PEARSON

移动开发经典丛书

AV Foundation 开发秘籍

——实践掌握iOS & OS X应用的视听处理技术

Learning AV Foundation:
A Hands-on Guide to Mastering the AV Foundation Framework

PEARSON

[美] Bob McCune 著
冯宗翰 孙婷婷 译

清华大学出版社

快速掌握使用AV Foundation开发iOS和OS X应用的技术

如果想开发富媒体iOS或OS X应用程序，你可以使用苹果公司推出的AV Foundation框架来实现诸多神奇功能。不过AV Foundation框架基于最前沿的语言特征，极其庞大，而且相关文档较少；现在，Bob McCune撰写的这本《AV Foundation开发秘籍——实践掌握iOS & OS X应用的视听处理技术》为你铺就一条精通掌握这些卓越技术的高效途径。

McCune生动的讲解和精彩的GitHub示例项目已帮助数以千计的苹果开发者领略到AV Foundation的魅力。McCune基于丰富的开发经验，带你创建紧贴实用的应用程序，让你切实掌握相关技术。

通过学习本书，你将创建语音备忘录、自定义视频播放器、视频编辑器和图片/视频相机应用程序，将掌握从头开发高级媒体应用程序需要的所有技术。每章都提供了McCune精心编写的“挑战”内容和完整的初始应用程序，使你更深入地理解并最大限度地利用突破性的AV Foundation媒体框架。

涵盖内容

- 了解AV Foundation如何应用于苹果媒体环境及其每个核心功能区域
- 为应用添加音频播放和录制功能
- 为视频播放功能增加更丰富、更动态的体验
- 使用AV Kit让程序拥有类似于iOS Videos和OS X QuickTime播放器的用户界面
- 使用OpenGL ES和Core Image开发具有实时视频处理功能的媒体捕捉应用
- 实现自定义视频播放器
- 整合高级捕捉功能，如视频缩放、人脸检测、条码扫描和高帧率捕捉
- 混合多音频轨道和自动音量控制
- 创建高级的非线性、无损编辑应用
- 添加高级视频编辑功能，如视频过渡和动画效果

Bob McCune是iOS咨询和培训公司TapHarmonic(位于明尼苏达州明尼阿波利斯市)的所有者。他拥有丰富的iPhone和iPad开发经验，对相关的技术挑战和机遇了如指掌。McCune目前帮助多个客户开发大型企业级iOS应用程序。他的2012 Infinite Skills 视频培训课程*Learning iOS Programming*已成为全球最优秀的iOS培训课程之一。他创立并领导着Minnesota CocoaHeads用户组。

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

PEARSON
www.pearson.com

ISBN 978-7-302-40550-4
9 787302 405504 >
定价：49.80元

移动开发经典丛书

AV Foundation 开发秘籍

——实践掌握 iOS & OS X 应用的视听处理技术

[美] Bob McCune 著
冯宗翰 孙婷婷 译

清华大学出版社
北 京

Authorized translation from the English language edition,entitled Learning AV Foundation: A Hands-on Guide to Mastering the AV Foundation Framework, 978-0-321-96180-8 by Bob McCune, published by Pearson Education, Inc, publishing as Addison-Wesley, Copyright © 2015.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and TSINGHUA UNIVERSITY PRESS Copyright © 2015.

北京市版权局著作权合同登记号 图字：01-2015-2185

本书封面贴有 Pearson Education(培生教育出版集团)防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

AV Foundation 开发秘籍——实践掌握 iOS & OS X 应用的视听处理技术 / (美) 麦昆(McCune, B.) 著；冯宗翰，孙婷婷 译. —北京：清华大学出版社，2015
(移动开发经典丛书)
书名原文：Learning AV Foundation: A Hands-on Guide to Mastering the AV Foundation Framework
ISBN 978-7-302-40550-4

Ⅰ. ①A… Ⅱ. ①麦… ②冯… ③孙… Ⅲ. ①多媒体软件—图形软件 Ⅳ. ①P391.41

中国版本图书馆 CIP 数据核字(2015)第 137447 号

责任编辑：王　军　韩宏志
封面设计：牛艳敏
版式设计：思创景点
责任校对：邱晓玉
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：三河市君旺印务有限公司
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：20.5　　字　　数：512 千字
版　　次：2015 年 7 月第 1 版　　印　　次：2015 年 7 月第 1 次印刷
印　　数：1～3500
定　　价：49.80 元

产品编号：062459-01

译 者 序

随着 iPhone6 和 iPhone6 Plus 的推出，苹果手机又一次让世人眼前一亮。苹果公司的伟大就在于它的每一个产品都臻于完美，让使用者赞叹不已。我也是一个不折不扣的“果粉”，我第一个使用到的苹果产品是在 2009 年收获的一个小小的 iPod；至今它仍在服役，而且无论是外形还是音质都处于业内领先地位。iPod 的成功为苹果公司日后在数字媒体方面的发展奠定了坚实基础，随后推出的 iTouch 更是成为 MP4 领域中的宠儿，那时能拥有一个价格不菲的 iTouch 成为年轻人和音乐发烧友的梦想。之后 iPhone 的推出更是将苹果在数字媒体领域的地位带向高潮，用户可在 iPhone 手机上浏览清晰的图片、聆听高质量的音乐和欣赏高清的影片。苹果在数字媒体界的霸主地位也催生了多媒体开发这一庞大市场，随着 iPhone 手机硬件的进步，越来越多之前看似难以实现的功能都出现在用户面前，数字媒体领域的革命也正在进行中。

在过去几年中，数字媒体正在发生着革命性的变化。iPhone 的推出和移动计算的兴起，以及高速网络的发展，从根本上改变了人们创建、使用和分享数字媒体的习惯。现在我们乘坐公交车和地铁，常会看到用手机听音乐、看视频的年轻人。感谢苹果公司推动数字媒体的发展，让我们可以随时随地欣赏手机中的视频资源，让视频这一形式又焕发了青春。另外拍摄个性、高质量的图片也不再是高级相机的专属，有些手机拍摄的照片已经可以和专业机器拍摄的照片相媲美了。iPhone 中的照片应用程序数不胜数，它们都极具个性和特点，现在只需要一个笔记本或者一台 iPhone 手机，就可以创造出专业级照片！数字媒体革命方兴未艾，对于 iOS 和 OS X 来说，这一革命的核心技术就是 AV Foundation，也正是本书介绍的主要内容。

Bob McCune 撰写的这本著作是一本专业性很强又通俗易懂的程序开发指导书籍，它从 AV Foundation 最基础的知识入手，通过一个个真实的例子指导读者朋友进行实际操作，其中对 AV Foundation 框架涉及的所有知识进行了全面的渗透。AV Foundation 涉及的内容很多，有音频的内容和视频的内容，作者很好地设计了本书的结构，并循序渐进地展开讲解，重点内容必配有相关程序的解读，这让阅读本书的开发者能更清晰地理解一些深奥的算法。

我有幸拜读本书并完成本书的翻译工作，就像是同这样一位 iOS 开发界的大佬对话一般，怀着欣喜和紧张完成了本书的翻译工作。期间不敢有一点怠慢和疏忽，在这里要感谢清华大学出版社李阳编辑对我帮助，他为本书的翻译投入了巨大热情并付出了很多心血。没有你的

帮助和鼓励，本书不可能顺利付梓。也谢谢我的家人，是你们的支持让我能够专心完成本书的翻译工作。虽然在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。感激不尽！本书所有章节由冯宗翰翻译，参与本次翻译活动的还有孙婷婷、许亦男、孙伟、冯树彪、张怀洲、李志锋、李爽等，在此一并谢过。

最后，希望读者通过阅读本书能够领略 AV Foundation 框架的魅力，实现自己的 Apple 软件开发梦，创造出改变世界的 APP！

作者简介

Bob McCune是一名来自明尼苏达州的iOS开发者和培训师。他于2007年开始从事Mac开发，在2008年iPhone SDK首次发布之后转而从事iOS开发。他是TapHarmonic有限公司的所有者，这是一家位于明尼苏达州的小型iOS咨询和培训公司。Bob于2008年春成立了MN chapter of CocoaHeads，至今一直担任该团队的领导。Bob和妻子Linda养育着两个可爱的孩子。拥有这样一个充满爱和支持的家庭是多么幸福啊！

致　　谢

在这里感谢父母给予我的全部爱和支持。我现在所拥有的一切都源自于他们之前所带给我的。还要感谢爱妻Linda和我可爱的孩子们Michael和Kayla，如果不是他们的耐心和支持，我无法完成本书的撰写。谢谢Pearson Education的Trina MacDonald让本书顺利出版以及在整个过程中对我的指导和帮助。谢谢Chris Zahn、Olivia Basegio、Elaine Wiley和所有参与本书出版的Pearson公司的朋友们。尤其要感谢技术编辑Chris Adamson、JonSteinmetz和Ryder Mackay，我非常感谢你们的修订和反馈，尤其是Chris经常在Comic Sans中给我写一些评论，让我能快速地对本书进行修订。最后感谢苹果公司和它神奇的Mac、iOS开发者社区，我从事软件开发工作已近20年，从未像今天这般开心。

序　言

我们知道 QuickTime 要完蛋了！

QuickTime 已经为我们忠诚地服务了 20 年，不过这也正是问题所在。苹果公司的核心媒体框架于 20 世纪 80 年代末期推出，在 21 世纪中期已经汇集了大量烦琐的功能：比如过时的程序开发惯例，基于系统的 API 已经不再适用和流行，并且一些特性也经不住时间的考验了(除了 WiredSprites 互动格式的经久不衰，似乎再没有了)。哎，同时它还喜欢使用 big-endian 数值，因为这是 Motorola 68000 系列 CPU 所使用的格式。是的，QuickTime 已经经历了两代 CPU 的变迁：从 680x0 到 PowerPC，再到 Intel x86。

21 世纪第一个十年，QuickTime 的变革很难理解，苹果公司创建了一个有关 QuickTime 的 Java 封装，之后对它进行了更新并删除了一半的功能；在 Objective-C 方面同样是功能不完整的，称之为 QTKit。并且之后出现的 QuickTime 的 Windows 版本居然停止了所有有意义的更新，苹果公司也没有人告诉大家这是为什么。

按照苹果公司一贯的做事风格，这意味着他们正在做着一些其他事情。

当然他们正在做的就是 iPhone。不过为 iPhone 推出的第一个 SDK 包含了很少有关媒体方面的支持，仅有的是支持应用程序全屏播放视频和一些底层的 Core Audio 库。显而易见的是，苹果公司声称自己造出了最好的 iPod，其实在媒体软件方面还存在很大的缺陷。那时候我们知道程序还无法从 iPhone 导出 QuickTime 文件，不过在桌面应用上已经对此有所突破了。

是的，他们是在做一些事。

iPhone 上的一些新媒体功能在接下来几年中层出不穷：媒体播放器框架让我们可以查询和播放音乐库中的歌曲，使用 Objective-C 语言封装了 Core Audio 的 Audio Queue 队列，从文件中读取内容和录制内容保存到文件不再是 200 行代码这样繁重的工作了。这些最新的类都是由一个新框架给出的，它就是 AV Foundation，这个名字在 iPhone OS 3 时代看起来像是错误的写法，那时候都是字母 A 而没有字母 V。

回顾过去，我们真的发现苹果公司正在做着一些事情。

到了 2010 年，苹果公司终于向人们展示这些年都在做什么。苹果公司的 Meriko Borogove

站在 WWDC “Graphics State of the Union” 的舞台上展示 iPhone 版本的 iMovie 应用，并宣称之前视频编辑所提供的功能目前在 iOS 上全部支持。AV Foundation 之前包含的那些奇怪的 Core Audio 封装类，现在变成了 40 个音频和视频处理类。采集、编辑、播放和导出这些我们在实际使用 QuickTime 时最常用的功能(不好意思，还有 WiredSprites 互动格式)都已经在 iOS 平台上实现了。

现在我们已经可以将这些功能装进口袋和钱包了。

摆脱了 20 世纪 90 年代固有的观念后，新的类使用了一种真正现代的思维。在 iOS 中，首先利用 Objective-C 代码块来处理异步事件，比如长时间的媒体导出操作，现在这已经是 iOS 开发者习惯处理的事情了。回到 Mac 平台，有很多 iOS 平台所羡慕的功能，那时能想到的都是一些带有过时字节和已经删除的 QTKit 功能的仅支持 32 位的 QuickTime。所以当 AV Foundation 第一次出现在 Mac 平台的 OS X 10.7 (Lion)系统中，在 OS X 10.9 (Mavericks)系统中 QuickTime 正式被 AV Foundation 弃用时，没有引起特别的轰动。

不要认为故事中都是彩虹和可爱的小狗，媒体开发仍然是一项复杂的工作。我们需要处理大量数据，操作针对实时处理的极小时间窗口，以及用户对我们工作的极高期盼。

还需要理解很多有关素材的知识：声学和视觉相关的科学理论，熟练的程序开发技术和由 AV Foundation 而引出的其他框架的知识，比如 Core Media、Core Video、Core Image、Core Audio、Media Player (iOS)和 Video Toolbox (OS X)，等等。并不总能通过类名的含义凭直觉就知道 API 的用法，互换诸如 AV、Composition、Instruction、Video、Layer 和 Mutable 这些术语创建了至少一打实际类名(应该给能找到最多的人颁个奖)。AVMutableComposition 和 AVMutableVideoComposition 二者之间根本没有正式的关联，这也是难以想象的。

除了上述困难之外，精通苹果框架的挑战还在于：未说明的假设，对于其他框架和库函数的粗略说明，和三年前的 WWDC 会话中示例代码的引用不再有效。也不要想把所有的内容都打上书签，苹果公司每年至少要重组一次开发者网站，会打断所有的外部链接。

诚然，只要 AV Foundation 还是一个公开的 API，我们就确实需要一本合适的参考书。

你即将阅读的这本书经历了反复的试验，深入挖掘了参考文档、头文件、各种论坛的知识碰撞，并反复研究编译器、模拟器和设备直到成功。它对出自不同地方的知识进行了整合，将许多开发者的专长和经验聚到一起，集思广益。几年前就开始研究 AV Foundation 的我们这一批人，推进这些简单的例子并找到框架究竟能做什么，很高兴看到 Bob McCune 接过了这个火炬，为 AV Foundation 开发者指明了前进的方向，让他们更好地掌握这一复杂的框架。Bob 研究这一领域已有一年时间了，当查找论坛的帖子和博客时也和我们那时一样满腹牢骚。

在 Twitter 上，Bob 和我开玩笑说道，我们来来回回的一段会话会产生双倍的 Google 点击量，比如术语 AVVideoCompositionLayerInstruction 就这么长的名字，不过像这种冗长的类名 Twitter 居然还真支持。我们开玩笑说散列标签#avfoundation 所带来的有用信息都和荒谬的散列标签#sidewaysbondagecake 一样多了。我们需要更多的信息让人们看见，Bob 在这方面做得非常棒。

很高兴看到这么长的一个项目终于要结束了。随着越来越多的开发者进入 AV Foundation

领域，我们将看到，接下来几年，新一轮的音频视频应用程序将层出不穷。在 20 世纪 90 年代时，我们在一台全是斑点的设备上观看邮票大小的视频，而现在我们在 iPhone 上欣赏 HD 视频并可以将它通过 AirPlay 立即发送到 50 寸彩电上显示。

现在正是加入 AV Foundation 开发者行列的黄金时代，相信你在学习完本书之后能够开发出精彩的应用程序。

——Chris Adamson

注：Chris Adamson 是 *Learning Core Audio* 和 *QuickTime for Java: A Developer's Notebook* 的作者。

前　言

我们非常激动地看到：在过去几年中，数字媒体经历了革命性的变化。iPhone的推出和移动计算的兴起，以及高速网络的发展，从根本上改变了人们创建、使用和分享数字媒体的习惯。人们不再限于在家里观看视频。今天的视频焕发活力，任何人都可以随时随地欣赏它们。捕捉高分辨、有个性的照片不再是有着昂贵摄影设备的专业摄影师的专利，如今只需要在iOS设备上点点手指即可实现。电影制作者和音乐家们以前只能在专业工作室完成理想的作品，而如今只需一台笔记本电脑或一台移动设备即可。数字媒体革命正在发生，不过这一过程的确刚刚开始，对于iOS和OS X来说，这一革命的核心技术就是AV Foundation。

有机会撰写本书令我备感荣幸，因为我认为这是一本姗姗来迟的专业书。App Store中很多排名靠前的应用程序都用到了AV Foundation，不过这一框架对于大部分开发者而言并不很好掌握。学习AV Foundation框架的使用具有一定的挑战性。它是一个庞大的高级框架，具有很多特性和功能。*AV Foundation Programming Guide*文档虽然在过去一年中进行了改进，但仍不够完善，仅仅介绍了有关AV Foundation的一些浅显概念。苹果公司在ADC上给出了大量有用的示例项目，不过对于初学者来说，就像把一个还没学会游泳的人推入深水池一样。

我撰写这本书的目标就是让AV Foundation框架更容易理解。本书并不是要成为AV Foundation框架领域的权威参考，而是把注意力放在框架最关键的几个部分，让开发者能够完全掌握该框架的概念、功能和习惯用法。我们通过紧贴实用的示例应用程序一步一步完成讲述，从最简单的语音备忘应用程序到诸如iOS平台的iMovie的功能齐全的视频编辑器。对我来说读者朋友们能够牢牢掌握这些概念非常重要，并让大家掌握如何使用AV Foundation的功能在实际应用程序中加以应用。

其实在早些年我就希望推出一本AV Foundation开发秘籍这样的图书，并希望能够帮助大家在iOS和OS X平台上开发出优异的媒体应用程序！

本书读者对象

本书面向需要学习如何开发数字媒体应用程序的Mac或iOS开发者。我们假设大家可能没有AV Foundation或媒体应用程序的开发经验，不过认为大家都具有苹果公司平台下的框架、模式和概念的开发经验。确切地讲，你需要熟悉以下知识：

C和Objective-C：本框架依赖于很多高级语言和Cocoa特性，比如Grand Central Dispatch (GCD)、Blocks和Key-Value Observing相关知识。你不需要是GCD专家，不过应该了解调度语法和调度队列的基础知识。AV Foundation是一个Objective-C框架，不过我们经常会用到框架支持的C语言库，尤其在一些高级场景中，所以你也应该具备一些C语言的基础知识。

Core Animation(可选)：AV Foundation总的来说是一个不可见框架，不过在渲染视频内容时会用到一些Core Animation的知识。掌握Core Animation框架的知识虽然不是必需的，但很有帮助。

绘制框架(可选)：高级用例中经常会用到绘制和渲染框架，如Quartz、Core Image、Open GL或OpenGL ES。本书对如何整合这些技术进行讲解，但并未假设大家知道这些框架的用法。

本书组织结构

AV Foundation是一个包含了许多功能和特性的大框架。根据相关的功能把框架进行分解，本书共分3个主要部分："AV Foundation基础"、"媒体捕捉"以及"媒体创建和编辑"。第Ⅰ部分"AV Foundation基础"介绍了框架的基础知识和对于大部分AV Foundation应用程序都适用的知识点。第Ⅱ部分"媒体捕捉"主要介绍有关捕捉的API，并用学到的方法创建静态图片和视频的捕捉应用程序。最后一部分"媒体创建和编辑"更深入地讨论AV Foundation在创建和编辑媒体资源时的功能和特性。

本书各章综述如下：

第1章：AV Foundation入门——该章是学习AV Foundation的第一步，对框架进行了解析，帮助读者了解相关的功能和特性。该章还对媒体相关知识给出了高级概述，并介绍诸如数字采样和媒体压缩等方面的知识，掌握这些知识对本书的学习很有帮助。

第2章：播放和录制音频——AV Foundation关于播放和录制音频的功能是该框架中最常用的功能。本章将讨论如何使用框架的音频类，并使用这些类创建音频循环器和语音备忘录。同时介绍如何使用音频会话来帮助开发者优化应用程序的音频使用体验。

第3章：资源和元数据——该框架中大部分对象的创建都是围绕资源的概念展开的，资源代表了媒体资源，比如QuickTime影片或MP3音频文件。你可以学到如何使用资源对象，并通过构建元数据编辑应用程序来了解如何使用该框架的元数据特性。

第4章：视频播放——播放视频是AV Foundation实现的最核心功能之一。在很多媒

体应用程序中，这一功能都是最主要或次要的功能。在该章你会详细了解如何使用框架的播放功能来创建一个自定义的视频播放器，该视频播放器带有全部播放控件、字幕并支持Airplay。

第5章：AV Kit用法——AV Kit是在Mac OS X 10.9和现在的iOS 8版本中引入的一个新框架。它可以让你快速创建与OS X中的QuickTime Player和iOS中的视频应用程序具有相同用户界面的视频播放器。如果你希望创建一个忠于原始操作系统，同时又具有AV Foundation视频API全部功能的播放器，那么这个框架是一个很好的选择。

第6章：捕捉媒体——该章介绍了框架的音频和视频捕捉功能。学习使用这些功能来控制iOS设备和最新Mac设备内置的摄像头硬件。这是该框架应用最广的一个领域，可以帮助开发者创建出功能强大的主流摄像头捕捉应用程序。

第7章：高级捕捉功能——该章介绍了大量高级捕捉相关的知识。学习使用元数据捕捉来实现条码扫描和人脸检测功能。学习使用框架提供的高级缩放功能。同时还学习如何支持高帧率捕捉，为应用程序添加慢动作效果。学习如何整合OpenGL ES来处理摄像头捕捉到的视频样本，这一方面又包含了大量可能的处理方法。

第8章：读取和写入媒体——AV Foundation提供了很多高级功能，不过从来没有关闭底层开发的大门。该章会讨论该框架的底层读写工具，使你对媒体资源的处理更加随心所欲。学习如何从资源对象中读取音频样本并实现音频示波器的展示。同时还使用摄像头捕捉API来实现实时视频效果。

第9章：媒体资源的组合和编辑——从该章开始我们将学习该框架有关媒体编辑的功能。这是该框架非常强大的功能之一，可对多个来源的媒体进行组合和编辑来创建新媒体。这里开始创建一个本书最复杂的应用程序——15 Seconds，它是一个与iOS下的iMovie类似的视频编辑器。

第10章：混合音频——创建组合媒体对象时，最重要的一点是对多个音频轨道的混合。该章将介绍如何使用音频渐隐等混合技术帮助我们优化音频的呈现。

第11章：创建视频过渡效果——视频过渡在表示视频中的位置或故事情节发生变化时经常用到，AV Foundation为组合资源对象提供了应用视频过渡效果的可靠支持。在该章你会学到使用该框架的视频组合功能来控制多个视频轨道的组合。在15 Seconds应用程序中实际使用这些功能，为它添加溶解、推出和擦除等过渡效果。

第12章：动画图层内容——该章讨论如何使用Core Animation框架添加标题、下沿字幕和其他动画叠加效果。学习如何使用Core Animation创建与视频播放做到无缝同步的动画序列。还讨论如何将这些动画效果整合到最终导出的视频中。

关于示例代码

我们将大量时间都花在开发本书的示例应用程序上，学习AV Foundation的一个主要目标是在实际应用程序的创建中用到各种特性和功能。为达到这一目的，在本书整个讲

述过程中包含了大量真实的示例项目。这些项目可以作为参考，或者甚至可以对它们进行自定义作为你自己应用程序的基础。一些项目很简单(如Hello AVF)，一些项目很复杂(如15 Seconds)，但每个项目都展示了一个或多个框架功能，并且创建项目的过程很有趣。

AV Foundation在OS X和iOS平台上的应用基本相同，所有示例项目虽然以一个平台为例进行讲解，不过都适用于两个平台。示例应用程序已经创建好了用户界面和支持代码，并且代码部分也仅仅关注有关AV Foundation内容的实现。这样便于读者更好地阅读应用程序代码而不需要考虑平台的区别，从OO设计的角度来讲述，使代码具有很好的可重用性并且便于测试。

示例项目可以在以下Github网站找到：https://github.com/tapharmonic/Learning-AV-Foundation。

还可以访问www.tupwk.com.cn/downpage，输入本书的书名或中文ISBN，下载部分资料。

联系作者

可以通过个人主页http://bobmccune.com联系Bob，也可在Twitter上(@bobmccune)找到他。

目　录

第 I 部分　AV Foundation 基础

第 1 章　AV Foundation 入门 …… 3
1.1　AV Foundation 的含义 …… 3
1.2　AV Foundation 的适用范围 …… 4
1.3　解析 AV Foundation …… 5
1.3.1　音频播放和记录 …… 5
1.3.2　媒体文件检查 …… 6
1.3.3　视频播放 …… 6
1.3.4　媒体捕捉 …… 6
1.3.5　媒体编辑 …… 6
1.3.6　媒体处理 …… 6
1.4　了解数字媒体 …… 6
1.4.1　数字媒体采样 …… 7
1.4.2　音频采样介绍 …… 7
1.5　数字媒体压缩 …… 11
1.5.1　色彩二次抽样 …… 11
1.5.2　编解码器压缩 …… 13
1.5.3　视频编解码器 …… 13
1.5.4　音频编解码器 …… 15
1.6　容器格式 …… 16
1.7　初识 AV Foundation …… 16
1.8　小结 …… 20
1.9　挑战 …… 20

第 2 章　播放和录制音频 …… 21
2.1　Mac 和 iOS 的音频环境 …… 21
2.2　理解音频会话 …… 22
2.2.1　音频会话分类 …… 22
2.2.2　配置音频会话 …… 23
2.3　使用 AVAudioPlayer 播放音频 …… 23
2.3.1　创建 AVAudioPlayer …… 23
2.3.2　对播放进行控制 …… 24
2.4　创建 Audio Looper …… 25
2.5　配置音频会话 …… 29
2.6　处理中断事件 …… 30
2.7　对线路改变的响应 …… 33
2.8　使用 AVAudioRecorder 录制音频 …… 35
2.8.1　创建 AVAudioRecorder …… 36
2.8.2　控制录音过程 …… 37
2.9　创建 Voice Memo 应用程序 …… 38
2.9.1　配置音频会话 …… 38
2.9.2　实现录音功能 …… 39
2.10　使用 Audio Metering …… 44
2.11　小结 …… 48

