 stopped
* thread #1: tid = 0x6e6ad, 0x293cdbb6 Foundation`NSLog +
34, queue = 'com.apple.main-thread, stop reason =
breakpoint 1.1
    frame #0: 0x293cdbb6 Foundation`NSLog + 34
Foundation`NSLog + 34:
-> 0x293cdbb6:  bx     lr
Foundation`NSLogv:
   0x293cdbb8:  push   {r4, r5, r6, r7, lr}
   0x293cdbba:  add    r7, sp, #12
   0x293cdbbc:  sub    sp, #12
```

注意“->”上方的文字，它指示了当前的模块。接

着执行“ni”命令，如下：

```
(lldb) ni
Process 452269 stopped
* thread #1: tid = 0x6e6ad, 0x00017fa6 MainBinary`main +
22, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x00017fa6 MainBinary`main + 22
MainBinary`main + 22:
-> 0x17fa6:  movs   r0, #0
   0x17fa8:  movt   r0, #0
   0x17fac:  add    sp, #12
   0x17fae:  pop    {r7, pc}
```

进入了MainBinary，停在了0x17fa6。0x17fa6–0xc000=0xbfa6，对照图6-51，我们找到了NSLog的调用者TestFunction3。

两种寻找调用者的方法都很简单粗暴，大家根据自己的喜好随便选一种就可以了。

6.3.2 更改进程执行逻辑

为什么要更改进程执行逻辑？最常见的原因之一是因为有些时候，你想要调试的代码需要满足一定的条件才能触发执行，而这种条件不借助外界力量很难

重现，所以可以更改进程执行逻辑，把进程引导向目标代码，从而调试它们。这句话听起来很拗口，举一个例子你就清楚了。看下面这样一段代码：

```
// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --
show-sdk-path` -framework Foundation -framework UIKit -o
MainBinary main.m
#include <stdio.h>
#include <dlfcn.h>
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>
extern void ImportantAndComplicatedFunction(void)
{
      NSLog(@"iOSRE: Suppose I'm a very important and
complicated function");
}
int main(int argc, char **argv)
{
      if ([[[UIDevice currentDevice] systemVersion]
isEqualToString:@"8.1.1"])
ImportantAndComplicatedFunction();
      return 0;
}
```

把这段代码存成名为main.m的文件，用注释里的那句话编译它，然后把MainBinary拷到iOS的“/var/tmp/”下，如下：

```
snakeninnys-MacBook:6 snakeninny$ scp MainBinary
root@iOSIP:/var/tmp/
MainBinary                          100%49KB   48.6KB/s
00:00
```

运行它，效果如下：

```
FunMaker-5:~ root# /var/tmp/MainBinary
FunMaker-5:~ root#
```

因为笔者的iOS系统是8.1，所以自然没有任何输出。笔者对ImportantAndComplicated-Function很感兴趣，想动态调试它，但手头没有8.1.1的系统，怎么办呢？那就动态更改代码，让这个函数得到执行。下面来操作一遍，请读者注意观察。先把MainBinary拖进IDA，定位到ImportantAndComplicatedFunction被调用之前的指令，如图6-52所示。

然后用debugserver启动MainBinary，用LLDB挂接过去，直到进入MainBinary内部，再查看MainBinary的ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x0000e000
/private/var/tmp/MainBinary(0x0000000000012000)
……
```

因为图6-52最上面的那个“CMP R0,#0”地址是0xBF46，所以把断点下在0xbf46+0xe000=0x19F46，然后执行“c”命令触发它，然后看看R0的值，如下：

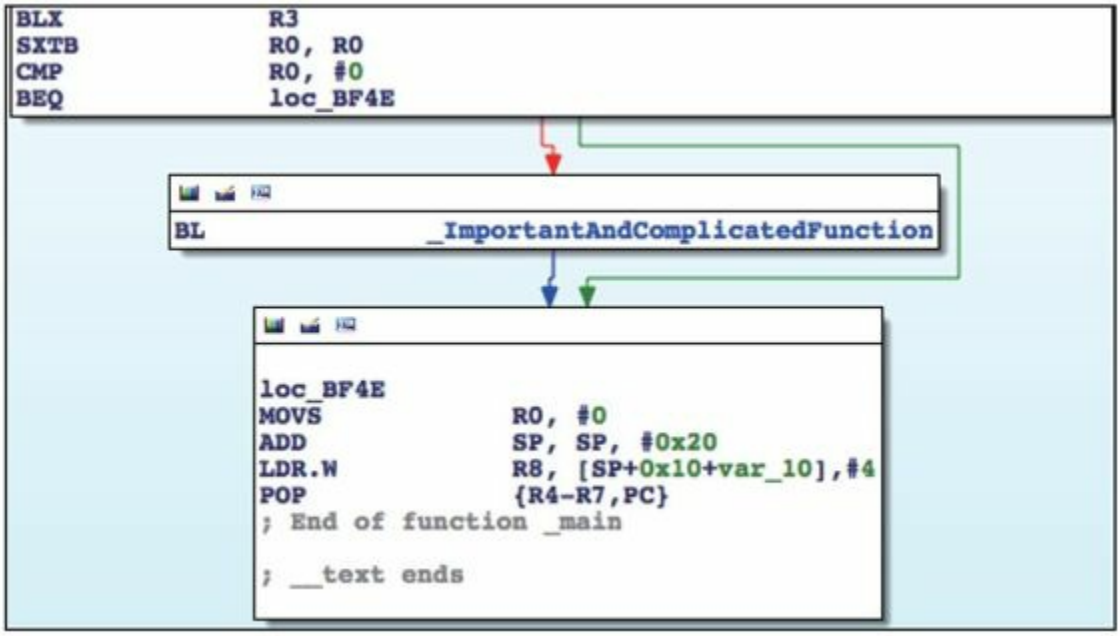


图6-52　ImportantAndComplicatedFunction得到调用之前

```
(lldb) br s -a 0x19F46
Breakpoint 1: where = MainBinary`main + 134, address =
0x00019f46
(lldb) c
Process 456316 resuming
Process 456316 stopped
* thread #1: tid = 0x6f67c, 0x00019f46 MainBinary`main +
134, queue = 'com.apple.main-thread, stop reason =
breakpoint 1.1
    frame #0: 0x00019f46 MainBinary`main + 134
MainBinary`main + 134:
-> 0x19f46:  cmp    r0, #0
   0x19f48:  beq    0x19f4e                    ; main +
142
```

```
   0x19f4a:  bl     0x19ea4                   ;
ImportantAndComplicatedFunction
   0x19f4e:  movs   r0, #0
(lldb) p $r0
(unsigned int) $0 = 0
```

R0是0，因此ImportantAndComplicatedFunction得不到执行。如果把R0改成1，情况就不同了，如下：

```
(lldb) register write r0 1
(lldb) p $r0
(unsigned int) $1 = 1
(lldb) c
Process 456316 resuming
(lldb) 2014-12-01 00:41:47.779 MainBinary[3482:457105]
iOSRE: Suppose I'm a very important and complicated
function
Process 456316 exited with status = 0 (0x00000000)
```

我们通过动态更改寄存器的值来改变进程执行逻辑，达到了目的。

6.4 小结

IDA和LLDB两大神器的作用当然不止于本章所介绍的这些，它们的有效范围很广，小到分析App，大到动手越狱，是两款“老少咸宜”的工具。不过，相信在iOS逆向工程初级阶段，大家应用它们的场合不会超出本章的内容范围。当然，熟练掌握它们以后，对iOS的理解一定会上升到一个新的层次；届时，大家就能举一反三，根据自己的需求摸索它们的新用法了。在ARM汇编级别的iOS逆向工程里，值得悉心研究的课题还有很多，我们会在http://bbs.iosre.com上展开旷日持久的讨论。

本章的内容虽然有些艰深，却是入门iOS逆向工程的基础。接下来的4章会将本章内容运用到实战中去，在阅读完那些内容之后，大家就能根据自己的掌握情况判断是知难而退，还是迎难而上了。无论如何，这是一个很有意思的方向，能走多远则完全看个

人的功底和兴趣了。

第四部分　实战篇

· 第7章　实战1：Characount for Notes 8

· 第8章　实战2：自动将指定电子邮件标记为已读

· 第9章　实战3：保存与分享微信小视频

· 第10章　实战4：检测与发送iMessage

前面三个部分重点介绍了iOS应用逆向工程的基本概念、工具应用和相关理论，其中穿插的实例有助于增进大家对iOS逆向工程的了解，相信大家也都已经感受到，只有把理论、工具和思想有机结合，才能发挥逆向工程的最大威力。

在完成前三部分之后，很多朋友可能会觉得前面那些零散的简单例子还是过于保守，缺乏一种酣畅淋漓的感觉。因此在实战篇中，我们会用4个原创实例

演示理论、工具和思想的有机结合，这部分的逆向目标是：

· Characount for Notes 8

· 自动将指定电子邮件标记为已读

· 保存与分享微信小视频

· 检测与发送iMessage

接下来，就请进入本书最精彩的部分，通过一个个精彩实例去感受iOS逆向工程的强大威力。

第7章　实战1：Characount for Notes 8

7.1　备忘录

iOS的备忘录（以下简称Notes）想必是所有果粉最熟悉的App之一了，从iOS出生到现在，备忘录的风格和功能就没有过大的变动，足见其经典。简洁的风格和便捷的输入让它抓住了笔者的心，在笔者的Notes里记满了自己的小秘密，如图7-1所示。

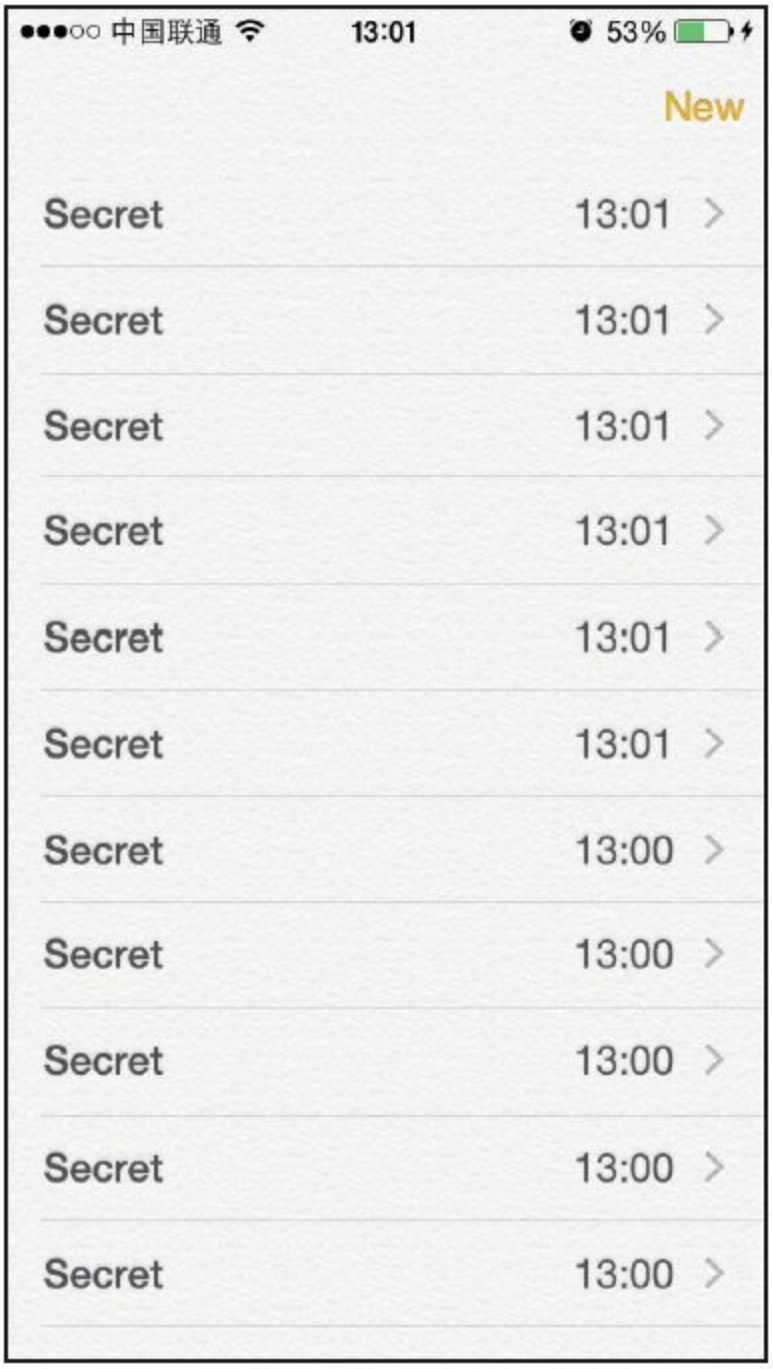


图7-1　Notes界面

作为Notes的重度用户，笔者不但用它来记录秘密，对于一些需要精雕细琢的短信、微博，笔者都会

在Notes里编辑完成后，再发到相应的平台上。因为这类内容都有字数限制，所以笔者也希望Notes可以多一项苹果没有提供的功能——显示每条Note的字数。出于自己动手，丰衣足食的原则，笔者自行开发了Characount for Notes，它是笔者在iOS 6时代使用频率最高的插件之一。因为它难度不大，适合作为iOS逆向工程初学者的敲门砖，所以本章的任务就是在iOS 8上重写Characount for Notes，下面的操作在iPhone 5，iOS 8.1中完成。

7.2 搭建tweak原型

在iOS 8中，Notes原始的阅览界面是这样的，如图7-2所示。

要在这个界面显示这条note的字数，在哪里显示比较好看呢？不知道你对iOS 6上的Notes有没有印象，那时的每条note都有一个居中显示的标题，如图7-3所示。

图7-2　iOS 8的Notes阅览界面

图7-3　iOS 6的Notes阅览界面

iOS 8把这个标题给去掉了，整个导航栏显得空空荡荡的，不如我们就把字数加在标题所在的位置，如图7-4所示。

效果还不赖！要把Notes改造成这样，需要做些什么工作呢？还记得第5章里说过，在iOS里你看见的每个东西都是一个对象吗？记住这个准则，一起来思考。

1）每条note都是一个对象，阅览界面涵盖了一条note的内容及修改时间等信息，这些信息都来源于这条note。阅览界面是一个view，可以通过nextResponder追溯到它的controller，用controller又可以访问note的相关数据，可以用于在刚刚进入阅览界面的时候初始化字数标题。

2）当编辑一条note时，阅览界面的右上方会出现一个“Done”的按钮，如图7-5所示。

点击“Done”之后，这条note被保存下来。这个现象说明一条note在编辑过程中是不会实时保存的，不然就不需要这个按钮了。而字数标题随着内容的编辑实时变化的效果是最好的，要达到这种效果，就需要

一个实时监测note内容变化的方法，且要从这个方法里拿到当前的字数，实时更新标题——因为这种方法一般是定义在protocol里的，所以要留意各种protocol里有没有出现这类函数。

3）如果已经搞定了字数，要怎么把它放在导航栏上呢？阅览界面的controller想必是一个UIViewController的子类，而UIViewController有一个名为title的property，因此只要简简单单地调用setTitle:就可以了。

图7-4　加上字数之后的阅览界面

图7-5　阅览界面出现“Done”按钮

如果能解决上面3个问题，Characount for Notes的技术难点就算全部拿下。没有更多需要解释的了，我们开始动手吧~

7.2.1 定位Notes的可执行文件

在“/Applications/”下踅摸了一圈，没有名为“Notes.app”的文件夹。这种情况下，该怎么定位Notes所在的文件夹呢？还记得在dumpdecrypted章节里找App目录的小技巧吗？是的，就是用ps命令：先关掉所有的App，然后打开Notes。接着ssh到iOS中，用ps命令看看当前有哪些进程来自“/Applications/”，如下：

```
FunMaker-5:~ root# ps -e | grep /Applications/
 592 ??         0:37.70
/Applications/MobileMail.app/MobileMail
 761 ??         0:02.78
/Applications/MessagesNotificationViewService.app/Message

 1807 ??         0:00.55
/private/var/db/stash/_.29LMeZ/Applications/MobileSafari.

 2016 ??         0:05.23
/Applications/InCallService.app/InCallService
 2619 ??         0:02.66
/Applications/MobileSMS.app/MobileSMS
 2672 ??         0:01.20
/Applications/MobileNotes.app/MobileNotes
 2678 ttys000    0:00.01 grep /Applications/
```

其中，最可疑的当然就是MobileNotes了，怎么验

证呢？kill掉它，看看已经打开的Notes会不会闪退，如下：

```
FunMaker-5:~ root# killall MobileNotes
```

Notes果然退出了，说明“/Applications/MobileNotes.app/MobileNotes”就是Notes的可执行文件，而且同时还知道了“/Applications/”下运行在后台的一些应用。把MobileNotes拷贝到OSX中，准备class-dump。

7.2.2 class-dump出MobileNotes的头文件

因为Notes不是从AppStore下载的，没有加壳，所以可以直接使用class-dump，如下：

```
snakeninnys-MacBook:~ snakeninny$ class-dump -S -s -H
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/Mobi
 -o
/Users/snakeninny/Code/iOSPrivateHeaders/8.1/MobileNotes
```

一共有88个头文件，粗略扫一眼，看看能发现什

么，如图7-6所示。

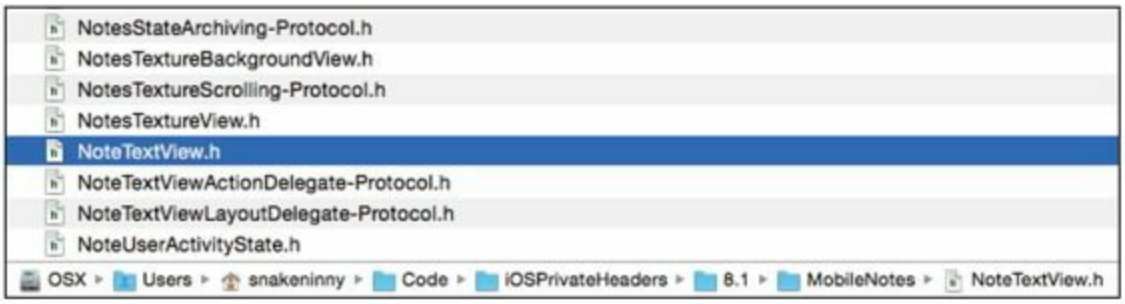


图7-6　class-dump头文件

看到图7-6中选中的文件了吗？是不是它，我们现在不知道，也不用急于猜测，结果马上就会揭晓了。

7.2.3　用Cycript找到阅览界面及其controller

百试不爽的recursiveDescription又要派上用场了，如下：

```
FunMaker-5:~ root# cycript -p MobileNotes
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
@"<UIWindow: 0x17688db0; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x17689620>; layer =
<UIWindowLayer: 0x17688fc0>>
   | <UILayoutContainerView: 0x175bb880; frame = (0 0;
320 568); autoresize = W+H; layer = <CALayer:
```

```
0x175bb900>>
   |    | <UILayoutContainerView: 0x17699350; frame = (0
0; 320 568); clipsToBounds = YES; gestureRecognizers =
<NSArray: 0x1769cf60>; layer = <CALayer: 0x17699530>>
   |    |    | <UINavigationTransitionView: 0x176564c0;
frame = (0 0; 320 568); clipsToBounds = YES; autoresize =
W+H; layer = <CALayer: 0x17658ec0>>
   |    |    |    | <UIViewControllerWrapperView:
0x176d13b0; frame = (0 0; 320 568); layer = <CALayer:
0x176d1530>>
   |    |    |    |    | <UILayoutContainerView:
0x1769dd80; frame = (0 0; 320 568); clipsToBounds = YES;
gestureRecognizers = <NSArray: 0x176a16f0>; layer =
<CALayer: 0x1769de00>>
   |    |    |    |    |    |
<UINavigationTransitionView: 0x1769ebb0; frame = (0 0;
320 568); clipsToBounds = YES; autoresize = W+H; layer =
<CALayer: 0x1769ec40>>
   |    |    |    |    |    |    |
<UIViewControllerWrapperView: 0x175109e0; frame = (0 0;
320 568); layer = <CALayer: 0x175109b0>>
   |    |    |    |    |    |    |    |
<NotesBackgroundView: 0x175ee3e0; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x17510a70>; layer =
<CALayer: 0x175ee580>>
   |    |    |    |    |    |    |    |    |
<NotesTextureBackgroundView: 0x175ee5b0; frame = (0 0;
320 568); clipsToBounds = YES; layer = <CALayer:
0x175ee630>>
   |    |    |    |    |    |    |    |    |    |
<NotesTextureView: 0x175ee940; frame = (0 -64; 320 640);
layer = <CALayer: 0x175ee9c0>>
   |    |    |    |    |    |    |    |    |
<NoteContentLayer: 0x176c5110; frame = (0 0; 320 568);
layer = <CALayer: 0x176ca850>>
   |    |    |    |    |    |    |    |    |    |
<UIView: 0x175f2130; frame = (16 0; 288 0); hidden = YES;
layer = <CALayer: 0x175dd2b0>>
   |    |    |    |    |    |    |    |    |    |
<NotesScrollView: 0x175f2a10; baseClass = UIScrollView;
frame = (0 0; 320 568); clipsToBounds = YES;
gestureRecognizers = <NSArray: 0x175f1b70>; layer =
<CALayer: 0x175f28d0>; contentOffset: {0, -64};
contentSize: {320, 460}>
   |    |    |    |    |    |    |    |    |    |    |
<UIView: 0x175f09a0; frame = (0 0; 320 0); layer =
<CALayer: 0x175f2790>>
   |    |    |    |    |    |    |    |    |    |    |
<UIView: 0x175f27e0; frame = (0 0; 0 460); layer =
```

```
<CALayer: 0x175f2850>>
   |    |    |    |    |    |    |    |    |    |    |
<NoteDateLabel: 0x175f3400; baseClass = UILabel; frame =
(69 5.5; 182 18); text = 'November 24, 2014, 20:44';
userInteractionEnabled = NO; layer = <_UILabelLayer:
0x175f3560>>
   |    |    |    |    |    |    |    |    |    |    |
<NoteTextView: 0x175ee3e0; baseClass =
_UICompatibilityTextView; frame = (6 28; 308 418); text =
'Secret'; clipsToBounds = YES; gestureRecognizers =
<NSArray: 0x176c7ed0>; layer = <CALayer: 0x176d88e0>;
contentOffset: {0, 0}; contentSize: {308, 52}>
……
```

果不其然有一个NoteTextView，且“Secret”就位于其中。持续调用nextResponder，找出它的controller，如下：

```
cy# [#0x175ee3e0 nextResponder]
#"<NotesScrollView: 0x17d307c0; baseClass = UIScrollView;
frame = (0 0; 320 568); clipsToBounds = YES;
gestureRecognizers = <NSArray: 0x17e502a0>; layer =
<CALayer: 0x17d30b60>; contentOffset: {0, -64};
contentSize: {320, 251}>"
cy# [#0x17d307c0 nextResponder]
#"<NoteContentLayer: 0x17e505b0; frame = (0 0; 320 568);
layer = <CALayer: 0x17e50470>>"
cy# [#0x17e505b0 nextResponder]
#"<NotesBackgroundView: 0x17e52320; frame = (0 0; 320
568); gestureRecognizers = <NSArray: 0x17d0c940>; layer =
<CALayer: 0x17e522f0>>"
cy# [#0x17e52320 nextResponder]
#"<NotesDisplayController: 0x17edc340>"
```

好的，就是NoteDisplayController了。看看如下直接调用setTitle:能否改变阅览界面的标题：

```
cy# [#0x17edc340 setTitle:@"Characount = Character
count"]
```

效果如图7-7所示。

图7-7　setTitle:的效果

没有任何问题，第一目标达成！

7.2.4 从NoteDisplayController找到当前note对象

趁热打铁，到NoteDisplayController.h里看看它的定义，如下：

```
@interface NotesDisplayController : UIViewController
<NoteContentLayerDelegate, UIActionSheetDelegate,
AFContextProvider,
UIPopoverPresentationControllerDelegate,
UINavigationControllerDelegate,
UIImagePickerControllerDelegate,
NotesQuickLookActivityItemDelegate,
ScrollViewKeyboardResizerDelegate,
NSUserActivityDelegate, NotesStateArchiving>
{
……
@property(nonatomic, getter=isVisible) BOOL visible; //
@synthesize visible=_visible;
- (void)loadView;
@property(retain, nonatomic) NoteObject *note; //
@synthesize note=_note;
……
}
```

文件的内容很多，通览之后，我们发现了一个NoteObject类型的属性。虽说一个note就是一个对象，但NoteObject的名称含义是不是也太明显了……在Cycript里把它打印出来看看，如下：

```
cy# [#0x17edc340 note]
#'<NoteObject: 0x176aa170> (entity: Note; id: 0x176a9040
<x-coredata://4B88CC7C-7A5F-4F15-9275-
53C6D0ABE0C3/Note/p15> ; data: {\n    attachments =
(\n    );\n    author = nil;\n    body = "0x176a8b20 <x-
coredata://4B88CC7C-7A5F-4F15-9275-
53C6D0ABE0C3/NoteBody/p15>";\n    containsCJK = 0;\n
contentType = 0;\n    creationDate = "2014-11-24 05:00:59
+0000";\n    deletedFlag = 0;\n    externalFlags = 0;\n
externalSequenceNumber = 0;\n    externalServerIntId =
"-4294967296";\n    guid = "781B6C87-2855-4512-8864-
50618754333A";\n    integerId = 3865;\n
isBookkeepingEntry = 0;\n    modificationDate = "2014-11-
24 12:44:08 +0000";\n    serverId = nil;\n    store =
"0x175a2b60 <x-coredata://4B88CC7C-7A5F-4F15-9275-
53C6D0ABE0C3/Store/p1>";\n    summary = nil;\n    title =
Secret;\n})'
```

很明显，NoteObject就是当前显示的note，各个字段含义都比较清晰，现在去看看它的定义，如下：

```
@interface NoteObject : NSManagedObject
{
}
- (BOOL)belongsToCollection:(id)arg1;
@property(nonatomic) unsigned long long sequenceNumber;
- (BOOL)containsAttachments;
@property(retain, nonatomic) NSString
*externalContentRef;
@property(retain, nonatomic) NSData
*externalRepresentation;
@property(readonly, nonatomic) BOOL hasValidServerIntId;
@property(nonatomic) long long serverIntId;
@property(nonatomic) unsigned long long flags;
@property(readonly, nonatomic) NSURL *noteId;
@property(readonly, nonatomic) BOOL
isBeingMarkedForDeletion;
@property(readonly, nonatomic) BOOL isMarkedForDeletion;
- (void)markForDeletion;
@property(nonatomic) BOOL isPlainText;
- (id)contentAsPlainTextPreservingNewlines;
```

```
@property(readonly, nonatomic) NSString
*contentAsPlainText;
@property(retain, nonatomic) NSString *content;
// Remaining properties
@property(retain, nonatomic) NSSet *attachments; //
@dynamic attachments;
@property(retain, nonatomic) NSString *author; //
@dynamic author;
@property(retain, nonatomic) NoteBodyObject *body; //
@dynamic body;
@property(retain, nonatomic) NSNumber *containsCJK; //
@dynamic containsCJK;
@property(retain, nonatomic) NSNumber *contentType; //
@dynamic contentType;
@property(retain, nonatomic) NSDate *creationDate; //
@dynamic creationDate;
@property(retain, nonatomic) NSNumber *deletedFlag; //
@dynamic deletedFlag;
@property(retain, nonatomic) NSNumber *externalFlags; //
@dynamic externalFlags;
@property(retain, nonatomic) NSNumber
*externalSequenceNumber; // @dynamic
externalSequenceNumber;
@property(retain, nonatomic) NSNumber
*externalServerIntId; // @dynamic externalServerIntId;
@property(readonly, retain, nonatomic) NSString *guid; //
@dynamic guid;
@property(retain, nonatomic) NSNumber *integerId; //
@dynamic integerId;
@property(retain, nonatomic) NSNumber
*isBookkeepingEntry; // @dynamic isBookkeepingEntry;
@property(retain, nonatomic) NSDate *modificationDate; //
@dynamic modificationDate;
@property(retain, nonatomic) NSString *serverId; //
@dynamic serverId;
@property(retain, nonatomic) NoteStoreObject *store; //
@dynamic store;
@property(retain, nonatomic) NSString *summary; //
@dynamic summary;
@property(retain, nonatomic) NSString *title; // @dynamic
title;
@end
```

非常好，这么多的property表明NoteObject是个非

常标准的model。如何获取它的文字内容呢？在上面的代码中，看到了一个名为contentAsPlainText的property，像下面这样调用它看看是什么效果：

```
cy# [#0x176aa170 contentAsPlainText]
@"Secret"
```

为了进一步确认，改一下这条note的文字，再配一张图，如图7-8所示。

图7-8　重新编辑这条note

然后重新调用contentAsPlainText，如下：

```
cy# [#0x176aa170 contentAsPlainText]
@"bbs.iosre.com"
```

基本可以确定这个函数能够正确返回当前note的文字内容了，对它调用length就可以拿到这条note的文字个数，如下：

```
cy# [[#0x176aa170 contentAsPlainText] length]
13
```

还有最后一项任务，咱们快马加鞭，把它搞定！

7.2.5 找到实时监测note内容变化的方法

在本章开头部分已经提到，“实时监测note内容变化的方法一般是定义在protocol里的”。因为设置标题的函数，以及获取note对象的操作都是通过NotesDisplayController类完成的，所以如果能在这个类里找到一个符合条件的方法，那另两项操作就可以放在这个方法里完成了，可以极大地简化代码。打开NotesDisplayController.h，看看它实现了哪些协议，如下：

```
@interface
NotesDisplayController:UIViewController<NoteContentLayerD

……
@end
```

其中UIActionSheetDelegate、UIPopoverPresentationControllerDelegate、UINavigation-ControllerDelegate、UIImagePickerControllerDelegate都是公开协议，明显跟note内容的变化没关系，可以直接排除掉了。剩下的NoteContentLayerDelegate、AFContextProvider、NotesQu-ickLookActivityItemDelegate、ScrollViewKeyboardResizerDelegate、NSUserActivityDelegate和NotesStateArchiving都不能轻易放过，需要逐个排查。先看NoteContentLayerDelegate-Protocol.h，如下：

```
@protocol NoteContentLayerDelegate <NSObject>
- (BOOL)allowsAttachmentsInNoteContentLayer:(id)arg1;
- (BOOL)canInsertImagesInNoteContentLayer:(id)arg1;
- (void)insertImageInNoteContentLayer:(id)arg1;
- (BOOL)isNoteContentLayerVisible:(id)arg1;
- (BOOL)noteContentLayer:(id)arg1
acceptContentsFromPasteboard:(id)arg2;
- (BOOL)noteContentLayer:(id)arg1
acceptStringIncreasingContentLength:(id)arg2;
```

```
- (BOOL)noteContentLayer:(id)arg1
canHandleLongPressOnElement:(id)arg2;
- (void)noteContentLayer:(id)arg1 containsCJK:(BOOL)arg2;
- (void)noteContentLayer:(id)arg1
contentScrollViewWillBeginDragging:(id)arg2;
- (void)noteContentLayer:(id)arg1 didChangeContentSize:
(struct CGSize)arg2;
- (void)noteContentLayer:(id)arg1
handleLongPressOnElement:(id)arg2 atPoint:(struct
CGPoint)arg3;
- (void)noteContentLayer:(id)arg1 setEditing:(BOOL)arg2
nimated:(BOOL)arg3;
- (void)noteContentLayerContentDidChange:(id)arg1
updatedTitle:(BOOL)arg2;
- (BOOL)noteContentLayerShouldBeginEditing:(id)arg1;
@optional
- (void)noteContentLayerKeyboardDidHide:(id)arg1;
@end
```

其中，noteContentLayer:didChangeContentSize:和noteContentLayerContentDidChange:u-pdatedTitle:这两个方法有些可疑，编辑一条note时，这条note的内容和内容所占的尺寸都在实时变化，因此这两个方法确实有被实时调用的可能性。查看NotesDisplayController.h，这两个协议方法也都被实现了。为了确定它们有没有被实时调用，考虑用LLDB验证一下。

用LLDB附加MobileNotes，看看MobileNotes的

ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x00035000
/private/var/db/stash/_.29LMeZ/Applications/MobileNotes.a

[  1] 0x00197000
/Library/MobileSubstrate/MobileSubstrate.dylib
(0x0000000000197000)
[  2] 0x06db3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/QuickLook.fram

……
```

ASLR偏移是0x35000。然后把MobileNotes拖进IDA，待初始分析完成后，查看[NotesDisplay-Controller noteContentLayer:didChangeContentSize:]和[NotesDisplayController noteContent-LayerContent-DidChange:updatedTitle:]的基地址，如图7-9和图7-10所示。

```
__text:00016E70 ; NotesDisplayController - (void)noteContentLayer:(id) didChangeContentSize:(struc
__text:00016E70
__text:00016E70 ; void __cdecl -[NotesDisplayController noteContentLayer:didChangeContentSize:](st
__text:00016E70 __NotesDisplayController_noteContentLayer_didChangeContentSize__
__text:00016E70                                         ; DATA XREF: __objc_const:0004C680↓o
__text:00016E70                 MOV             R1, #(selRef_reloadSearchedTermHighlight - 0x16E7C
__text:00016E78                 ADD             R1, PC ; selRef_reloadSearchedTermHighlight
__text:00016E7A                 LDR             R1, [R1] ; "reloadSearchedTermHighlight"
__text:00016E7C                 B.W             j__objc_msgSend
__text:00016E7C ; End of function -[NotesDisplayController noteContentLayer:didChangeContentSize:]
```

图7-9　[NotesDisplayController

noteContentLayer:didChangeContentSize:]

图7-10　[NotesDisplayController noteContentLayerContentDidChange:updatedTitle:]

两者的基地址分别是0x16E70和0x1AEB8，因此断点地址分别是0x4BE70和0x4FEB8。下2个断点，然后随便打开一条note并编辑它，看看断点会不会停，如下：

```
(lldb) br s -a 0x4BE70
Breakpoint 1: where =
MobileNotes`___lldb_unnamed_function382$$MobileNotes,
address = 0x0004be70
(lldb) br s -a 0x4FEB8
Breakpoint 2: where =
MobileNotes`___lldb_unnamed_function458$$MobileNotes,
address = 0x0004feb8
```

你得到的结果肯定跟笔者的一模一样——2个断点都会被触发很多次！协议方法被调用，一般是因为

方法名中提到的那个事件发生了；而那件事发生的对象，一般是协议方法的参数。在当前情况下，则表明发生了didChangeContentSize和ContentDidChange事件，而content本身很可能就是参数。接着来看看这两个函数的第一个参数是什么，如下：

```
(lldb) br com add 1
Enter your debugger command(s).  Type 'DONE' to end.
> po $r2
> c
> DONE
(lldb) br com add 2
Enter your debugger command(s).  Type 'DONE' to end.
> po $r2
> c
> DONE
(lldb) c
```

可以看到，输出中有很多的NoteContentLayer，如下：

```
Process 24577 resuming
Command #2 'c' continued the target.
<NoteContentLayer: 0x14ecdf50; frame = (0 0; 320 568);
animations = { bounds.origin=<CABasicAnimation:
0x16fee090>; bounds.size=<CABasicAnimation: 0x16fee4a0>;
position=<CABasicAnimation: 0x16fee500>; }; layer =
<CALayer: 0x14eca900>>
Process 24577 resuming
Command #2 'c' continued the target.
<NoteContentLayer: 0x14ecdf50; frame = (0 0; 320 568);
animations = { bounds.origin=<CABasicAnimation:
```

```
0x16fee090>; bounds.size=<CABasicAnimation: 0x16fee4a0>;
position=<CABasicAnimation: 0x16fee500>; }; layer =
<CALayer: 0x14eca900>>
Process 24577 resuming
Command #2 'c' continued the target.
<NoteContentLayer: 0x14ecdf50; frame = (0 0; 320 568);
layer = <CALayer: 0x14eca900>>
Process 24577 resuming
Command #2 'c' continued the target.
```

既然能拿到NoteContentLayer，多半能够从中获取NoteContent。打开NoteContentLayer.h，看看它提供了些什么方法，如下：

```
@interface NoteContentLayer : UIView
<NoteTextViewActionDelegate, Note TextViewLayoutDelegate,
UITextViewDelegate>
……
@property(retain, nonatomic) NoteTextView *textView; //
@synthesize textView=_textView;
……
@end
```

NoteContentLayer有一个NoteTextView的属性，而在本章的开头，用Cycript打印UI层次的时候，发现一条note的文字就是显示在NoteTextView之上的。更改一下断点命令，把NoteTextView打印出来看看，如下：

```
(lldb) br com add 1
Enter your debugger command(s).  Type 'DONE' to end.
> po [$r2 textView]
> c
> DONE
(lldb) br com add 2
Enter your debugger command(s).  Type 'DONE' to end.
> po [$r2 textView]
> c
> DONE
```

然后继续编辑这条note，发现对note文字的改动全都体现在了LLDB的输出上，如下：

```
Process 24577 resuming
Command #2 'c' continued the target.
<NoteTextView: 0x15aace00; baseClass =
_UICompatibilityTextView; frame = (6 28; 308 209); text =
'Secre'; clipsToBounds = YES; gestureRecognizers =
<NSArray: 0x14eddfc0>; layer = <CALayer: 0x14ee7da0>;
contentOffset: {0, 0}; contentSize: {308, 52}>
Process 24577 resuming
Command #2 'c' continued the target.
<NoteTextView: 0x15aace00; baseClass =
_UICompatibilityTextView; frame = (6 28; 308 209); text =
'Secret'; clipsToBounds = YES; gestureRecognizers =
<NSArray: 0x14eddfc0>; layer = <CALayer: 0x14ee7da0>;
contentOffset: {0, 0}; contentSize: {308, 52}>
```

最后一个步骤，就是从NoteTextView拿到text。打开NoteTextView.h，如下：

```
@interface NoteTextView : _UICompatibilityTextView
<UIGestureRecognizerDelegate>
{
    id <NoteTextViewActionDelegate> _actionDelegate;
```

```
    id <NoteTextViewLayoutDelegate> _layoutDelegate;
        ……
}
……
@property(nonatomic) __weak id
<NoteTextViewActionDelegate> actionDelegate; //
@synthesize actionDelegate=_actionDelegate;
……
@property(nonatomic) __weak id
<NoteTextViewLayoutDelegate> layoutDelegate; //
@synthesize layoutDelegate=_layoutDelegate;
……
@end
```

它的实现并不长，但里面唯一含有text字样的，是2个delegate，显然不会返回NSString对象。text不在它自己实现里，就一定在它的父类里，打开_UICompatibilityTextView.h，如下：

```
@interface _UICompatibilityTextView : UIScrollView
<UITextLinkInteraction, UITextInput>
……
@property(nonatomic) int textAlignment;
@property(copy, nonatomic) NSString *text;-
(BOOL)hasText;
@property(retain, nonatomic) UIColor *textColor;
@property(retain, nonatomic) UIFont *font;
@property(copy, nonatomic) NSAttributedString
*attributedText;
……
```

原来text在这里。用LLDB做最后的确认，如下：

```
(lldb) br com add 1
```

```
Enter your debugger command(s).  Type 'DONE' to end.
> po [[$r2 textView] text]
> c
> DONE
(lldb) br com add 2
Enter your debugger command(s).  Type 'DONE' to end.
> po [[$r2 textView] text]
> c
> DONE
Secret
Process 24577 resuming
Command #2 'c' continued the target.
Secret i
Process 24577 resuming
Command #2 'c' continued the target.
```

至此，我们成功找到了2个实时监测note内容变化的方法（随便选一个就好了，我们选第二个），并且可以拿到note的实时文本数据，早前设计的3个功能现在已经全部实现。不难吧？

7.3 逆向结果整理

本章的实例是针对iOS系统App的，在完全脱离IDA，仅凭Cycript和LLDB的情况下就可实现相应的功能（其实这里是用LLDB做了hook的工作，换用Theos可达到同样效果），虽然有一定的运气成分，但这也正体现了逆向工程的不确定性。为了完成Characount for Notes 8，我们的大致思路是下面这样的。

1.在界面上寻找适合显示字数的地方和方法

Notes从iOS 6演变到iOS 8时，原来的标题给演变没了，正好给我们留下了一个显示字数的地方。本章从note阅览界面入手，通过Cycript拿到NoteDisplayController，成功搞定显示字数的方法。

2.浏览class-dump出的头文件，在controller类里找到访问model的方法

通过controller访问model是MVC设计标准里规定的，苹果自身出品的App一定会遵守这个标准，因此在NoteDisplayController里一定能找到访问model的方法。我们仅仅通过浏览头文件，用Cycript测试可疑的属性，就拿到了NoteObject，从而拿到了一条note的字数。

3.在protocol里寻找实时监测字数变化的方法

实时调用的函数一般定义在protocol中，因为Objective-C的函数名可读性高，所以我们没有严谨地使用IDA和LLDB来寻找能够实时监测字数变化的方法，而是遍历了含有protocol关键词的头文件，人工筛选后再用LLDB测试，结果很快就找到了满足要求的方法。运气也好，猜测也罢，这就是逆向工程的魅力所在。

7.4 编写tweak

本章的例子比较简单，所有的操作都可以在NotesDisplayController一个类中完成。

7.4.1 用Theos新建tweak工程“CharacountForNotes8”

新建CharacountForNotes8工程的命令如下：

```
snakeninnys-MacBook:Code snakeninny$
/opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): CharacountForNotes8
Package Name [com.yourcompany.characountfornotes8]:
com.naken.characountfornotes8
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.apple.mobilenotes
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]: MobileNotes
Instantiating iphone/tweak in characountfornotes8/...
Done.
```

7.4.2 构造CharacountForNotes8.h

编辑后的CharacountForNotes8.h内容如下：

```
@interface NoteObject : NSObject
@property (readonly, nonatomic) NSString
*contentAsPlainText;
@end
@interface NoteTextView : UIView
@property (copy, nonatomic) NSString *text;
@end
@interface NoteContentLayer : UIView
@property (retain, nonatomic) NoteTextView *textView;
@end
@interface NotesDisplayController : UIViewController
@property (retain, nonatomic) NoteContentLayer
*contentLayer;
@property (retain, nonatomic) NoteObject *note;
@end
```

这个头文件的所有内容均摘自类对应的头文件，构造它的目的仅仅是通过编译，避免任何报错和警告。

7.4.3 编辑Tweak.xm

编辑后的Tweak.xm内容如下：

```
#import "CharacountForNotes8.h"
%hook NotesDisplayController
- (void)viewWillAppear:(BOOL)arg1 // Initialze title
{
      %orig;
      NSString *content = self.note.contentAsPlainText;
      NSString *contentLength = [NSString
stringWithFormat:@"%lu", (unsigned long)[content
length]];
      self.title = contentLength;
}
- (void)viewDidDisappear:(BOOL)arg1 // Reset title
{
      %orig;
      self.title = nil;
}
- (void)noteContentLayerContentDidChange:
(NoteContentLayer
*)arg1 updatedTitle:(BOOL)arg2 // Update title
{
      %orig;
      NSString *content =
self.contentLayer.textView.text;
      NSString *contentLength = [NSString
stringWithFormat:@"%lu", (unsigned long)[content
length]];
      self.title = contentLength;
}
%end
```

7.4.4 编辑Makefile及control

编辑后的Makefile内容如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
TWEAK_NAME = CharacountForNotes8
CharacountForNotes8_FILES = Tweak.xm
include $(THEOS_MAKE_PATH)/tweak.mk
```

```
after-install::
	install.exec "killall -9 MobileNotes"
```

编辑后的control内容如下：

```
Package: com.naken.characountfornotes8
Name: CharacountForNotes8
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Add a character count to Notes
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

7.4.5 测试

将写好的tweak编译打包安装到iOS后，打开Notes随便编辑一条note，查看标题的字数变化是否可做到完全实时，如图7-11至图7-17所示。

图7-11　Characount for Notes 8效果演示（1）

图7-12　Characount for Notes 8效果演示（2）

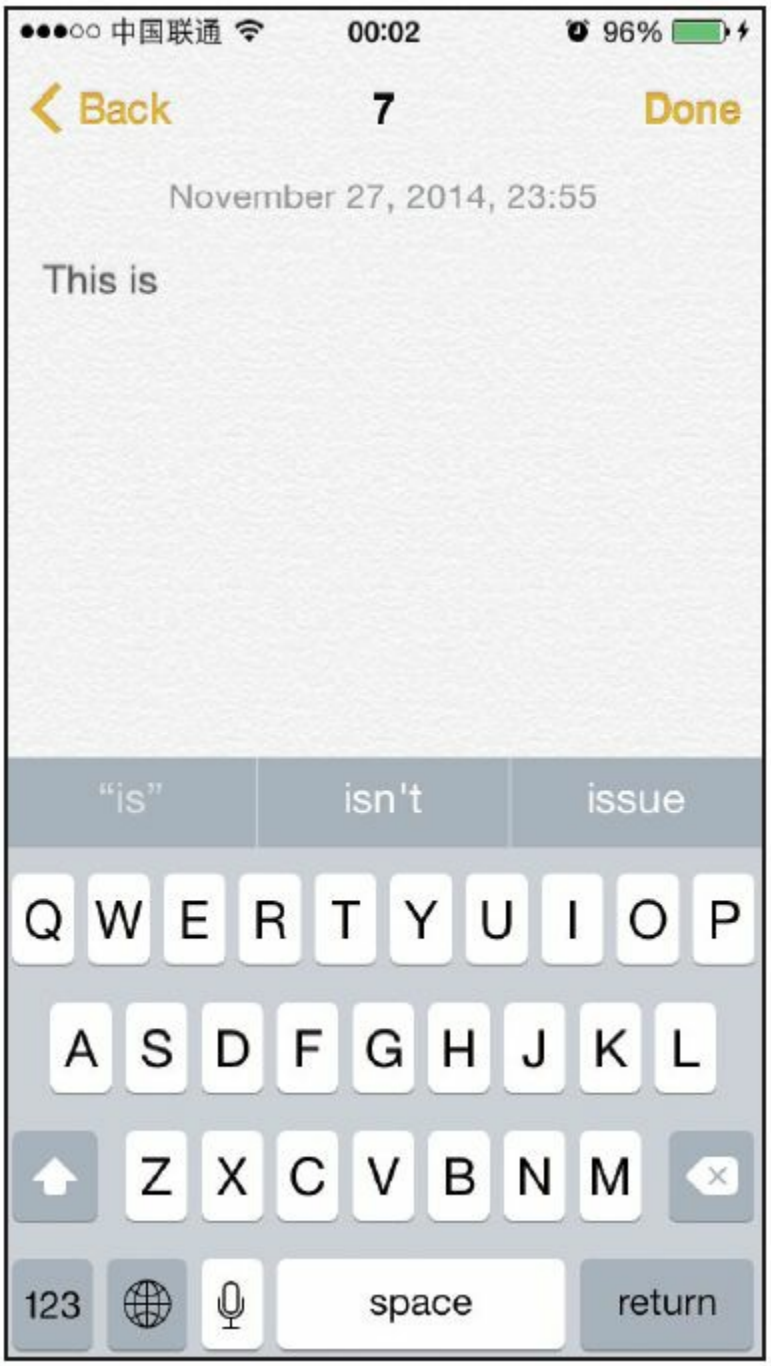


图7-13　Characount for Notes 8效果演示（3）

图7-14　Characount for Notes 8效果演示（4）

图7-15　Characount for Notes 8效果演示（5）

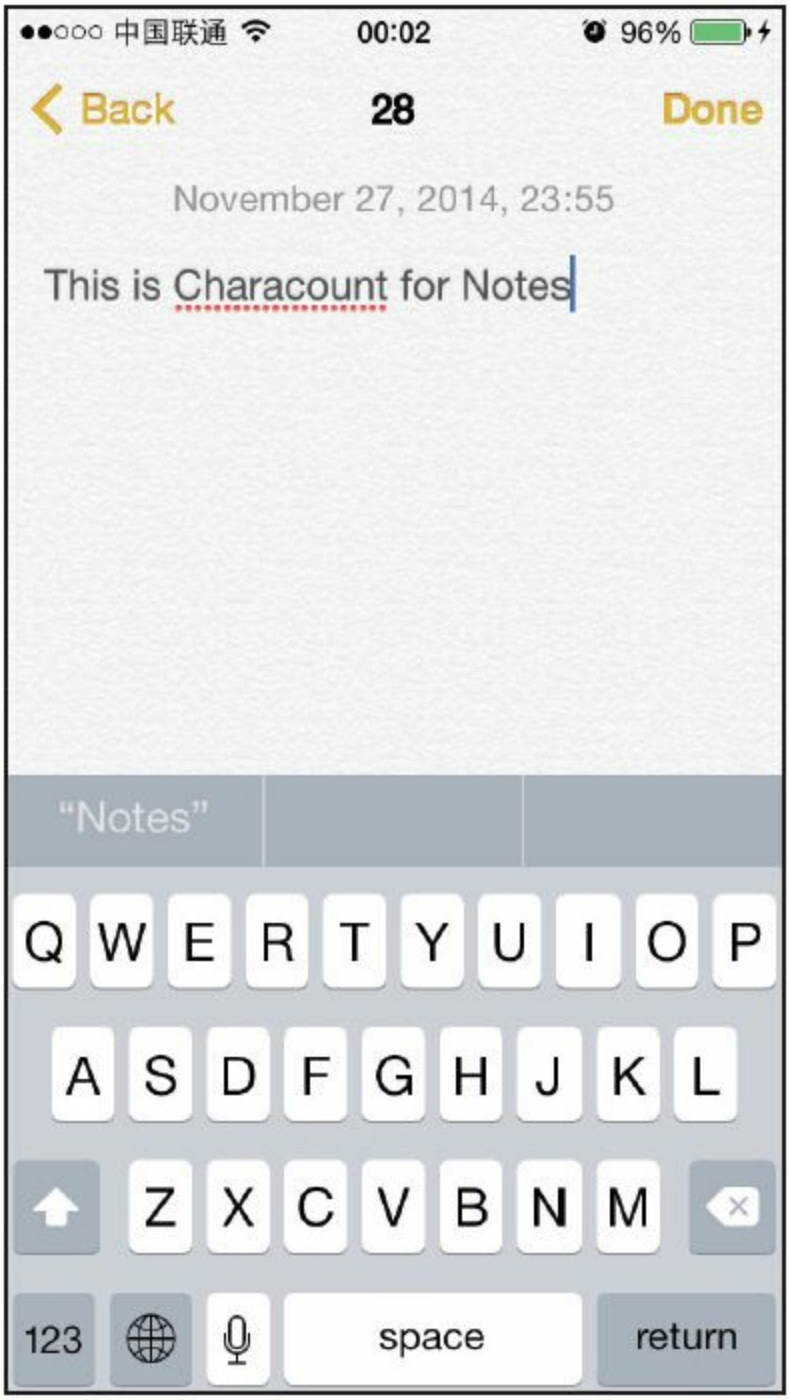


图7-16　Characount for Notes 8效果演示（6）

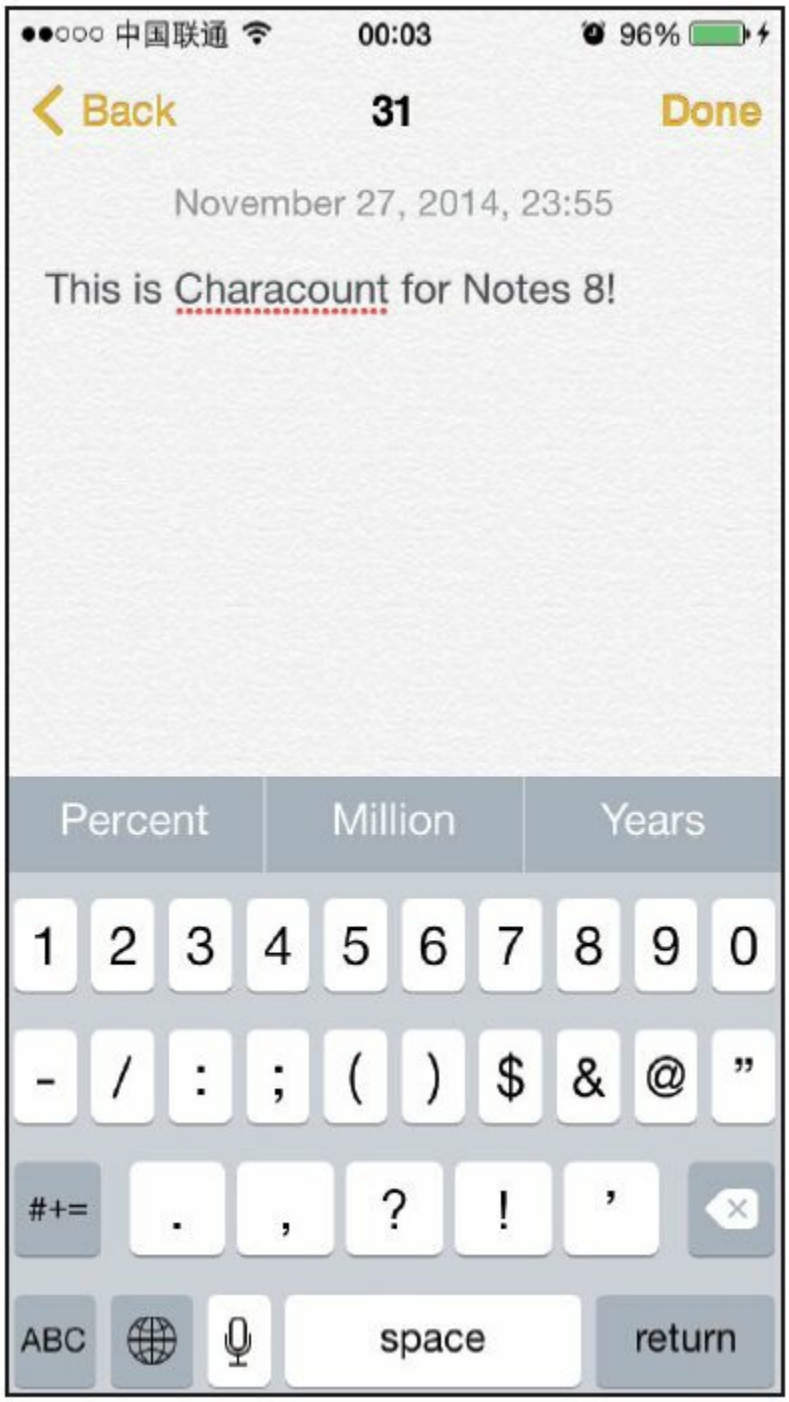


图7-17　Characount for Notes 8效果演示（7）

插件的表现与预期完全一致。

7.5 小结

作为iOS上的元老，Notes的功能虽然简单，它却是很多人日常生活中使用频度最高的App之一。本章编写的Characount for Notes 8功能比较简单，几乎可以脱离高级逆向工程工具来完成整个分析过程，希望初学者看起来不会太吃力。汇编级别的逆向工程学习起来难度较大，需要的时间投入较多，在对IDA和LLDB还比较生疏的档口，模仿本章的思路和方法，做一些简单的Objective-C逆向工程，既可以培养逆向工程的思路，又可以给自己营造小小的成就感，何乐而不为呢？

第8章　实战2：自动将指定电子邮件标记为已读

8.1　电子邮件

电子邮件是互联网时代最受欢迎的沟通渠道之一，很多人每天都会收发邮件。虽然AppStore上有很多优秀的邮件App可供选择，如Sparrow、Inbox等，但论及与iOS的整合度，终究没有原生邮件客户端（以下简称Mail）那样高，因此在笔者的日常使用中，Mail仍是首选。

在我们日常收到的邮件里，很大一部分是没有太多价值的“订阅”邮件——它们要么是活动通知，要么是软文广告——这些邮件都是我们在各种网站上无意间勾选“订阅”而收到的，如图8-1所示。

这类邮件很让人纠结。它们不算严格意义上的垃

圾邮件，却容易分散注意力；可是把它们标记为垃圾邮件吧，又有可能错过有价值的信息。那么该怎样处理这一类介于正常邮件和垃圾邮件之间的邮件呢？笔者有一个想法：给Mail加一个白名单功能，把常用联系人加入白名单；来自白名单以外的邮件全都自动标记为已读，这样就能在不遗漏信息的情况下突出重点，如图8-2所示。

把这个想法给具体化，就是本章的任务。

1）在Mail界面上的某个地方加个按钮，点击后出现可编辑的白名单列表，以便进行添加、删除白名单操作。

2）每次Mail的收件箱刷新后，自动把白名单以外的邮件标记为已读。

明确了本章的任务，接下来就一步步地实现目标。下面的操作在iPhone 5，iOS 8.1.1中完成。

●○○○○ 中国移动
23:46
Mailboxes
Inbox
Edit
招商银行
Wednesday
招行爱家季-Lifevc购物满268减20!(AD)
如果邮件无法正常显示，请点击这里<http://
www.cmbchina.com/Personal/Promotion/PrmotionI...
印象笔记团队
Wednesday
重要提示：请更新你的印象笔记 Mac 版
重要提示：请更新你的印象笔记 Mac 版 ## 我们注
意到你当前使用的印象笔记 Mac 版并不是最新版，请...
SegmentFault问答社区
Wednesday
12月热门活动推荐
SegmentFault・广播站<http://segmentfault.com> 访
问网站 ➔<http://segmentfault.com/> [SegmentFault...
Facebook
Wednesday
Iris Ning 更新了状态："I'm finally here! Can't wait to...
==========
查看帖子 https://www.facebook.com/n/?iris.ning%2...
阿里云计算
Wednesday
阿里云【1218】年度大促：￥12.18抢云服务器，满5...
[阿里云] <http://www.aliyun.com/?
spm=&msctype=email&mscmsgid=1424141210002...
Today on Twitter
Wednesday
Q. and A. About the C.I.A. Torture Report; More tha...
Today on Twitter Q. and A. About the C.I.A. Torture
Report https://twitter.com/ExploringBird/timelines/5...
Facebook
Wednesday
Sam，你有2 条新通知
Updated Just Now
65 Unread

图8-1　邮件

图8-2　把白名单以外邮件标记为已读

8.2 搭建tweak原型

Mail的初始界面如图8-3所示。

把白名单按钮添加在哪个位置比较直观呢？在图8-3所示的All Inboxes界面中，可以看到，其左下角是空缺的，或许可以把按钮添加在这里，试试看效果吧，如图8-4所示。

虽然添加的白名单按钮与右下方的“编写”按钮在方位上对齐了，但前者是文字，后者是图标，形式不统一，不够美观，可见，左下角不适合添加文字按钮。如果改成一个图标按钮呢？可能也会有问题，因为“白名单”没有约定俗成的图形表达方式，找不到一个比较有代表性的图标来表示白名单，所以用图标按钮表示白名单不够直观。直观、美观不可兼得，看来在这个界面上不大适合添加白名单按钮。点击左上角的“Mailboxes”，去上一级界面看看，如图8-5所示。

图8-5所示的界面左上和左下均是空着的，其中左下不适合添加白名单按钮，刚才已经讨论过了。把按钮添加在左上看看效果，如图8-6所示。

图8-3　Mail初始界面

图8-4　在左下角添加白名单按钮

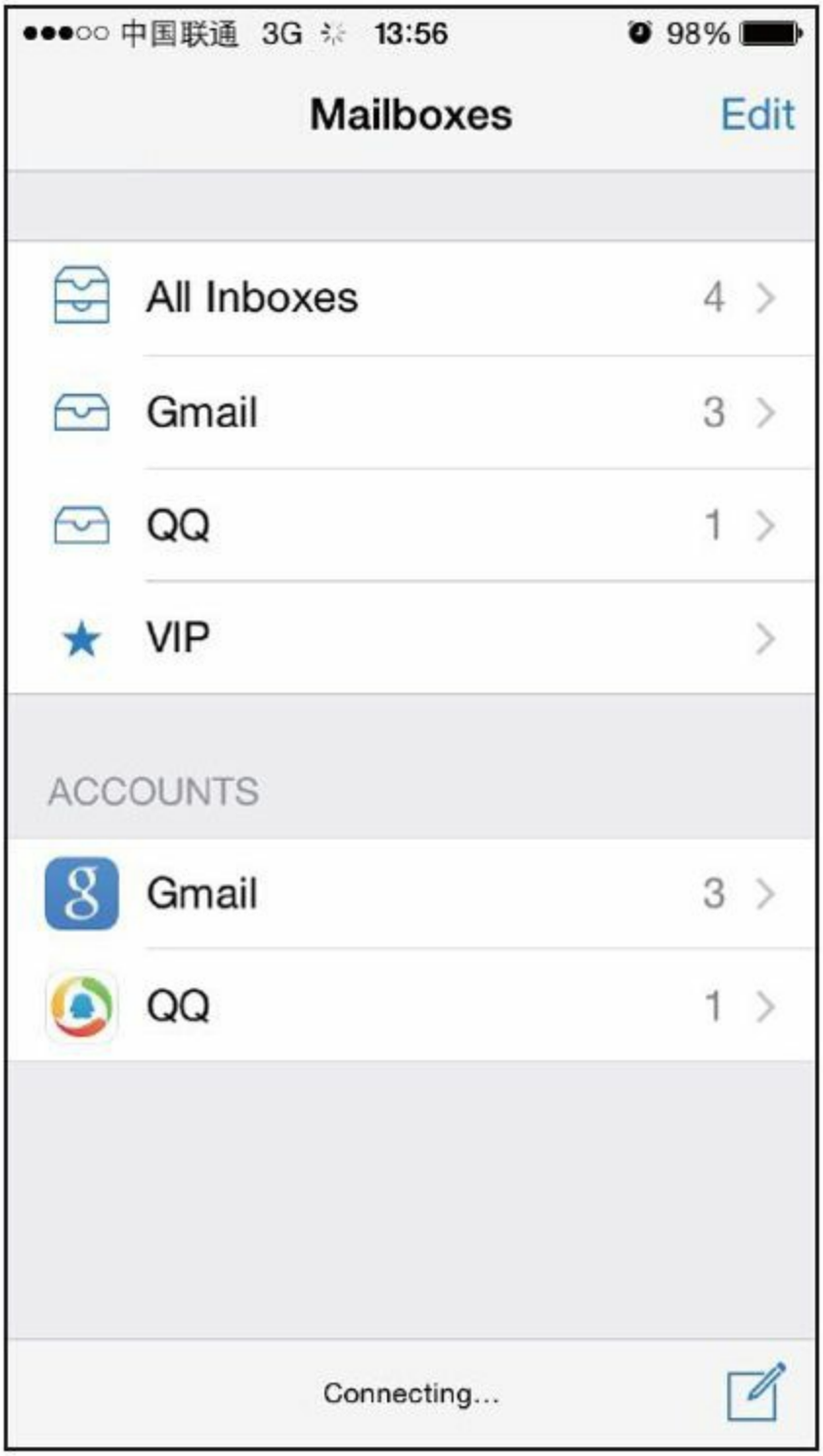


图8-5　Mailboxes界面

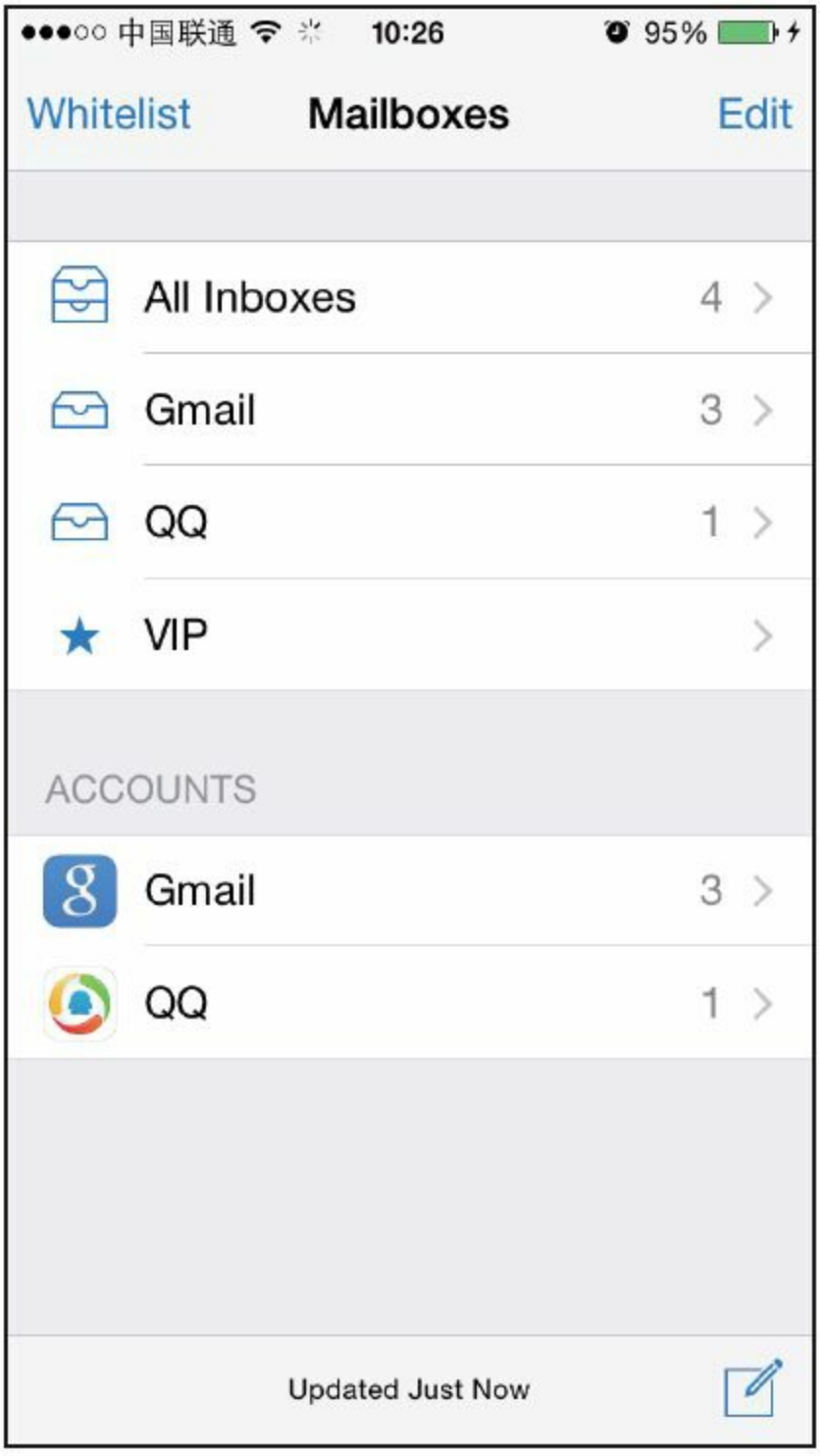


图8-6　在左上角添加白名单按钮

看起来效果还不错，就把白名单按钮放在这里

吧！要达到图8-6所示的效果，只需要找到Mailboxes界面的controller，然后通过[controller.navigationItem setLeftBarButtonItem:]来添加白名单按钮就可以了。通过V找C的过程前面已经重复过很多遍，是可行的。搞定了按钮，接下来就是对白名单工作逻辑的梳理了，可以分为以下3步：

1）拿到所有邮件；

2）提取出它们的地址；

3）根据白名单决定是否将它们标记为已读。

下面来一步步分析，一起来思考：

· 要如何才能拿到所有邮件呢？图8-3所示的界面可以下拉刷新收件箱，如图8-7所示。

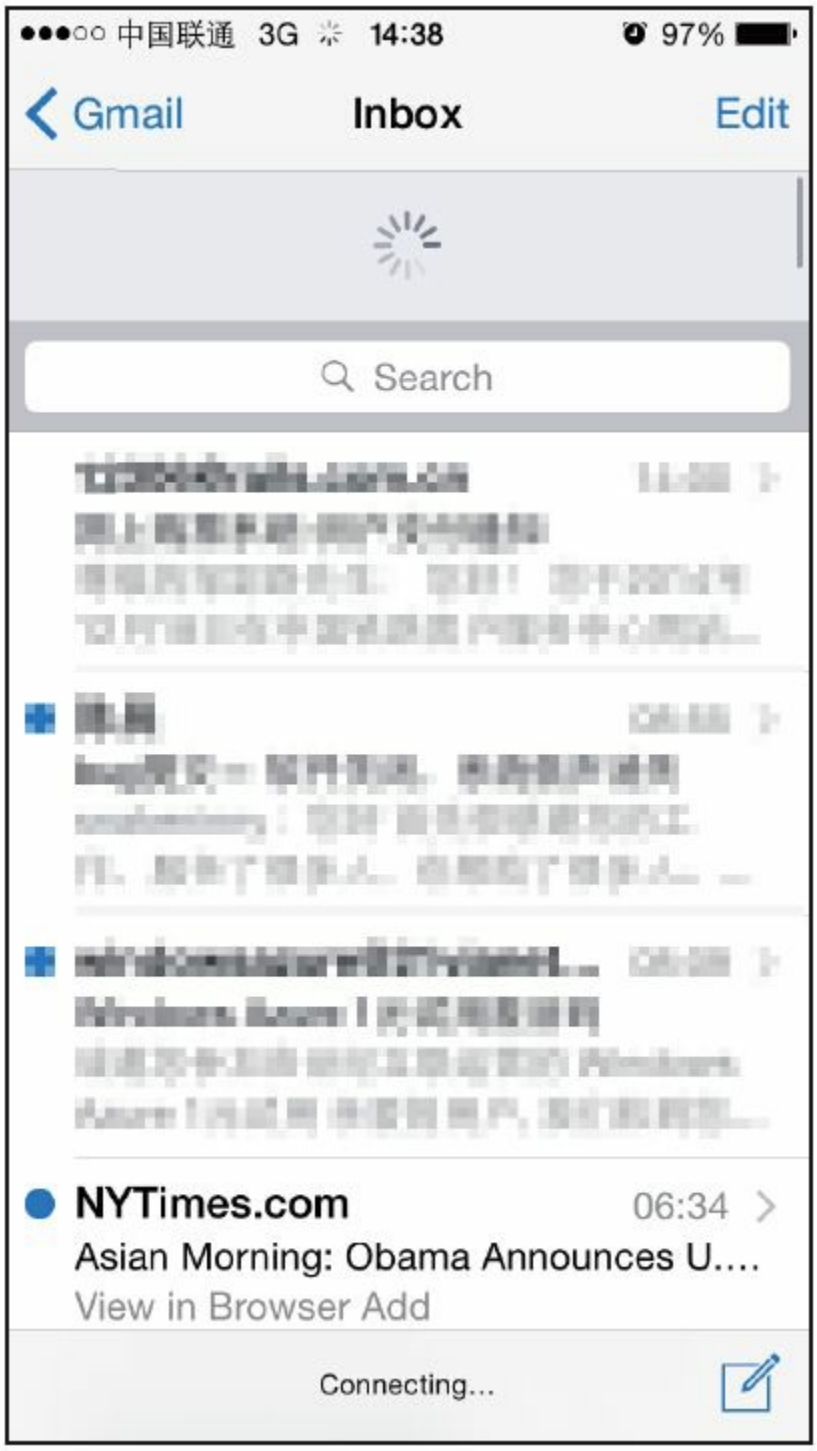


图8-7　下拉刷新

在刷新的过程中，Mail会去邮件服务器上获取最新邮件；刷新完成后，界面恢复到图8-3所示的样

子，此时收件箱里存放的就是所有邮件。如果能捕获“刷新完成”事件，然后读取收件箱，就可以拿到所有邮件了。因此，“拿到所有邮件”可以分为2步：一是捕获“刷新完成”事件；二是读取收件箱。其中，“刷新完成”的响应函数一般是定义在protocol里的，在分析class-dump头文件的时候，要留意各种protocol里有没有出现didRefresh、didUpdate、didReload之类名字中含有完成时态动词的函数，钩住（hook）它，然后寻找读取收件箱的方法，从而拿到所有邮件。

· 一封邮件就是一个对象，它一般会用一个类来表示，从这个类中可以提取出邮件的收件人、发件人、标题、内容和是否已读等信息。如果能拿到邮件对象，就可以一石二鸟，完成后两步操作。

看上去整体思路并不复杂，下面就来各个击破。

8.2.1 定位Mail的可执行文件并class-dump它

通过ps命令很容易就能定位到Mail的可执行文件“/Applications/MobileMail.app/MobileMail”。因为MobileMail是iOS原生系统App，没有加壳，所以不需要砸壳，直接class-dump即可，如下：