第 3 章　资源和元数据 …… 49
3.1　理解资源的含义 …… 49
3.2　创建资源 …… 50
3.2.1　iOS Assets 库 …… 50
3.2.2　iOS iPod 库 …… 51
3.2.3　Mac iTunes 库 …… 52
3.3　异步载入 …… 52
3.4　媒体元数据 …… 54
3.5　使用元数据 …… 58
3.5.1　查找元数据 …… 59
3.5.2　使用 AVMetadataItem …… 60
3.6　创建 MetaManager 应用程序 …… 63
3.6.1　THMediaItem …… 63
3.6.2　THMetadata 的实现 …… 67
3.6.3　数据转换器 …… 69
3.6.4　简单转换 …… 70
3.6.5　转换 Artwork …… 71
3.6.6　转换注释 …… 72
3.6.7　转换音轨数据 …… 73
3.6.8　转换唱片数据 …… 75
3.6.9　转换风格数据 …… 76
3.6.10　完成 THMetadata …… 79
3.7　保存元数据 …… 81

3.8 小结 ····· 82
3.9 挑战 ····· 83

第 4 章 视频播放 ····· 85
4.1 播放功能综述 ····· 85
4.1.1 AVPlayer ····· 86
4.1.2 AVPlayerLayer ····· 86
4.1.3 AVPlayerItem ····· 88
4.2 播放秘籍 ····· 88
4.3 处理时间 ····· 90
4.4 创建视频播放器 ····· 91
4.4.1 创建视频视图 ····· 91
4.4.2 创建视频控制器 ····· 93
4.4.3 监听状态改变 ····· 96
4.5 时间监听 ····· 97
4.5.1 定期监听 ····· 98
4.5.2 边界时间监听 ····· 98
4.5.3 条目结束监听 ····· 99
4.5.4 播放栏委托回调 ····· 100
4.6 创建可视搓擦条 ····· 102
4.7 显示字幕 ····· 106
4.8 Airplay ····· 110
4.9 小结 ····· 111
4.10 挑战 ····· 112

第 5 章 AV Kit 用法 ····· 113
5.1 针对 iOS 平台的 AV Kit 框架 ····· 113
5.2 针对 Mac OS X 平台的 AV Kit 框架 ····· 115
5.3 迈出第一步 ····· 115
5.4 控件类型 ····· 119
5.4.1 内嵌类型 ····· 119
5.4.2 浮动类型 ····· 120
5.4.3 最小化类型 ····· 120
5.4.4 None 类型 ····· 121
5.5 拓展学习 ····· 121
5.6 章的处理 ····· 124
5.7 启用修剪 ····· 129
5.8 导出 ····· 130
5.9 传统资源的兼容 ····· 132
5.10 小结 ····· 135
5.11 挑战 ····· 136

第Ⅱ部分 媒体捕捉

第 6 章 捕捉媒体 ····· 139
6.1 捕捉功能综述 ····· 139
6.1.1 捕捉会话 ····· 140
6.1.2 捕捉设备 ····· 140
6.1.3 捕捉设备的输入 ····· 140
6.1.4 捕捉的输出 ····· 141
6.1.5 捕捉连接 ····· 141
6.1.6 捕捉预览 ····· 141
6.2 简单的秘籍 ····· 143
6.3 创建相机应用程序 ····· 144
6.3.1 创建预览视图 ····· 145
6.3.2 创建捕捉控制器 ····· 148
6.3.3 设置捕捉会话 ····· 149
6.3.4 启动和停止会话 ····· 151
6.3.5 处理隐私需求 ····· 152
6.3.6 切换摄像头 ····· 153
6.3.7 配置捕捉设备 ····· 155
6.3.8 调整焦距和曝光 ····· 156
6.3.9 调整闪光灯和手电筒模式 ····· 160
6.3.10 拍摄静态图片 ····· 161
6.3.11 使用 Assets Library 框架 ····· 163
6.3.12 视频捕捉 ····· 165
6.4 小结 ····· 170
6.5 挑战 ····· 170

第 7 章 高级捕捉功能 ····· 171
7.1 视频缩放 ····· 171
7.2 人脸检测 ····· 176
7.3 机器可读代码识别 ····· 186
7.3.1 一维条码 ····· 186
7.3.2 创建条码扫描器 ····· 188
7.3.3 创建条码识别的委托 ····· 191
7.4 使用高帧率捕捉 ····· 196
7.4.1 高帧率捕捉综述 ····· 197
7.4.2 支持高帧率捕捉 ····· 197
7.5 视频处理 ····· 201
7.6 学习 CMSampleBuffer ····· 202

7.6.1 样本数据 ····· 202
7.6.2 格式描述 ····· 203
7.6.3 时间信息 ····· 204
7.6.4 附加的元数据 ····· 204
7.6.5 AVCaptureVideoData-Output ····· 205
7.7 小结 ····· 209
7.8 挑战 ····· 209

第 8 章 读取和写入媒体 ····· 211
8.1 综述 ····· 211
8.1.1 AVAssetReader ····· 212
8.1.2 AVAssetWriter ····· 212
8.1.3 读写示例 ····· 213
8.2 创建音频波形视图 ····· 215
8.2.1 读取音频样本 ····· 216
8.2.2 缩减音频样本 ····· 220
8.2.3 渲染音频样本 ····· 222
8.3 捕捉录制的高级方法 ····· 224
8.4 小结 ····· 237
8.5 挑战 ····· 238

第Ⅲ部分 媒体创建和编辑

第 9 章 媒体的组合和编辑 ····· 241
9.1 组合媒体 ····· 241
9.2 时间的处理 ····· 243
9.2.1 CMTime ····· 243
9.2.2 CMTimeRange ····· 245
9.3 基础方法 ····· 246
9.4 15 Seconds 示例应用程序 ····· 249
9.5 创建一个组合 ····· 251
9.6 导出组合 ····· 256
9.7 小结 ····· 260
9.8 挑战 ····· 260

第 10 章 混合音频 ····· 261
10.1 混合音频 ····· 261
10.2 15 Seconds 应用程序中的音频混合 ····· 264
10.3 小结 ····· 269
10.4 挑战 ····· 269

第 11 章 创建视频过渡效果 ····· 271
11.1 综述 ····· 271
11.1.1 AVVideoComposition ····· 272
11.1.2 AVVideoComposition-Instruction ····· 272
11.1.3 AVVideoCompositionLayer-Instruction ····· 272
11.2 概念阶段 ····· 272
11.2.1 部署视频布局 ····· 273
11.2.2 定义重叠区域 ····· 274
11.2.3 计算通过和过渡的时间范围 ····· 276
11.2.4 创建组合和层指令 ····· 278
11.2.5 创建和配置 AVVideo-Composition ····· 279
11.2.6 创建视频组合的捷径 ····· 280
11.3 为 15 Seconds 应用程序添加视频过渡 ····· 281
11.4 小结 ····· 290
11.5 挑战 ····· 290

第 12 章 动画图层内容 ····· 291
12.1 Core Animation 应用 ····· 291
12.2 在 AV Foundation 中使用 Core Animation ····· 293
12.2.1 使用 AVSynchronizedLayer 播放 ····· 294
12.2.2 使用 AVVideoComposition-CoreAnimationTool 导出 ····· 295
12.3 15 Seconds 应用程序：添加动画标题 ····· 296
12.4 准备组合 ····· 304
12.4.1 播放时应用 Core Animation ····· 305
12.4.2 导出时应用 Core Animation ····· 306
12.5 小结 ····· 307
12.6 挑战 ····· 308

第 I 部分

AV Foundation 基础

第 1 章　AV Foundation 入门

第 2 章　播放和录制音频

第 3 章　资源和元数据

第 4 章　视频播放

第 5 章　AV Kit 用法

第1章

AV Foundation 入门

苹果公司一直引领着全球数字媒体的前进潮流，1991年苹果公司推出的QuickTime首次将数字音频和数字视频展现在用户面前。QuickTime架构在之后20年间给数字多媒体这一领域带来了变革，对教育领域、游戏行业和娱乐产业的发展影响巨大。2001年苹果公司在全球范围内推出了iTunes和iPod，从根本上改变了用户收听音乐的方式。两年后推出的iTunes Store，彻底颠覆了传统音乐产业，并从此成为苹果公司不断完善的数字媒体生态环境中的核心环节。2007年苹果公司推出了iPhone手机，并在几年后发布了iPad平板电脑。这一系列事件开辟了一个全新的计算时代，并永远改变了我们创建、使用和分享媒体信息的方式。

数字媒体的世界已经不仅是技术控们的话题了。如今的数字媒体更加简单，成为人们的必需品，无处不在且功能强大。诸如Instagram的app可以让用户很容易地拍摄出漂亮且经过艺术修饰的图片，并将其分享到全世界。从Skype到TangoMe这样的视频聊天应用程序可以让世界各地的朋友在一起聊天。由YouTube和Netflix提供的流视频也不再局限于仅在LTE或Wi-Fi环境下才能使用了。另外，像Final Cut Pro X和iMovie这类基于iPad的工具可以让初学者像高手一样轻松编辑视频文件。

数字媒体革命正在发生，而我们正在见证这一时代的到来！掌握AV Foundation是基于Mac OS X和iOS开发下一代媒体应用程序的关键，本书将为你提供悉心的指导。首先会对该框架进行整体介绍，让你深刻理解精通该框架所需要学习的内容。那么下面开始吧！

1.1 AV Foundation 的含义

AV Foundation是苹果OS X系统和iOS系统中用于处理基于时间的媒体数据的高级Objective-C框架。通过开发所需的工具提供了大量强大的功能集，让开发者能够基于苹果平台创建当下最先进的媒体应用程序。AV Foundation的构建考虑到了目前的硬件环境和应用程序，其设计过程高度依赖多线程机制。充分利用了多核硬件的优势并大量使用block和Grand Central Dispatch(GCD)机制将复杂的计算进程放在后台线程运行。会自动提供硬件加速操作，确保在大部分设备上应用程序能以最佳性能运行。AV Foundation的设计也充分考虑了电量效率来满足诸如iPhone和iPad这类移动设备对电量控制的高要求。此外，从一开始该框架就是针对64位处理器设计的，可以发挥64位处理器的所有优势。

1.2 AV Foundation 的适用范围

学习AV Foundation的首要步骤之一就是要知道它在苹果体系所有的媒体资源分类图中的适用范围。Mac OS X和iOS为开发者提供了一系列高层级和低层级的框架来实现时基媒体(timed media)的各功能。图1-1展示了AV Foundation在整个体系中所处的角色。

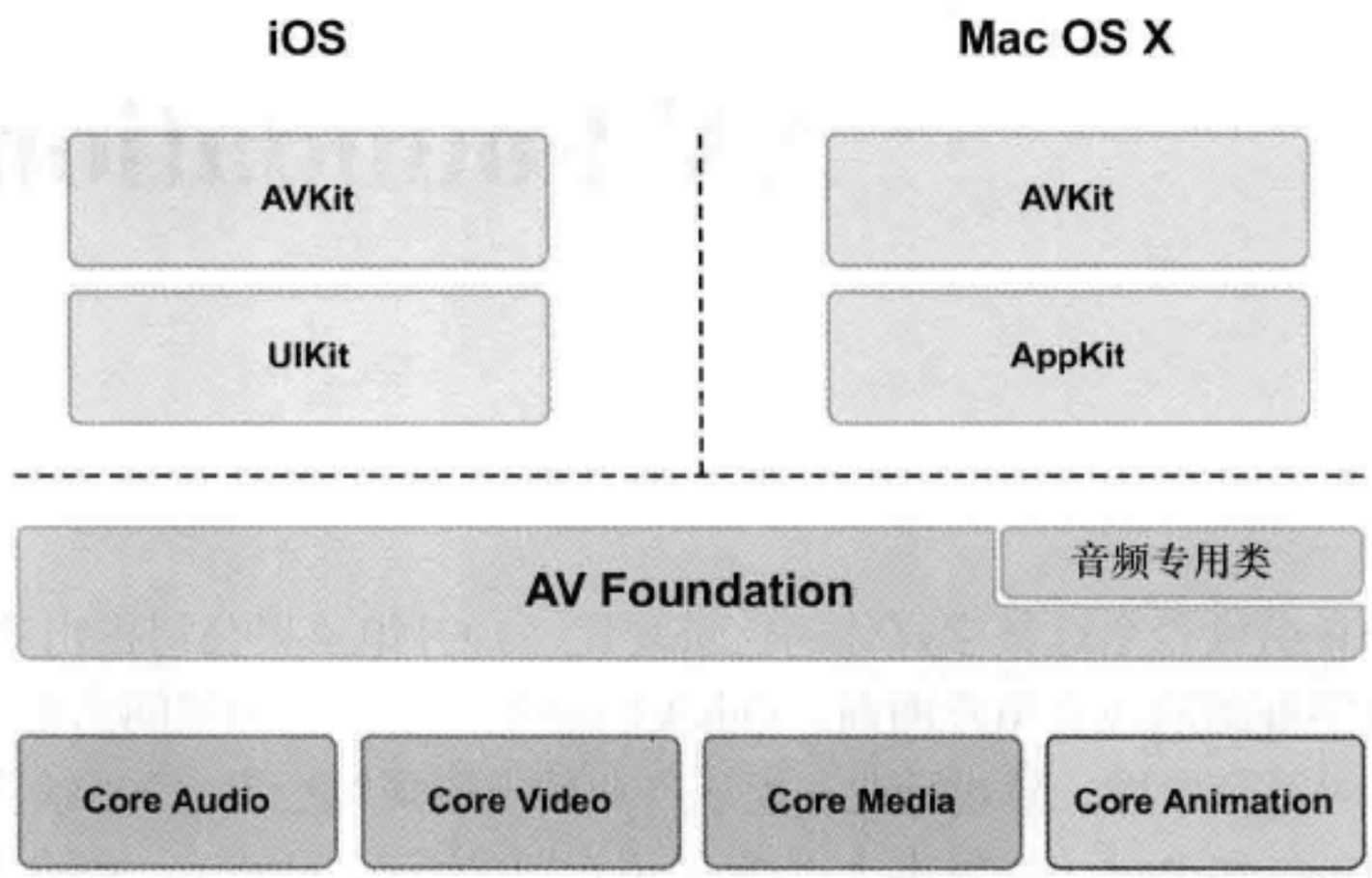

图 1-1 Mac OS X 和 iOS 媒体环境

两个平台都针对处理媒体相关操作提供了一系列高层级的解决方案。在iOS中，利用UIKit框架可以很容易地将基本的静态图片和视频捕捉整合到你的应用程序中。Mac OS X和iOS两个系统都可以通过在WebView或UIWebView中添加<audio>和<video>HTML5标签来播放音频和视频内容。此外，两个系统都还提供了AVKit框架，用于简化目前流行的视频播放应用程序的创建过程。所有这些方法使用起来都很方便并相对简单，当你需要向应用程序加入媒体相关功能时应该考虑这些方法。虽然这些方法都很方便，但不够灵活，并且经常需要更高级的应用程序来对其进行控制。

在这些适用范围的另一端同样给出了一些低层级的框架，它们支持所有高层级方法所使用的功能。大部分这些框架都是低层级、基于C应用程序框架且功能强大和高性能的。但是这些框架往往学习起来比较复杂并需要对硬件级媒体处理有非常深入的理解才行。下面来了解几个最主要的支撑框架及其提供的功能。

Core Audio

Core Audio是OS X和iOS系统上处理所有音频事件的框架。Core Audio是由多个框架整合在一起的总称，为音频和MIDI内容的录制、播放和处理提供相应接口。Core Audio也提供高层级的接口，比如通过Audio Queue Services框架所提供的那些接口，主要处理基本的音频播放和录音相关功能。同时还会提供相对低层级的接口，尤其是Audio Units接口，它们提供了针对音频信号进行完全控制的功能，并通过Audio Units让你能够构建一些复杂的音频处理模式，就像通过苹果公司的Logic Pro X和Avid's Pro Tools工具所实现的功能一样。要详细了解这一框架，强烈建议你阅读由Chris Adamson和Kevin Avila共同撰写的*Learning Core Audio*一书。

Core Video

Core Video是OS X和iOS系统上针对数字视频所提供的管道模式。Core Video为其相对的Core Media提供图片缓存和缓存池支持，提供了一个能够对数字视频逐帧访问的接口。该框架通过像素格式之间的转换并管理视频同步事项使得复杂的工作得到了有效简化。

Core Media

Core Media是AV Foundation所用到的低层级媒体管道的一部分。它提供针对音频样本和视频帧处理所需的低层级数据类型和接口。Core Media还提供了AV Foundation用到的基于CMTime数据类型的时基模型。CMTime及其相关数据类型一般在AV Foundation处理基于时间的操作时使用。

Core Animation

Core Animation是OS X和iOS提供的合成及动画相关框架。主要功能就是提供苹果平台所具有的美观、流畅的动画效果。提供了一个简单、声明性的编程模式，并已经封装了支持OpenGL和OpenGL ES功能的基于Objective-C的各种类。使用Core Animation时，对于视频内容的播放和视频捕获这两个动作，AV Foundation提供了硬件加速机制来对整个流程进行优化。AV Foundation还可以利用Core Animation让开发者能够在视频编辑和播放过程中添加动画标题和图片效果。

处于高层级框架和低层级框架之间的就是AV Foundation。在整体媒体分类图中AV Foundation的地位举足轻重，其提供了很多低层级框架才能实现的功能和性能，并且是以更简单的Objective-C接口方式实现的。同时它也可以和高层级的框架无缝衔接，比如Media Player和Assets Library，使用这两个框架所提供的服务并在需要更高级的功能时，也可以直接与Core Media和Core Audio进行交互。此外，由于AV Foundation处于UIKit和AppKit层之下，这也意味着你可以在两个平台下使用这个媒体框架。你只需在一个系统下学习这个框架即可，之后就可以有机会将代码和所学到的知识和经验用于另一个平台。

1.3　解析 AV Foundation

在学习AV Foundation之初，最大的挑战就是理解该框架所提供的大量类及其功能。该框架包含的类超过100个、大量协议的集合及许多不同的功能和常量。在第一次遇到这些问题时看似掌握这个框架有一定的难度，但是如果你将这个框架仔细地按功能单元进行分解，就会变得比较容易理解。下面就看一下AV Foundation所提供的一些核心功能。

1.3.1　音频播放和记录

如果回顾一下图1-1，会发现AV Foundation方框的右上角有一个小方格被单独标记为“音频专用类”。这是由AV Foundation提供的关于音频处理的一些早期功能。AVAudioPlayer和AVAudioRecorder可以在应用程序中提供一种更简单的整合音频播放和记录的功能。这些方法并不是AV Foundation用于播放和记录音频的唯一方式，却是学习起来最简单并且功能最强大的方法。

1.3.2 媒体文件检查

AV Foundation提供检查正在使用的媒体文件的功能。可以查看这些媒体资源来确定是否适合一些特定的任务，比如是否可以用于回放或其是否可以被编辑和导出。还可以获取该媒体资源相关的技术参数，比如内容持续时间、创建日期或首选播放音量等。此外，该框架还基于AVMetadataItem类提供功能强大的元数据支持。这就允许开发者读写关于媒体资源的描述信息，比如唱片簿和艺术家信息。

1.3.3 视频播放

AV Foundation提供的其中一个最常用的功能是视频播放功能，这个功能通常是很多媒体应用程序最主要或次要的功能。AV Foundation框架可以让你播放从本地文件或远程流中获取的视频资源，并对视频播放和内容的展示进行控制。这一部分的核心类是AVPlayer和AVPlayerItem，正是这两个类让你能够对资源的播放进行控制，此外它还可以整合其他更高级的功能，如控制子标题和章节信息等。或者让你接连访问音频和视频这两个不同的资源。

1.3.4 媒体捕捉

最近推出的所有Mac和iOS设备都内置了摄像头，作为捕捉静态图片和视频的高性能外设。AV Foundation提供了一个丰富的API集来让你可以对这些设备进行精密控制。摄像头捕捉的核心类是AVCaptureSession，其作为所有活动的汇集点来接收摄像头设备由各路流发过来的电影和图片。这也是AV Foundation框架的一个可靠功能，所以其可以根据近期发布的框架版本进行大幅改进。

1.3.5 媒体编辑

AV Foundation同样对媒体资源的整合和编辑提供了强有力的支持。该框架允许创建可以将多个音频和视频资源进行组合的应用程序，允许修改和编辑独立的媒体片段、随时修改音频文件的参数以及添加动画标题和场景切换效果。 Mac和iPad上的一些工具，比如Final Cut Pro X和iMovie，都是使用这个功能创建的最好示例应用程序。

1.3.6 媒体处理

虽然通过AV Foundation框架可以在不通过对媒体资源字节一级进行操作的前提下实现大量功能，但有时你可能还是需要访问更底层的数据并对其进行操作。幸运的是，当需要执行更高级的媒体处理任务时，可以使用AVAssetReader和AVAssetWriter类来实现这些功能。这些类提供直接访问视频帧和音频样本的功能，所以可以对媒体资源进行任何更高级的处理。

1.4 了解数字媒体

目前数字媒体已经融入大家的生活，我们在iTunes中购买音乐和唱片，从Netflix和Hulu上购买流媒体电影和TV节目，通过email和网站来分享数字照片。使用数字媒体已经成为广

大用户一种本能的习惯了，但是读者朋友们是否想过这些媒体内容最初是如何被数字化的呢？我们显然已经处在一个数字化的时代，但是我们还是更习惯模拟信息的世界。我们看到的信号标志和所听到的音乐都是通过模拟信号传递给我们的，我们的眼睛和耳朵的构造将这些信息转换为我们大脑能够解析出的电信号。现实生活中的信号是连续的，信号的频率和强度是在不断变化的；但是数字世界的信号是离散的，由1和0两个状态表示。要将模拟信号转换成我们能够存储并传输的数字信号，要经过模拟-数字转换过程，我们将这个过程称为采样(Sampling)。

1.4.1　数字媒体采样

对媒体内容进行数字化主要有两种方式。第一种称为时间采样，这种方法捕捉一个信号周期内的变化。比如当你在iPhone上记录一个音频备忘录时，在录制期间你所有的音高变化和声调变化都会被捕捉下来。第二种采样方式是空间采样，一般用在图片数字化和其他可视化媒体内容数字化的过程中。空间采样包含对一幅图片在一定分辨率之下捕捉其亮度和色度，进而创建由该图片的像素点数据所构成的数字化结果。当对一段视频进行数字化时，这两种方式都可以使用，因为通常的视频信号既有空间属性又有时间属性。

幸运的是，开发者不需要对这两个采样过程中所用到的具体数字化信号处理原理有很深的研究，因为设备硬件可以帮助我们完成模拟信号到数字信号的转换。但是如果不了解这些采样过程的基本原理及之后数字媒体内容的存储格式的话，在你进一步深入学习AV Foundation更高级、更有趣功能的时候会感到比较吃力。所以我们还需要了解采样过程的相关知识，下面以音频采样为例逐步对其进行研究。

1.4.2　音频采样介绍

当你听到一个人的声音、号角声或吉他的弹奏声时，你真正听到的是声波通过一定介质传播过来的振动。比如当你在吉他上弹奏G弦时，当触击吉他琴弦时，会产生特定频率和振幅的振动。琴弦前后振动的速率或频率决定了声音的音调，低音符产生低且慢速的频率；高音符产生高且快速的频率。振幅用来测量频率的相对强度，可以大致表示出你所听到的声音的音量。对于诸如吉他这种弦乐器，当你拨动琴弦的时候可以直观地看到信号的频率和振幅。这个振动会导致周围的空气分子移动，这些分子会依次同其临近的空气分子进行碰撞，之后再依次传递这个碰撞，将最初的振动能量不断地向各个方向进行传播。当这些波动传到接收者的耳中时，会带动耳膜以同样的频率和振幅振动。这些振动继续传递到内耳的耳蜗上，此时会将这些振动转换成电信号传给大脑，大脑会处理这个信号并知道“我听到了一个G调声音”。

当我们记录一个声音时，比如诸如钢琴或吉他等乐器所发出的声音，或捕捉其他环境的声音时，一般会使用麦克风设备。麦克风是将机械能量(声波)转换成电能量(电压信号)的转换设备。目前在用的麦克风种类很多，但是这里讨论的麦克风类型我们称为电动式麦克风(dynamic microphone)。图1-2展示了电动式麦克风内部的高级视图。