```
snakeninnys-MacBook:~ snakeninny$ class-dump -S -s -H
/Users/snakeninny/Code/iOSSystemBinaries/8.1.1_iPhone5/Mo
 -o
/Users/snakeninny/Code/iOSPrivateHeaders/8.1.1/MobileMail
```

程序执行后，共得到393个头文件，如图8-8所示。

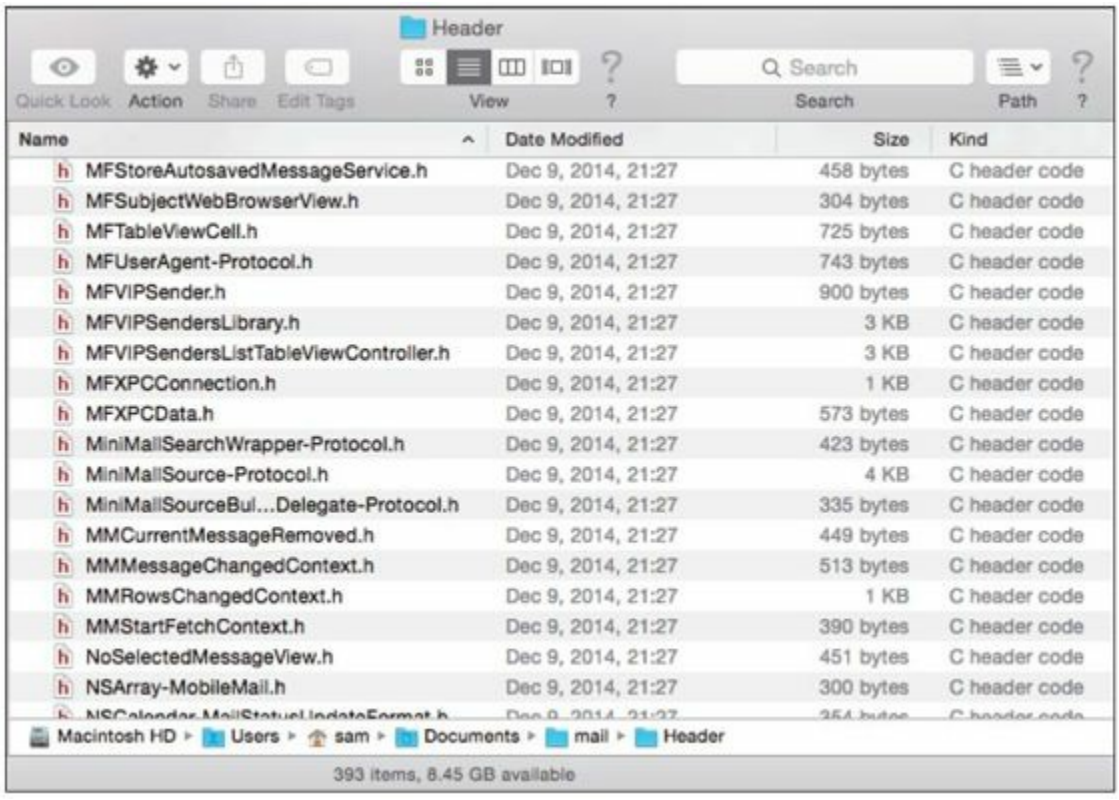


图8-8　class-dump头文件

8.2.2　把头文件导入Xcode

Xcode自带的查找功能和代码高亮显示能够较为美观整洁地展示大量头文件，如图8-9所示。

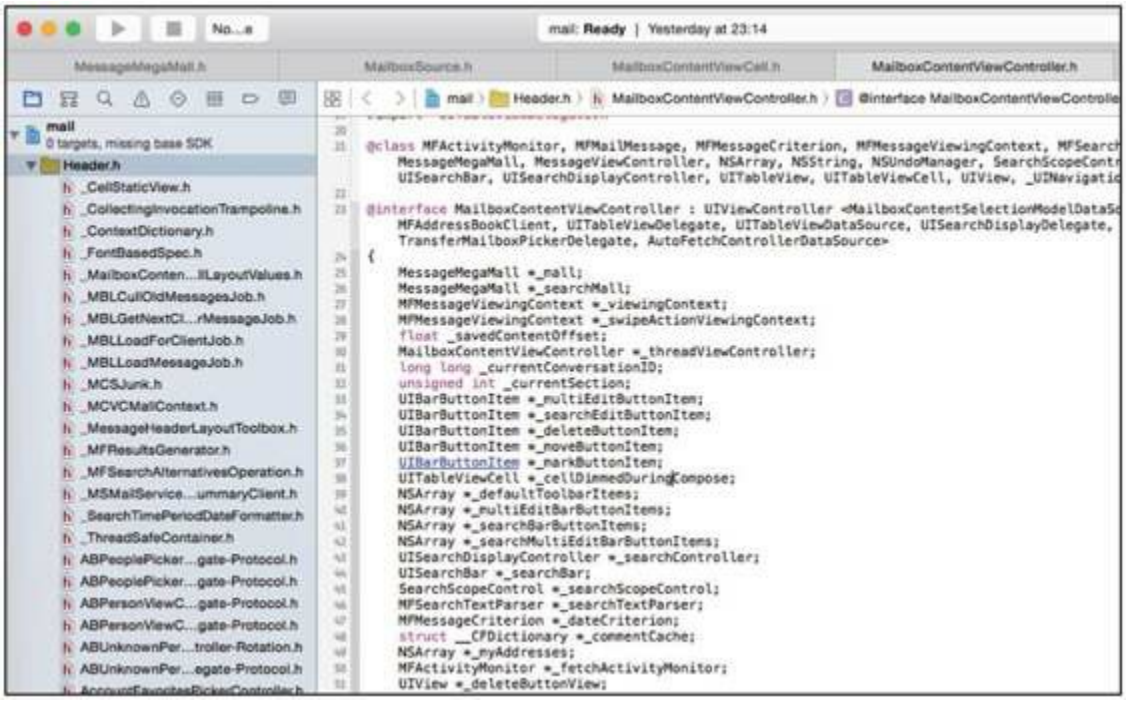


图8-9　把头文件导入Xcode

接下来，开始寻找线索，从App切入代码。

8.2.3　用Cycript找到Mailboxes界面的controller

首先用recursiveDescription打印出Mailboxes界面的UI布局，如下：

```
FunMaker-5:~ root# cycript -p MobileMail
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
@"<UIWindow: 0x156bffe0; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x156bd390>; layer =
<UIWindowLayer: 0x156c1be0>>
   | <UIView: 0x15611490; frame = (0 0; 320 568);
autoresize = W+H; gestureRecognizers = <NSArray:
0x15618e70>; layer = <CALayer: 0x15611420>>
   |    | <UIView: 0x15611210; frame = (0 0; 320 568);
layer = <CALayer: 0x15611280>>
   |    |    | <_MFActorItemView: 0x15614660; frame = (0
0; 320 568); layer = <CALayer: 0x15614840>>
   |    |    |    | <UIView: 0x156150f0; frame = (-0.5
-0.5; 321 569); alpha = 0; layer = <CALayer: 0x15615160>>
   |    |    |    | <_MFActorSnapshotView: 0x15614bb0;
baseClass = UISnapshotView; frame = (0 0; 320 568);
clipsToBounds = YES; hidden = YES; layer = <CALayer:
0x15614e00>>
   |    |    |    |    | <UIView: 0x15614f40; frame = (-1
-1; 322 570); layer = <CALayer: 0x15614fb0>>
   |    |    |    | <UILayoutContainerView: 0x1572ec40;
frame = (0 0; 320 568); clipsToBounds = YES; autoresize =
LM+W+RM+TM+H+BM; layer = <CALayer: 0x1572ecc0>>
   |    |    |    |    | <UIView: 0x1683d890; frame = (0
0; 320 0); layer = <CALayer: 0x16848140>>
   |    |    |    |    | <UILayoutContainerView:
0x157246b0; frame = (0 0; 320 568); clipsToBounds = YES;
gestureRecognizers = <NSArray: 0x156088e0>; layer =
<CALayer: 0x15724890>>
……
   |    |    |    |    |    |    |    |    |
<MailboxTableCell: 0x1572ad50; baseClass =
UITableViewCell; frame = (0 28; 320 44.5); autoresize =
W; layer = <CALayer: 0x168299f0>>
   |    |    |    |    |    |    |    |    |    |
<UITableViewCellContentView: 0x16829b70; frame = (0 0;
286 44); gestureRecognizers = <NSArray: 0x1682b060>;
layer = <CALayer: 0x16829be0>>
   |    |    |    |    |    |    |    |    |    |
```

```
| <UILabel: 0x1682b0a0; frame = (55 12; 84.5 20.5); text
= 'All Inboxes'; userInteractionEnabled = NO; layer =
<_UILabelLayer: 0x1682b160>>
……
```

其中，最下方的这个UILabel上的文字是“All Inboxes”，其对应的MailboxTableCell自然就是图8-5中最上面的一个cell了。连续调用nextResponder，找出这个界面的controller，如下：

```
cy# [#0x1572ad50 nextResponder]
#"<UITableViewWrapperView: 0x1572fe60; frame = (0 0; 320
568); gestureRecognizers = <NSArray: 0x15730370>; layer =
<CALayer: 0x157301a0>; contentOffset: {0, 0};
contentSize: {320, 568}>"
cy# [#0x1572fe60 nextResponder]
#"<UITableView: 0x1585a000; frame = (0 0; 320 568);
clipsToBounds = YES; autoresize = W+H; gestureRecognizers
= <NSArray: 0x1572fa20>; layer = <CALayer: 0x1572f540>;
contentOffset: {0, -64}; contentSize: {320, 371}>"
cy# [#0x1585a000 nextResponder]
#"<MailboxPickerController: 0x156e9260>"
```

很轻松地拿到了MailboxPickerController。试试看用它能不能添加一个leftBarButtonItem，如下：

```
cy# #0x156e9260.navigationItem.leftBarButtonItem =
#0x156e9260.navigationItem.rightBarButtonItem
#"<UIBarButtonItem: 0x15729f00>"
```

效果如图8-10所示。

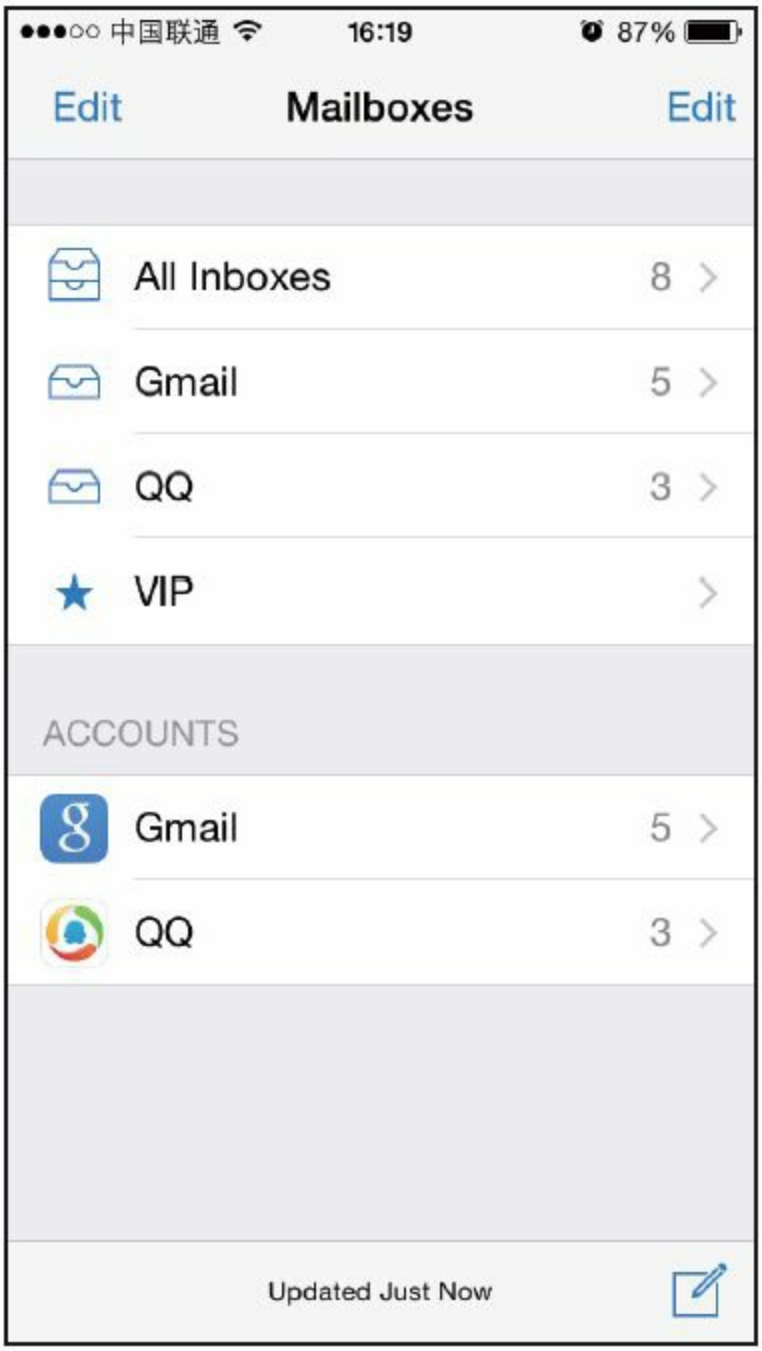


图8-10　setLeftBarButtonItem:的效果

没有问题，MailboxPickerController就是Mailboxes

界面的controller，可以通过它添加白名单按钮。

8.2.4 用Reveal和Cycript找到All Inboxes界面的delegate

搞定了白名单按钮，就要开始梳理白名单的工作逻辑了，先看看如何捕获“刷新完成”事件。因为“刷新完成”是直观表现在All Inboxes界面上的，所以“刷新完成”的响应函数很可能定义在这个界面的delegate中。转战到图8-3所示的All Inboxes界面，这里不再重复8.2.3节的方法，而是先通过Reveal定位这个界面的cell，再用Cycript找出它所在的UITableView，从而得知其delegate。

下面就用Reveal查看Mail，很容易就可以定位到最上方的那个cell，如图8-11所示。

图8-11　用Reveal查看Mail的UI布局

MailboxContentViewCell就是显示邮件发件人、标题和摘要的cell。接着用Cycript找出它所在的UITableView：因为我们知道当前界面一定存在MailboxContentViewCell对象，所以可以尝试通过choose命令获取这些对象，而不用劳烦recursiveDescription她老人家了，如下：

```
FunMaker-5:~ root# cycript -p MobileMail
cy# choose(MailboxContentViewCell)
[#"<MailboxContentViewCell: 0x161f4000> cellContent",#"
<MailboxContentViewCell: 0x1621c400> cellContent",#"
<MailboxContentViewCell: 0x1621d000> cellContent",#"
<MailboxContentViewCell: 0x16234800> cellContent",#"
<MailboxContentViewCell: 0x1623ee00> cellContent",#"
<MailboxContentViewCell: 0x1623f200> cellContent",#"
<MailboxContentViewCell: 0x159c2c00> cellContent"]
```

choose命令返回了一个由MailboxContentViewCell对象组成的NSArray，从中随便挑选一个MailboxContentViewCell对象，对其连续调用nextResponder，如下：

```
cy# [choose(MailboxContentViewCell)[0] nextResponder]
#"<UITableViewWrapperView: 0x15660b80; frame = (0 0; 320
612); gestureRecognizers = <NSArray: 0x16855170>; layer =
<CALayer: 0x16888f20>; contentOffset: {0, 0};
contentSize: {320, 612}>"
cy# [#0x15660b80 nextResponder]
#"<MFMailboxTableView: 0x16095000; baseClass =
UITableView; frame = (0 0; 320 568); clipsToBounds = YES;
autoresize = W+H; gestureRecognizers = <NSArray:
0x15607850>; layer = <CALayer: 0x16838210>;
contentOffset: {0, -64}; contentSize: {320, 52364}>"
```

它所在的UITableView是一个MFMailboxTableView对象，看看它的delegate是什么，如下：

```
cy# [#0x16095000 delegate]
#"<MailboxContentViewController: 0x16106000>"
```

它的delegate是MailboxContentViewController。继续调用nextResponder，看看它的controller又是什么，

如下：

```
cy# [#0x16095000 nextResponder]
#"<MailboxContentViewController: 0x16106000>"
```

也就是说，MFMailboxTableView的controller和delegate均是MailboxContentView-Controller。简单验证一下controller的正确性，如下：

```
cy# [#0x16106000 setTitle:@"iOSRE"]
```

效果如图8-12所示。

图8-12　setTitle:的效果

这个结果说明上述一系列推导没有问题，MailboxContent-ViewController中很可能既含有“刷新完成”的响应函数，又能找到读取收件箱的蛛丝马

迹，接下来就把焦点放在它身上了。

8.2.5 在MailboxContentViewController中定位“刷新完成”的响应函数

与第7章一样，先来看看MailboxContentViewController实现了哪些protocol，能不能在其中找到可疑的响应函数，如下：

```
@interface MailboxContentViewController :
UIViewController <MailboxContentSelectionModelDataSource,
MFSearchTextParserDelegate, MessageMegaMallObserver,
MFAddressBookClient, MFMailboxTableViewDelegate,
UIPopoverPresentationControllerDelegate,
UITableViewDelegate, UITableViewDataSource,
UISearchDisplayDelegate, UISearchBarDelegate,
TransferMailboxPickerDelegate,
AutoFetchControllerDataSource>
```

先从名字上做一次简单的排查，MFSearchTextParserDelegate、MFAddressBookClient、UIPopoverPresentationControllerDelegate、UITableViewDelegate、UITableViewDataSource、

UISearchDisplayDelegate、UISearchBarDelegate看起来跟“刷新完成”没什么关系，可以直接排除。剩下的MailboxContentSelectionModelDataSource、MessageMegaMallObserver、MFMailboxTableViewDelegate、TransferMailboxPickerDelegate和AutoFetchControllerDataSource还不好说，那就挨个过一遍。先看MailboxContentSelectionModelDataSource.h，内容如下：

```
@protocol MailboxContentSelectionModelDataSource
<NSObject>
- (BOOL)selectionModel:(id)arg1
deleteMovesToTrashForTableIndexPath:(id)arg2;
- (void)selectionModel:(id)arg1
getConversationStateAtTableIndexPath:(id)arg2 hasUnread:
(char *)arg3 hasUnflagged:(char *)arg4;
- (void)selectionModel:(id)arg1 getSourceStateHasUnread:
(char *)arg2 hasUnflagged:(char *)arg3;
- (id)selectionModel:(id)arg1 indexPathForMessageInfo:
(id)arg2;
- (id)selectionModel:(id)arg1
messageInfosAtTableIndexPath:(id)arg2;
- (id)selectionModel:(id)arg1 messagesForMessageInfos:
(id)arg2;
- (BOOL)selectionModel:(id)arg1
shouldArchiveByDefaultForTableIndexPath:(id)arg2;
- (id)selectionModel:(id)arg1 sourceForMessageInfo:
(id)arg2;
```

```
- (BOOL)selectionModel:(id)arg1
supportsArchivingForTableIndexPath:(id)arg2;
- (id)sourcesForSelectionModel:(id)arg1;
@end
```

这个protocol的作用看上去是读取数据源，跟“刷新数据源”没太大关系。接着看MessageMegaMallObserver.h，内容如下：

```
@protocol MessageMegaMallObserver <NSObject>
- (void)megaMallCurrentMessageRemoved:(id)arg1;
- (void)megaMallDidFinishSearch:(id)arg1;
- (void)megaMallDidLoadMessages:(id)arg1;
- (void)megaMallFinishedFetch:(id)arg1;
- (void)megaMallGrowingMailboxesChanged:(id)arg1;
- (void)megaMallMessageCountChanged:(id)arg1;
- (void)megaMallMessagesAtIndexesChanged:(id)arg1;
- (void)megaMallStartFetch:(id)arg1;
@end
```

这个类中的不少函数名含有完成时态动词，同时，从“LoadMessages”、“FinishedFetch”、“MessageCountC函数的名字上来看，它可能会在刷新完成的前后得到调用。接下来用LLDB在这3个函数的开头部分下断点，然后下拉刷新收件箱，看看它们的调用情况。首先用LLDB附加MobileMail，查看其ASLR偏移，如

下：

```
(lldb) image list -o -f
[  0]
0x000b2000/private/var/db/stash/_.lnBgU8/Applications/Mob

[   1]
0x003b7000/Library/MobileSubstrate/MobileSubstrate.dylib(

[   2]
0x090d1000/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/usr/lib/libarchive.2.dylib
[  3] 0x090c3000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1.1
(12B435)/Symbols/System/Library/Frameworks/CloudKit.frame

……
```

可以看到，ASLR偏移是0x000b2000。然后把MobileMail拖进IDA，待初始分析完成后，查看[MailboxContentViewController megaMallDidLoadMessages:]、[MailboxContentViewControllermegaMallFinishedFetch:]和[MailboxContentViewController megaMallMessageCountChanged:]的基地址，如图8-13、图8-14和图8-15所示。

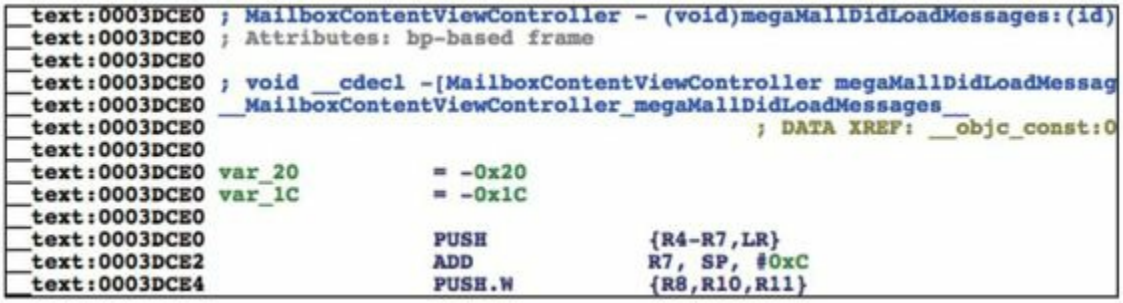


图8-13 [MailboxContentViewController megaMallDidLoadMessages:]

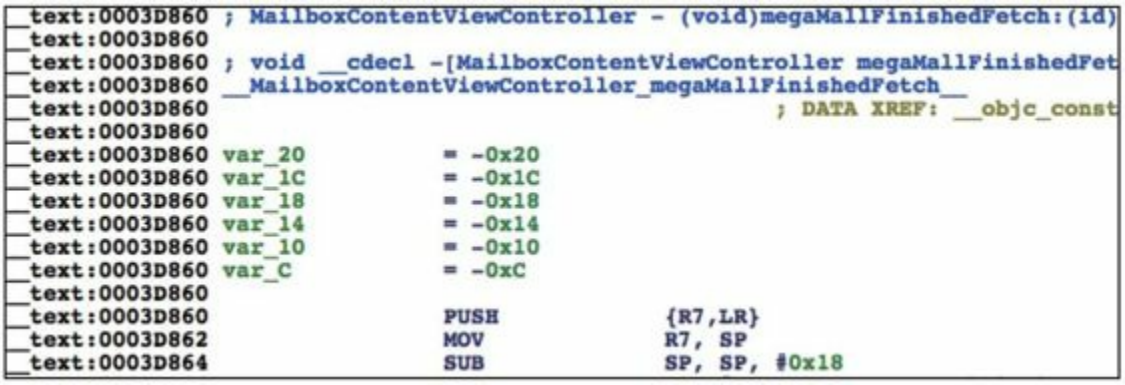


图8-14 [MailboxContentViewController megaMallFinishedFetch:]

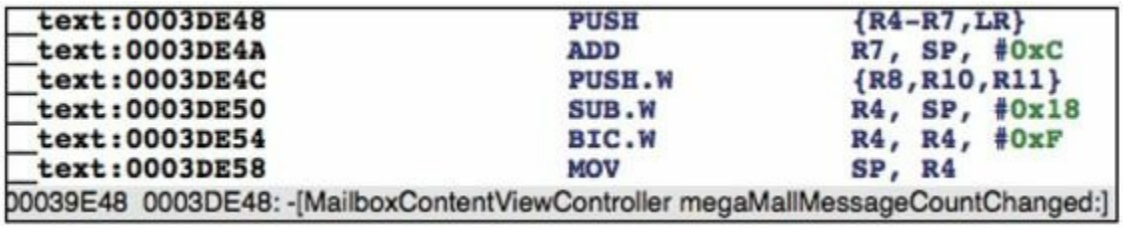


图8-15 [MailboxContentViewController megaMallMessageCountChanged:]

它们的基地址分别是0x3dce0、0x3d860和

0x3de48。用LLDB在这些地址上下断点，然后下拉刷新，触发断点，如下：

```
(lldb) br s -a '0x000b2000+0x3dce0'
Breakpoint 1: where =
MobileMail`___lldb_unnamed_function992$$MobileMail,
addrss = 0x000efce0
(lldb) br s -a '0x000b2000+0x3d860'
Breakpoint 2: where =
MobileMail`___lldb_unnamed_function987$$MobileMail,
addrss = 0x000ef860
(lldb) br s -a '0x000b2000+0x3de48'
Breakpoint 3: where =
MobileMail`___lldb_unnamed_function993$$MobileMail,
addrss = 0x000efe48
```

可能有读者在这里会碰到跟笔者相同的情况：三个断点一个也没有触发。从事过网络编程的朋友可能会猜到原因——为了减轻邮件服务器的负担，节省iOS流量，并不是每次下拉刷新都会去服务器取数据。如果刷新时间间隔不长，收件箱的数据源就会是本地缓存，不会调用MessageMegaMallObserver中的方法。为了验证这个猜测，我们往自己的邮箱发一封邮件，然后下拉刷新，看看断点触发情况，如下：

```
Process 73130 stopped
* thread #44: tid = 0x14c10, 0x000ef860
```

```
MobileMail`___lldb_unnamed_function987$$ MobileMail, stop
reason = breakpoint 2.1
    frame #0: 0x000ef860
MobileMail`___lldb_unnamed_function987$$MobileMail
MobileMail`___lldb_unnamed_function987$$MobileMail:
-> 0xef860:  push   {r7, lr}
   0xef862:  mov    r7, sp
   0xef864:  sub    sp, #24
   0xef866:  movw   r1, #44962
(lldb) c
Process 73130 resuming
Process 73130 stopped
* thread #44: tid = 0x14c10, 0x000ef860
MobileMail`___lldb_unnamed_function987$$ MobileMail, stop
reason = breakpoint 2.1
    frame #0: 0x000ef860
MobileMail`___lldb_unnamed_function987$$MobileMail
MobileMail`___lldb_unnamed_function987$$MobileMail:
-> 0xef860:  push   {r7, lr}
   0xef862:  mov    r7, sp
   0xef864:  sub    sp, #24
   0xef866:  movw   r1, #44962
(lldb) c
Process 73130 resuming
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$ MobileMail,
queue = 'MessageMiniMall.0x157c2d90, stop reason =
breakpoint 3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
   0xefe4c:  push.w {r8, r10, r11}
   0xefe50:  sub.w  r4, sp, #24
(lldb)
Process 73130 resuming
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$ MobileMail,
queue = 'MessageMiniMall.0x157c2d90, stop reason =
breakpoint 3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
   0xefe4c:  push.w {r8, r10, r11}
```

```
   0xefe50:  sub.w  r4, sp, #24
(lldb)
Process 73130 resuming
Process 73130 stopped
* thread #44: tid = 0x14c10, 0x000ef860
MobileMail`___lldb_unnamed_function987$$ MobileMail, stop
reason = breakpoint 2.1
    frame #0: 0x000ef860
MobileMail`___lldb_unnamed_function987$$MobileMail
MobileMail`___lldb_unnamed_function987$$MobileMail:
-> 0xef860:  push   {r7, lr}
   0xef862:  mov    r7, sp
   0xef864:  sub    sp, #24
   0xef866:  movw   r1, #44962
(lldb) c
Process 73130 resuming
```

果不其然，megaMallFinishedFetch:和megaMallMessageCountChanged:被交替调用。从名字上来看，一封邮件就是一个message，megaMallFinishedFetch:应该会在iOS成功地从服务器取回邮件之后得到调用，而megaMallMessageCountChanged:应该会在邮件数量发生变动，即收邮件和删邮件时得到调用，两者自然都会在“刷新完成”时得到调用，都可以看作“刷新完成”的响应函数。两者随便选其一，这里选择megaMallMessageCountChanged:，接下来的任务是寻找拿到所有邮件的方法。

8.2.6 从MessageMegaMall中拿到所有邮件

还记得第7章中说过的“协议方法被调用，一般是因为方法名中提到的那个事件发生了；而那件事发生的对象，一般是协议方法的参数”吗？删掉前2个断点，保留第3个，也就是megaMallMessageCountChanged:上的断点，看看它的参数是什么，如下：

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$ MobileMail,
queue = 'MessageMiniMall.0x157c2d90, stop reason =
breakpoint 3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
   0xefe4c:  push.w {r8, r10, r11}
   0xefe50:  sub.w  r4, sp, #24
(lldb) po $r2
NSConcreteNotification 0x157e8af0 {name =
MegaMallMessageCountChanged; object = <MessageMegaMall:
0x1576c320>; userInfo = {
    "added-message-infos" =     (
        "<MFMessageInfo: 0x157c86d0> uid=1185,
conversation=2777228998582613276"
    );
    destination = "{(\n)}";
    inserted = "{(\n    <NSIndexPath: 0x157e8ac0> {length
= 2, path = 0 - 0}\n)}";
    relocated = "{(\n)}";
    updated = "{(\n)}";
```

```
}}
```

可以看到，参数是一个NSConcreteNotification对象。查看其头文件，可知它继承自NSNotification。它的name是MegaMallMessageCountChanged，object是一个MessageMegaMall对象，userInfo是一些改动信息。“MegaMall”这个名字很值得玩味，“大型购物中心”，看似与邮件毫不相关，却又与“Message”寸步不离，与8.2.4节分析的MessageMegaMallObserver遥相呼应，疑似为一个存储“Message”的类。打开MessageMegaMall.h，看看它的内容，如下：

```
@interface MessageMegaMall : NSObject
<MessageMiniMallObserver, Message SelectionDataSource>
……
- (id)copyAllMessages;
@property (retain, nonatomic) MFMailMessage
*currentMessage;
- (void)loadOlderMessages;
- (unsigned int)localMessageCount;
- (unsigned int)messageCount;
- (void)markAllMessagesAsNotViewed;
- (void)markAllMessagesAsViewed;
- (void)markMessagesAsNotViewed:(id)arg1;
- (void)markMessagesAsViewed:(id)arg1;
……
@end
```

线索有些明朗了：复制所有邮件、当前邮件、读取早期邮件、本地邮件计数、邮件计数、标为已读……MessageMegaMall应该就是一个管理所有邮件对象的M，它被苹果形象地比喻为“大型购物中心”。那么到底能不能通过copyAllMessages拿到所有的邮件呢？在LLDB里试一下，如下：

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$ MobileMail,
queue = 'MessageMiniMall.0x157c2d90, stop reason =
breakpoint 3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
   0xefe4c:  push.w {r8, r10, r11}
   0xefe50:  sub.w  r4, sp, #24
(lldb) po [[$r2 object] copyAllMessages]
{(
    <MFLibraryMessage 0x15612030: library id 89, remote
id 13020, 2014-11-25 20:32:16 +0000, 'Cydia/APT(A):
LowPowerBanner (1.4.5)'>,
    <MFLibraryMessage 0x1572ef10: library id 604, remote
id 12718, 2014-10-01 21:34:28 +0000, 'Asian Morning: Told
to End Protests, Organizers in Hong Kong Vow to Expand
Them'>,
    <MFLibraryMessage 0x168bd170: library id 906, remote
id 13142, 2014-12-17 22:34:30 +0000, 'Asian Morning:
Obama Announces U.S. and Cuba Will Resume Relations'>,
……
)}
(lldb) p (int)[[[$r2 object] copyAllMessages] count]
(int) $7 = 580
(lldb) p (int)[[$r2 object] localMessageCount]
(int) $8 = 580
```

```
(lldb) p (int)[[$r2 object] messageCount]
(int) $0 = 553
(lldb) po [[[$r2 object] copyAllMessages] class]
__NSSetM
```

copyAllMessages返回了一个NSSet，其中含有580个MFLibraryMessage对象，MFLibraryMessage对象中含有邮件摘要信息，且NSSet中对象的个数与localMessageCount的值相同。这个结果很好理解：为了节省带宽流量和本地空间，iOS没有必要一次性下载邮件服务器上的所有邮件，因此会先存储个百十来封，用户如果要看更多的邮件，再去服务器获取（即loadOlderMessages）。因此，copyAllMessages就是拿到所有邮件的方法，第二目标达成！同时，留意[MessageMegaMall markMessagesAsViewed:]函数，如果不出意外，它就是把邮件标记为已读的方法，而参数则很有可能是一个含有MFLibraryMessage对象的NSArray或NSSet。到底是不是这样呢？我们马上就会验证。

8.2.7 从MFLibraryMessage中提取发件人地址，用MessageMegaMall标记已读

从8.2.4节的分析可知，一封邮件就是一个MFLibraryMessage对象，它的description里显示的正是邮件摘要。不过，在MobileMail的头文件中是找不到它的身影的，想必你也能猜到大致原因——MFLibraryMessage来自一个外部dylib。在iOS 8的所有class-dump头文件里搜索MFLibraryMessage，发现它来自Messages私有库，如图8-16所示。

看看MFLibraryMessage.h的内容，如下：

```
@interface MFLibraryMessage : MFMailMessage
……
- (id)copyMessageInfo;
……
- (void)markAsNotViewed;
- (void)markAsViewed;
- (id)account;
……
- (unsigned long long)uniqueRemoteId;
- (unsigned long)uid;
- (unsigned int)hash;
- (id)remoteID;
- (void)_updateUID;
- (unsigned int)messageSize;
- (id)originalMailboxURL;
- (unsigned int)originalMailboxID;
```

```
- (unsigned int)mailboxID;
- (unsigned int)libraryID;
- (id)persistentID;
- (id)messageID;
@end
```

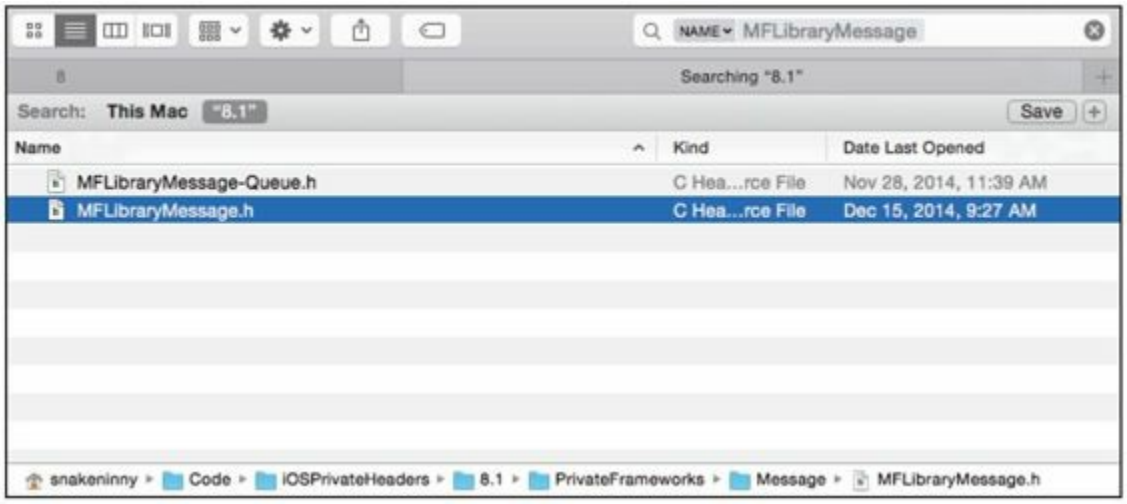


图8-16　定位MFLibraryMessage

MFLibraryMessage.h里充斥着各种ID，但没有我们要找的发件人地址等信息。这个结果不正常：我们已经在MFLibraryMessage的description里看到了邮件摘要信息，却没有在MFLibraryMessage.h里找到读取摘要信息的方法，说明分析过程有遗漏。重新审视MFLibraryMessage.h，这时，copyMessageInfo进入了我们的视线，看看它返回的“邮件信息”里会有什么数据，如下：

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$ MobileMail,
queue = 'MessageMiniMall.0x157c2d90, stop reason =
breakpoint 3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
   0xefe4c:  push.w {r8, r10, r11}
   0xefe50:  sub.w  r4, sp, #24
 (lldb) po [[[[$r2 object] copyAllMessages] anyObject]
copyMessageInfo]
<MFMessageInfo: 0x157c8040> uid=89,
conversation=594030790676622907
```

从中拿到了一个8.2.5节出现过的MFMessageInfo对象，马上去看看MFMessageInfo.h里有没有邮件摘要信息，如下：

```
@interface MFMessageInfo : NSObject
{
    unsigned int _flagged:1;
    unsigned int _read:1;
    unsigned int _deleted:1;
    unsigned int _uidIsLibraryID:1;
    unsigned int _hasAttachments:1;
    unsigned int _isVIP:1;
    unsigned int _uid;
    unsigned int _dateReceivedInterval;
    unsigned int _dateSentInterval;
    unsigned int _mailboxID;
    long long _conversationHash;
    long long _generationNumber;
}
+ (long long)newGenerationNumber;
@property(readonly, nonatomic) long long
generationNumber; // @synthesize
generationNumber=_generationNumber;
```

```
@property(nonatomic) unsigned int mailboxID; //
@synthesize mailboxID=_mailboxID;
@property(nonatomic) long long conversationHash; //
@synthesize conversationHash= _conversationHash;
@property(nonatomic) unsigned int dateSentInterval; //
@synthesize dateSentInterval=_ dateSentInterval;
@property(nonatomic) unsigned int dateReceivedInterval;
// @synthesize dateReceivedInterval=_
dateReceivedInterval;
@property(nonatomic) unsigned int uid; // @synthesize
uid=_uid;
- (id)description;
- (unsigned int)hash;
- (BOOL)isEqual:(id)arg1;
- (int)generationCompare:(id)arg1;
- (id)initWithUid:(unsigned int)arg1 mailboxID:(unsigned
int)arg2 dateReceivedInterval:(unsigned int)arg3
dateSentInterval:(unsigned int)arg4 conversationHash:
(long long)arg5 read:(BOOL)arg6 knownToHaveAttachments:
(BOOL)arg7 flagged:(BOOL)arg8 isVIP:(BOOL)arg9;
- (id)init;
@property(nonatomic) BOOL isVIP;
@property(nonatomic, getter=isKnownToHaveAttachments)
BOOL knownToHave Attachments;
@property(nonatomic) BOOL uidIsLibraryID;
@property(nonatomic) BOOL deleted;
@property(nonatomic) BOOL flagged;
@property(nonatomic) BOOL read;
@end
```

MFMessageInfo中含有已读信息，但不含有邮件摘要信息，说明分析仍不够严密。再回过头仔细观察MFLibraryMessage.h，发现它继承自MFMailMessage，从名字上看，MailMessage用来代表邮件显然比LibraryMessage更贴切。打开MFMailMessage.h，看看它的内容，如下：

```
@interface MFMailMessage : MFMessage
……
- (BOOL)shouldSetSummary;
- (void)setSummary:(id)arg1;
- (void)setSubject:(id)arg1 to:(id)arg2 cc:(id)arg3 bcc:
(id)arg4 sender:(id)arg5 dateReceived:(double)arg6
dateSent:(double)arg7 messageIDHash:(long long)arg8
conversationIDHash:(long long)arg9 summary:(id)arg10
withOptions:(unsigned int)arg11;
- (id)subject;
@end
```

summary、subject、sender、cc、bcc等邮件常用词汇出现在我们面前，但除了subject，MFMailMessage.h中只出现了setter，而不见getter。还记得刚才我们的注意力是怎么从MFLibraryMessage.h转移到MFMailMessage.h上的吗？想必你一定也注意到了MFMailMessage的父类MFMessage。在查看它的头文件前，先用LLDB看看[MFMailMessage subject]的返回，验证一下到目前为止的分析，如下：

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue
= 'MessageMiniMall.0x157c2d90, stop reason = breakpoint
3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
```

```
    0xefe4c:  push.w {r8, r10, r11}
    0xefe50:  sub.w  r4, sp, #24
(lldb) po [[[[$r2 object] copyAllMessages] anyObject]
subject]
Asian Morning: Told to End Protests, Organizers in Hong
Kong Vow to Expand Them
```

可以看到，[MFMailMessage subject]返回的正是邮件的标题。打开MFMessage.h（注意，MFMessage是MIME.framework里的类），看看它的内容，如下：

```
@interface MFMessage : NSObject <NSCopying>
……
- (id)headerData;
- (id)bodyData;
- (id)summary;
- (id)bccIfCached;
- (id)bcc;
- (id)ccIfCached;
- (id)cc;
- (id)toIfCached;
- (id)to;
- (id)firstSender;
- (id)sendersIfCached;
- (id)senders;
- (id)dateSent;
- (id)subject;
- (id)messageData;
- (id)messageBody;
- (id)headers;
……
@end
```

其中，邮件的收件人、发件人、标题、内容等信

息一应俱全。用LLDB简单看看它们的返回值，如下：

```
Process 73130 stopped
* thread #1: tid = 0x11daa, 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail, queue
= 'MessageMiniMall.0x157c2d90, stop reason = breakpoint
3.1
    frame #0: 0x000efe48
MobileMail`___lldb_unnamed_function993$$MobileMail
MobileMail`___lldb_unnamed_function993$$MobileMail:
-> 0xefe48:  push   {r4, r5, r6, r7, lr}
   0xefe4a:  add    r7, sp, #12
   0xefe4c:  push.w {r8, r10, r11}
   0xefe50:  sub.w  r4, sp, #24
 (lldb) po [[[[$r2 object] copyAllMessages] anyObject]
firstSender]
NYTimes.com <nytdirect@nytimes.com>
(lldb) po [[[[$r2 object] copyAllMessages] anyObject]
sendersIfCached]
<__NSArrayI 0x16850850>(
NYTimes.com <nytdirect@nytimes.com>
)
(lldb) po [[[[$r2 object] copyAllMessages] anyObject]
senders]
<__NSArrayI 0x16850850>(
NYTimes.com <nytdirect@nytimes.com>
)
(lldb) po [[[[$r2 object] copyAllMessages] anyObject] to]
<__NSArrayI 0x16850840>(
snakeninny@gmail.com
)
(lldb) po [[[[$r2 object] copyAllMessages] anyObject]
dateSent]
2014-10-01 21:30:32 +0000
(lldb) po [[[[$r2 object] copyAllMessages] anyObject]
subject]
Asian Morning: Told to End Protests, Organizers in Hong
Kong Vow to Expand Them
(lldb) po [[[[$r2 object] copyAllMessages] anyObject]
messageBody]
<MFMimeBody: 0x16852fc0>
```

打印的信息含义都很明显，想必不用解释了。其中，firstSender返回了一个发件人，而sendersIfCached和senders都返回了一个NSArray，说明默认情况下，在iOS中一封邮件是有可能存在多个发件人的。尽管多个发件人的情况在现实生活中不常见（笔者没见过），但为了避免遗漏，这里仍采用senders函数来提取一封邮件中所有可能的发件人地址。最后的任务就是把邮件标记为已读——还记得8.2.5节末尾出现的[MessageMegaMall markMessagesAsViewed:]吗？它是不是把邮件标为已读的方法呢？在这个方法上下一个断点，看看在把一封邮件标记为已读时，它会不会得到调用。

先在IDA里定位到[MessageMegaMall markMessagesAsViewed:]，看看它的基地址，如图8-17所示。

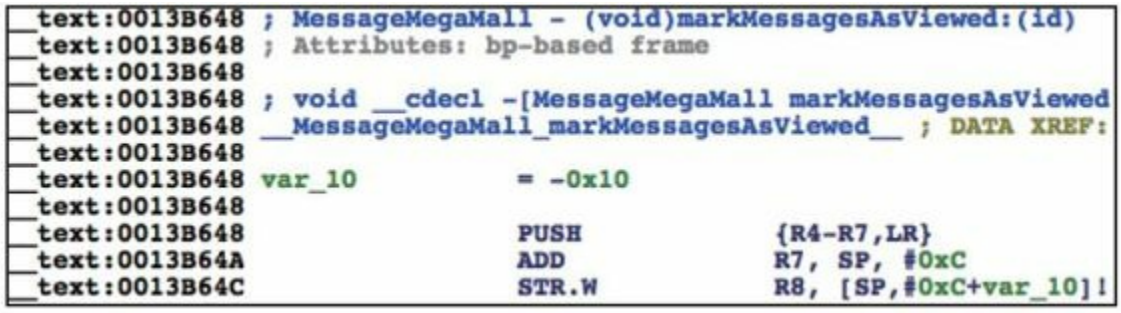


图8-17 [MessageMegaMall markMessagesAsViewed:]

它的基地址是0x13b648。因为已知MobileMail的ASLR偏移是0xb2000，所以可以直接下断点，如下：

```
(lldb) br s -a '0x000b2000+0x0013B648'
Breakpoint 4: where =
MobileMail`___lldb_unnamed_function7357$$MobileMail,
address = 0x001ed648
Process 103910 stopped
* thread #1: tid = 0x195e6, 0x001ed648
MobileMail`___lldb_unnamed_function7357$$MobileMail,
queue = 'com.apple.main-thread, stop reason = breakpoint
4.1
    frame #0: 0x001df648
MobileMail`___lldb_unnamed_function7357$$MobileMail
MobileMail`___lldb_unnamed_function7357$$MobileMail:
-> 0x1ed648:  push   {r4, r5, r6, r7, lr}
   0x1ed64a:  add    r7, sp, #12
   0x1ed64c:  str    r8, [sp, #-4]!
   0x1ed650:  mov    r8, r0
(lldb) po $r2
{(
    <MFLibraryMessage 0x157b70b0: library id 906, remote
id 13142, 2014-12-17 22:34:30 +0000, 'Asian Morning:
Obama Announces U.S. and Cuba Will Resume Relations'>
)}
(lldb) po [$r2 class]
__NSSetI
```

LLDB的输出验证了之前的猜测，[MessageMegaMall markMessagesAsViewed:]就是把邮件标为已读的方法，且其参数是一个由MFLibraryMessage对象组成的NSSet。至此，我们成功地在界面上添加了白名单按钮，捕获到了“刷新完成”事件，拿到了所有邮件，提取了其中的发件人地址，并能将它们标为已读。tweak原型搭建完毕，在写代码前，先整理一下逆向结果。

8.3 逆向结果整理

本章的实例模块划分明显，任务分工明确，搭建tweak原型时的思路非常清晰，具体如下。

1.在界面上寻找添加白名单按钮的地方和方法

本着既要直观又要美观的原则，我们在简单尝试了几种方案之后，决定把白名单按钮添加在Mailboxes界面的左上角。通过Cycript拿到MailboxPickerController的套路已是驾轻就熟，在导航栏添加按钮自然是水到渠成。

2.在protocol里寻找“刷新完成”的响应函数

同第7章“寻找实时监测字数变化”的方法一脉相承，本章以MailboxContentView-Controller.h中的protocol为线索，通过遍历头文件，猜测关键词的方法，找到了“刷新完成”的响应函数。经过测试，megaMallMessageCountChanged:会在邮件数量发生变

化时得到调用，符合条件。

3.从MessageMegaMall中拿到所有邮件

根据“协议方法被调用，一般是因为方法名中提到的那个事件发生了；而那件事发生的对象，一般是协议方法的参数”的经验，在megaMallMessageCountChanged:的参数中发现了MessageMegaMall这个类。“大型购物中心”的名字很隐晦，通过对它的调查，发现它就是一个保存了所有邮件的M。调用[MessageMegaMall copyAllMessages]，可以拿到所有邮件。

4.从MFLibraryMessage中提取发件人地址

通过[MessageMegaMall copyAllMessages]拿到的邮件类型是MFLibraryMessage。通览MFLibraryMessage及相关头文件，用LLDB测试几个可疑的property和函数，很容易就可以从中提取发件

人地址。

5.用MessageMegaMall将邮件标记为已读

在调查MessageMegaMall时，就已经注意到了可疑的markMessagesAsViewed:，几乎不需要测试就能肯定它是将邮件标记为已读的函数，当然，LLDB的测试结果也直接证明了这个结论的正确性。

注意　为了简化示例，8.4节的白名单仅含有一个邮箱地址，以UIAlertView的形式展现给用户。作为练习，你可以把它扩充成一个UITableView，让它变得更实用。

8.4 编写tweak

在搭建tweak原型的阶段，所有的难点都已被一一攻破，正式编写tweak时只需要简单地整理一下8.3节的结论，就可以得出的漂亮的代码了。

8.4.1 用Theos新建tweak工程“iOSREMailMarker”

新建iOSREMailMarker工程的命令如下：

```
hangcom-mba:Documents sam$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): iOSREMailMarker
Package Name [com.yourcompany.iosremailmarker]:
com.iosre.mailmarker
Author/Maintainer Name [sam]: sam
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.apple.mobilemail
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]: MobileMail
Instantiating iphone/tweak in iosremailmarker/...
```

```
Done.
```

8.4.2 构造iOSREMailMarker.h

编辑后的iOSREMailMarker.h内容如下：

```
@interface MailboxPickerController :
UITableViewController
@end
@interface NSConcreteNotification : NSNotification
@end
@interface MessageMegaMall : NSObject
- (void)markMessagesAsViewed:(NSSet *)arg1;
- (NSSet *)copyAllMessages;
@end
@interface MFMessageInfo : NSObject
@property (nonatomic) BOOL read;
@end
@interface MFLibraryMessage : NSObject
- (NSArray *)senders;
- (MFMessageInfo *)copyMessageInfo;
@end
```

这个头文件的所有内容均摘自类对应的头文件，构造它的目的仅仅是通过编译，避免出现任何报错信息和警告。

8.4.3 编辑Tweak.xm

编辑后的Tweak.xm内容如下：

```
#import "iOSREMailMarker.h"
%hook MailboxPickerController
%new
- (void)iOSREShowWhitelist
{
      UIAlertController *alertController =
[UIAlertController alertControllerWithTitle:@"Whitelist"
message:@"Please input an email address"
preferredStyle:UIAlertControllerStyleAlert];
      UIAlertAction *okAction = [UIAlertAction
actionWithTitle:@"OK" style:UIAlert ActionStyleDefault
handler:^(UIAlertAction * action) {
            UITextField *whitelistField =
alertController.textFields.firstObject;
            if ([whitelistField.text length] != 0)
[[NSUserDefaults standardUser Defaults]
setObject:whitelistField.text forKey:@"whitelist"];
      }];
      UIAlertAction *cancelAction = [UIAlertAction
actionWithTitle:@"Cancel" style: UIAlertActionStyleCancel
handler:nil];
      [alertController addAction:okAction];
      [alertController addAction:cancelAction];
    [alertController
addTextFieldWithConfigurationHandler:^(UITextField
*textField) {
            textField.placeholder =
@"snakeninny@gmail.com";
            textField.text = [[NSUserDefaults
standardUserDefaults] objectForKey: @"whitelist"];
      }];
      [self presentViewController:alertController
animated:YES completion:nil];
}
- (void)viewWillAppear:(BOOL)arg1
{
    self.navigationItem.leftBarButtonItem =
[[[UIBarButtonItem alloc] initWith Title: @"Whitelist"
style:UIBarButtonItemStylePlain target:self
action:@selector(iOSREShowWhitelist)] autorelease];
      %orig;
}
%end
```

```
%hook MailboxContentViewController
- (void)megaMallMessageCountChanged:
(NSConcreteNotification *)arg1
{
      %orig;
      NSMutableSet *targetMessages = [NSMutableSet
setWithCapacity:600];
      NSString *whitelist = [[NSUserDefaults
standardUserDefaults] objectForKey: @"whitelist"];
      MessageMegaMal *mall = [arg1 object];
      NSSet *messages = [mall copyAllMessages]; //
Remember to release it later
      for (MFLibraryMessage *message in messages)
      {
            MFMessageInfo *messageInfo = [message
copyMessageInfo]; // Remember to release it later
            for (NSString *sender in [message senders])
if (!messageInfo.read && [sender rangeOfString:[NSString
stringWithFormat:@"<%@>", whitelist]].location ==
NSNotFound) [targetMessages addObject:message];
            [messageInfo release];
      }
      [messages release];
      [mall markMessagesAsViewed:targetMessages];
}
%end
```

8.4.4 编辑Makefile及control

编辑后的Makefile内容如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
TWEAK_NAME = iOSREMailMarker
iOSREMailMarker_FILES = Tweak.xm
iOSREMailMarker_FRAMEWORKS = UIKit
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
      install.exec "killall -9 MobileMail"
```

编辑后的control内容如下：

```
Package: com.iosre.mailmarker
Name: iOSREMailMarker
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Mark non-whitelist emails as read!
Maintainer: sam
Author: sam
Section: Tweaks
Homepage: http://bbs.iosre.com
```

8.4.5 测试

将写好的tweak编译打包安装到iOS后，打开Mail，因为还没配置iOSREMailMarker，所以Mail跟以前没什么两样。此时，收件箱里一共有44封未读邮件，如图8-18所示。

进入Mailboxes界面，左上角新增了一个“Whitelist”按钮。点击它，弹出一个白名单编辑对话框，如图8-19所示。

图8-18　iOSREMailMarker未配置时的Mail

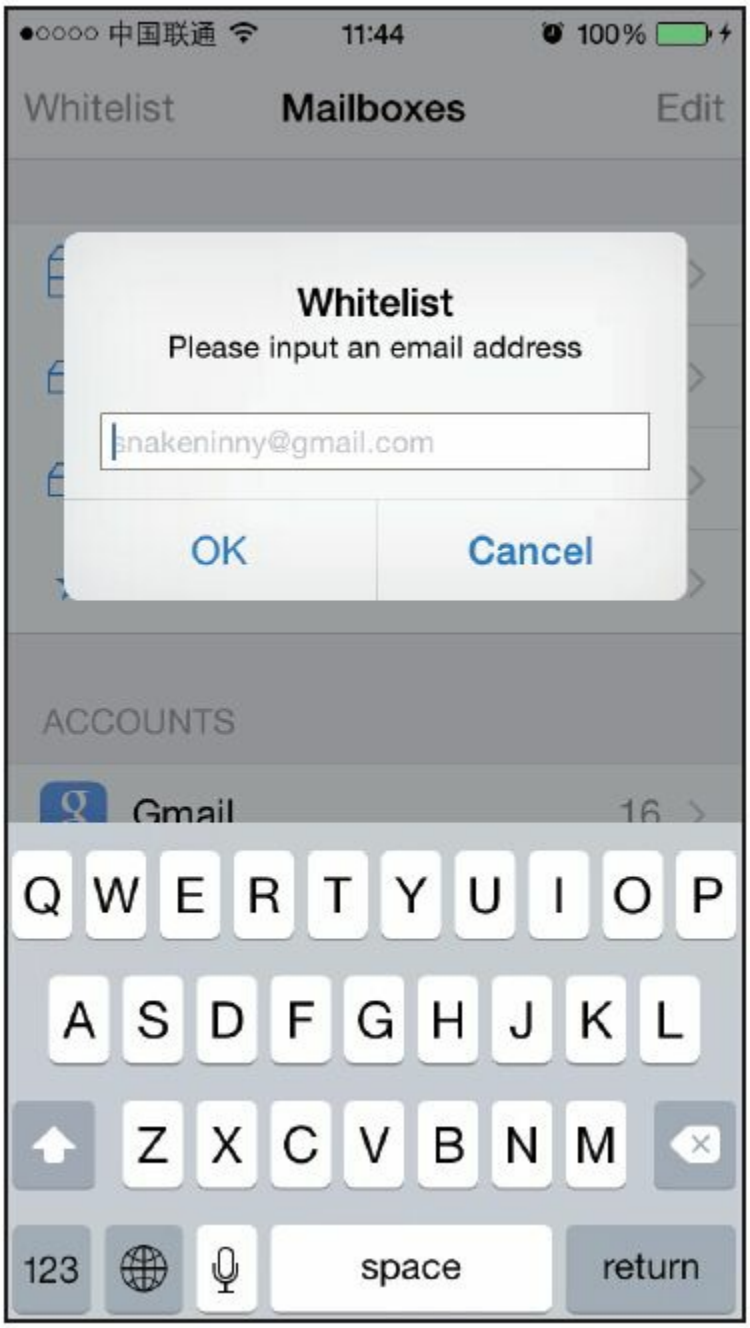


图8-19　白名单编辑对话框

笔者订阅了一份纽约时报，每天都会花上15分钟左右了解当日时事。把纽约时报的订阅邮箱地址加入白名单，如图8-20所示。

最后给自己发一封邮件，触发megaMallMessageCountChanged:函数。在成功收到这封邮件后，除了纽约时报以外的所有邮件均被标记为已读，收件箱里只剩1封来自纽约时报的未读邮件，如图8-21所示。

图8-20　把纽约时报的订阅邮箱地址加入白名单

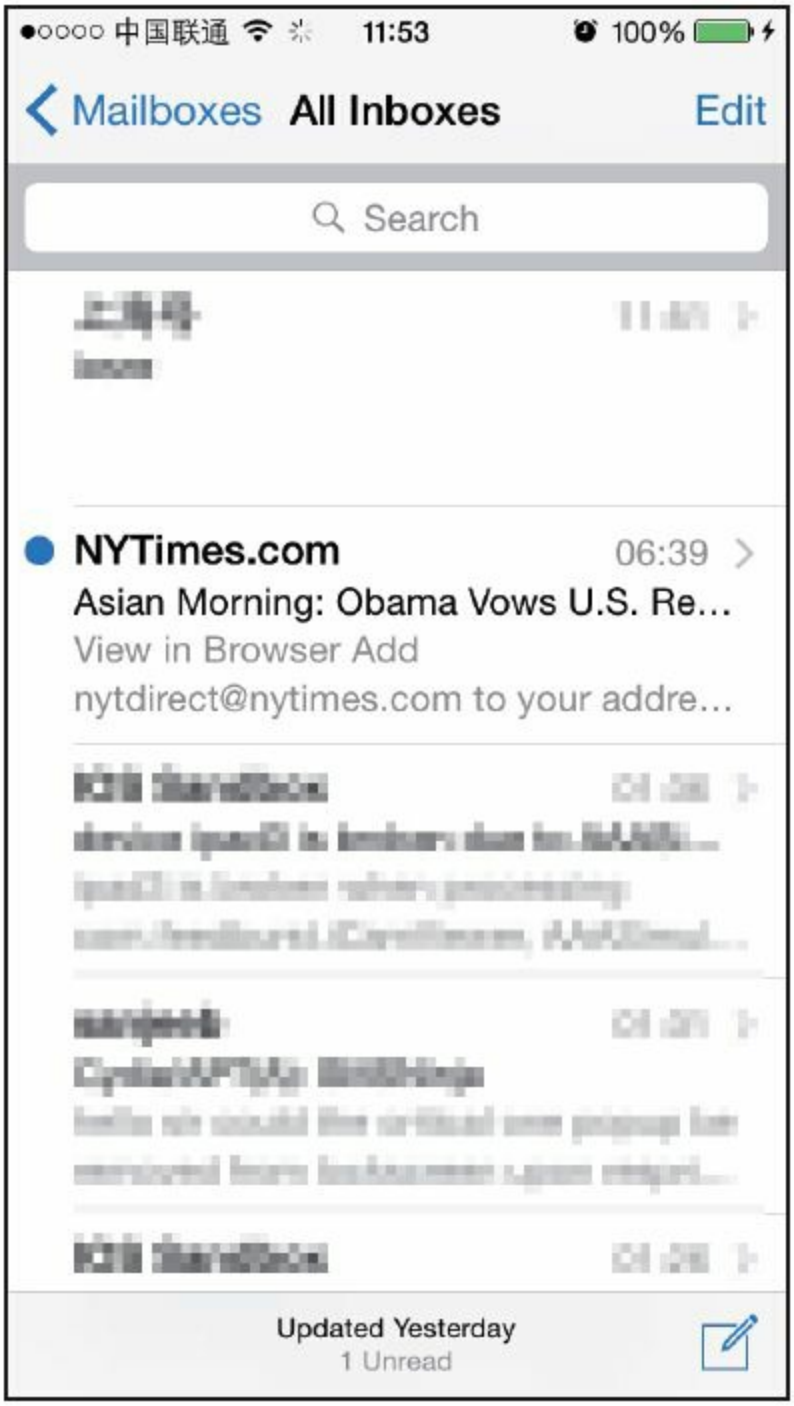


图8-21　iOSREMailMarker将纽约时报以外的邮件标记为已读

本章任务圆满结束。

8.5 小结

本章以Mail为目标，给它添加了白名单以外的邮件均自动标记为已读的功能，能够从一定程度上突出收件箱中的重点内容。iOSREMailMarker的过滤条件略显简单，将邮件直接标记为已读的方案也不一定适合所有人，希望读者能够在阅读过程中举一反三，模仿本章思路打造属于自己的独一无二的Mail插件，然后来http://bbs.iosre.com分享你的成果。经过两章实战，相信大家有跟笔者一样的体会：在iOS逆向工程中，需要工具未动，思想先行。只有在前期对目标的分析过程中下足工夫，后期的tweak编写才会如鱼得水。逆向工程行业的前辈TiGa曾说：“A reverser should know how/what is done before grabbing tools to complete the tasks automatically”，相信大家在不断的逆向求索中也会逐渐感悟这句话的含义。

第9章　实战3：保存与分享微信小视频

9.1　微信

微信是移动互联网时代即时通讯领域的佼佼者，在国内更是当之无愧的业界老大。它已经融入到我们大多数人的生活中了，想必不用再多费口舌去介绍它。微信的启动画面如图9-1所示，大气之中流露出一股淡淡的忧伤。

2014年10月3日，微信更新了6.0版，新增了小视频功能。这个功能很好玩，各种各样的小视频很快就占领了朋友圈，如图9-2所示。

虽然我们已经可以通过长按小视频窗口，在弹出的菜单里点击“Favorite”，把感兴趣的内容标记下来（如图9-3所示），但笔者还不太满意——如果能把小视频保存在本地，不管联不联网，有没有微信，都能想看就看，那就好了！另外，小视频是从微信的服

务器下载的，如果能拿到下载的URL，就可以在PC中下载，或者把它发布到其他平台，跟非微信好友分享那就更好了。既然如此，本章的目标就是在小视频播放窗口的长按菜单里添加“保存到本地”和“复制URL”两个选项，然后完善对应的功能。

微信6.0版已经超过80MB，逆向工程的工作量不小，难度也比绝大多数App要大。按照惯例，在使用工具开始逆向工程之前，先分析一下抽象的目标，把它具体化，然后制定这次逆向工程的思路，再贯彻思想实地执行。下面的操作是在iPhone 5、iOS 8.1、微信6.0中完成的。在本书出版后，微信很可能也升级到了更高的版本，下面的操作会有一些细节上的变化，但总体思路是不变的，笔者会及时把最新的分析过程更新在http://bbs.iosre.com上，请大家关注。

图9-1　微信启动画面

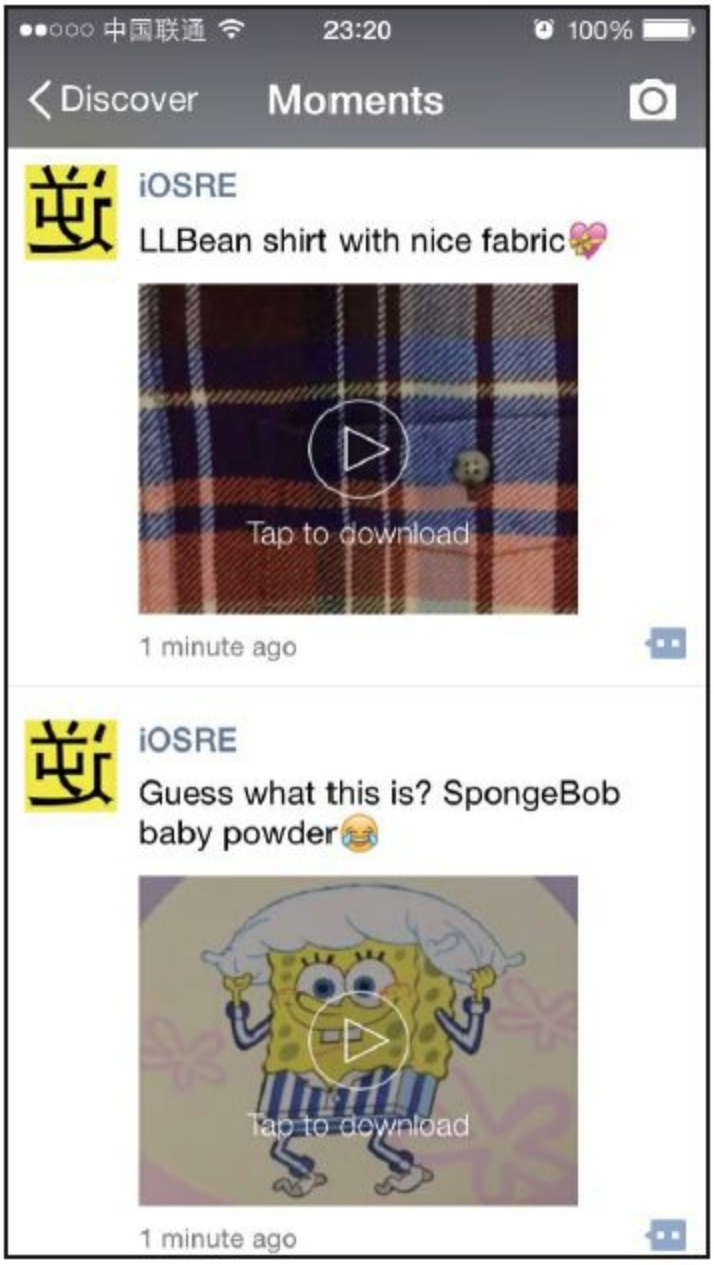

●●○○○ 中国联通
23:20
100%
Discover
Moments
iOSRE
LLBean shirt with nice fabric
Tap to download
1 minute ago
iOSRE
Guess what this is? SpongeBob baby powder
Tap to download
1 minute ago

图9-2　微信小视频

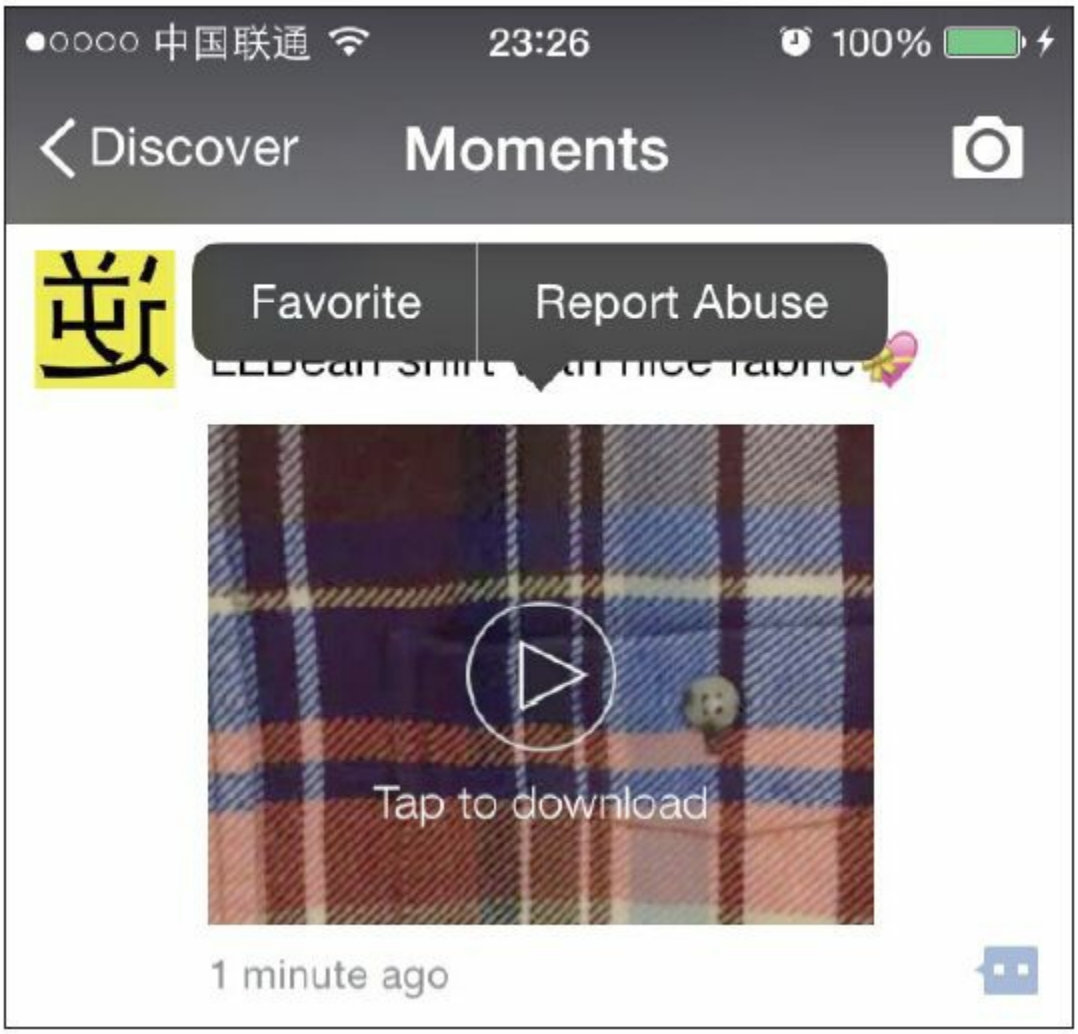

●○○○○ 中国联通
23:26
100%
Discover
Moments
Favorite
Report Abuse
Tap to download
1 minute ago

图9-3　小视频菜单

9.2 搭建tweak原型

9.2.1 观察小视频播放窗口，寻找逆向切入点

首先在“微信”→“Me”→“Settings”→“General”→“Sights in Moments”中，将小视频的自动播放选项调整到“Never”，如图9-4所示。

图9-4　调整自动播放选项

再看一下图9-3所示的界面，长按小视频播放窗口，会弹出“Favorite”和“Report Abuse”两个选项。这

个现象说明播放窗口已经可以响应长按手势，现在只要找到长按手势对应的函数，钩住（hook）它，就可以添加含有“保存到本地”和“复制URL”两个选项的自定义菜单了。

小视频播放窗口的播放按钮下有一行字，“Tap to download”，也就是说微信会先下载小视频到iOS中，再离线播放。这一现象说明小视频模块里本来就含有一个下载URL，和一个下载好的视频文件，要达到目标，只需通过逆向工程找到这个URL和视频文件就行了。经过前几章的洗礼，相信读者对MVC的理解一定比开发App更深入了，如果能够拿到小视频的V，那么含有URL和视频对象的M就近在咫尺了。

好了，本章的目标功能已经被微信实现，只需要找到它们在微信中的位置，拿来为我们所用就好了，没有必要重新发明轮子。为了追求性价比，用尽可能少的逆向工程达到目的，我们不会过分严格地推导微

信的逻辑，而是尽可能地在通过class-dump导出的头文件中寻找关键字，然后用其他工具配合验证猜测，最终达到提取小视频信息的目的。

9.2.2 class-dump获取头文件

首先用dumpdecrypted给微信砸壳，过程比较简单，这里就不细致描述了。值得一提的是，微信的可执行文件名既不叫“WeiXin”，也不叫“WeChat”，而是叫“MicroMessenger”。拿到脱壳后的可执行文件时，先把它丢到IDA里开始分析，然后用class-dump导出它的头文件，如下：

```
snakeninnysiMac:~ snakeninny$ class-dump -S -s -H
~/MicroMessenger -o ~/header6.0
```

执行上述命令，发现一共生成了5225个头文件，如图9-5所示。

微信算是笔者见过的头文件数目最多的App了，

5000+的数目如果真要让咱们一个个过，那得看到猴年马月去。即使是正向开发，这种级别的工程量也不大可能由一个团队单独完成——估计腾讯内部是把微信拆分成了若干模块，比如朋友圈是一个模块，IM是一个模块，漂流瓶是一个模块，小视频是一个模块，然后分别组建团队编写各自负责的模块，最后整合成了一个大工程，通过这样的分工协作最终实现了微信这样一个App。

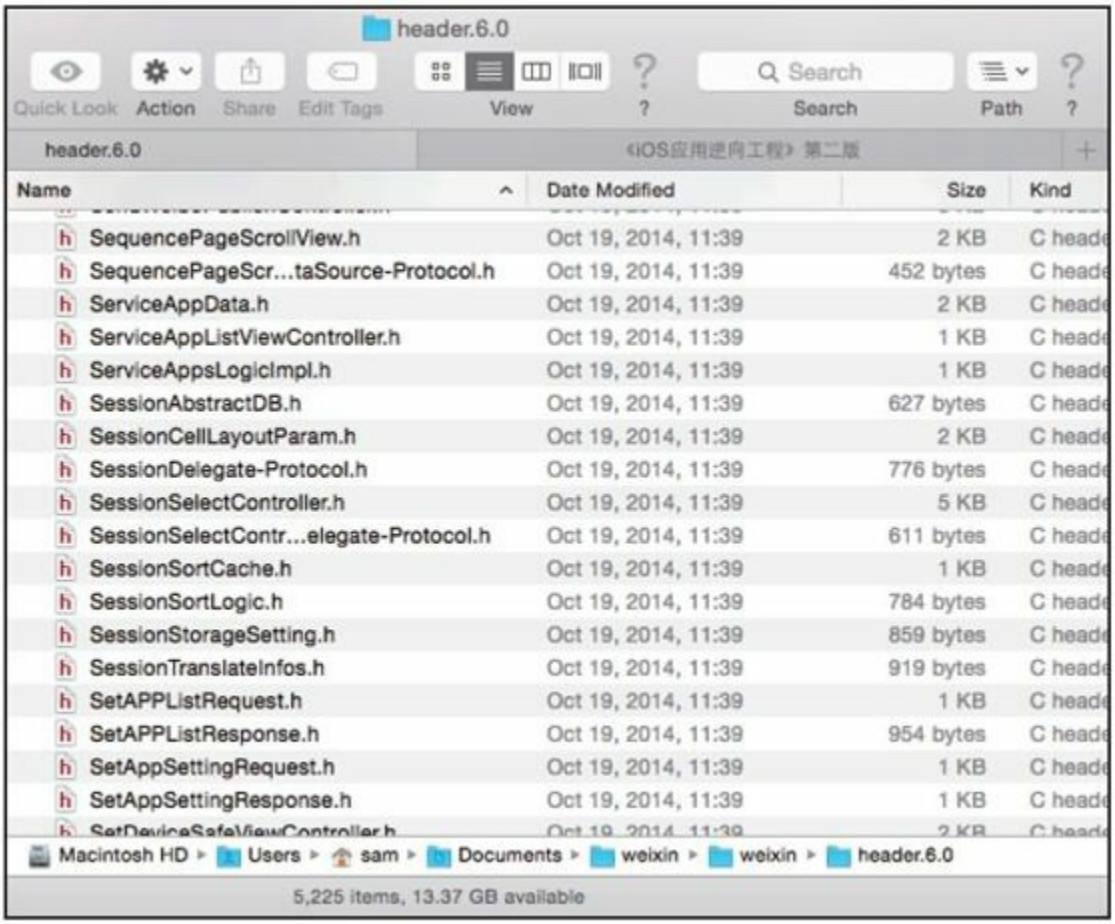


图9-5　微信6.0头文件

9.2.3　把头文件导入Xcode

把微信的头文件导入一个空的Xcode工程中，如图9-6所示。

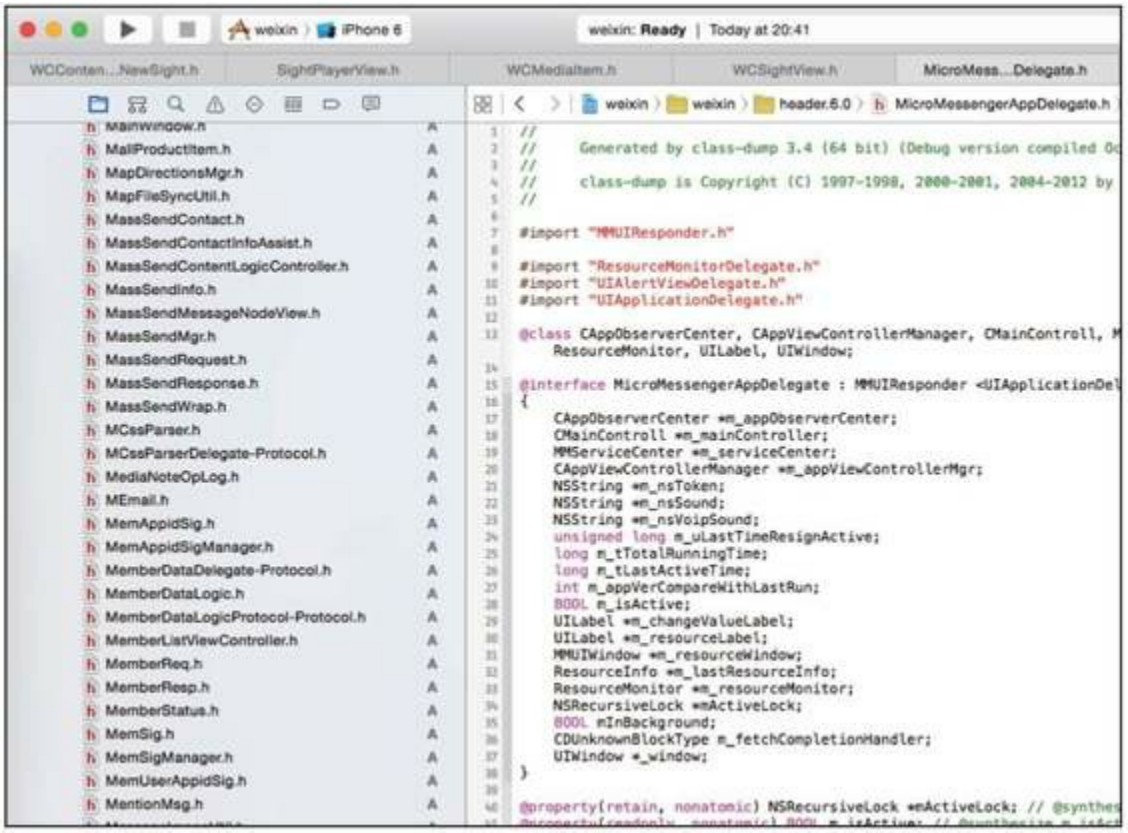


图9-6　把头文件导入Xcode

Xcode自带的查找功能和代码高亮显示能够较为美观整洁地展示大量头文件。接下来，我们开始寻找

线索，从App切入代码。

9.2.4　用Reveal找到小视频播放窗口

配置Reveal查看微信的方法也很简单，此处不再赘述。启动微信并进入朋友圈，找一个小视频，用Reveal看看当前的UI布局，如图9-7所示。

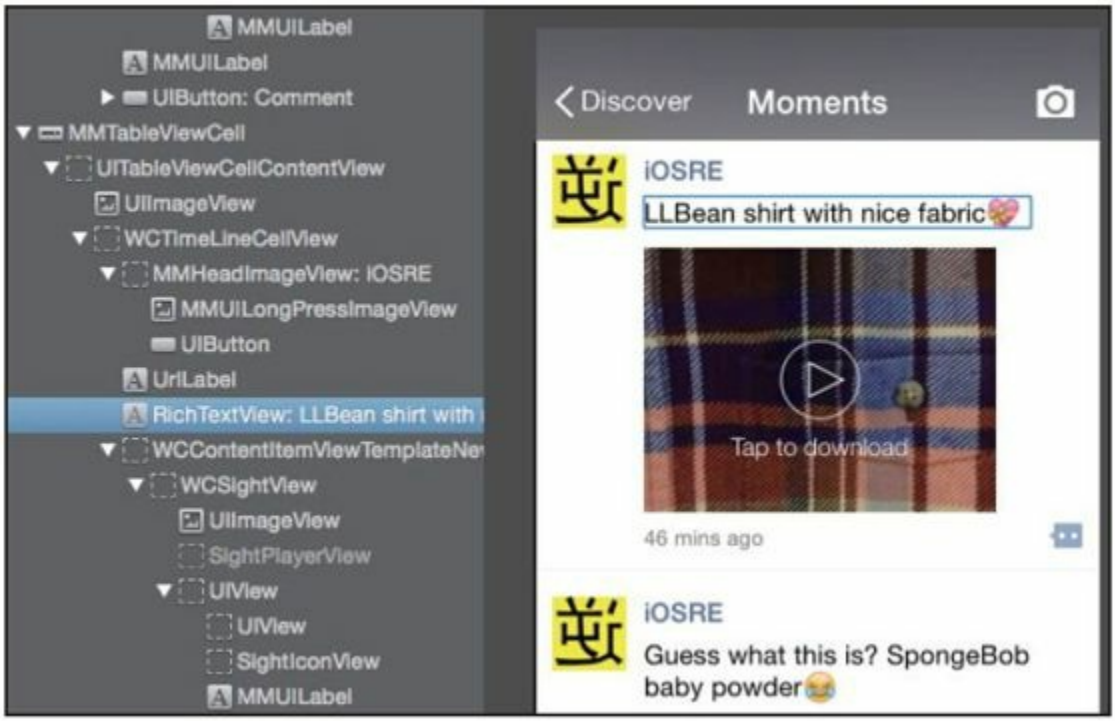


图9-7　用Reveal查看微信UI布局

在图9-7中一眼就能看到左侧的树形结构图中出现了“LLBean shirt with nice fabric”的字眼，与UI中显

示的文字吻合。继续查看这个RichTextView附近的view，很容易就可以定位小视频播放窗口，如图9-8所示。

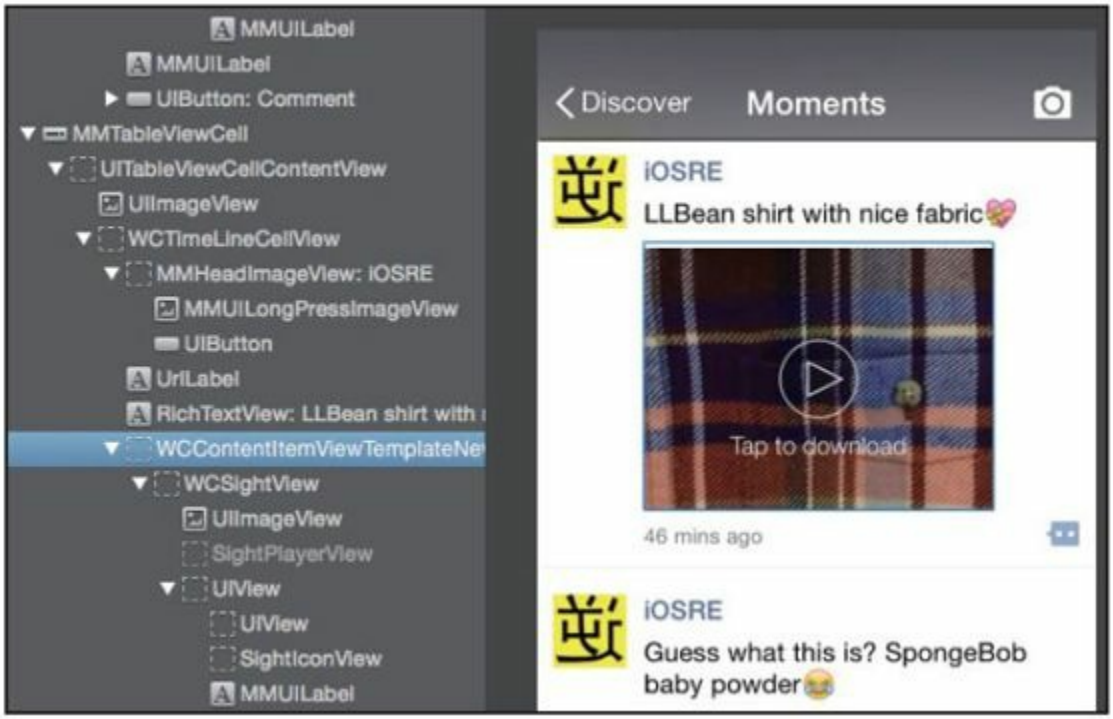


图9-8　找到小视频播放窗口

小视频的播放窗口是一个WCContentItemViewTemplateNewSight对象。还记得前面在recursiveDescription部分讲过的缩进原则吗？依照“缩进多的view是缩进少的view的subview”的原则，可分析出WCContentItemViewTemplateNewSight

的subview有WCSightView等，而WCSightView的subview则有UIImageView和SightPlayerView等。因为微信小视频的英文名就叫“sight”，所以这几个类是重点关注对象。

9.2.5 找到长按手势响应函数

要在iOS中添加长按手势，一般是通过addGestureRecognizer:方法来实现的，既然长按小视频播放窗口会弹出菜单，那么长按手势很有可能是直接添加在小视频播放窗口上的。这个播放窗口是一个WCContentItemViewTemplateNewSight对象，看看它的头文件里有些什么，如下：

```
@interface WCContentItemViewTemplateNewSight :
WCContentItemBaseView <WCAction Sheet Delegate,
SessionSelectControllerDelegate, WCSightView Delegate>
……
- (void)onMore:(id)arg1;
- (void)onFavoriteAdd:(id)arg1;
- (void)onLongTouch;
- (void)onShowSightAction;
- (void)onLongPressedWCSightFullScreenWindow:(id)arg1;
- (void)onLongPressedWCSight:(id)arg1;
- (void)onClickWCSight:(id)arg1;
……
```

```
@end
```

在上面的代码中，那几个含有“LongTouch”、“LongPressed”字眼的函数很可能就是我们寻找的长按手势响应函数。IDA对微信的初始分析应该已经结束了，在IDA里瞟一眼这几个函数都做了些什么，先看看onLongTouch，如图9-9所示。

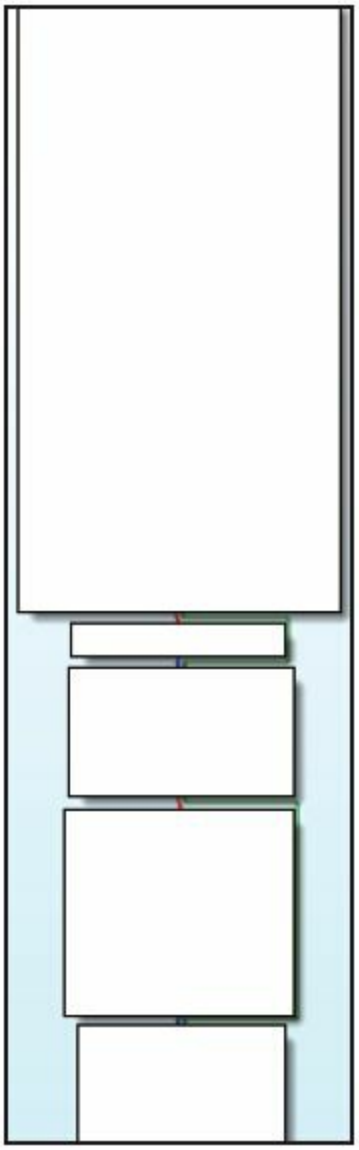
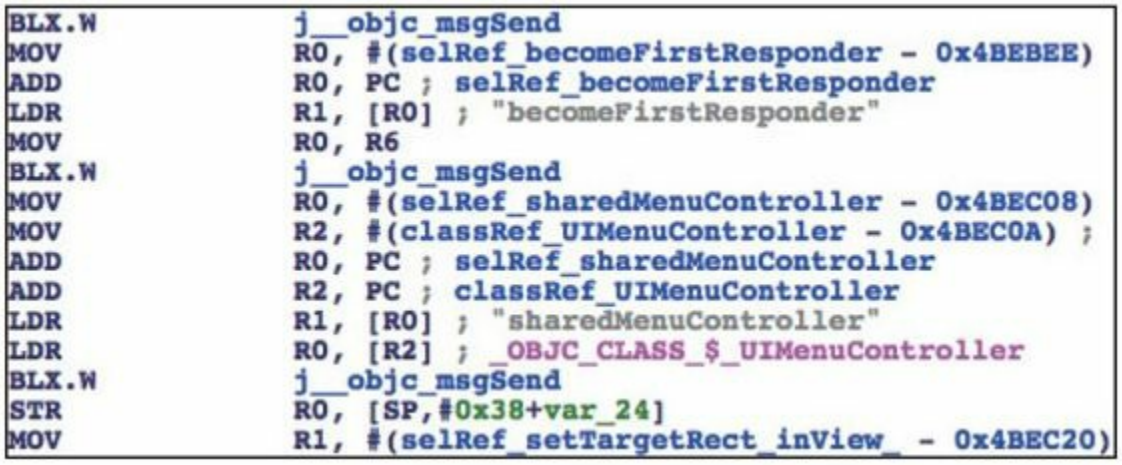


图9-9　onLongTouch

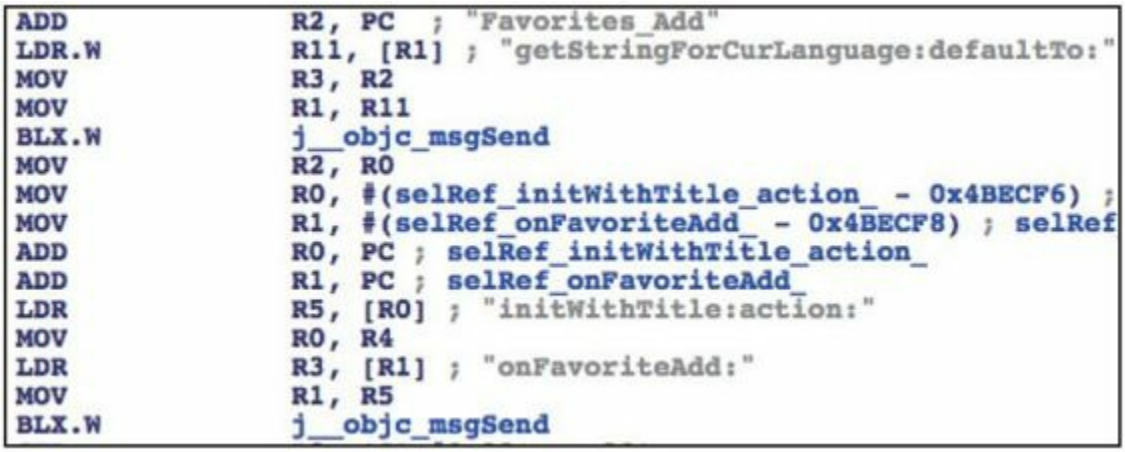
这个函数的流程非常简单，从上往下浏览，很容易就可以看到“UIMenuController”的字眼，如图9-10所示。



```
BLX.W           j__objc_msgSend
MOV             R0, #(selRef_becomeFirstResponder - 0x4BEBEE)
ADD             R0, PC ; selRef_becomeFirstResponder
LDR             R1, [R0] ; "becomeFirstResponder"
MOV             R0, R6
BLX.W           j__objc_msgSend
MOV             R0, #(selRef_sharedMenuController - 0x4BEC08)
MOV             R2, #(classRef_UIMenuController - 0x4BEC0A) ;
ADD             R0, PC ; selRef_sharedMenuController
ADD             R2, PC ; classRef_UIMenuController
LDR             R1, [R0] ; "sharedMenuController"
LDR             R0, [R2] ; _OBJC_CLASS_$_UIMenuController
BLX.W           j__objc_msgSend
STR             R0, [SP,#0x38+var_24]
MOV             R1, #(selRef_setTargetRect_inView_ - 0x4BEC20)
```

图9-10　onLongTouch（1）

也可以看到“Favorite”的字眼，如图9-11所示。

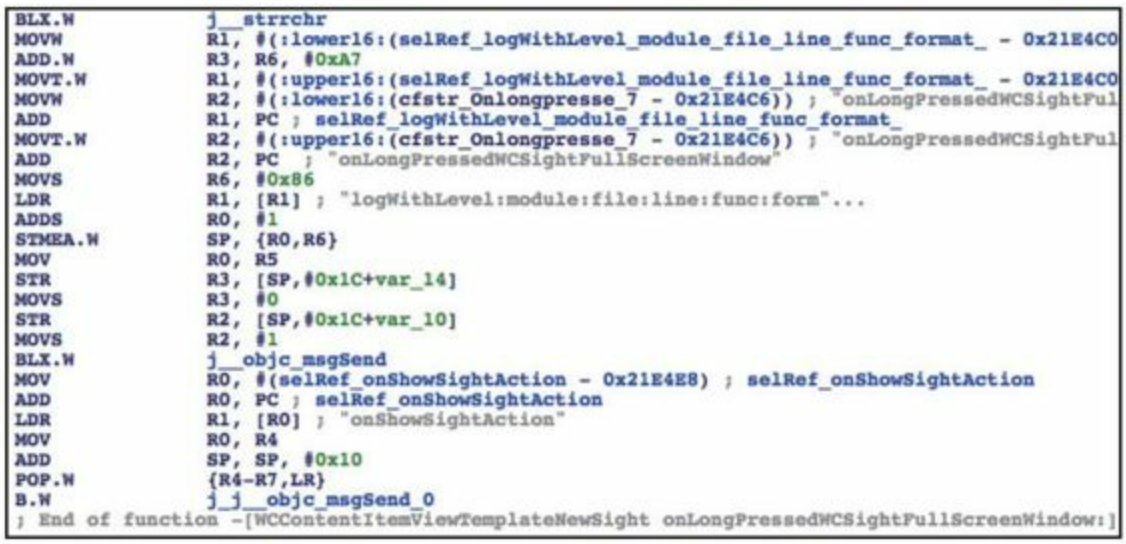


```
ADD             R2, PC  ; "Favorites_Add"
LDR.W           R11, [R1] ; "getStringForCurLanguage:defaultTo:"
MOV             R3, R2
MOV             R1, R11
BLX.W           j__objc_msgSend
MOV             R2, R0
MOV             R0, #(selRef_initWithTitle_action_ - 0x4BECF6) ;
MOV             R1, #(selRef_onFavoriteAdd_ - 0x4BECF8) ; selRef
ADD             R0, PC ; selRef_initWithTitle_action_
ADD             R1, PC ; selRef_onFavoriteAdd_
LDR             R5, [R0] ; "initWithTitle:action:"
MOV             R0, R4
LDR             R3, [R1] ; "onFavoriteAdd:"
MOV             R1, R5
BLX.W           j__objc_msgSend
```

图9-11　onLongTouch（2）

除非这些字眼都是微信有意拿来迷惑我们的，否则这个[WCContentItemViewTemplate-NewSight onLongTouch]十有八九就是要找的长按手势响应函数。先不着急下结论，把带有“LongPressed”关键字的

函数也浏览一遍，如图9-12所示。



```
BLX.W           j__strrchr
MOVW            R1, #(:lower16:(selRef_logWithLevel_module_file_line_func_format_ - 0x21E4C0
ADD.W           R3, R6, #0xA7
MOVT.W          R1, #(:upper16:(selRef_logWithLevel_module_file_line_func_format_ - 0x21E4C0
MOVW            R2, #(:lower16:(cfstr_Onlongpresse_7 - 0x21E4C6)) ; "onLongPressedWCSightFul
ADD             R1, PC ; selRef_logWithLevel_module_file_line_func_format_
MOVT.W          R2, #(:upper16:(cfstr_Onlongpresse_7 - 0x21E4C6)) ; "onLongPressedWCSightFul
ADD             R2, PC  ; "onLongPressedWCSightFullScreenWindow"
MOVS            R6, #0x86
LDR             R1, [R1] ; "logWithLevel:module:file:line:func:form"...
ADDS            R0, #1
STMEA.W         SP, {R0,R6}
MOV             R0, R5
STR             R3, [SP,#0x1C+var_14]
MOVS            R3, #0
STR             R2, [SP,#0x1C+var_10]
MOVS            R2, #1
BLX.W           j__objc_msgSend
MOV             R0, #(selRef_onShowSightAction - 0x21E4E8) ; selRef_onShowSightAction
ADD             R0, PC ; selRef_onShowSightAction
LDR             R1, [R0] ; "onShowSightAction"
MOV             R0, R4
ADD             SP, SP, #0x10
POP.W           {R4-R7,LR}
B.W             j_j__objc_msgSend_0
; End of function -[WCContentItemViewTemplateNewSight onLongPressedWCSightFullScreenWindow:]
```

图9-12　onLongPressedWCSightFullScreenWindow:

看上去是记录了一些信息，然后调用了onShowSightAction。接下来看看onShowSight-Action都做了些什么，如图9-13所示。

从图9-13可以看到，这个函数的一开始就创建了一个WCActionSheet对象，既然名字是ActionSheet，它的表现形式可能与UIActionSheet差不多，而我们在长按小视频播放窗口根本没看到与UIActionSheet类似的效果出现，因此可以判断onLongPressedWCSightFull-ScreenWindow:可能不是我

们要找的函数。

接着看最后一个函数，onLongPressedWCSight:，如图9-14所示。

从图9-14可以看到，它记录了一些信息，然后调用了onLongTouch，这从侧面印证了我们的猜测。接下来，请出LLDB，测测onLongPressedWCSightFullScreenWindow:和onLongTouch的调用情况。先用debugserver附加MicroMessenger，如下：

```
; WCContentItemViewTemplateNewSight - (void)onShowSightAction
; Attributes: bp-based frame

; void __cdecl -[WCContentItemViewTemplateNewSight onShowSightAction]
__WCContentItemViewTemplateNewSight_onShowSightAction_

var_38= -0x38
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
PUSH.W          {R8,R10,R11}
SUB             SP, SP, #0x20
MOV             R4, R0
STR             R4, [SP,#0x38+var_1C]
MOV             R0, #(selRef_alloc - 0x21E516) ; selRef_alloc
MOV             R2, #(classRef_WCActionSheet - 0x21E518) ; classRef_W
ADD             R0, PC ; selRef_alloc
ADD             R2, PC ; classRef_WCActionSheet
LDR             R1, [R0] ; "alloc"
LDR             R0, [R2] ; _OBJC_CLASS_$_WCActionSheet
BLX.W           j__objc_msgSend
MOVW            R1, #(:lower16:(selRef_initWithTitle_delegate_cancelB
MOVS            R2, #0
MOVT.W          R1, #(:upper16:(selRef_initWithTitle_delegate_cancelB
STR             R2, [SP,#0x38+var_38]
ADD             R1, PC ; selRef_initWithTitle_delegate_cancelButtonTi
STR             R2, [SP,#0x38+var_34]
STR             R2, [SP,#0x38+var_30]
```

图9-13　onShowSightAction

```
BLX.W           j__strrchr
MOVW            R1, #(:lower16:(selRef_logWithLevel_module_file_line_func_fo
ADD.W           R3, R6, #0x6C
MOVT.W          R1, #(:upper16:(selRef_logWithLevel_module_file_line_func_fo
MOVW            R2, #(:lower16:(cfstr_Onlongpressedw - 0x21E456)) ; "onLongP
ADD             R1, PC ; selRef_logWithLevel_module_file_line_func_format_
MOVT.W          R2, #(:upper16:(cfstr_Onlongpressedw - 0x21E456)) ; "onLongP
ADD             R2, PC  ; "onLongPressedWCSight"
MOVS            R6, #0x7F
LDR             R1, [R1] ; "logWithLevel:module:file:line:func:form"...
ADDS            R0, #1
STMEA.W         SP, {R0,R6}
MOV             R0, R5
STR             R3, [SP,#0x1C+var_14]
MOVS            R3, #0
STR             R2, [SP,#0x1C+var_10]
MOVS            R2, #1
BLX.W           j__objc_msgSend
MOV             R0, #(selRef_onLongTouch - 0x21E478) ; selRef_onLongTouch
ADD             R0, PC ; selRef_onLongTouch
LDR             R1, [R0] ; "onLongTouch"
MOV             R0, R4
ADD             SP, SP, #0x10
POP.W           {R4-R7,LR}
B.W             j_j__objc_msgSend_0
; End of function -[WCContentItemViewTemplateNewSight onLongPressedWCSight:]
```

图9-14　onLongPressedWCSight:

```
snakeninnysiMac:Documents snakeninny$ ssh root@localhost
-p 2222
FunMaker-5:~ root# debugserver *:1234 -a MicroMessenger
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-
320.2.89
 for armv7.
Attaching to process MicroMessenger...
Listening to port 1234 for a connection from *...
Waiting for debugger instructions for process 0.
```

然后看看微信的ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x00000000
/private/var/mobile/Containers/Bundle/Application/E4EBD04
1A75-4830-BC65-
0132C0EBC1CA/MicroMessenger.app/MicroMessenger(0x00000000

[  1] 0x022dc000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000

……
```

微信的ASLR偏移是0。接着看看onLongPressedWCSightFullScreenWindow:和onLongTouch的基地址，如图9-15和图9-16所示。

图9-15　onLongPressedWCSightFullScreenWindow:的基地址

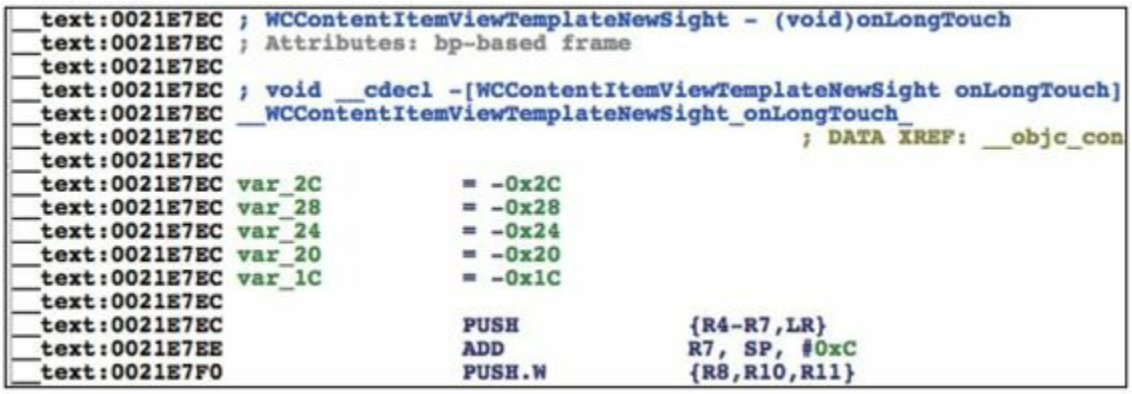


图9-16　onLongTouch的基地址

它们的基地址分别是0x21e484和0x21e7ec。下面在两个函数的开头各下一个断点，看看长按小视频播放窗口后，断点会不会被触发，如下：

```
(lldb) br s -a 0x21e484
Breakpoint 3: where =
MicroMessenger`___lldb_unnamed_function9789$$MicroMesseng
 address = 0x0021e484
(lldb) br s -a 0x21e7ec
Breakpoint 4: where =
MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng
 address = 0x0021e7ec
Process 184500 stopped
```

```
* thread #1: tid = 0x2d0b4, 0x0021e7ec
MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng
 queue = 'com.apple.main-thread, stop reason = breakpoint
4.1
    frame #0: 0x0021e7ec
MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng

MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng

-> 0x21e7ec:  push   {r4, r5, r6, r7, lr}
   0x21e7ee:  add    r7, sp, #12
   0x21e7f0:  push.w {r8, r10, r11}
   0x21e7f4:  sub    sp, #32
(lldb) p (char *)$r1
(char *) $0 = 0x017fdc2b "onLongTouch"
(lldb) c
Process 184500 resuming
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e7ec
MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng
 queue = 'com.apple.main-thread, stop reason = breakpoint
4.1
    frame #0: 0x0021e7ec
MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng

MicroMessenger`___lldb_unnamed_function9791$$MicroMesseng

-> 0x21e7ec:  push   {r4, r5, r6, r7, lr}
   0x21e7ee:  add    r7, sp, #12
   0x21e7f0:  push.w {r8, r10, r11}
   0x21e7f4:  sub    sp, #32
(lldb) p (char *)$r1
(char *) $1 = 0x017fdc2b "onLongTouch"
```

可以看到，onLongTouch被调用了2次，而onLongPressedWCSightFullScreenWindow:没有被调用。再看看onLongPressedWCSight:的调用情况，它的基地址如图9-17所示。

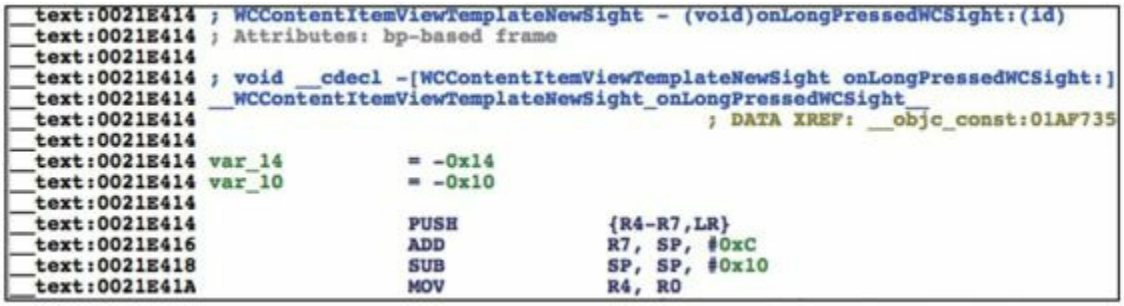


图9-17　onLongPressedWCSight:的基地址

然后下个断点，看看它会不会被触发，如下：

```
(lldb) c
Process 184500 resuming
(lldb) br del
About to delete all breakpoints, do you want to do that?:
[Y/n] y
All breakpoints removed. (2 breakpoints)
(lldb) br s -a 0x21e414
Breakpoint 5: where =
MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng
 address = 0x0021e414
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e414
MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng
 queue = 'com.apple.main-thread, stop reason = breakpoint
5.1
    frame #0: 0x0021e414
MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng

MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng

-> 0x21e414:  push   {r4, r5, r6, r7, lr}
   0x21e416:  add    r7, sp, #12
   0x21e418:  sub    sp, #16
   0x21e41a:  mov    r4, r0
(lldb) p (char *)$r1
(char *) $2 = 0x0182c799 "onLongPressedWCSight:"
(lldb) c
Process 184500 resuming
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x0021e414
MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng
```

```
 queue = 'com.apple.main-thread, stop reason = breakpoint
5.1
    frame #0: 0x0021e414
MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng

MicroMessenger`___lldb_unnamed_function9788$$MicroMesseng

-> 0x21e414:  push   {r4, r5, r6, r7, lr}
   0x21e416:  add    r7, sp, #12
   0x21e418:  sub    sp, #16
   0x21e41a:  mov    r4, r0
(lldb) p (char *)$r1
(char *) $3 = 0x0182c799 "onLongPressedWCSight:"
(lldb) po $r2
<WCSightView: 0x2454dc0; baseClass = UIControl; frame =
(0 3; 200 150); gestureRecognizers = <NSArray:
0x87e5110>; layer = <CALayer: 0xd3be460>>
```

这里onLongPressedWCSight:也被调用了2次，且其参数是一个WCSightView对象。到此，我们已经定位到了小视频播放窗口的长按响应函数，即onLongPressedWCSight:或onLongTouch，接下来就要开始寻找小视频的踪影了。

9.2.6 用Cycript定位小视频的controller

首先点击小视频窗口中的“Tap to download”，把视频下载到本机，如图9-18所示。

图9-18　下载小视频

下载成功后，“Tap to download”字样消失。通过V拿到C进而定位M的过程前面已经重复过很多次了，这里直接操作起来，如下：

```
FunMaker-5:~ root# cycript -p MicroMessenger
cy# ?expand
```

```
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
@"<iConsoleWindow: 0x2392e50; baseClass = UIWindow; frame
= (0 0; 320 568); gestureRecognizers = <NSArray:
0x2391b00>; layer = <UIWindowLayer: 0x2391690>>
   | <UILayoutContainerView: 0x7e71870; frame = (0 0; 320
568); autoresize = W+H; layer = <CALayer: 0x7e71830>>
   |    | <UITransitionView: 0x7e720b0; frame = (0 0; 320
568); clipsToBounds = YES; autoresize = W+H; layer =
<CALayer: 0x7e722a0>>
……
   |    |    |    |    |    |    |    |    |    |    |
|    | <WCContentItemViewTemplateNewSight: 0xd3be3e0;
frame = (61 64; 200 153); clipsToBounds = YES; layer =
<CALayer: 0x7e922d0>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    | <WCSightView: 0x2454dc0; baseClass =
UIControl; frame = (0 3; 200 150); gestureRecognizers =
<NSArray: 0x87e5110>; layer = <CALayer: 0xd3be460>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    |    | <UIImageView: 0xd34e8d0; frame = (0 0;
200 150); opaque = NO; userInteractionEnabled = NO; layer
= <CALayer: 0xd34e950>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    |    | <SightPlayerView: 0x7e50ff0; frame = (0
0; 200 150); layer = <CALayer: 0xd302770>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    |    | <UIView: 0xd37d9e0; frame = (0 0; 200
150); layer = <CALayer: 0xd37da50>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    | <UIView: 0xd30d5f0; frame = (0 0;
200 150); tag = 10050; layer = <CALayer: 0x87e5650>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    | <SightIconView: 0xd3be2e0; frame =
(0 0; 200 150); layer = <CALayer: 0xd3be380>>
   |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    | <MMUILabel: 0x7ee7530; baseClass =
UILabel; frame = (0 103; 200 20); text = 'Tap to play';
hidden = YES; userInteractionEnabled = NO; tag = 10040;
layer = <_UILabelLayer: 0x7e50dd0>>
……
cy# [#0xd3be3e0 nextResponder]
#"<WCTimeLineCellView: 0x872c530; frame = (0 0; 313 243);
tag = 1048577; layer = <CALayer: 0x872ce80>>"
cy# [#0x872c530 nextResponder]
#"<UITableViewCellContentView: 0x8729d80; frame = (0 0;
320 251); gestureRecognizers = <NSArray: 0x8729f80>;
layer = <CALayer: 0x8729df0>>"
cy# [#0x8729d80 nextResponder]
```

```
#"<MMTableViewCell: 0x8729be0; baseClass =
UITableViewCell; frame = (0 1164.33; 320 251); autoresize
= W; layer = <CALayer: 0x8729b50>>"
cy# [#0x8729be0 nextResponder]
#"<UITableViewWrapperView: 0xab09890; frame = (0 0; 320
568); gestureRecognizers = <NSArray: 0xab09b00>; layer =
<CALayer: 0x7e6e4b0>; contentOffset: {0, 0}; contentSize:
{320, 568}>"
cy# [#0xab09890 nextResponder]
#"<MMTableView: 0x30c3200; baseClass = UITableView; frame
= (0 0; 320 568); gestureRecognizers = <NSArray:
0xab09600>; layer = <CALayer: 0xab09160>; contentOffset:
{0, 1090}; contentSize: {320, 3186.3333}>"
cy# [#0x30c3200 nextResponder]
#"<UIView: 0x7e3b040; frame = (0 0; 320 568); autoresize
= W+H; layer = <CALayer: 0x7e3afd0>>"
cy# [#0x7e3b040 nextResponder]
#"<WCTimeLineViewController: 0x28bd200>"
```

我们拿到了C，即WCTimeLineViewController；同时也能猜到，朋友圈的内部代号是“Time Line”。

9.2.7 从WCTimeLineViewController找到小视频对象

通览WCTimeLineViewController的头文件，你会发现其中的property很少，也没有很明显地访问M的方法，比较可疑的地方是其中的2个全局变量，如下：

```
WCDataItem *_inputDataItem;
WCDataItem *_cacheDateItem;
```

但它们全都是null，如下：

```
cy# #0x28bd200->_cacheDateItem
null
cy# #0x28bd200->_inputDataItem
null
```

线索貌似到此中断了，难道要就此放弃？当然不是！因为朋友圈是以TableView的形式展示的，而WCTimeLineViewController中存在一个名为tableView:cellForRowAtIndexPath:的方法，说明它实现了UITable-ViewDataSource协议，因此一定和M有千丝万缕的关系。那就去IDA里一探究竟，如图9-19所示。

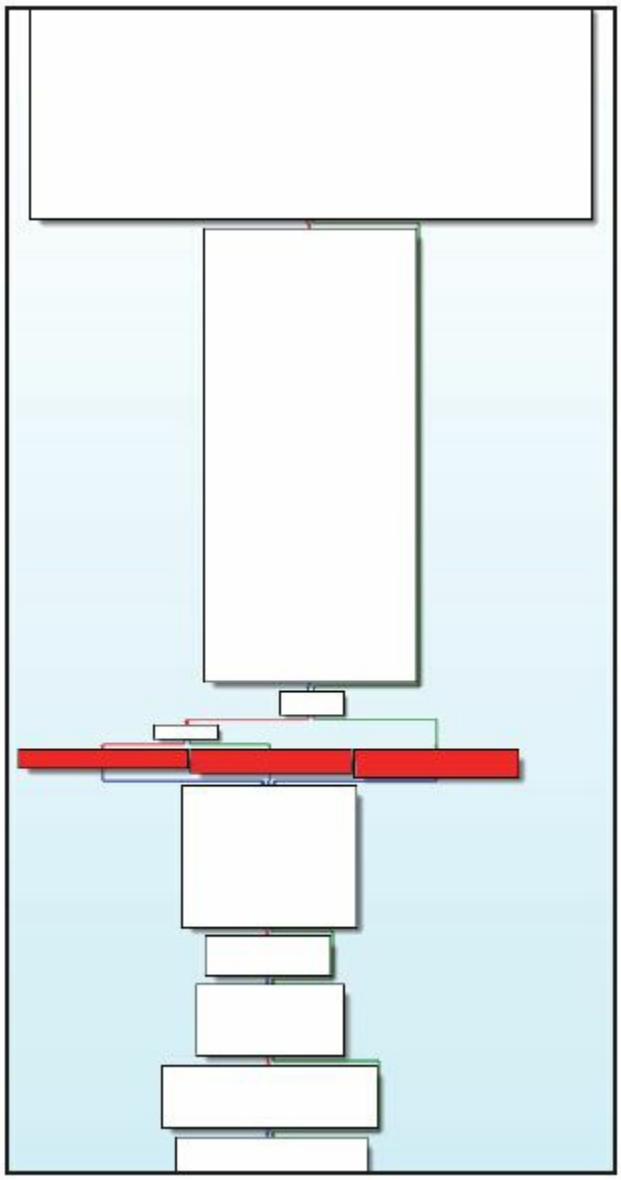


图9-19

[WCTimeLineViewControllertableView:cellForRowAtInde

通览这个函数，你会发现图9-19中的三个深色方块是整个函数的核心，其他的部分只是在给这个cell设置背景图、主题、颜色等周边元素。现在近距离看

看这三个深色方块，如图9-20所示。

图9-20　三个深色方块

图比较小，从左至右的三个函数分别是genUploadFailCell:indexPath、genNormalCell:indexPath:和genRedHeartCell:indexPath:。小视频是哪种cell呢？我想你应该也会猜它是“NormalCell”，下面看看genNormalCell:indexPath:的实现，如图9-21所示。

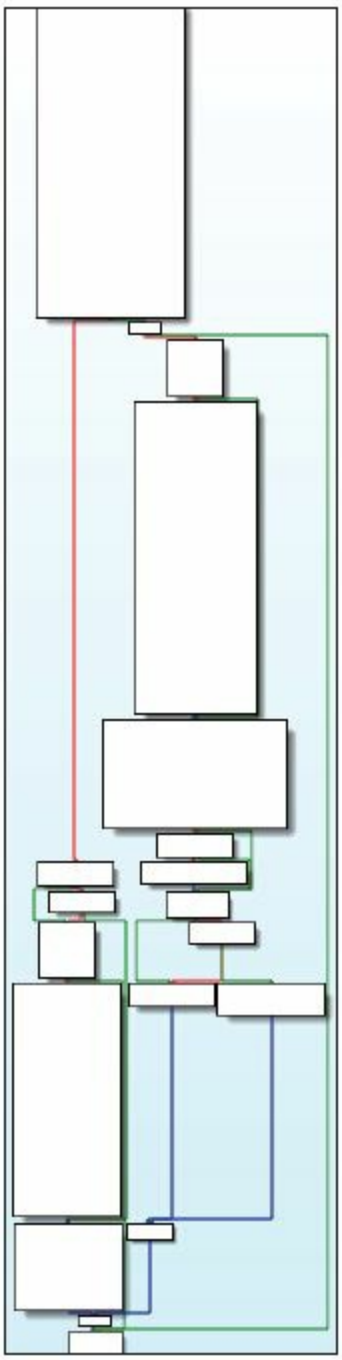


图9-21　[WCTimeLineViewController genNormalCell:indexPath:]

它的逻辑并不复杂，从上到下浏览，很快就能发现一个可疑的函数，如图9-22所示。

图9-22的getTimelineDataItemOfIndex:很有可能就是当前cell的数据源。我们在最下方的“__text:002A091C BLX.W j__objc_msgSend”上下一个断点，然后想办法触发它——当UITableView需要显示新的cell时，tableView:cellForRowAtIndexPath:会得到调用。因此，为了让断点停在带有小视频播放窗口的cell上，要先把小视频滑出当前界面，然后再滑进来。因为在把小视频滑出去的时候，新的cell会触发断点，但这种断点不符合要求，所以这里先“dis断点号”，待小视频窗口完全滑出当前界面后，再“en断点号”，然后把小视频滑回来，这时断点就会停在小视频cell上，如下：

```
MOV             R2, R0
MOV             R0, #(selRef_calcDataItemIndex_ - 0x2A08B2)
ADD             R0, PC ; selRef_calcDataItemIndex_
LDR             R1, [R0] ; "calcDataItemIndex:"
MOV             R0, R4
BLX.W           j__objc_msgSend
MOV             R5, R0
MOV             R0, #(selRef_defaultCenter - 0x2A08CE)
MOV             R2, #(classRef_MMServiceCenter - 0x2A08D0)
ADD             R0, PC ; selRef_defaultCenter
ADD             R2, PC ; classRef_MMServiceCenter
LDR             R1, [R0] ; "defaultCenter"
LDR             R0, [R2] ; _OBJC_CLASS_$_MMServiceCenter
STR             R1, [SP,#0xC8+var_A4]
BLX.W           j__objc_msgSend
MOV             R6, R0
MOV             R0, #(selRef_class - 0x2A08E6)
ADD             R0, PC ; selRef_class
LDR             R1, [R0] ; "class"
STR             R1, [SP,#0xC8+var_A8]
MOV             R0, #(classRef_WCFacade - 0x2A08F4)
ADD             R0, PC ; classRef_WCFacade
LDR             R0, [R0] ; _OBJC_CLASS_$_WCFacade
BLX.W           j__objc_msgSend
MOV             R2, R0
MOV             R0, #(selRef_getService_ - 0x2A0906)
ADD             R0, PC ; selRef_getService_
LDR             R1, [R0] ; "getService:"
MOV             R0, R6
STR             R1, [SP,#0xC8+var_AC]
BLX.W           j__objc_msgSend
MOVW            R1, #(:lower16:(selRef_getTimelineDataItemOfIndex_ - 0x2A091C))
MOV             R2, R5
MOVT.W          R1, #(:upper16:(selRef_getTimelineDataItemOfIndex_ - 0x2A091C))
ADD             R1, PC ; selRef_getTimelineDataItemOfIndex_
LDR             R1, [R1] ; "getTimelineDataItemOfIndex:"
BLX.W           j__objc_msgSend
```

图9-22　[WCTimeLineViewController genNormalCell:indexPath:]

```
(lldb) br s -a 0x2A091C
Breakpoint 6: where =
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208, address = 0x002a091c
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208, queue = 'com.apple.main-thread, stop reason =
breakpoint 6.1
    frame #0: 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208:
-> 0x2a091c:  blx    0xe08e0c     ;
___lldb_unnamed_function70162$$MicroMessenger
   0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
```

```
   0x2a0926:  movt   r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212:
-> 0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
   0x2a092a:  add    r0, pc
(lldb) po $r0
Class name: WCDataItem, addr: 0x80f52b0
tid: 11896185303680028954
username: wxid_hqouu9kgsgw3e6
createtime: 1418135798
commentUsers: (
)
contentObj: <WCContentItem: 0x8724c20>
```

我们拿到了一个WCDataItem对象，它的内部还有一个WCContentItem对象。那这个WCDataItem对象到底是不是小视频的数据呢？用LLDB来测试一下，把这个返回值给置NULL，看看是什么效果。重复刚才的操作，在小视频滑回来时触发断点，如下：

```
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208, queue = 'com.apple.main-thread, stop reason =
breakpoint 6.1
    frame #0: 0x002a091c
```

```
MicroMessenger`___lldb_unnamed_function11980$$
MicroMessenger + 208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208:
-> 0x2a091c:  blx    0xe08e0c                  ;
___lldb_unnamed_function70162$$MicroMessenger
   0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$
MicroMessenger + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212:
-> 0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
   0x2a092a:  add    r0, pc
(lldb) register write r0 0
(lldb) br del
About to delete all breakpoints, do you want to do that?:
[Y/n] y
All breakpoints removed. (1 breakpoint)
(lldb) c
```

此时，第一条小视频完全消失了，效果如图9-23所示。说明它的数据源就是WCDataItem。在分析WCDataItem之前，我们面临的问题是，如何从被钩住（hook）的函数[WCContentItemViewTemplate-NewSight onLongTouch]中拿到它的WCDataItem对象？

图9-23　把返回值置NULL的效果

9.2.8　从WCContentItemViewTemplateNew-Sight中提取WCDataItem对象

还记得在刚才的分析中是怎么拿到WCDataItem对象的吗？答案是通过getTimeline-DataItemOfIndex:

函数。回到图9-22中，看看这个函数的调用者和参数都是什么。

可以看到，它的调用者是getService:的返回值，参数是calcDataItemIndex:的返回值，如图9-24所示。

getService:和calcDataItemIndex:又要怎么调用呢？下面逐个来分析，先看getService:。它的调用者来自“MOV R0,R6”，即R6；R6来自[MMServiceCenter defaultCenter]的返回值。它的参数来自[WCFacade class]的返回值，如图9-25所示。

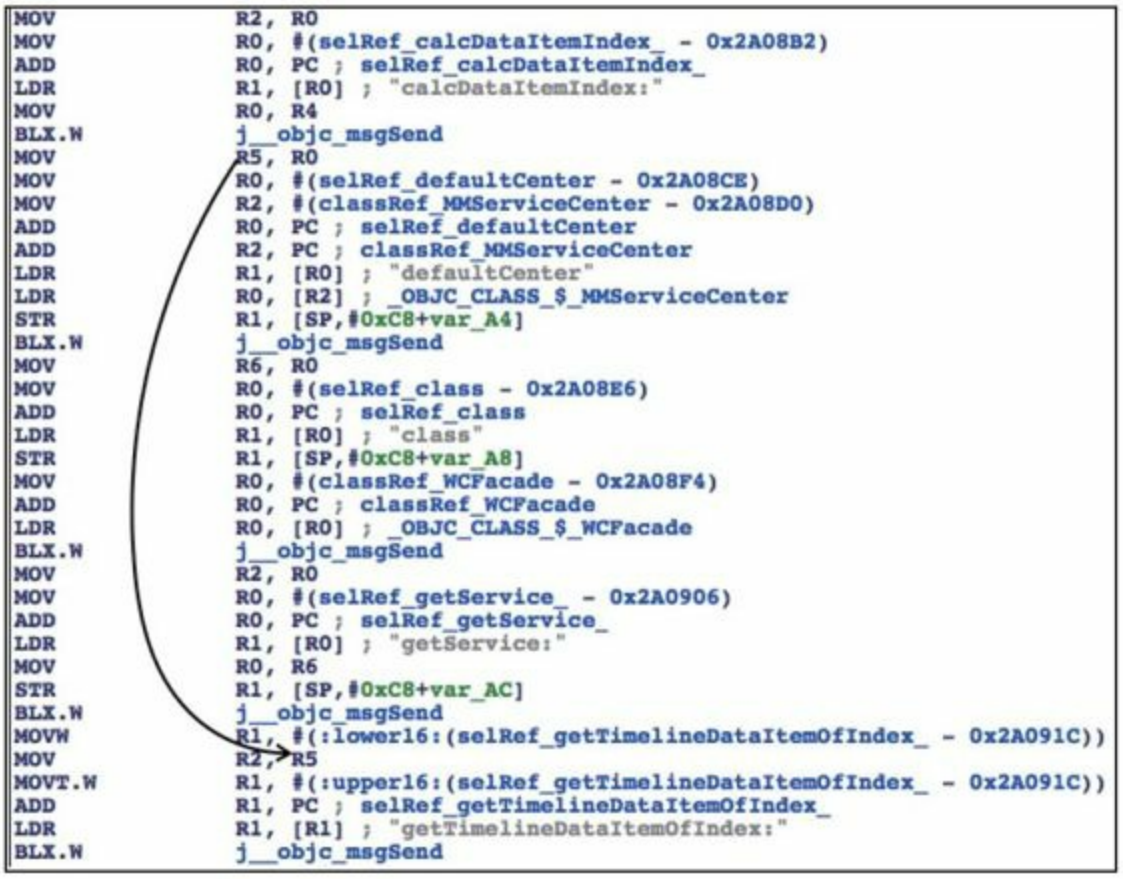


图9-24　解析getTimelineDataItemOfIndex:（1）

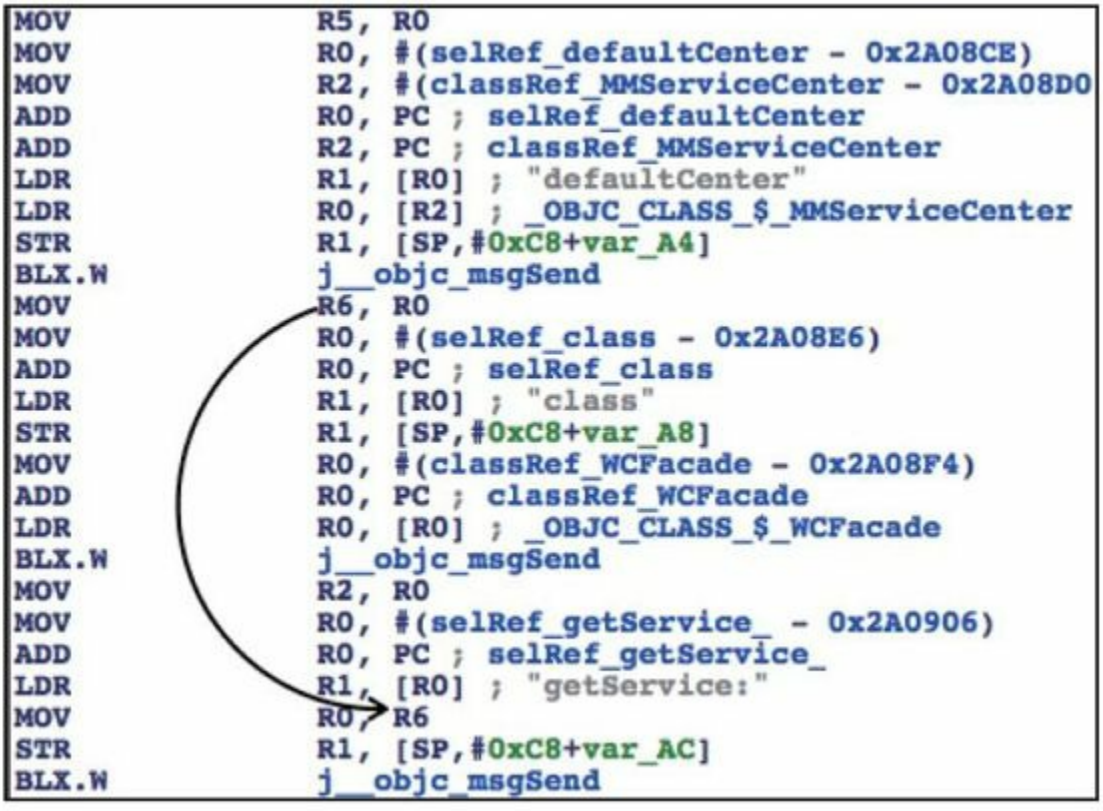


图9-25　解析getTimelineDataItemOfIndex:（2）

因此getTimelineDataItemOfIndex:的调用者可以通过[[MMServiceCenter defaultCenter]getService:[WCFacade class]]来获得。接着分析calcDataItemIndex:，它的调用者来自“MOV R0,R4”，即R4；而R4就是self。它的参数来自[indexPath section]的返回值，如图9-26和图9-27所示。

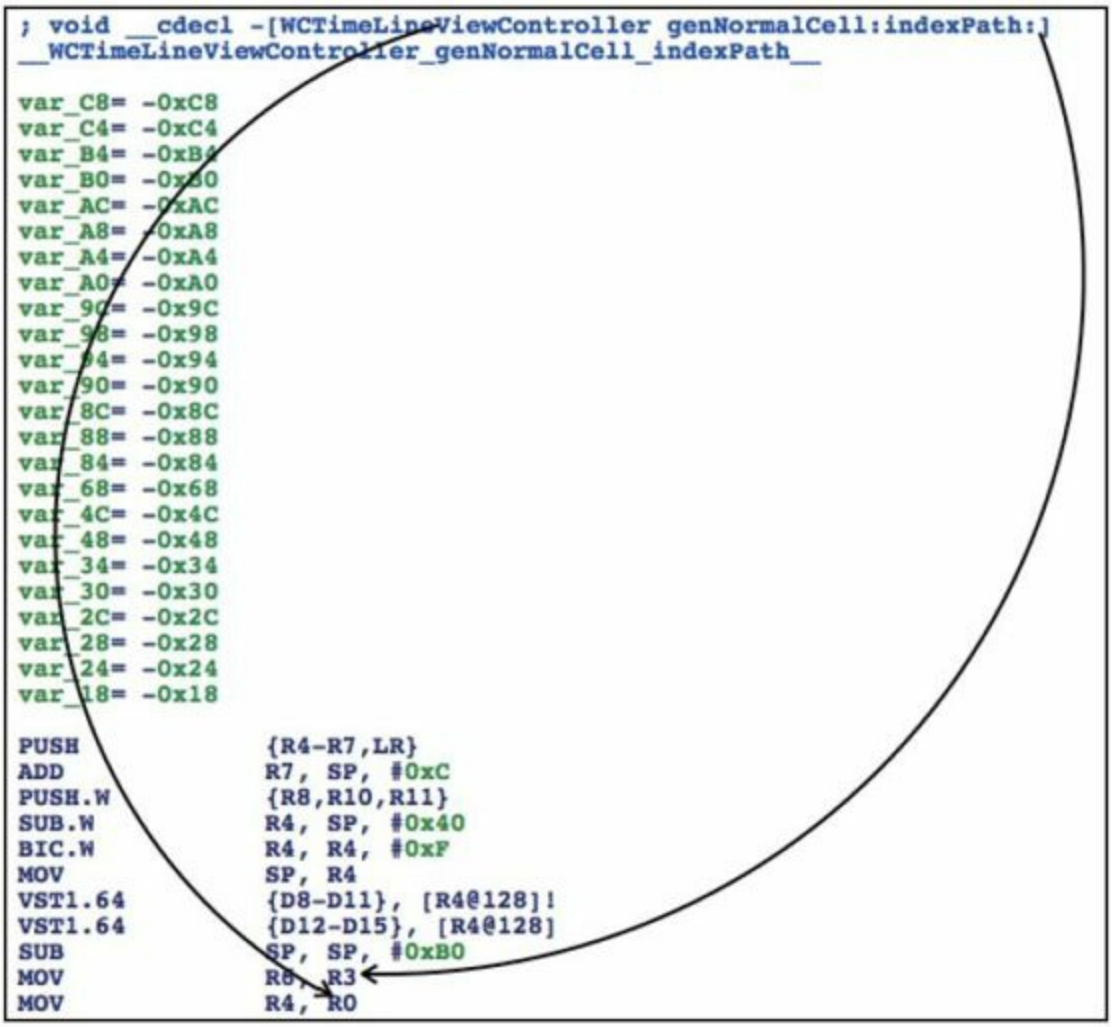


图9-26　解析getTimelineDataItemOfIndex:（3）

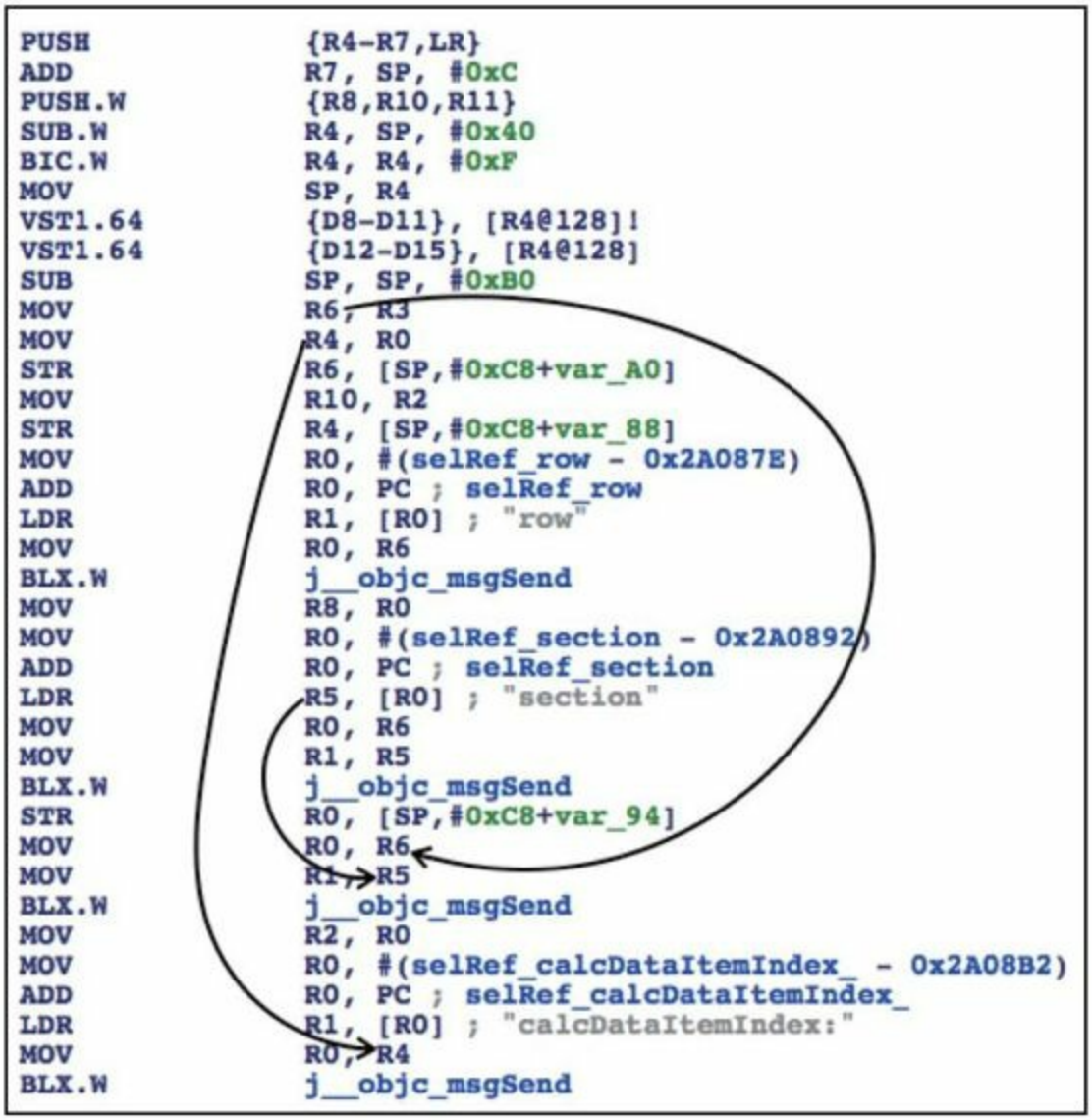


图9-27　解析getTimelineDataItemOfIndex:（4）

因此getTimelineDataItemOfIndex:的参数可以通过[WCTimeLineViewController calcDataItem-Index:[indexPath section]]获取。因为我们位于[WCContentItemViewTemplateNewSight onLongTouch]中，所以可以通过[self nextResponder]依次拿到

MMTableViewCell、MMTableView和WCTimeLineViewController，再通过[MMTableView indexPathForCell：MMTableViewCell]拿到indexPath，这个过程在9.2.6节已经得到了验证。虽然看起来有些麻烦，但至少通过符合MVC标准的方式从WCContentItemViewTemplateNewSight中成功提取了WCDataItem对象。值得一提的是，WCTimeLineViewController和WCContentItemViewTemplateNewSight的前缀是WC，笔者猜它是“WeChat”的缩写；而MMTableViewCell和MMTableView的前缀是MM，故而猜它是“MicroMessenger”的缩写——这种命名上的不统一，可能就是因为不同模块不同分工而造成的。接下来，重点剖析WCDataItem，把小视频的本地路径和下载地址从中提取出来。

9.2.9　从WCDataItem中提取目标信息

打开WCDataItem.h，大致浏览一下，如下：