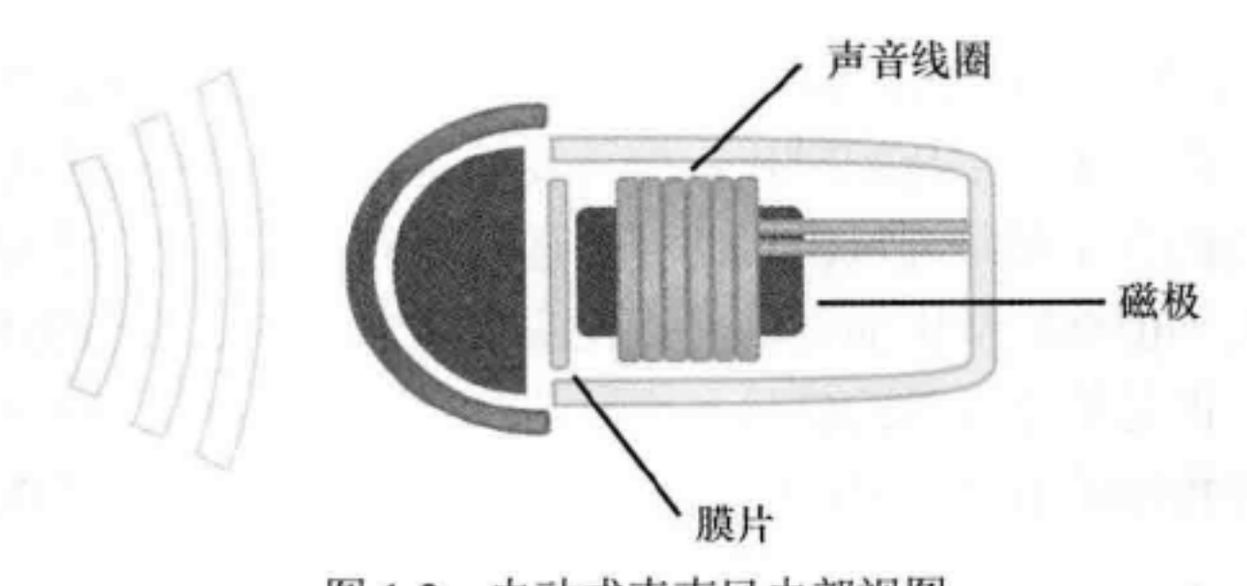

图 1-2 电动式麦克风内部视图

在麦克风的头部设备中所包含的，作为接收声音的一部分，是薄薄的被称为“膜片”的薄膜。这个膜片连接缠绕着磁极的线圈，当你对着麦克风讲话时，膜片会根据其感受到的声波进行振动，再依次带动线圈振动，产生同输入信号相同频率和振幅的电流信号。使用示波器我们可以看到电信号的振荡，如图1-3所示。

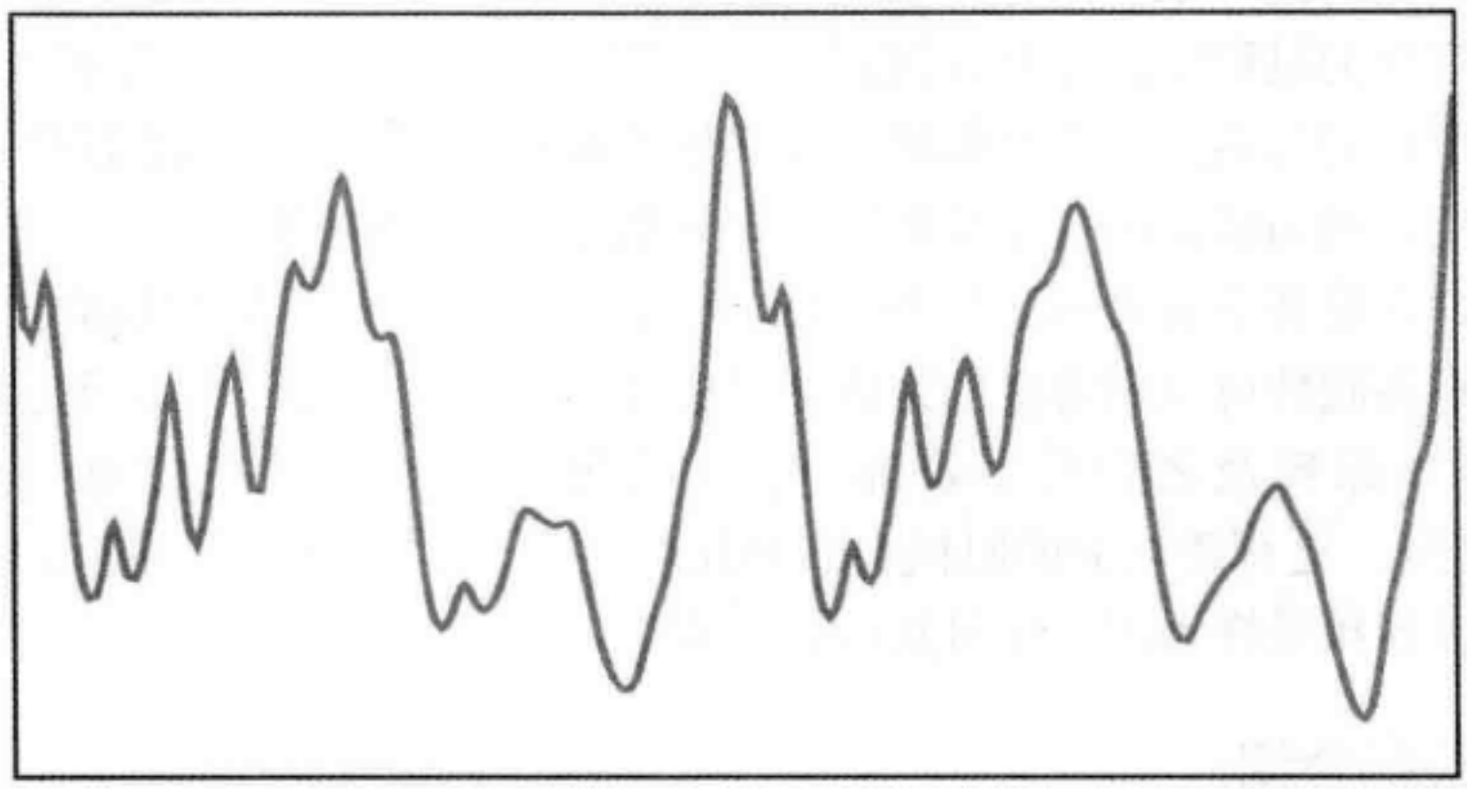

图 1-3 声音信号电压

下面回到之前关于采样的主题上，我们应该如何将这个连续的信号转换成相应的离散形式呢？先灌输一点之后会详细介绍的音频信号处理的核心知识，即使用音频生成器，这里我们创建了两个不同的音频正弦波，如图1-4所示。

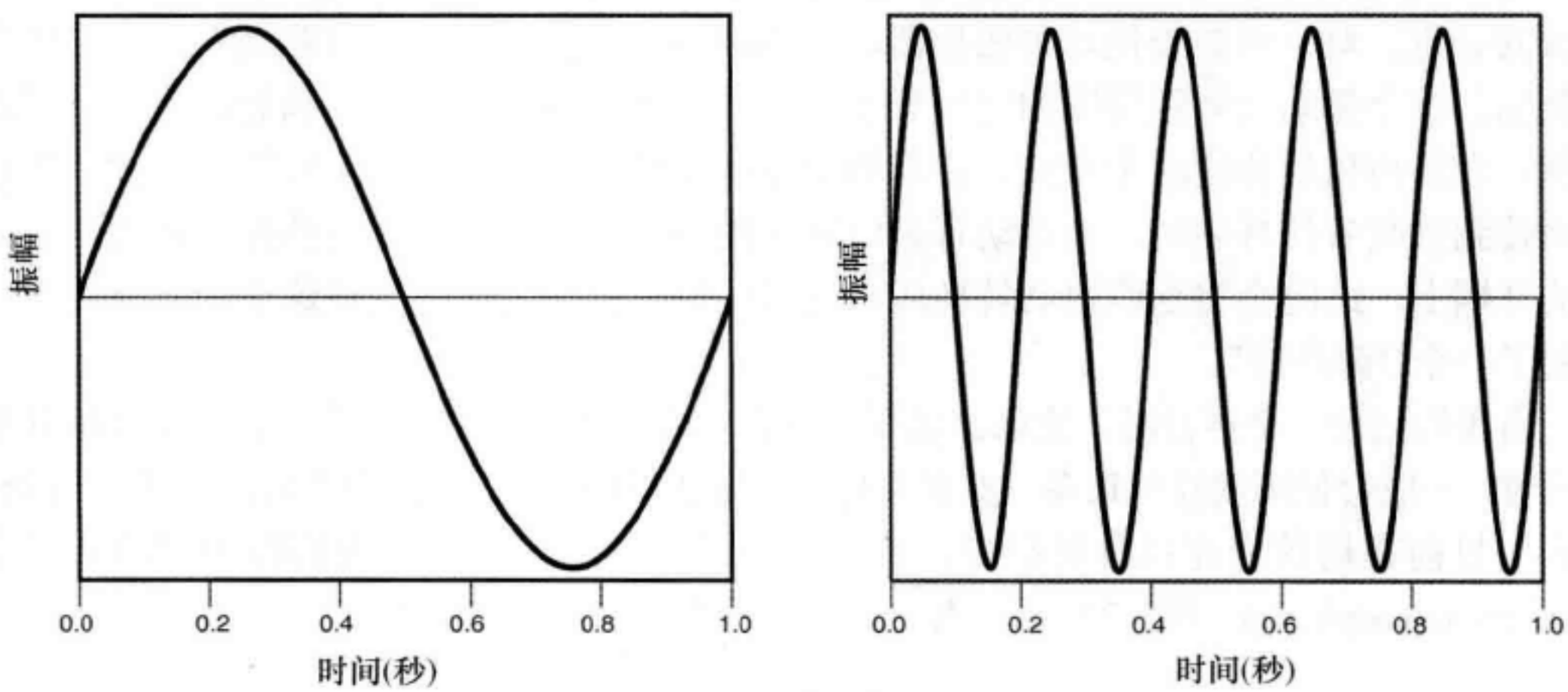

图 1-4 正弦波 1Hz(左)和 5Hz(右)

我们对这个信号的两个方面比较感兴趣。第一个是振幅，它代表了电压的强度或相应信号的强度。虽然可以通过不同坐标系来呈现正弦曲线的变化，但是通常我们会选择-1.0f到1.0f作为坐标系的最大值和最小值。对于这个信号，我们感兴趣的另一个方面是频率，信号频率的测量单位是赫兹(Hz)，表示在一定周期内振动完成循环的次数。图1-4左边的图片展示了1Hz情况下音频信号的一个周期，右边的图片展示了5Hz情况下的信号情况。人类可以听到的音频范围是20Hz～20kHz(20 000Hz)，所以图上的这两个信号都是不能被听见的，只是其更适合展示而已。

> **注意：**
>
> 虽然人类可以接收的音频范围为 20Hz 到 20kHz，不过这个频率范围也只是理论上的范围。因为几乎没有人可以听见这两个边界频率之外的声音，因为一旦处于嘈杂的环境之下听力就会减弱，同样随着年龄的增长听力也会变弱。如果你在一个摇滚演唱会的现场，相信这个理论上的频率范围的限制也不复存在了。
>
> 这里我们提供一些不同频率的声音参照，如钢琴、AO 最低音调的频率是 27.5Hz 和 C8，最高频率接近 4.1kHz。

音频数字化的过程包含一个编码方法，称为线性脉冲编码调制(linear pulse-code modulation)，比较常见的说法是Linear PCM或LPCM。这个过程采样或测量一个固定的音频信号，过程的周期率被称为采样率。图1-5展示了在一秒内对信号进行7次采样及信号的数字化结果图。

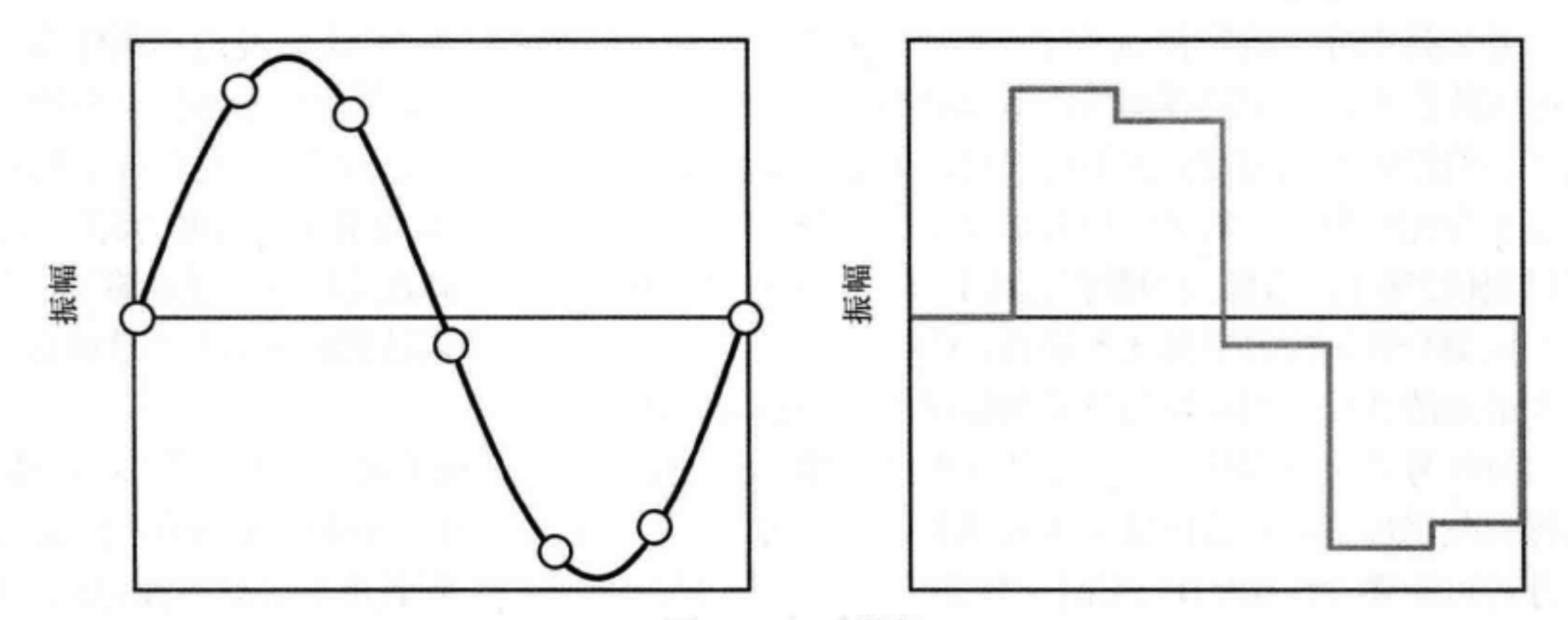

图 1.5　低采样率

显然低采样率的数字信号版本无法很好地表现原始数据。播放这样的音频信息通常会用在点击或弹出等动作的声音情况下。问题主要在于图1-5所示的采样频率尚无法准确地表示信号的原意。我们按图1-6所示的情况再做一次，这次我们提高采样频率。

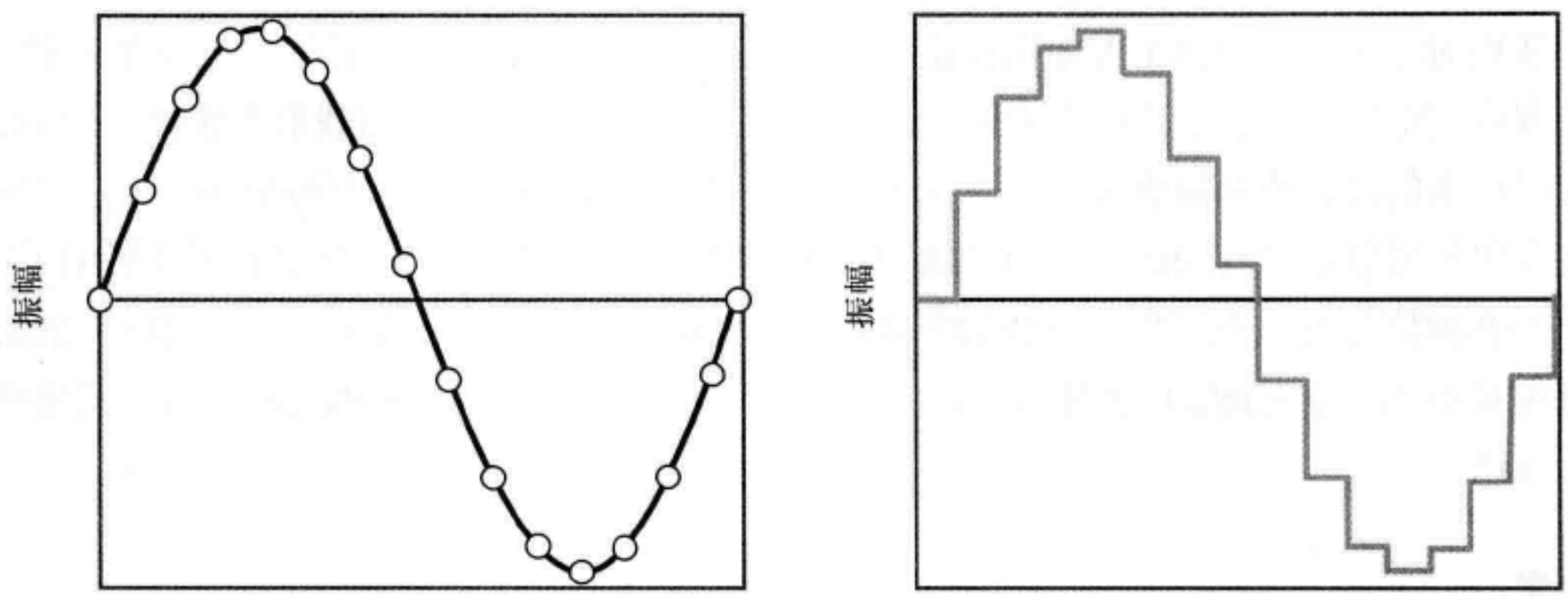

图 1-6 高采样率

这次效果显然得到提升，但仍不能准确表示原始信号。不过通过这个示例你可以推测如果不断提高采样的频率，我们就有可能以数字化方式准确表现原始信号的信息。鉴于硬件条件的限制，我们还不能复制出完全一样的效果，但是我们能否找到一个采样频率用于生成足够好的数字呈现效果？答案是肯定的。我们称其为尼奎斯特频率(Nyquist rate)。20世纪30年代，Harry Nyquist是贝尔实验室的一名工程师，他精确地捕捉到了一个特定频率，该频率为需要采样对象的最高频率的两倍。比如一个你需要捕捉的音频素材的最高频率为10kHz，你所需要的采样率最起码为20kHz才能得到较好的数字化效果。使用CD录制的音频采样率为44.1kHz，这就意味着能捕捉到的最大频率为22.05kHz，刚刚高过人耳能够识别的频率范围(20kHz)。44.1kHz的采样率可能还不能捕捉到初始资源中的所有频率范围，这意味着采样点可能会受到录制环境的干扰，因为其无法捕捉到Abbey Road会话的细微差别，不过对于人耳的听觉来说，这已经足够好了。

除采样率外，数字音频采样的另一个重要方面是我们能够捕捉到什么精度的音频样本。振幅在线性坐标系中进行测量，所以会有Linear PCM这个术语。用于保存样本值的字节数定义了在线性维度上可行的离散度，同时这个信息也被称为音频的位元深度。为每个样本的整体量化分配过少的位结果信息会导致数字音频信号产生噪声和扭曲。使用位元深度为8的方法可以提供256个离散级别的数据，对于一些音频资源来说，这个级别的采样率已经足够了，但对于大部分音频内容来说还不够高。CD音质的位元深度为16，可以达到65 536个离散级别。专业级别的音频录制环境的位元深度可以达到24或更高。

对信号进行数字化时，如果能够保留原始、未压缩的数字呈现效果，就是该媒体资源最纯粹的数字形式，但这样做需要大量的存储空间。比如一个44.1kHz、16位LPCM的音频文件每分钟可能要占用10MB的空间。要数字化一个含12首歌的唱片，每首歌曲时间大概为5分钟的话，共需要近600MB的存储空间。即使在当今的海量存储和高带宽的情况下，这个文件体积仍然很大。所以我们可以看到不经过压缩的数字音频资源会占用大量的存储空间，但是未压缩的视频文件的情况是不是这样呢？下面了解一下组成数字视频文件的元素来判断能否满足其对存储空间的需求。

视频文件由一系列称为“帧”的图片组成，在视频文件的时间轴线上每一帧都表示一个场景。要创建连续的运动画面，我们需要在短时间间隔内提供特定数量的帧。视频文件一秒钟内所能展现的帧数称为视频的帧率，并用FPS作为单位进行测量。常见的帧率是24FPS、

25FPS和30FPS。

要知道未压缩的视频内容所需的存储空间，我们首先需要确定每一个独立的帧有多大。我们知道有许多通用的视频尺寸，但是目前视频资源最流行的宽高比为16:9，意思是每16个水平像素对应9个垂直像素。在这一宽高比之下最常见的视频尺寸是1280×720和1920×1080。那么各自的像素情况是什么样的呢？如果对每个像素点使用8位的RGB色彩空间，这就意味着红色占8位，绿色占8位，蓝色占8位。所有的输入收集好后，我们执行一些计算。表1-1展示了对于未压缩视频在30FPS帧率的情况下，上述两个分辨率的存储空间需求。

表 1-1　未压缩视频存储需求

颜色	分辨率	帧率	MB/s	GB/h
24 位	1280×720	30FPS	79MB/s	278GB/h
24 位	1920×1080	30FPS	178MB/s	625GB/h

哦，天啊！我们现在遇到问题了！显然，作为存储和传输文件的格式，这简直难以想象。也许十年后这些文件看起来能够接受，但是目前对于大多数情况我们都无法方便地使用这么大的媒体文件。由于大部分情况下对视频进行存储或传输都是不可取的，所以我们需要找到一种方式缩小资源的尺寸。这就带来了我们接下来讨论的资源压缩的话题。

1.5　数字媒体压缩

为缩小数字媒体文件的大小，我们需要对其使用压缩技术。一般来说我们所欣赏的媒体内容都进行过一定程度的压缩。无论是电视上的视频、蓝光碟片、网页上的视频流还是从iTunes Store购买的资源，我们其实都是和这些内容的原始文件的压缩格式在打交道。对数字媒体进行压缩可以大幅缩小文件的尺寸，但是通常会在资源的质量上有小幅可见的衰减。

1.5.1　色彩二次抽样

视频数据是使用称之为$Y'C_bC_r$颜色模式的典型案例，$Y'C_bC_r$也常称为YUV。虽然YUV术语并不很准确，但是YUV读起来要比Y-Prime-C-B-C-R方便很多。不过大部分软件开发者都更熟悉RGB颜色模式，即每个像素都是由红、绿、蓝三个颜色组合而成。$Y'C_bC_r$或YUV则使用色彩(颜色)通道UV替换了像素的亮度通道Y(亮度)。图1-7展示了一幅图片分离亮度和色彩通道后的效果。

可以看到图片的所有细节都保存在亮度通道中，如果除去亮度，剩下的就是一幅灰度图片，我们再看整合的色彩通道中几乎所有的细节都丢失了。这是因为我们的眼睛对亮度的敏感度要高于颜色，聪明的工程师们认识到，我们可以大幅减少存储在每个像素中的颜色信息，而不至于图片的质量严重受损。这个减少颜色数据的过程就称为色彩二次抽样。

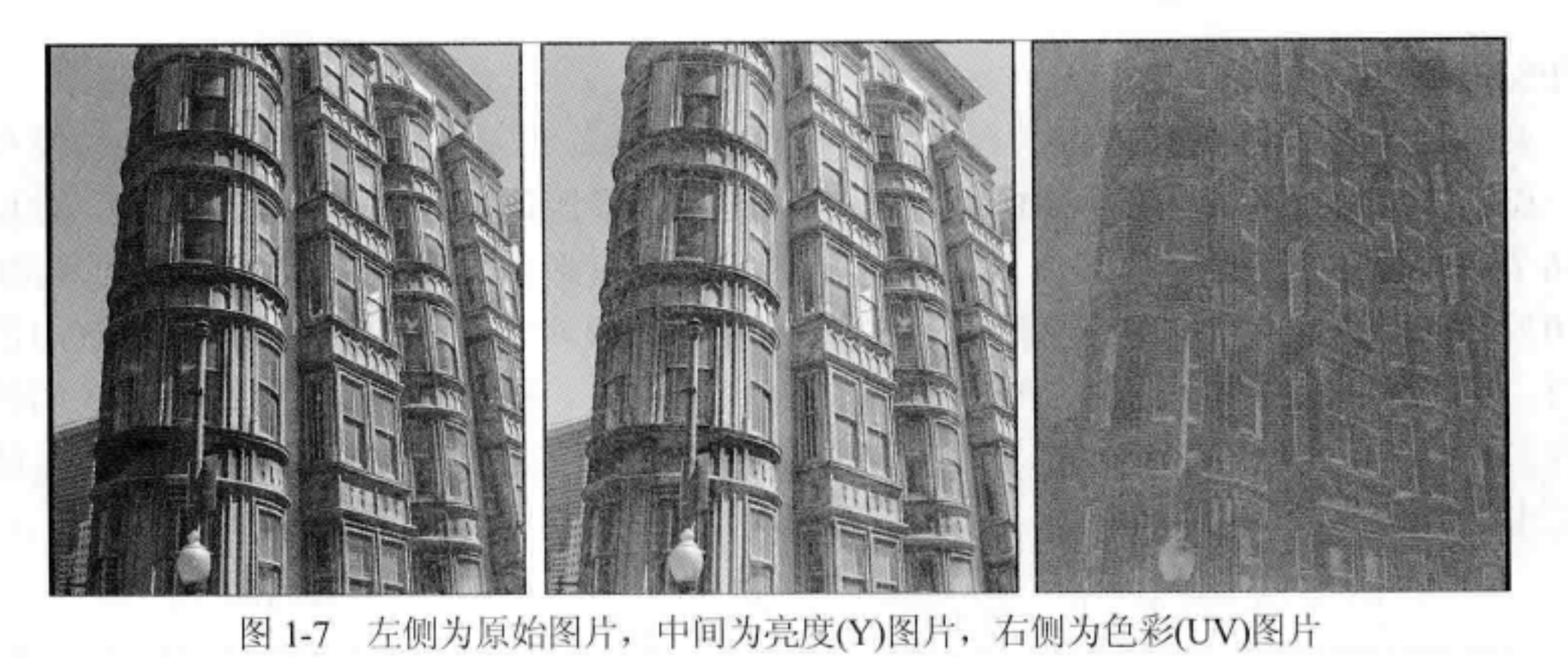

图 1-7 左侧为原始图片，中间为亮度(Y)图片，右侧为色彩(UV)图片

当每次看到诸如摄像头规范和其他视频设备硬件或软件中提到的4:4:4、4:2:2及4:2:0时，这些值的含义就是这些设备所使用的色彩二次抽样的参数。根据这些值按如下格式将亮度比例表示为色度值，这个格式写作J:a:b，具体含义如下：