```
@interface WCDataItem : NSObject <NSCoding>
{
    int cid;
    NSString *tid;
    int type;
    int flag;
    NSString *username;
    NSString *nickname;
    int createtime;
    NSString *sourceUrl;
    NSString *sourceUrl2;
    WCLocationInfo *locationInfo;
    BOOL isPrivate;
    NSMutableArray *sharedGroupIDs;
    NSMutableArray *blackUsers;
    NSMutableArray *visibleUsers;
    unsigned long extFlag;
    BOOL likeFlag;
    int likeCount;
    NSMutableArray *likeUsers;
    int commentCount;
    NSMutableArray *commentUsers;
    int withCount;
    NSMutableArray *withUsers;
    WCContentItem *contentObj;
    WCAppInfo *appInfo;
    NSString *publicUserName;
    NSString *sourceUserName;
    NSString *sourceNickName;
    NSString *contentDesc;
    NSString *contentDescPattern;
    int contentDescShowType;
    int contentDescScene;
    WCActionInfo *actionInfo;
    unsigned int hash;
    SnsObject *snsObject;
    BOOL isBidirectionalFan;
    BOOL noChange;
    BOOL isRichText;
    NSMutableDictionary *extData;
    int uploadErrType;
    NSString *statisticsData;
}
```

```
+ (id)fromBuffer:(id)arg1;
+ (id)fromServerObject:(id)arg1;
+ (id)fromUploadTask:(id)arg1;
@property(retain, nonatomic) WCActionInfo *actionInfo; //
@synthesize actionInfo;
@property(retain, nonatomic) WCAppInfo *appInfo; //
@synthesize appInfo;
@property(retain, nonatomic) NSArray *blackUsers; //
@synthesize blackUsers;
@property(nonatomic) int cid; // @synthesize cid;
@property(nonatomic) int commentCount; // @synthesize
commentCount;
@property(retain, nonatomic) NSMutableArray
*commentUsers; //
@synthesize commentUsers;
- (int)compareDesc:(id)arg1;
- (int)compareTime:(id)arg1;
@property(retain, nonatomic) NSString *contentDesc; //
@synthesize contentDesc;
@property(retain, nonatomic) NSString
*contentDescPattern; //
@synthesize contentDescPattern;
@property(nonatomic) int contentDescScene; // @synthesize
contentDescScene;
@property(nonatomic) int contentDescShowType; //
@synthesize
contentDescShowType;
@property(retain, nonatomic) WCContentItem *contentObj;
//
@synthesize contentObj;
@property(nonatomic) int createtime; // @synthesize
createtime;
- (void)dealloc;
- (id)description;
- (id)descriptionForKeyPaths;
- (void)encodeWithCoder:(id)arg1;
@property(retain, nonatomic) NSMutableDictionary
*extData; //
@synthesize extData;
@property(nonatomic) unsigned long extFlag; //
@synthesize extFlag;
@property(nonatomic) int flag; // @synthesize flag;
- (id)getDisplayCity;
- (id)getMediaWraps;
- (BOOL)hasSharedGroup;
- (unsigned int)hash;
- (id)init;
- (id)initWithCoder:(id)arg1;
@property(nonatomic) BOOL isBidirectionalFan; //
```

```
@synthesize isBidirectionalFan;
- (BOOL)isEqual:(id)arg1;
@property(nonatomic) BOOL isPrivate; // @synthesize
isPrivate;
- (BOOL)isRead;
@property(nonatomic) BOOL isRichText; // @synthesize
isRichText;
- (BOOL)isUploadFailed;
- (BOOL)isUploading;
- (BOOL)isValid;
- (id)itemID;
- (int)itemType;
- (id)keyPaths;
@property(nonatomic) int likeCount; // @synthesize
likeCount;
@property(nonatomic) BOOL likeFlag; // @synthesize
likeFlag;
@property(retain, nonatomic) NSMutableArray *likeUsers;
//
@synthesize likeUsers;
- (void)loadPattern;
@property(retain, nonatomic) WCLocationInfo
*locationInfo; //
@synthesize locationInfo;
- (void)mergeLikeUsers:(id)arg1;
- (void)mergeMessage:(id)arg1;
- (void)mergeMessage:(id)arg1 needParseContent:
(BOOL)arg2;
@property(retain, nonatomic) NSString *nickname; //
@synthesize nickname;
@property(nonatomic) BOOL noChange; // @synthesize
noChange;
- (void)parseContentForNetWithDataItem:(id)arg1;
- (void)parseContentForUI;
- (void)parsePattern;
@property(retain, nonatomic) NSString *publicUserName; //
@synthesize publicUserName;
- (id)sequence;
- (void)setCreateTime:(unsigned long)arg1;
- (void)setHash:(unsigned int)arg1;
- (void)setIsUploadFailed:(BOOL)arg1;
- (void)setSequence:(id)arg1;
@property(retain, nonatomic) NSMutableArray
*sharedGroupIDs; // @synthesize sharedGroupIDs;
@property(retain, nonatomic) SnsObject *snsObject; //
@synthesize snsObject;
@property(retain, nonatomic) NSString *sourceNickName; //
@synthesize sourceNickName;
```

```
@property(retain, nonatomic) NSString *sourceUrl2; //
@synthesize sourceUrl2;
@property(retain, nonatomic) NSString *sourceUrl; //
@synthesize sourceUrl;
@property(retain, nonatomic) NSString *sourceUserName; //
@synthesize sourceUserName;
@property(retain, nonatomic) NSString *statisticsData; //
@synthesize statisticsData;
@property(retain, nonatomic) NSString *tid; //
@synthesize tid;
@property(nonatomic) int type; // @synthesize type;
@property(nonatomic) int uploadErrType; // @synthesize
uploadErrType;
@property(retain, nonatomic) NSString *username; //
@synthesize username;
@property(retain, nonatomic) NSArray *visibleUsers; //
@synthesize visibleUsers;
@property(nonatomic) int withCount; // @synthesize
withCount;
@property(retain, nonatomic) NSMutableArray *withUsers;
//
@synthesize withUsers;
- (id)toBuffer;
@end
```

可以看到，文件中一共有4处出现了“path”和“url”关键词，如下：

```
- (id)descriptionForKeyPaths;
- (id)keyPaths;
@property(retain, nonatomic) NSString *sourceUrl2;
@property(retain, nonatomic) NSString *sourceUrl;
```

下面用LLDB来看看它们都会返回什么。重复刚才的操作，在小视频滑回来时触发断点，如下：

```
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208, queue = 'com.apple.main-thread, stop reason =
breakpoint 7.1
    frame #0: 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$Micro
Messenger +
208MicroMessenger`___lldb_unnamed_function11980$$MicroMes
 + 208:
-> 0x2a091c:  blx    0xe08e0c                  ;
___lldb_unnamed_function70162$$MicroMessenger
   0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$Micro
Messenger + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212:
-> 0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
   0x2a092a:  add    r0, pc
(lldb) po [$r0 descriptionForKeyPaths]
Class name: WCDataItem, addr: 0x80f52b0
tid: 11896185303680028954
username: wxid_hqouu9kgsgw3e6
createtime: 1418135798
commentUsers: (
)
contentObj: <WCContentItem: 0x8724c20>
(lldb) po [$r0 keyPaths]
<__NSArrayI 0x87b5260>(
tid,
username,
createtime,
commentUsers,
contentObj
)
(lldb) po [$r0 sourceUrl2]
 nil
(lldb) po [$r0 sourceUrl]
```

```
nil
```

这几个函数的返回值并没有让人眼前一亮的信息，但多次出现的WCContentItem却引起了笔者的注意。显然，“content”比“data”的含义更明确，小视频对象的信息有可能就是它提供的，下面来看看WCContentItem.h，如下：

```
@interface WCContentItem : NSObject <NSCoding>
{
    NSString *title;
    NSString *desc;
    NSString *titlePattern;
    NSString *descPattern;
    NSString *linkUrl;
    NSString *linkUrl2;
    int type;
    int flag;
    NSString *username;
    NSString *nickname;
    int createtime;
    NSMutableArray *mediaList;
}
@property(nonatomic) int createtime; // @synthesize
createtime;
- (void)dealloc;
@property(retain, nonatomic) NSString *desc; //
@synthesize desc;
@property(retain, nonatomic) NSString *descPattern; //
@synthesize descPattern;
- (void)encodeWithCoder:(id)arg1;
@property(nonatomic) int flag; // @synthesize flag;
- (id)init;
- (id)initWithCoder:(id)arg1;
- (BOOL)isValid;
@property(retain, nonatomic) NSString *linkUrl; //
@synthesize linkUrl;
```

```
@property(retain, nonatomic) NSString *linkUrl2; //
@synthesize linkUrl2;
@property(retain, nonatomic) NSMutableArray *mediaList;
//
@synthesize mediaList;
@property(retain, nonatomic) NSString *nickname; //
@synthesize nickname;
@property(retain, nonatomic) NSString *title; //
@synthesize title;
@property(retain, nonatomic) NSString *titlePattern; //
@synthesize titlePattern;
@property(nonatomic) int type; // @synthesize type;
@property(retain, nonatomic) NSString *username; //
@synthesize username;
@end
```

可以看到，文件中一共有2处出现了“url”关键词，如下：

```
@property(retain, nonatomic) NSString *linkUrl;
@property(retain, nonatomic) NSString *linkUrl2;
```

通过[WCDataItem contentObj]函数可以获取其对应的WCContentItem对象，我们用LLDB看看上面2个property的值。重复刚才的操作，在小视频滑回来时触发断点，如下：

```
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208, queue = 'com.apple.main-thread, stop reason =
breakpoint 7.1
```

```
    frame #0: 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$Micro
Messenger + 208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208:
-> 0x2a091c:  blx    0xe08e0c                  ;
___lldb_unnamed_function70162$$MicroMessenger
   0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$Micro
Messenger + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212:
-> 0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
   0x2a092a:  add    r0, pc
(lldb) po [$r0 descriptionForKeyPaths]
Class name: WCDataItem, addr: 0x80f52b0
tid: 11896185303680028954
username: wxid_hqouu9kgsgw3e6
createtime: 1418135798
commentUsers: (
)
contentObj: <WCContentItem: 0x8724c20>
(lldb) po [$r0 keyPaths]
<__NSArrayI 0x87b5260>(
tid,
username,
createtime,
commentUsers,
contentObj
)
(lldb) po [$r0 sourceUrl2]
 nil
(lldb) po [$r0 sourceUrl]
 nil
```

接下来在浏览器里输入这个url，看看对应的是个什么东西，如图9-28所示。

图9-28　[[$r0 contentObj]linkUrl]

跟我们想要的结果驴唇不对马嘴。

WCContentItem.h里的内容本来就不多，小视频会藏

在哪里呢？回看这个文件，一个名为mediaList的property引起了笔者的注意。相对于“content”、“media”的定位更精确了，小视频会不会藏在它里面呢？还是用LLDB测一测。重复刚才的操作，在小视频滑回来时触发断点，如下：

```
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`__lldb_unnamed_function11980$$MicroMesseng
 + 208, queue ='com.apple.main-thread, stop reason =
breakpoint 8.1
    frame #0: 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$Micro
Messenger + 208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208:
-> 0x2a091c:  blx    0xe08e0c                  ;
___lldb_unnamed_function70162$$MicroMessenger
   0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212:
-> 0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
   0x2a092a:  add    r0, pc
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
pathForData]
```

```
/var/mobile/Containers/Data/Application/E9BE84D8-9982-
4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e

(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
pathForPreview]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-
4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e
 95feda4bed05f9b82
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
pathForSightData]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-
4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e

(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
dataUrl]
type[1],
url[http://vcloud1023.tc.qq.com/1023_0114929ce86949a8bfb6

(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
lowBandUrl]
 nil
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
previewUrls]
<__NSArrayM 0x8725950>(
type[1],
url[http://mmsns.qpic.cn/mmsns/WiaWbRORjpHsUXcNL3dNsVLDib

)
```

此时，一个新的类WCMediaItem出现了。下面看看它的头文件WCMediaItem.h，如下：

```
@interface WCMediaItem : NSObject <NSCoding>
{
    NSString *mid;
    int type;
    int subType;
    NSString *title;
    NSString *desc;
```

```
    NSString *titlePattern;
    NSString *descPattern;
    NSString *userData;
    NSString *source;
    NSMutableArray *previewUrls;
    WCUrl *dataUrl;
    WCUrl *lowBandUrl;
    struct CGSize imgSize;
    BOOL likeFlag;
    int likeCount;
    NSMutableArray *likeUsers;
    int commentCount;
    NSMutableArray *commentUsers;
    int withCount;
    NSMutableArray *withUsers;
    int createTime;
}
- (id).cxx_construct;
- (id)cityForData;
@property(nonatomic) int commentCount; // @synthesize
comentCount;
@property(retain, nonatomic) NSMutableArray
*commentUsers; //
@synthesize commentUsers;
- (id)comparativePathForPreview;
@property(nonatomic) int createTime; // @synthesize
createTime;
@property(retain, nonatomic) WCUrl *dataUrl; //
@synthesize dataUrl;
- (void)dealloc;
@property(retain, nonatomic) NSString *desc; //
@synthesize desc;
@property(retain, nonatomic) NSString *descPattern; //
@synthesize descPattern;
- (void)encodeWithCoder:(id)arg1;
- (BOOL)hasData;
- (BOOL)hasPreview;
- (BOOL)hasSight;
- (id)hashPathForString:(id)arg1;
- (id)imageOfSize:(int)arg1;
@property(nonatomic) struct CGSize imgSize; //
@synthesize imgSize;
- (id)init;
- (id)initWithCoder:(id)arg1;
- (BOOL)isValid;
@property(nonatomic) int likeCount; // @synthesize
likeCount;
@property(nonatomic) BOOL likeFlag; // @synthesize
likeFlag;
```

```
@property(retain, nonatomic) NSMutableArray *likeUsers;
//
@synthesize likeUsers;
- (CDStruct_c3b9c2ee)locationForData;
@property(retain, nonatomic) WCUrl *lowBandUrl; //
@synthesize lowBandUrl;
- (id)mediaID;
- (int)mediaType;
@property(retain, nonatomic) NSString *mid; //
@synthesize mid;
- (id)pathForData;
- (id)pathForPreview;
- (id)pathForSightData;
@property(retain, nonatomic) NSMutableArray *previewUrls;
//
@synthesize previewUrls;
- (BOOL)saveDataFromData:(id)arg1;
- (BOOL)saveDataFromMedia:(id)arg1;
- (BOOL)saveDataFromPath:(id)arg1;
- (BOOL)savePreviewFromData:(id)arg1;
- (BOOL)savePreviewFromMedia:(id)arg1;
- (BOOL)savePreviewFromPath:(id)arg1;
- (BOOL)saveSightDataFromData:(id)arg1;
- (BOOL)saveSightDataFromMedia:(id)arg1;
- (BOOL)saveSightDataFromPath:(id)arg1;
- (BOOL)saveSightPreviewFromMedia:(id)arg1;
@property(retain, nonatomic) NSString *source; //
@synthesize source;
@property(nonatomic) int subType; // @synthesize subType;
@property(retain, nonatomic) NSString *title; //
@synthesize title;
@property(retain, nonatomic) NSString *titlePattern; //
@synthesize titlePattern;
@property(nonatomic) int type; // @synthesize type;
@property(retain, nonatomic) NSString *userData; //
@synthesize userData;
@property(nonatomic) int withCount; // @synthesize
withCount;
@property(retain, nonatomic) NSMutableArray *withUsers;
//
@synthesize withUsers;
- (id)videoStreamForData;
- (id)voiceStreamForData;
@end
```

可以看到，在头文件中出现了8次“path”关键词，如下：

```
- (id)comparativePathForPreview;
- (id)hashPathForString:(id)arg1;
- (id)pathForData;
- (id)pathForPreview;
- (id)pathForSightData;
- (BOOL)saveDataFromPath:(id)arg1;
- (BOOL)savePreviewFromPath:(id)arg1;
- (BOOL)saveSightDataFromPath:(id)arg1;
```

还有3次“url”关键词，如下：

```
@property(retain, nonatomic) WCUrl *dataUrl;
@property(retain, nonatomic) WCUrl *lowBandUrl;
@property(retain, nonatomic) NSMutableArray *previewUrls;
```

其中，pathForData、pathForPreview和pathForSightData极有可能返回一个path；dataUrl、lowBandUrl和previewUrls极有可能返回url，马上用LLDB看看这些返回值是什么。重复刚才的操作，在小视频滑回来时触发断点，如下：

```
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208, queue = 'com.apple.main-thread, stop reason =
```

```
breakpoint 8.1
    frame #0: 0x002a091c
MicroMessenger`___lldb_unnamed_function11980$$Micro
Messenger + 208
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 208:
-> 0x2a091c:  blx    0xe08e0c                  ;
___lldb_unnamed_function70162$$MicroMessenger
   0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
(lldb) ni
Process 184500 stopped
* thread #1: tid = 0x2d0b4, 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x002a0920
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212
MicroMessenger`___lldb_unnamed_function11980$$MicroMessen
 + 212:
-> 0x2a0920:  mov    r11, r0
   0x2a0922:  movw   r0, #32442
   0x2a0926:  movt   r0, #436
   0x2a092a:  add    r0, pc
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
pathForData]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-
4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e

(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
pathForPreview]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-
4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e
 95feda4bed05f9b82
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
pathForSightData]
/var/mobile/Containers/Data/Application/E9BE84D8-9982-
4814-9289-
823D5FD91144/Library/WechatPrivate/c5f5eb23e53bb2ee021b0e

(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
dataUrl]
type[1],
url[http://vcloud1023.tc.qq.com/1023_0114929ce86949a8bfb6
```

```
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
lowBandUrl]
 nil
(lldb) po [[[[$r0 contentObj] mediaList] objectAtIndex:0]
previewUrls]
<__NSArrayM 0x8725950>(
type[1],
url[http://mmsns.qpic.cn/mmsns/WiaWbRORjpHsUXcNL3dNsVLDib

)
```

从文件名就可以看出，这些应该就是我们要找的小视频信息了。不管你是直接在ssh中操作，还是用iFunBox浏览本地文件；不管你是用MobileSafari，还是用Chrome打开URL，都可以得出以下结论：

· pathForData返回小视频的本地路径，不带后缀名；

· pathForPreview返回小视频的预览图片路径，没有后缀名；

· pathForSightData返回小视频的本地路径，带后缀名；

· dataUrl返回小视频的网络URL；

· lowBandUrl返回nil，笔者猜测，当网络状况不好时，它的值不为nil；为了节省带宽，这个URL对应的mp4文件很可能比dataUrl对应的文件要小；

· previewUrls返回小视频的预览图URL。

tweak原型搭建到此结束，下面先整理一下思路，再开始写代码。

9.3 逆向结果整理

本章的实例综合运用了Cycript、IDA和LLDB工具，在没有严格推导微信代码逻辑的情况下完成了tweak的原型搭建工作。大致的分析思路是这样的。

1.在小视频播放窗口添加长按手势

因为微信本身已经在小视频播放窗口添加了长按手势，所以没有必要重新发明轮子，只需要找到长按手势的响应函数，然后hook它就可以了。用Reveal很容易就可定位小视频播放窗口，从而找到长按手势的响应函数。值得一提的是，找到的函数在长按后会被连续调用2次，导致相同的代码重复执行，效率不高。在撰写tweak的过程中要考虑到这种情况，用简单的条件判断把2次重复调用简化为1次调用。

2.在C里寻找小视频对象

虽然MVC设计标准里约定了可以通过C访问M，

但是在本例中，C里并没有出现比较明显的访问M的方法。因此本章从最基本的tableView:cellForRowAtIndexPath:数据源函数入手，在IDA里找到了可疑的cell数据源，并通过观察头文件的方式定位到小视频对象，最终提取出了想要的信息。或许不那么严谨，但是在达到目标的基础上节省了时间，这个结果也还不错！

9.4 编写tweak

本节的目标是把长按小视频播放窗口的菜单选项给替换成“Save to Disk”和“Copy URL”，并实现相应动作。有了9.3节的原型作为铺垫，这一节就没什么太多可解释的了，咱们直接动手吧。

9.4.1 用Theos新建tweak工程“iOSREWCVideoDownloader”

新建iOSREWCVideoDownloaders工程的命令如下：

```
hangcom-mba:Documents sam$ /opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
  [4.] iphone/library
  [5.] iphone/notification_center_widget
  [6.] iphone/preference_bundle
  [7.] iphone/sbsettingstoggle
  [8.] iphone/tool
  [9.] iphone/tweak
  [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): iOSREWCVideoDownloader
```

```
Package Name [com.yourcompany.iosrewcvideodownloader]:
com.iosre.iosrewcvideodownloader
Author/Maintainer Name [sam]: sam
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.tencent.xin
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]: MicroMessenger
Instantiating iphone/tweak in iosrewcvideodownloader/...
 Done.
```

9.4.2 构造iOSREWCVideoDownloader.h

编辑后的iOSREWCVideoDownloader.h内容如下：

```
@interface WCContentItem : NSObject
@property (retain, nonatomic) NSMutableArray *mediaList;
@end
@interface WCDataItem : NSObject
@property (retain, nonatomic) WCContentItem *contentObj;
@end
@interface WCUrl : NSObject
@property (retain, nonatomic) NSString *url;
@end
@interface WCMediaItem : NSObject
@property (retain, nonatomic) WCUrl *dataUrl;
- (id)pathForSightData;
@end
@interface WCContentItemViewTemplateNewSight : UIView
- (WCMediaItem *)iOSREMediaItemFromSight;
- (void)iOSREOnSaveToDisk;
- (void)iOSREOnCopyURL;
@end
@interface MMServiceCenter : NSObject
+ (id)defaultCenter;
- (id)getService:(Class)arg1;
@end
@interface WCFacade : NSObject
```

```
- (WCDataItem *)getTimelineDataItemOfIndex:(int)arg1;
@end
@interface WCTimeLineViewController : NSObject
- (int)calcDataItemIndex:(int)arg1;
@end
@interface MMTableViewCell : UITableViewCell
@end
@interface MMTableView : UITableView
@end
```

这个头文件的所有内容均摘自类对应的头文件，构造它的目的仅仅是通过编译，避免出现任何报错信息和警告。

9.4.3 编辑Tweak.xm

编辑后的Tweak.xm内容如下：

```
#import "iOSREWCVideoDownloader.h"
static MMTableViewCell *iOSRECell;
static MMTableView *iOSREView;
static WCTimeLineViewController *iOSREController;
%hook WCContentItemViewTemplateNewSight
%new
- (WCMediaItem *)iOSREMediaItemFromSight
{
      id responder = self;
      while (![responder
isKindOfClass:NSClassFromString(@"WCTimeLineViewControlle

      {
            if ([responder
isKindOfClass:NSClassFromString(@"MMTableViewCell")])
iOSRECell = responder;
            else if ([responder
```

```
isKindOfClass:NSClassFromString(@"MMTableView")])
iOSREView = responder;
            responder = [responder nextResponder];
      }
      iOSREController = responder;
      if (!iOSRECell || !iOSREView || !iOSREController)
      {
            NSLog(@"iOSRE: Failed to get video object.");
            return nil;
      }
      NSIndexPath *indexPath = [iOSREView
indexPathForCell:iOSRECell];
      int itemIndex = [iOSREController calcDataItemIndex:
[indexPath section]];
      WCFacade *facade = [(MMServiceCenter *)
[%c(MMServiceCenter) defaultCenter] getService:
[%c(WCFacade) class]];
      WCDataItem *dataItem = [facade
getTimelineDataItemOfIndex:itemIndex];
      WCContentItem *contentItem = dataItem.contentObj;
      WCMediaItem *mediaItem = [contentItem.mediaList
count] != 0 ? (contentItem.mediaList)[0] : nil;
      return mediaItem;
}
%new
- (void)iOSREOnSaveToDisk
{
      NSString *localPath = [[self
iOSREMediaItemFromSight] pathForSightData];
      UISaveVideoAtPathToSavedPhotosAlbum(localPath, nil,
nil, nil);
}
%new
- (void)iOSREOnCopyURL
{
      UIPasteboard *pasteboard = [UIPasteboard
generalPasteboard];
      pasteboard.string = [self
iOSREMediaItemFromSight].dataUrl.url;
}
static int iOSRECounter;
- (void)onLongTouch
{
      iOSRECounter++;
      if (iOSRECounter % 2 == 0) return;
      [self becomeFirstResponder];
      UIMenuItem *saveToDiskMenuItem = [[UIMenuItem
alloc] initWithTitle:@"Save to Disk"
action:@selector(iOSREOnSaveToDisk)];
```

```
        UIMenuItem *copyURLMenuItem = [[UIMenuItem alloc]
initWithTitle:@"Copy URL"
action:@selector(iOSREOnCopyURL)];
        UIMenuController *menuController =
[UIMenuController sharedMenuController];
        [menuController setMenuItems:@[saveToDiskMenuItem,
copyURLMenuItem]];
        [menuController setTargetRect:CGRectZero
inView:self];
        [menuController setMenuVisible:YES animated:YES];
        [saveToDiskMenuItem release];
        [copyURLMenuItem release];
}
%end
```

9.4.4 编辑Makefile及control

编辑后的Makefile内容如下：

```
THEOS_DEVICE_IP = iOSIP
TARGET = iphone:latest:8.0
ARCHS = armv7 arm64
include theos/makefiles/common.mk
TWEAK_NAME = iOSREWCVideoDownloader
iOSREWCVideoDownloader_FILES = Tweak.xm
iOSREWCVideoDownloader_FRAMEWORKS = UIKit
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
        install.exec "killall -9 MicroMessenger"
```

编辑后的control内容如下：

```
Package: com.iosre.iosrewcvideodownloader
Name: iOSREWCVideoDownloader
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
```

```
Architecture: iphoneos-arm
Description: Play with Sight!
Maintainer: sam
Author: sam
Section: Tweaks
Homepage: http://bbs.iosre.com
```

9.4.5 测试

将写好的tweak编译打包安装到iOS后，打开微信，长按小视频播放窗口，就会弹出自定义菜单，如图9-29所示。

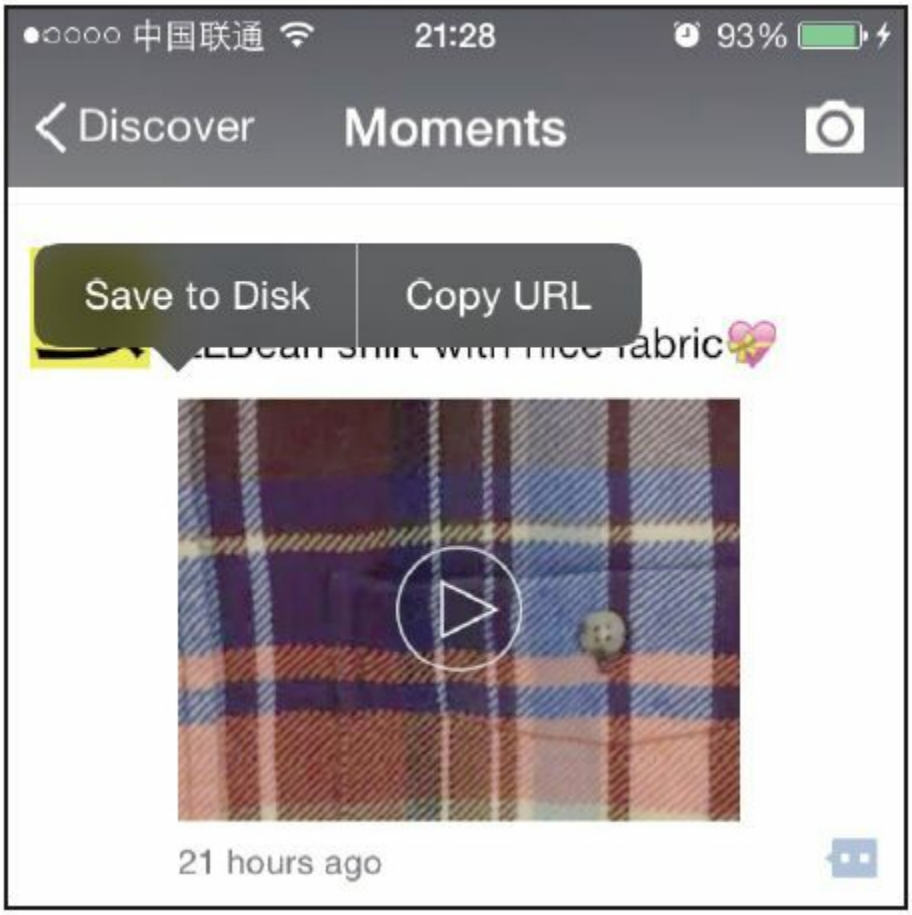


图9-29　自定义菜单

点击“Save to Disk”，这个小视频会被保存到相册中，如图9-30所示。

点击“Copy URL”，然后到OPlayer Lite（或任意支持在线视频播放的App）中打开这个网址，如图9-31所示。

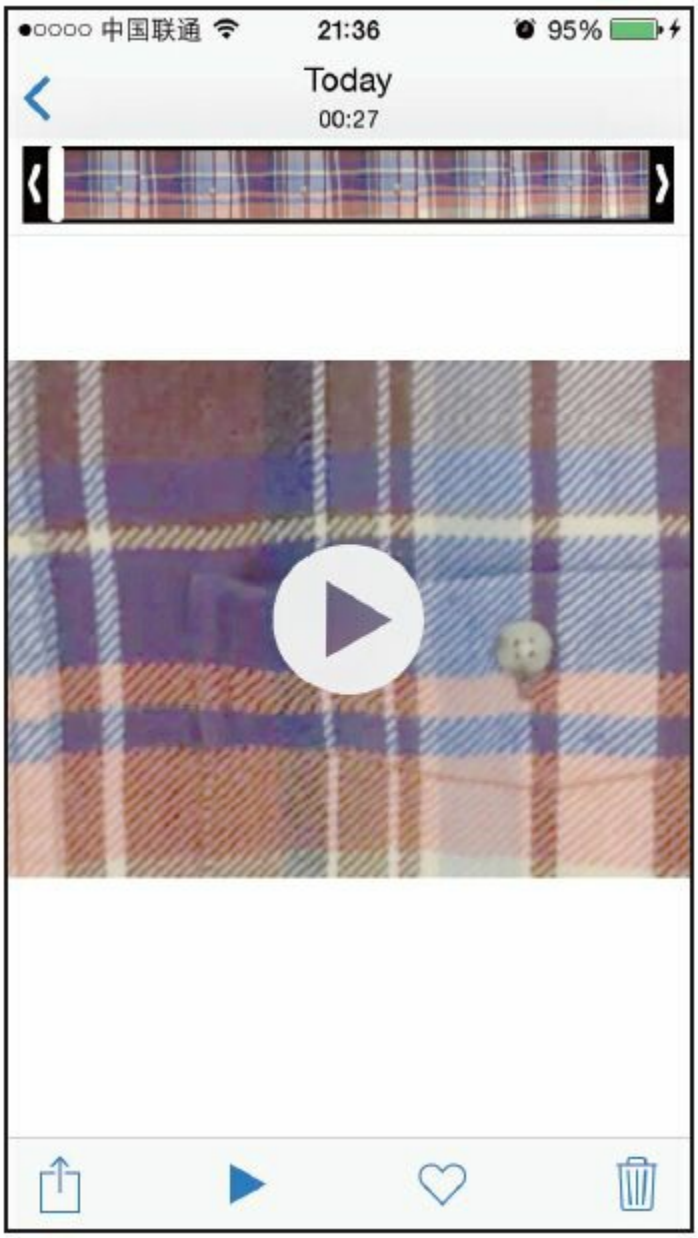


图9-30　把小视频保存到相册

图9-31　在OPlayer Lite中播放在线mp4

所有功能均可正常工作，本章任务达成！

9.5 彩蛋放送

9.5.1 从UIMenuItem切入，找到小视频对象

在9.2.7节中，从WCTimeLineViewController切入，找到了小视频对象。但这个过程并不顺利，因为没有找到通过C直接访问M的方法，所以才“不得已”从tableView:cellForRowAtIndexPath:中寻找小视频对象的线索，最终达到了目标。如果跳出MVC的通用思路，从微信本身的角度考虑问题，事情或许可以简单得多。

一起来思考：长按小视频播放窗口，会出现菜单。选择菜单中的选项，会对小视频做出相应的操作——也就是说，UIMenuItem的action中可能会有小视频对象的线索。在图9-11中，我们曾看到过“Favorite”选项的响应函数，即onFavoriteAdd:，那就去IDA中看看它的实现是怎样的吧，如图9-32所

示。

```
; WCContentItemViewTemplateNewSight - (void)onFavoriteAdd:(id)
; Attributes: bp-based frame

; void __cdecl -[WCContentItemViewTemplateNewSight onFavoriteAdd:]
__WCContentItemViewTemplateNewSight_onFavoriteAdd__

var_28= -0x28
var_24= -0x24
var_20= -0x20
var_1C= -0x1C

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
PUSH.W          {R8,R10,R11}
SUB             SP, SP, #0x10
MOV             R10, R0
MOV             R0, #(off_1A002FC - 0x21EAF4) ; off_1A002FC
MOVW            R1, #(:lower16:(selRef_contentObj - 0x21EAFA))
ADD             R0, PC ; off_1A002FC
MOVT.W          R1, #(:upper16:(selRef_contentObj - 0x21EAFA))
ADD             R1, PC ; selRef_contentObj
LDR             R0, [R0] ; WCDataItem *_oDataItem;
LDR             R1, [R1] ; "contentObj"
LDR             R4, [R0]
LDR.W           R0, [R10,R4]
BLX.W           j__objc_msgSend
MOV             R5, R0
MOV             R0, #(selRef_mediaList - 0x21EB14) ; selRef_mediaL
ADD             R0, PC ; selRef_mediaList
LDR             R6, [R0] ; "mediaList"
MOV             R0, R5
MOV             R1, R6
BLX.W           j__objc_msgSend
```

图9-32　[WCContentItemViewTemplateNewSight onFavoriteAdd:]

从图9-32可以看到，这个函数的开头部分出现了我们熟悉的WCDataItem、contentObj和mediaList。如果当时从这个函数入手，整个分析过程的工作量至少可以减轻一半。看来，虽然以MVC标准为线索从App

切入代码是一条通用的思路，但打破常规往往能取得意想不到的效果，让iOS逆向工程变得更好玩。

9.5.2 微信历史版本头文件个数变迁

从微信历代版本头文件数量的变迁中（如图9-33到图9-38所示）可以看出，微信是如何一步步迈向优秀的。不积跬步无以至千里，向微信致敬。

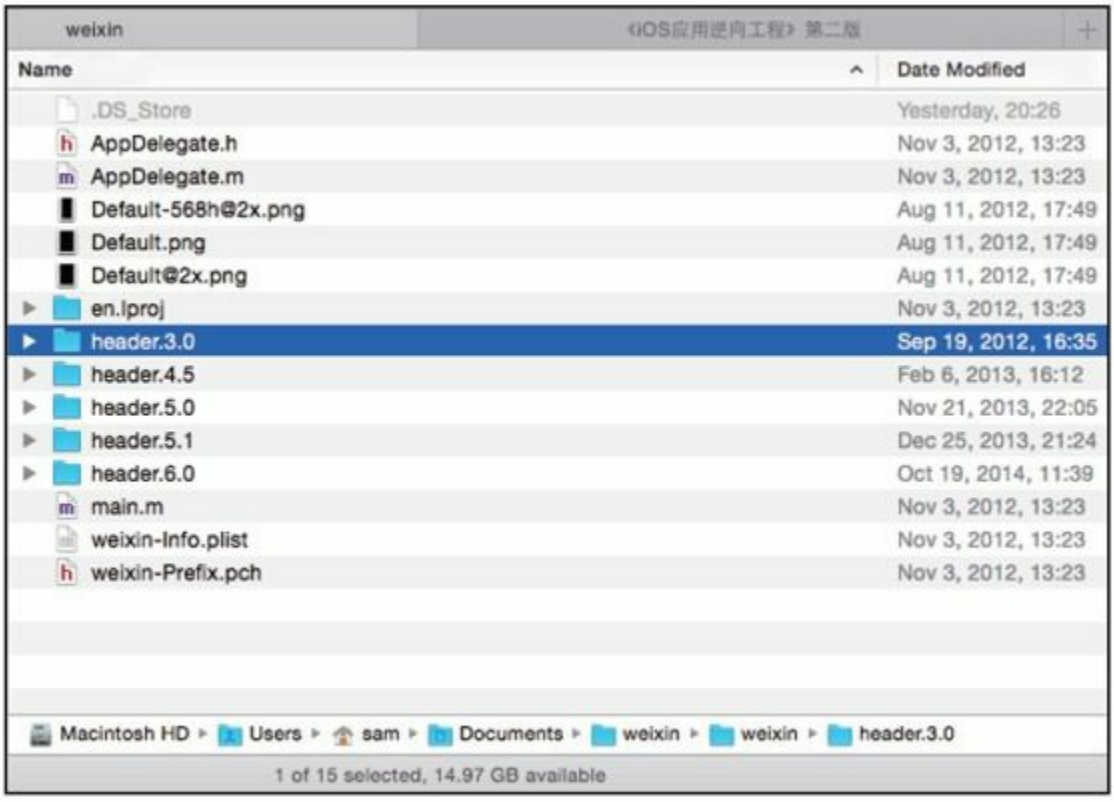


图9-33 微信不同版本的头文件目录

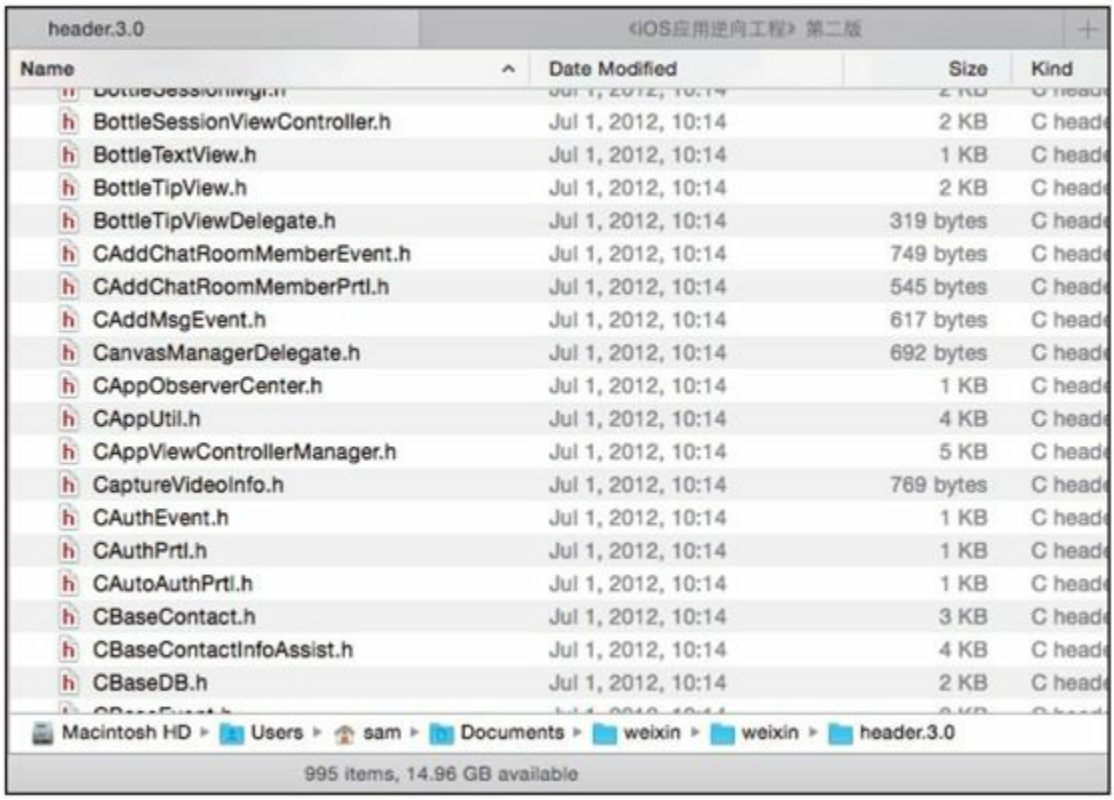


图9-34　微信3.0，995个头文件

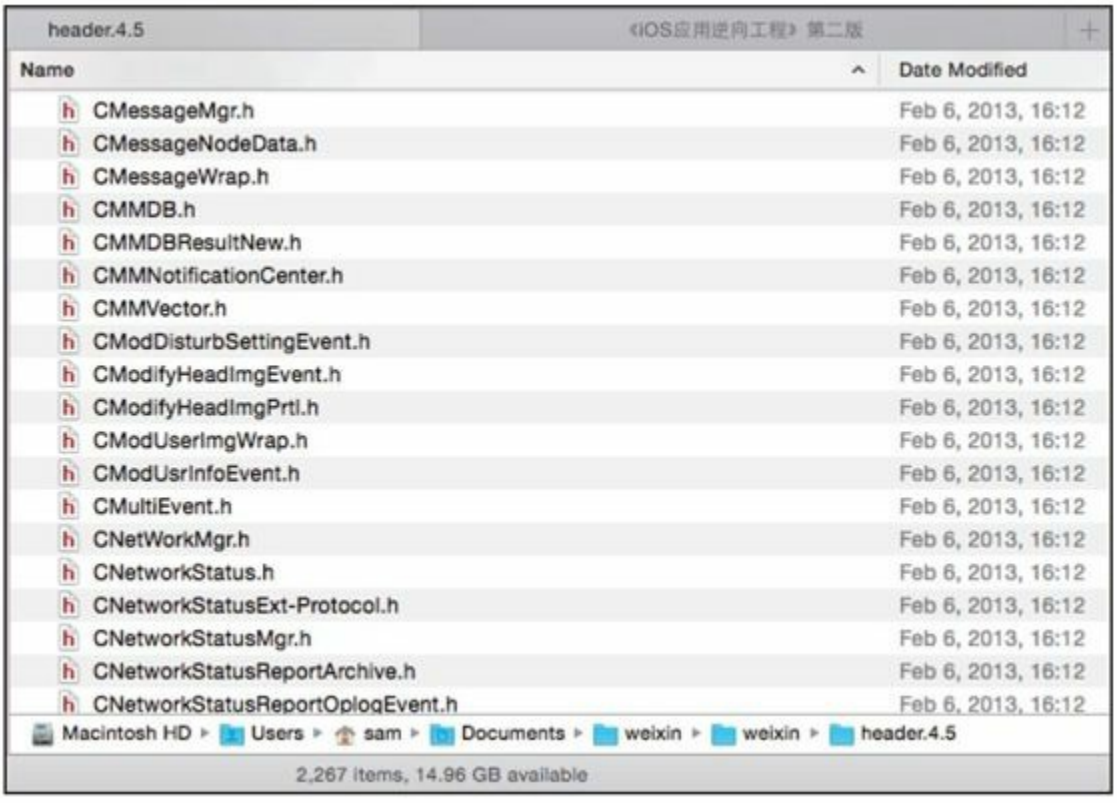


图9-35　微信4.5，2267个头文件

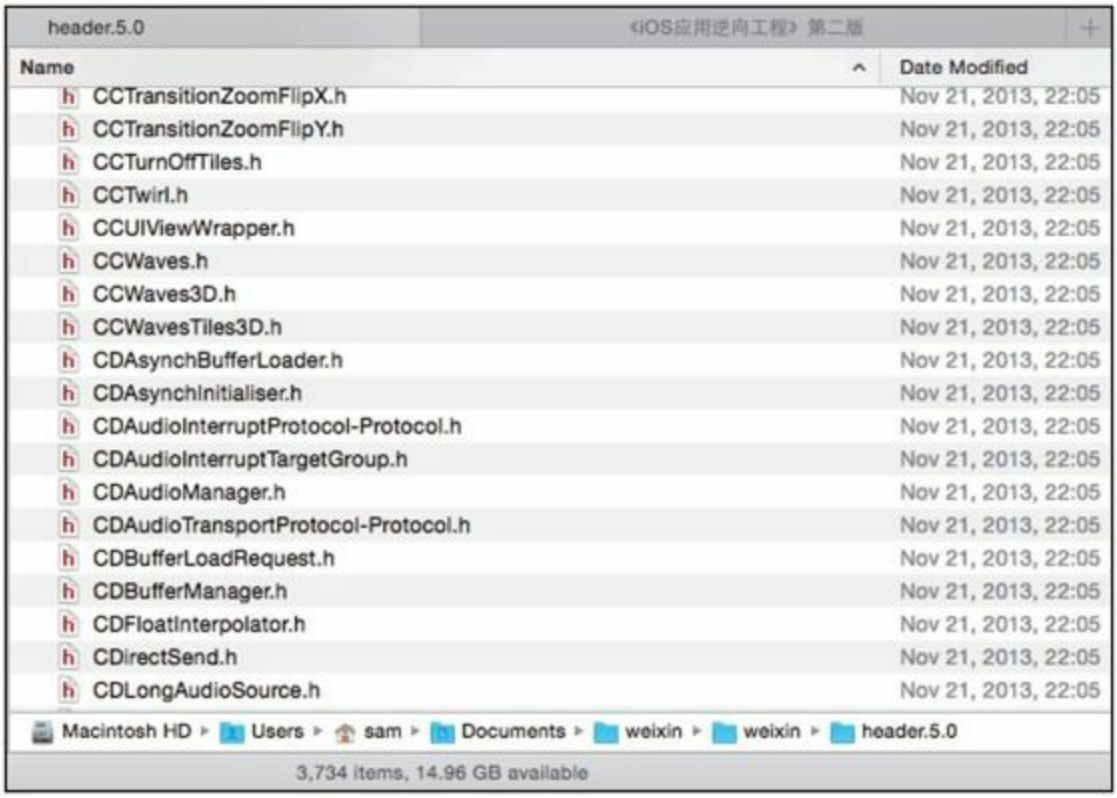


图9-36　微信5.0，3734个头文件

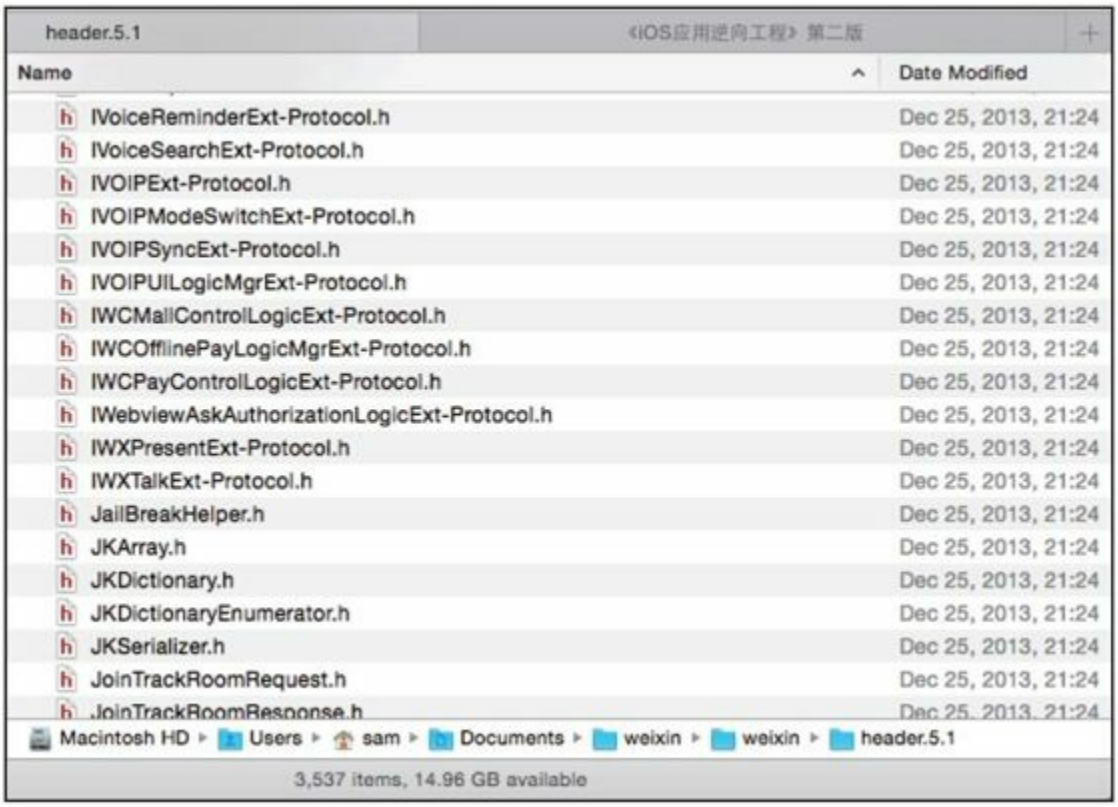


图9-37　微信5.1，3537个头文件，比5.0版还少几个

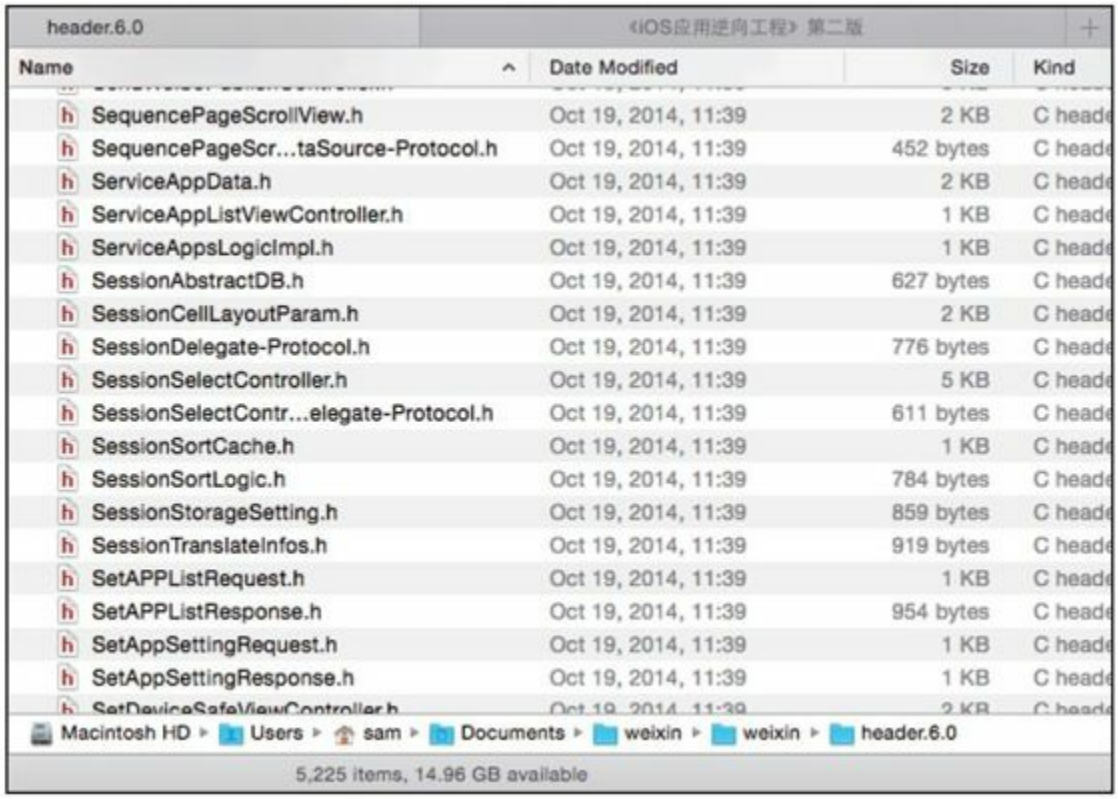


图9-38　微信6.0，5225个头文件

经历3、4、5、6这4个大版本的迭代，微信的头文件个数从最初的不足1000到现在突破5000，增加了5倍有余。随着微信在全球范围内的普及，App头文件个数突破10000已经指日可待了。

9.6 小结

本章以微信为目标，给小视频功能添加了保存到本地和复制URL的功能，丰富了小视频的玩法和传播渠道。微信作为一个功能强大的平台，结构复杂、代码量大；它的架构设计非常考究，模块划分非常清晰，仅仅浏览头文件就能借鉴学习很多经验，甚至连编码风格都能看出各种年代的程序员的痕迹。建议大家用逆向工程的方式深入了解微信的设计理念，相信你会受益匪浅；我们会在http://bbs.iosre.com上讨论交流对微信的研究心得，欢迎关注。

第10章　实战4：检测与发送iMessage

10.1　iMessage

iMessage是苹果公司无缝整合在系统原生信息应用（以下简称MobileSMS）内的即时通信服务，其出生于iOS 5，壮大于iOS 8，无论是文字传输、图片展示、语音还是视频播放，都能够保证安全、节能、快速而且高效。我们都爱iMessage！

在iMessage的所有功能中，“检测对方是否支持iMessage通信”及“发送iMessage”的功能想必是大家最感兴趣的目标，没有之一。我国甚至出现了大批以发送iMessage广告为业务的公司，有的已经赚得盆满钵满，但给iMessage用户带来的骚扰却无人买单，而这也是笔者出品Cydia应用“SMSNinja”的主要原因之一。知道怎么攻击，才能了解如何防守，知己知彼方能百战不殆，本章就结合前面章节的所有知识点，从

零开始逆向出检测与发送iMessage的功能，作为全书案例的总结。以下的操作在iPhone 5，iOS 8.1中完成。

10.2 检测一个号码或邮箱地址是否支持iMessage

按照惯例，在使用工具开始逆向工程之前，要先分析一下抽象的目标，并将其具体化，然后制定这次逆向工程的思路，再贯彻思想实地执行。

10.2.1 观察MobileSMS界面元素的变化，寻找逆向切入点

使用MobileSMS的朋友都会注意到，在发送一条信息的整个过程中，苹果都会通过文字及颜色的变化，来提示用户当前发送的是一条短信（以下简称SMS）还是一条iMessage，具体表现在以下几方面。

A.当开始编写一条信息，刚输入完对方的地址，还没有输入信息内容时，如果iOS检测到对方支持iMessage，信息输入框处的占位符（placeholder）就

会由“Text Message”变成“iMessage”，如图10-1所示。

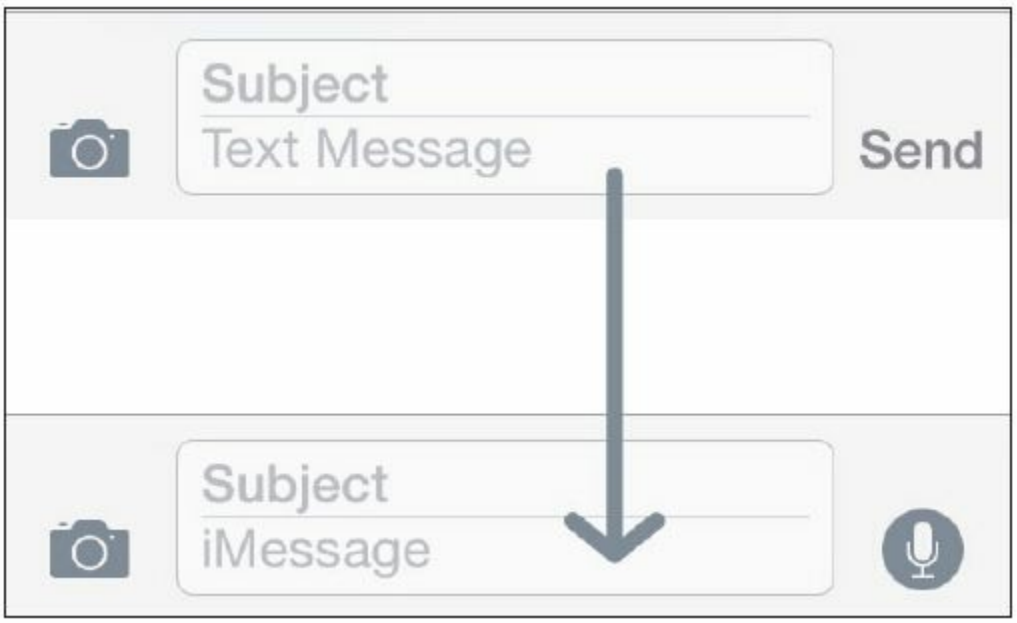


图10-1　placeholder的变化

B.当输入信息内容时，如果对方仅支持SMS，则输入框旁的“Send”字样是绿色的；如果对方支持iMessage，则“Send”是蓝色的；

C.当点击“Send”发送此条信息时，如果这是一条SMS，信息气泡的颜色是绿色的；如果是一条iMessage，气泡是蓝色的。

这3种现象会依次出现，不过，因为检测

iMessage的操作在第一个现象里已经出现了，所以仅把这一个现象作为切入点来分析就已经能够达到本节的目标了。下面会把火力集中在现象A上。

确定了切入点之后，跟笔者一起思考，怎么把这个现象给具象化成逆向工程的思路：

我们能够观察到的是发生在UI上的现象，即“Text Message”变成“iMessage”。我们知道，UI上显示的内容不是凭空生成的，它显示的是其数据源的值——那么就可以根据这个现象，用Cycript找到UI的数据源，即placeholder。

placeholder也不是凭空生成的，它的值也来自它的数据源——它之所以发生改变，是因为它的数据源（的数据源的数据源……以下简称N重数据源）发生了改变，类似于下面的伪代码：

```
id dataSource = ?;
id a = function(dataSource);
id b = function(a);
id c = function(b);
```