- J：几个关联色块(一般是 4 个)中所包含的像素数。
- a：用来保存位于第一行中的每个 J 像素的色度像素个数。
- b：用来保存位于第二行中的每个 J 像素的附加像素个数。

为维持图片质量，每个像素点都需要有各自的亮度值，却不一定需要色度值。图1-8展示了几个常见的二次抽样比率及其效果。

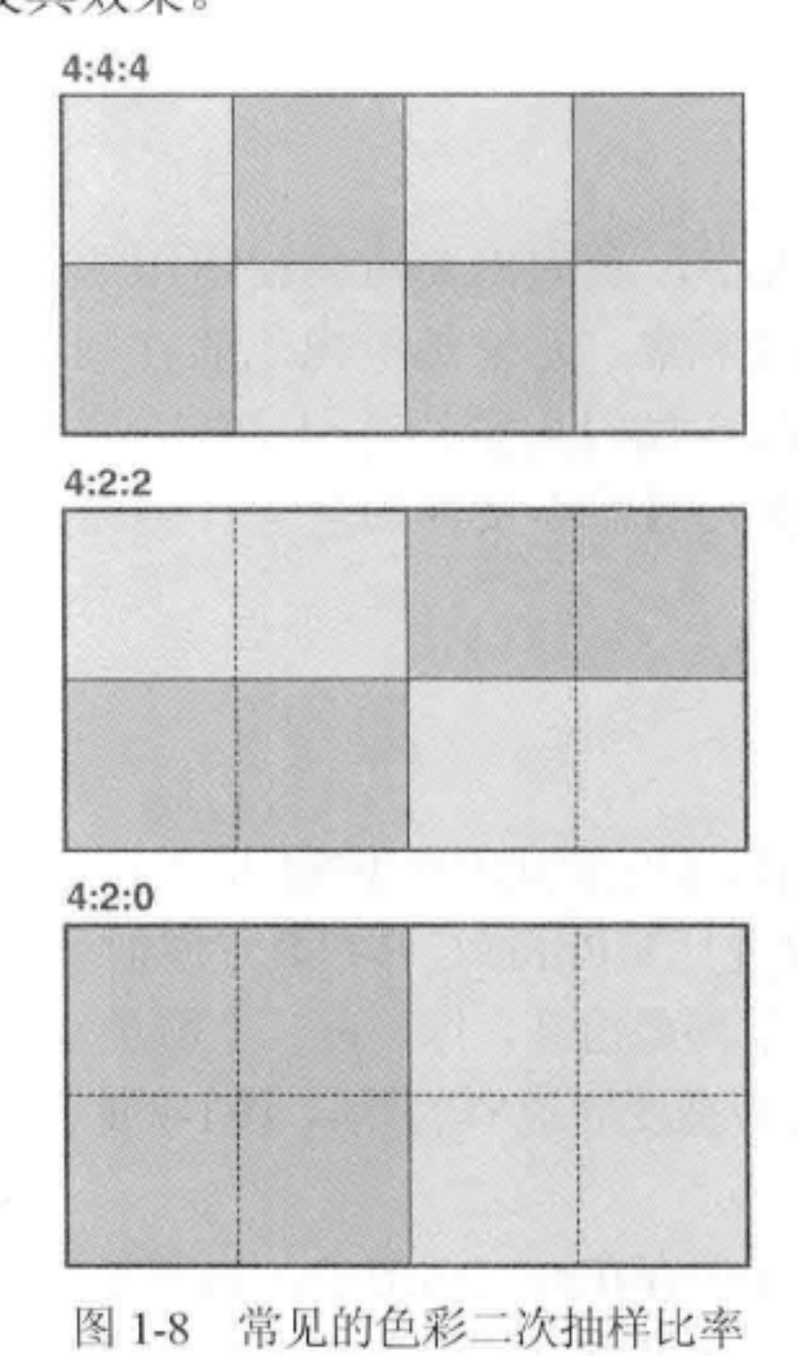

图 1-8 常见的色彩二次抽样比率

在所有格式下，每个像素的全部亮度信息都被保存下来，在4:4:4的比例下全彩色信息也都被保存下来。在4:2:2的情况下，色彩信息为每两个水平像素的平均值，即亮度和色度比率为2:1。在4:2:0的情况下，色彩信息为水平和垂直两个方向的4个像素的平均值，结果就是亮

度和色度的比率为4:1.

色彩二次抽样一般发生在取样时，一些专业的相机以4:4:4的参数捕捉图像，但大部分情况下对于图片的拍摄是使用4:2:2的方式进行的。面向消费者的摄像头装置，比如iPhone手机上的摄像头，通常以4:2:0的方式进行拍摄。即使经过大量层级的二次抽象之后也仍然可能捕捉到高质量的图片，iPhone手机上拍出来的高质量视频就是很好的例证。当以色彩作为核心或在生产后期校正颜色的时候，如果色彩方面出现了衰减，那会成为一个棘手的问题。由于我们的色彩信息是通过多个像素点平均得来的，噪声点和其他一些失真问题也会出现在图片上。

1.5.2　编解码器压缩

大部分音频和视频都是使用编解码器(codec)来压缩的，编解码器这个术语是由编码器/解码器结合简写得来的(encoder/decoder)。编解码器使用高级压缩算法对需要保存或发送的音频或视频数据进行压缩和编码，同时它还可以将压缩文件解码成适合播放和编辑的媒体资源文件。

编解码器既可以进行无损压缩也可以进行有损压缩，无损压缩编解码器以一种可以完美重构解码的方式对媒体文件进行压缩，使其成为无论编辑还是发布都比较理想的文件，有时也会作为归档文件用。我们经常使用这类压缩方式，比如我们最常用的zip和gzip即是这种方式的压缩。顾名思义，有损编解码器就是在压缩过程中会有部分数据损失掉。编解码器为这一形式的压缩使用基于人类可感知的高级压缩算法。比如即使人类可以理论上听见介于20Hz和20kHz之间的声音，但我们可能真正敏感的频率区间是1kHz～5kHz。我们对于频率的感知随着远离上述区间而逐渐减弱。在了解这一原理后，我们知道在音频文件中可以利用过滤技术来减少或消除特定频率。这只是众多方法中的一个，但有损压缩的目的是使用psycho-acoustic或psycho-visual模式作为一种方法来减少媒体内容中的冗余数据，这样会使原文件质量的损耗达到最小。

下面看一下由AV Foundation提供的编解码器支持。

1.5.3　视频编解码器

对于视频编解码而言，AV Foundation提供有限的编解码器集合，只提供苹果公司认定的目前最主流的几种媒体类型的支持。具体对于视频文件来说，主要可以归结为H.264和Apple ProRes。下面开始学习H.264视频格式标准。

1. H.264

当我们需要对发送的视频文件进行编码时，Henry Ford曾经说过，只要是H.264标准的文件，AV Foundation都提供视频编解码器支持。幸运的是，行业内也默认对该编解码器进行了整合。这个标准广泛用于消费者视频摄像头捕捉到的资源并成为网页流媒体视频所使用的最主要格式。所有从iTunes Store下载的视频文件也会使用这个编解码器进行编码。H.264规范是Motion Picture Experts Group(MPEG)所定义的MPEG-4的一部分。H.264遵循早期的MEPG-1和MPEG-2标准，但是在以更低比特率得到更高图片质量方面有了长足进步，使其更好地用

于流媒体文件和移动设备及视频摄像头。

H.264与其他形式的MPEG压缩一样，通过以下两个维度缩小了视频文件的尺寸：

- 空间：压缩独立视频帧，被称为帧内压缩。
- 时间：通过以组为单位的视频帧压缩冗余数据，这一过程称为帧间压缩。

帧内压缩通过消除包含在每个独立视频帧内的色彩及结构中的冗余信息来进行压缩，因此可在不降低图片质量的情况下尽可能缩小尺寸。这类压缩同JEPG压缩的原理类似。帧内压缩也可以作为有损压缩算法，但通常用于对原始图片的一部分进行处理以生成极高质量的照片。通过这一过程创建的帧称为I-frames。

在帧间压缩中，很多帧被组合在一起作为一组图片(简称GOP)，对于GOP所存在的时间维度的冗余可以被消除。如果想象视频文件中的典型场景，就会有一些特定运动元素的概念，比如行驶的汽车或街上走路的行人，场景的背景环境通常是固定的。固定的背景环境就代表一个时间维度上的冗余，这个冗余就可以通过压缩方式进行消除。

图1-9给出了存储在GOP中的三个不同类型的帧。

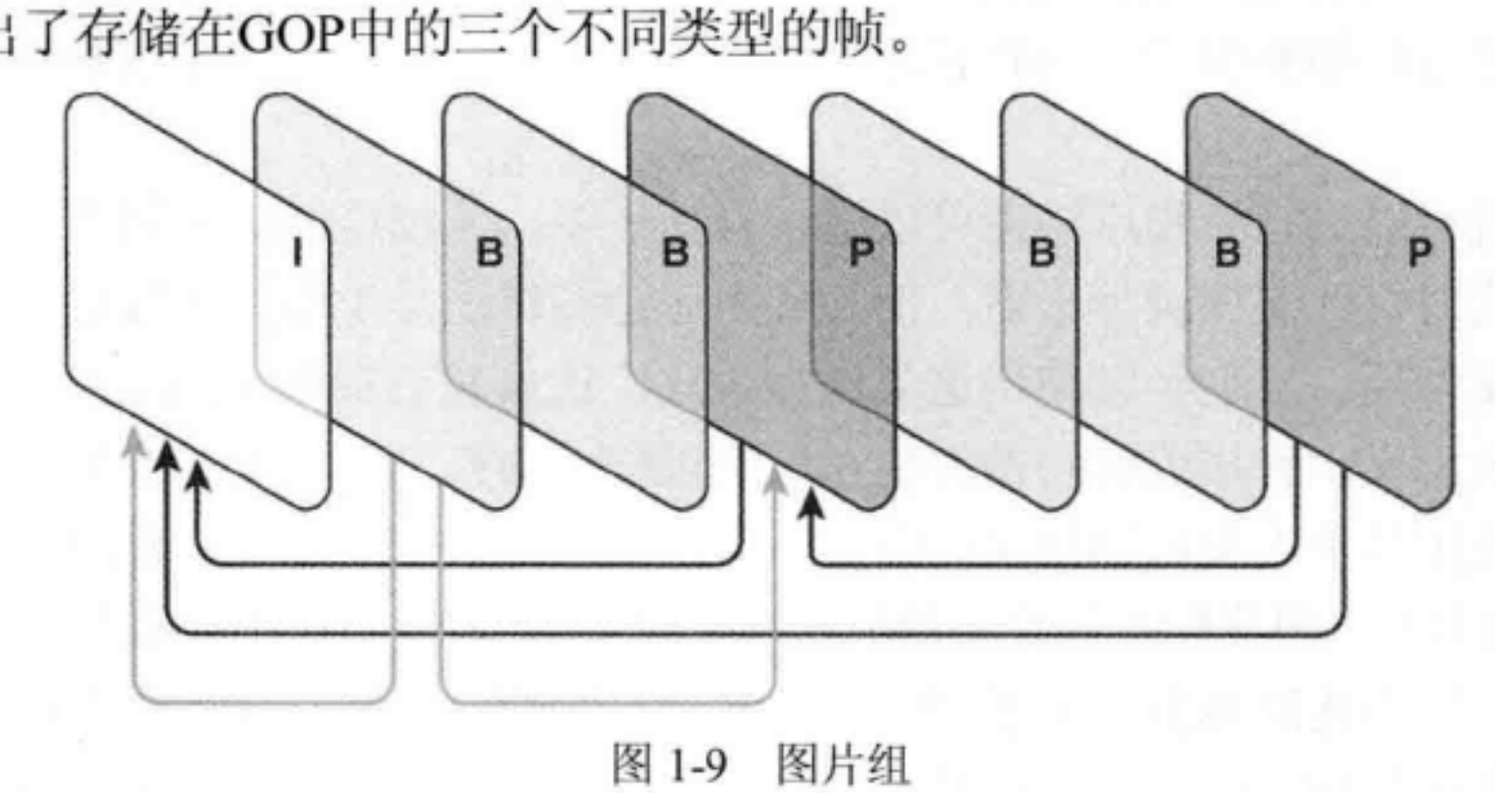

图 1-9 图片组

- **I-frames：**这些帧都是一些单独的帧或关键帧，包含创建完整图片需要的所有数据。每个 GOP 都正好有一个 I-frames。由于它是一个独立帧，其尺寸是最大的，但也是解压最快的。
- **P-frames：**P-frames 又称为预测帧，是从基于最近 I-frames 或 P-frames 的可预测的图片进行编码得到的。P-frames 可以引用最近的预测 P-frames 或一组 I-frames。你将会经常看到这些被称为“reference frames”的帧，临近的 P-frames 和 B-frames 都可以对其进行引用。
- **B-frames：**B-frames 又称为双向帧，是基于使用之前和之后的帧信息进行编码后得到的帧。几乎不需要存储空间，但其解压过程会耗费较长时间，因为它依赖于周围其他的帧。

H.264还支持编码视图，用于确定在整个编码过程中所使用的算法。共定义了3个高级标准：

- **Baseline：**这个标准通常用于对移动设备的媒体内容进行处理，提供了最低效的压缩，因此经过这个标准压缩后的文件仍较大，但是同时这种方法也是最少计算强度的方法，因为它不支持 B-frames。如果开发者的编译目标是年代比较久远的 iOS 设备，比如 iPhone 3GS，可能需要用到 Baseline 标准。

- **Main**：这个标准的计算强度要比 Baseline 的高，因为它使用的算法更多，但可以达到比较高的压缩率。
- **High**：高标准的方法会得到最高质量的压缩效果，但它也是 3 种方法中计算复杂度最高的，因为所有能用到的编码技术和算法几乎都用到了。

2. Apple ProRes

AV Foundation支持Apple ProRes编解码器的两种不同风格。Apple ProRes被认为是一个中间件或中间层编解码器，因为它的目的是为专业编辑和生产工作流服务。Apple ProRes编解码器是独立于帧的，意味着只有I-frames可以被使用，这就使其更适合用在内容编辑上。此外，Apple ProRes还使用可变比特率编码的方式来对复杂场景中的每一帧进行编码。

ProRes是有损编解码器，但是它具有最高的编解码质量。Apple ProRes 422使用4:2:2的色彩二次抽样和10位的采样深度。Apple ProRes 4444使用4:4:4色彩二次抽样，具有最终4个用于表示支持无损alpha通道和高达12位的采样深度。

ProRes编解码器只在OS X上可用。如果开发者只针对iOS进行开发工作，只能使用H.264。然而苹果公司提供了一种方法，即当我们捕捉图像的目的是对其进行编辑时，可以将目标变换成通用的H.264编码格式，称为iFrame。对于编辑环境而言这是一个I-frames-only变量生成H.264视频更合适的一种方法。这种格式在AV Foundation中是受支持的，并且它还被众多的摄像头生产企业所支持，比如Canon、Panasonic和Nikon。

> **注意：**
>
> 对 H.264 和 Apple ProRes 来说，AV Foundation 还支持很多摄像头设备的编解码器，如 MPEG-1、MPEG-2、MPEG-4、H.263 和 DV，允许用户以多种不同的视频捕捉设备导入内容资源。

1.5.4　音频编解码器

只要是Core Audio框架支持的音频编解码，AV Foundation都可以支持，这意味着AV Foundation能够支持大量不同格式的资源。然而在不用线性PCM音频的情况下，更多的只能使用AAC。

AAC

高级音频编码(AAC)是H.264标准相应的音频处理方式，目前已成为音频流和下载的音频资源中最主流的编码方式。这种格式比MP3格式有着显著的提升，可以在低比特率的前提下提供更高质量的音频，是在Web上发布和传播的音频格式中最为理想的。此外，AAC没有来自证书和许可方面的限制，这一限制曾经在MP3格式上饱受诟病。

> **注意：**
>
> AV Foundation 和 Core Audio 提供对 MP3 数据解码的支持，但是不支持对其进行编码。

1.6 容器格式

如果你和大家一样，喜欢在自己的电脑上查找不同类型的媒体文件，你可能会注意到以各种扩展名结尾的文件，比如.mov、.m4v、.mpg和.m4a等。虽然我们通常将这些类型都认为是文件格式，但其正确定义应该是这些类型都是文件的容器格式(container format)。

容器格式被认为是元文件格式。从更高的角度看，可将容器格式视为包含一种或更多种媒体类型(以及描述其内容的元数据)的目录。比如QuickTime文件可以包含多种不同的媒体类型，包括视频、音频、字幕和章节信息等，并且包含用于描述每个媒体片段细节的元数据。

每种格式都有一个规范用于确定文件的结构。所谓的结构并不仅是其包含的媒体资源技术领域的问题，比如媒体的周期、编码和时间信息，一般来说还定义了描述性原数据，比如电影标题或一首歌曲的作者信息等。这些元数据可以通过工具进行呈现，比如常见的iTunes或iOS的Music应用程序，而且AV Foundation还提供了相关的类用于在开发者的应用程序内读取或写入该类型的数据。

当开发者使用AV Foundation撰写代码时，将遇到两类主要的容器格式，它们分别是：

- **QuickTime：** QuickTime 是苹果公司在更宏观 QuickTime 架构中定义的最常用格式。其具有非常高的可靠性并且是一种有着非常清晰定义的格式，被专业领域人士和普通消费者广泛使用。苹果公司在 *QuickTime File Format Specification* 文档中对其细节进行了详细描述，可在 Apple Developer Connection 网站上找到该文档。我们建议所有 AV Foundation 开发者都应该至少阅读上述文档中基本知识点的介绍章节，因为了解相关的知识对之后的应用程序开发有帮助。
- **MPEG-4：** MPEG-4 Part14 规范定义 MPEG-4(MP4)容器格式。这是从 QuickTime 规范中直接派生出来的一种行业标准格式，所以这两个规范在结构和功能方面非常类似。MP4 容器格式的官方文件扩展名是.mp4，但有很多不同的变化扩展名也在使用，尤其是在苹果系统生态环境中。这些变化的文件扩展名仍然使用相同的基本 MP4 容器格式，它们通常用来区分一些特定的媒体类型，如 m4a 格式的音频文件，还可以使用这些扩展名来标识一些基本的 MP4 容器，如 m4v 格式的视频文件。

1.7 初识 AV Foundation

现在你应该对AV Foundation有了比较深入的了解，并且对数字媒体的细节也有了一定认识，下面增添一些有趣的东西。

Mac OS X一直都有一个NSSpeechSynthesizer类，使用这个类可以很方便地在Cocoa应用程序中添加“文本到语音”功能。开发者可使用AV Foundation中的AVSpeedSynthesizer类向iOS应用程序中添加类似功能。这个类用来播放一个或多个语音内容，这些语音内容都是名为AVSpeechUtterance的类的实例。如果你希望播放语句“Hello World!”，具体的实现代码如下所示：

```
AVSpeechSynthesizer *synthesizer = [[AVSpeechSynthesizeralloc] init];
AVSpeechUtterance *utterance =
    [[AVSpeechUtterancealloc] initWithString:@"Hello World!"];
[synthesizerspeakUtterance:utterance];
```

如果运行这段代码，将听见以默认区域设置的声音读出语句“Hello World!”。现在通过创建一个简单的应用程序实现这个功能，在应用程序中将用到AV Foundation会话。

本书中创建的所有项目在书中的示例代码库里都会有一个初始版本和一个最终版本。最终版本是一个完整的项目并随时可以进行编译和运行。初始版本具有用户接口和完整的支持类，并包含一些需要读者继续编写代码来实现的不完整的类描述。此外，大部分的示例项目都有一种将AV Foundation代码从其余应用程序代码中分离出来的代码分解方法。这就让读者能将更多注意力集中在AV Foundation相关的内容上，而避免受到用户接口细节的干扰，同时也让无论最初是接触OS X系统还是iOS系统的开发者都可以顺利地使用该示例项目。

在本书的示例代码库中，你将在Chapter 1目录中发现一个名为HelloAVF_Starter的初始项目。图1-10展示了这个应用程序的实际效果。

图 1-10　Hello AV Foundation

在项目中你将发现一个名为THSpeechController的类，这个类是要在其中撰写“文本到语音”功能的实现类。代码清单1-1给出了关于这个类的接口信息。

代码清单 1-1　THSpeechController.h

```
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController : NSObject

@property (strong, nonatomic, readonly) AVSpeechSynthesizer *synthesizer;

+ (instancetype)speechController;

- (void)beginConversation;

@end
```

这个类具有一个简单接口，只需讲述几点。代码头部导入了<AVFoundation/AVFoundation.h>，作为框架的总体头文件。在本书所有课程中，这一写法是所有代码通用的。这个类的核心方法是beginConversation，其会立即创建并开启“文本到语音”功能并运行应用程序。下面转过来看一下该类的具体实现(见代码清单1-2)。

代码清单 1-2 THSpeechController.m

```
#import "THSpeechController.h"
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController ()
@property (strong, nonatomic) AVSpeechSynthesizer *synthesizer;          // 1
@property (strong, nonatomic) NSArray *voices;
@property (strong, nonatomic) NSArray *speechStrings;
@end

@implementation THSpeechController

+ (instancetype)speechController {
    return [[self alloc] init];
}

- (id)init {
    self = [super init];
    if (self) {
        _synthesizer = [[AVSpeechSynthesizer alloc] init];               // 2

        _voices = @[[AVSpeechSynthesisVoice voiceWithLanguage:@"en-US"], // 3
                    [AVSpeechSynthesisVoice voiceWithLanguage:@"en-GB"]];

        _speechStrings = [self buildSpeechStrings];
    }
    return self;
}
- (NSArray *)buildSpeechStrings {                                        // 4
    return @[@"Hello AV Foundation. How are you?",
             @"I'm well! Thanks for asking.",
             @"Are you excited about the book?",
             @"Very! I have always felt so misunderstood",
             @"What's your favorite feature?",
             @"Oh, they're all my babies. I couldn't possibly choose.",
             @"It was great to speak with you!",
             @"The pleasure was all mine! Have fun!"];
}
- (void)beginConversation {
}
@end
```

(1) 在类的扩展部分定义类所需的属性，重新对之前在头部定义的synthesizer属性进行定义，这样就可以支持读写操作。此外，还需要为具体对话中用到的声音和语音字符串定义属性。

(2) 创建一个新的AVSpeechSynthesizer实例。该对象用于执行具体的“文本到语音”会话。对于一个或多个AVSpeechUtterance实例，该对象起到队列的作用，提供了接口供控制和监视正在进行的语音播放。

(3) 创建一个包含两个AVSpeechSynthesisVoice实例的NSArray对象。对于声音的支持目前非常有限。开发者不能像在Mac机器上那样使用指定名称的语音进行播报。取而代之的是每种语言/区域设置都有预先设置好的声音。这种情况下，1号扬声器使用的是美式英语，2号扬声器使用的是英式英语的发音。开发者可以通过调用AVSpeechSynthesisVoice中的speechVoices类方法来查看完整的声音支持列表。

(4) 创建一个字符串数组用来定义所设计会话的有关前进和后退的处理。

现在基本完成了类的设置，下面继续讨论beginConversation方法的具体实现，如代码清单1-3所示。

代码清单 1-3　实现 beginConversation 方法

```
- (void)beginConversation {
    for (NSUInteger i = 0; i < self.speechStrings.count; i++) {
        AVSpeechUtterance *utterance =                          // 1
            [[AVSpeechUtterance alloc] initWithString:self.speechStrings[i]];
        utterance.voice = self.voices[i % 2];                   // 2
        utterance.rate = 0.4f;                                  // 3
        utterance.pitchMultiplier = 0.8f;                       // 4
        utterance.postUtteranceDelay = 0.1f;                    // 5
        [self.synthesizer speakUtterance:utterance];            // 6
    }
}
```

(1) 首先对语音字符串进行循环，并且每个循环都要创建一个新的AVSpeechUtterance实例，传递字符串到其initWithString:初始化器。

(2) 之后在预先定义的两个语音之间向后或向前进行切换。甚至可能有些语音使用美语进行播报，剩下的一些语音使用英式英语进行播报。

(3) 明确将要播放语音内容的速率。这里将该值设置为0.4，略低于默认速率。我们应该指出文档状态，使播放率介于AVSpeechUtteranceMinimumSpeechRate和AVSpeechUtterance-MaximumSpeechRate之间。目前这两个值分别是0.0和1.0。不过由于这些值是常量，数值可能在之后的iOS新版中发生变化。如果开发者修改rate属性，按照最小值和最大值范围的百分比进行计算，可能更安全些。

(4) 为语句指定pitchMultiplier，可在播放特定语句时改变声音的音调。pitchMultiplier的允许值一般介于0.5(低音调)和2.0(高音调)之间。

(5) 指定postUtteranceDelay的值为1.0f，这一动作的目的是让语音合成器在播放下一语句之前有短时间的暂停。开发者可以类似地设置preUtteranceDelay。