```
…
id z = function(y);
NSString *placeholder = function(z);
```

从上面的伪代码可以知晓，原始数据源是dataSource，dataSource发生了变化，间接导致placeholder变化。可以理解吧？那原始数据源是什么呢？在现象A里，我们唯一输入的就是收件人地址，因此原始数据源当然就是收件人地址啊！对于“检测iMessage”来说，MobileSMS中应该存在一个dataSource转换为placeholder的过程，这个过程就是“检测iMessage”的确切含义，也就是本节的目标，如图10-2所示。

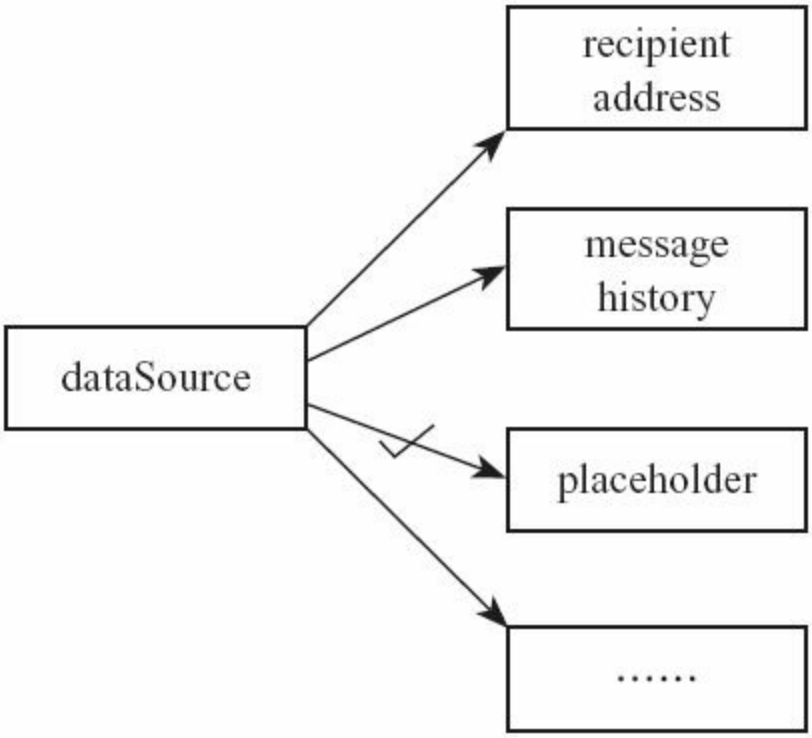


图10-2　dataSource转化为placeholder的过程（1）

你可能会想，图10-2这么直观，既然dataSource已知，那就直接从它入手，找到placeholder，不就达到目的了？但是现实往往没有这么美好——我们没有源代码，进程流程一般也没有这么简单，在大多数时候，dataSource转换为placeholder的过程如图10-3所示。

dataSource要经过多重转换才能生成placeholder，它们之间的关系非常复杂。如果从dataSource入手，

如何知道它要走4条路中的哪条路才能到达placeholder？在这种情况下，因为placeholder是唯一的，所以从placeholder入手，用终点倒推起点，来还原过程，是更高效更可行的做法。

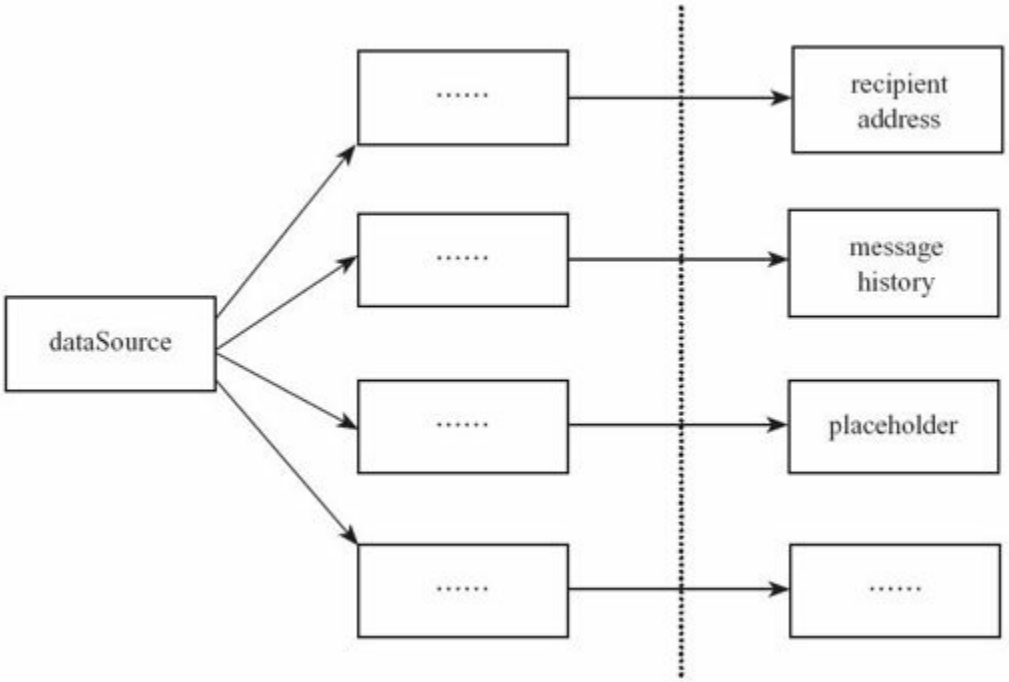


图10-3　dataSource转化为placeholder的过程（2）

总结一下，逆向工程的思路是这样的：先用Cycript定位placeholder，然后通过IDA和LLDB寻找placeholder的N重数据源，直到找到dataSource，最后还原dataSource生成placeholder的过程。看起来很简单是吧？实际操作起来你就知道有多复杂了。这就开始

吧。

10.2.2 用Cycript找出placeholder

打开MobileSMS，新建一条信息，在地址输入框中填写“bbs.iosre.com”，然后点击“return”结束输入，如图10-4所示。

图10-4　新建一条信息

既然要用Cycript找placeholder，那就用Cycript来找找显示placeholder当前值“Text Message”的是哪个view，placeholder一定跟那个view密切相关，对吧？

走起！执行下面的代码：

```
FunMaker-5:~ root# cycript -p MobileSMS
cy# ?expand
expand == true
cy# [[UIApp keyWindow] recursiveDescription]
```

上面代码执行之后，Cycript会打印出keyWindow的视图结构，内容很多，就不贴在这里了。在输出里搜索“Text Message”，发现一个匹配也搜不到。这是怎么回事？相信你也想到了——“Text Message”不在keyWindow里。为了验证我们的猜想，来看看当前这个界面有几个window，如下：

```
cy# [UIApp windows]
@[#"<UIWindow: 0x1575ca10; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x15629c60>; layer =
<UIWindowLayer: 0x156e36f0>>",#"<UITextEffectsWindow:
0x1579ab70; frame = (0 0; 320 568); opaque = NO;
autoresize = W+H; gestureRecognizers = <NSArray:
0x1579b300>; layer = <UIWindowLayer: 0x1579adf0>>",#"
<CKJoystickWindow: 0x1552bf90; baseClass =
UIAutoRotatingWindow; frame = (0 0; 320 568); hidden =
YES; gestureRecognizers = <NSArray: 0x1552b730>; layer =
<UIWindowLayer: 0x1552bdc0>>",#"<UITextEffectsWindow:
0x1683a2e0; frame = (0 0; 320 568); hidden = YES;
gestureRecognizers = <NSArray: 0x1688b9e0>; layer
<UIWindowLayer: 0x168b9ad0>>"]
```

可以看到，每一个以“#”开头的都是一个window，这里一共有4个window，其中第一个是keyWindow。那么哪一个是“Text Message”所在的window呢？从关键字上也能猜测出，带“Text”字样的第2和第4个window可能是我们的目标，而第4个window的hidden属性是YES，它压根儿没有显示在界面上，因此肯定不是它。可见，第2个window很可能就是我们的目标，用Cycript测测看，如下：

```
cy# [#0x1579ab70 setHidden:YES]
```

回车之后发现，不只是信息输入框，整个键盘都被隐藏起来了，如图10-5所示。

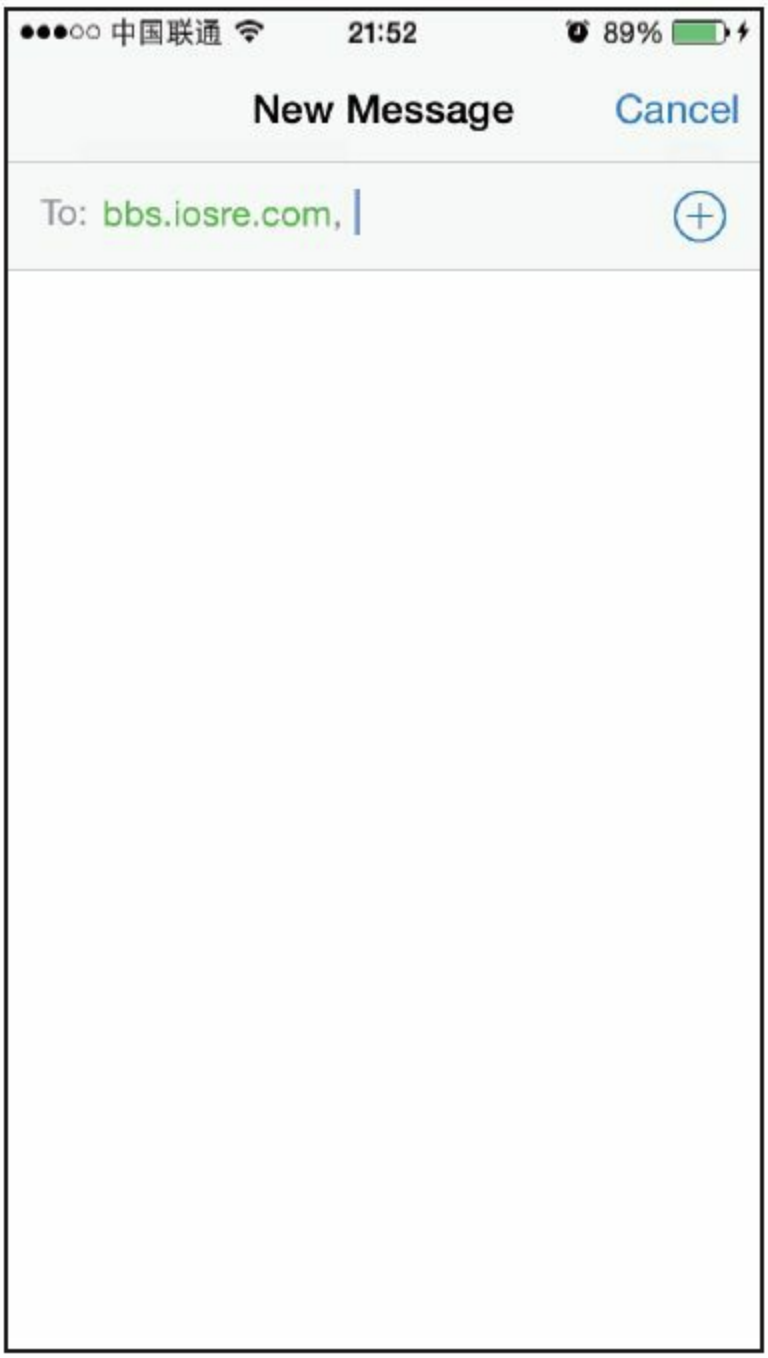


图10-5　信息输入框及键盘被隐藏

从而可以得出结论，“Text Message”就位于这个window中，继续通过Cycript来定位它，如下所示：

```
cy# [#0x1579ab70 setHidden:NO]
cy# [#0x1579ab70 subviews]
@[#"<UIInputSetContainerView: 0x1551fb10; frame = (0 0;
320 568); autoresize = W+H; layer = <CALayer:
0x1551f950>>"]
cy# [#0x1551fb10 subviews]
@[#"<UIInputSetHostView: 0x1551f5e0; frame = (0 250; 320
318); layer = <CALayer: 0x1551f480>>"]
cy# [#0x1551f5e0 subviews]
@[#"<UIKBInputBackdropView: 0x16827620; frame = (0 65;
320 253); userInteractionEnabled = NO; layer = <CALayer:
0x1681c3f0>>",#"<_UIKBCompatInputView: 0x157b88d0; frame
= (0 65; 320 253); layer = <CALayer: 0x157b8a10>>",#"
<CKMessageEntryView: 0x1682ca50; frame = (0 0; 320 65);
opaque = NO; autoresize = W; layer = <CALayer:
0x168ec520>>"]
```

上面代码中又有3个subview，哪个是“Text Message”的所在？用下面的排除法来过一遍：

```
cy# [#0x16827620 setHidden:YES]
```

上面语句执行之后变成了图10-6所示的界面，说明这个view只是键盘背景而已。

执行下面的代码：

```
cy# [#0x16827620 setHidden:NO]
cy# [#0x157b88d0 setHidden:YES]
```

这两个语句执行之后界面变成了图10-7所示的状

态。

图10-6　键盘背景被隐藏

图10-7　键盘按键被隐藏

说明这个view就是键盘本身。也就是说，UIKBInputBackdropView和UIKBCompatInputView共同构成了一个键盘的view，这种官方设计模式可以供第

三方键盘开发者或键盘主题制作者参考。

现在只剩最后一个subview了，CKMessageEntryView这个名字的意义其实也很明显，还是像下面这样验证一下吧：

```
cy# [#0x157b88d0 setHidden:NO]
cy# [#0x1682ca50 setHidden:YES]
```

执行后界面如图10-8所示。

图10-8　信息输入框被隐藏

通过验证，说明“Text Message”在CKMessageEntryView中。继续缩小范围，如下：

```
cy# [#0x1682ca50 setHidden:NO]
cy# [#0x1682ca50 subviews]
@[#"<_UIBackdropView: 0x168ce210; frame = (0 0; 320 65);
opaque = NO; autoresize = W+H; userInteractionEnabled =
NO; layer = <_UIBackdropViewLayer: 0x168f5300>>",#"
<UIView: 0x168d2b70; frame = (0 0; 320 0.5); layer =
<CALayer: 0x168d2be0>>",#"<UIButton: 0x1684b240; frame =
(266 27; 53 33); opaque = NO; layer = <CALayer:
0x168d64b0>>",#"<UIButton: 0x168b88b0; frame = (266 30;
53 26); hidden = YES; opaque = NO; gestureRecognizers =
<NSArray: 0x16840030>; layer = <CALayer: 0x16858420>>",#"
<UIButton: 0x16833ac0; frame = (15 33.5; 25 18.5); opaque
= NO; gestureRecognizers = <NSArray: 0x1682d9b0>; layer =
<CALayer: 0x16838780>>",#"
<_UITextFieldRoundedRectBackgroundViewNeue: 0x168fba00;
frame = (55 8; 209.5 49.5); opaque = NO;
userInteractionEnabled = NO; layer = <CALayer:
0x1682da50>>",#"<UIView: 0x168dcf10; frame = (55 8; 209.5
49.5); clipsToBounds = YES; opaque = NO; layer =
<CALayer: 0x168e4170>>",#"<CKMessageEntryWaveformView:
0x1571b710; frame = (15 25.5; 251 35); alpha = 0; opaque
= NO; userInteractionEnabled = NO; layer = <CALayer:
0x1578fc90>>"]
```

还是使用排除法寻找“Text Message”所在的view，这里的过程就不再重复了，留给读者自己练习。在定位到了“UIView：0x168dcf10”（注意，是第二个UIView对象）之后，继续查看它的subview，如下：

```
cy# [#0x168dcf10 subviews]
@[#"<CKMessageEntryContentView: 0x16389000; baseClass =
UIScrollView; frame = (3 -4; 203.5 57.5); clipsToBounds =
YES; opaque = NO; gestureRecognizers = <NSArray:
0x168f0730>; layer = <CALayer: 0x168e41a0>;
contentOffset: {0, 0}; contentSize: {203.5, 57}>"]
```

上面代码中只有一个subview，继续查看这个subview的subview，如下：

```
cy# [#0x16389000 subviews]
@[#"<CKMessageEntryRichTextView: 0x16295200; baseClass =
UITextView; frame = (0 20.5; 203.5 36.5); text = '';
clipsToBounds = YES; opaque = NO; gestureRecognizers =
<NSArray: 0x168f5a60>; layer = <CALayer: 0x168f59c0>;
contentOffset: {0, 0}; contentSize: {203.5, 36.5}>",#"
<CKMessageEntryTextView: 0x15ad2a00; baseClass =
UITextView; frame = (0 0; 203.5 36.5); text = '';
clipsToBounds = YES; opaque = NO; gestureRecognizers =
<NSArray: 0x1578e600>; layer = <CALayer: 0x157dcff0>;
contentOffset: {0, 0}; contentSize: {203.5, 36.5}>",#"
<UIView: 0x157e9160; frame = (5 28; 193.5 0.5); layer =
<CALayer: 0x15733bd0>>",#"<UIImageView: 0x157308d0; frame
= (-0.5 55; 204 2.5); alpha = 0; opaque = NO; autoresize
= TM; userInteractionEnabled = NO; layer = <CALayer:
0x15730950>>",#"<UIImageView: 0x157ef530; frame = (201 0;
2.5 57.5); alpha = 0; opaque = NO; autoresize = LM;
userInteractionEnabled = NO; layer = <CALayer:
0x157ef5b0>>"]
```

还是用排除法寻找“Text Message”所在的view，我们发现，当执行“[#0x16295200 setHidden:YES]”时，只有“Text Message”被隐藏了，界面上的其他控件并未受影响，如图10-9所示。

这说明CKMessageEntryRichTextView就是我们想要定位的view。打开CKMessageEntry-Rich-

TextView.h，看看能不能找到placeholder的踪影，如图10-10所示。

可是，在CKMessageEntryRichTextView中搜不到placeholder。难道推理出了差错？不要着急，转战到它的父类CKMessageEntryTextView里去看看，如图10-11所示。

可以看到，满屏的placeholder字眼。其中，placeholderLabel和placeholderText这2个property很可疑，难道它们就是我们在找的placeholder？用Cycript验证一下，执行如下命令：

```
cy# [#0x16295200 setPlaceholderText:@"iOSRE"]
```

图10-9　placeholder被隐藏

图10-10　CKMessageEntryRichTextView.h

此时，界面变成了下面这个样子（如图10-12所示）。

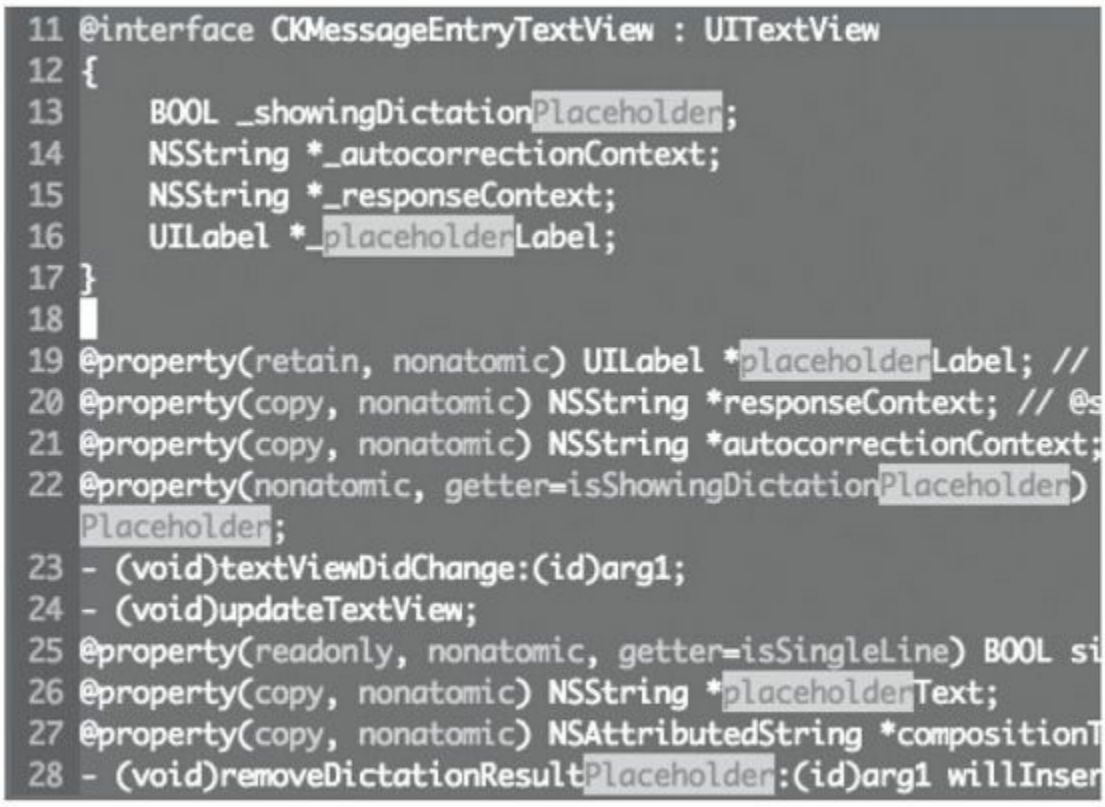


图10-11　CKMessageEntryTextView.h

图10-12　更改placeholder为“iOSRE”

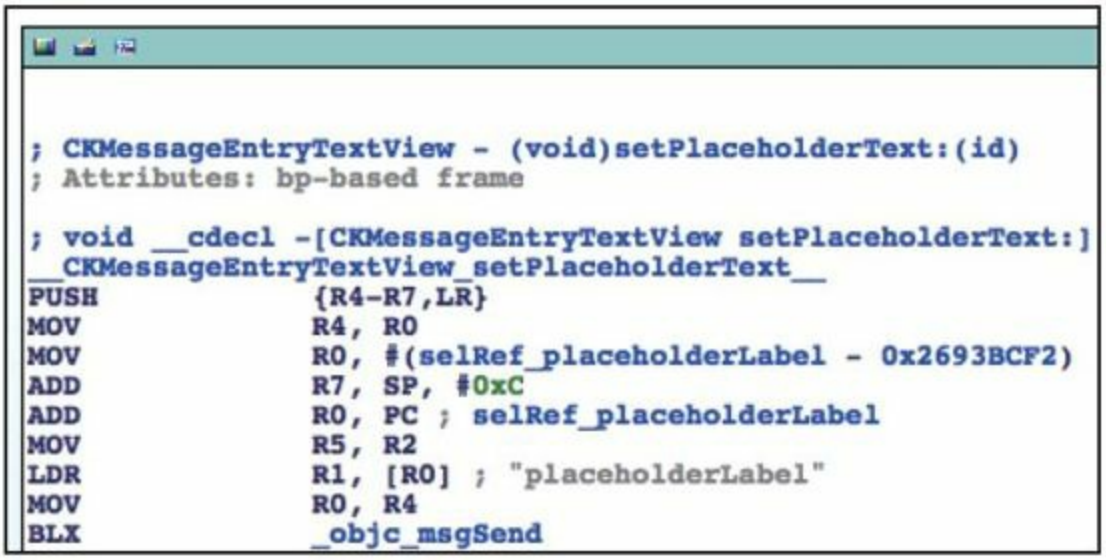
不错，placeholderText就是我们要找的placeholder，自此它们两者等同，为了避免混淆，下文统一使用placeholderText。旗开得胜，我们跨出了

万里长征的第一步！

10.2.3 用IDA和LLDB找出placeholderText的一重数据源

placeholderText是一个property，而要改变一个property，笔者的第一反应就是它的setter。在通过调用setPlaceholderText:函数，把placeholderText从“Text Message”改为“iOSRE”的同时，不妨大胆假设一下，MobileSMS会不会也是通过调用这个setter来改变placeholderText的呢？实际验证一下是最好不过的，接下来轮到IDA和LLDB上场了。

因为最后定位到的CKMessageEntryTextView类来自ChatKit，所以接下来的目标是分析MobileSMS这个可执行文件中的ChatKit库，可以理解吧？好的，把ChatKit的二进制文件丢到IDA中，初始分析结束后，定位到[CKMessageEntryTextView setPlaceholderText:]，如图10-13所示。



```
; CKMessageEntryTextView - (void)setPlaceholderText:(id)
; Attributes: bp-based frame

; void __cdecl -[CKMessageEntryTextView setPlaceholderText:]
__CKMessageEntryTextView_setPlaceholderText__
PUSH            {R4-R7,LR}
MOV             R4, R0
MOV             R0, #(selRef_placeholderLabel - 0x2693BCF2)
ADD             R7, SP, #0xC
ADD             R0, PC ; selRef_placeholderLabel
MOV             R5, R2
LDR             R1, [R0] ; "placeholderLabel"
MOV             R0, R4
BLX             _objc_msgSend
```

图10-13　[CKMessageEntryTextView setPlaceholderText:]（1）

然后用LLDB附加MobileSMS，待初始化完成后执行“c”命令，让MobileSMS运行起来，如下：

```
(lldb) process connect connect://iOSIP:1234
Process 200596 stopped
* thread #1: tid = 0x30f94, 0x316554f0
libsystem_kernel.dylib`mach_msg_trap + 20, queue =
'com.apple.main-thread, stop reason = signal SIGSTOP
    frame #0: 0x316554f0
libsystem_kernel.dylib`mach_msg_trap + 20
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x316554f0:  pop    {r4, r5, r6, r8}
   0x316554f4:  bx     lr
libsystem_kernel.dylib`mach_msg_overwrite_trap:
   0x316554f8:  mov    r12, sp
   0x316554fc:  push   {r4, r5, r6, r8}
(lldb) c
Process 200596 resuming
```

之后，再看看ChatKit的ASLR偏移，如下：

```
(lldb) image list -o -f
[  0] 0x00079000
/private/var/db/stash/_.29LMeZ/Applications/MobileSMS.app

[  1] 0x0019c000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000

[  2] 0x01eac000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/Frameworks/Foundation.fra

……
[  9] 0x01eac000
/Users/snakeninny/Library/Developer/Xcode/iOS
DeviceSupport/8.1
(12B411)/Symbols/System/Library/PrivateFrameworks/ChatKit
```

这个偏移值是0x1eac000。有了这个值，就可以在[CKMessageEntryTextView setPlaceholderText:]中下个断点，看看它有没有被调用，如果被调用，调用者又是谁。为了把断点下在这个函数的开头位置，要先在IDA中看看这个函数的基地址，如图10-14所示，可以看到，是0x2693BCE0。

```
_text:2693BCE0 ; CKMessageEntryTextView - (void)setPlaceholderText:(id)
_text:2693BCE0 ; Attributes: bp-based frame
_text:2693BCE0
_text:2693BCE0 ; void __cdecl -[CKMessageEntryTextView setPlaceholderText:]
_text:2693BCE0 __CKMessageEntryTextView_setPlaceholderText__
_text:2693BCE0                                         ; DATA XREF: __objc_
_text:2693BCE0                 PUSH            {R4-R7,LR}
_text:2693BCE2                 MOV             R4, R0
```

图10-14　[CKMessageEntryTextView setPlaceholderText:]（2）

所以断点地址是0x1eac000+0x2693BCE0=0x287E7CE0。

```
(lldb) br s -a 0x287E7CE0
Breakpoint 1: where = ChatKit`-
[CKMessageEntryTextViewsetPlaceholderText:], address =
0x287e7ce0
```

接着，把地址输入框中的“bbs.iosre.com”换成一个支持iMessage的地址“snakeninny@gmail.com”，看看进程会不会停在断点上。这时，你会发现，在实际操作中，随着地址的变动，进程会多次停在断点上，这说明[CKMessageEntryTextView setPlaceholderText:]被调用了很多次。那么如何知道是在哪一次调用中，placeholderText从“Text Message”变成“iMessage”呢？此刻，之前介绍的comm命令就派上用场了，命令如下：

```
(lldb) br com add 1
Enter your debugger command(s).  Type 'DONE' to end.
> po $r2
> p/x $lr
> c
> DONE
```

这个命令的意思很简单，就是在断点触发时打印R2的“Objective-C description”，即setPlaceholdeText:的参数；然后以十六进制格式打印LR的值，即[CKMessageEntry-TextView setPlaceholderText:]的返回地址。如果R2的值是“iMessage”，说明系统自身也是通过setPlaceholderText:来改变placeholderText的，这个函数的参数就是placeholderText的一重数据源；同时因为对应LR的值位于调用者的地址范围内，所以可以找到setPlaceholderText:的调用者，继续跟踪参数的数据源，即placeholderText的二重数据源。清空地址输入框，再重新输入“snakeninny@gmail.com”，看看LLDB什么时候会打印“iMessage”，如下：

```
<object returned empty description>
(unsigned int) $11 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
<object returned empty description>
```

```
(unsigned int) $13 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
<object returned empty description>
(unsigned int) $15 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
Text Message
(unsigned int) $17 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
iMessage
(unsigned int) $19 = 0x28768b33
Process 200596 resuming
Command #3 'c' continued the target.
```

可以看到，当placeholder变成“iMessage”时，LR的值是0x28768b33。根据第4章“偏移后指令基地址=偏移前指令基地址+指令所在模块的ASLR偏移”公式，0x28768b33-0x1eac000=0x268BCB33。可见，这个地址位于ChatKit内，如图10-15所示。

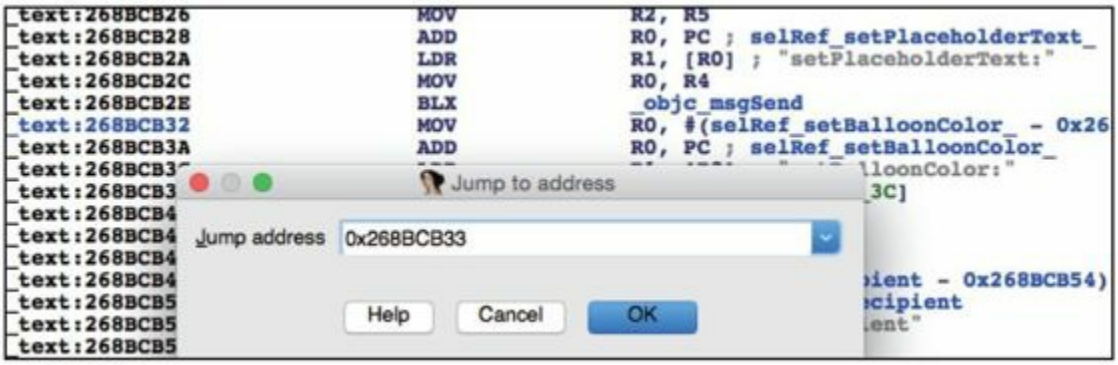


图10-15　跳转到ChatKit中的0x268BCB33

说明MobileSMS确实就是通过setPlaceholder:函数来改变placeholder的，函数的参数就是placeholder的一重数据源，同时二重数据源的线索也有了着落。万里长征第二步走完，有惊无险！

10.2.4 用IDA和LLDB找出placeholderText的N重数据源

前面在操作中多次编辑地址时，不知大家有没有注意到一个现象，那就是当我们正在编辑时，placeholderText是空白的；只有在点击了键盘上的“return”结束编辑后，placeholderText才会显示“Text Message”或“iMessage”。也就是说，当地址编辑结束时，iOS才会检测当前地址是否支持iMessage，出于节省性能的考虑，这样的设计合情合理。也正是基于这样的设计，在接下来的调试中，可以先把目标地址编辑好，然后设置断点，最后点击“return”，这样断点如果被触发，则说明进程停在了iMessage检测的过

程中。下面把IDA往上拉，看看[CKMessageEntryTextView setPlaceholderText:]的调用者是谁，如图10-16所示。

图10-16　[CKMessageEntryTextView setPlaceholderText:]的调用者

虽然在“更新输入视图”的时候“设置占位符”，说得过去，但是，[CKMessageEntryView updateEntryView]没有参数，它怎么知道placeholderText应该设置成“Text Message”还是“iMessage”呢？那只有一种可能，就是它的内部进行了判断，得出了该地址支持iMessage的结论，改变了placeholderText的二重数据源。回到IDA，看看二重数据源来自哪里，如图10-17所示。

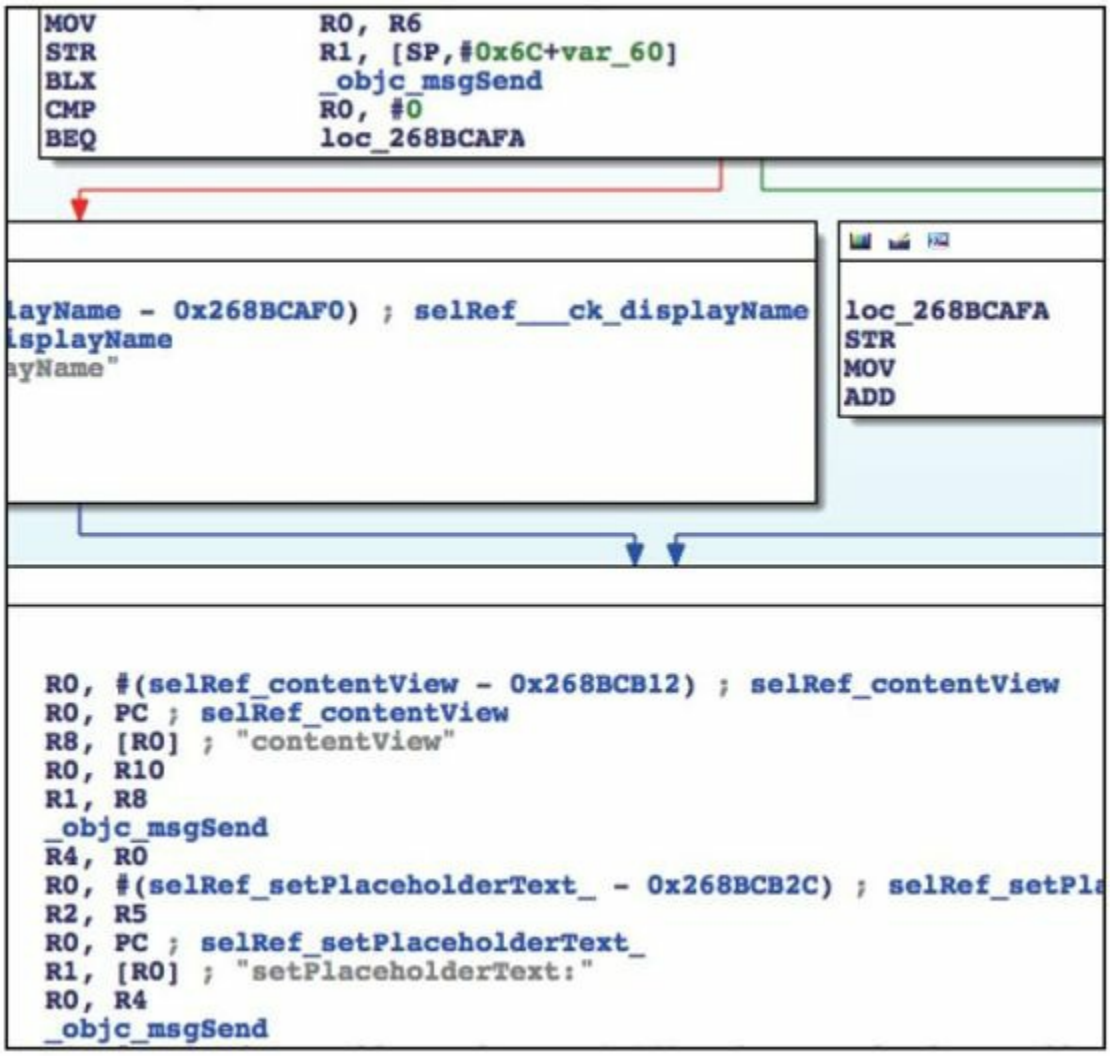


图10-17　寻找二重数据源

R2即函数参数，也就是一重数据源；而R2来自于R5，因此R5就是二重数据源。R5又是来自于哪里呢？这里出现了分支，我们看看分支的跳转条件，如图10-18所示。

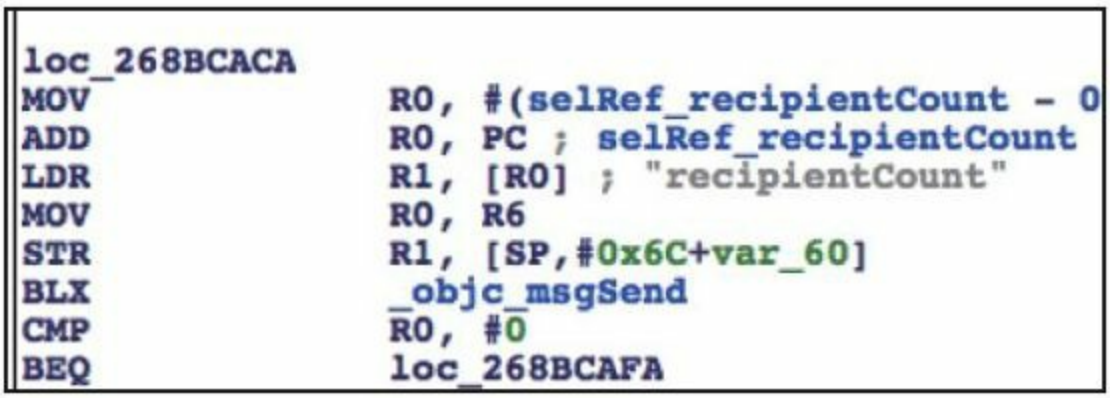


图10-18　分支的跳转条件

可以看到，跳转条件是“[$r0 recipientCount]==0”。“recipient”的字面意思很明显，就是指信息的收件人，当收件人数为0，即没有收件人时，进程走右边，否则走左边。因为这里已经有一个收件人了，收件人数不为0，所以进程很有可能走左边。要验证也很简单，先清空地址输入框，然后在地址输入框中输入“snakeninny@gmail.com”，接着在右边的分支尾部下一个断点，最后点击键盘上的“return”。此时，会发现断点并没有得到触发，因此可以肯定地说，R5来自于左边的[$r8__ck_displayName]，即[$r8__ck_displayName]是三重数据源。但R8又来自哪里呢？往上拉IDA，在

[CKMessageEntryView updateEntryView]的开始部分，可以看到R8来自[[self conversation]sendingService]，如图10-19所示。

因此，[[self conversation]sendingService]是四重数据源。再用LLDB来验证一下判断：清空地址输入框，输入“snakeninny@gmail.com”，然后在图10-19的“MOV R8,R0”处下一个断点，接着点击“return”。待断点触发时执行“po[$r0__ck_displayName]”，看看是否会输出“iMessage”，如下：

```
PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
PUSH.W          {R8,R10,R11}
SUB             SP, SP, #0x54
MOV             R10, R0
MOV             R0, #(selRef_conversation - 0x26
ADD             R0, PC ; selRef_conversation
LDR             R1, [R0] ; "conversation"
MOV             R0, R10
STR             R1, [SP,#0x6C+var_58]
BLX             _objc_msgSend
MOV             R6, R0
MOV             R0, #(selRef_sendingService - 0x
ADD             R0, PC ; selRef_sendingService
LDR             R1, [R0] ; "sendingService"
MOV             R0, R6
STR             R1, [SP,#0x6C+var_44]
BLX             _objc_msgSend
MOV             R8, R0
```

图10-19　寻找四重数据源

```
(lldb) br s -a 0x28768962
Breakpoint 14: where = ChatKit`-[CKMessageEntryView
updateEntryView] + 54, address = 0x28768962
(lldb) br com add 14
Enter your debugger command(s).  Type 'DONE' to end.
> po [$r0 __ck_displayName]
> c
> DONE
Text Message
Process 200596 resuming
Command #2 'c' continued the target.
iMessage
Process 200596 resuming
Command #2 'c' continued the target.
```

从上面代码来看，断点被触发两次，第二次被触发时，iOS检测到“snakeninny@gmail.com”支持iMessage。既然“iMessage”来自于[[[self conversation]sendingService]__ck_displayName]，那么[self conversation]是个什么类型的对象？[[self conversation]sendingService]呢？别急，下面一个一个来看。

重新输入地址，然后在[CKMessageEntryView updateEntryView]的前两个“objc_msgSend”上各下一个

断点，最后点击“return”，现在来看看这里的对象类型，如下：

```
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b528948 ChatKit`-
[CKMessageEntryView updateEntryView] + 28, queue =
'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x2b528948 ChatKit`-[CKMessageEntryView
updateEntryView] + 28
ChatKit`-[CKMessageEntryView updateEntryView] + 28:
-> 0x2b528948:  blx    0x2b5f5f44                ; symbol
stub for: MarcoShouldLogMadridLevel$shim
   0x2b52894c:  mov    r6, r0
   0x2b52894e:  movw   r0, #51162
   0x2b528952:  movt   r0, #2547
(lldb) p (char *)$r1
(char *) $6 = 0x2b60cc16 "conversation"
(lldb) ni
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b52894c ChatKit`-
[CKMessageEntryView updateEntryView] + 32, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2b52894c ChatKit`-[CKMessageEntryView
updateEntryView] + 32
ChatKit`-[CKMessageEntryView updateEntryView] + 32:
-> 0x2b52894c:  mov    r6, r0
   0x2b52894e:  movw   r0, #51162
   0x2b528952:  movt   r0, #2547
   0x2b528956:  add    r0, pc
(lldb) po $r0
CKPendingConversation<0x1587e870>{identifier:'(null)'
guid:'(null)'}(null)
```

可见，[self conversation]返回的是一个CKPendingConversation类型的对象。接着看下一个，如下：

```
(lldb) c
Process 14235 resuming
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b52895e ChatKit`-
[CKMessageEntryView updateEntryView] + 50, queue =
'com.apple.main-thread, stop reason = breakpoint 2.1
    frame #0: 0x2b52895e ChatKit`-[CKMessageEntryView
updateEntryView] + 50
ChatKit`-[CKMessageEntryView updateEntryView] + 50:
-> 0x2b52895e:  blx    0x2b5f5f44                ; symbol
stub for: MarcoShouldLogMadridLevel$shim
   0x2b528962:  mov    r8, r0
   0x2b528964:  movw   r0, #52792
   0x2b528968:  movt   r0, #2547
(lldb) p (char *)$r1
(char *) $8 = 0x2b6105e1 "sendingService"
(lldb) ni
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b528962 ChatKit`-
[CKMessageEntryView updateEntryView] + 54, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2b528962 ChatKit`-[CKMessageEntryView
updateEntryView] + 54
ChatKit`-[CKMessageEntryView updateEntryView] + 54:
-> 0x2b528962:  mov    r8, r0
   0x2b528964:  movw   r0, #52792
   0x2b528968:  movt   r0, #2547
   0x2b52896c:  add    r0, pc
(lldb) po $r0
IMService[SMS]
(lldb) po [$r0 class]
IMServiceImpl
```

显然，[CKPendingConversation sendingService]返回的是一个IMSercice Impl对象，且其值为“IMService[SMS]”（第2次被触发时就会变成“IMService[iMessage]”）。因此，四重数据源就

是“[CKPendingConversation sendingService]”，没问题吧？

分析到这里，再回到IDA，定位[CKPendingConversation sendingService]，看看它的内部是如何工作的，如图10-20所示。

其中的逻辑并不算太复杂，但在若干分支里，进程实际走的是哪一条路呢？用LLDB调试一下（下断点前先重新输入地址），在所有条件跳转指令上留意一下跳转条件的值和下一条指令的地址，如下：

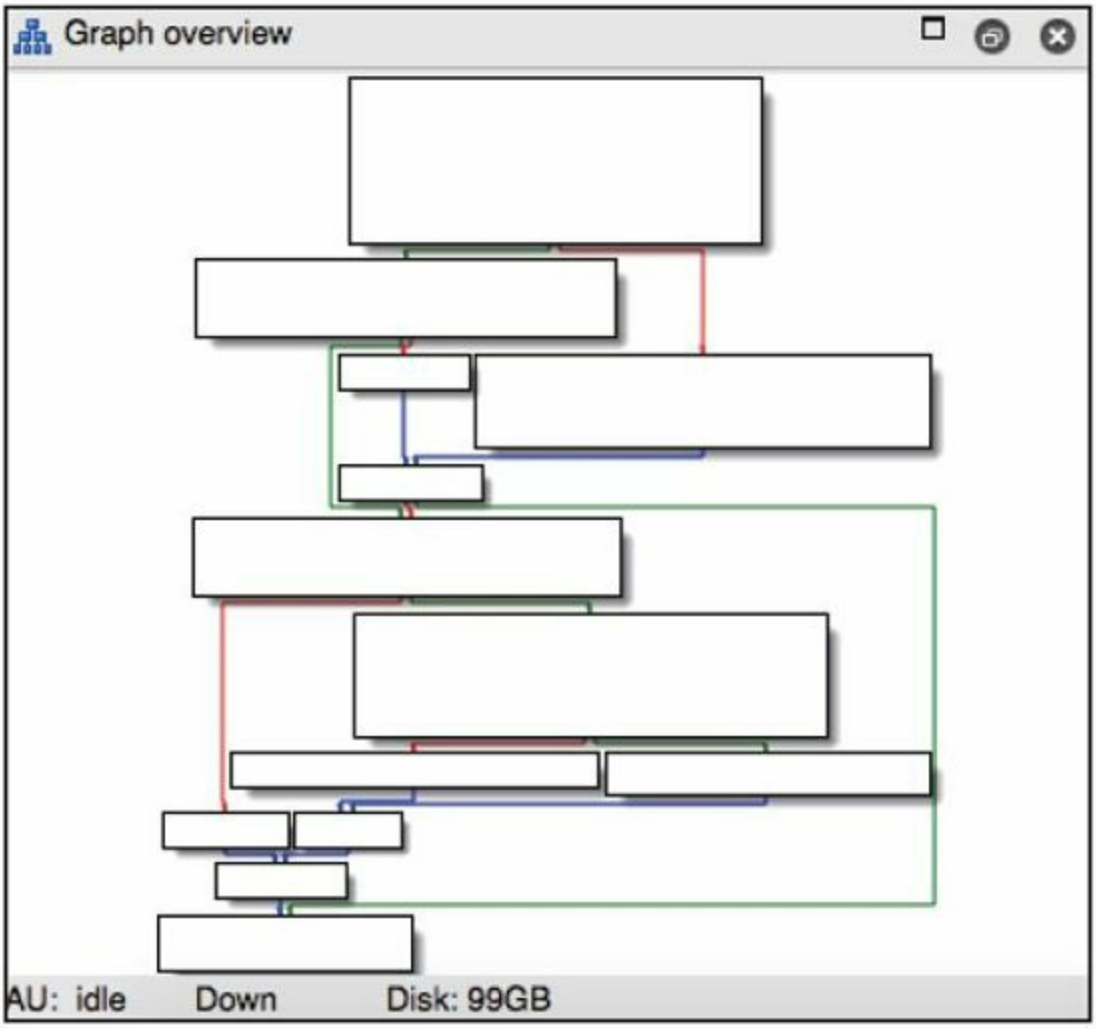


图10-20　[CKPendingConversation sendingService]

```
Process 14235 stopped
* thread #1: tid = 0x379b, 0x2b5f0264 ChatKit`-
[CKPendingConversation sendingService], queue =
'com.apple.main-thread, stop reason = breakpoint 3.1
    frame #0: 0x2b5f0264 ChatKit`-[CKPendingConversation
sendingService]
ChatKit`-[CKPendingConversation sendingService]:
-> 0x2b5f0264:  push   {r4, r5, r7, lr}
   0x2b5f0266:  add    r7, sp, #8
   0x2b5f0268:  sub    sp, #8
   0x2b5f026a:  mov    r4, r0
(lldb) ni
Process 14235 stopped
……
* thread #1: tid = 0x379b, 0x2b5f027e ChatKit`-
[CKPendingConversation sendingService] + 26, queue =
```

```
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2b5f027e ChatKit`-[CKPendingConversation
sendingService] + 26
ChatKit`-[CKPendingConversation sendingService] + 26:
-> 0x2b5f027e:  cbz    r0, 0x2b5f02a4            ; -
[CKPendingConversation sendingService] + 64
   0x2b5f0280:  movw   r0, #38082
   0x2b5f0284:  movt   r0, #2535
   0x2b5f0288:  str    r4, [sp]
(lldb) p $r0
(unsigned int) $11 = 0
(lldb) ni
Process 14235 stopped
……
* thread #1: tid = 0x379b, 0x2b5f02b8 ChatKit`-
[CKPendingConversation sendingService] + 84, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2b5f02b8 ChatKit`-[CKPendingConversation
sendingService] + 84
ChatKit`-[CKPendingConversation sendingService] + 84:
-> 0x2b5f02b8:  cbz    r0, 0x2b5f02c4            ; -
[CKPendingConversation sendingService] + 96
   0x2b5f02ba:  mov    r0, r4
   0x2b5f02bc:  mov    r1, r5
   0x2b5f02be:  blx    0x2b5f5f44                ; symbol
stub for: MarcoShouldLogMadridLevel$shim
(lldb) p $r0
(unsigned int) $12 = 341691792
(lldb) ni
Process 14235 stopped
……
* thread #1: tid = 0x379b, 0x2b5f02c2 ChatKit`-
[CKPendingConversation sendingService] + 94, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2b5f02c2 ChatKit`-[CKPendingConversation
sendingService] + 94
ChatKit`-[CKPendingConversation sendingService] + 94:
-> 0x2b5f02c2:  cbnz   r0, 0x2b5f032c            ; -
[CKPendingConversation sendingService] + 200
   0x2b5f02c4:  movw   r0, #35464
   0x2b5f02c8:  movt   r0, #2535
   0x2b5f02cc:  add    r0, pc
(lldb) p $r0
(unsigned int) $13 = 341691792
(lldb) ni
Process 14235 stopped
```

```
……
* thread #1: tid = 0x379b, 0x2b5f032e ChatKit`-
[CKPendingConversation sendingService] + 202, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x2b5f032e ChatKit`-[CKPendingConversation
sendingService] + 202
ChatKit`-[CKPendingConversation sendingService] + 202:
-> 0x2b5f032e:  pop    {r4, r5, r7, pc}
ChatKit`-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]:
   0x2b5f0330:  push   {r4, r5, r6, r7, lr}
   0x2b5f0332:  add    r7, sp, #12
   0x2b5f0334:  push.w {r8, r10, r11}
```

进程的执行流程就显而易见了。这里一共有3次条件跳转，分别是CBZ、CBZ和CBNZ，而此时的R0分别是0、341691792和341691792，由此可知进程的执行流程是绿线、红线、蓝线、绿线，如图10-21所示。

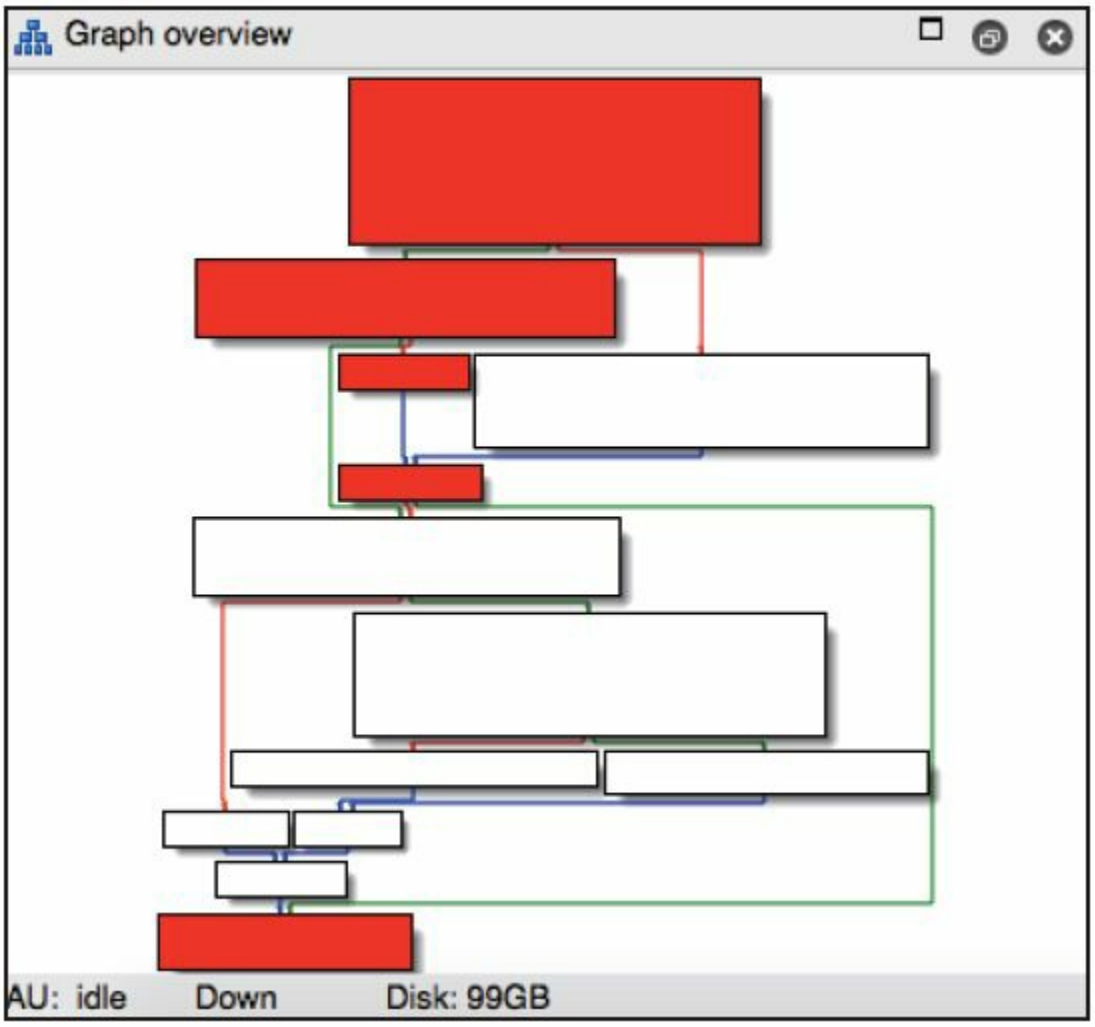


图10-21　进程的执行流程

那么[CKPendingConversation sendingService]的值实际来自[CKPendingConversation composeSendingService]，它是五重数据源，没问题吧？现在，在IDA中定位到了新的函数（如图10-22所示），继续分析。

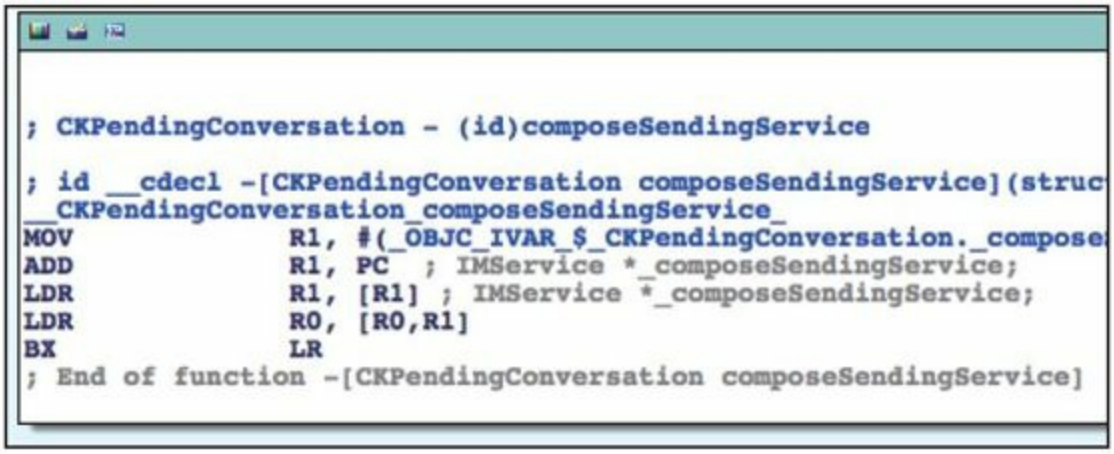


图10-22 [CKPendingConversation composeSendingService]

很明显，[CKPendingConversation composeSendingService]只是读取了实例变量_composeSendingService的值，也就是说，_composeSendingService是六重数据源。既然如此，我们只要找到写入此实例变量的位置，就找到了七重数据源了。

单击_OBJC_IVAR_$_CKPendingConversation._composeSenc让光标停留在其上，然后按下“x”，打开此变量的xrefs窗口，如图10-23所示。

在这里，显式调用_composeSendingService的正好是一个getter和一个setter，因此composeSendingService很可能是一个property，打开CKPendingConversation.h来验证一下，如图10-24所示。

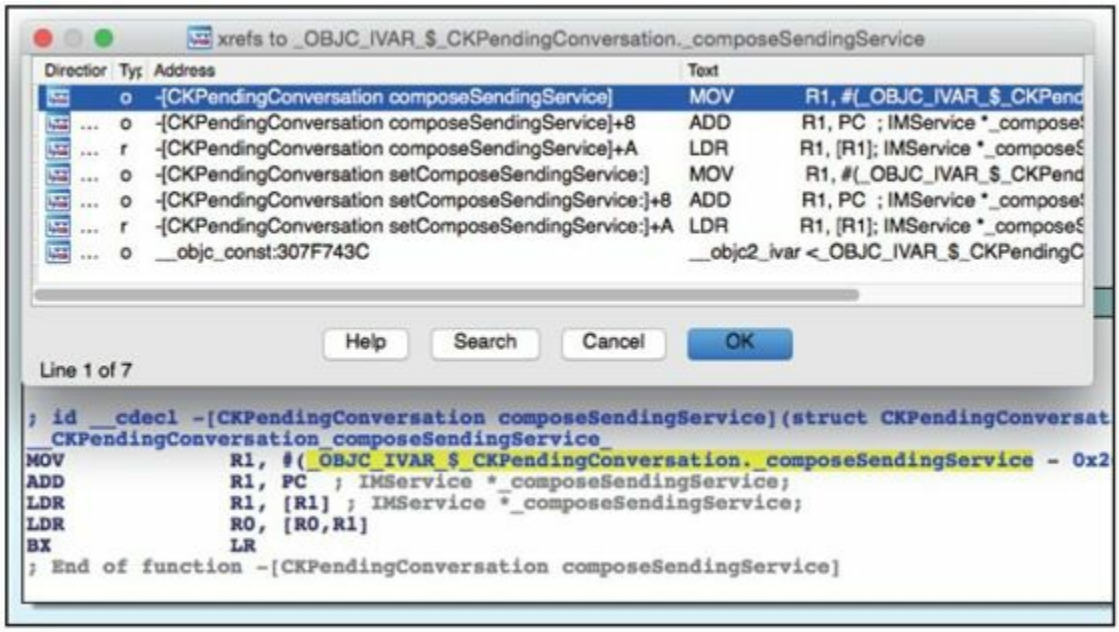


图10-23　查看交叉引用

```
11 @interface CKPendingConversation : CKConversation
12 {
13     BOOL _noAvailableServices;
14     IMService *_previousSendingService;
15     IMService *_composeSendingService;
16 }
17
18 @property(nonatomic) IMService *composeSendingService;
19 @property(nonatomic) IMService *previousSendingService;
```

图10-24　CKPendingConversation.h

在Objective-C中，对property的写操作一般是通过setter完成的，那么要寻找七重数据源，就要在[CKPendingConversation setComposeSendingService:]上下一个断点，然后查看其调用者的情况。操作流程跟前面完全一样，先重新输入地址，然后在[CKPendingConversation setComposeSendingService:]的开始处下一个断点，最后点击键盘上的“return”，触发断点，如下：

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b3665c ChatKit`-
[CKPendingConversation setComposeSendingService:], queue
= 'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x30b3665c ChatKit`-[CKPendingConversation
setComposeSendingService:]
ChatKit`-[CKPendingConversation
setComposeSendingService:]:
-> 0x30b3665c:  movw   r1, #41004
   0x30b36660:  movt   r1, #2535
   0x30b36664:  add    r1, pc
   0x30b36666:  ldr    r1, [r1]
(lldb) p/x $lr
(unsigned int) $0 = 0x30b3656d
```

用这里的LR减去ChatKit的ASLR偏移得到

0x2698456D，得出其偏移前的值。再在IDA中跳到这个值所在的地址，如图10-25所示。

图10-25　跳转到ChatKit中的0x2698456D

[CKPendingConversation setComposeSendingService:]的参数R2就是七重数据源。R2来自R6，因此R6是八重数据源。再往上看看R6是什么，如图10-26所示。

```
sub_26984530

var_18= -0x18
arg_0=  8

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
PUSH.W          {R8,R10}
SUB             SP, SP, #4
MOV             R6, R1
MOV             R1, #(selRef_composeSendingService -
MOV             R4, R0
ADD             R1, PC ; selRef_composeSendingService
LDR             R0, [R4,#0x14]
MOV             R8, R3
LDR             R1, [R1] ; "composeSendingService"
MOV             R10, R2
BLX             _objc_msgSend
CMP             R0, R6
BEQ             loc_269845B0
```

图10-26　寻找九重数据源

R6来自R1，可见R1是九重数据源。那么R1又是哪里来的？因为现在位于sub_26984530的内部，而R1没有被赋值就直接取值了，说明R1来自sub_26984530的调用者，对吧？那就去看看sub_26984530的交叉引用信息，寻找它的调用者信息，如图10-27所示。

```
; Attributes: bp-based frame

sub_26984530

xrefs to sub_26984530
Address
-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]+24
-[CKPendingConversation refreshComposeSendingServiceForAddresses:withCompletionBlock:]+34
```

图10-27　查看交叉引用

刷新发送服务？这个名字有点“暧昧”啊！到图10-28中所示的[CKPendingConversation refreshComposeSendingServiceForAddresses:withComple中看一看，sub_26984530明显是refreshStatusForAddresses:withCompletionBlock:的第二个参数，即completionBlock，如图10-28所示。

这里虽然显式用到了sub_26984530，但只是作为参数传递给objc_msgSend的，并没有直接调用。那么它的调用者到底是谁呢？这个套路前面已经反复演练过了：重新输入地址，在sub_26984530的开始处下断点，点击键盘上的“return”，触发断点，如下：

```
ADD             LR, PC ; sub_26984530
STR             R1, [SP,#0x24+var_20]
MOVS            R1, #0
STR             R1, [SP,#0x24+var_1C]
LDR.W           R1, [R12] ; "refreshStatusForAddresses:withCompletio"...
STR.W           LR, [SP,#0x24+var_18]
STR.W           R9, [SP,#0x24+var_14]
STR             R0, [SP,#0x24+var_10]
STR             R3, [SP,#0x24+var_C]
MOV             R3, SP
BLX             _objc_msgSend
```

图10-28　[CKPendingConversation

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b36530 ChatKit`__86-
[CKPendingConversation
refreshComposeSendingServiceForAddresses:withCompletionBl
 queue = 'com.apple.main-thread, stop reason = breakpoint
6.1
    frame #0: 0x30b36530 ChatKit`__86-
[CKPendingConversation
refreshComposeSendingServiceForAddresses:withCompletionBl

ChatKit`__86-[CKPendingConversation
refreshComposeSendingServiceForAddresses:withCompletionBl

-> 0x30b36530:  push   {r4, r5, r6, r7, lr}
   0x30b36532:  add    r7, sp, #12
   0x30b36534:  push.w {r8, r10}
   0x30b36538:  sub    sp, #4
(lldb) p/x $lr
(unsigned int) $38 = 0x30b364bb
```

LR偏移前的值是0x30b364bb-0xa1b2000=0x269844BB。定位到IDA中，如图10-29所示。

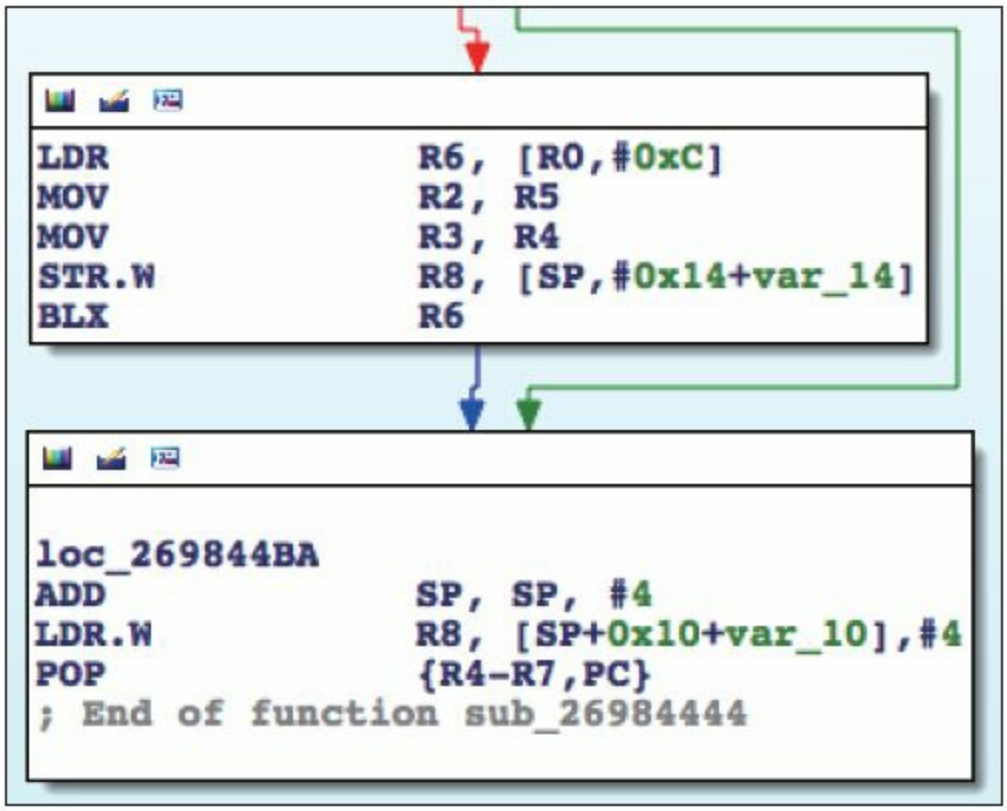


图10-29　sub_26984530的调用者

可见，sub_26984530并没有被显式调用，而是在sub_26984444内部把函数的地址存放到了R6中，然后跳转过去隐式调用。那么自然，九重数据源就来自于sub_26984444，再往上看看它的来源，如图10-30所示。

图10-30　寻找十重数据源（1）

这个subroutine内部进行了多次判断，才决定是把[IMServiceImpl smsService]还是把[IMServiceImpl iMessageService]赋给R1。我们看看判断条件是什么（如图10-31所示）。

图10-31　寻找十重数据源（2）

当R0为2时，[IMServiceImpl iMessageService]是十重数据源，否则还要判断R1的值，若其值是0，则十重数据源是[IMServiceImpl smsService]，否则是[IMServiceImpl iMessageService]。用伪代码表示如下：

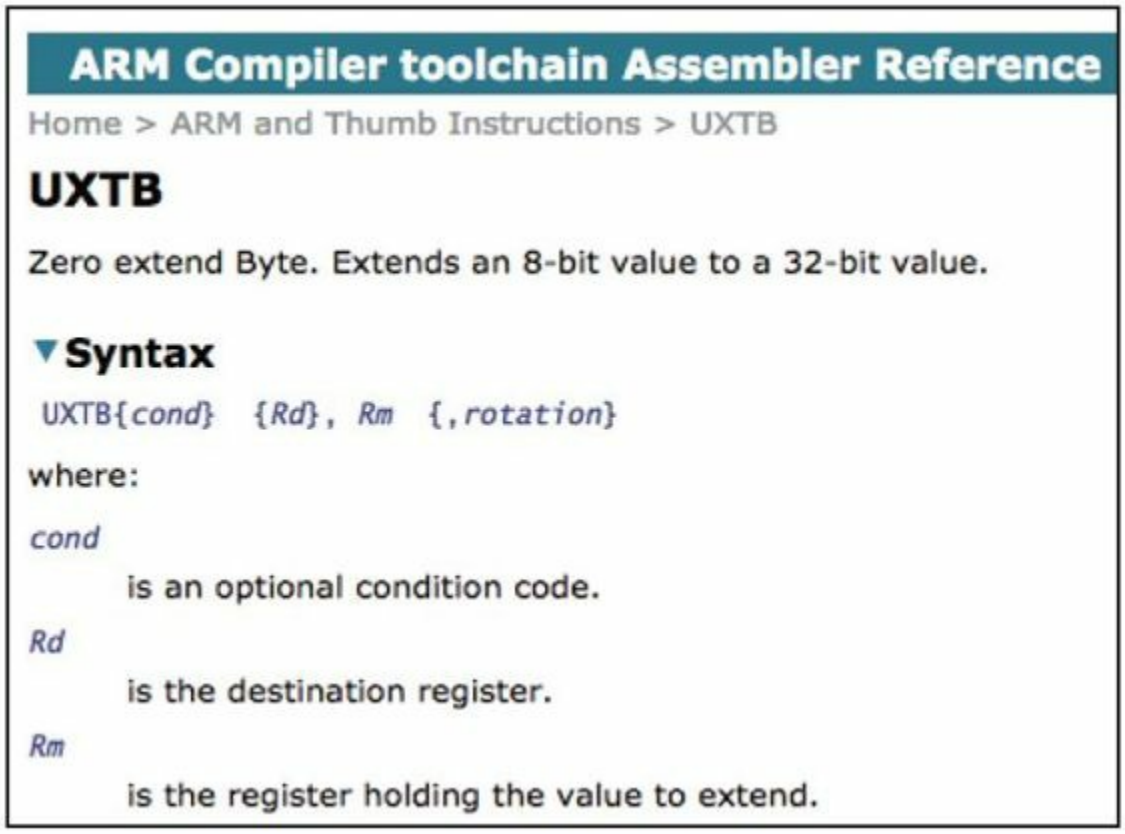
```
- (BOOL)supportIMessage
{
      if (R0 == 2 || R1 != 0) return YES;
      return NO;
}
```

也就是说，这里的R0与R1共同决定十重数据源

的值，它们俩共同担当起了十一重数据源的重任，我们分别称它们为十一重数据源A和十一重数据源B，上面的伪代码也可以写作：

```
- (BOOL)supportIMessage
{
      if (十一重数据源A == 2 || 十一重数据源B != 0) return
YES;
      return NO;
}
```

回到图10-31，再看看十一重数据源是哪里来的——R0来自于“UXTB.W R0,R8”。


ARM Compiler toolchain Assembler Reference

Home > ARM and Thumb Instructions > UXTB

UXTB

Zero extend Byte. Extends an 8-bit value to a 32-bit value.

▼Syntax

UXTB{*cond*} {*Rd*}, *Rm* {,*rotation*}

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

图10-32　UXTB的作用

依图10-32所示的ARM官方文档可知，UXTB的作用是给R8中存放的8位数值高位填充0，将其扩展到32位，然后放到R0里去（R0是32位寄存器），也就是说，R0来自于R8，R8是十二重数据源A。而arg_0=0x8，R8=*(R7+arg_0）=*(R7+0x8)，R7=SP+0xc，因此R8=*(SP+0x14)，即*(SP+0x14)是十三重数据源A。*(SP+0x14)又是哪里来的呢？它不是凭空产生的，因此在“LDR.W R8,[R7,#8]”之前，一定有一条指令往*(SP+0x14)写了数据，对吧？那里就是十四重数据源A的所在之处，我们要倒推回去找到这个往*(SP+0x14)写数据的地方。

但是事情远没有说起来这么简单——因为PUSH和POP等操作会改变SP的值，所以现在的*(SP+0x14)在其他的指令中可能会由于SP发生变化而变成*(SP’+offset)，而offset的值现在还没法确定！这下子麻

烦了，我们必须找出“LDR.W R8,[R7,#8]”之前每一个往*(SP’+offset)写数据的操作，然后检查(SP+0x14)和(SP’+offset)是否相等。而SP的变化又比较频繁，这就成了难点。请大家一定跟紧了，现在，要从“LDR.W R8,[R7,#8]”开始，倒推并检查每一个往*(SP’+offset)写数据的操作。

在sub_26984444，“LDR.W R8,[R7,#8]”前的4条指令全都跟SP相关，把当前指令还未执行时SP的值用SP1~SP4标注到指令上，如图10-33所示。

图10-33　标记不同的SP

当“PUSH{R4-R7,LR}”还未执行时，SP的值是SP1，执行之后变成SP2，可以理解吧？下面一条条指令推导一下这里的SP是怎么变化的：

“PUSH{R4-R7,LR}”将R4、R5、R6、R7和LR共5个寄存器入栈，每个寄存器都是32位即4字节，而ARM的栈是满递减的，因此，SP2=SP1-5*0x4=SP1-0x14；“ADD R7,SP,#0xC”，即R7=SP2+0xc，没有影响SP的值；“STR.W R8,[SP,#0xC+var_10]!”中的var_10=-0x10，因此这条指令等价于“STR.W R8,[SP,#-4]!”，即*(SP2–0x4）=R8，且此条指令仍未影响SP的值；“SUB SP,SP,#4”，即SP3=SP2-0x4。按照这种表达方式，十三重数据源A是*(SP2+0x14)，在通过“LDR.W R8,[R7,#8]”取值时尚未在sub_26984444内赋值，因此*(SP2+0x14)的值一定来自sub_26984444的调用者。类似地，在sub_26984444内部R1未被赋值即已取值，它也一定来自sub_26984444的调用者，可以理解吗？如果还不理解，就把这一段重看一遍，一

定要先想清楚，再继续看下面的内容。

好了，十三重数据源A和十一重数据源B都来自sub_26984444的调用者，下一步的任务已经很明确了：去sub_26984444的调用者寻找十四重数据源A和十二重数据源B。

重输地址，在sub_26984444的开始处下断点，点击键盘上的“return”，触发断点，如下：

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b36444 ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 queue = 'com.apple.main-thread, stop reason = breakpoint
7.1
    frame #0: 0x30b36444 ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv

ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv

-> 0x30b36444:  push   {r4, r5, r6, r7, lr}
   0x30b36446:  add    r7, sp, #12
   0x30b36448:  str    r8, [sp, #-4]!
   0x30b3644c:  sub    sp, #4
(lldb) p/x $lr
(unsigned int) $39 = 0x331f0d75
```

LR偏移前的值是0x331f0d75–

0xa1b2000=0x2903ED75，并不在ChatKit中。这时候该怎么定位0x2903ED75位于哪一个模块呢？方法之前也说过了，那就是在sub_26984444的末尾下断点，然后“ni”到调用者的内部，看看它在哪个模块就好了，如下：

```
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x30b364c0 ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 124, queue = 'com.apple.main-thread, stop reason =
breakpoint 8.1
    frame #0: 0x30b364c0 ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 124
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 124:
-> 0x30b364c0:  pop    {r4, r5, r6, r7, pc}
   0x30b364c2:  nop
ChatKit`__copy_helper_block_:
   0x30b364c4:  ldr    r1, [r1, #20]
   0x30b364c6:  adds   r0, #20
(lldb) ni
Process 30928 stopped
* thread #1: tid = 0x78d0, 0x331f0d74
IMCore`___lldb_unnamed_function425$$IMCore + 1360, queue
= 'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x331f0d74
IMCore`___lldb_unnamed_function425$$IMCore + 1360
IMCore`___lldb_unnamed_function425$$IMCore + 1360:
-> 0x331f0d74:  movw   r0, #26972
   0x331f0d78:  movt   r0, #2081
   0x331f0d7c:  add    r0, pc
   0x331f0d7e:  ldr    r1, [r0]
```

我们来到了IMCore的内部。刚才已经计算出了sub_26984444执行完成后的返回地址是0x2903ED75，因此把IMCore拖进IDA，待分析完毕后跳转到0x2903ED75，如图10-34所示。

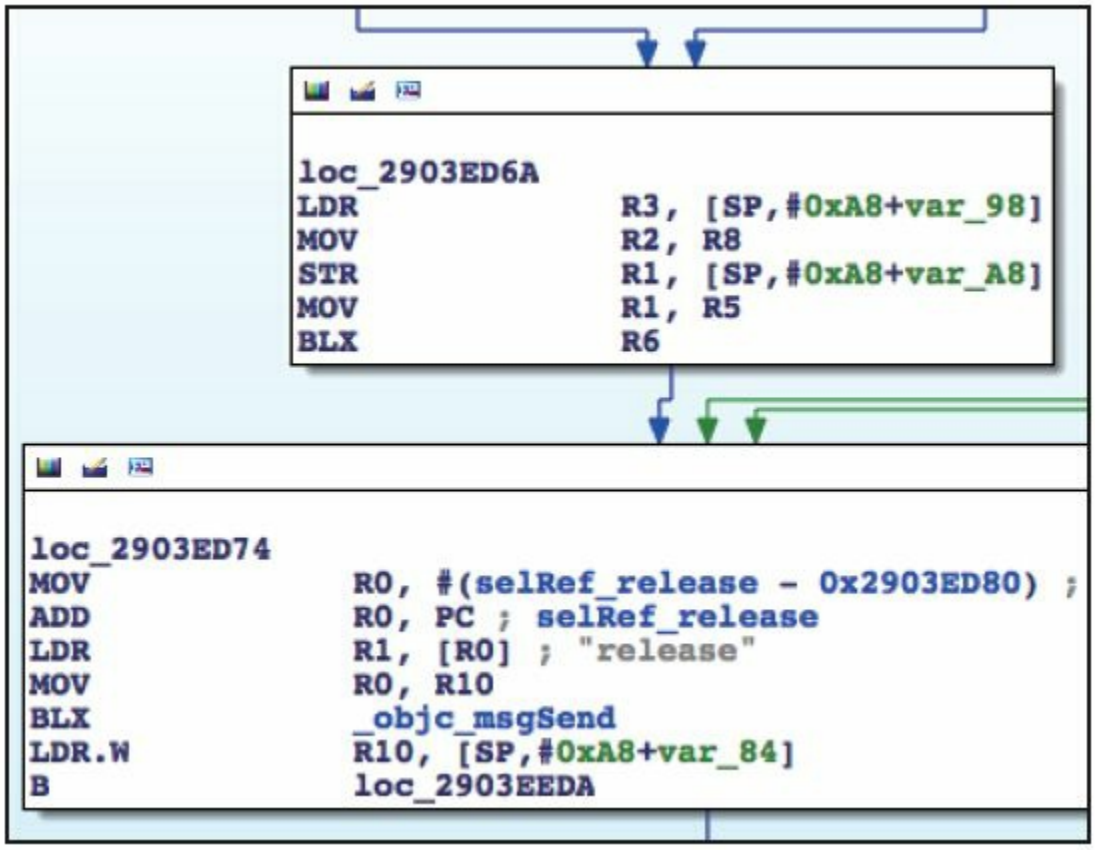


图10-34　sub_26984444的调用者

又是一个来自sub_2903E824的隐式调用，且它上面的4条指令又有2条跟SP相关。为方便阅读，下面把“BLX R6”前后两个模块的指令给拼到一张图里，

拼接前后的效果如图10-35和图10-36所示。

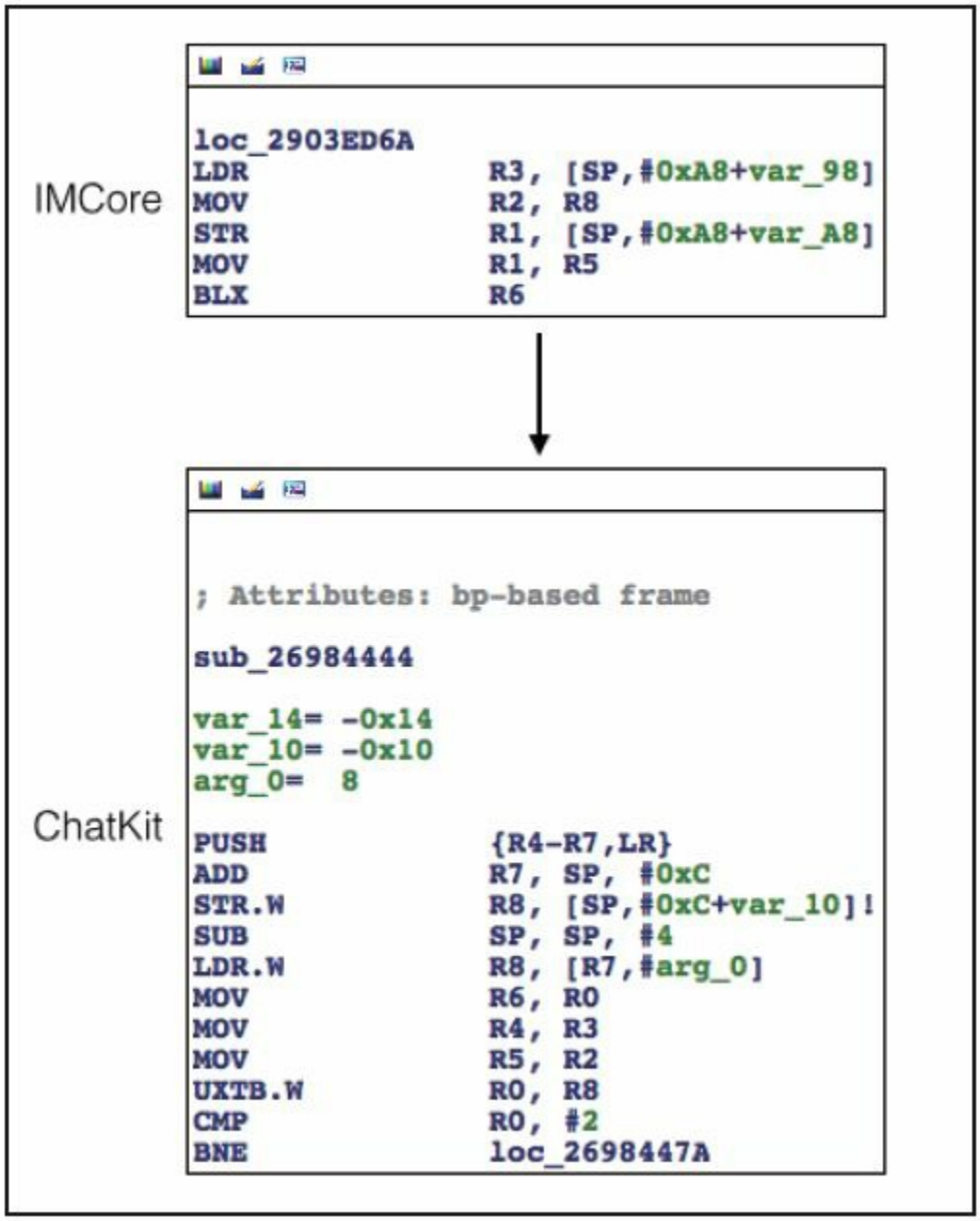


图10-35　拼接前

这里先寻找十四重数据源A，它被写入了*(SP2+0x14)，还记得吗？模仿上面的步骤，把

loc_2903ED6A里的SP标号，如图10-37所示。

然后从loc_2903ED6A的第一条指令开始，看看这里的SP是怎么变化的：

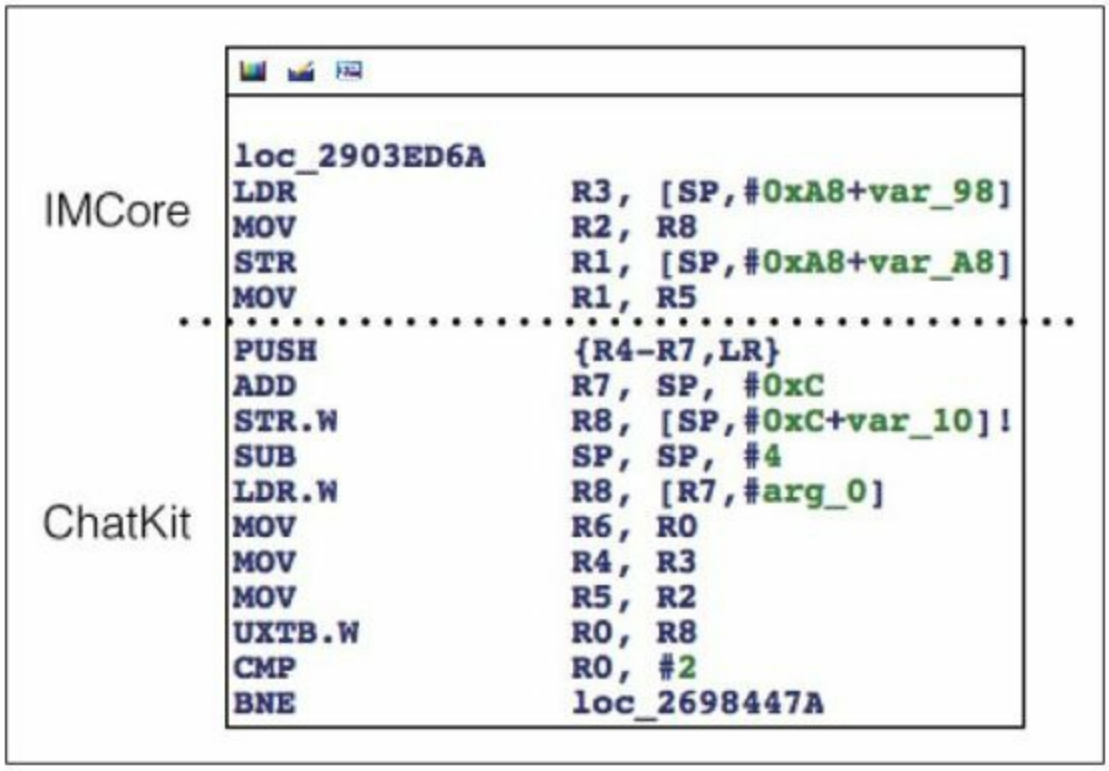


图10-36　拼接后

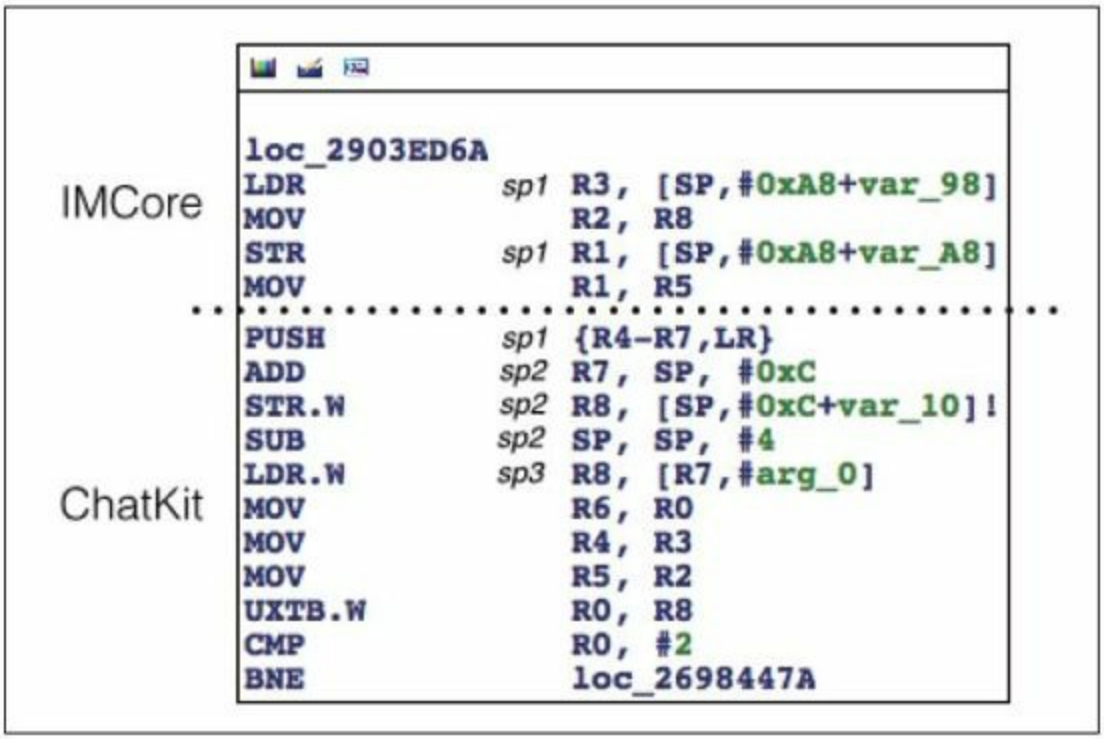


图10-37　标记不同的SP

“LDR R3,[SP,#0xA8+var_98]”，即R3=*(SP1+0xA8+var_98)，而var_98=-0x98（如图10-38所示）。

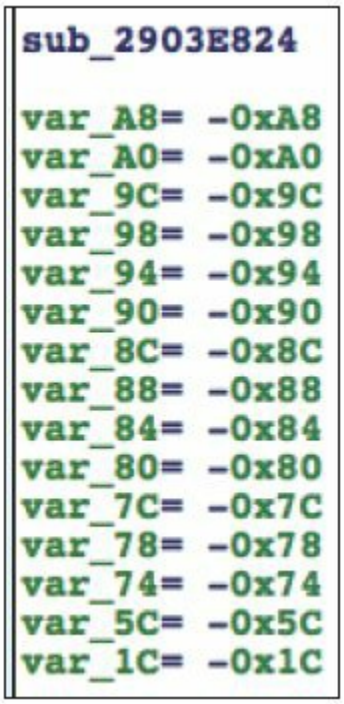


图10-38　sub_2903e824

因此R3=*(SP1+0x10)，且本条指令不影响SP的值；“MOV R2,R8”跟SP无关；“STR R1, [SP,#0xA8+var_A8]”中的var_A8=-0xA8，即*SP1=R1，且本条指令也不影响SP的值；“MOV R1,R5”跟SP无关。这么多SP可能会把你绕晕，再梳理一下其中的逻辑：

目的：找到写入*(SP2+0x14)的地方

因为SP2=SP1-0x14

且*SP1=R1

所以，“STR R1,[SP,#0xA8+var_A8]”就是写入*(SP2+0x14)的地方，十四重数据源A就是这条指令中的R1！而十二重数据源B，显然就是“MOV R1,R5”中的R5。从十三重数据源A到十四重数据源A，从十一重数据源B到十二重数据源B的追踪跨模块，逻辑比较复杂，用图10-39来展示会较为直观，建议大家对照着这张图，把这个跨模块的地方弄清楚。

在继续分析之前，先用LLDB来验证一下到此为止的判断：重输地址，然后在“STR R1, [SP,#0xA8+var_A8]”上设置断点，打印R1，即十四重数据源A；执行“ni”命令到“MOV R1,R5”，打印R5，即十二重数据源B；接着要经历一次模块切换，执行“si”命令到“CMP R0,#2”，打印R0，即十三重数据源A；执行“ni”命令到“TST.W R1,#0xFF”，打印R1，即十一重数据源B。点击“return”，触发断点后，看看

它们的值是不是像图10-39里示意的那样两两相等，如下：

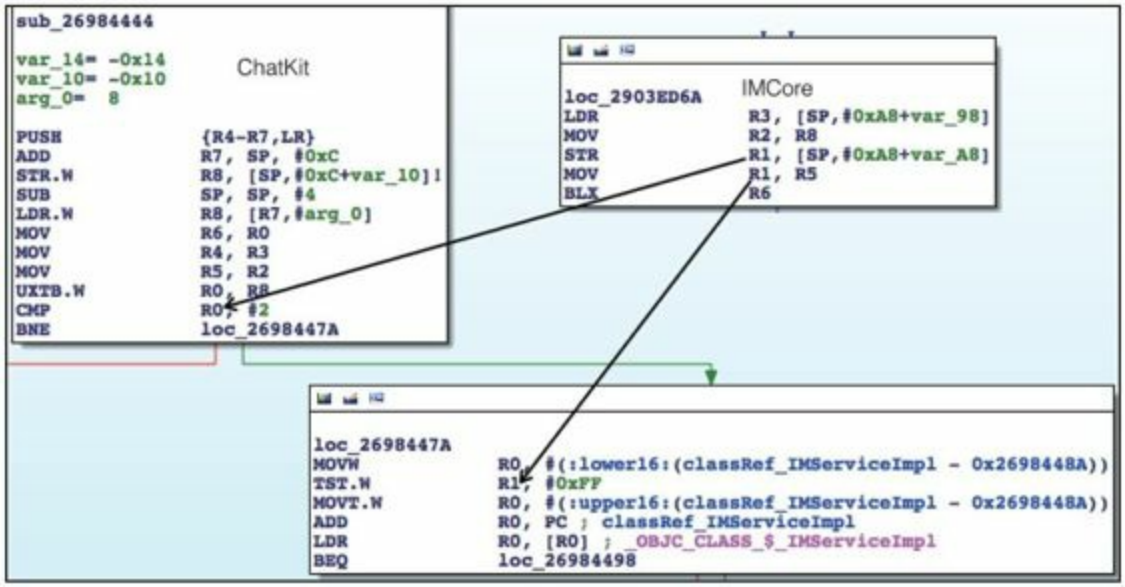


图10-39　数据源间的关系

```
(lldb) br s -a 0x30230D6E
Process 37477 stopped
* thread #1: tid = 0x9265, 0x30230d6e
IMCore`___lldb_unnamed_function425$$IMCore + 1354, queue
= 'com.apple.main-thread, stop reason = breakpoint 11.1
    frame #0: 0x30230d6e
IMCore`___lldb_unnamed_function425$$IMCore + 1354
IMCore`___lldb_unnamed_function425$$IMCore + 1354:
-> 0x30230d6e:  str    r1, [sp]
   0x30230d70:  mov    r1, r5
   0x30230d72:  blx    r6
   0x30230d74:  movw   r0, #26972
(lldb) p $r1
(unsigned int) $27 = 0
(lldb) ni
Process 37477 stopped
* thread #1: tid = 0x9265, 0x30230d70
IMCore`___lldb_unnamed_function425$$IMCore + 1356, queue
= 'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x30230d70
```

```
IMCore`___lldb_unnamed_function425$$IMCore + 1356
IMCore`___lldb_unnamed_function425$$IMCore + 1356:
-> 0x30230d70:  mov    r1, r5
   0x30230d72:  blx    r6
   0x30230d74:  movw   r0, #26972
   0x30230d78:  movt   r0, #2081
(lldb) p $r5
(unsigned int) $28 = 1
(lldb) ni
Process 37477 stopped
* thread #1: tid = 0x9265, 0x30230d72
IMCore`___lldb_unnamed_function425$$IMCore + 1358, queue
= 'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x30230d72
IMCore`___lldb_unnamed_function425$$IMCore + 1358
IMCore`___lldb_unnamed_function425$$IMCore + 1358:
-> 0x30230d72:  blx    r6
   0x30230d74:  movw   r0, #26972
   0x30230d78:  movt   r0, #2081
   0x30230d7c:  add    r0, pc
(lldb) si
Process 37477 stopped
* thread #1: tid = 0x9265, 0x2db76444 ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 queue = 'com.apple.main-thread, stop reason =
instruction step into
    frame #0: 0x2db76444 ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv

ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv

-> 0x2db76444:  push   {r4, r5, r6, r7, lr}
   0x2db76446:  add    r7, sp, #12
   0x2db76448:  str    r8, [sp, #-4]!
   0x2db7644c:  sub    sp, #4
(lldb) ni
……
Process 37477 stopped
* thread #1: tid = 0x9265, 0x2db7645c ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 24, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x2db7645c ChatKit`__71-
[CKPendingConversation
```

```
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 24
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 24:
-> 0x2db7645c:  cmp    r0, #2
   0x2db7645e:  bne    0x2db7647a                ; __71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 54
   0x2db76460:  movw   r0, #19376
   0x2db76464:  movt   r0, #2535
(lldb) p $r0
(unsigned int) $29 = 0
(lldb) ni
……
Process 37477 stopped
* thread #1: tid = 0x9265, 0x2db7647e ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 58, queue = 'com.apple.main-thread, stop reason =
instruction step over
    frame #0: 0x2db7647e ChatKit`__71-
[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 58
ChatKit`__71-[CKPendingConversation
refreshStatusForAddresses:withCompletionBlock:]_block_inv
 + 58:
-> 0x2db7647e:  tst.w  r1, #255
   0x2db76482:  movt   r0, #2535
   0x2db76486:  add    r0, pc
   0x2db76488:  ldr    r0, [r0]
(lldb) p $r1
(unsigned int) $30 = 1
```

打印结果验证了分析结果，且十四重数据源A的值是0，十二重数据源B的值是1。接下来把火力集中在IMCore上，继续寻找十五重数据源A和十三重数据源B，先从前者下手。

十五重数据源A已经直观表现在了图10-40中。

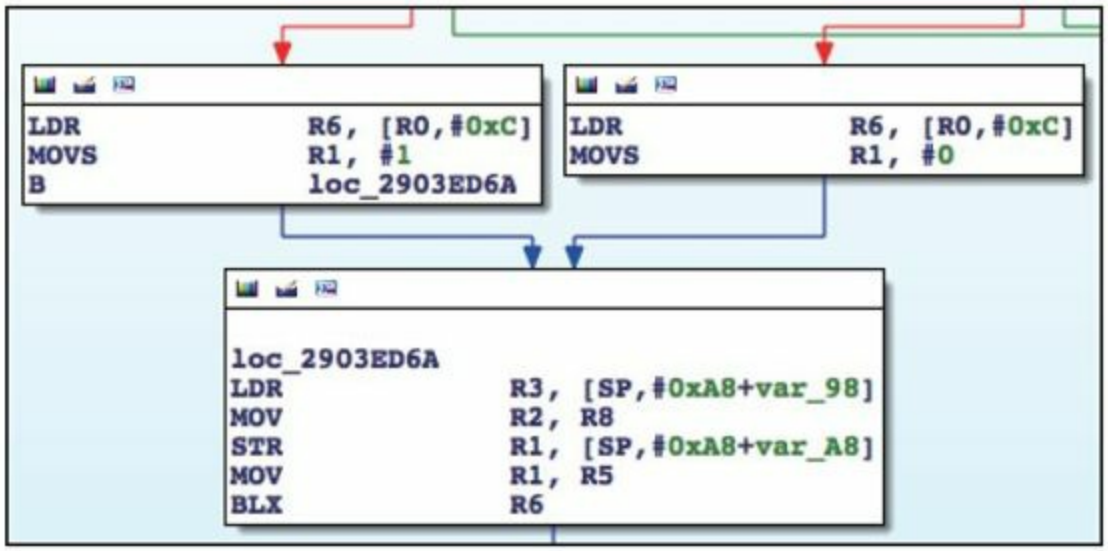


图10-40　十五重数据源A

它要么来自“MOVS R1,#1”，要么来自“MOVS R1,#0”，也就是说十五重数据源的值要么是0，要么是1。事情变得有意思起来了……

不知道大家有没有注意到，从十一重数据源A开始，数据源A的值就是一脉相承的，即十一重=十二重=十三重=十四重=十五重数据源A=0或1。但是，我们之前分析的伪代码是这样的：

```
- (BOOL)supportIMessage
{
```

```
        if (十一重数据源A == 2 || 十一重数据源B != 0) return
YES;
        return NO;
}
```

而十一重数据源A的值非0即1，是不可能等于2的。这样的话，数据源A已经没有意义了，不是吗？伪代码可以改成：

```
- (BOOL)supportIMessage
{
        if (十一重数据源B != 0) return YES;
        return NO;
}
```

因此，可以把重点放在寻找十三重数据源B上来，以下简称十三重数据源。因为十二重数据源B是R5，所以十三重数据源一定被某条指令写入R5了，对吧？单击R5，IDA会贴心地帮我们将其标记为黄色，方便从大量的汇编代码中定位R5。继续逆向，看看R5是什么时候被写入的。

当向上寻找十三重数据源，跟踪到loc_2903EAE0时，会发现它的上游有4条分支（如图10-41所示）。

图10-41　loc_2903EAE0

在图10-41中，从左到右的3条分支中均含有“MOVS R5,#0”的操作，但这跟R5=1的结果是矛盾的，因此loc_2903EAE0一定是经由最右边的那一条线路到达的，十三重数据源就位于这条线路上，顺着这条线路继续向上寻找R5的踪影。

跟踪到loc_2903EA3E时，出现的情况似曾相识。它的上游虽然有3条分支，但左1和左2分支均含有“MOVS R5,#0”的操作（如图10-42所示），因此可以直接排除。

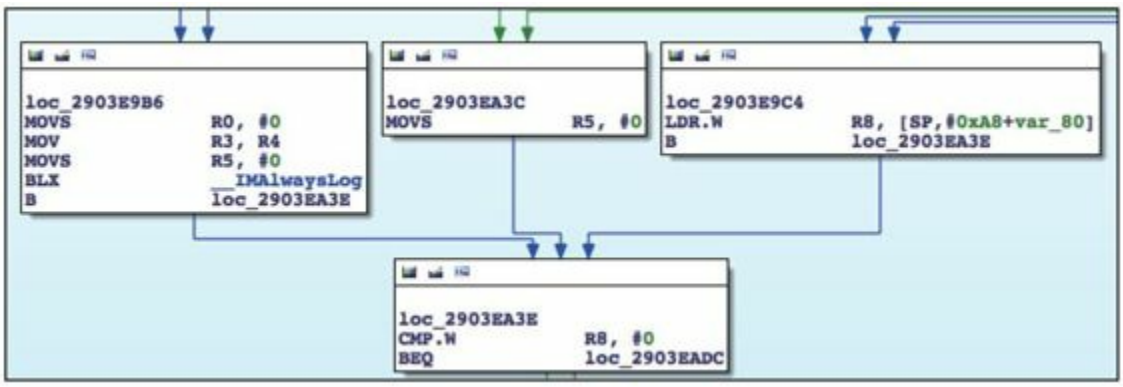


图10-42　loc_2903EA3E

它的实际上游是左3分支，即loc_2903E9C4；而loc_2903E9C4的上游有2条分支，其中均含有“MOVS R5,#1”的操作，那么进程的实际执行流程是从这2条分支中的哪一条到达loc_2903E9C4的呢？重输地址，在2个分支的跳转处各下一个断点，点击键盘上的“return”，看看会触发哪个断点，这样就一清二楚了。这里的操作流程笔者就省略掉了，请读者独立完成，相信在简单的操作之后，你也会发现，左1分支才是进程实际执行的流程，即图10-43。

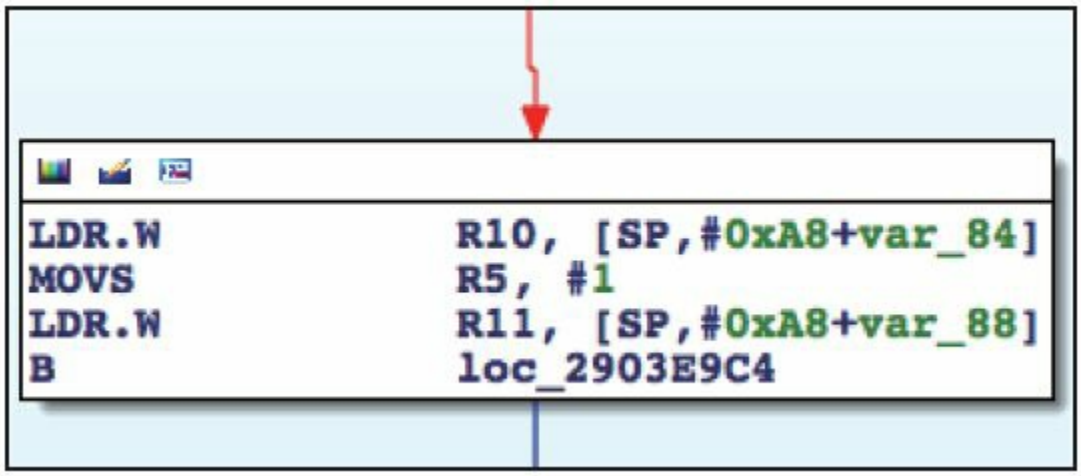


图10-43　左1分支

现在，找到了十三重数据源，它是常量1。你可能会问，十三重数据源是常量的话，十四重数据源还存在吗？数据源的线索看似中断了，下一步该怎么办？问得好！

在刚才的代码中，我们看到了几处“MOVS R5,#0”的操作，而十三重数据源来自“MOVS R5,#1”，看似数据源是常量，但按照程序设计的思想，到底往R5里写0还是写1，应该由一个条件判断来决定，就像下面的伪代码：

```
if (iMessageIsAvailable) R5 = 1;
else R5 = 0;
```

用熟悉的IDA流程图表示，就是图10-44所示的样子。

从宏观角度来讲，其实这个条件判断就是十四重数据源，不是吗？相信你也反应过来了，上面的伪代码其实就是：

```
R5 = iMessageIsAvailable;
```

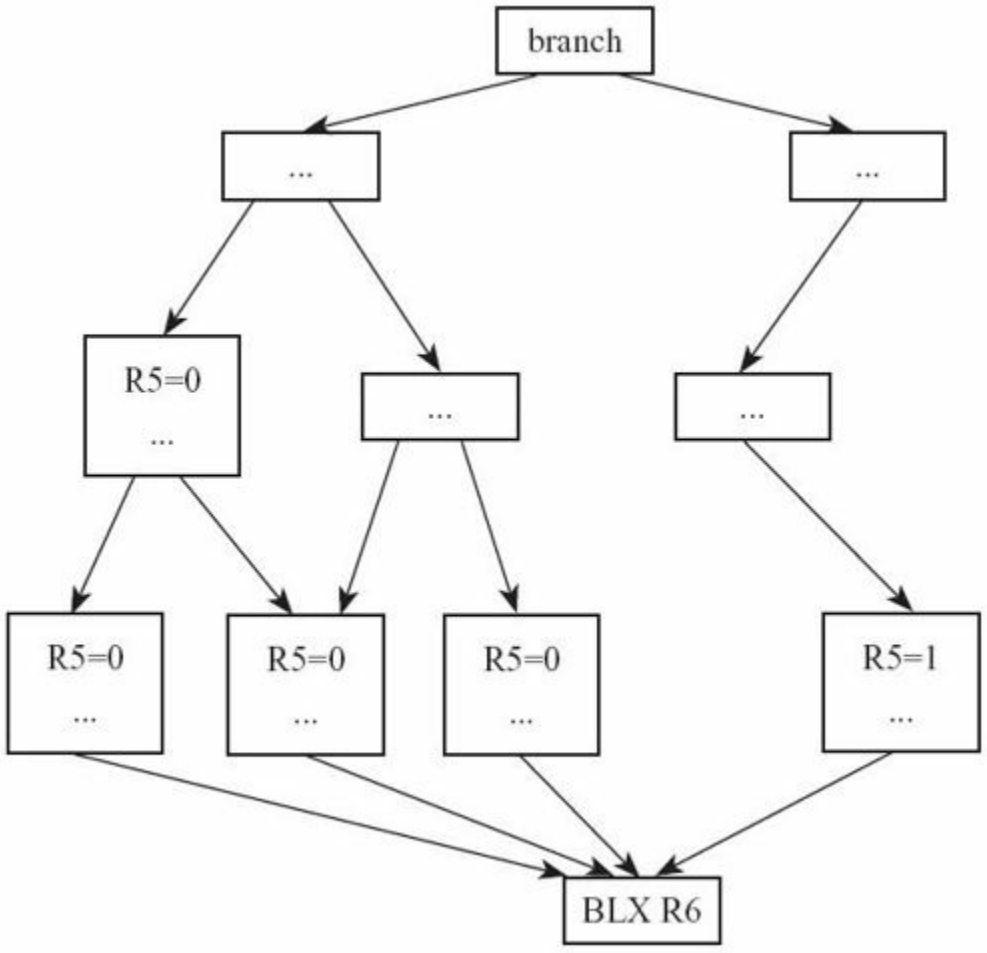


图10-44　伪IDA流程图

弄清了这个概念，接下来的任务就是继续回溯，分析每个出现分支的地方，如果它的不同分支会往R5里写不同的值，就要搞清楚分支条件是什么，而这个分支条件就是我们要找的数据源。到图10-45所示的代码段里去看看。

如果分支走了左边，R5是有可能被置0的。由于分支条件是objc_msgSend的返回值，因此下个断点看看执行的到底是什么函数，如下：

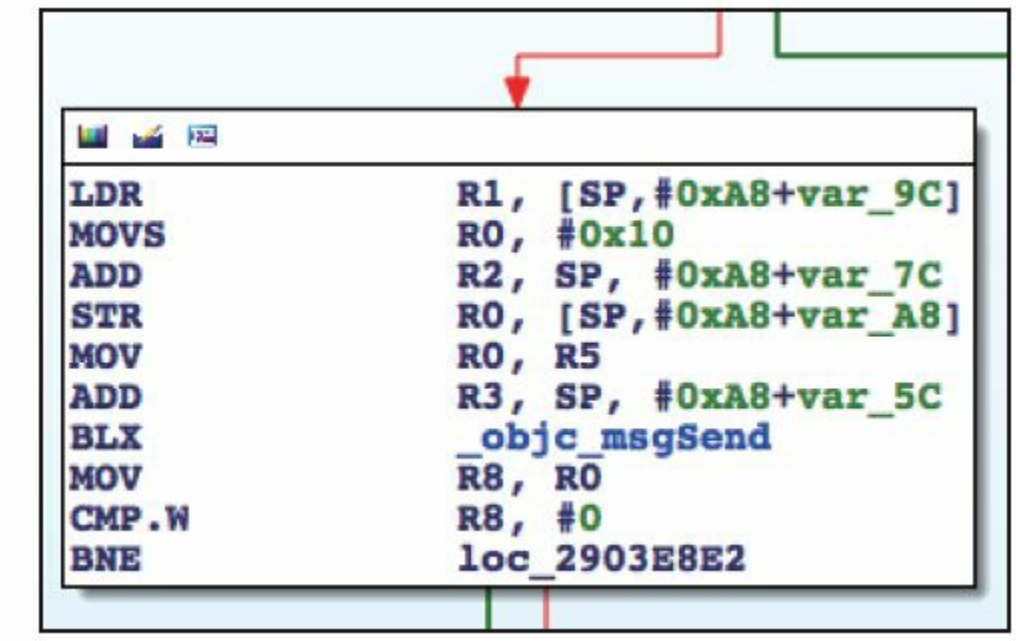


图10-45　分支

```
Process 132234 stopped
* thread #1: tid = 0x2048a, 0x331f092e
IMCore`___lldb_unnamed_function425$$IMCore + 266, queue =
'com.apple.main-thread, stop reason = breakpoint 5.1
    frame #0: 0x331f092e
IMCore`___lldb_unnamed_function425$$IMCore + 266
IMCore`___lldb_unnamed_function425$$IMCore + 266:
-> 0x331f092e:  blx    0x332603b0     ; symbol stub for:
objc_msgSend
   0x331f0932:  mov    r8, r0
   0x331f0934:  cmp.w  r8, #0
   0x331f0938:  bne    0x331f08e2     ;
___lldb_unnamed_function425$$IMCore + 190
(lldb) p (char *)$r1
(char *) $6 = 0x2f7d81d9
"countByEnumeratingWithState:objects:count:"
(lldb) po $r0
<__NSArrayI 0x16706930>(
mailto:snakeninny@gmail.com
)
```

可以看到，是一个对收件人队列的遍历函数，如果收件人队列不为空，分支就会走右边。实际上收件人队列肯定不会为空，因此这个分支条件不会成立，类似于已弃用的数据源A，是不会产生分支的。继续往上，寻找上一个分支发生的地方，如图10-46所示。

在图10-46中，R11和R8各是什么？在IDA里可以

很直观地看到，R11来自图10-47。

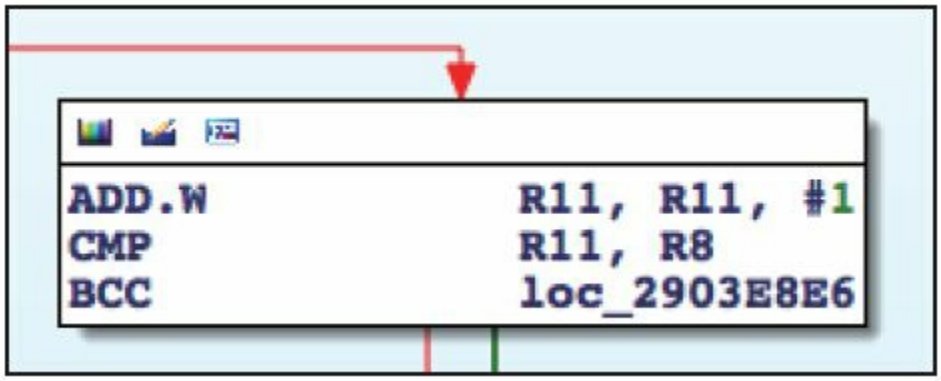


图10-46　分支

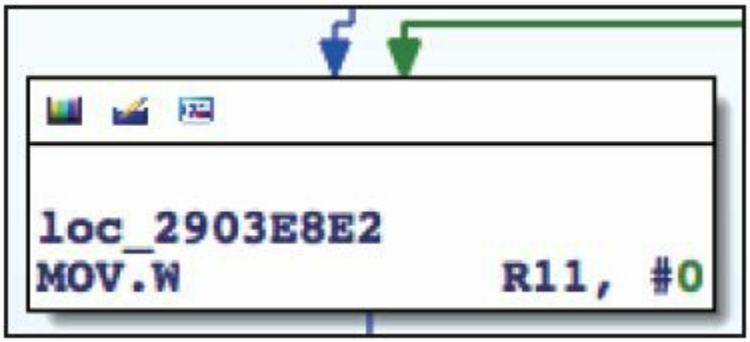


图10-47　loc_2903e8e2

R11的初始值是0，每次在执行“CMP R11,R8”之前，R11都递增1。这么看起来，R11充当了计数器的作用。“CMP”执行了减法操作，如果产生借位，则将Carry置0，否则Carry置1。这里的分支指令是“BCC”，“CC”代表“Carry Clear”，也就是Carry位为

0。也就是说，如果R11–R8产生借位，即R8大于R11，则分支走右边，否则走左边。我们看看R8是什么，如图10-48所示。

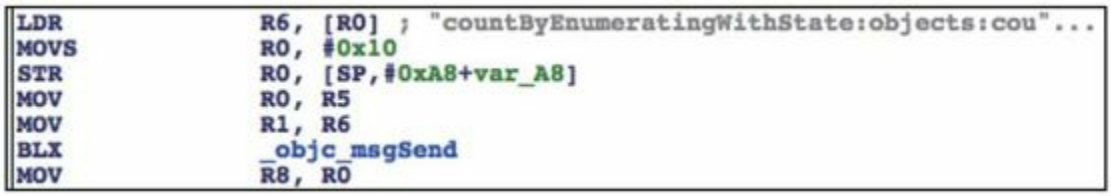


图10-48　R8来源

R8来自[NSArray countByEnumeratingWithState:objects:count:]。重输地址，下断点，点击“return，”看看NSArray是什么，如下：

```
(lldb) br s -a 0x3023089C
Breakpoint 2: where =
IMCore`___lldb_unnamed_function425$$IMCore + 120, address
= 0x3023089c
Process 102482 stopped
* thread #1: tid = 0x19052, 0x3023089c
IMCore`___lldb_unnamed_function425$$IMCore + 120, queue =
'com.apple.main-thread, stop reason = breakpoint 2.1
    frame #0: 0x3023089c
IMCore`___lldb_unnamed_function425$$IMCore + 120
IMCore`___lldb_unnamed_function425$$IMCore + 120:
-> 0x3023089c:  blx    0x302a03b0                ; symbol
stub for: objc_msgSend
   0x302308a0:  mov    r8, r0
   0x302308a2:  cmp.w  r8, #0
   0x302308a6:  beq.w  0x302309c2                ;
```

```
___lldb_unnamed_function425$$IMCore + 414
(lldb) p (char *)$r1
(char *) $5 = 0x2c8181d9
"countByEnumeratingWithState:objects:count:"
(lldb) po $r0
<__NSArrayI 0x178d6b20>(
mailto:snakeninny@gmail.com
)
```

NSArray是收件人队列，因此R8是收件人个数，如果收件人个数大于1，在第一次执行“CMP R11,R8”时R11是1，则R8大于R11，分支走右边，到达图10-49。

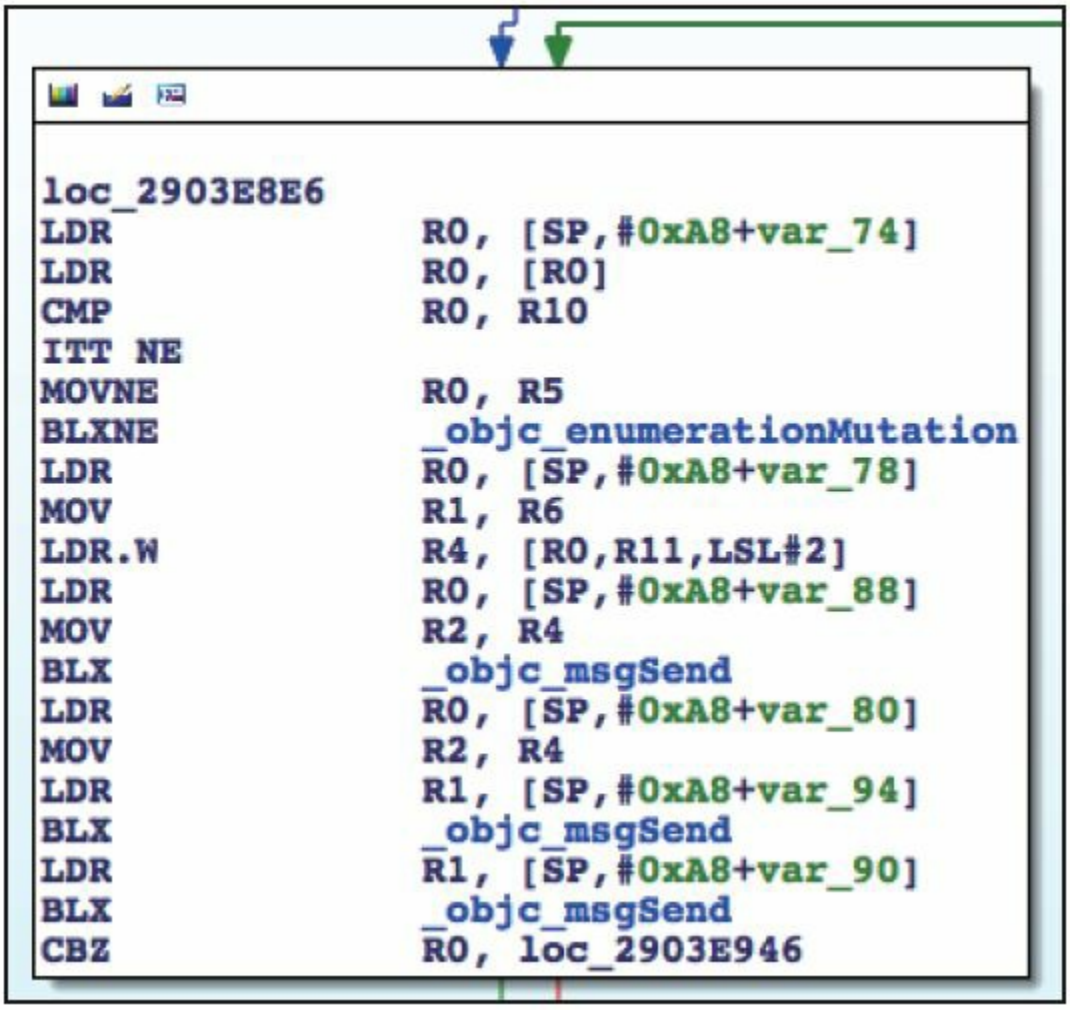


图10-49　分支

loc_2903E8E6的分支条件是R0，如果R0==0则走左边，不支持iMessage；否则走右边，到达图10-50。

图10-50的分支条件仍是R0，如果R0==2则走左边，不支持iMessage；否则走右边，回到图10-46。这3段代码循环没有更改R8的值，只要loc_2903E8E6最

下面的R0!=0&&R0!=2，图10-46的分支就没有意义——R11一直递增，而R8不变，这个分支早晚会走左边，得出支持iMessage的结论；也就是说，在这个循环里，本质的分支条件是R0的值。还记得我们刚刚得出的结论吗——“分析每个出现分支的地方，如果它的不同分支会往R5里写不同的值，就要搞清楚分支条件是什么，而这个分支条件就是我们要找的数据源。”——R0就是十四重数据源。

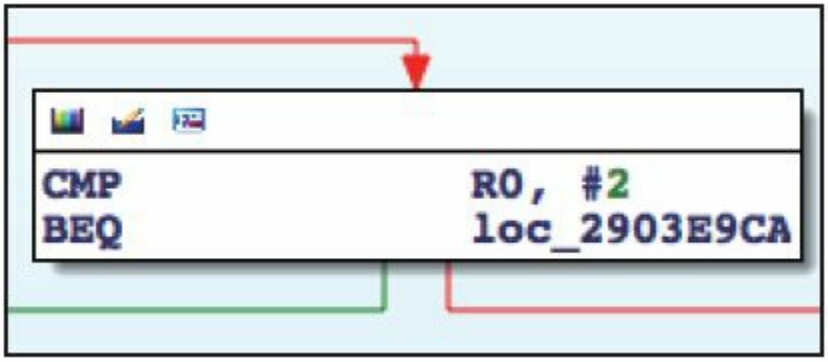


图10-50　分支

下面用LLDB看看这里的几个objc_msgSend都是什么，R0是怎么来的，如下：

```
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0900
IMCore`___lldb_unnamed_function425$$IMCore + 220, queue =
```

```
'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x331f0900
IMCore`___lldb_unnamed_function425$$IMCore + 220
IMCore`___lldb_unnamed_function425$$IMCore + 220:
-> 0x331f0900:  blx    0x332603b0                ; symbol
stub for: objc_msgSend
   0x331f0904:  ldr    r0, [sp, #40]
   0x331f0906:  mov    r2, r4
   0x331f0908:  ldr    r1, [sp, #20]
(lldb) p (char *)$r1
(char *) $7 = 0x2f7d897a "removeObject:"
(lldb) po $r0
<__NSArrayM 0x170ec120>(
mailto:snakeninny@gmail.com
)
(lldb) po $r2
mailto:snakeninny@gmail.com
(lldb) ni
……
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f090a
IMCore`___lldb_unnamed_function425$$IMCore + 230, queue =
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x331f090a
IMCore`___lldb_unnamed_function425$$IMCore + 230
IMCore`___lldb_unnamed_function425$$IMCore + 230:
-> 0x331f090a:  blx    0x332603b0                ; symbol
stub for: objc_msgSend
   0x331f090e:  ldr    r1, [sp, #24]
   0x331f0910:  blx    0x332603b0                ; symbol
stub for: objc_msgSend
   0x331f0914:  cbz    r0, 0x331f0946            ;
___lldb_unnamed_function425$$IMCore + 290
(lldb) p (char *)$r1
(char *) $10 = 0x2f7d8113 "valueForKey:"
(lldb) po $r2
mailto:snakeninny@gmail.com
(lldb) po $r0
{
    "mailto:snakeninny@gmail.com" = 1;
}
(lldb) po [$r0 class]
__NSCFDictionary
(lldb) ni
……
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0910
IMCore`___lldb_unnamed_function425$$IMCore + 236, queue =
```

```
'com.apple.main-thread, stop reason = instruction step
over
    frame #0: 0x331f0910
IMCore`___lldb_unnamed_function425$$IMCore + 236
IMCore`___lldb_unnamed_function425$$IMCore + 236:
-> 0x331f0910:  blx    0x332603b0                ; symbol
stub for: objc_msgSend
   0x331f0914:  cbz    r0, 0x331f0946            ;
___lldb_unnamed_function425$$IMCore + 290
   0x331f0916:  cmp    r0, #2
   0x331f0918:  beq    0x331f09ca                ;
___lldb_unnamed_function425$$IMCore + 422
(lldb) p (char *)$r1
(char *) $14 = 0x2f7de6f3 "integerValue"
(lldb) po $r0
1
(lldb) po [$r0 class]
__NSCFNumber
(lldb) c
```

将这3个objc_msgSend还原成ObjC函数，分别是[NSArray removeObject:@"mailto:snakeninny@gmail.com"]、[NSDictionary valueForKey：@"mailto:snakeninny@gmail.com"]和[NSNumber integerValue]，其中第二个objc_msgSend的R0值得关注，正是它（NSDictionary）中包含的键值对，决定了十四重数据源；因此，这个NSDictionary就是十五重数据源。由图10-49可知，它来自于[SP,#0xA8+var_80]，因此[SP,#0xA8+var_80]是十六重

数据源。接下来的套路已经做过好几遍了：把光标放在var_80上，然后按下“x”，看看它的交叉引用，如图10-51所示。

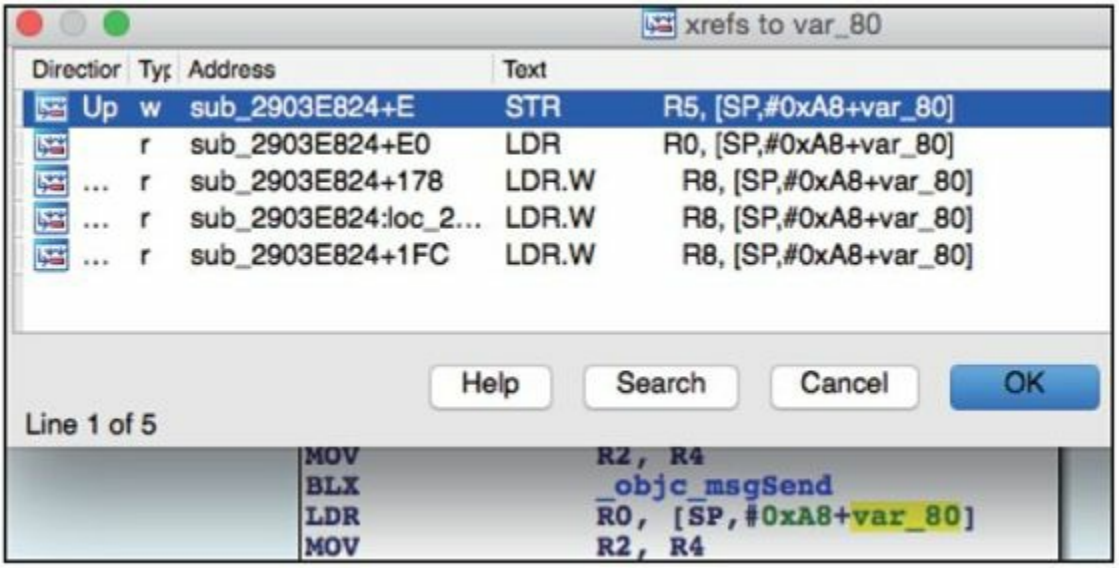


图10-51　查看交叉引用

可以看到，只有一处指令写入了这个地址，双击这条指令，直接跳转到了sub_2903E824的头部，如图10-52所示。

```
sub_2903E824

var_A8= -0xA8
var_A0= -0xA0
var_9C= -0x9C
var_98= -0x98
var_94= -0x94
var_90= -0x90
var_8C= -0x8C
var_88= -0x88
var_84= -0x84
var_80= -0x80
var_7C= -0x7C
var_78= -0x78
var_74= -0x74
var_5C= -0x5C
var_1C= -0x1C

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
PUSH.W          {R8,R10,R11}
SUB             SP, SP, #0x90
MOV             R5, R1
STR             R2, [SP,#0xA8+var_98]
STR             R5, [SP,#0xA8+var_80]
STR             R0, [SP,#0xA8+var_8C]
```

图10-52　sub_2903E824

十六重数据源来自于R5，故R5是十七重数据源；十七重数据源又来自于R1，因此R1是十八重数据源，而它没有被赋值就直接取值了，说明R1来自sub_2903E824的调用者，对吧？看看它的交叉引用，如图10-53所示。

图10-53　查看交叉引用

“为发送新的信息计算服务”这个名字的含义已经很明显了，双击第一条交叉引用，去它的显式调用者那瞅瞅，如图10-54所示。

这里要先确认一下sub_2903E824的调用者是不是来自“IMChatCalculateServiceForSending-NewCompose”——清空地址输入框，输入地址，在sub_2903E824的第一条指令上下一个断点，点击键盘上的“return”，触发断点，如下：

```
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0824
IMCore`___lldb_unnamed_function425$$ IMCore, queue =
'com.apple.main-thread, stop reason = breakpoint 2.1
    frame #0: 0x331f0824
IMCore`___lldb_unnamed_function425$$IMCore
IMCore`___lldb_unnamed_function425$$IMCore:
-> 0x331f0824:  push   {r4, r5, r6, r7, lr}
   0x331f0826:  add    r7, sp, #12
   0x331f0828:  push.w {r8, r10, r11}
```

```
   0x331f082c:  sub    sp, #144
(lldb) p/x $lr
(unsigned int) $17 = 0x331f067b
(lldb)
```

```
LDR.W           R6, [R9] ; "sharedInstance"
LDR             R0, [R1] ; _OBJC_CLASS_$_IDSIDQueryController
MOV             R1, R6
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(_IDSServiceNameiMessage_ptr - 0x2903E64C))
MOV             R11, R4
MOVT.W          R1, #(:upper16:(_IDSServiceNameiMessage_ptr - 0x2903E64C))
MOVW            R2, #(:lower16:(selRef__currentIDStatusForDestinations_service_list
ADD             R1, PC ; _IDSServiceNameiMessage_ptr
MOVT.W          R2, #(:upper16:(selRef__currentIDStatusForDestinations_service_list
LDR             R3, [R1]
ADD             R2, PC ; selRef__currentIDStatusForDestinations_service_listenerID_
MOVW            R5, #(:lower16:(cfstr___kimchatservi - 0x2903E662)) ; "__kIMChatSer
LDR             R1, [R2] ; "_currentIDStatusForDestinations:service"...
MOVT.W          R5, #(:upper16:(cfstr___kimchatservi - 0x2903E662)) ; "__kIMChatSer
MOV             R2, R4
ADD             R5, PC  ; "__kIMChatServiceForSendingIDSQueryControllerListenerID"
LDR.W           R10, [R3]
STR             R5, [SP,#0x64+var_64]
MOV             R3, R10
BLX             _objc_msgSend
MOV             R5, R0
ADD             R0, SP, #0x64+var_38
MOV             R1, R5
MOVS            R2, #0
MOV             R8, R0
BL              sub_2903E824
```

图10-54　sub_2903E824的调用者

这里的ASLR偏移是0xa1b2000，所以调用者的实际地址是0x2903E67B，正是来自“IMChatCalculateServiceForSendingNewCompose”——十八重数据源来自R5，因此R5是十九重数据源；而十九重数据源来自objc_msgSend的返回值，故而该返回值是二十重数据源。万事俱备，只欠东风，我们看看这个神秘的objc_msgSend到底做了些什么，如下：

```
Process 154446 stopped
* thread #1: tid = 0x25b4e, 0x331f0668
IMCore`IMChatCalculateServiceForSendingNewCompose + 688,
queue = 'com.apple.main-thread, stop reason = breakpoint
3.1
    frame #0: 0x331f0668
IMCore`IMChatCalculateServiceForSendingNewCompose + 688
IMCore`IMChatCalculateServiceForSendingNewCompose + 688:
-> 0x331f0668:  blx    0x332603b0                ; symbol
stub for: objc_msgSend
   0x331f066c:  mov    r5, r0
   0x331f066e:  add    r0, sp, #44
   0x331f0670:  mov    r1, r5
(lldb) p (char *)$r1
(char *) $18 = 0x33274340
"_currentIDStatusForDestinations:service:listenerID:"
(lldb) po $r0
<IDSIDQueryController: 0x15dcb010>
(lldb) po $r2
<__NSArrayM 0x170e7900>(
mailto:snakeninny@gmail.com
)
(lldb) po $r3
com.apple.madrid
(lldb) po [$r3 class]
__NSCFConstantString
(lldb) x/10 $sp
0x001e4548: 0x3b3f52b8 0x001e459c 0x3b4227b4 0x3c01b05c
0x001e4558: 0x00000001 0x00000000 0x170828d0 0x001e4594
0x001e4568: 0x2baac821 0x00000000
(lldb) po 0x3b3f52b8
__kIMChatServiceForSendingIDSQueryControllerListenerID
(lldb) po [0x3b3f52b8 class]
__NSCFConstantString
(lldb) c
```

锲而不舍，终有斩获。这个objc_msgSend还原之后，是[[IDSIDQueryController sharedInstance]_currentIDStatusForDestinations:@[@"ma 因为后两个参数是常量，所以可变参数只有第一个数

组，也就是收件人数组，我们终于跟踪到了原始数据源！

笔者知道本节的内容很难，你可能已经被绕晕了，但行百里者半九十，还差最后一步了，打起精神来！

10.2.5 还原原始数据源生成placeholderText的过程

函数都被我们找到了，貌似可以通过更改第一个NSArray参数，来达到检测任意目标地址是否支持iMessage的效果，只要它的返回值（NSDictionary）中key所对应的value非0且非2，则key支持iMessage，否则key仅支持SMS。真的是这样吗？我们已经知道，对于邮件地址来说，参数格式为“mailto:”，那电话号码的参数格式呢？在_currentIDStatus-ForDestinations:service:listenerID:上下个断点看一看，如下：

```
Process 102482 stopped
* thread #1: tid = 0x19052, 0x30230668
IMCore`IMChatCalculateServiceForSendingNewCompose + 688,
queue = 'com.apple.main-thread, stop reason = breakpoint
6.1
    frame #0: 0x30230668
IMCore`IMChatCalculateServiceForSendingNewCompose + 688
IMCore`IMChatCalculateServiceForSendingNewCompose + 688:
-> 0x30230668:  blx    0x302a03b0                ; symbol
stub for: objc_msgSend
   0x3023066c:  mov    r5, r0
   0x3023066e:  add    r0, sp, #44
   0x30230670:  mov    r1, r5
(lldb) po $r2
<__NSArrayM 0x17820560>(
tel:+86PhoneNumber
)
```

LLDB和debugserver可以暂时休息一下了。回到Cycript中，实际验证一下猜测，如下：

```
FunMaker-5:~ root# cycript -p MobileSMS
cy# [[IDSIDQueryController sharedInstance]
_currentIDStatusForDestinations:@[@"mailto:snakeninny@gma
 @"mailto:snakeninny@icloud.com", @"tel:bbs.iosre.com",
@"mailto:bbs.iosre.com", @"tel:911",
@"tel:+86PhoneNumber"] service:@"com.apple.madrid"
listenerID:@"__kIMChatServiceForSendingIDSQueryController
 ":1}
```

哈哈，输出的结果再清楚不过了，2个支持iMessage的邮箱和1个手机号均返回了1，而另3个不支持iMessage的地址返回了2，实验结果验证了分析，而且还知道了iMessage的内部称呼为“Madrid”。

任务完成，万岁！

10.3 发送iMessage

经过10.2节的洗礼，相信部分读者会产生跟笔者相同的感觉：一步一步用LLDB调试虽然准确严谨，但工作量巨大，容易让人感到厌倦。逆向工程就是要勇于试错，不走寻常路，用跳跃的思维和大胆的猜想来达到目的。本节就会采用这种方式——尽量少地使用LLDB，尽量多地通过IDA和class-dump中看到的关键词，并用Cycript配合联想来达到发送iMessage的目的。

10.3.1 从MobileSMS界面元素寻找逆向切入点

相对于检测iMessage，发送iMessage的切入点就要明显得多，在图10-55所示的iOS截图上，这个大大的“Send”按钮，不就是苹果送给我们的大礼么？

图10-55　显眼的“Send”

按下“Send”，发送一条iMessage，这就是发送iMessage最直观的表现。跟10.2节一样，先想想怎么把这个现象给具象化成逆向工程：

“Send”按钮是一个UIView，具体地说，可能是一个UIButton；点击这个UIButton，调用UIButton的响应动作；全套响应动作包括更新界面、发送信息、添加已发送记录，等等，也就是说，发送信息的操作只是全套响应动作的子集。

在MobileSMS发送界面中，我们的输入只有收件人地址和信息内容，它们是原始数据源。因为可以拿到全套响应动作，而发送信息的操作一定要以原始数据源为参数，所以可以根据这2个条件，在全套响应动作里筛选出发送信息的操作。与上节的终点倒推起点不同，这次将从起点到达终点，演示逆向工程的另一种思路。