运行应用程序并聆听应用程序播放对话内容，现在我们以AV Foundation的方式成功播放了Hello World！

为AVSpeechUtterance设置尝试不同的值来了解其运行原理，试听其他一些声音。创建一

个AVSpeechUtterance实例来阅读《战争与和平》的全部内容，然后坐下来慢慢放松欣赏。

> **注意：**
>
> 在本书完稿之际，iOS 8 和 Xcode 6 最终版本都已经发布了。请查阅源代码库中的 Xcode 6 and iOS 8 Notes.pdf 文件来了解在 Xcode 6 和 iOS 8 中运行示例项目的附加信息。

1.8 小结

本章对AV Foundation框架进行了介绍。现在你应该更了解在苹果公司的媒体环境中什么时候会用到AV Foundation，及该框架能提供哪些功能。你现在应该对数字媒体这一领域的知识有了更深理解，虽然使用AV Foundation可以让开发者在不需要了解很多媒体资源细节知识的情况下构建一些功能强大的应用程序，但是读者朋友们会发现当你对数字媒体这一领域有了更深的理解时，将能更方便地创建出你所期望的应用程序。AV Foundation是Mac OS X和iOS系统之下媒体资源的未来，本书提供了详尽的介绍，向读者朋友们讲述如何成功使用该框架来创建下一代媒体应用程序。

1.9 挑战

打开AV Foundation的API文档，该文档可在Xcode的文档浏览器中找到，也可以在Apple Developer Connection站点找到。利用一些时间浏览一下全部文档的内容，对其中所涉及的类的逻辑关系进行了解，同时对整个框架中所使用的命名约定有所了解。这样做可以让你开始了解该框架所提供的功能范围，并且使你更好地熟悉框架使用过程中的编码模式和约定。

第2章

播放和录制音频

AV Foundation最开始是一个仅针对音频的框架。该框架的前身在iOS 2.2版本中引入，只包含一个处理音频播放的类；在iOS 3.0中，苹果公司增加了音频录制功能。虽然这些类都是目前该框架中最古老的，但它们仍是最常用的几个类。本章我们通过学习AVAudioPlayer和AVAudioRecorder类来实现在应用程序中添加音频播放及音频录制的功能。

2.1 Mac 和 iOS 的音频环境

在开始讨论音频播放和录制前，了解Mac和iOS平台所提供的音频环境是非常重要的。Mac系统的音频环境非常自由和灵活——大部分情况都由用户控制。用户可以同时播放iTunes上的音频、欣赏QuickTime上的电影、听网络上的流媒体音频和录制GarageBand应用程序中的吉他演奏。虽然这样做会导致声音过载，不过OS X的确是可以这样做的。系统会主动对硬件和软件进行管理使其成为可能，不过系统在管理音频用户体验上扮演了被动角色。

下面通过一个通用场景来对比iOS平台上的情况：

你在iPhone上点开一首歌曲，音频从内置扬声器中播放出来，此时有电话拨入，音乐会立即停止并处于暂停状态，此时听到的是手机呼叫的铃声。你猜到电话的内容可能是你的老板询问你是否在TPS报告中加了封面，所以你决定要挂掉电话。刚才的音乐声再次响起，此时你意识到使用iPhone的内置扬声器播放这首经典的Parliament-Funkadelic歌曲不太合适，所以戴上了耳机。当音乐继续播放时音频输出到了耳机里。当听完这首音乐摘下耳机后，你会发现声音自动转回内置扬声器并处于暂停状态。

从技术角度看，上面描述的场景非常复杂，但从用户的视角看，这一切只是有些神奇而已。iOS系统提供了一个可管理的音频环境(managed audio environment)，可以带给所有iOS用户非常好的用户体验，但这一神奇的过程具体是如何实现的呢？这里会用到音频会话(audio session)。

2.2 理解音频会话

音频会话在应用程序和操作系统之间扮演着中间人的角色。它提供了一种简单实用的方法使OS得知应用程序应该如何与iOS音频环境进行交互。你不需要了解与音频硬件交互的细节，只需要对应用程序的行为进行语义上的描述即可。这一点使得你可以指明应用程序的一般音频行为，并可以把对该行为的管理委托给音频会话，这样OS系统就可以对用户使用音频的体验进行最适当的管理。

所有iOS应用程序都具有音频会话，无论其是否使用。默认音频会话来自于以下一些预配置：

- 激活了音频播放，但是音频录制未激活。
- 当用户切换响铃/静音开关到“静音”模式时，应用程序播放的所有音频都会消失。
- 当设备显示解锁屏幕时，应用程序的音频处于静音状态。
- 当应用程序播放音频时，所有后台播放的音频都会处于静音状态。

默认音频会话提供了很多实用的功能，但重要的一点是要确保它是应用程序当前正确的行为。在大部分应用程序中，默认的行为都是好的，但在开发媒体应用程序时一般是不需要的。幸运的是，通过使用“分类”功能，可以很容易地定制我们的特殊需求。

2.2.1 音频会话分类

AV Foundation定义了7种分类来描述应用程序所使用的音频行为。表2-1列出了这些分类名称及它们相应的作用和行为。

表 2-1 音频会话分类

分类	作用	是否允许混音	音频输入	音频输出
Ambient	游戏、效率应用程序	✔		✔
Solo Ambient (默认)	游戏、效率应用程序			✔
Playback	音频和视频播放器	可选		✔
Record	录音机、音频捕捉		✔	
Play and Record	VoIP、语音聊天	可选	✔	✔
Audio Processing	离线会话和处理			
Multi-Route	使用外部硬件的高级 A/V 应用程序		✔	✔

当为应用程序选择合适的分类时，你需要问自己关于这个核心行为的一些问题。比如音频播放是核心功能还是次要功能？应用程序的音频是否可以和背景声音相混合？应用程序是否需要捕捉音频输入进行录制或通过网络发送音频？确定了应用程序的核心音频行为后，选择合适的分类就变得比较容易了。

上述分类所提供的几种常见行为可以满足大部分应用程序的需要，不过如果开发者需要更复杂的功能，其中一些分类可以通过使用options和modes方法进一步自定义开发。options

可以让开发者使用一些附加行为，如使用Playback分类后，应用程序允许将输出音频和背景声音进行混合。modes可以通过引入被定制的行为进一步对分类进行修改以满足一些特殊需求。VoIP和视频聊天应用程序常使用这些模式来获得需要的功能，还有一些模式也针对视频播放和录制应用程序提供了非常有用的功能。

2.2.2　配置音频会话

下面看一个有关如何配置音频会话的示例。音频会话在应用程序的生命周期中是可以修改的，但通常我们只对其配置一次，就是在应用程序启动时。配置音频会话的最佳位置就是应用程序委托的application:didFinishLaunchingWithOptions:方法。

```
- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

  AVAudioSession *session = [AVAudioSession sharedInstance];
  NSError *error;
  if (![session setCategory:AVAudioSessionCategoryPlayback error:&error]) {
      NSLog(@"Category Error: %@", [error localizedDescription]);
  }
  if (![session setActive:YES error:&error]) {
      NSLog(@"Activation Error: %@", [error localizedDescription]);
  }
  return YES;

}
```

AVAudioSession提供了与应用程序音频会话交互的接口，所以开发者需要取得指向该单例的指针。通过设置合适的分类，开发者可为音频的播放指定需要的音频会话，其中定制一些行为。最后告知该音频会话激活该配置。

2.3　使用 AVAudioPlayer 播放音频

音频播放是很多应用程序的常见需求，AV Foundation让这一功能的实现变得非常简单，这一点要归功于一个名为AVAudioPlayer的类。这个类的实例提供了一种简单地从文本或内存中播放音频的方法。虽然接口很简单，不过它是具有很强功能的组件，并且在Mac和iOS系统中经常被作为实现音频播放的最佳选择。

AVAudioPlayer构建于Core Audio中的C-based Audio Queue Services的最顶层。所以它可以提供所有你在Audio Queue Services中所能找到的核心功能，比如播放、循环甚至音频计量，但使用的是非常简单并友好的Objective-C接口。除非你需要从网络流中播放音频、需要访问原始音频样本或者需要非常低的时延，否则AVAudioPlayer都能胜任。

2.3.1　创建 AVAudioPlayer

有两种方法可以创建一个AVAudioPlayer，使用包含要播放音频的内存版本的NSData，或者本地音频文件的NSURL。如果基于iOS系统，URL必须在应用程序沙盒之内或者该URL一

定是用户iPod库中的一个元素。下面看一个示例：

```
NSURL *fileURL =
    [[NSBundle mainBundle] URLForResource:@"rock" withExtension:@"mp3"];

// Must maintain a strong reference to player
self.player =
    [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL error:nil];

if (self.player) {
    [self.player prepareToPlay];
}
```

在上面的示例中，我们从应用程序bundle中包含的一个指向"rock.mp3"位置的URL创建了一个AVAudioPlayer实例。如果返回一个有效的播放实例，建议开发者调用其prepareToPlay方法。这样做会取得需要的音频硬件并预加载Audio Queue的缓冲区。调用prepareToPlay这个动作是可选的，当调用play方法时会隐性激活，不过在创建时准备播放器可以降低调用play方法和听到声音输出之间的延时。

2.3.2 对播放进行控制

播放实例包含了所有开发者期望的对播放行为进行控制的方法。调用play方法可以实现立即播放音频的功能，pause方法可以对播放暂停，那么可想而知stop方法可以停止播放行为。有趣的是，pause和stop方法在应用程序外面看来实现的功能都是停止当前播放行为。下一时间里我们调用play方法，通过pause和stop方法停止的音频都会继续播放。这两者最主要的区别在底层处理上。调用stop方法会撤消调用prepareToPlay时所做的设置，而调用pause方法则不会。

除了前面描述的标准常规方法之外，开发者还可以使用其他一些有趣的方法，如下所示：

- **修改播放器的音量**：播放器的音量独立于系统的音量，我们可以通过对播放器音量的处理实现很多有趣的效果，比如声音渐隐效果。音量或播放增益定义为 0.0(静音)到 1.0(最大音量)之间的浮点值。
- **修改播放器的 pan 值**，允许使用立体声播放声音：播放器的 pan 值由一个浮点数表示，范围从-1.0(极左)到 1.0(极右)。默认值为 0.0(居中)。
- **调整播放率**：iOS 5 版本中加入了一个强大功能，即允许用户在不改变音调的情况下调整播放率，范围从 0.5(半速)到 2.0(2 倍速)。如果正记录一首复杂的音乐或语音，放慢速度会有很大的帮助；当我们想快速浏览一份政府常规会议内容时，加速播放就很有帮助。
- **通过设置 numberOfLoops 属性实现音频无缝循环**：给这个属性设置一个大于 0 的数，可以实现播放器 n 次循环播放。相反，为该属性赋值-1 会导致播放器无限循环。

音频循环可以是未压缩的线性 PCM 音频，也可以是诸如 AAC 之类的压缩格式音频。MP3 文件片段可以实现无缝循环，但是 MP3 格式用作循环格式并不被推崇。MP3 格式的音频要实现循环的目的通常需要使用特殊工具进行处理。如果希望使用压缩格式的资源，建议使用诸如 AAC 或 AppleLossless 格式的内容。

- **进行音频计量**：当播放发生时从播放器读取音量力度的平均值及峰值。可将这些数据提供给 VU 计量器或其他可视化元件，向用户提供可视化的反馈效果。稍后讨论音频录制时，再做更详细的讲解。

如上所示，尽管AVAudioPlayer接口很简单，但仍功能强大，在进行音频播放时应该最先想到。下面深入了解AVAudioPlayer的细节，通过实际的应用来看看(听听)这些功能的具体用法。

2.4　创建 Audio Looper

本节会创建Audio Looper应用程序(如图2-1所示)，以便同步播放三个播放器实例，通过控制每个播放器的音量等级和立体声方面的pan值将这些声音混合，进而控制整体播放率。读者可在Chapter 2目录中找到名为AudioLooper_Starter的示例项目。

图 2-1　Audio Looper 应用程序

与本书中的其他示例应用程序一样，这个示例已经预先创建好一个带有所有动作和outlet的用户界面。管理音频播放器和控制其播放行为的代码写在一个名为THPlayerController的单独类中，这使得读者可以专注于AV Foundation的实现代码。代码清单2-1给出了针对这个类的接口。

代码清单 2-1　THPlayerControlier 接口

```
@interface THPlayerController : NSObject

@property (nonatomic, readonly, getter = isPlaying) BOOL playing;

// Global methods
- (void)play;
- (void)stop;
- (void)adjustRate:(float)rate;
```

```
// Player-specific methods
- (void)adjustPan:(float)pan forPlayerAtIndex:(NSUInteger)index;
- (void)adjustVolume:(float)volume forPlayerAtIndex:(NSUInteger)index;

@end
```

播放控制器定义了play、stop和adjustRate方法，一起对三个播放器实例的播放行为进行控制，还为每个播放器定义了诸如控制pan值和音量的几个方法。

如果你回头查看实现代码，可能会觉得对这个类的版本有些难懂。你会开始想从初始化阶段实现每一个细节。代码清单2-2给出了init方法和一个私有方法，这两个方法用于创建播放器实例。

代码清单 2-2 THPlayerController 初始化

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>

@interface THPlayerController ()
@property (nonatomic) BOOL playing;
@property (strong, nonatomic) NSArray *players;
@end

@implementation THPlayerController

- (instancetype)init {
    self = [super init];
    if (self) {
        AVAudioPlayer *guitarPlayer = [self playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self playerForFile:@"drums"];
        _players = @[guitarPlayer, bassPlayer, drumsPlayer];
    }
    return self;
}

- (AVAudioPlayer *)playerForFile:(NSString *)name {
    NSURL *fileURL = [[NSBundle mainBundle] URLForResource:name
                                            withExtension:@"caf"];
    NSError *error;
AVAudioPlayer *player = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL
                                                              error:&error];
    if (player) {
        player.numberOfLoops = -1; // loop indefinitely
        player.enableRate = YES;
        [player prepareToPlay];
    } else {
        NSLog(@"Error creating player: %@", [error localizedDescription]);
    }

    return player;
}
```

开发者一旦需要访问AV Foundation框架中的类，首先要导入<AVFoundation/AVFoundation.h>头文件。得益于Xcode引入了Modules机制，开发者不再需要手动建立框架关联，取而代之的是头文件将为相应的框架自动建立关联。

定义class extension并声明一个NSArray类型的属性用于保存播放器实例。这么做可以让开发者以组的方式操作这些实例，见稍后的讨论。

应用程序包含三个循环：guitar.caf、bass.caf和drums.caf。这些都是在虚拟乐队播放器中生成的4节拍声音文件。通过在init方法中调用私有的playerForFile:方法来为每个文件创建一个播放器实例。该方法会读取文件的URL并创建一个新的播放器实例。假设初始化过程是顺利的，应用程序会设置循环计数为-1，使得播放器无限循环播放，设置enableRate属性为YES可以对播放率进行控制，最后调用prepareToPlay启动播放。初始化就顺利完成了，下面继续研究播放方法(如代码清单2-3所示)。

代码清单 2-3　THPlayerController play 方法的实现

```
- (void)play {
    if (!self.playing) {
        NSTimeInterval delayTime = [self.players[0] deviceCurrentTime] + 0.01;
        for (AVAudioPlayer *player in self.players) {
            [player playAtTime:delayTime];
        }
        self.playing = YES;
    }
}
```

要对三个播放器实例的播放进行同步，需要捕捉当前设备时间并添加一个小延时，这样就会具有一个从开始播放时间计算的参照时间。通过对每个实例调用playAtTime:方法并传递延时参照时间，遍历播放器数组并开始播放。这就保证了这些播放器在音频播放时始终保持紧密同步。下面继续学习stop方法的实现过程，如代码清单2-4所示。

代码清单 2-4　THPlayerController stop 方法的实现

```
- (void)stop {
    if (self.playing) {
        for (AVAudioPlayer *player in self.players) {
            [player stop];
            player.currentTime = 0.0f;
        }
        self.playing = NO;
    }
}
```

stop方法很直接，如果音频正在播放，遍历播放器数组并对每一个对象调用stop方法。开发者可能还需要为设置每个播放器的currentTime属性为0.0f，这样做会让播放进度回到音频文件的原点。最终你需要更新playing状态来指明播放已经停止。

用户界面上具有一个高级播放率旋钮，可以让用户在不改变音调的前提下减慢或加快播放速度。速率旋钮调整的最小值为0.5，及减慢一半的速率；最大值为1.5，即以1.5倍速率播

放。adjustRate:方法接收传进来的数值并将其应用于每个播放器(如代码清单2-5所示)。

代码清单 2-5 THPlayerController adjustRate:方法的实现

```
- (void)adjustRate:(float)rate {
    for (AVAudioPlayer *player in self.players) {
        player.rate = rate;
    }
}
```

现在我们已经实现了所有的功能，开始运行应用程序吧。我们可以对内容进行播放和停止操作，并调整播放速率，三个播放器完美实现同步。

> **注意：**
> 双击界面上的旋钮可让它重新回到默认值。

控制单独音频

应用程序还支持用户对每个播放器调整音量和pan值。控件都已经配置好，只不过现在还没有实现具体功能，现在我们就为其添加具体功能。实现这些方法的代码如代码清单2-6所示。

代码清单 2-6 THPlayerController 的调节音量和 pan 值的方法

```
- (void)adjustPan:(float)pan forPlayerAtIndex:(NSUInteger)index {
    if ([self isValidIndex:index]) {
        AVAudioPlayer *player = self.players[index];
        player.pan = pan;
    }
}
- (void)adjustVolume:(float)volume forPlayerAtIndex:(NSUInteger)index {
    if ([self isValidIndex:index]) {
        AVAudioPlayer *player = self.players[index];
        player.volume = volume;
    }
}

- (BOOL)isValidIndex:(NSUInteger)index {
    return index == 0 || index < self.players.count;
}
```

代码中的“adjust”方法会接收一个浮点类型的调整值和一个用于表示所修改播放器的索引值。pan值的区间为-1.0(极左)到1.0(极右)，音量的区间为0.0(静音)到1.0(最大音量)。

再次运行该应用程序，现在可以带起你的耳机试着调整旋钮感受一下了。

Audio Looper应用程序是应用AVAudioPlayer非常好的示例。我们实现了一组复杂功能，不过只用了少量代码就实现了这些功能。如果使用底层Audio Queue Services来实现相同的功能，那么现在我们的手指和大脑肯定都已经筋疲力尽了。

2.5　配置音频会话

上面的Audio Looper看起来已经足够好了，不过我们还需要在更大范围的iOS音频环境下对其进行几组测试，以确保应用程序能够良好运行。如果你还没有这样做，现在就在设备上编译并部署这个应用程序。开始播放音频并切换设备侧面的“铃音/静音”开关，可以听到音频输出在两种状态下切换。第二个测试，当音频正在播放时，点击设备上的Lock按钮，你应该听到声音逐渐消失。由于应用程序的核心功能是播放音频资源，这两个动作都是我们不希望的。这又回到有关音频会话的主题上。

如果你回顾上一节中对AVAudioSession的讨论，会发现所有iOS应用程序都自动带有一个默认音频会话，分类名称为Solo Ambient。这个类型可以播放音频，但对于一个主要功能为音频播放的应用程序来说并不合适。由于默认分类不能提供我们期望的功能，需要对音频会话进行更明确的配置。音频会话通常会在应用程序启动时进行一次配置，所以可将配置代码写在THAppDelegate的application:didFinishLaunchingWithOptions:方法中。如代码清单2-7所示，对Audio Looper应用程序的音频会话进行了合理配置。

代码清单 2-7　THAppDelegate 音频会话设置

```
#import "THAppDelegate.h"
#import <AVFoundation/AVFoundation.h>

@implementation THAppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    AVAudioSession *session = [AVAudioSession sharedInstance];

    NSError *error;
    if (![session setCategory:AVAudioSessionCategoryPlayback error:&error]) {
        NSLog(@"Category Error: %@", [error localizedDescription]);
    }
    if (![session setActive:YES error:&error]) {
        NSLog(@"Activation Error: %@", [error localizedDescription]);
    }

    return YES;
}

@end
```

我们首先处理AVAudioSession单例，由于播放音频是应用程序最主要的功能，需要指定AVAudioSessionCategoryPlayback分类。最后通过调用setAction:error:方法并传递YES值来激活会话。在这个示例中，我们只是简单地将出现的错误打印出来，读者朋友记得在实际的产品代码中要对错误进行适当处理。

当音频会话正确配置完毕后，重新在你的设备上部署应用程序并运行测试。你会发现切换“铃音/静音”开关不能让声音消失。这个行为看起来不错，那么下面实现在锁定设备时能够继续播放音频的功能。如果按下Lock按钮，可以听见音乐持续播放，那么说明是有问题的。

我们遇到了与前面相同的情况。是不是应该将分类设置成AVAudioSessionCategoryPlayback呢？答案是否定的。设置成上述分类可以让应用程序在后台播放音频，这是设备锁定时所处的状态，但是我们仍需对应用程序的Info.plist文件进行细微修改来实现这个功能。

在Xcode Project Navigator面板找到应用程序的Info.plist文件，用Xcode属性列表编辑器打开。添加一个新的Required background modes类型的数组，在其中添加名为App plays audio or streams audio/video using AirPlay的选项(如图2-2所示)。

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development r...	String	en
Bundle display name	String	${PRODUCT_NAME}
Executable file	String	${EXECUTABLE_NAME}
Bundle identifier	String	com.tapharmonic.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone envir...	Boolean	YES
▼ Required background modes	Array	(1 item)
Item 0	String	App plays audio or streams audio/video using AirPlay
Main storyboard file base name	String	App plays audio or streams audio/video using AirPlay
▶ Required device capabilities	Array	App registers for location updates
▶ Supported interface orientations	Array	App provides Voice over IP services

图 2-2 设置 Required background modes

另外，也可以右击ProjectNavigator中的Info.plist文件，在相应的XML部分编辑plist，依次选择Open As | Source Code。将下面条目添加到文件底部的</dict>标签前：

```
<key>UIBackgroundModes</key>
<array>
    <string>audio</string>
</array>
```

添加这一设置表示应用程序现在允许在后台播放音频内容。再次编译并部署应用程序，再次播放。现在如果按下设备上的Lock按钮，可以听到音频仍然可以从后台播放出来。太棒了！

2.6 处理中断事件

上面完成了Audio Looper应用程序的核心功能，现在要考虑如何完善用户体验了。需要改进的地方是我们要确保应用程序可以正确地处理中断事件。中断在iOS设备中经常出现，在使用设备的过程中经常会有诸如电话呼入、闹钟响起及弹出FaceTime请求等情况。虽然iOS系统本身可以很好地处理这些事件，不过我们还是要确保“我们自己”针对这些情况的处理也足够完美。

下面做几个测试来查看Audio Looper应用程序面对中断事件时是如何响应的。要进行测试，我们还需要一部iPhone来拨打电话，或者一台能够发起FaceTime请求的iOS设备或Mac。按照下面的步骤进行操作：

(1) 在设备上运行Audio Looper应用程序并播放音频。

(2) 当音频处于播放状态时，从另一台设备上向当前设备发起电话呼叫或FaceTime呼叫

以制造中断。

(3) 在另一台设备上按Decline按钮终止呼叫或FaceTime请求。

如果可以按上述场景进行测试，会发现一些有趣的行为。当中断发生时，播放中的音频会慢慢消失和暂停。这一效果是自动实现的，我们没有编写任何代码。不过当我们点击另一台设备上的Decline按钮以终止中断时，会发现一些问题。Play/Stop按钮处于不可用状态，音频播放也没有如预期般恢复。显然这不是我们所希望的效果，下面来看如何解决这些问题。

音频会话通知

在准备为出现的中断事件采取动作前，首先需要得到中断出现的通知，注册应用程序的AVAudioSession发送的通知AVAudioSessionInterruptionNotification。在控制器的init方法中注册该通知，如代码清单2-8所示。

代码清单 2-8　注册中断通知

```
- (instancetype)init {
self = [super init];
if (self) {
    AVAudioPlayer *guitarPlayer = [self playerForFile:@"guitar"];
    AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
    AVAudioPlayer *drumsPlayer = [self playerForFile:@"drums"];

    _players = @[guitarPlayer, bassPlayer, drumsPlayer];

    NSNotificationCenter *nsnc = [NSNotificationCenter defaultCenter];

    [nsnc addObserver:self
             selector:@selector(handleInterruption:)
                 name:AVAudioSessionInterruptionNotification
               object:[AVAudioSession sharedInstance]];
    }
    return self;
}
- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
- (void)handleInterruption:(NSNotification *)notification {
}
```

推送的通知会包含一个带有许多重要信息的userInfo字典，根据这个字典可以确定采取哪些合适的操作。

在handleInterruption:方法中，首先通过检索AVAudioSessionInterruptionTypeKey的值确定中断类型(type)。返回值是AVAudioSessionInterruptionType，这是用于表示中断开始或结束的枚举类型(如代码清单2-9所示)。

代码清单 2-9 确定中断类型

```
- (void)handleInterruption:(NSNotification *)notification {

    NSDictionary *info = notification.userInfo;

    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey] unsignedIntegerValue];

    if (type == AVAudioSessionInterruptionTypeBegan) {
        // Handle AVAudioSessionInterruptionTypeBegan
    } else {
        // Handle AVAudioSessionInterruptionTypeEnded
    }
}
```

当中断出现时，需要采取的一个动作就是正确更新Play/Stop按钮的状态及相关的标签。THPlayerController不对用户界面进行管理，所以你需要一个方法将这些中断事件通知给视图控制器。如果你查看THPlayerController头文件的开头，会看到如代码清单2-10所示的协议。

代码清单 2-10 协议