总结一下，逆向工程的思路是这样的：先用Cycript定位“Send”按钮的响应函数，然后用IDA纵览全套响应动作，结合LLDB和数据源，寻找可疑的发送操作。

10.3.2 用Cycript找出“Send”按钮的响应函数

因为前面在10.2节中已经找到了“Send”所在的view是一个CKMessageEntryView，所以这里就可以直接重复10.2.2节的分析思路，得出下面的结果：

```
cy# ?expand
expand == true
cy# [UIApp windows]
@[#"<UIWindow: 0x14e12fa0; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x14e11f50>; layer =
<UIWindowLayer: 0x14ee4570>>",#"<UITextEffectsWindow:
0x14fa6000; frame = (0 0; 320 568); opaque = NO;
gestureRecognizers = <NSArray: 0x14fa66d0>; layer =
<UIWindowLayer: 0x14fa5fc0>>",#"<CKJoystickWindow:
0x14d22310; baseClass = UIAutoRotatingWindow; frame = (0
0; 320 568); hidden = YES; gestureRecognizers = <NSArray:
0x14d21ab0>; layer = <UIWindowLayer: 0x14d22140>>"]
cy# [#0x14fa6000 subviews]
@[#"<UIInputSetContainerView: 0x14d03930; frame = (0 0;
320 568); autoresize = W+H; layer = <CALayer:
0x14d03770>>"]
cy# [#0x14d03930 subviews]
@[#"<UIInputSetHostView: 0x14d033f0; frame = (0 250; 320
318); layer = <CALayer: 0x14d03290>>"]
cy# [#0x14d033f0 subviews]
@[#"<UIKBInputBackdropView: 0x160441a0; frame = (0 65;
320 253); userInteractionEnabled = NO; layer = <CALayer:
0x16043b60>>",#"<_UIKBCompatInputView: 0x14f78a20; frame
= (0 65; 320 253); layer = <CALayer: 0x14f78920>>",#"
<CKMessageEntryView: 0x160c6180; frame = (0 0; 320 65);
opaque = NO; autoresize = W; layer = <CALayer:
0x16089920>>"]
cy# [#0x160c6180 subviews]
@[#"<_UIBackdropView: 0x16069d40; frame = (0 0; 320 65);
opaque = NO; autoresize = W+H; userInteractionEnabled =
NO; layer = <_UIBackdropViewLayer: 0x14d627c0>>",#"
<UIView: 0x16052920; frame = (0 0; 320 0.5); layer =
<CALayer: 0x160529d0>>",#"<UIButton: 0x1605a8b0; frame =
```

```
(266 27; 53 33); opaque = NO; layer = <CALayer:
0x16052a00>>",#"<UIButton: 0x14d0b2c0; frame = (266 30;
53 26); hidden = YES; opaque = NO; gestureRecognizers =
<NSArray: 0x160f9800>; layer = <CALayer: 0x1605a140>>",#"
<UIButton: 0x1606f040; frame = (15 33.5; 25 18.5); opaque
= NO; gestureRecognizers = <NSArray: 0x14d07970>; layer =
<CALayer: 0x1605aaa0>>",#"
<_UITextFieldRoundedRectBackgroundViewNeue: 0x160e5ed0;
frame = (55 8; 209.5 49.5); opaque = NO;
userInteractionEnabled = NO; layer = <CALayer:
0x160d3a10>>",#"<UIView: 0x160a3390; frame = (55 8; 209.5
49.5); clipsToBounds = YES; opaque = NO; layer =
<CALayer: 0x160b8ab0>>",#"<CKMessageEntryWaveformView:
0x160c4750; frame = (15 25.5; 251 35); alpha = 0; opaque
= NO; userInteractionEnabled = NO; layer = <CALayer:
0x160c47e0>>"]
```

其中，“UIView：0x16052920”就是“iMessage”所在的view，还记得吧？那么，紧随其后的2个UIButton就显得十分可疑了，直觉告诉笔者，“Send”就是它俩其中之一。同时我们注意到，第三个UIButton的hidden属性是YES，也就是说这个按钮是隐藏的，那么可见的“Send”肯定就是“UIButton：0x1605a8b0”了。还是用Cycript来确认一下，如下：

```
cy# [#0x1605a8b0 setHidden:YES]
```

执行之后，界面变成了图10-56的样子：

图10-56　隐藏“Send”

准确无误。按下这个UIButton后，发送一条iMessage；UIButton与其点击之后的动作一般是通过addTarget:action:forControlEvents:函数来关联的，这是UIButton的父类UIControl中的一个函数。而UIControl

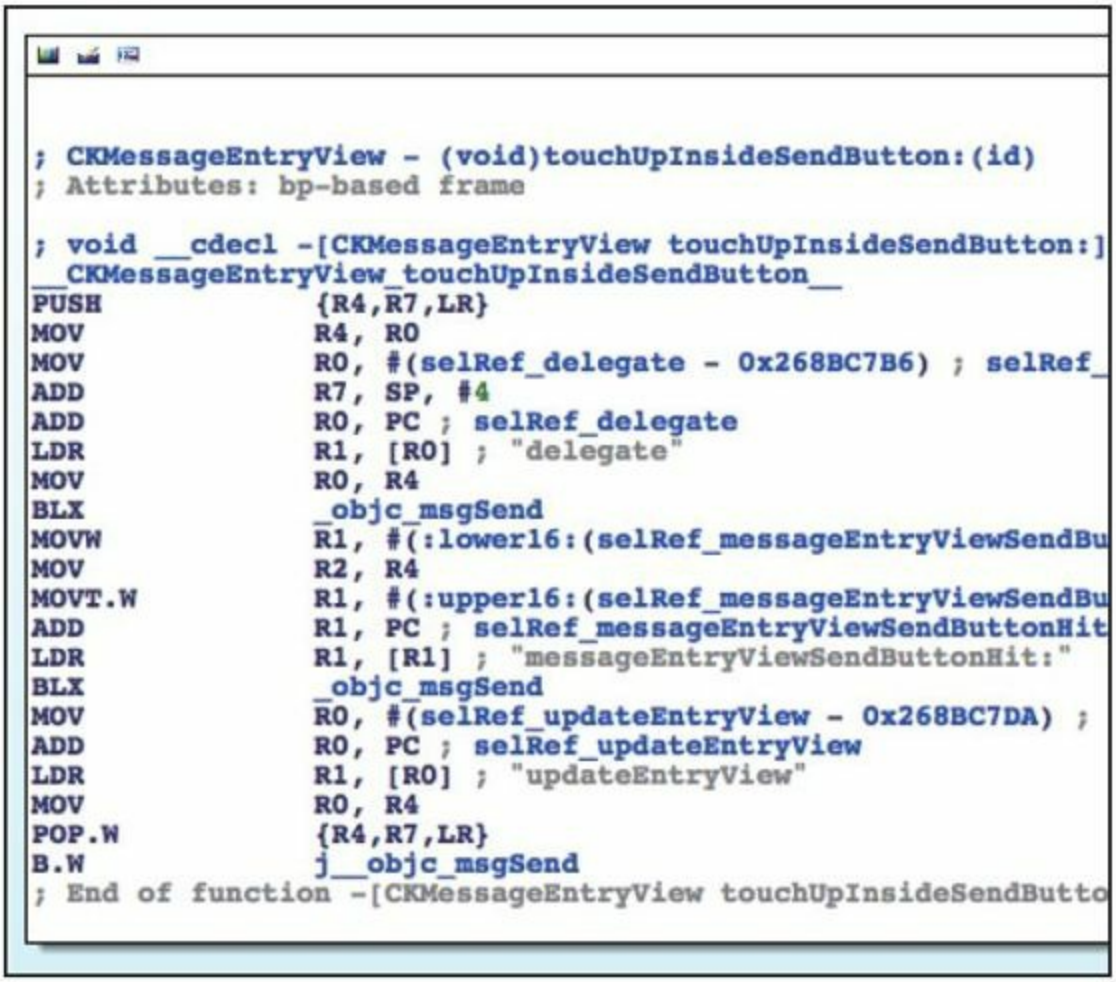
类本身就提供了一个actionsForTarget:forControlEvent:来反查UIControl对象的动作。可以利用这个函数，来看看按下“Send”之后会触发什么动作，如下：

```
cy# [#0x1605a8b0 setHidden:NO]
cy# button = #0x1605a8b0
#"<UIButton: 0x1605a8b0; frame = (266 27; 53 33); hidden
= YES; opaque = NO; layer = <CALayer: 0x16052a00>>"
cy# [button allTargets]
[NSSet setWithArray:@[#"<CKMessageEntryView: 0x160c6180;
frame = (0 0; 320 65); opaque = NO; autoresize = W; layer
= <CALayer: 0x16089920>>"]]]
cy# [button allControlEvents]
64
cy# [button actionsForTarget:#0x160c6180
forControlEvent:64]
@["touchUpInsideSendButton:"]
```

可以看到，触发的函数是[CKMessageEntryView touchUpInsideSendButton:button]。现在，转战到IDA和LLDB上，看看这个函数的内部实现。

10.3.3 在响应函数中寻找可疑的发送操作

[CKMessageEntryView touchUpInsideSendButton:button]的实现很简单，如图10-57所示。



```
; CKMessageEntryView - (void)touchUpInsideSendButton:(id)
; Attributes: bp-based frame

; void __cdecl -[CKMessageEntryView touchUpInsideSendButton:]
__CKMessageEntryView_touchUpInsideSendButton__
PUSH            {R4,R7,LR}
MOV             R4, R0
MOV             R0, #(selRef_delegate - 0x268BC7B6) ; selRef_
ADD             R7, SP, #4
ADD             R0, PC ; selRef_delegate
LDR             R1, [R0] ; "delegate"
MOV             R0, R4
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(selRef_messageEntryViewSendBu
MOV             R2, R4
MOVT.W          R1, #(:upper16:(selRef_messageEntryViewSendBu
ADD             R1, PC ; selRef_messageEntryViewSendButtonHit
LDR             R1, [R1] ; "messageEntryViewSendButtonHit:"
BLX             _objc_msgSend
MOV             R0, #(selRef_updateEntryView - 0x268BC7DA) ;
ADD             R0, PC ; selRef_updateEntryView
LDR             R1, [R0] ; "updateEntryView"
MOV             R0, R4
POP.W           {R4,R7,LR}
B.W             j__objc_msgSend
; End of function -[CKMessageEntryView touchUpInsideSendButto
```

图10-57　[CKMessageEntryView touchUpInsideSendButton:button]

先[[self delegate]messageEntryViewSendButtonHit:self]，然后[self updateEntryView]。看名字就知道后者是简单地更新视图，那发送的动作应该就包含在前者内。下面先用Cycript看看[self delegate]是什么，如下：

```
cy# [#0x160c6180 delegate]
#"<CKTranscriptController: 0x15537200>"
```

在IDA中前往[CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntry View]。这个函数的逻辑比较简单，如图10-58所示。

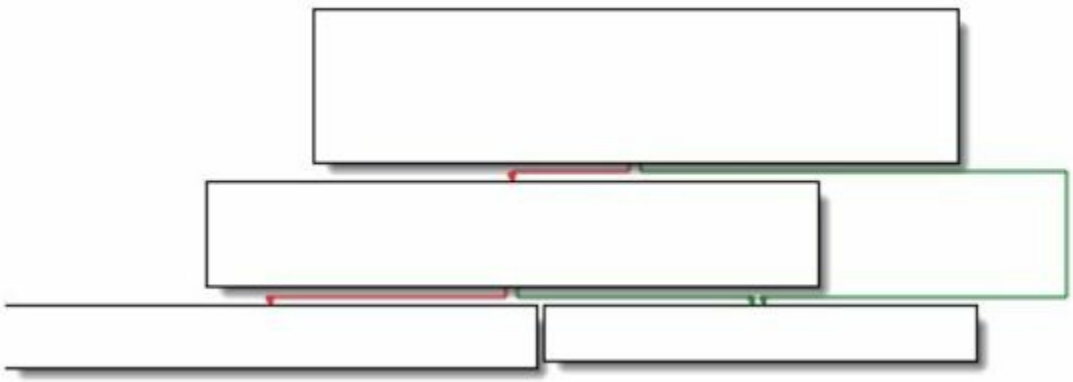


图10-58　[CKTranscriptController messageEntryViewSendButtonHit:CKMessageEntryView]

相信你用肉眼就能看出，实际的发送操作暗藏在[self sendComposition:[CKMessageEntry-View compositionWithAcceptedAutocorrection]]中。接下来在Cycript中看看[self composition-WithAcceptedAutocorrection]是什么：

```
cy# [#0x160c6180 compositionWithAcceptedAutocorrection]
#"<CKComposition: 0x160b79d0> text:'iMessage {\n}'
subject:'(null)'"
```

它是一个CKComposition对象，且明明白白地显示了要发送的标题和内容。继续看sendComposition:的内部实现，如图10-59所示。

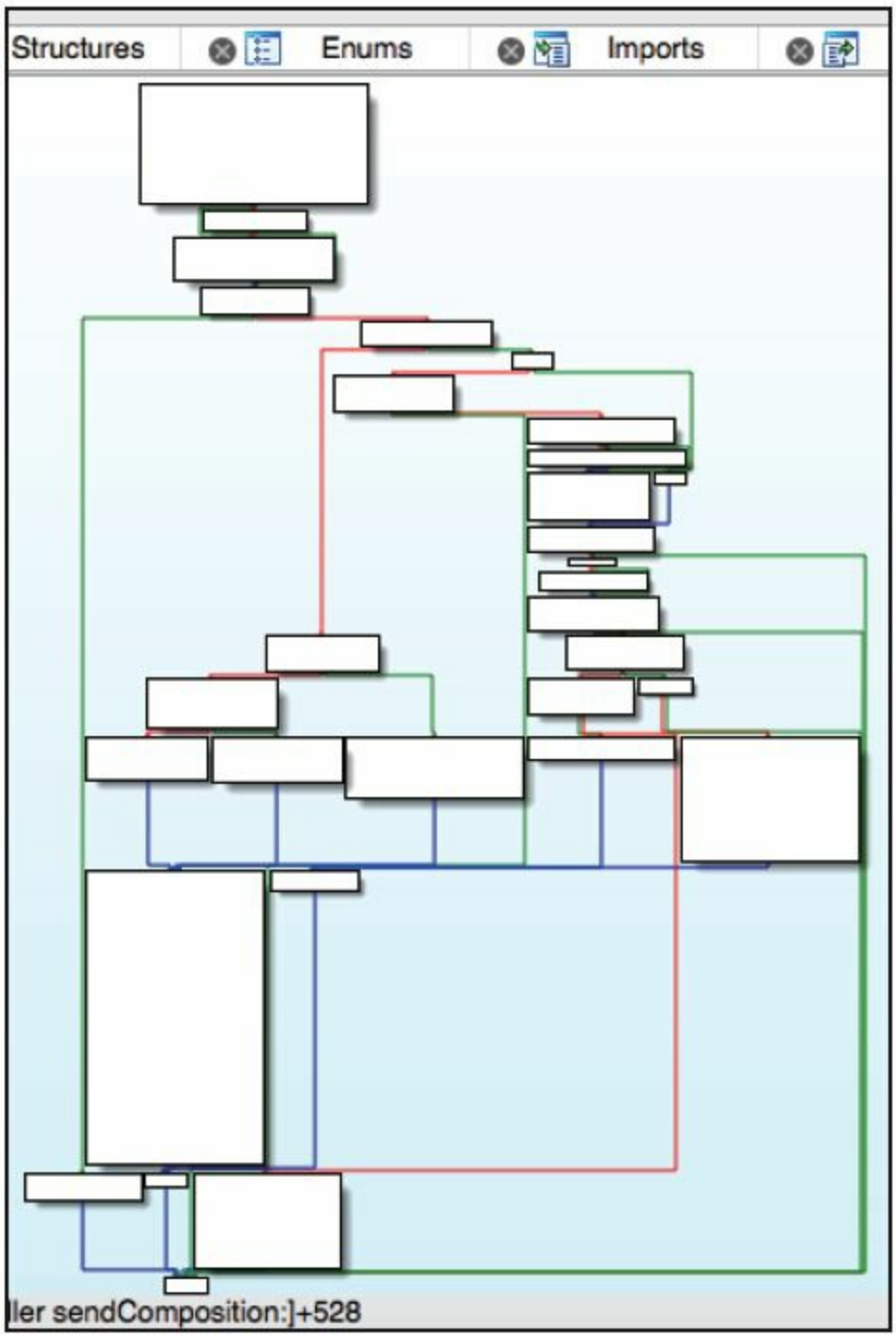


图10-59　[self sendComposition:]

其实现很复杂，有必要在到LLDB上一步一步调试之前，先大概地过一下进程流程中的几个分支，看

看其逻辑走向。先来到loc_268D427C中，如图10-60所示。

图10-60　loc_268D427C

如果“有内容”就走右边，我们发送的内容是“iMessage”，当然算是“有内容”，走右边，到达图10-61。

图10-61　分支

“下一个待调整的媒体对象”？难道是指的图片、

语音、视频这类东西？我们要发的iMessage是纯文字，应该不涉及这些东西，走右边，到达图10-62。

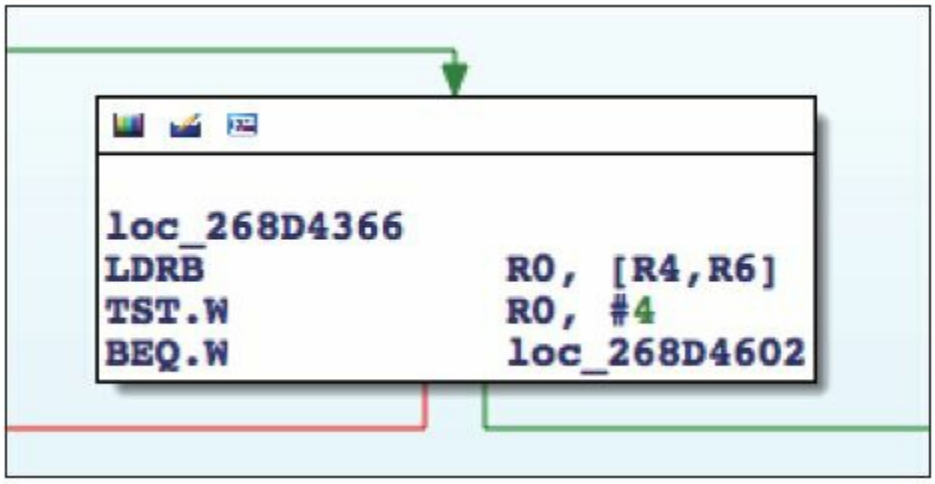


图10-62　分支

R0是个啥？回到sendComposition:的开始部分，如图10-63所示。

R0原来是self->_newRecipient，在Cycript中看看它的值是多少，如下：

```
cy# #0x15537200->_newRecipient
1
```

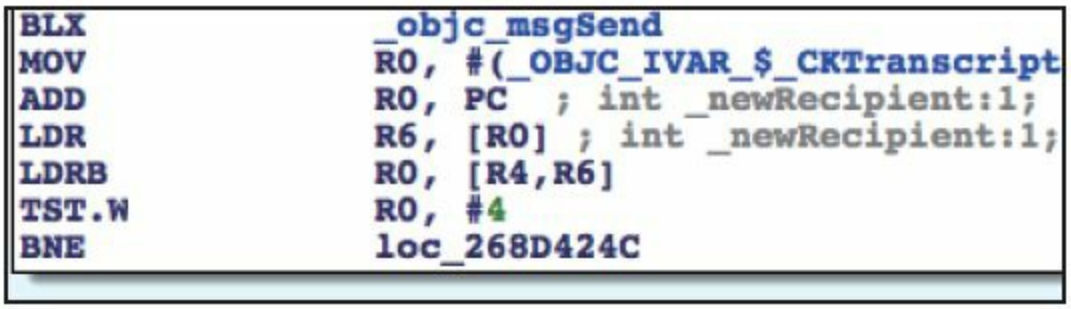


图10-63　追踪R0的值

因此“TST.W R0,#4”的结果是0，走右边到达loc_268D4604，如图10-64所示。

图10-64　loc_268D4604

“正在发送信息”？按下“Send”之后才发送信息，那这里的“正在”指的是按下“Send”之前还是之后呢？像下面这样分别测试一下好了：

```
cy# [#0x15537200 isSendingMessage]
0
```

然后按下“Send”，再测一次：

```
cy# [#0x15537200 isSendingMessage]
0
```

可见，不管是按下“Send”之前还是之后，[self isSendingMessage]的返回值都是0，走左边那条路，继续寻找下一个分支，如图10-65所示。

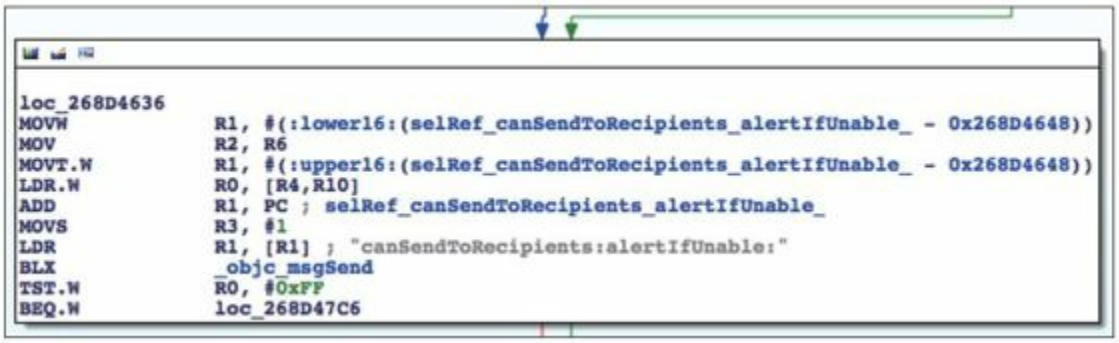


图10-65　分支

“能发送给收件人吗？”我们的目标地址是一个有效的iMessage账号，当然能了！走左边，到达图10-66。

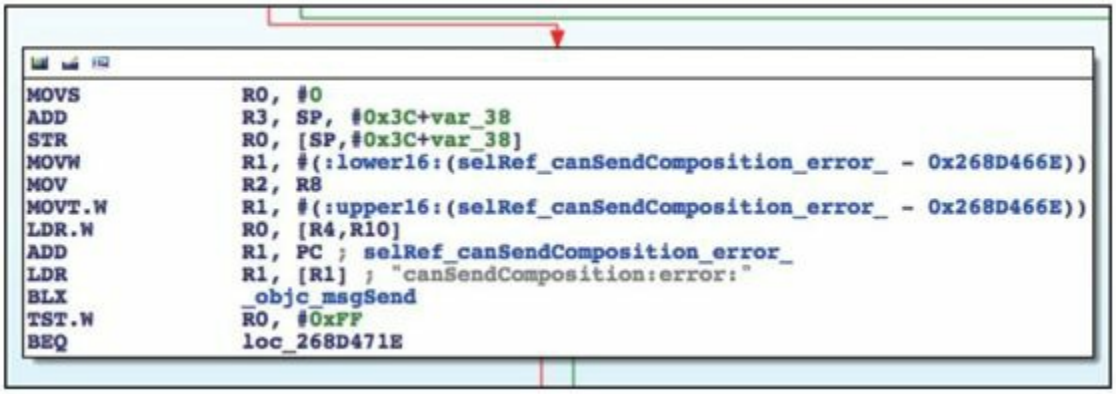


图10-66　分支

“能发送写好的内容吗？”因为刚才都已经把CKComposition的内容打印出来了，所以这里也没问题，走左边，到达图10-67。

图10-67　分支

这里又是一个分支。往上看看，就可以在图10-60中找到R5的值，可见，这里要再次判断发送的信息

是否“有内容”了。走右边，到达图10-68。

图10-68这张图的信息量略大，但仔细看看，你就会发现前面做的一系列动作都是UI层面的刷新操作，只有最后一个sendMessage:十分可疑。这个函数的参数是什么？往上回溯，可以看到其实就是[self sendComposition:]的参数，即一个CKComposition对象。继续分析[CKTranscriptController sendMessage:]的实现，如图10-69所示。

这个函数的流程看起来分支众多，但在浏览一遍（思路跟浏览sendComposition:时一样）后就会发现，分支里都只是做了一些准备工作，“_startCreatingNewMessageForSending:”才是可能发出信息的地方。去它的实现里看看，如图10-70所示。

这里的逻辑略纠结。按照上面描述过的思路，大致浏览一遍。相信你在浏览关键词的时候，也会同笔

者一样注意到“sendMessage:newComposition:”，且它一共出现了2次，即图10-71所示的2个深色方块。

下面来看看这个函数的实现，如图10-72所示。

```
MOV             R0, #(selRef_activeKeyboard - 0x268D46B4) ; selRef_activeKeyboard
MOV             R2, #(classRef_UIKeyboard - 0x268D46B6) ; classRef_UIKeyboard
ADD             R0, PC ; selRef_activeKeyboard
ADD             R2, PC ; classRef_UIKeyboard
LDR             R1, [R0] ; "activeKeyboard"
LDR             R0, [R2] ; _OBJC_CLASS_$_UIKeyboard
BLX             _objc_msgSend
MOV             R1, #(selRef_removeAutocorrectPrompt - 0x268D46C8) ; selRef_removeAutocorrectPrompt
ADD             R1, PC ; selRef_removeAutocorrectPrompt
LDR             R1, [R1] ; "removeAutocorrectPrompt"
BLX             _objc_msgSend
MOV             R0, R4
MOV             R1, R11
BLX             _objc_msgSend
MOV             R1, #(selRef_view - 0x268D46E0) ; selRef_view
ADD             R1, PC ; selRef_view
LDR             R1, [R1] ; "view"
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(selRef_setUserInteractionEnabled_ - 0x268D46F2))
MOVS            R2, #0
MOVT.W          R1, #(:upper16:(selRef_setUserInteractionEnabled_ - 0x268D46F2))
ADD             R1, PC ; selRef_setUserInteractionEnabled_
LDR             R1, [R1] ; "setUserInteractionEnabled:"
BLX             _objc_msgSend
MOV             R0, #(selRef_updateNavigationButtons - 0x268D4702) ; selRef_updateNavigationButtons
ADD             R0, PC ; selRef_updateNavigationButtons
LDR             R1, [R0] ; "updateNavigationButtons"
MOV             R0, R4
BLX             _objc_msgSend
MOVW            R0, #(:lower16:(selRef_sendMessage_ - 0x268D4716))
MOV             R2, R8
MOVT.W          R0, #(:upper16:(selRef_sendMessage_ - 0x268D4716))
ADD             R0, PC ; selRef_sendMessage_
LDR             R1, [R0] ; "sendMessage:"
MOV             R0, R4
BLX             _objc_msgSend
B               loc_268D47C6
```

图10-68　分支

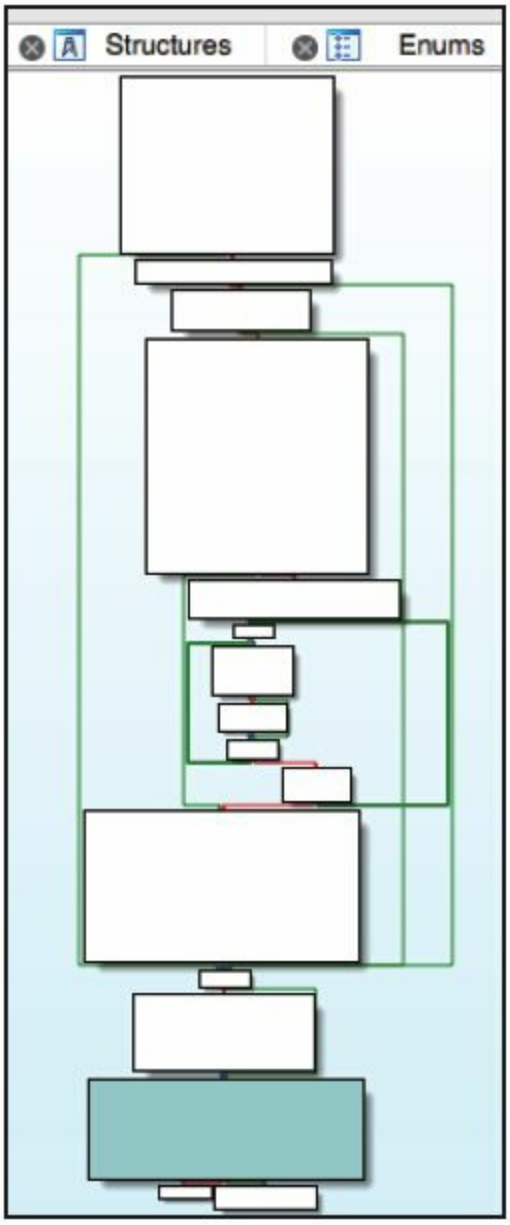


图10-69　[CKTranscriptControllersendMessage:]

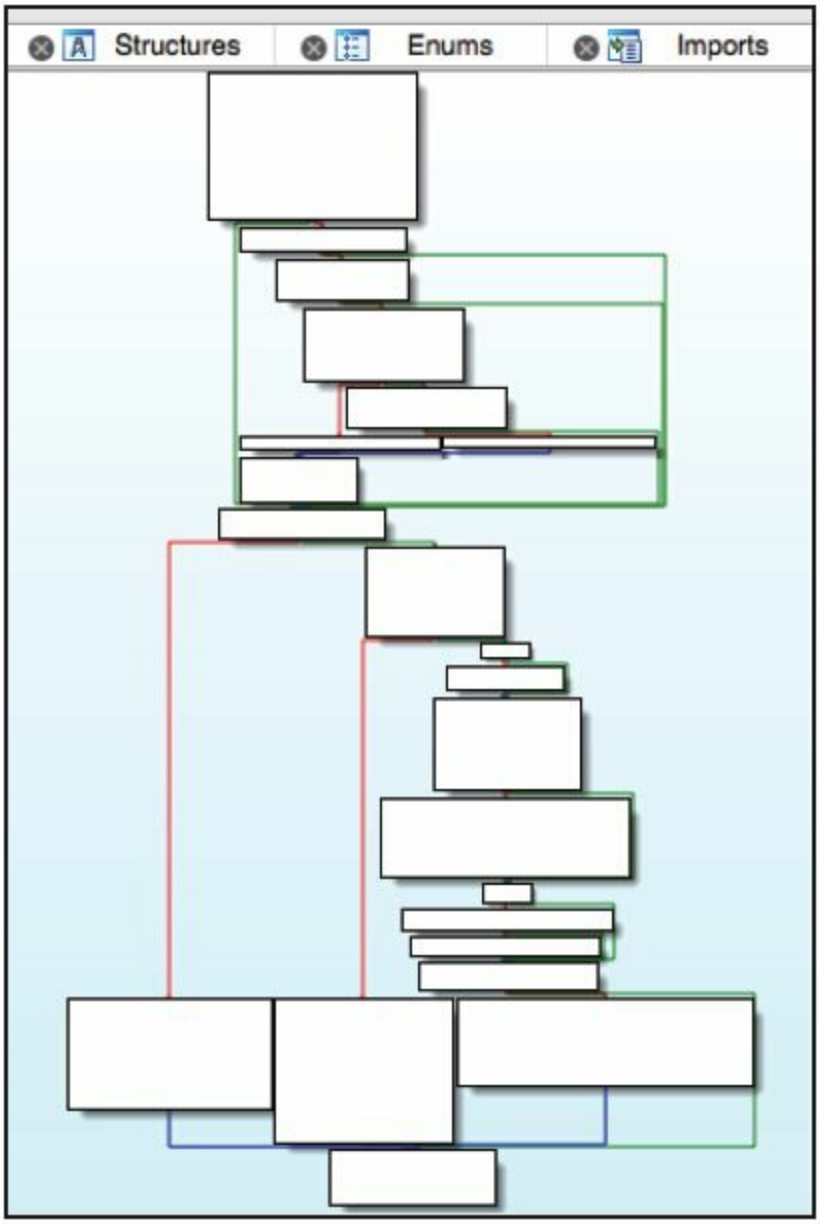


图10-70

[CKTranscriptController_startCreatingNewMessageForSen

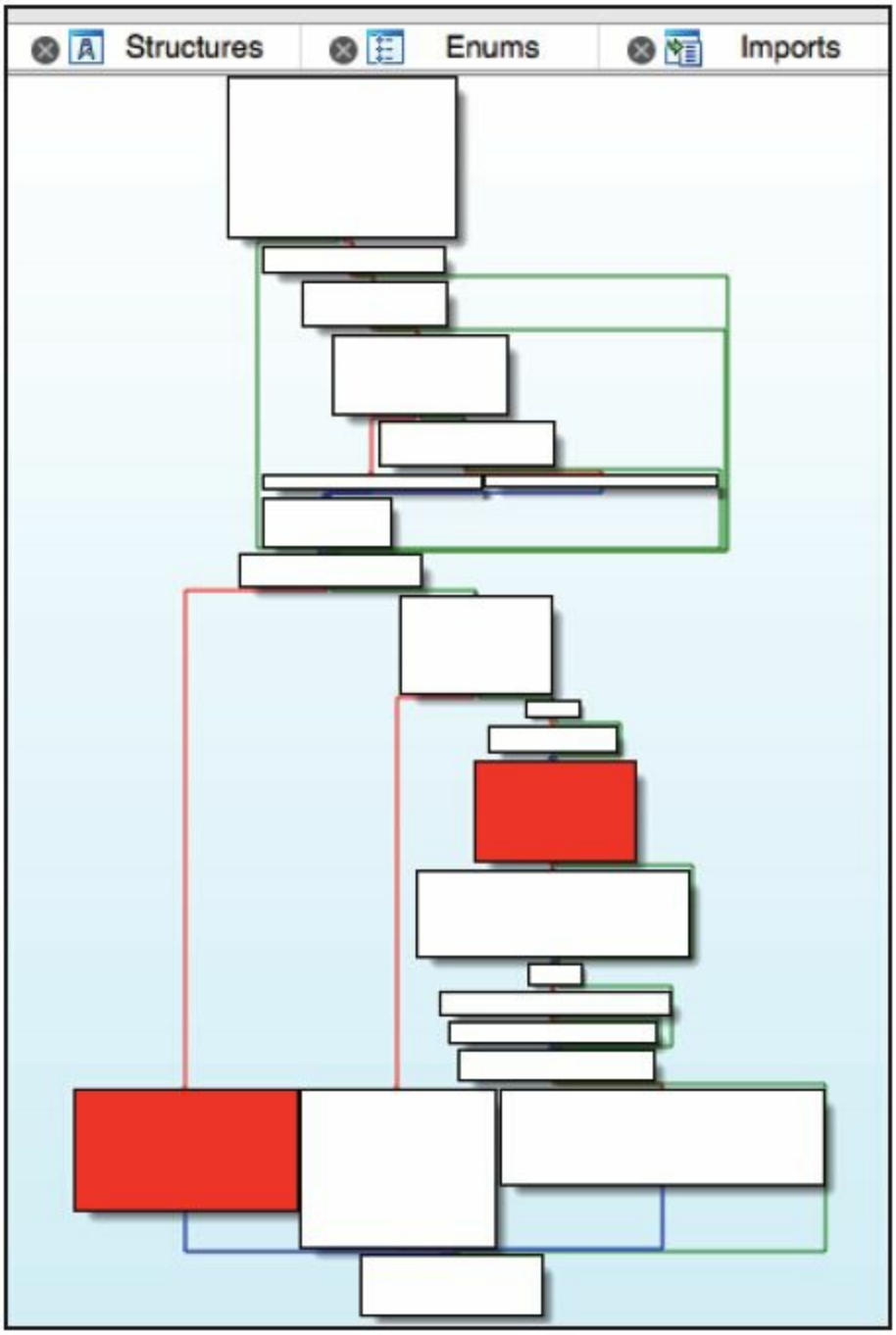


图10-71

[CKTranscriptController_startCreatingNewMessageForSen

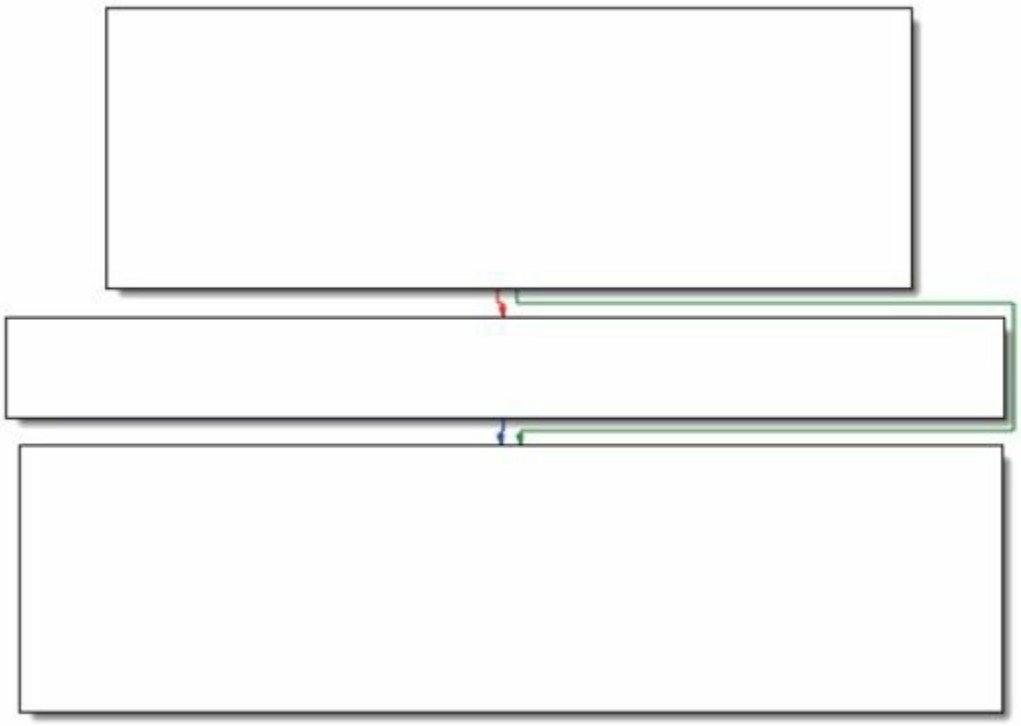


图10-72　[CKConversation sendMessage:newComposition:]

它进一步调用了“sendMessage:onService:newComposition:”，继续跟过去，如图10-73所示。

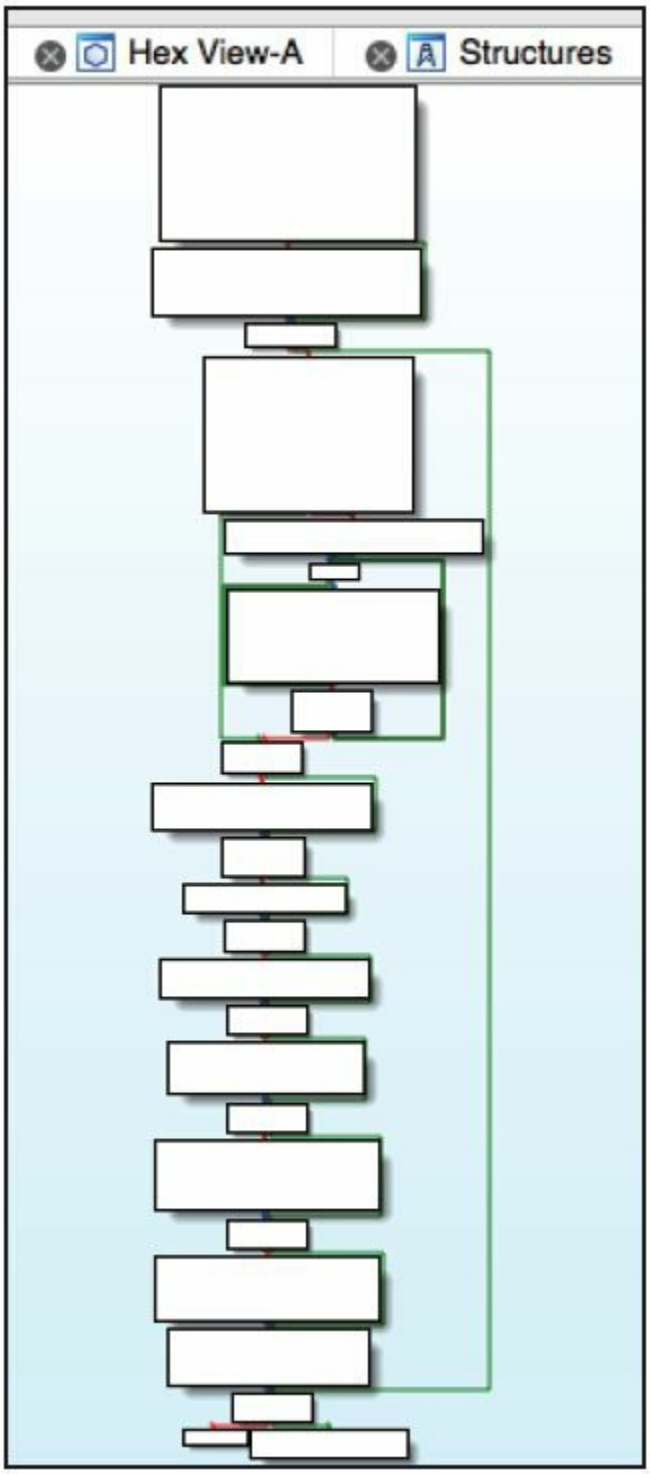


图10-73　[CKConversation sendMessage:onService:newComposition:]

流程比较简洁，大致浏览一下，就会看到诸如“Sending message with guid：%@”、“=>Sending account：%@”、“=>Recipients：[%@]”等的字眼，且它们大都是_CKLogExternal的参数——都已经开始记录这些字眼了，不恰恰说明正在发生“发送信息”这件事吗？进一步，在图10-74中又看到了可疑的字眼“sendMessage:”。

图10-74　loc_2691f836

它的调用者和参数是什么？还是直接用IDA把它们给找出来。先看调用者R0，它来自R5。R5又来自哪里呢？往上走，往上走，到loc_2691F726处，如图10-75所示。

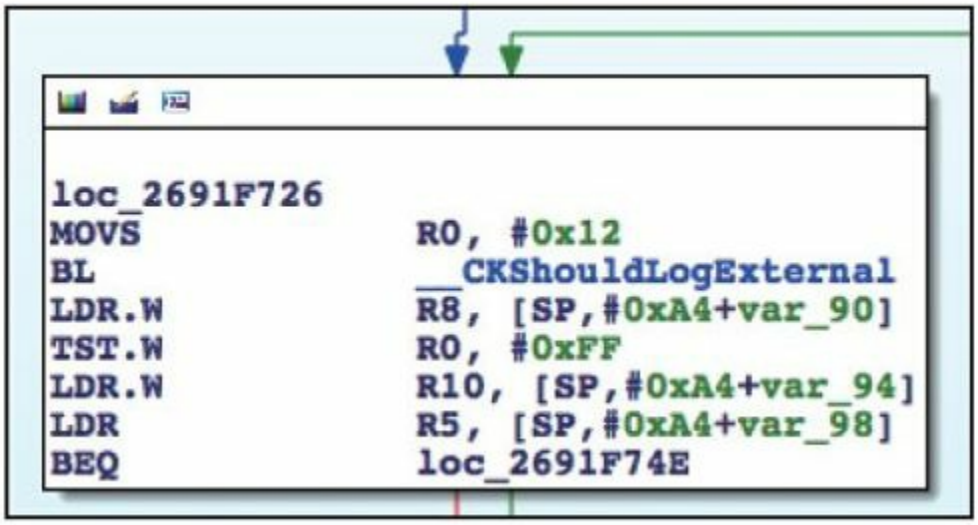


图10-75　loc_2691f726

其中，“LDR R5,[SP,#0xA4+var_98]”决定了R5的值，那[SP,#0xA4+var_98]是什么呢？还记得在10.2节中是如何处理这类问题的吗？把光标放在var_98上，然后按下“x”，查看其交叉引用，如图10-76所示。

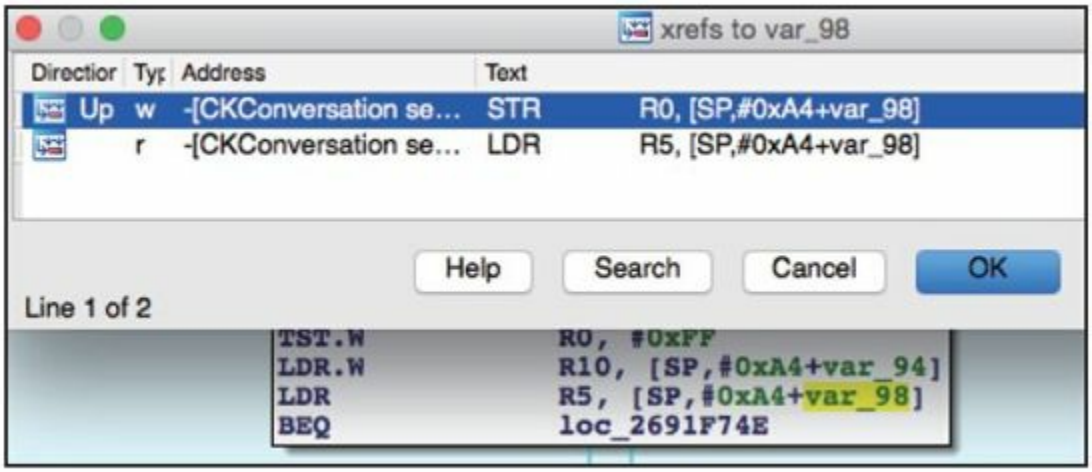


图10-76　查看交叉引用

双击第一条交叉引用，跳转至“STR R0, [SP,#0xA4+var_98]”，R0来自[R6 chat]。R6在[CKConversation sendMessage:onService:newComposition:]的开始部分第一次出现，很明显是self，所以“sendMessage:”的调用者是[self chat]。接着回到图10-74中，可看到它的参数R2来自R6。往上拉一点，可以看到R6来自loc_2691F6F4中的“LDR R6,[SP,#0xA4+var_80]”，如图10-77所示。

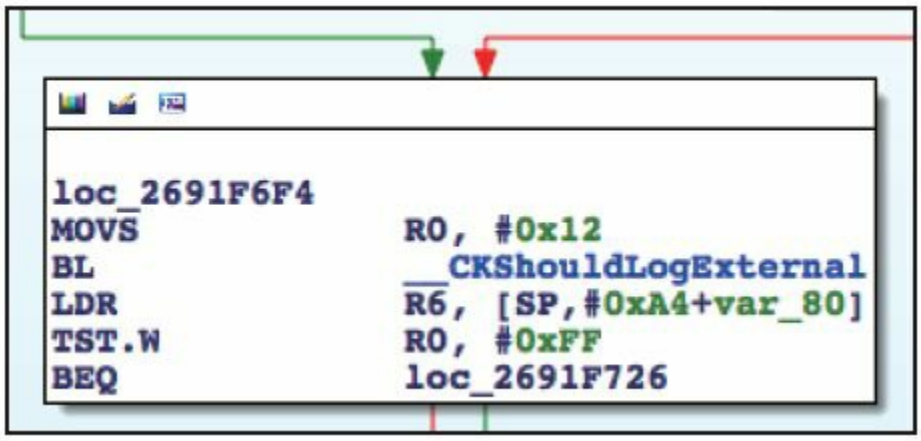


图10-77　loc_2691f6f4

接下来该怎么办？我们在1分钟前刚完成了一次类似的操作，这里就不再用文字描述了，只用几张图

片（如图10-78至图10-80所示）作为小提示，由读者自己来完成这个操作。

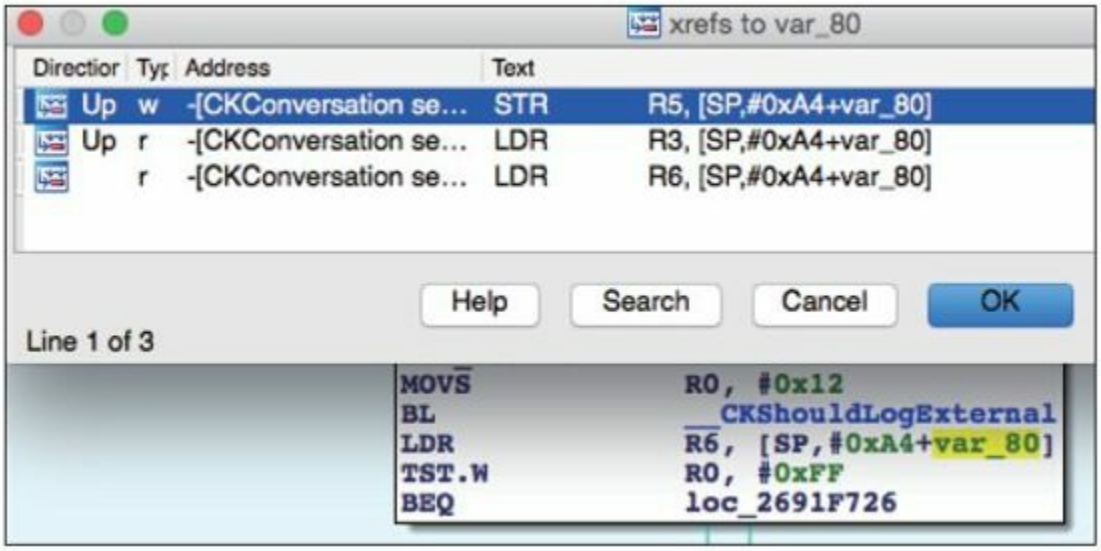


图10-78　查看交叉引用

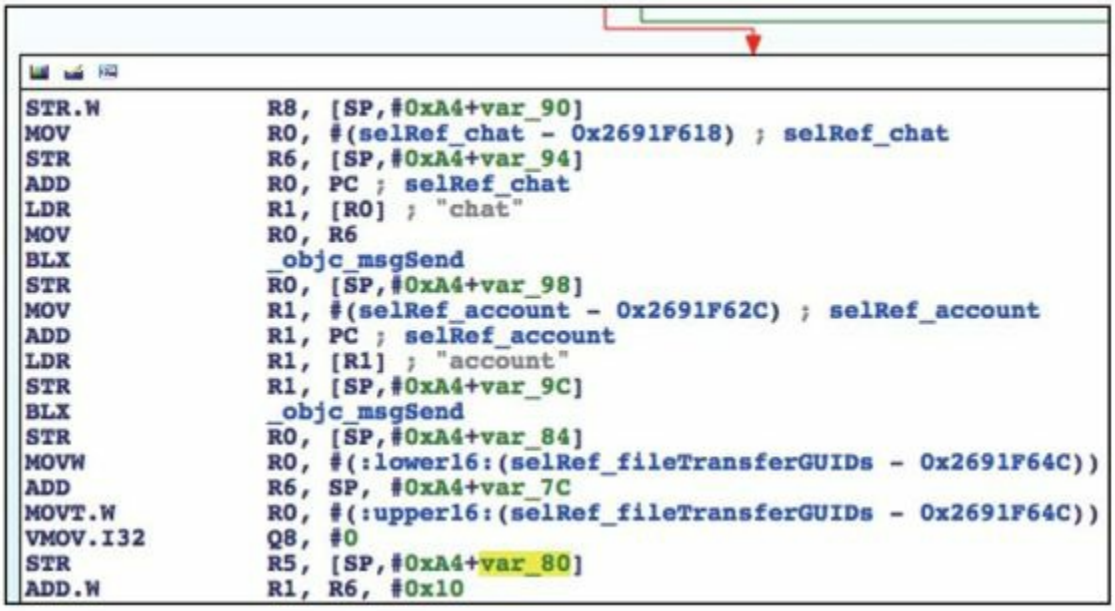


图10-79　[CKConversation setChat:]

图10-80 [CKConversation sendMessage:onService:newComposition:]

所以[[self chat]sendMessage:]的参数与[self sendMessage:onService:newComposition:]的第一个参数一脉相承。那么[self chat]是什么类型，参数又是什么类型呢？在上面的静态分析中，我们并没有在IDA中找到什么明显的线索，因此，是时候让LLDB出来热热身了。

先编写一条iMessage，然后在[CKConversation

sendMessage:onService:newComposition:]尾部“sendMessage:”下方的那个objc_msgSend上下个断点，接着按下“Send”，触发断点，如下：

```
Process 233590 stopped
* thread #1: tid = 0x39076, 0x30ad1846 ChatKit`-
[CKConversation sendMessage:onService:newComposition:] +
686, queue = 'com.apple.main-thread, stop reason =
breakpoint 1.1
    frame #0: 0x30ad1846 ChatKit`-[CKConversation
sendMessage:onService:newComposition:] + 686
ChatKit`-[CKConversation
sendMessage:onService:newComposition:] + 686:
-> 0x30ad1846:  blx    0x30b3bf44                ; symbol
stub for: MarcoShouldLogMadridLevel$shim
   0x30ad184a:  movw   r0, #49322
   0x30ad184e:  movt   r0, #2541
   0x30ad1852:  add    r0, pc
(lldb) p (char *)$r1
(char *) $0 = 0x32b26146 "sendMessage:"
(lldb) po $r0
<IMChat 0x5ef2ce0> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com   Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363   Style: -   State: 3  Participants: 1  Room
Name: (null)  Display Name: (null)  Last Addressed:
(null) Group ID: F399B0B5-800F-47A4-A66C-72C43ACC0428
Unread Count: 0  Failure Count: 0]
(lldb) po $r2
IMMessage[from=(null); msg-subject=(null); account:
(null); flags=100005; subject='<< Message Not Loggable
>>' text='<< Message Not Loggable >>' messageID: 0
GUID:'966C2CD6-3710-4D0F-BCEF-BCFEE8E60FE9'
date:'437730968.559627' date-delivered:'0.000000' date-
read:'0.000000' date-played:'0.000000' empty: NO
finished: YES sent: NO read: NO delivered: NO audio: NO
played: NO from-me: YES emote: NO dd-results: NO dd-
scanned: YES error: (null)]
(lldb) ni
```

结果不能再明显了，[IMChat sendMessage:IMMessage]就是我们要找的答案。注意，笔者在打印完所有需要的信息后执行了“ni”命令，然后听到iPhone发出了一个熟悉的“信息已发送”的提示音。这从侧面说明，实际的发送操作正是在[IMChat sendMessage:IMMessage]内完成的。因为IMChat和IMMessage的前缀均为IM，所以它们来自ChatKit以外的库，ChatKit所提供的最底层的发送信息函数到[CKConversation sendMessage:onService:newComposition:]为止。此时，可以肯定，只要能够构造我们自己的IMChat和IMMessage，就可以实现发送iMessage的功能了。那么问题来了，怎么构造这2个类的对象呢？化繁为简，在class-dump的头文件里找找看有没有线索。

要构造IMChat和IMMessage，就先看看它们的头文件里有没有什么明显的构造方法。先打开IMChat.h，看看有没有包含“init”字眼的关键词，如

下：

```
- (id)_initWithDictionaryRepresentation:(id)arg1 items:
(id)arg2 participantsHint:(id)arg3 accountHint:(id)arg4;
- (id)init;
- (id)_initWithGUID:(id)arg1 account:(id)arg2 style:
(unsigned char)arg3 roomName:(id)arg4 displayName:
(id)arg5 items:(id)arg6 participants:(id)arg7;
```

上面的代码中参数众多，如何一个个构造它们，我们一点头绪都没有。那接下来该怎么办呢？

还记得是如何找到“sendMessage:”调用者的吗？对了，使用[self chat]——self是一个CKConversation对象，看看[CKConversation chat]是怎么来的，不就知道IMChat是如何生成的了吗？在IDA里定位到[CKConversation chat]，如图10-81所示。

```
; CKConversation - (id)chat

; id __cdecl -[CKConversation chat](struct CKConversation *self, SEL)
__CKConversation_chat_
MOV             R1, #(_OBJC_IVAR_$_CKConversation._chat - 0x26920778) ; IMChat *_chat;
ADD             R1, PC  ; IMChat *_chat;
LDR             R1, [R1] ; IMChat *_chat;
LDR             R0, [R0,R1]
BX              LR
; End of function -[CKConversation chat]
```

图10-81　[CKConversation chat]

[CKConversation chat]就是实例变量_chat的值，这个场景似曾相识啊——还记得大明湖畔的夏雨荷，哦不，是图10-22里的_composeSendingService吗？此处的想法与操作与那里完全一致，LLDB又要在逆向推导中派上大用场了！删掉已发送的iMessage对话（即删掉这个CKConversation），新建一条iMessage（新建一条CKConversation），然后在[CKConversation setChat:]上下一个断点，点击“Send”，触发断点，如下：

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30ad277c ChatKit`-
[CKConversation setChat:], queue = 'com.apple.main-
thread, stop reason = breakpoint 13.1
    frame #0: 0x30ad277c ChatKit`-[CKConversation
setChat:]
ChatKit`-[CKConversation setChat:]:
-> 0x30ad277c:  movw   r3, #55168
   0x30ad2780:  movt   r3, #2541
   0x30ad2784:  add    r3, pc
   0x30ad2786:  ldr    r3, [r3]
(lldb) po $r2
<IMChat 0x1594f7e0> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com   Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363   Style: -   State: 0  Participants: 1  Room
Name: (null)  Display Name: (null)  Last Addressed:
(null)Group ID: (null)  Unread Count: 0  Failure Count:
0]
(lldb) p/x $lr
(unsigned int) $20 = 0x30acf625
```

LR偏移前的值是0x30acf625–0xa1b2000=0x2691d625，这个地址位于[CKConversation initWithChat:]中。不过，[CKConversation initWithChat:]的调用者又是谁呢？如法炮制（注意，每次下断点前都要删掉已发送的iMessage对话，再新建一条iMessage，下同）：

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30acf5ec ChatKit`-
[CKConversation initWithChat:], queue = 'com.apple.main-
thread, stop reason = breakpoint 14.1
    frame #0: 0x30acf5ec ChatKit`-[CKConversation
initWithChat:]
ChatKit`-[CKConversation initWithChat:]:
-> 0x30acf5ec:  push   {r4, r5, r6, r7, lr}
   0x30acf5ee:  add    r7, sp, #12
   0x30acf5f0:  push.w {r8, r10, r11}
   0x30acf5f4:  sub    sp, #8
(lldb) po $r2
<IMChat 0x1470a520> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363 Style: - State: 0 Participants: 1  Room
Name: (null)  Display Name: (null)  Last Addressed:
(null) Group ID: (null) Unread Count: 0 Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $22 = 0x30a8d131
```

LR偏移前的值是0x30a8d131–0xa1b2000=0x268db131，这个地址位于

[CKConversationList_beginTrackingConversationWithCha

中。继续：

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30a8d09c ChatKit`-
[CKConversationList _beginTrackingConversationWithChat:],
queue = 'com.apple.main-thread, stop reason = breakpoint
15.1
    frame #0: 0x30a8d09c ChatKit`-[CKConversationList
_beginTrackingConversationWithChat:]
ChatKit`-[CKConversationList
_beginTrackingConversationWithChat:]:
-> 0x30a8d09c:  push   {r4, r5, r6, r7, lr}
   0x30a8d09e:  mov    r5, r0
   0x30a8d0a0:  movs   r0, #25
   0x30a8d0a2:  add    r7, sp, #12
(lldb) po $r2
<IMChat 0x15a326a0> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com   Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363   Style: -   State: 0  Participants: 1  Room
Name: (null)  Display Name: (null)  Last Addressed:
(null) Group ID: (null)  Unread Count: 0  Failure Count:
0]
(lldb) p/x $lr
(unsigned int) $24 = 0x30a8d4f1
```

LR偏移前的值是0x30a8d4f1–

0xa1b2000=0x268db131，这个地址位于

[CKConversationList_handleRegistryDidRegisterChatNoti

中，且这里的IMChat对象来自于[notification object]。

因为这里的IMChat对象是通过notification传播的，所

以下一个目标不是找到[CKConversationList_handleRegistryDidRegisterChatNoti的调用者，而是找到这条notification的发布者，它才是“罪魁祸首”。在这个函数的第一条指令上下一个断点，看看这条notification的结构，如下：

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x30a8d4ac ChatKit`-
[CKConversationList
_handleRegistryDidRegisterChatNotification:], queue =
'com.apple.main-thread, stop reason = breakpoint 16.1
    frame #0: 0x30a8d4ac ChatKit`-[CKConversationList
_handleRegistryDidRegisterChatNotification:]
ChatKit`-[CKConversationList
_handleRegistryDidRegisterChatNotification:]:
-> 0x30a8d4ac:  push   {r4, r5, r6, r7, lr}
   0x30a8d4ae:  add    r7, sp, #12
   0x30a8d4b0:  push.w {r8, r10, r11}
   0x30a8d4b4:  sub.w  r4, sp, #64
(lldb) po $r2
NSConcreteNotification 0x15934340 {name =
__kIMChatRegistryDidRegisterChatNotification; object =
<IMChat 0x147c39f0> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com   Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363   Style: -   State: 0  Participants: 1  Room
Name: (null)  Display Name: (null)  Last Addressed:
(null) Group ID: (null)  Unread Count: 0  Failure Count:
0]}
```

这条notification的name是“__kIMChatRegistryDidRegisterChatNotification”，

object是一个IMChat对象，因此这个IMChat对象一定是在发布（post）这条notification之前就已经生成了。要找出这条notification的发布者，最好的方法，就是执行grep命令搜索一遍系统文件，看看“__kIMChatRegistryDidRegisterChatNotification”的关键字都会在哪些文件里出现，如下：

```
FunMaker-5:~ root# grep -r
_handleRegistryDidRegisterChatNotification: /System/
Binary file
/System/Library/Caches/com.apple.dyld/dyld_shared_cache_a
 matches
grep: /System/Library/Caches/com.apple.dyld/enable-
dylibs-to-override-cache: No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep:
/System/Library/Frameworks/CoreGraphics.framework/Resourc
 No such file or directory
grep: /System/Library/Frameworks/System.framework/System:
No such file or directory
```

因为它在cache里，所以下面grep一遍decache出的文件，如下：

```
snakeninnys-MacBook:~ snakeninny$ grep -r
```

```
__kIMChatRegistryDidRegisterChatNotification
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5/
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//dyl
 matches
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//Sys
 Too many levels of symbolic links
grep:
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//Sys
 Too many levels of symbolic links
Binary file
/Users/snakeninny/Code/iOSSystemBinaries/8.1_iPhone5//Sys
 matches
```

其实到这里，相信你也能猜到，IMCore与ChatKit都负责与信息相关的操作，但IMCore比ChatKit更底层，ChatKit接到的命令都交给IMCore完成，IMCore完成的结果再交给ChatKit展示给用户——MobileSMS是餐馆，ChatKit是服务员，IMCore是厨师，这么比喻，就好理解多了吧。

接下来，在IDA中打开IMCore，全文搜索“__kIMChatRegistryDidRegisterChatNotifi-cation”，如图10-82所示。

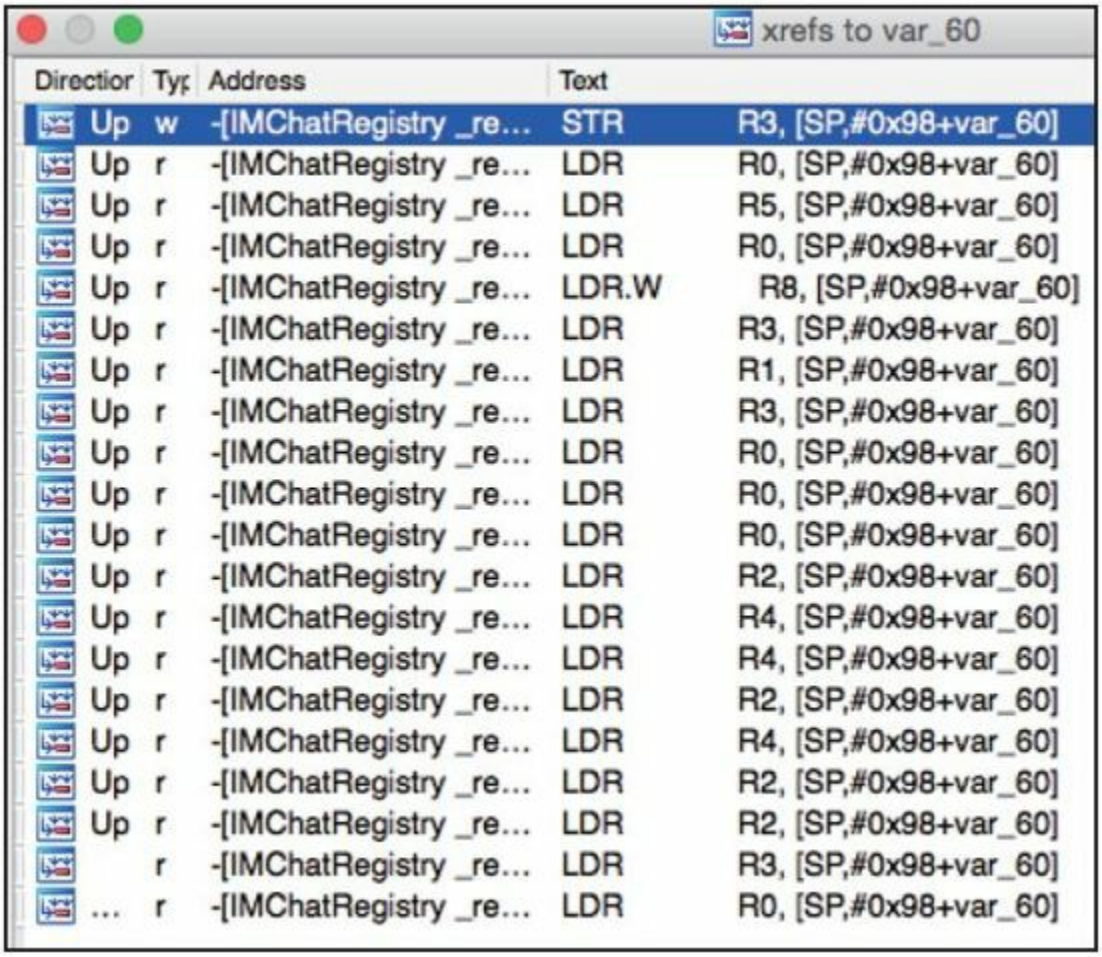
图10-82　在IDA里搜索“__kIMChatRegistryDidRegisterChatNotification”

很好，直接双击第一条搜索结果，看看它的上下文，如图10-83所示。

看到“PostNotification”字眼，我们就知道了，ChatKit收到的那条notification正是来自于此。IMChat对象是第二个参数，即R3，而R3来自[SP,#0x98+var_60]。还记得怎么操作吗？还是以提示图（如图10-84与图10-85所示）代替文字，请读者自己来摸索着操作吧。

```
loc_2908423E
MOV             R0, #(selRef_defaultCenter - 0x29084252) ; selRef_defaultCenter
MOV             R2, #(classRef_NSNotificationCenter - 0x29084254) ; classRef_NSNot
ADD             R0, PC ; selRef_defaultCenter
ADD             R2, PC ; classRef_NSNotificationCenter
LDR             R1, [R0] ; "defaultCenter"
LDR             R0, [R2] ; _OBJC_CLASS_$_NSNotificationCenter
MOVS            R2, #1
STR             R2, [SP,#0x98+var_48]
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(selRef___mainThreadPostNotificationName_object_use
MOVS            R5, #2
MOVT.W          R1, #(:upper16:(selRef___mainThreadPostNotificationName_object_use
LDR             R3, [SP,#0x98+var_60]
MOV             R2, #(cfstr___kimchatregis - 0x2908427C) ; "__kIMChatRegistryDidRe
ADD             R1, PC ; selRef___mainThreadPostNotificationName_object_userInfo_
LDR             R6, [SP,#0x98+var_50]
LDR             R1, [R1] ; "__mainThreadPostNotificationName:object"...
ADD             R2, PC  ; "__kIMChatRegistryDidRegisterChatNotification"
STR             R5, [SP,#0x98+var_48]
STR             R6, [SP,#0x98+var_98]
BLX             _objc_msgSend
```