```
@protocol THPlayerControllerDelegate <NSObject>
- (void)playbackStopped;
- (void)playbackBegan;
@end
```

应用程序的视图控制器已经采用该协议并将其设置为委托。这提供了一种简洁的方法来更新应用程序的用户界面。基于上面的讲解，下面开始处理AVAudioSessionInterruptionTypeBegan类型，如代码清单2-11所示。

代码清单 2-11 开始处理中断

```
- (void)handleInterruption:(NSNotification *)notification {

    NSDictionary *info = notification.userInfo;

    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey] unsignedIntegerValue];

    if (type == AVAudioSessionInterruptionTypeBegan) {
        [self stop];
        if (self.delegate) {
            [self.delegate playbackStopped];
        }
    } else {
        // Handle interruption ended
    }
}
```

我们调用了stop方法，并通过调用委托函数的playbackStopped方法向委托通知中断状态。很重要的一点是当通知被接收时，音频会话已经被终止，且AVAudioPlayer实例处于暂停状态。

调用控制器的stop方法只能更新内部状态，并不能停止播放。

如果中断类型为AVAudioSessionInterruptionTypeEnded，userInfo字典会包含一个AVAudioSessionInterruptionOptions值来表明音频会话是否已经重新激活以及是否可以再次播放(如代码清单2-12所示)。

代码清单 2-12　处理中断结束

```
- (void)handleInterruption:(NSNotification *)notification {

    NSDictionary *info = notification.userInfo;

    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey] unsignedIntegerValue];

    if (type == AVAudioSessionInterruptionTypeBegan) {
        [self stop];
        if (self.delegate) {
            [self.delegate playbackStopped];
        }
    } else {
         AVAudioSessionInterruptionOptions options =
             [info[AVAudioSessionInterruptionOptionKey] unsignedIntegerValue];
         if (options == AVAudioSessionInterruptionOptionShouldResume) {
             [self play];
             if (self.delegate) {
                 [self.delegate playbackBegan];
             }
         }
    }
}
```

如果上述代码中options的值为AVAudioSessionInterruptionOptionShouldResume，需要调用控制器的play方法并通过调用自身的playbackBegan方法通知委托。

再次编译和部署应用程序并重新执行之前的中断测试。现在当中断开始时，播放器的用户界面会正确地更新；当应用程序的控制权返回应用程序时，会继续播放音频并且一切正常。

2.7　对线路改变的响应

在结束AVAudioPlayer知识讲解前，最后一个需要了解的领域就是如何确保应用程序对线路变换做出正确的响应。在iOS设备上添加或移除音频输入、输出线路时，会发生线路改变。有多重原因会导致线路的变化，比如用户插入耳机或断开USB麦克风。当这些事件发生时，音频会根据情况改变输入或输出线路，同时AVAudioSession会广播一个描述该变化的通知给所有相关的侦听器。为遵循Apple的Human Interface Guidelines(HIG)的相关定义，应用程序应该成为这些相关侦听器中的一员。

下面对Audio Looper应用程序再做一个测试。开始播放，并在播放期间插入耳机。音频输出线路变为耳机插孔并继续正常播放，这正是我们所期望的效果。保持音频处于播放状态，

断开耳机连接。音频线路再次回到设备的内置扬声器，我们再次听到了声音。虽然线路变化同预期一样，不过按照苹果公司的相关文档，该音频应该处于静音状态。当用户插入耳机时，隐含的意思是用户不希望外界听到具体的音频内容。这就意味着当用户断开耳机时，播放的内容可能需要继续保密，所以我们需要停止音频播放。那么应该怎么做呢？

当线路发生变化时要有通知，那么首先我们需要注册AVAudioSession发送的通知，该通知名为AVAudioSessionRouteChangeNotification。该通知包含一个userInfo字典，该字典带有相应通知发送的原因信息及前一个线路的描述，这样我们就可以确定线路变化的情况了。

收这些变化通知前，需要为这些通知进行注册。与注册中断通知的情况一样，在THPlayerController的init方法中实现注册过程(如代码清单2-13所示)。

代码清单 2-13 注册线路变化通知

```
- (instancetype)init {
    self = [super init];
    if (self) {
    AVAudioPlayer *guitarPlayer = [self playerForFile:@"guitar"];
    AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
    AVAudioPlayer *drumsPlayer = [self playerForFile:@"drums"];

    _players = @[guitarPlayer, bassPlayer, drumsPlayer];

    NSNotificationCenter *nsnc = [NSNotificationCenter defaultCenter];

    [nsnc addObserver:self
              selector:@selector(handleInterruption:)
                  name:AVAudioSessionInterruptionNotification
                object:[AVAudioSession sharedInstance]];

    [nsnc addObserver:self
              selector:@selector(handleRouteChange:)
                  name:AVAudioSessionRouteChangeNotification
                object:[AVAudioSession sharedInstance]];
    }
    return self;
}

- (void)handleRouteChange:(NSNotification *)notification {

}
```

接收到通知后要做的第一件事是判断线路变更发生的原因。查看保存在userInfo字典中的表示原因的AVAudioSessionRouteChangeReasonKey值。这个返回值是一个用于表示变化原因的无符号整数。通过原因可以推断出不同的事件，比如有新设备接入或改变音频会话类型，不过我们需要特殊注意的是耳机断开这个事件。这个事件对应的原因为AVAudioSession-RouteChangeReasonOldDeviceUnavailable(如代码清单2-14所示)。

代码清单 2-14 判断通知原因

```
- (void)handleRouteChange:(NSNotification *)notification {

    NSDictionary *info = notification.userInfo;
```

```
    AVAudioSessionRouteChangeReason reason =
        [info[AVAudioSessionRouteChangeReasonKey] unsignedIntValue];

    if (reason == AVAudioSessionRouteChangeReasonOldDeviceUnavailable) {

    }
}
```

知道有设备断开连接后，需要向userInfo字典提出请求，以获得其中用于描述前一个线路的AVAudioSessionRouteDescription。线路的描述信息整合在一个输入NSArray和一个输出NSArray中。数组中的元素都是AVAudioSessionPortDescription的实例，用于描述不同的I/O接口属性。在上述情况下，你需要从线路描述中找到第一个输出接口并判断其是否为耳机接口。如果是，则停止播放，并调用委托函数的playbackStopped方法(如代码清单2-15所示)。

代码清单 2-15　耳机断开时停止播放

```
- (void)handleRouteChange:(NSNotification *)notification {
    NSDictionary *info = notification.userInfo;
    AVAudioSessionRouteChangeReason reason =
        [info[AVAudioSessionRouteChangeReasonKey] unsignedIntValue];

    if (reason == AVAudioSessionRouteChangeReasonOldDeviceUnavailable) {

        AVAudioSessionRouteDescription *previousRoute =
            info[AVAudioSessionRouteChangePreviousRouteKey];

        AVAudioSessionPortDescription *previousOutput = previousRoute.outputs[0];
        NSString *portType = previousOutput.portType;

        if ([portType isEqualToString:AVAudioSessionPortHeadphones]) {
            [self stop];
            [self.delegate playbackStopped];
        }
    }
}
```

将最新更改编译和部署到设备上，并测试最新的行为。现在当我们断开耳机时，音频播放会如期望般停止。

当开发者创建媒体应用程序时，处理线路变化和正确对中断做出响应是常见的核心问题。为正确处理各场景而做出的一些努力会在将来用户具体使用时给应用程序带来极大帮助。

Audio Looper应用程序现在全部完成了，下面将注意力转移到AVAudioPlayer的同伴上，即AVAudioRecorder。

2.8　使用 AVAudioRecorder 录制音频

你已经了解在AV Foundation中使用AVAudioPlayer类向应用程序添加音频播放功能是多么简单。同样在AV Foundation中使用AVAudioRecorder类添加音频录制功能也很简单。AVAudioRecorder同其用于播放音频的兄弟类一样，构建于Audio Queue Services之上，是一个

功能强大且代码简单易用的Objective-C接口。我们可以在Mac机器和iOS设备上使用这个类来从内置的麦克风录制音频，也可从外部音频设备进行录制，比如数字音频接口或USB麦克风等。

2.8.1 创建 AVAudioRecorder

创建AVAudioRecorder实例时需要为其提供数据的一些信息，分别是：

- 用于表示音频流写入文件的本地文件 URL。
- 包含用于配置录音会话键值信息的 NSDictionary 对象。
- 用于捕捉初始化阶段各种错误的 NSError 指针。

上述条件的具体设置代码如下所示：

```
NSString *directory = // output directory
NSString *filePath = [directory stringByAppendingPathComponent:@"voice.m4a"];
NSURL *url = [NSURL fileURLWithPath:filePath];

NSDictionary *settings = @{AVFormatIDKey : @(kAudioFormatMPEG4AAC),
                           AVSampleRateKey : @22050.0f,
                           AVNumberOfChannelsKey : @1};
NSError *error;
// Must maintain a strong reference to player
self.recorder =
    [[AVAudioRecorderalloc] initWithURL:urlsettings:settings error:&error];
if (self.recorder) {
    [self.recorderprepareToRecord];
} else {
    // Handle error
}
```

为成功创建 AVRecorder 实例，建议调用其 prepareToRecord 方法。与 AVPlayer 的 prepareToPlay方法类似，这个方法执行底层Audio Queue初始化的必要过程。该方法还在URL参数指定的位置创建一个文件，将录制启动时的延时降到最小。

在设置字典中指定的键值信息也值得讨论一番。开发者可以使用的完整可用键信息在<AVFoundation/AVAudioSettings.h>中定义。大部分的键都专门定义了特有的格式，不过下面介绍的都是一些通用的音频格式。

1. 音频格式

AVFormatIDKey键定义了写入内容的音频格式，下面的常量都是音频格式所支持的值：

```
kAudioFormatLinearPCM
kAudioFormatMPEG4AAC
kAudioFormatAppleLossless
kAudioFormatAppleIMA4
kAudioFormatiLBC
kAudioFormatULaw
```

指定kAudioFormatLinearPCM会将未压缩的音频流写入到文件中。这种格式的保真度最

高，不过相应的文件也最大。选择诸如AAC(kAudioFormatMPEG4AAC)或Apple IMA4(kAudioFormat-AppleIMA4)的压缩格式会显著缩小文件，还能保证高质量的音频内容。

> **注意：**
>
> 你所指定的音频格式一定要和 URL 参数定义的文件类型兼容。比如，如果录制一个名为 test.wav 的文件，隐含的意思就是录制的音频必须满足 Waveform Audio File Format(WAVE) 的格式要求，即低字节序、Linear PCM。为 AVFormatIDKey 值指定除 kAudioFormatLinearPCM 之外的值会导致错误。查询 NSError 的 localizedDescription 会返回下面的错误信息描述：
>
> ```
> The operation couldn't be completed. (OSStatus error 1718449215.)
> ```
>
> 1718449215 错误状态是 4 字节编码的整数值，'fmt?'意指定义了一种不兼容的音频格式。

2. 采样率

AVSampleRateKey用于定义录音器的采样率。采样率定义了对输入的模拟音频信号每一秒内的采样数。在录制音频的质量及最终文件大小方面，采样率扮演着至关重要的角色。使用低采样率，比如8kHz，会导致粗粒度、AM广播类型的录制效果，不过文件会比较小；使用44.1kHz的采样率(CD质量的采样率)会得到非常高质量的内容，不过文件就比较大。对于使用什么采样率最好没有一个明确的定义，不过开发者应该尽量使用标准的采样率，比如8 000、16 000、22 050或44 100。最终是我们的耳朵在进行判断。

3. 通道数

AVNumberOfChannelsKey用于定义记录音频内容的通道数。指定默认值1意味着使用单声道录制，设置为2意味着使用立体声录制。除非使用外部硬件进行录制，否则通常应该创建单声道录音。

4. 指定格式的键

处理Linear PCM或压缩音频格式时，可以定义一些其他指定格式的键。可在Xcode帮助文档中的AV Foundation Audio Settings Constants引用中找到完整的列表。

2.8.2　控制录音过程

创建了一个可用的录音器实例后，就可以开始录音了。AVAudioRecorder包含一些方法可以支持无限时长的录制，比如在未来某一时间点开始录制或录制指定时长的内容等。开发者甚至可以暂停录制并在之后从这个停止的地方继续重启录制。如此灵活的功能非常方便，也使得AVAudioRecorder成为Mac和iOS平台上很多标准录音的首选方法。

现在我们基本了解了AVAudioRecorder，我们一起构建一个简单的语音备忘应用程序来体会这些功能的实际用法。

2.9 创建 Voice Memo 应用程序

Voice Memo应用程序(如图2-3所示)的基本功能是让用户记录一段语音备忘，支持在录制过程中暂停，并可以保存多条记录以备之后播放。在实现了这些基本功能后，我们再考虑如何改善用户体验。可在Chapter 2目录中找到一个名为VoiceMemo_Starter的启动项目。

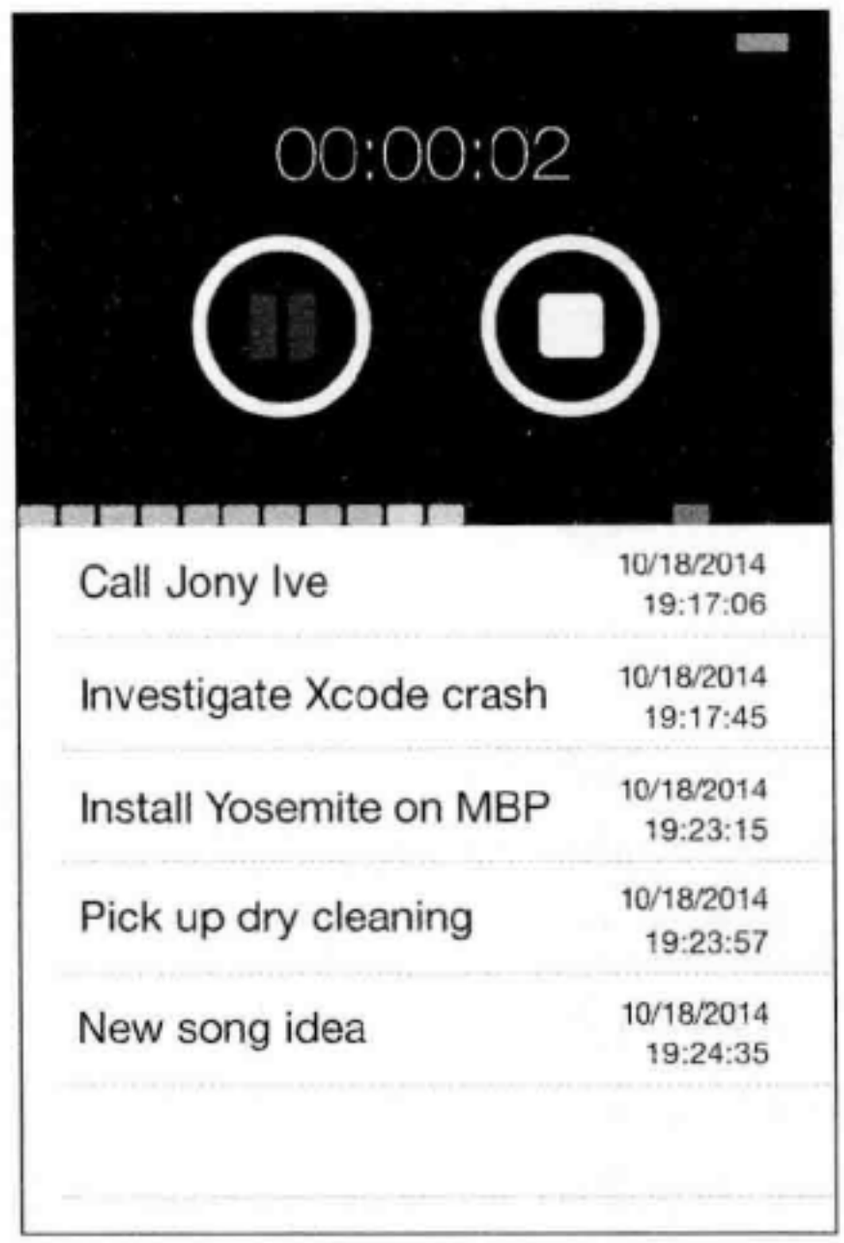

图 2-3 Voice Memo 应用程序

该应用程序已经预配置好了基本的用户界面，只是尚未实现具体的功能。我们会在THRecorderController类中实现核心录制行为，但在此之前，下面先对音频会话的属性进行配置使其支持录音。

2.9.1 配置音频会话

该应用程序的核心功能是录制并播放语音备忘录。我们知道不能使用默认的Solo Ambient分类(AVAudioSessionCategorySoloAmbient)，因为该分类不支持音频输入。由于我们既需要录音又需要对外播放，那么合适的分类应该是AVAudioSessionCategoryPlayAndRecord。在应用程序委托THAppDelegate中配置音频会话，并使用上述分类，如代码清单2-16所示。

代码清单 2-16 THAppDelegate 音频会话设置

```
- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

  AVAudioSession *session = [AVAudioSession sharedInstance];

  NSError *error;
  if (![session setCategory:AVAudioSessionCategoryPlayAndRecord error:&error]) {
    NSLog(@"Category Error: %@", [error localizedDescription]);
  }
```

```
    if (![session setActive:YES error:&error]) {
      NSLog(@"Activation Error: %@", [error localizedDescription]);
    }

    return YES;
}
```

从iOS 7开始，在使用麦克风之前，操作系统要求应用程序必须得到用户的明确许可。当应用程序试图访问麦克风时，操作系统会为用户弹出如图2-4所示的对话框。

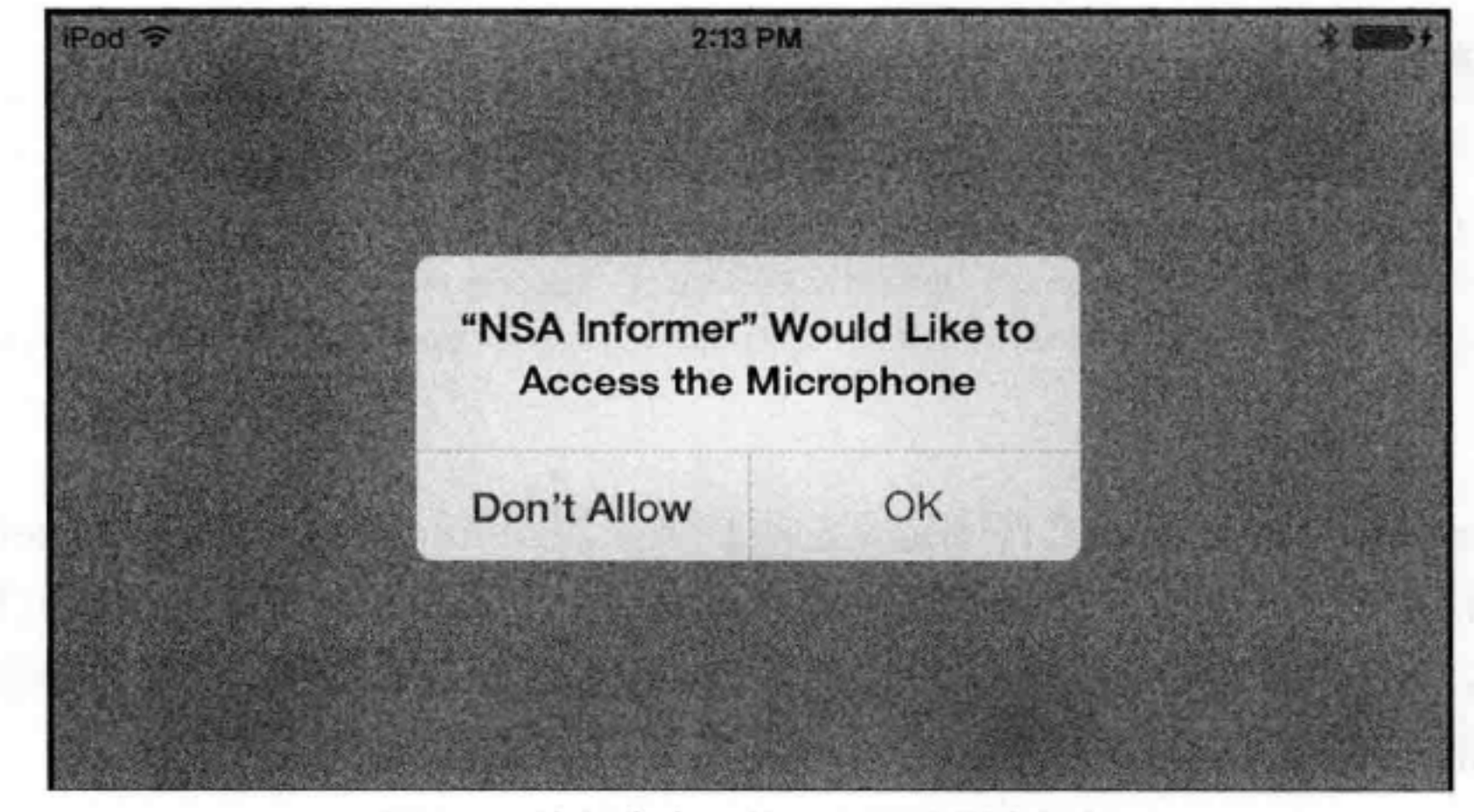

图 2-4　访问麦克风前 iOS 要求用户授权

如果用户选择“Don’t Allow”按钮，那么所有试图录制的内容都是没有声音的。如果用户改变了想法，允许应用程序录制，需要改变一些设置，具体方法为在Setting应用程序中设置Privacy | Microphone settings。

现在已经配置好音频会话，开始实现录音功能吧。

2.9.2　实现录音功能

代码清单2-17给出了THRecorderController类的接口。

代码清单 2-17　THRecorderController 接口

```
typedef void(^THRecordingStopCompletionHandler)(BOOL);
typedef void(^THRecordingSaveCompletionHandler)(BOOL, id);

@class THMemo;

@interface THRecorderController : NSObject

@property (nonatomic, readonly) NSString *formattedCurrentTime;

// Recorder methods
- (BOOL)record;

- (void)pause;

- (void)stopWithCompletionHandler:(THRecordingStopCompletionHandler)handler;
```

```
- (void)saveRecordingWithName:(NSString *)name
            completionHandler:(THRecordingSaveCompletionHandler)handler;
// Player methods
- (BOOL)playbackMemo:(THMemo *)memo;

@end
```

该类定义了核心的传输操作来控制整个录制周期，并定义了方法来管理之前录制完毕的备忘信息的播放。下面看一下该类的实现代码并开始实现这些方法，如代码清单2-18所示。

代码清单 2-18 THRecorderController 类扩展

```
@interface THRecorderController () <AVAudioRecorderDelegate>

@property (strong, nonatomic) AVAudioPlayer *player;
@property (strong, nonatomic) AVAudioRecorder *recorder;
@property (strong, nonatomic) THRecordingStopCompletionHandler completionHandler;

@end
```

在THRecorderController.m文件中定义的类扩展采用AVAudioRecorderDelegate协议。录音器的委托协议定义了一个方法，当录制完成时会发送一个通知。所以得到录制过程完成的信息非常重要，这样你才能让用户为这段录音命名并保存起来。下面继续研究控制器的init方法是如何实现的，如代码清单2-19所示。

代码清单 2-19 THRecorderController init 方法

```
- (instancetype)init {
    self = [super init];
    if (self) {
        NSString *tmpDir = NSTemporaryDirectory();
        NSString *filePath = [tmpDir stringByAppendingPathComponent:@"memo.caf"];
        NSURL *fileURL = [NSURL fileURLWithPath:filePath];

        NSDictionary *settings = @{
            AVFormatIDKey : @(kAudioFormatAppleIMA4),
            AVSampleRateKey : @44100.0f,
            AVNumberOfChannelsKey : @1,
            AVEncoderBitDepthHintKey : @16,
            AVEncoderAudioQualityKey : @(AVAudioQualityMedium)
        };
        NSError *error;
        self.recorder = [[AVAudioRecorder alloc] initWithURL:fileURL
                                                   settings:settings
                                                       error:&error];
        if (self.recorder) {
            self.recorder.delegate = self;
            [self.recorder prepareToRecord];
        } else {
            NSLog(@"Error: %@", [error localizedDescription]);
        }
    }
```

```
    return self;
}
```

在init方法中我们对录音器进行了配置，我们记录到tmp目录中一个名为memo.caf的文件。在录制音频的过程中，Core Audio Format(CAF)通常是最好的容器格式，因为它和内容无关并可以保存Core Audio支持的任何音频格式。你需要定义录音设置，以便使用Apple IMA4作为音频格式，采样率为44.1kHz，位深为16位，单声道录制。这些设置考虑了质量和文件大小的平衡。

录音器实例就这样创建完成了，下面继续看一下代码清单2-20中所示的多种传输方法。

代码清单 2-20　THRecorderController 传输方法

```
- (BOOL)record {
    return [self.recorder record];
}

- (void)pause {
    [self.recorder pause];
}

- (void)stopWithCompletionHandler:(THRecordingStopCompletionHandler)handler {
   self.completionHandler = handler;
   [self.recorder stop];
}

- (void)audioRecorderDidFinishRecording:(AVAudioRecorder *)recorder
    successfully:(BOOL)success {
    if (self.completionHandler) {
        self.completionHandler(success);
    }
}
```

这些方法的实现很直接，因为我们将其委托给录音器实例来完成这些工作。不过stopWithCompletionHandler:需要做一下解释。当用户点击Stop按钮时，会调用该方法并传递一个完成块。我们保存对这个块的引用并调用录音器实例的stop方法，录音器实例启动结束音频录制的过程。当音频录制完毕后，录音器调用其委托方法，此时需要执行完成块来告知发起方，以便其采取适当的操作。这种情况下，视图控制器会弹出一个警告框给用户，让用户为录制的内容命名并保存。下面看一下代码清单2-21所示的保存操作。

代码清单 2-21　THRecorderController 保存方法