图10-83　查看搜索结果



xrefs to var_60

Direction	Type	Address	Text	
Up	w	-[IMChatRegistry _re...	STR	R3, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R5, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR.W	R8, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R3, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R1, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R3, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R0, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R4, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R4, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R4, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R2, [SP,#0x98+var_60]
Up	r	-[IMChatRegistry _re...	LDR	R2, [SP,#0x98+var_60]
	r	-[IMChatRegistry _re...	LDR	R3, [SP,#0x98+var_60]
...	r	-[IMChatRegistry _re...	LDR	R0, [SP,#0x98+var_60]

图10-84　查看交叉引用

通过上面的分析可知，IMChat对象来自于[IMChatRegistry_registerChatDictionary:forChat:isIncomi的第二个参数。这个函数的调用者如下：

```
Process 248623 stopped
 * thread #1: tid = 0x3cb2f, 0x33235944
IMCore`___lldb_unnamed_function2048$$IMCore, queue =
'com.apple.main-thread, stop reason = breakpoint 17.1
    frame #0: 0x33235944
IMCore`___lldb_unnamed_function2048$$IMCore
IMCore`___lldb_unnamed_function2048$$IMCore:
-> 0x33235944:  push   {r4, r5, r6, r7, lr}
   0x33235946:  add    r7, sp, #12
   0x33235948:  push.w {r8, r10, r11}
   0x3323594c:  sub.w  r4, sp, #64
(lldb) po $r3
<IMChat 0x147c7f30> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363 Style: - State: 0  Participants: 1 Room
Name: (null) Display Name: (null) Last Addressed: (null)
Group ID: (null) Unread Count: 0 Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $27 = 0x3323646f
```

```
; void __cdecl -[IMChatRegistry _registerChatDictionary:forChat:isIncoming:newGUID:]
__IMChatRegistry__registerChatDictionary_forChat_isIncoming_newGUID__

var_98= -0x98
var_94= -0x94
var_90= -0x90
var_70= -0x70
var_6C= -0x6C
var_68= -0x68
var_64= -0x64
var_60= -0x60
var_5C= -0x5C
var_58= -0x58
var_54= -0x54
var_50= -0x50
var_4C= -0x4C
var_48= -0x48
var_34= -0x34
var_30= -0x30
var_2C= -0x2C
var_28= -0x28
var_24= -0x24
var_18= -0x18
arg_0=  8
arg_4=  0xC

PUSH            {R4-R7,LR}
ADD             R7, SP, #0xC
PUSH.W          {R8,R10,R11}
SUB.W           R4, SP, #0x40
BIC.W           R4, R4, #0xF
MOV             SP, R4
VST1.64         {D8-D11}, [R4@128]!
VST1.64         {D12-D15}, [R4@128]
SUB             SP, SP, #0x80
MOV             R5, R0
MOV             R0, #(selRef__shouldRegisterChat - 0x29083972) ; selRef__shouldRegis
STR             R3, [SP,#0x98+var_60]
```

图10-85

[IMChatRegistry_registerChatDictionary:forChat:isIncomin

LR偏移前的值是0x3323646f–0xa1b2000=0x2908446F，这个地址位于[IMChatRegistry_registerChat:isIncoming:guid:]中。继续：

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x3323644c
IMCore`___lldb_unnamed_function2049$$IMCore, queue =
'com.apple.main-thread, stop reason = breakpoint 20.1
    frame #0: 0x3323644c
```

```
IMCore`___lldb_unnamed_function2049$$IMCore
IMCore`___lldb_unnamed_function2049$$IMCore:
-> 0x3323644c:  push   {r4, r5, r7, lr}
   0x3323644e:  add    r7, sp, #8
   0x33236450:  sub    sp, #8
   0x33236452:  movw   r1, #9840
(lldb) po $r2
<IMChat 0x15972f20> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363 Style: - State: 0  Participants: 1  Room
Name: (null)  Display Name: (null) Last Addressed: (null)
Group ID: (null) Unread Count: 0 Failure Count: 0]
(lldb) p/x $lr
(unsigned int) $30 = 0x33237173
```

LR偏移前的值是0x33237173–0xa1b2000=0x29085173，这个地址位于[IMChatRegistry chatForIMHandle:]中，且[IMChatRegistry_registerChat:isIncoming:guid:]的第一个参数，即IMChat对象来自于R5，在[IMChatRegistry chatForIMHandle:]的最后阶段，R5是以返回值的形象出现的。也就是说，[IMChatRegistry chatForIMHandle:]这个函数返回了一个IMChat！且从IMChatRegistry的名字来看，就知道这个类负责注册登记IMChat，一个IMChat对象由此而来，合情合理。但是，解决了1个老问题，带来了2个新问题——

IMChatRegistry和chatForIMHandle:的参数从哪里来？饭要一口一口地吃，逆向要一步一步地来。打开IMChatRegistry.h（如图10-86所示），先从IMChatRegistry下手。

```
@interface IMChatRegistry : NSObject <NSFastEnumeration>
{
    NSMutableArray *_allChats;
    NSMutableDictionary *_chatGUIDToCurrentThreadMap;
    NSMutableDictionary *_chatGUIDToInfoMap;
    NSMutableDictionary *_chatGUIDToChatMap;
    NSMutableDictionary *_threadNameToChatMap;
    NSMutableDictionary *_chatGUIDToiMessageSentOrReceivedMap;
    NSMutableArray *_allChatsInThreadNameMap;
    NSMutableArray *_pendingQueries;
    NSMutableArray *_waitingForQueries;
    NSString *_historyModificationStamp;
    IMTimer *_markAsReadTimer;
    double _timerStartTimeInterval;
    BOOL _firstLoad;
    BOOL _loading;
    BOOL _daemonHadTerminated;
    BOOL _wantsHistoryReload;
    BOOL _postMessageSentNotifications;
    unsigned int _defaultNumberOfMessagesToLoad;
    unsigned int _daemonUnreadCount;
    long long _daemonLastFailedMessageID;
    NSUserActivity *_userActivity;
}

+ (Class)messageClass;
+ (void)setMessageClass:(Class)arg1;
+ (Class)chatClass;
+ (void)setChatClass:(Class)arg1;
+ (Class)chatRegistryClass;
+ (void)setChatRegistryClass:(Class)arg1;
+ (id)sharedInstance;
```

图10-86　IMChatRegistry.h

其中，第44行的sharedInstance，说明IMChatRegistry是一个单例，通过调用[IMChatRegistry sharedInstance]就可以获取到它的实例。So easy！

那chatForIMHandle:的参数从哪里来？自然是从它的调用者那里来，继续用LLDB追踪吧，如下：

```
Process 248623 stopped
* thread #1: tid = 0x3cb2f, 0x33236d8c
IMCore`___lldb_unnamed_function2054$$IMCore, queue =
'com.apple.main-thread, stop reason = breakpoint 21.1
    frame #0: 0x33236d8c
IMCore`___lldb_unnamed_function2054$$IMCore
IMCore`___lldb_unnamed_function2054$$IMCore:
-> 0x33236d8c:  push   {r4, r5, r6, r7, lr}
   0x33236d8e:  add    r7, sp, #12
   0x33236d90:  str    r11, [sp, #-4]!
   0x33236d94:  sub    sp, #20
(lldb) po $r2
[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No
AB Match>) (Account: P:+86PhoneNumber]
(lldb) p/x $lr
(unsigned int) $32 = 0x30a8dca5
```

LR偏移前的值是0x30a8dca5–0xa1b2000=0x268dbca5，这个地址已经不再位于IMCore的地址范围内了。刚才也说了，现在正不断地

在IMCore和ChatKit间徘徊，正好ChatKit的ASLR偏移也是0xa1b2000，那就去ChatKit看看0x268dbca5在不在它里面，如图10-87所示。

图10-87　[CKConversationList conversationForHandles:displayName:joinedChatsOnly:cre

0x268dbca5位于[CKConversationList conversationForHandles:displayName:joinedChatsOnly:cr内部，chatForIMHandle:的参数也是来自于[CKConversationList conversation ForHandles:displayName:joinedChatsOnly:create:]的第一个参数。继续回溯，如下：

```
Process 292950 stopped
* thread #1: tid = 0x47856, 0x30a8dc60 ChatKit`-
[CKConversationList
conversationForHandles:displayName:joinedChatsOnly:create
 queue = 'com.apple.main-thread, stop reason = breakpoint
1.1
    frame #0: 0x30a8dc60 ChatKit`-[CKConversationList
conversationForHandles:displayName:joinedChatsOnly:create
```

```
ChatKit`-[CKConversationList
conversationForHandles:displayName:joinedChatsOnly:create

-> 0x30a8dc60:  push   {r4, r5, r6, r7, lr}
   0x30a8dc62:  add    r7, sp, #12
   0x30a8dc64:  sub    sp, #8
   0x30a8dc66:  mov    r6, r0
(lldb) po $r2
<__NSArrayM 0x178d2290>(
[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person: <No
AB Match>) (Account: P:+86PhoneNumber]
)
(lldb) p/x $lr
(unsigned int) $1 = 0x30a84efd
```

LR偏移前的值是0x30a84efd–0xa1b2000=0x268d2efd，这个地址位于[CKTranscript-Controller sendMessage:]中！你！敢！信！吗！绕了一大圈，又回到了原点，让人不禁感叹，缘，妙不可言。擦干喜悦的泪珠，来看看，这个IMHandle数组到底是怎么来的，如图10-88所示。

```
MOVS            R2, #1
MOVT.W          R1, #(:upper16:(selRef_conversationForHandles_displa
LDR             R6, [SP,#0xA8+var_80]
MOVS            R3, #0
ADD             R1, PC ; selRef_conversationForHandles_displayName_j
STR             R2, [SP,#0xA8+var_A8]
LDR             R1, [R1] ; "conversationForHandles:displayName:join"
STR             R2, [SP,#0xA8+var_A4]
MOV             R2, R6
BLX             _objc_msgSend
```

图10-88　IMHandle数组的来源

R2来自R6，R6来自[SP,#0xA8+var_80]。熟悉的套路又回来了，下面还是只给出提示图（如图10-89和图10-90所示），请读者自行分析。

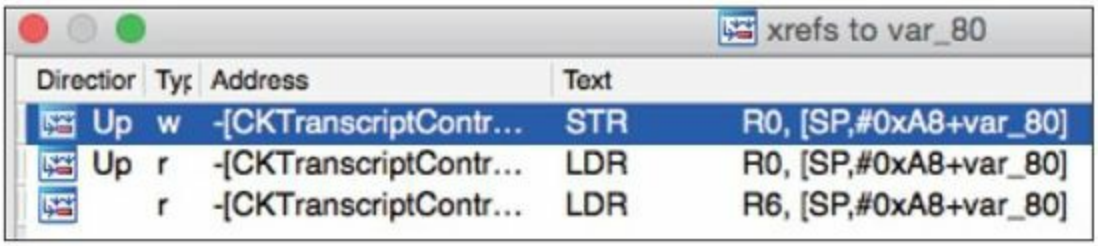


图10-89　查看交叉引用

```
LDR     R1, [R0] ; "alloc"
LDR     R0, [R2] ; _OBJC_CLASS_$_NSMutableArray
BLX     _objc_msgSend
MOV     R6, R0
MOV     R0, #(selRef_count - 0x268D2DA6) ; selRef_count
ADD     R0, PC ; selRef_count
LDR     R1, [R0] ; "count"
MOV     R0, R8
BLX     _objc_msgSend
MOV     R2, R0
MOV     R0, #(selRef_initWithCapacity_ - 0x268D2DBA) ;
ADD     R0, PC ; selRef_initWithCapacity_
LDR     R1, [R0] ; "initWithCapacity:"
MOV     R0, R6
BLX     _objc_msgSend
STR     R0, [SP,#0xA8+var_80]
```

图10-90　[CKTranscriptController sendMessage:]

你可能也会发现，这里的情况跟前几次不一样了——“STR R0,[SP,#0xA8+var_80]”貌似只是往[SP,#0xA8+var_80]里存了一个初始化过的NSMutableArray而已啊！说好的IMHandle呢？嘿嘿，

既然是一个NSMutableArray，那么就可能调用addObject:，往里面加东西，所以图10-89里中间的那一个“LDR R0,[SP,#0xA8+var_80]”……双击跳转过去看看，如图10-91所示。

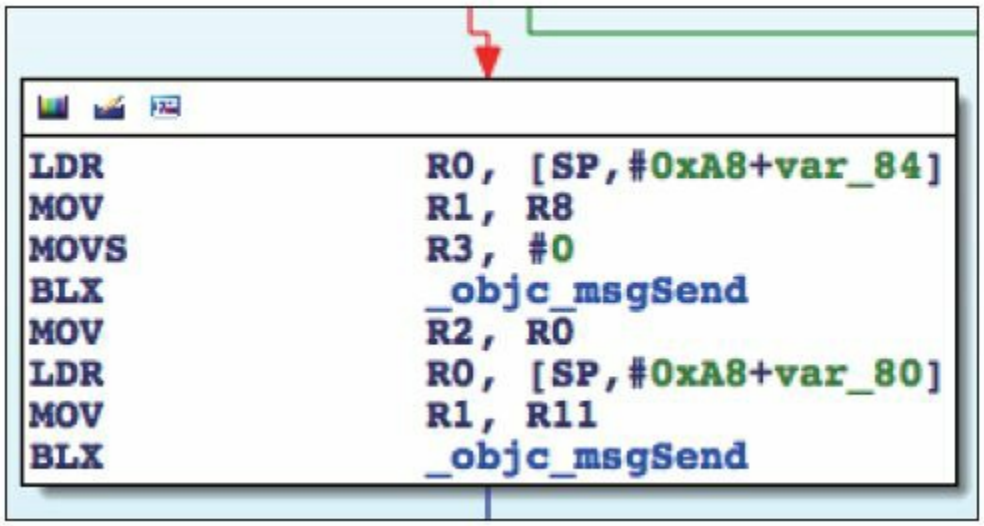


图10-91　寻找IMHandle

果不其然，它是一个addObject:，而且通过简单观察上下文就会发现，addObject:的参数来自于imHandleWithID:alreadyCanonical:，并且从这个函数的名字就可以看出来，它返回了一个IMHandle。看来，我们的IMHandle有着落了！在图10-91所示的第一个objc_msgSend上下一个断点，看看

imHandleWithID:alreadyCanonical:的调用者和参数，如下：

```
Process 343388 stopped
* thread #1: tid = 0x53d5c, 0x30a84e98 ChatKit`-
[CKTranscriptController sendMessage:] + 516, queue =
'com.apple.main-thread, stop reason = breakpoint 1.1
    frame #0: 0x30a84e98 ChatKit`-[CKTranscriptController
sendMessage:] + 516
ChatKit`-[CKTranscriptController sendMessage:] + 516:
-> 0x30a84e98:  blx    0x30b3bf44                ; symbol
stub for: MarcoShouldLogMadridLevel$shim
   0x30a84e9c:  mov    r2, r0
   0x30a84e9e:  ldr    r0, [sp, #40]
   0x30a84ea0:  mov    r1, r11
(lldb) p (char *)$r1
(char *) $0 = 0x30b55fb4
"imHandleWithID:alreadyCanonical:"
(lldb) po $r0
IMAccount: 0x145e30d0 [ID: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363 Service: IMService[iMessage] Login:
P:+86PhoneNumber Active: YES LoginStatus: Connected]
(lldb) po $r2
snakeninny@icloud.com
(lldb) p $r3
(unsigned int) $3 = 0
```

2个参数都搞定了，第1个就是iMessage地址，第2个是0（即BOOL的NO），那调用者，这个IMAccount对象是哪里来的呢？如图10-91所示，R0来自[SP,#0xA8+var_84]，所以根据提示图10-92和图10-93可知，IMAccount对象来自[[IMAccountController

sharedInstance]__ck_defaultAccountForService:

[CKConversation sendingService]]。

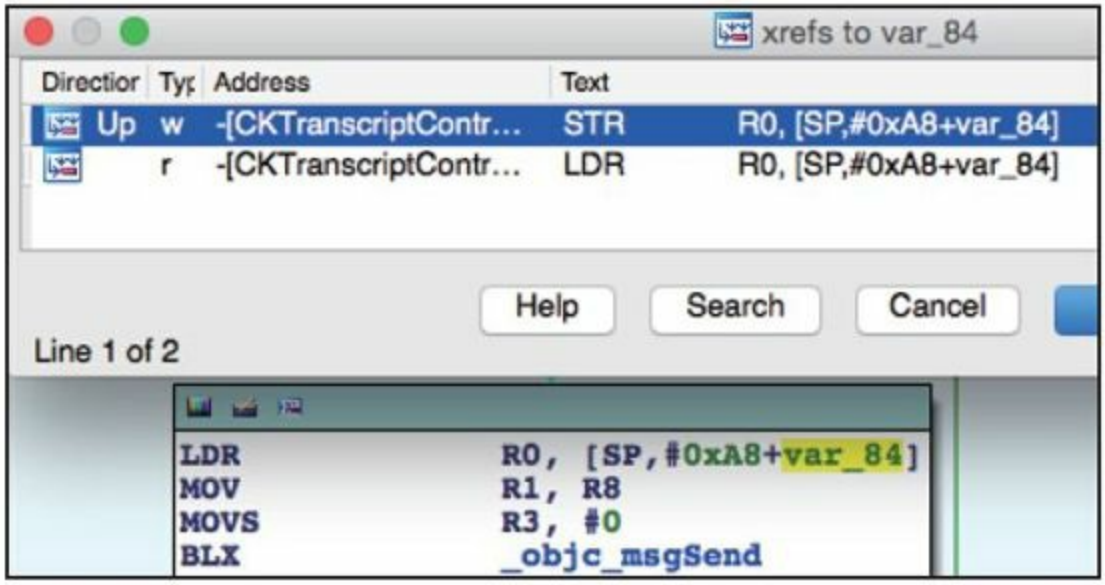


图10-92　查看交叉引用

```
LDR             R1, [R0] ; "sharedInstance"
LDR             R0, [R2] ; _OBJC_CLASS_$_IMAccountController
BLX             _objc_msgSend
MOVW            R1, #(:lower16:(selRef___ck_defaultAccountForService_
MOV             R2, R4
MOVT.W          R1, #(:upper16:(selRef___ck_defaultAccountForService_
ADD             R1, PC ; selRef___ck_defaultAccountForService_
LDR             R1, [R1] ; "__ck_defaultAccountForService:"
BLX             _objc_msgSend
STR             R0, [SP,#0xA8+var_84]
```

图10-93　[CKTranscriptController sendMessage:]

趁热打铁，在图10-93的第2个objc_msgSend上下一个断点，看看[CKConversation sendingService]是什么，如下：

```
Process 343388 stopped
* thread #1: tid = 0x53d5c, 0x30a84e08 ChatKit`-
[CKTranscriptController sendMessage:] + 372, queue =
'com.apple.main-thread, stop reason = breakpoint 2.1
    frame #0: 0x30a84e08 ChatKit`-[CKTranscriptController
sendMessage:] + 372
ChatKit`-[CKTranscriptController sendMessage:] + 372:
-> 0x30a84e08:  blx    0x30b3bf44                ; symbol
stub for: MarcoShouldLogMadridLevel$shim
   0x30a84e0c:  str    r0, [sp, #36]
   0x30a84e0e:  movw   r0, #23756
   0x30a84e12:  add    r2, sp, #44
(lldb) p (char *)$r1
(char *) $4 = 0x30b55f95 "__ck_defaultAccountForService:"
(lldb) po $r2
IMService[iMessage]
(lldb) po [$r2 class]
IMServiceImpl
```

可见，它是一个IMServiceImpl对象，那在我们自己的代码中，该如何得到这样一个IMServiceImpl对象呢？其实在10.2节中已经拿到这个类的对象了，打开IMServiceImpl.h，如图10-94所示。

图10-94　IMServiceImpl.h

其中的[IMServiceImpl iMessageService]就是了。用Cycript再次确认，如下：

```
cy# [IMServiceImpl iMessageService]
#"IMService[iMessage]"
```

到此为止，一个可用的IMChat类是如何生成的，被我们完整地逆向了出来。下面在Cycript里验证一下可行性，如下：

```
FunMaker-5:~ root# cycript -p MobileSMS
cy# service = [IMServiceImpl iMessageService]
#"IMService[iMessage]"
cy# account = [[IMAccountController sharedInstance]
__ck_defaultAccountForService:service]
#"IMAccount: 0x145e30d0 [ID: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363 Service: IMService[iMessage] Login:
P:+86PhoneNumber Active: YES LoginStatus: Connected]"
cy# handle = [account
imHandleWithID:@"snakeninny@icloud.com"
alreadyCanonical:NO]
#"[IMHandle: <snakeninny@icloud.com:<None>:cn> (Person:
<No AB Match>) (Account: P:+86 MyPhoneNumber]"
cy# chat = [[IMChatRegistry sharedInstance]
chatForIMHandle:handle]
#"<IMChat 0x15809000> [Identifier: snakeninny@icloud.com
GUID: iMessage;-;snakeninny@icloud.com Persistent ID:
snakeninny@icloud.com   Account: 26B3EC90-783B-4DEC-82CF-
F58FBBB22363   Style: -   State: 3  Participants: 1  Room
Name: (null)  Display Name: (null)  Last Addressed:
(null) Group ID: 6592DD84-4B34-4D54-BB40-E2AB17B2FC67
Unread Count: 0  Failure Count: 0]"
```

完美！最后的任务，就是构造一个可用的IMMessage对象，这样就可以实现iMessage的发送了，这就行动起来！

打开IMMessage.h，如图10-95所示。

```
@interface IMMessage : NSObject <NSCopying>
{
    IMHandle *_sender;
    IMHandle *_subject;
    NSAttributedString *_text;
    NSString *_plainBody;
    NSDate *_time;
    NSDate *_timeDelivered;
    NSDate *_timeRead;
    NSDate *_timePlayed;
    NSString *_guid;
    NSAttributedString *_messageSubject;
    NSArray *_fileTransferGUIDs;
    NSError *_error;
    unsigned long long _flags;
    BOOL _isInvitationMessage;
    long long _messageID;
}

+ (id)messageFromIMMessageItemDictionary:(id)arg1 sender:(id)arg2 subject:(id)arg3;
+ (id)messageFromIMMessageItem:(id)arg1 sender:(id)arg2 subject:(id)arg3;
+ (id)fromMeIMHandle:(id)arg1 withText:(id)arg2 fileTransferGUIDs:(id)arg3 flags:(unsigned long long)arg4;
+ (id)instantMessageWithText:(id)arg1 messageSubject:(id)arg2 fileTransferGUIDs:(id)arg3 flags:(unsigned long long)arg4;
+ (id)instantMessageWithText:(id)arg1 messageSubject:(id)arg2 flags:(unsigned long long)arg3;
+ (id)instantMessageWithText:(id)arg1 flags:(unsigned long long)arg2;
+ (id)defaultInvitationMessageFromSender:(id)arg1 flags:(unsigned long long)arg2;
+ (id)locationMessageWithGuid:(id)arg1 error:(id)arg2;
+ (id)messageWithLocation:(id)arg1 flags:(unsigned long long)arg2 error:(id)arg3 guid:(id)arg4;
+ (id)_vCardDataWithCLLocation:(id)arg1;
```

图10-95　IMMessage.h

又是一堆类方法，其中“instantMessageWithText:flags:”引起了我们的注意，这2个参数应该各传什么？针对第一个“text”传一个NSString进去试试，但第二个“flags”呢？不知道你还有没有印象，在本节的前面，当我们找到[IMChat

sendMessage：IMMessage]时，曾在LLDB中打印出了一个IMMessage的结构，如下：

```
(lldb) po $r2
IMMessage[from=(null); msg-subject=(null); account:
(null);flags=100005; subject='<< Message Not Loggable >>'
text='<<Message Not Loggable >>' messageID: 0
GUID:'966C2CD6-3710-4D0F-BCEF-BCFEE8E60FE9'
date:'437730968.559627' date-delivered:'0.000000' date-
read:'0.000000' date-played:'0.000000' empty: NO
finished: YES sent: NO read: NOdelivered: NO audio: NO
played: NO from-me: YES emote: NOdd-results: NO dd-
scanned: YES error: (null)]
```

这里的“text”无法显示，而“flags”是100005。在Cycript中试试，如下：

```
cy# [IMMessage instantMessageWithText:@"iOSRE test"
flags:100005]
-[__NSCFString string]: unrecognized selector sent to
instance 0x1468c140
```

Cycript告诉我们，NSString不能响应@selector(string)，也就是说，第一个参数并不是一个NSString对象，正确类型的参数应该是可以响应这个@selector(string)的。重新审视图10-95，看看能不能找出一些蛛丝马迹。注意到第17行

的“NSAttributedString*_text”了吗？查阅苹果官方提供的文档，NSAttributedString确实有一个“-(NSString*)string”方法，如图10-96所示。

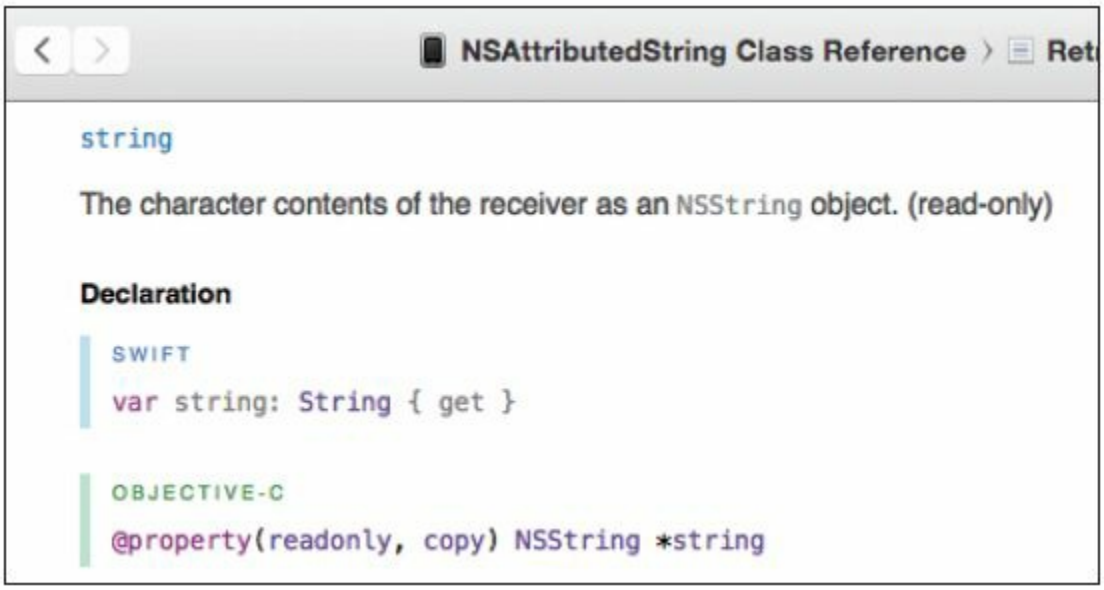


图10-96　[NSAttributedString string]

重新尝试传一个NSAttributedString对象进去试试，如下：

```
cy# attributedString = [[NSAttributedString alloc]
initWithString:@"iOSRE test"]
#"iOSRE test{\n}"
cy# message = [IMMessage
instantMessageWithText:attributedString flags:100005]
#"IMMessage[from=(null); msg-subject=(null); account:
(null);flags=186a5; subject='<< Message Not Loggable >>'
text='<<Message Not Loggable >>' messageID: 0
GUID:'00A8C645-D207-4F93-9739-07AAC94E7465'
date:'437812476.099226' date-delivered:'0.000000' date-
```

```
read:'0.000000' date-played:'0.000000' empty: NO
finished: YES sent: YES read:NOdelivered: NO audio: NO
played: NO from-me: YES emote: NOdd-results: YES dd-
scanned: NO error: (null)]"
cy# [attributedString release]
```

这里成功构造了一个IMMessage对象。接下来，“这是我生命中美好的时刻，我要完成我最喜欢的测试，在这美丽的月光下在这美丽的Cycript里”：

```
cy# [chat sendMessage:message]
```

效果如图10-97所示。打完收工！

图10-97　成功发送iMessage

10.4 逆向结果整理

相对于前几章的示例来说，本章展示的逆向工程的整体思路虽未变，但复杂程度增大了不少，尤其是与同为系统App的第7章Notes相比，两者的难度相差了若干数量级。为了逆向出看似很简单的iMessage检测和发送操作，大致的思路是下面这样的。

（1）从表面现象入手

“Text Message”变成“iMessage”，绿色变成蓝色，“Send”按钮，这些表层现象都来自于底层代码。只要能描述出所观察到的东西，就可以以此入手，开始逆向分析。本章，就是从信息输入框的占位符和“Send”按钮入手，通过Cycript定位到其实现代码，并切入代码层的。

（2）浏览class-dump出的头文件，找到感兴趣的点

Objective-C头文件结构清晰，函数名的含义明确，可读性强，是寻找蛛丝马迹的理想场所。用Cycript对那些简单的函数、属性、实例变量做测试，有助于我们对类的功能有大体了解。本章，在获取一些重要变量的时候，并没有严谨地借助IDA和LLDB这两样大杀器，而是仅靠阅读头文件，通过函数名猜测参数的类型与用法，配合Cycript进行测试，最终得出了理想的结果。黑猫白猫，抓到老鼠的就是好猫。

（3）在IDA中查看函数是如何形成一个面的

在查看函数内部实现时，IDA无疑是最好用的神器之一。不管是通过交叉引用、地址跳转，还是全局搜索，都可以快速定位关键词，并方便地浏览上下文，对关键词的前因后果有准确的把握。在检测iMessage时，我们用IDA理顺了[CKMessageEntryView updateEntryView]、[CKPendingConversation sendingService]、[CKPendingConversation

composeSendingService]和
IMChatCalculateServiceForSendingNewCompose等函数的调用关系，其中
IMChatCalculateServiceForSendingNewCompose是一个C函数，对class-dump免疫；在发送iMessage时，从上层的[CKTranscriptController
sendComposition:CKComposition]，一路经过
[CKTranscriptController_startCreatingNewMessageForSe
[CKConversation sendMessage:newComposition:]、
[CKConversation
sendMessage:onService:newComposition:]，追踪到了底层的[IMChat sendMessage:IMMessage]，都是依靠IDA提供的关键词及关联性从一个面中，手工地把那条线给挑出来的。虽然没有机器自动完成方便，但工作量也完全在可以接受的范围内，这都要归功于IDA提供的强大分析结果。

（4）用LLDB确认唯一的那条线

LLDB的使用贯穿本章的始终，即使是在有意“克制”的10.3节，我们也在寻找函数调用者、动态查看参数的时候“不得不”惊动LLDB它“老人家”。相对于GDB，LLDB对iOS的支持要好得多，基本不会出现崩溃等Bug，对于Objective-C的支持也很到位，让我们可以专注在调试本身上。在进行iMessage的检测及发送的环节中，用LLDB澄清了大量细节，通过对数据源一环扣一环的分析，基本厘清了iOS发送iMessage的一小段流程。由小见大，从中也可以窥探出苹果的设计思路：MobileSMS是一间邮局，邮局的建筑材料、办公设备和工作人员均来自ChatKit，而充当邮差工作的是IMCore。用户去邮局发一封信，他把信件放在邮筒里，由工作人员整理后交付给邮差，送信的进度和结果再由邮差反馈给工作人员，工作人员再通知用户，这样就完成了整个的服务流程闭环。三者各司其职，为果粉带来良好的用户体验；我们通过逆向工程学习到的这种设计思路，如果能融会贯通，运用到

自己的产品设计里去，给产品所带来的优雅度、设计感、健壮性都将是仅仅阅读开发文档所无法企及的。

10.5 编写tweak

在使用Cycript完成核心功能的测试后，用Theos编写代码就仅仅是简单的体力劳动了。本节，就用最直观的代码，给MobileSMS中的SMSApplication类添加2个实例函数，分别是“-(int)madridStatusForAddress:(NSString*)address”和“-(void)sendMadridMessage ToAddress:(NSString*)address withText:(NSString*)text”，然后用Cycript测试这2个类函数的有效性。开始行动！

10.5.1 用Theos新建tweak工程“iOSREMadridMessenger”

新建iOSREMadridMessenger工程的命令如下：

```
snakeninnys-MacBook:Code snakeninny$
/opt/theos/bin/nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/application
  [2.] iphone/cydget
  [3.] iphone/framework
```

```
    [4.] iphone/library
    [5.] iphone/notification_center_widget
    [6.] iphone/preference_bundle
    [7.] iphone/sbsettingstoggle
    [8.] iphone/tool
    [9.] iphone/tweak
    [10.] iphone/xpc_service
Choose a Template (required): 9
Project Name (required): iOSREMadridMessenger
Package Name [com.yourcompany.iosremadridmessenger]:
com.iosre.iosremadridmessenger
Author/Maintainer Name [snakeninny]: snakeninny
[iphone/tweak] MobileSubstrate Bundle filter
[com.apple.springboard]: com.apple.MobileSMS
[iphone/tweak] List of applications to terminate upon
installation (space-separated, '-' for none)
[SpringBoard]:MobileSMS
Instantiating iphone/tweak in iosremadridmessenger/...
Done.
```

10.5.2 构造iOSREMadridMessenger.h

在10.2节的检测及10.3节的发送中，用到了私有框架IDS、ChatKit和IMCore中的多个私有类和私有函数，我们必须给出它们的定义，才能避免编译器报错或警告。当然，iOSREMadridMessenger.h的内容并不是凭空构造出来的，所有的定义均来自于class-dump出的头文件，我们只是把用到的东西挑选出来，再整合到一个头文件里而已——我们构造的只是一个“精选头文件”。编辑后的iOSREMadridMessenger.h内容

如下：

```
@interface IDSIDQueryController
+ (instancetype)sharedInstance;
- (NSDictionary *)_currentIDStatusForDestinations:
(NSArray*)arg1 service:(NSString *)arg2 listenerID:
(NSString *)arg3;
@end
@interface IMServiceImpl : NSObject
+ (instancetype)iMessageService;
@end
@class IMHandle;
@interface IMAccount : NSObject
- (IMHandle *)imHandleWithID:(NSString *)arg1
alreadyCanonical:(BOOL)arg2;
@end
@interface IMAccountController : NSObject
+ (instancetype)sharedInstance;
- (IMAccount *)__ck_defaultAccountForService:
(IMServiceImpl*)arg1;
@end
@interface IMMessage : NSObject
+ (instancetype)instantMessageWithText:
(NSAttributedString*)arg1 flags:(unsigned long long)arg2;
@end
@interface IMChat : NSObject
- (void)sendMessage:(IMMessage *)arg1;
@end
@interface IMChatRegistry : NSObject
+ (instancetype)sharedInstance;
- (IMChat *)chatForIMHandle:(IMHandle *)arg1;
@end
```

10.5.3 编辑Tweak.xm

编辑后的Tweak.xm内容如下：

```
#import "iOSREMadridMessenger.h"
```

```
%hook SMSApplication
%new
- (int)madridStatusForAddress:(NSString *)address
{
      NSString *formattedAddress = nil;
     if ([address rangeOfString:@"@"].location !=
NSNotFound) formattedAddress = [@"mailto:"
stringByAppendingString:address];
      else formattedAddress = [@"tel:"
stringByAppendingString:address];
    NSDictionary *status = [[IDSIDQueryController
sharedInstance] _current
IDStatusForDestinations:@[formattedAddress]
service:@"com.apple.madrid" listenerID:
@"__kIMChatServiceForSendingIDSQueryControllerListenerID"

      return [status[formattedAddress] intValue];
}
%new
- (void)sendMadridMessageToAddress:(NSString *)address
withText:(NSString *)text
{
      IMServiceImpl *service = [IMServiceImpl
iMessageService];
      IMAccount *account = [[IMAccountController
sharedInstance] __ck_defaultAccountForService:service];
      IMHandle *handle = [account imHandleWithID:address
alreadyCanonical:NO];
      IMChat *chat = [[IMChatRegistry sharedInstance]
chatForIMHandle:handle];
      NSAttributedString *attributedString =
[[NSAttributedString alloc] initWithString: text];
     IMMessage *message = [IMMessage
instantMessageWithText:attributedString flags: 100005];
      [chat sendMessage:message];
      [attributedString release];
}
%end
```

10.5.4 编辑Makefile及control

编辑后的Makefile内容如下：

```
THEOS_DEVICE_IP = iOSIP
ARCHS = armv7 arm64
TARGET = iphone:latest:8.0
include theos/makefiles/common.mk
TWEAK_NAME = iOSREMadridMessenger
iOSREMadridMessenger_FILES = Tweak.xm
iOSREMadridMessenger_PRIVATE_FRAMEWORKS = IDS ChatKit
IMCore
include $(THEOS_MAKE_PATH)/tweak.mk
after-install::
      install.exec "killall -9 MobileSMS"
```

编辑后的control内容如下：

```
Package: com.iosre.iosremadridmessenger
Name: iOSREMadridMessenger
Depends: mobilesubstrate, firmware (>= 8.0)
Version: 1.0
Architecture: iphoneos-arm
Description: Detect and send iMessage example
Maintainer: snakeninny
Author: snakeninny
Section: Tweaks
Homepage: http://bbs.iosre.com
```

10.5.5 用Cycript测试

将写好的tweak编译打包安装到iOS后，执行ssh命令连接到iOS中，然后执行如下代码：

```
FunMaker-5:~ root# cycript -p MobileSMS
cy# [UIApp
madridStatusForAddress:@"snakeninny@icloud.com"]
1
```

```
cy# [UIApp
sendMadridMessageToAddress:@"snakeninny@icloud.com"
withText:@"Sent from iOSREMadridMessenger"]
```

“snakeninny@icloud.com”的检测结果是1，支持iMessage，且此条iMessage被成功发送，如图10-98所示。

图10-98　成功发送iMessage

如果你按照上面的思路和方法成功搞定了iMessage的检测和发送，就给“snakeninny@gmail.com”发一条iMessage吧！

10.6 小结

作为苹果在iOS 5之后重点打造的核心服务之一，iMessage的功能在iOS 8中得到了大幅度增强，不管是单纯的文字，还是多媒体照片、语音，甚至是视频，iMessage都能完美地hold住。本章的iMessage检测与发送虽然仅仅是所有iMessage操作的冰山一角，但都已经要在IDS、ChatKit和IMCore这3个模块间来回切换了，可见整个iMessage系统的复杂度之高。从上面的分析过程中可知道，负责管理iMessage发件人的是IMAccountController，作为发送人的我们是一个个IMAccount；收件人是一个IMHandle；一条对话就是一个IMChat或CKConversation；IMChatRegistry管理所有的对话；一条iMessage就是一个IMMessage或CKComposition。对于那些有意涉足即时通信的开发者来说，iMessage的这些设计方式非常值得借鉴。如果你对iMessage很感兴趣，觉得本章的内容仍意犹未

尽，不妨尝试分析笔者留下的下面3个“隐藏关卡”，它们由易到难，运用本章的所用到的逆向思路和技巧，就可以各个击破。

· 搞定SMS的发送（提示：只要更换IMServiceImpl对象即可）；

· 用ChatKit类搞定iMessage发送（提示：CKConversation对象可以由IMChat对象生成）；

· 把发送iMessage的操作移植到SpringBoard进程中（提示：在SpringBoard中调用[IMChat sendMessage:IMMessage]之所以无效，是因为SpringBoard缺少某种“capabilities”）。

如果本章的内容你能完全吃透，并“脱稿”完成，那么恭喜你，你已经是一名优秀的iOS逆向工程师了，可以朝着更高的目标（比如越狱？）迈进了。在开始新的征程前，先来我们的论

坛http://bbs.iosre.com，与各位同好分享这份喜悦吧！

越狱开发一览

Much has been said about Apple’s closed approach to selling devices and running an app platform.But what few know is that behind closed doors there’s a massive ecosystem of libraries and hardware features waiting to be unlocked by developers.All of the APIs Apple uses internally to build their services,applications,and widgets are available once the locks are broken via a process called jailbreaking.Most of them are written in Objective-C,a dynamic language that provides very rich introspection capabilities and has a culture of self-describing code.Further tearing down barriers,most people install something called CydiaSubstrate shortly after jailbreaking,which allows running custom code inside every existing process on the device.This is very powerful—not only have we broken out of the walled garden into the rest of the forest,all of the footpaths are

already labeled.Building code that targets jailbroken iOS devices involves unique ways of inspecting APIs,injecting code into processes,and writing code that modifies existing classes and finalized behaviors of the system.

The APIs implemented on iOS can be divided into four categories:framework-level Objective-C APIs,app-level Objective-C classes,C-accessible APIs and JavaScript-accessible APIs.Objective-C frameworks are the most easily accessible.Normally the structure of a component is only accessible to the programmer and those the source code or documentation have been made available to,but compiled Objective-C binaries include method tables describing all of the classes,protocols,methods and instance variables contained in the binary.An entire family of“class-dump”tools exists to take these method tables and convert them to header-like output for easy consumption

by adventurous programmers.Calling these APIs is as simple as adding the generated headers to your project and linking with the framework or library.The second category of app internal classes may be inspected via the same tools,but are not linkable via standard tools.To get to those classes,one has to have code injected into the app in question and use the Objective-C runtime function objc_getClass to get a reference to the class;from there,one can call APIs via the headers generated by the tool.Inspecting C-level functions are more difficult.No information about what the parameters or data structures are baked into the binaries,only the names of exported functions.The developer tools that ship with OS X come with a disassembler named“otool”which can dump the instructions used to implement the code in the device.Paired with knowledge of ARM assembly,the type information can be reconstructed by hand with much

effort.This is much more cumbersome than with Objective-C code.Luckily,some of the components implemented in C are shared with OS X and have headers available in the OS X SDK,or are available as open-source from Apple.JavaScript-level APIs are most often facades over Objective-C level APIs to make additional functionality accessible to web pages hosted inside the iTunes,App Store,iCloud and iAd sections of the operating system.

Putting the APIs one has uncovered to use often requires having code run inside the process where their implementations are present.This can be done using the DYLD_INSERT_LIBRARIES environment variable on systems that use dyld,but this facility offers very few provisions for crash protection and can easily leave a device in a state where a restore is necessary.Instead,the gold standard on iOS devices is a system known as Cydia

Substrate,a package that standardizes process injection and offers safety features to limit the damage testing new code can do.Once Cydia Substrate is installed,one needs only to drop a dynamic library compiled for the device in/Library/MobileSubstrate/DynamicLibraries,and substrate will load it automatically in every process on the device.Filtering to only a specific process can be achieved by dropping a property list of the same name alongside it with details on which process or grouping of processes to filter to.Once inside,one can register for events,call system APIs and perform any of the same behaviors that the process normally could.This applies to apps that come preinstalled on the device,apps available from the App Store,the window manager known as SpringBoard,UI services that apps can make use of such as the mail composer,and background services such as the media decoder daemon.It is important to note that

any state that the injected code has will be unique to the process it’s injected into and to share state mandates use inter-process communication techniques such as sockets,fifos,mach ports and shared memory.

Modifying existing code is where it really starts to get powerful and allows tweaking existing functionality of the device in simple or even radical ways.Because Objective-C method lookup is all done at runtime and the runtime offers APIs to modify methods and classes,it is really straightforward to replace the implementations of existing methods with new ones that add new functionality,suppress the original behavior or both.This is known as method hooking and in Objective-C is done through a complicated dance of calls to the class_addMethod,class_getInstanceMethod,method_getIm and method_setImplementation runtime functions.This is very unwieldy;tools to automate this have been built.The

simplest is Cydia Substrate's own MSHookMessage function.It takes a class,the name of the method you want to replace,the new implementation,and gives back the original implementation of the function so that the replacement can perform the original behavior if necessary.This has been further automated in the Logos Objective-C preprocessor tool,which introduces syntax specifically for method hooking and is what most tweaks are now written in.Writing Logos code is as simple as writing what would normally be an Objective-C method implementation,and sticking it inside of a%hook ClassName...%end block instead of an@implementation ClassName...%end block,and calling%orig(） instead of[super...].Simple tweaks to how the system behaves can often done by replacing a single method with a different implementation,but complicated adjustments often require assembling numerous method hooks.Since

most of iOS is implemented in Objective-C,the vast majority of tweaks need only these building blocks to apply the modifications they require.For the lower levels of the system that are written in C,a more complicate hooking approach is required.The lowest level and most compatible way of doing so is to simply rewrite the assembly instructions of the victim function.This is very dangerous and does not compose well when many developers are modifying the same parts of the system.Again,CydiaSubstrate introduces an API to automate this in form of MSHookFunction.Just like MSHookMessage,one needs only to pass in the target function,new replacement implementation function,and it applies the hook and returns the old implementation that the new replacement can call if necessary.With the tools the community has made available,the details of the very complex mechanics of hooking have been abstracted and

simplified to the point where they’re hidden from view and a developer can concentrate on what new features they’re adding.

Combining these techniques unique to the jailbreak scene,with those present in the standard iOS and OS X development communities yields a very flexible and powerful tool chest for building features and experiences that the world hasn’t seen yet.

（关于苹果的封闭性可谓路人皆知。但其实这扇紧闭的大门背后有不计其数的宝藏在等待着开发者们让它们重见天日——这些宝藏就是苹果内部使用的所有API，而让它们重见天日的过程就是越狱。这些API中的大多数都是使用Objective-C——一门动态性强、语义清晰的语言——编写的。越狱之后，为了进一步扩展iDevice的功能，绝大多数人都会安装CydiaSubstrate，它使我们能够在其他进程中运行自己

编写的代码。它的意义非比寻常——我们不但破除了苹果的限制，还能够以其人之道，还治其人之身！越狱iOS开发与普通的App Store开发不尽相同，它需要你去摸索API的用法，把代码注入别的进程，修改现有的类，最终达到改变系统行为的目的。

iOS调用的API可以分为4类：framework使用的Objective-C API、App使用的Objective-C API、C API和JavaScript API，其中framework层面的API是最容易使用的。一般情况下，一个程序的设计、代码、文档只为少数人所有，但经过编译的Objective-C二进制文件中含有所有的类、协议、方法和实例变量信息，“class-dump”大家族的工具可以把这些信息导出来，形成头文件，供那些好奇的开发者研究并调用。App使用的API可以通过同样的方式导出，但不能直接调用。这时，可以把代码注入对应的进程里，然后调用Objective-C的运行时函数objc_getClass来获取一个类的对象，进而调用类方法。C API就要复杂一

点，没有class-dump这样的工具把C函数的参数和数据结构给导出来，能找到的只有导出函数的函数名；但是，利用OSX自带的反汇编工具otool，结合一些ARM汇编语言的知识，我们可以手动分析，还原C函数的原貌，这个过程比分析Objective-C方法要复杂得多。幸运的是，iOS中的C实现与OSX是部分相通的，有一些C函数的头文件可以从OSX的SDK中找到参考，还有一些是开源的。JavaScript API通常只是Objective-C API的封装，供iTunes、App Store、iCloud和iAd等应用中的网页调用。

通过逆向工程得知的API通常需要注入对应的进程才能调用，因为只有这些进程中含有API的实现。进程注入可以通过DYLD_INSERT_LIBRARIES方式实现，但这种方式提供的防崩溃保护不够强，很容易导致设备白苹果，只有重刷系统才能解决。更好的替代方案是CydiaSubstrate，它为进程注入制定了一套标准，并引入了更加安全的防崩溃保护机制。安装

CydiaSubstrate之后，只需要把一个dylib放到/Library/MobileSubstrate/DynamicLibraries下即可，CydiaSubstrate会自动在每个进程启动时尝试把这个dylib加载进去；我们可以进一步通过编写一个plist指定dylib加载的对象。一旦我们的代码得到注入，就可以监听事件，调用内部API，做这个进程本身能够做的任何事。这种注入方式可以用在任何进程上：系统App、App Store App、iOS上的窗口管理器SpringBoard、UI服务（如MailComposer）和守护进程（如mediaserverd）。值得一提的是，我们的代码只在注入的进程内部生效，如果要使其跨进程作用，则需要用到sockets、fifos、mach ports和shared memory等技术。

从我们能够通过简单方便的CydiaSubstrate更改现有的代码开始，我们能做的事就多了起来。因为Objective-C方法调用都是在运行时决定的，而Objective-C的运行时函数提供了修改类和方法的

API，所以这个过程就比较直观了，大体是通过class_addMethod、class_getInstanceMethod、method_getImplementation和method_setImplementation这4个运行时函数来实现hook的功能。这个过程并不困难，但比较繁复，许多工具就是为了自动化这个过程而存在的，其中比较简单的是CydiaSubstrate提供的MSHookMessage函数，它有4个参数，分别是：一个类、一个你要hook的方法名、一个新的实现和一个原始实现。现在越狱社区又出现了一种更简单的Logos预处理工具，它引入了专为hook设计的语法，因此在广大越狱开发者中流行了起来。Logos语法与Objective-C语法十分类似，把@implementation ClassName...%end的首尾替换掉，改成%hook ClassName...%end，把[super...]替换为%orig就行了。简单地更改系统功能往往只需要hook一两个独立函数，把它们的实现改掉就可以了；但多数情况下我们需要组合hook好几个函数并协调它们之间的调用关

系。因为iOS的大部分功能是用Objective-C写的，所以大多数tweak仅用上面提到的语法就够了，但一旦涉及了更底层的C函数，hook的方法就要复杂许多，一般需要重写一段ARM汇编指令，当然这种方法的危险系数极高，不建议一般开发者尝试。好在CydiaSubstrate也提供了一个简单的API，即MSHookFunction，调用者只需要把需要hook的函数及其新的实现作为参数传进去就可以了。越狱社区提供的这些工具把hook操作复杂的一面给抽象化了，封装成几个简单的接口供开发者使用，从而使我们能够把精力集中在tweak的编写上。

把这些越狱社区独有的技术同App Store开发的普遍性技术结合起来，我们得到了一套用法灵活、功能强大的工具集，利用这套工具能够打造的功能也将是前所未有的。）

Ryan Petrich

牧“码”人，Activator之父

沙箱逃脱

As a security measure and to keep apps on the device from sharing data or interfering with each other,iOS includes a security system known as the sandbox.The sandbox blocks access to files,network sockets,bootstrap service names,and the ability to spawn subprocesses.Part of the jailbreaking process involves modifying the sandbox so that all processes can load Cydia Substrate,but much of the sandbox is left intact to respect the security and privacy of the user’s data.

With each new release,Apple further improves the sandbox to improve privacy and security.When building extensions or tweaks that need to share information across processes or persist data to disk,this can be restrictive.One approach is to survey the sandbox restrictions that exist on the processes where the

extension is to be run,and choose file paths and names based on them.This is common,but can leave oneself stranded when Apple tightens the tourniquet and as of iOS 8 there is no location that all processes can read and write successfully.A better approach is to do all of the interesting work inside a privileged process such as SpringBoard,backboardd or even a manually created launch daemon of your own.Child processes can then send work to the privileged service.This ensures that as the sandbox tightens,your extension will still behave properly as long as it can communicate with the service.

Oddly enough,as of iOS 8 Apple has also decided to limit which services an app store process may query.This makes nearly all forms of inter-process communication ineffective on iOS,outside of the well-defined static services that Apple has designated.RocketBootstrap was created as a way around this that simultaneously allows

additional services to be registered and respects the security and privacy of the user’s data.Services registered with RocketBootstrap are made globally accessible even in spite of very restrictive sandbox rules and it will serve as a single project that needs updating as the rules change.

（出于安全考虑，也为了让每个App运行在自己的独立空间里，iOS引入了名为“沙箱”（sandbox）的安全体系。沙箱会拦截文件访问、网络套接字、Bootstrap服务，以及对子进程的spawn。越狱操作对沙箱作出了适量修改，使所有进程都能够加载CydiaSubstrate，但为了用户的隐私安全，大多数沙箱限制仍保留原样。

在每一版iOS中，苹果公司都会增强沙箱的作用。当我们的tweak需要跨进程访问数据或向硬盘写数据时，沙箱会给我们的操作带来限制。绕过这些限

制的方案之一是视tweak运行在哪些进程中而定，选择一个这些进程都能访问的路径或文件，达到共享数据的目的。这是一种常规的方法，但一旦苹果公司再次收紧沙箱的限制，这种方法就可能失效，比如iOS 8中已经不存在一个所有进程皆可读写的路径了。另一种更好的方案是把类似操作放在权限高的进程，如SpringBoard、backboardd，甚至是我们自己编写的守护进程里去完成，我们的tweak所在的进程只需要把受限制的操作丢给这些高权限的进程，然后坐等结果就可以了。这么做的好处是，即使沙箱的限制越来越严，只要tweak所在的进程能够与高权限进程通信，tweak就能够正常运行。

奇怪的是，在iOS 8里，苹果公司甚至限制了AppStore App所能访问的服务（即能够通信的进程），导致几乎所有AppStore App的进程间通信都失效了。RocketBootstrap应运而生，它既破除了上面提到的访问限制，又较好地保留了沙箱的防护。向

RocketBootstrap注册的服务可以被全系统进程访问，包括那些沙箱限制非常严格的进程；RocketBootstrap是一个单独的程序，随着苹果限制规则的变化，我会不断调整RocketBootstrap的代码使其能够正常运行。）

Ryan Petrich

编写tweak——新时代的hacking

I am not a prolific programmer by any means.I have a programmer’s mind,and I have proven in my days I am capable of writing working solutions.I have a few tweaks in my name,and more ideas to be realized.Creating more has been about having more free time.However,my time has been spent becoming familiar with iOS-internals,because I find that I am a good learner.I have a fair understanding due to the tools we have available,made by great programmers before our time,and from documentation and examples shared by the community.Because of the nature of Cocoa and Objective-C,we can take a great adventure and introspection into the workings of third-party software,and Apple’s operating system.This provides a foundation and skills for making tweaks.We want to encourage tweak making because it has been the driving

initiative behind the audience that wants to have jailbroken devices,besides for the groups that wish to only have a jailbreak for pirating apps and games.The growth of this jailbreak ecosystem has gone with the proliferation of new tweaks,ever pushing the boundaries of modification while maintaining a safe environment for the end-users.

The jailbreak development scene has given a unique opportunity to developers to express themselves in a new way.In the days before CydiaSubstrate,apps and games were not tweaked.This is a new concept;examining and debugging existing software and then rewriting portions of it with the least invasive tools available,the changes are nonpermanent and for the most part free of worry for breaking something with any lasting effect.Tweaks allow for a redefining of how software works and behaves.We do this with tweaks,and there has really been nothing like

it before in the world of programming,even on the PC.There were opportunities throughout previous decades to make game patches,hacks and so forth,but it’s only with the emergence of the audience of jailbreakers and iOS that we find our unique situation.Only recently has it become feasible to make small adjustments to existing UI and modify how things work without requiring the replacement of whole parts of the code-CydiaSubstrate allows careful targeting of methods and functions.

It’s a lot of fun to discover how things work,and tweak making is the embodiment of that fun time for developers.One of the challenges for tweak making is coming up with new ideas to create,and sometimes these ideas only arise after studying the internals in some detail.If you make tweaks as a hobby,and not as a profession,you’re free to do as you wish and to focus on

projects that interest you.For new tweak makers,there’re quite a lot of existing projects to learn from,but a lot of the easier projects have already been realized.Creating new original ideas that are unique is a task of being familiar with the available tweaks on Cydia,and then going to work discovering how the internal parts work,debugging and testing until you have a diagram or picture in your mind how it’s put together.When you reach a near complete understanding,you are primed to tackle whatever challenge you make for yourself.

Some of our greatest tools and resources are free：Apple’s own documentation is excellent,and for tweak makers we have a wiki and the opportunity to use class-dump to examine what methods are exposed for hooking inside the target app or process.Debugging and disassembly tools that vary from free to paid,all can be great assets for tweak makers.A well-studied programmer

with some prior experience with standard projects will be in a good position to continue learning from these materials.To the contrary,a newcomer programmer,even a person with some good ideas will struggle at first with the learning curve.We recommend a core understanding of Objective-C and Cocoa principles for aspiring young tweak makers;this can be a significant investment of time,but it is really a hurdle for new tweak makers that haven’t a clue where to start.To the uninitiated,the object-oriented nature of the programming involved can be a daunting thing to realize.Generating tweak ideas can be a task for amateurs,but the writing of the code for the tweak implementation is often the result of planning and research and testing for a significant time.We find that many young new programmers are impatient because their ideas for new tweaks do take more time to materialize than they were willing to invest.Patience is a

virtue of course,and the best-made tweaks are all products of careful programmers.

The greatest tweak is Activator(libactivator).Based on a commonsense idea of having more triggers system-wide,activator is also a graciously open-sourced project;the product of many months and years of work by our most senior tweak maker,Ryan Petrich.His dedication and expertise shines through in Activator,which doubles as a platform for third-parties to harness the powerful triggers from anyplace to use in their own projects.It represents a lot of research and understanding of the most obscure internals on iOS：SpringBoard and backboard.If there is one shining example to point to as a goal for a tweak maker to show how much research and careful planning can go into a tweak,that is the example to look towards.It’s a lofty project that none should consider as being trivial to

do,however.For some aspiring developers it can be a great encouragement to see what is possible.Kudos to Ryan Petrich for making it,and for all he does to further the jailbreak development community.

As the repo maintainer for TheBigBoss,I have a job description for myself.Doing my job has given tremendous opportunity to be an influence or guidance for new tweak makers.Often their first experience with another member of the jailbreak development community is with myself when they first contact me or submit to the repo.We wish that all developers can be involved in the social channels of this scene：chat,forums,twitter et al.,however,it’s not uncommon that some developers work in relative isolation from these social groups.My involvement then can be seen as important：I may be the only other voice that the programmer will hear,and I will give an opinion on the technical merits of new tweak

projects;often this first encounter is invaluable because those developers that work in isolation are not wise to many of the caveats and conventions we hold as important in this community.Our documentation and wikis have improved to make these details more available,but still I am often the first time a developer has some interaction with someone with a greater expertise than their own.I try to give my wisdom and guidance to the developers because its in our best interest to support,if not groom,newcomer developers so they feel as part of the group of jailbreak developers,and they can be pointed towards ways to avoid some of the pitfalls that many newcomers make.I take some pride in doing this and helping in part to strengthen the developer community that is based around the tweak-making culture.I want the jailbreak platform to continue to grow and mature by the great ideas that are envisioned and the

expertise to realize them.

Do not be discouraged when the task seems difficult.We have some developers with years and decades of programming experience,and we also have some with only a few weeks or months.I come from the school of thought that it should be well made and well tested,and not rushed or forced.If you have a goal,it should not be merely to have something of yours published on a Cydia repo,but to give something to the public,which will enrich their jailbreak experiences-that is for hobbyists like myself.If you have some commercial interest in Cydia,and for making an selling tweaks,do wide testing with users and alongside other tweaks to help assure a product that works for many users and their combinations of tweaks;your duty as a responsible tweak maker is to be careful while you modify the insides of others'programs or apps,and to be thorough in testing

compatibility with others’tweaks.

Tweak making is the new-age hacking.There’re already enough reasons for you to get started with tweak development,and we need tweaks to keep the jailbreak community in bloom.Join us,learn from others,work hard,be patient,and have fun.

（怎么说我都不算是一个高产的程序员，不过我的思维方式完全是程序员式的，因为在写程序时，我已经证明了自己具备解决问题的能力。虽然我以个人开发者的身份发布了几款tweak，但是还有很多想法没来得及实现，因为我的时间大都花在研究iOS内部构造上了，而且作品越多，意味着花在它们上面的时间就越多，所以……

通过使用前辈们创造的分析工具，参考iOS社区共享的文档和例子，我对iOS的了解还算深入。因为Cocoa和Objective-C语言的本质，我们有机会观察分

析iOS及其软件的工作原理，这些都是编写tweak的先决条件。很多越狱iOS用户越狱自己设备的根本原因就是要安装那些方便好用的tweak，这也是我们鼓励iOS程序员来研究/开发tweak的原因之一，而绝不是为了安装盗版软件。随着新tweak的出现，这种越狱生态系统也在不断蓬勃发展，同时也促成iOS上发生越来越多的改变。

在传统AppStore开发之外，越狱开发提供了一种非常独特的方式让开发者们表达自己的想法。在CydiaSubstrate出现之前，tweak的概念十分抽象，而现在它已经有了非常具体的定义：调试分析现有的软件，然后重写它的某个部分，并尽可能减小对其他部分的影响。重写所造成的改变并不是永久的，即使这种改变破坏了原有软件，也可以很方便地修复。用一句话概括就是：tweak重新定义了软件的功能。在编程的历史中，这种概念前所未有。在过去，做游戏修改器、给软件打补丁是很常见的现象，但随着iOS和

越狱平台的出现，借助CydiaSubstrate的力量，我们可以把对软件的修改精确到函数级别，这是越狱开发的一大特色。

探索事物的工作原理是一件充满乐趣的事情，开发者编写tweak就是一例。在编写tweak之前，我们最先面对的挑战就是寻找灵感，而灵感往往是在对iOS进行重重分析之后才产生的。初学者可以从很多现成开源工程中学习，但很多简单的想法都已得到实现。如果想要原创tweak，那么首先你要熟悉Cydia上现有的tweak，然后分析它们的实现方法，直到理清所有逻辑，并且有把握编写相同功能的tweak。不断地重复这个过程，等你熟悉这个套路之后，也就基本具备把灵感变成tweak的能力了。

投入到iOS越狱开发并不是一件很无助的事情，因为许多最常用的工具和资源都是免费的：Apple官方文档编写十分详尽，iphonedevwiki上有很多有价值

的信息，class-dump可以导出详尽的头信息。而且调试和反汇编工具也有免费的，如IDA demo和LLDB，这些都是编写tweak的利器。一名合格的AppStore开发者完全可以从这个角度入手，开始越狱开发之路；不过如果是一名纯菜鸟，你的日子可能就不那么好过了，我建议你先完整学习Objective-C和Cocoa的概念及原理，之后再考虑进入越狱开发这个领域。虽然前期的概念学习过程需要花费较长时间，但这是新手必须迈过的坎。小小提醒：Objective-C语言的面向对象特性可不是那么好懂的。虽然tweak的创意构思不设门槛，普通用户即可轻松完成，但实现的过程却需要程序员投入大量时间。在这些年的经历中，我发现很多初学者都不够耐心，他们有很好的想法，但是太急于求成，一旦发现完成tweak所需的时间多于预期，就打退堂鼓了。殊不知，Cydia中最出色的tweak，往往都是由那些有耐心、能坚持的开发者完成的。

比如Activator，它是迄今为止我眼中当之无愧的

tweak界无冕之王，但它的创意并不高深，就是把常见的手势使用范围扩展至全系统（rpetrich不仅将它开源，还为第三方应用提供了接口，大将风范展露无疑）。虽然这个概念看似每个iOS用户都想得到，但Activator是iOS越狱社区最顶尖的开发者之一rpetrich经年累月完成的作品，凝结了他智慧的结晶，难度可想而知。Activator饱含rpetrich对iOS中最晦涩的SpringBoard和backboard的深入研究，如果一个tweak开发者想要拿一个tweak作为自己努力的终极目标，那么Activator再合适不过了。千万不要以为Activator完成的工作难度不大，如果你觉得自己足够强，可以试试看能不能写出同样功能的tweak。在此向rpetrich致敬！

作为TheBigBoss源的管理员，简单地说，我的工作是积极影响、向上引导初学者。对于这些人来说，我往往是他们进入越狱开发社区后碰到的第一个圈内人。我希望所有越狱开发者都能通过各种渠道，比如

聊天工具、论坛、微博等，进行交流，但很多情况下还是有些开发者不善交流，独自工作，而这时我的作用就凸显出来了：他们往TheBigBoss源提交作品，就必须跟我交流，而我也会从技术角度评价他们tweak的优劣。他们或许不熟悉这个圈子的各种规则，于是我就会对他们进行科普，让他们少走些弯路。我愿意尽可能向他们提供帮助，让他们感觉自己是这个大圈子中的一员。我为自己的所作所为感到骄傲，因为我为越狱社区的文化建设贡献了一份力量。我衷心希望越狱社区能蓬勃发展，各种创意层出不穷，各类人才百花齐放。

当你碰到难题时，不要气馁。在越狱社区里，有些程序员已身经百战，也有些才刚刚上路，不管你属于哪一类，都应该以严肃、认真、负责的态度对待自己的作品，水到方能渠成，揠苗焉能助长？你要设立的目标不应仅仅是在Cydia上发布什么软件，而是分享你的经验，让大家受益于此。当然，这是站在非商

业角度来说的。如果你想要靠卖Cydia软件赚钱，就尽可能广泛地测试你的软件，确保它能正常工作。作为一个负责任的tweak开发者，你要记住，你是在改动别人的软件，还要和其他的tweak兼容，所以要慎之又慎。

tweak开发是新时代的hacking。你已有足够的理由来说服自己加入tweak开发的大部队，越狱社区也需要tweak来保持旺盛的生命力。加入我们，从中学习，好好努力，保持耐心，你一定会感到由衷的快乐。）

Optimo

Cydia中最知名默认软件源TheBigBoss的管理员

Table of Contents

推荐序一 8
推荐序二 11
第2版序 15
第1版序 18
前言 28
第一部分 概念篇 37
第1章 iOS逆向工程简介 38
1.1 iOS逆向工程的要求 39
1.2 iOS应用逆向工程的作用 41
1.3 iOS应用逆向工程的过程 52
1.4 iOS应用逆向工程的工具 56
1.5 小结 62
第2章 越狱iOS平台简介 63
2.1 iOS系统结构 64
2.2 iOS二进制文件类型 81
2.3 小结 97
第二部分 工具篇 98
第3章 OSX工具集 100
3.1 class-dump 101
3.2 Theos 107
3.3 Reveal 167
3.4 IDA 178
3.5 iFunBox 208
3.6 dyld_decache 210
3.7 小结 213
第4章 iOS工具集 214
4.1 CydiaSubstrate 215
4.2 Cycript 241
4.3 LLDB与debugserver 252
4.4 dumpdecrypted 291

4.5 OpenSSH 299
4.6 usbmuxd 303
4.7 iFile 306
4.8 MTerminal 309
4.9 syslogd to/var/log/syslog 311
4.10 小结 314
第三部分 理论篇 315
第5章 Objective-C相关的iOS逆向理论基础 316
5.1 tweak在Objective-C中的工作方式 317
5.2 tweak的编写套路 324
5.3 实例演示 358
5.4 小结 385
第6章 ARM汇编相关的iOS逆向理论基础 387
6.1 ARM汇编基础 389
6.2 tweak的编写套路 419
6.3 LLDB的使用技巧 512
6.4 小结 528
第四部分 实战篇 530
第7章 实战1：Characount for Notes 8 532
7.1 备忘录 532
7.2 搭建tweak原型 535
7.3 逆向结果整理 563
7.4 编写tweak 565
7.5 小结 576
第8章 实战2：自动将指定电子邮件标记为已读 577
8.1 电子邮件 577
8.2 搭建tweak原型 581
8.3 逆向结果整理 621
8.4 编写tweak 624
8.5 小结 634
第9章 实战3：保存与分享微信小视频 635
9.1 微信 635
9.2 搭建tweak原型 640
9.3 逆向结果整理 695

9.4 编写tweak 697
9.5 彩蛋放送 706
9.6 小结 713
第10章 实战4：检测与发送iMessage 714
10.1 iMessage 714
10.2 检测一个号码或邮箱地址是否支持iMessage 716
10.3 发送iMessage 805
10.4 逆向结果整理 862
10.5 编写tweak 867
10.6 小结 874
越狱开发一览 877
沙箱逃脱 892
编写tweak——新时代的hacking 897