```
- (void)saveRecordingWithName:(NSString *)name
          completionHandler:(THRecordingSaveCompletionHandler)handler {

   NSTimeInterval timestamp = [NSDate timeIntervalSinceReferenceDate];
   NSString *filename =
       [NSString stringWithFormat:@"%@-%f.caf", name, timestamp];

   NSString *docsDir = [self documentsDirectory];
   NSString *destPath = [docsDir stringByAppendingPathComponent:filename];
```

```
    NSURL *srcURL = self.recorder.url;
    NSURL *destURL = [NSURL fileURLWithPath:destPath];

    NSError *error;
    BOOL success = [[NSFileManager defaultManager] copyItemAtURL:srcURL
                                                          toURL:destURL
                                                          error:&error];
    if (success) {
        handler(YES, [THMemo memoWithTitle:name url:destURL]);
        [self.recorder prepareToRecord];
    } else {
        handler(NO, error);
    }
}

- (NSString *)documentsDirectory {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                         NSUserDomainMask, YES);

    return [paths objectAtIndex:0];
}
```

当用户停止录音，进入对语音备忘命名阶段时，视图控制器会调用saveRecording-WithName:completionHandler:方法。该方法通过唯一文件名将tmp目录中的录制文件复制到Documents目录。如果复制操作成功，会调用完成块，传递回一个新的包含名字和录制音频URL的THMemo实例。之后视图控制器代码使用THMemo实例来创建录制好的语音备忘文件列表。

现在可以编译并运行该应用程序，观察实际操作中这些核心功能的实现效果。可以录制、暂停和继续录制、停止录制并保存到列表中。下面看一下最后还有哪些功能没有实现。

虽然现在能够成功录制音频和创建结果列表，但是还不能回过来播放这些内容。我们之前定义了一个playbackMemo:方法，但尚未实现该功能，现在就来实现吧。实现播放功能会用到前面的AVAudioPlayer(如代码清单2-22所示)。

代码清单 2-22 THRecorderController 播放方法

```
- (BOOL)playbackMemo:(THMemo *)memo {
    [self.player stop];
    self.player = [[AVAudioPlayer alloc] initWithContentsOfURL:memo.url
                                                        error:nil];
    if (self.player) {
        [self.player play];
        return YES;
    }
    return NO;
}
```

首先停止现有的播放器(如果存在播放器的话)。我们通过保存在THMemo中的URL创建一个新的AVAudioPlayer实例，调用播放器的play方法。本例忽略所有初始化过程中出现的错误，但在你实际编写代码时，一定要注意应用程序的可靠性。

播放功能做好后，最后一个尚未完成的功能就是展示播放时间。用户界面上有一个标签用于展示时间，不过现在这个标签的内容始终显示00:00:00，还是无法令人信服的功能。AVAudioRecorder具有的currentTime属性可以简单地构建一个用户界面，为用户提供时间反馈信息。该属性返回一个NSTimeInterval，用于指示从录制开始到现在的时间，以秒来计算。NSTimeInterval不适合在用户界面上展示，不过我们可以通过一些处理让其能够美观地展示。下面看一下formattedCurrentTime方法的实现代码(如代码清单2-23所示)。

代码清单 2-23　THRecorderController formattedCurrentTime 方法

```
- (NSString *)formattedCurrentTime {

    NSUInteger time = (NSUInteger)self.recorder.currentTime;
    NSInteger hours = (time / 3600);
    NSInteger minutes = (time / 60) % 60;
    NSInteger seconds = time % 60;

    NSString *format = @"%02i:%02i:%02i";
    return [NSString stringWithFormat:format, hours, minutes, seconds];
}
```

首先我们针对录音器对象读取currentTime，返回值是一个双精度类型的NSTimeInterval。由于我们不需要双精度这么精确的数据，需要将时间保存为一个NSUInteger。之后计算当前时间有效的小时、分钟和秒值，并创建一个HH:mm:ss格式的NSString。

你可能很好奇如何使用这个时间。该方法会返回一个用于呈现的完美格式字符串，但是它只是一个时间点。那么如何随着时间的推移实时更新展示呢？首先想到对currentTime属性使用Key-Value Observing(KVO)。虽然你可能经常在AV Foundation中使用KVO，不过在AVAudioRecorder(和AVAudioPlayer)的代码中是不可以使用的，因为currentTime属性是不可见的。你可以改用NSTimer并按照一定的时间间隔轮询值。代码清单2-24给出了应用程序的MainViewController类中的代码实现，设置了一个新的NSTimer实例并每半秒一次。

代码清单 2-24　MainViewController 时间轮询

```
- (void)startTimer {
    [self.timer invalidate];
    self.timer = [NSTimer timerWithTimeInterval:0.5
                                         target:self
                                       selector:@selector(updateTimeDisplay)
                                       userInfo:nil
                                        repeats:YES];
    [[NSRunLoop mainRunLoop] addTimer:self.timer forMode:NSRunLoopCommonModes];
}
- (void)updateTimeDisplay {
    self.timeLabel.text = self.controller.formattedCurrentTime;
}
```

运行应用程序并开始录制一段新的音频。时间现在可以更新了，展示当前录制的时间。如果点击Pause按钮，计时会暂停，再次点击Record后会从上次的停止时间开始继续计时。这

是一次非常好的用户体验，告诉用户录制正在进行，并使用户知道已经录制了多长时间。虽然展示时间可以表示录音正在进行，但没有告诉用户是否真的有声音在被录制。如果能将捕捉到的声音信号以可视化方式呈现出来就更好了。幸运的是，我们可以通过Metering(声音计量)来实现这个功能。

2.10 使用 Audio Metering

AVAudioRecorder和AVAudioPlayer中最强大和最实用的功能就是对音频进行测量。Audio Metering可让开发者读取音频的平均分贝和峰值分贝数据，并使用这些数据以可视化方式将声音的大小呈现给最终用户。

两个类使用的方法都是averagePowerForChannel:和peakPowerForChannel: 。两个方法都会返回一个用于表示声音分贝(dB)等级的浮点值。这个值的范围从表示最大分贝的0Db(full scale)到表示最小分贝或静音的-160dB。

在可以读取这些值之前，首先要通过设置录音器的meteringEnabled属性为YES才可以支持对音频进行测量。这就使得录音器可以对捕捉到的音频样本进行分贝计算。每当需要读取值时，首先需要调用updateMeters方法才能获取最新的值。

在Voice Memo应用程序中，我们需要在AVAudioRecorder实例上进行音频数据的测量，并通过结果数据为用户展示一个简单的可视化效果图。通过averagePowerForChannel:和peakPowerForChannel:两个方法提供的读数，可以得到一个表示分贝等级的浮点值，这是一个描述音量等级的对数单位。示例应用程序具有一个基于Quartz级别的计量视图，用于展示录制过程中的平均分贝和峰值分贝，不过在使用这个视图前，分贝值需要先从对数形式的-160到0范围转换为线性的0到1形式。每次请求这个计量数据，都需要做上述转换，不过更好的解决方案是只计算一次这些变化，之后按照需要进行查找。在示例应用程序中，我们找到一个名为THMeterTable(如代码清单2-25所示)的类。这是一个简化类，是基于Apple的C++的MeterTable类的Objective-C端口，在很多示例项目中都会用到这个类。

代码清单 2-25 THMeterTable 的实现

```
#import "THMeterTable.h"

#define MIN_DB -60.0f
#define TABLE_SIZE 300

@implementation THMeterTable {
    float _scaleFactor;
    NSMutableArray *_meterTable;
}

- (instancetype)init {
    self = [super init];
    if (self) {
        float dbResolution = MIN_DB / (TABLE_SIZE - 1);

        _meterTable = [NSMutableArray arrayWithCapacity:TABLE_SIZE];
        _scaleFactor = 1.0f / dbResolution;
```

```
        float minAmp = dbToAmp(MIN_DB);
        float ampRange = 1.0 - minAmp;
        float invAmpRange = 1.0 / ampRange;

        for (int i = 0; i < TABLE_SIZE; i++) {
        float decibels = i * dbResolution;
        float amp = dbToAmp(decibels);
        float adjAmp = (amp - minAmp) * invAmpRange;
        _meterTable[i] = @(adjAmp);
    }
}
  return self;
}

float dbToAmp(float dB) {
    return powf(10.0f, 0.05f * dB);
}

- (float)valueForPower:(float)power {
    if (power < MIN_DB) {
        return 0.0f;
    } else if (power >= 0.0f) {
       return 1.0f;
    } else {
        int index = (int) (power * _scaleFactor);
        return [ _meterTable[index] floatValue];
    }
}

@end
```

该类创建了一个内部数组，用于保存从计算前的分贝数到使用一定级别分贝解析之后的转换结果。这里使用的解析率为-0.2dB。解析等级可以通过修改MIN_DB和TABLE_SIZE值进行调整。

每个分贝值都通过调用dbToAmp函数转换为线性范围内的值，使其处于范围0(-60dB)到1之间，之后得到一条由这些范围内的值构成的平滑曲线，开平方计算并保存到内部查找表格中。这些值在之后需要时都可以通过调用valueForPower:方法来获取。

下面回到THRecorderController类，对其进行一些修改。添加一个新的属性来保存计量表格，在init方法中创建它的一个新实例(如代码清单2-26所示)。

代码清单 2-26　THRecorderController 计量表格设置

```
@interface THRecorderController () <AVAudioRecorderDelegate>

@property (strong, nonatomic) AVAudioPlayer *player;
@property (strong, nonatomic) AVAudioRecorder *recorder;
@property (strong, nonatomic) THRecordingStopCompletionHandler completionHandler;
@property (strong, nonatomic) THMeterTable *meterTable;

@end
```

```
@implementation THRecorderController
- (instancetype)init {
    self = [super init];
    if (self) {
        NSString *tmpDir = NSTemporaryDirectory();
        NSString *filePath = [tmpDir stringByAppendingPathComponent:@"memo.caf"];
        NSURL *fileURL = [NSURL fileURLWithPath:filePath];

        NSDictionary *settings = @{
            AVFormatIDKey : @(kAudioFormatAppleIMA4),
            AVSampleRateKey : @44100.0f,
            AVNumberOfChannelsKey : @1,
            AVEncoderBitDepthHintKey : @16,
            AVEncoderAudioQualityKey : @(AVAudioQualityMedium)
        };
        NSError *error;
        self.recorder = [[AVAudioRecorder alloc] initWithURL:fileURL
                                                    settings:settings
                                                       error:&error];
        if (self.recorder) {
            self.recorder.delegate = self;
            self.recorder.meteringEnabled = YES;
            [self.recorder prepareToRecord];
        } else {
            NSLog(@"Error: %@", [error localizedDescription]);
        }

        _meterTable = [[THMeterTable alloc] init];
    }

    return self;
}
```

现在添加一个新方法用于返回级别大小。代码清单2-27给出了实现代码。记得在类的头部添加声明。

代码清单 2-27 THRecorderController levels 方法

```
- (THLevelPair *)levels {
    [self.recorder updateMeters];
    float avgPower = [self.recorder averagePowerForChannel:0];
    float peakPower = [self.recorder peakPowerForChannel:0];
    float linearLevel = [self.meterTable valueForPower:avgPower];
    float linearPeak = [self.meterTable valueForPower:peakPower];
    return [THLevelPair levelsWithLevel:linearLevel peakLevel:linearPeak];
}
```

该方法首先调用录音器的updateMeters方法。该方法一定要正好在读取当前等级值之前调用，以保证读取的级别是最新的。之后向通道0请求平均值和峰值数据。通道都是0索引的，由于我们使用单声道录制，只需要询问第一个声道即可。之后在计量表格中查询线性声音强度值并最终创建一个新的THLevelPair实例。这个类在示例项目中已经存在，它只是用来保存

返回的平均值和峰值数据。

读取音频强度值与请求当前时间类似，当需要最新值时都需要轮询录音器。与请求当前时间时一样，客户端代码可能使用NSTimer。不过由于你希望频繁更新用于展示的计量值以保持动画效果比较平滑，所以可能改用CADisplayLink作为解决方案。CADisplayLink与NSTimer类似，不过它可以与显示刷新率自动同步。如果打开MainViewController.m文件，可看到如代码清单2-28所示的方法。

代码清单 2-28　THMainViewController 计量方法

```
- (void)startMeterTimer {
    [self.levelTimer invalidate];
    self.levelTimer = [CADisplayLink displayLinkWithTarget:self
                                                   selector:@selector(updateMeter)];
    self.levelTimer.frameInterval = 5;
    [self.levelTimer addToRunLoop:[NSRunLoop currentRunLoop]
                          forMode:NSRunLoopCommonModes];
}

- (void)stopMeterTimer {
    [self.levelTimer invalidate];
    self.levelTimer = nil;
    [self.levelMeterView resetLevelMeter];
}

- (void)updateMeter {
    THLevelPair *levels = [self.controller levels];
    self.levelMeterView.level = levels.level;
    self.levelMeterView.peakLevel = levels.peakLevel;
    [self.levelMeterView setNeedsDisplay];
}
```

当调用startLevelTimer方法时，会创建一个新的CADisplayLink实例，定期调用updateLevelMeter方法。默认情况下，调用的时间间隔与刷新率是同步的，但在本例中，设置frameInterval属性的值为4，也就是时间间隔为刷新率的1/4，这足以满足我们的需要了。

updateLevelMeter方法通过查询THRecorderController得到强度信息，将这些信息传递到levelMeterView，然后调用setNeedsDisplay重绘视图。

再次编译并运行应用程序，并开始录制一段新的内容。音量强度值会根据说话者声音的变化而变化，并不断更新展示。应用程序为用户提供了可视化的反馈效果来让他们知道语音备忘录正在被其正录制。

关于声音计量展示需要注意的一点是，这么做会增加开销。启用计量功能会导致一些额外计算，会影响设备的耗电量。此外，示例中的应用程序使用的是基于Quartz的计量。Quartz是一个卓越的框架，但会占用CPU资源，所以不断地计算音量数据会产生一定的开销。由于该应用程序旨在记录短的语音备忘信息，通常内容不会比说一句“去超市买牛奶”或“下周五与Charlie会面”更长，所以占用CPU资源并不是很大的问题。不过如果要录制长时间的音频内容，可能需要考虑禁用音频计量功能，或者选择一种更高效的绘制方法，比如使用OpenGL ES。

2.11 小结

本章我们见识了AV Foundation的音频类所能提供的强大功能。AVAudioSession作为应用程序和更大的iOS音频环境的中间环节，可通过使用分类在语义上定义应用程序的行为，并且提供工具来观察中断和线路变化。AVAudioPlayer和AVAudioRecorder提供了一种简单但功能强大的接口，用于处理音频的播放和录制。这两个类都构建于Core Audio框架之上，但为在应用程序中实现音频录制和播放提供了一种更便捷的方法。

第3章

资源和元数据

AV Foundation最初的功能并不强大。通过上一章的学习，我们知道iOS 2.2和3.0两个版本给开发者带来了专用于处理音频的类，虽然这些类被广泛用于各种应用程序，不过它们还仅是AV Foundation大概念的开始。直到iOS 4.0版本，AV Foundation才以现在的模样出现在开发者眼前，该版本带给我们一个非常大的且可扩展的框架，包括对媒体资源的捕捉、组合、播放和处理等广泛功能。同时它还有别于传统面向文件的音频类，框架把所有的代码设计围绕“资源(assert)”进行。本章将开始进入AV Foundation面向资源开发的世界，其中最核心的类就是AVAsset。

3.1 理解资源的含义

AV Foundation最重要的类就是AVAsset，它是AV Foundation设计的核心，在几乎所有特性和功能的开发中扮演着至关重要的角色。AVAsset是一个抽象类和不可变类，定义了媒体资源混合呈现的方式，将媒体资源的静态属性模块化成一个整体，比如它们的标题、时长和元数据等。明确理解AVAsset的目标和功能对成功应用AV Foundation非常重要。

AVAsset不需要考虑媒体资源所具有的两个重要范畴。第一个是它提供了对基本媒体格式的层抽象，这意味着无论是处理QuickTime影片、MPEG-4视频还是MP3音频，对你和对框架其余部分而言，面对的只有资源这个概念。这就使得开发者在面对不同格式的内容时有一个统一的处理方法，不需要考虑多种编解码器和容器格式因为细节不同而带来的困扰。不过如果仍然希望获取这些细节信息，也是可以通过其他方法实现的。此外，AVAsset隐藏了资源的位置信息，当我们对一个现有媒体对象进行处理时，会通过URL进行初始化来创建一个资源。这个地址可能是指向应用程序bundle的本地URL，可能是文件系统的其他位置，也可能是从用户iPod库中得到的URL，甚至可能是远程服务器上的一个音频流或视频流的URL。

另外，AVAsset会把你从这些细节中解脱出来，让框架来处理那些繁重的工作，这样做可以确保在不考虑位置的前提下合理地获取及载入媒体。由于不考虑媒体格式及位置等复杂问题，AVAsset为开发者处理时基媒体(timed media)提供了一种简单、统一的方式。

AVAsset本身并不是媒体资源，但是它可以作为时基媒体的容器。它由一个或多个带有描

述自身元数据的媒体组成。我们使用AVAssetTrack类代表保存在资源中的统一类型媒体，并对每个资源建立相应的模型。AVAssetTrack最常见的形态就是音频和视频流，但是它还可以用于表示诸如文本、副标题或隐藏字幕等媒体类型，如图3-1所示。

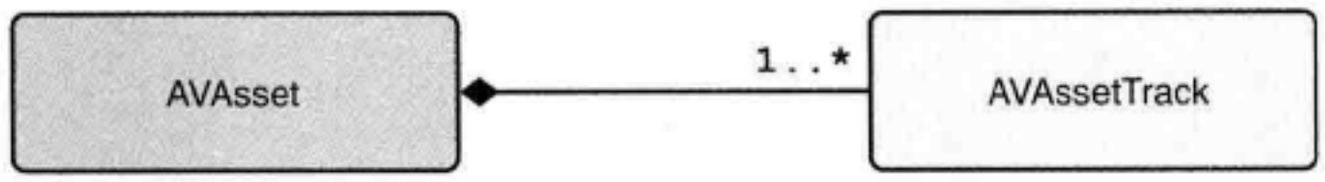

图 3-1 AVAsset 的组成

资源的曲目可通过其tracks属性进行访问。对该属性的请求会返回一个NSArray，该数组中的元素就是专辑包含的所有曲目。此外，AVAsset还可以通过标识符、媒体类型或媒体特征等信息找到相应的曲目。这使得在未来更高级的处理中我们可以很容易地获取一组需要的曲目。

3.2 创建资源

当你为一个现有媒体资源创建AVAsset对象时，可以通过URL对它进行初始化来实现。一般来说是一个本地文件URL，但也可能是远程资源的URL。

```
NSURL *assetURL = // url
AVAsset *asset = [AVAssetassetWithURL:assetURL];
```

AVAsset是一个抽象类，意味着它不能直接被实例化。当使用它的assetWithURL:方法创建实例时，实际上是创建了它子类的一个实例，子类名为AVURLAsset。有时我们会直接使用这个类，因为它允许通过传递选项字典来精细调整资源的创建方式。比如，如果创建一个用在音频或视频编辑场景中的资源，可能希望传递一个选项(option)来告诉程序提供更精确的时长和计时信息，如下例所示：

```
NSURL *assetURL = // url
NSDictionary *options = @{AVURLAssetPreferPreciseDurationAndTimingKey: @YES};
AVURLAsset *asset = [[AVURLAssetalloc] initWithURL:assetURLoptions:options];
```

传递该选项就暗示了开发者希望得到稍长一点的加载时间，以获取更准确的时长及时间信息。

有很多常见的位置是开发者希望创建资源对象的地方。在iOS设备中，开发者可能希望在用户的照片库中访问视频文件，或者在iPod库中访问歌曲。在Mac中，开发者可能希望从用户的iTunes库中找到媒体项。借助于iOS和OS X中一些可用的辅助框架，我们也可以使用这些媒体资源。让我们来看几个使用这些框架的例子。

3.2.1 iOS Assets 库

用户使用系统自带的Camera程序或第三方视频捕捉程序捕捉的视频，它们通常保存在用户的照片库中。iOS提供的Assets库框架可以实现从照片库中读写的功能。下例从用户资源库中的视频创建一个AVAsset：

```
ALAssetsLibrary *library = [[ALAssetsLibraryalloc] init];
```

```
[library enumerateGroupsWithTypes:ALAssetsGroupSavedPhotos
                        usingBlock:^(ALAssetsGroup *group, BOOL *stop) {

        // Filter down to only videos
        [groupsetAssetsFilter:[ALAssetsFilterallVideos]];

        // Grab the first video returned
        [groupenumerateAssetsAtIndexes:[NSIndexSet indexSetWithIndex:0]
                                 options:0
                              usingBlock:^(ALAsset *alAsset,
                                           NSUInteger index,
                                           BOOL *innerStop) {
            if (alAsset) {
                id representation = [alAssetdefaultRepresentation];
                NSURL *url = [representation url];
                AVAsset *asset = [AVAssetassetWithURL:url];
                // Asset created. Perform some AV Foundation magic.
            }
        }];
    } failureBlock:^(NSError *error) {
      NSLog(@"Error: %@", [error localizedDescription]);
}];
```

该例展示了如何获取保存在Saved Photos组中的视频资源，程序在这个组中筛选出视频内容，并获取结果集合中的第一个视频资源。库中的条目全部被建模为ALAsset对象。为默认呈现方式选用ALAsset类型会返回一个ALAssetRepresentation，它提供了一个适于创建AVAsset的URL。

3.2.2　iOS iPod 库

我们获取媒体的一个常见位置是用户的iPod库。MediaPlayer框架提供了API，实现在这个库中查询和获取条目。当需要的条目找到后，可以获取其URL并使用这个URL初始化一个资源，如下例所示：

```
MPMediaPropertyPredicate *artistPredicate =
    [MPMediaPropertyPredicatepredicateWithValue:@"Foo Fighters"
                                     forProperty:MPMediaItemPropertyArtist];

MPMediaPropertyPredicate *albumPredicate =
    [MPMediaPropertyPredicatepredicateWithValue:@"In Your Honor"
                                     forProperty:MPMediaItemPropertyAlbumTitle];

MPMediaPropertyPredicate *songPredicate =
[MPMediaPropertyPredicatepredicateWithValue:@"Best of You"
                                     forProperty:MPMediaItemPropertyTitle];

MPMediaQuery *query = [[MPMediaQueryalloc] init];

[queryaddFilterPredicate:artistPredicate];
[queryaddFilterPredicate:albumPredicate];
[queryaddFilterPredicate:songPredicate];
```

```
NSArray *results = [query items];
if (results.count> 0) {
    MPMediaItem *item = results[0];
    NSURL *assetURL = [item valueForProperty:MPMediaItemPropertyAssetURL];
    AVAsset *asset = [AVAssetassetWithURL:assetURL];
    // Asset created. Perform some AV Foundation magic.
}
```

MediaPlayer框架提供了一个名为MPMediaPropertyPredicate的类，它用于构建帮助用户在iPod库中查找具体内容所用的查询语句。上述示例代码在Foo Fighter的In Your Honor唱片中查找Best of You这首歌。在一条查询语句成功执行后，会捕捉到这个媒体条目的资源URL属性，并使用这个属性创建一个AVAsset。

3.2.3 Mac iTunes 库

在OS X上，iTunes是用户的中心媒体资源库。要识别这个库中的资源，开发者通常要对iTunes音乐目录中的iTunes Music Library.xml文件进行解析，从而得到相关数据。虽然这确实是一个可行方案，不过从Mac OS X 10.8和iTunes 11.0开始，我们有了更简单的方法，这要归功于iTunesLibrary框架。让我们看一下如何对库中资源进行查询。

```
ITLibrary *library = [ITLibrarylibraryWithAPIVersion:@"1.0" error:nil];

NSArray *items = self.library.allMediaItems;
NSString *query = @"artist.name == 'Robert Johnson' AND "
                  "album.title == 'King of the Delta Blues Singers' AND "
                  "title == 'Cross Road Blues'";
NSPredicate *predicate =
    [NSPredicatepredicateWithFormat:query];

NSArray *songs = [items filteredArrayUsingPredicate:predicate];
if (songs.count> 0) {
    ITLibMediaItem *item = songs[0];
    AVAsset *asset = [AVAssetassetWithURL:item.location];
    // Asset created. Perform some AV Foundation magic.
}
```

iTunesLibrary框架并没有像MediaPlayer框架那样给出具体的查询API。不过开发者可使用标准的Cocoa NSPredicate类来构建一个复杂查询，从而查找所需的条目。当筛选出所需的媒体条目集合后，可使用ITLibMediaItem获取location属性，这样就能得到一个适于创建AVAsset对象的URL。

3.3 异步载入

AVAsset具有多种有用的方法和属性，可以提供有关资源的信息，比如时长、创建日期和元数据等。AVAsset还包含一些用于获取和使用曲目集合的方法。不过有一点很重要，就是当创建时，资源就是对基础媒体文件的处理。AVAsset使用一种高效的设计方法，即延迟载入资源的属性，直到请求时才载入。这样就可以快速地创建资源，而不用考虑因为立即载入相关

媒体或元数据所带来的问题，不过有一点很重要，就是属性的访问总是同步发生的。如果正在请求的属性没有预先载入，程序就会阻塞，直到其可以做出适当的响应，显然这种情况一定会带来问题。比如，请求资源的duration可能是一个潜在的昂贵操作。如果开发者在使用MP3文件时没有在头文件中设置TLEN标签，这个标签用于定义duration值，则整个音频曲目都需要进行解析来准确确定它的duration值。假设这个请求发生在主线程，那么等待响应就会阻塞主线程，直到相关操作完成为止。在最好的情况下，可能会感觉应用程序变得迟钝，用户界面没有响应。在iOS上，最糟情况下，可能会出现卡顿，导致系统监视器介入，并终止应用程序的运行。如果出现这种情况，没理由不让用户给予程序“1星”评价。要解决这一问题，开发者应该使用异步的方式来查询资源的属性。

AVAsset和AVAssetTrack都采用了AVAsynchronousKeyValueLoading协议。该协议通过下面给出的这些方法实现了异步查询属性的功能：

```
- (AVKeyValueStatus)statusOfValueForKey:(NSString *)key
                                  error:(NSError **)outError

- (void)loadValuesAsynchronouslyForKeys:(NSArray *)keys
                       completionHandler:(void (^)(void))handler
```

可使用statusOfValueForKey:error:方法查询一个给定属性的状态，该方法会返回一个枚举类型的AVKeyValueStatus值，用于表示当前所请求的属性的状态。如果状态不是AVKeyValueStatusLoaded，意味着此时请求该属性可能导致程序卡顿。要异步载入一个给定的属性，可以调用loadValuesAsynchronouslyForKeys: completionHandler:方法，为其提供一个具有一个或多个key的数组(资源的属性名)和一个completionHandler块，当资源处于回应请求的状态时，就会调用这个completionHandler块，如下例所示：

```
// Create URL for 'sunset.mov' in the application bundle
NSURL *assetURL =
    [[NSBundle mainBundle] URLForResource:@"sunset" withExtension:@"mov"];

AVAsset *asset = [AVAsset assetWithURL:assetURL];

// Asynchronously load the assets 'tracks' property
NSArray *keys = @[@"tracks"];
[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{

   // Capture the status of the 'tracks' property
   NSError *error = nil;
   AVKeyValueStatus status =
       [asset statusOfValueForKey:@"tracks" error:&error];

   // Switch over the status to determine its state
   switch (status) {
      case AVKeyValueStatusLoaded:
           // Continue Processing
           break;
      case AVKeyValueStatusFailed:
           // Handle failure with error
           break;
```

```
            case AVKeyValueStatusCancelled:
                // Handle explicit cancellation
                break;
            default:
                // Handle all other cases
        }
    }];
```

该例为存储在应用程序bundle中的QuickTime电影创建了一个AVAsset，并异步载入了该对象的tracks属性。在completionHandler块中，我们希望通过调用资源的statusOfValueForKey:error:方法来确定请求属性的状态。将NSError指针传递给这个方法很重要，因为如果返回的状态为AVKeyValueStatusFailed，则说明资源包含错误信息。通常会切换状态值，并根据返回的状态采取适当操作。要注意的是这个completionHandler块可能会在任意一个队列中被调用。在对用户界面做出相应更新之前，必须首先回到主队列中。

> **注意：**
>
> 该例载入一个 tracks 属性，其实可以在一个调用中请求多个属性。当请求多个属性时，需要注意以下两点：
>
> (1) 每次调用 loadValuesAsynchronouslyForKeys:completionHandler:方法时只会调用一次 completionHandler 块。调用该方法的次数并不是根据传递给这个方法的键的个数而定的。
>
> (2) 需要为每个请求的属性调用 statusOfValueForKey:error:方法，不能假设所有的属性都返回相同的状态值。

3.4 媒体元数据

当创建一个媒体应用程序时，了解该媒体的组织格式非常重要。简单地向用户呈现一串文件名也许在文件不多的时候还能接受，但是这个方法显然无法满足大规模文件的呈现。所以我们真正需要的是，找到一种方法对媒体进行描述，让用户可以简单地找到、识别和组织这些媒体。幸运的是，我们所使用的AV Foundation中的主要媒体格式都可以嵌入描述其内容的元数据。对元数据的使用具有一定的挑战性，因为每个媒体类型都具有唯一的格式，并且通常要求开发者对相应格式读写操作的底层技术有所了解。不过AV Foundation让这一切变得简单，因为它使开发者不需要考虑大多数特定格式的细节；在处理媒体元数据方面，AV Foundation提供了一套统一的方法。本章剩余的内容将深入研究AV Foundation元数据的相关细节问题，学习如何在应用程序中实现这些功能。在我们开始详解前，先来看一下元数据是如何以多种不同格式保存的。

元数据格式

虽然存在多种格式的媒体资源，但是我们在Apple环境下遇到的媒体类型主要有4种，分别是：QuickTime(mov)、MPEG-4 video(mp4和m4v)、MPEG-4 audio(m4a)和MPEG-Layer III audio(mp3)。虽然AV Foundation在处理这些文件中嵌入的元数据时都使用一个接口，但是理

解这些不同类型资源的元数据如何存储及存储位置仍然很有价值。这里只做概述，但它是未来深入研究的基础。

1. QuickTime

QuickTime是由苹果公司开发的一种功能强大、跨平台的媒体架构。该架构的一部分是QuickTime File Format规范，定义了.mov文件的内部结构。QuickTime文件由一种称为atoms的数据结构组成。一般规则是这样的：一个atom包含了描述媒体资源某一方面的数据，或者它可以包含其他atom，但不可以两者都包含。有些情况下，苹果公司自己的方法实现可能会违背这一规则。atom以一种复杂的树状结构组合在一起，详细地对布局、音频样本格式、视频帧信息乃至需要呈现的元数据信息(如作者信息和版权信息)做了描述。

了解QuickTime格式的一个好办法就是在十六进制编辑器中打开一个.mov格式的文件，常见的十六进制编辑器有Hex Fiend或Synalyze It! Pro。典型的十六进制工具会将一个真实QuickTime文件的数据呈现到开发者眼前，不过想象其中的结构和atom间的关系也不是那么容易。更好的方法是借助于Apple Developer Center中可以找到的Atom Inspector工具。这个工具将atom结构以NSOutlineView方式呈现，所以就可以对atom之间的继承关系等信息有比较清晰的了解，这个工具还提供了一个小型的十六进制查看器，可以从中查看到实际字节布局。

《超人总动员》是作者非常喜欢的一部由Pixar公司出品的电影，图3-2给出了这部作品QuickTime版本结构的简化示意图。该QuickTime文件最小限度地包含了三个高级atom，分别是用于描述文件类型和兼容类型的ftyp，包含实际音频和视频媒体的mdat以及非常重要的moov atom(读作moo-vee)，它对媒体资源的所有相关细节做了完整描述，包括可呈现的元数据。

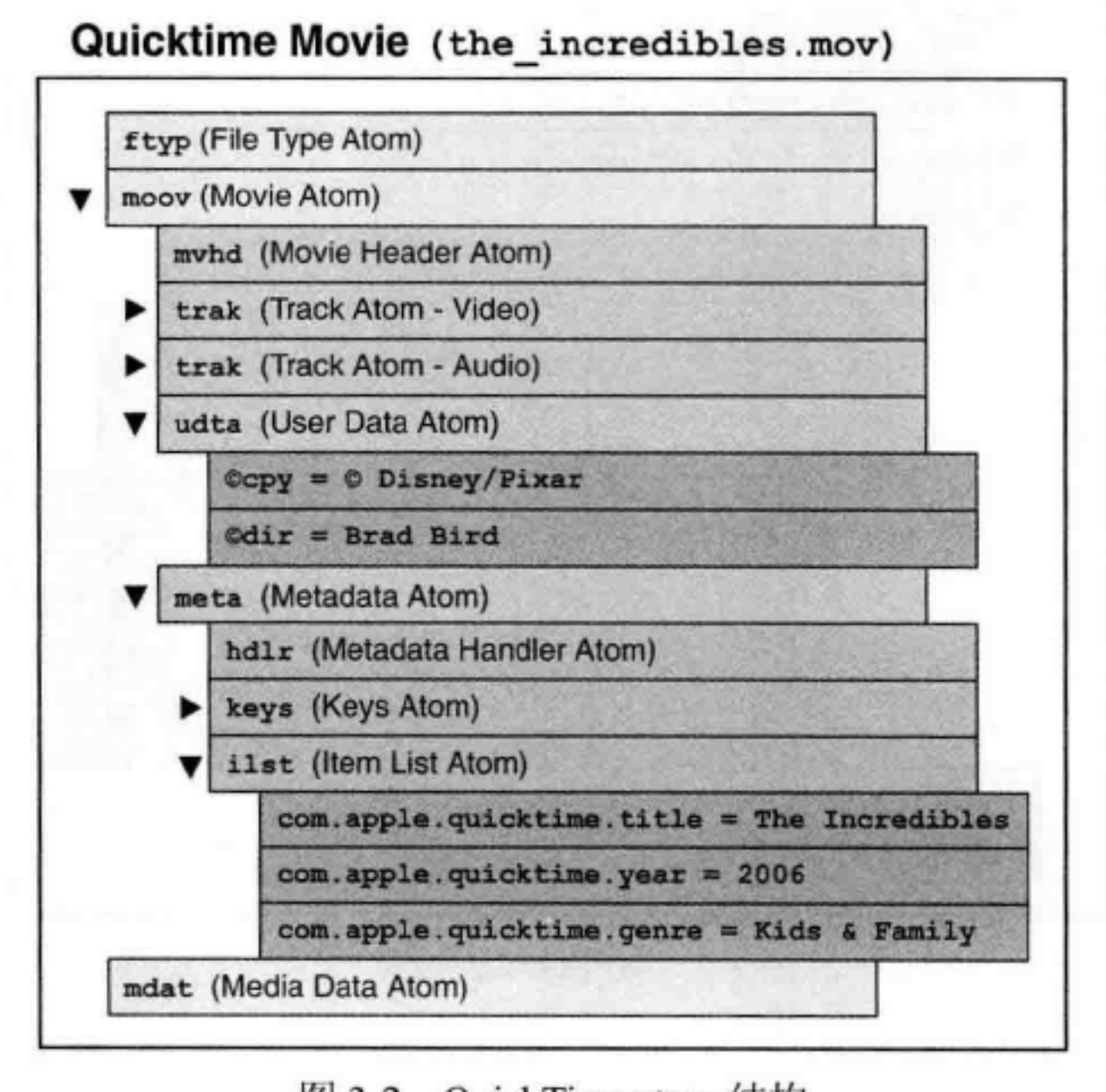

图 3-2　QuickTime atom 结构

当处理QuickTime电影时会遇到两种类型的元数据。标准QuickTime元数据由诸如Final Cut Pro X这样的工具编写，位于/moov/meta/ilst/中，它的键几乎都具有com.apple.quicktime前缀。其他类型的数据被认为是QuickTime用户数据，保存在/moov/udta/中。QuickTime用户数据可以包括播放器需要查找的标准数据，比如歌曲演唱者或版权信息，不过同时还可以包含任何对应用程序有帮助的信息。上述两种元数据类型在AV Foundation中都是可以读写的。

对不同atom间的关系和相关细节的理解不在我们的讨论范围内。实际上，苹果公司出具了一份400多页的名为QuickTime File Format Specification[1]的文档，该文档全面介绍了QuickTime文件格式的所有相关内容。虽然成为QuickTime领域的专家并不重要，但是我们还是建议大家对核心moov atom有所了解，因为这有助于你更好地掌握AV Foundation是如何使用这些数据的。

2. MPEG-4 音频和视频

MPEG-4 Part 14是定义MP4文件格式的规范。MP4直接派生于QuickTime文件格式，这就意味着它与QuickTime文件的结构是类似的。实际上，我们经常会发现，能够解析一种文件类型的工具也可以处理其他文件类型(有着不同程度的成功率)。就像QuickTime文件一样，MP4文件也由称为atom的数据结构组成。技术上讲，MPEG-4规范将这些称为boxes，不过考虑到该格式的QuickTime血统，大部分开发者还是把其称为atom。图3-3再次给出了《超人总动员》的例子，不过这次是MP4格式。

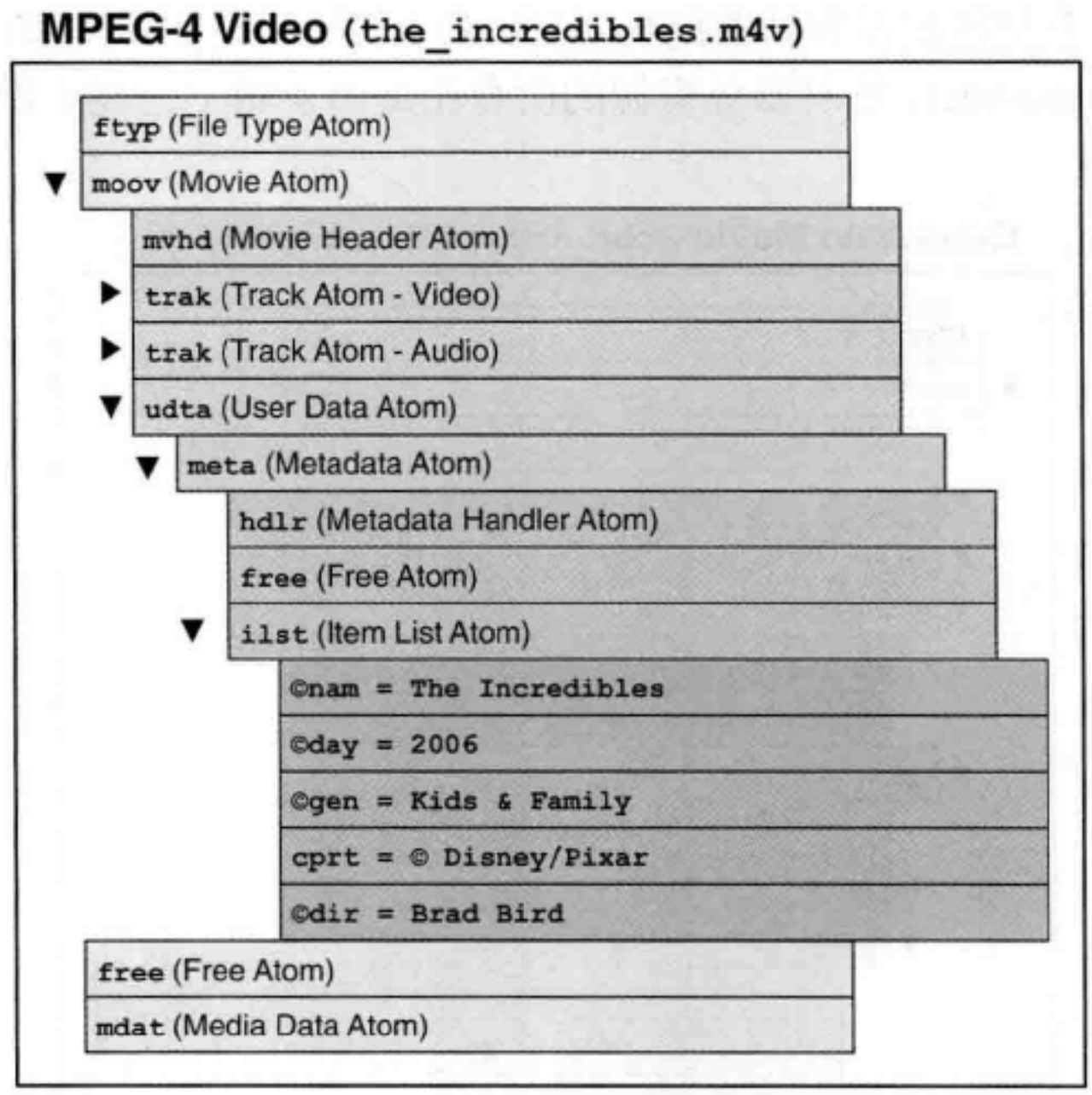

图 3-3 MPEG-4 atom 结构

1. QuickTime File Format Specification—https://developer.apple.com/library/mac/documentation/quicktime/QTFF/QTFFPreface/qtffPreface.html。

MPEG-4文件的元数据保存在/moov/udta/meta/ilst/中。虽然对于在atom中使用的键没有标准，但大部分工具都遵循苹果公司尚未发布的iTunes元数据规范中对键的定义。虽然没有正式发布，但是iTunes元数据格式的相关文档在互联网上已经广为人知了。我们可以在开源的mp4v2库[2]中找到一个对这种格式进行讲解的优秀文档。

在实践中，大家对于该文件的扩展名还存在一些困惑。.mp4是对MPEG-4媒体的标准扩展，但存在一些变化，如.m4v、.m4a、.m4p和.m4b。这些变体都使用MPEG-4容器格式，但有些包含了附加的扩展功能。M4V文件是带有苹果公司针对FairPlay加密及AC3-audio扩展的MPEG-4视频格式。如果不涉及FairPlay加密和AC3-audio扩展，则MP4和M4V就仅仅是扩展名不同而已。M4A专门针对音频，使用变化的扩展名的目的是让使用者知道该文件只带有音频资源。M4P是苹果较旧的iTunes音频格式，使用其FairPlay扩展。M4B用于有声读物，并通常包含章节标签以及提供书签功能，让读者可以返回到指定位置开始阅读。

3. MP3

MP3文件与上面介绍的两种格式有显著区别。MP3文件不使用容器格式，而使用编码音频数据，包含的可选元数据的结构块通常位于文件开头。MP3文件使用一种称为ID3v2的格式来保存关于音频内容的描述信息，包含的数据有歌曲演唱者、所属唱片和音乐风格等。

使用十六进制工具研究MP3格式的架构是一个好办法。不要担心，与上面小节中介绍的atom相比，ID3数据很容易理解。MP3文件的前10个字节带有嵌入的元数据，这10个字节定义了ID3块的头部。该头部的前三个字节始终为'49 44 33' (ID3)，用于表示这是一个ID3v2标签，后面两个字节用于定义主版本信息，即2、3、4和版本号。剩余字节用于定义标志集合及ID3块的大小，如图3-4所示。

ID3 Header				
ID3	Version	Revision	Flags	Size

图 3-4　ID3 头部的布局

ID3块中剩下的数据都是用于描述不同元数据键值的帧。每一帧都有一个带有实际标签名称的10字节的头，之后的4字节表示尺寸，再之后的两个字节用来定义选项标志。帧剩下的字节包含了实际的元数据值。如果值是文本类型，这是最常见的情况，tag中的第一个字节用来定义类型编码。类型编码通常被设置为0x00，代表ISO-8859-1，不过这里也支持其他类型的编码。图3-5给出了John Coltrane经典曲目Giant Steps的MP3版本中的ID3结构示意图。

AV Foundation支持读取ID3v2标签的所有版本，但不支持写入。MP3格式受到专利限制，所以AV Foundation无法支持对MP3或ID3数据进行编码。

---

2. mp4v2库— https://code.google.com/p/mp4v2/wiki/iTunesMetadata。

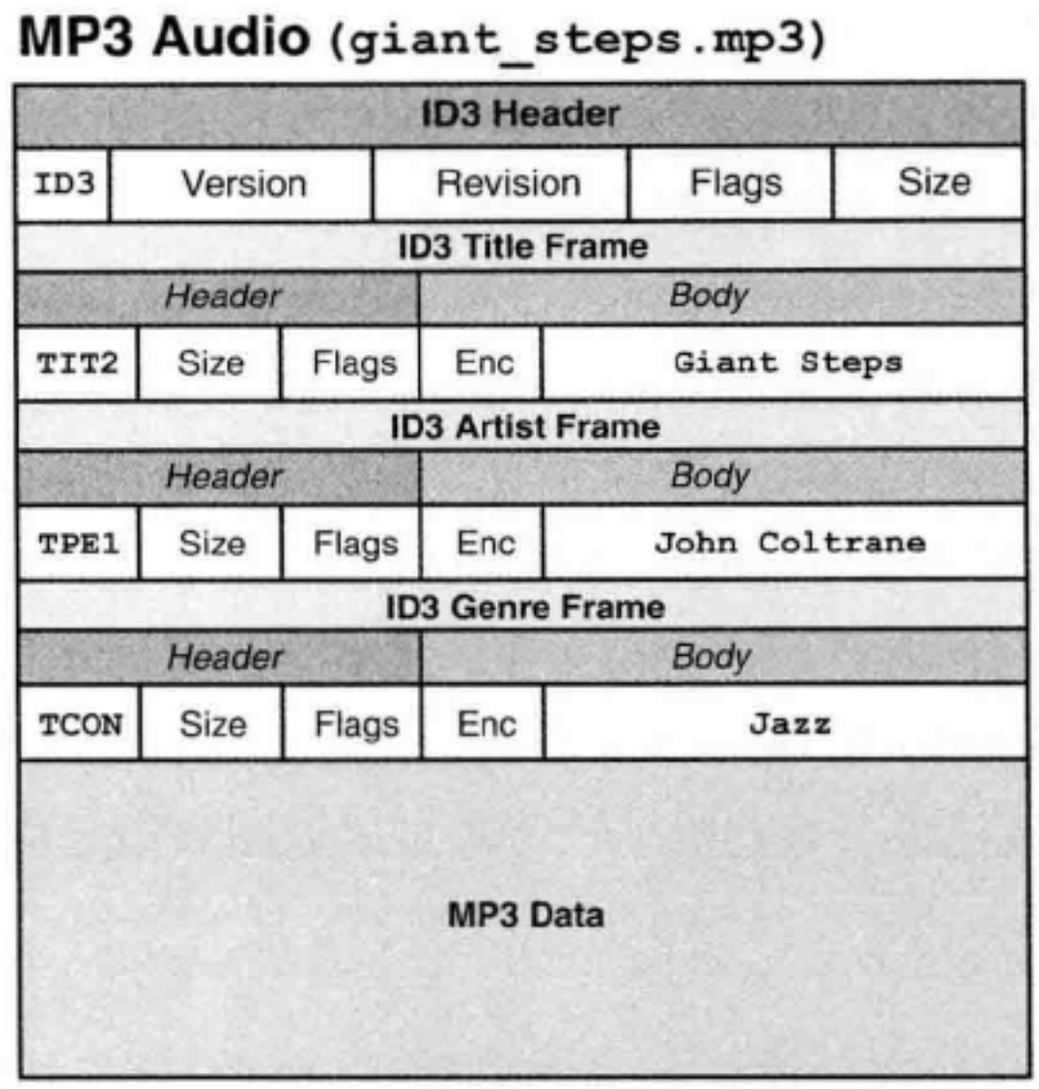

图 3-5 ID3 结构

> **注意：**
>
> AV Foundation 支持所有 ID3v2 标签格式的读取操作，但是 ID3v2 是要加星号的。ID3v2.2 的布局和 ID3v2.3 及之后版本的布局不同。值得注意的是，有些标签是由 3 个字符组成而非 4 个，比如一首歌曲的标注信息，当标签为 ID3v2.2 时，是被保存在 COM 帧中，但当同一首歌使用 ID3v2.3 标签或更新版本的标签时，歌曲的标注信息会被保存在 COMM 帧中。框架定义的字符常量只适用于 ID3v2.3 及之后版本。不过我们会在之后的示例程序中告诉你如何仍使用 ID3v2.2 数据。

3.5 使用元数据

AVAsset和AVAssetTrack都可以实现查询相关元数据的功能。大部分情况下我们会使用AVAsset提供的元数据，不过当涉及获取曲目一级元数据等情况时也会使用AVAssetTrack。读取具体资源元数据的接口由名为AVMetadataItem的类提供。这个类提供了一个面向对象的接口，让开发者可以对存储于QuickTime、MPEG-4 atom和ID3帧中的元数据进行访问。

AVAsset和AVAssetTrack提供了两种方法可以获取相关的元数据。要了解这两个不同的方法的适用范围，首先要知道键空间(key spaces)的含义。AV Foundation使用键空间作为将相关键组合在一起的方法，可以实现对AVMetadataItem实例集合的筛选。每个资源至少包含两个键空间，供从中获取元数据，如图3-6所示。

Common		
iTunes	Quicktime Metadata Quicktime User Data	ID3
MPEG 4	MOV	MP3
iTunes Audio/Video	Quicktime Movie	MPEG Layer III
.mp4, .m4v, .m4a	.mov	.mp3

图 3-6　元数据键空间

Common键空间用来定义所有支持的媒体类型的键，包括诸如曲名、歌手和插图信息等常见元素。这提供了一种对所有支持的媒体格式进行一定级别的元数据标准化的过程。开发者可以通过查询资源或曲目的commonMetadata属性从Common键空间获取元数据，这个属性会返回一个包含所有可用元数据的数组。

访问指定格式的元数据需要在资源或曲目上调用metadataForFormat:方法。这个方法包含一个用于定义元数据格式的NSString对象并返回一个包含所有相关元数据信息的NSArray。AVMetadataFormat.h为不同的元数据格式提供对应的字符串常量。与硬编码某个具体的元数据格式字符串不同，也可以查询资源或曲目的availableMetadataFormats，其会返回一个字符串数组，其中定义了资源中包含的所有元数据格式。开发者可以利用这个结果数组循环访问所有的格式，并为每种格式调用metadataForFormat:方法。让我们看一个例子：

```
NSURL *url = // url
AVAsset *asset = [AVAssetassetWithURL:url];

NSArray *keys = @[@"availableMetadataFormats"];
[assetloadValuesAsynchronouslyForKeys:keyscompletionHandler:^{

   NSMutableArray *metadata = [NSMutableArray array];
   // Collect all metadata for the available formats
   for (NSString *format in asset.availableMetadataFormats) {
       [metadataaddObjectsFromArray:[asset metadataForFormat:format]];
   }
   // Process AVMetadataItems
}];
```

3.5.1　查找元数据

当我们得到一个包含元数据项的数组时，通常希望找到所需的具体元数据值。一个特别有效的做法是使用AVMetadataItem提供的便利方法，来获取结果集合并对其进行筛选。例如，如果开发者希望得到一个M4A音频文件的演奏者和唱片的元数据，需要按如下方法获

取这些数据：

```
NSArray *metadata = // Collection of AVMetadataItems

NSString *keySpace = AVMetadataKeySpaceiTunes;
NSString *artistKey = AVMetadataiTunesMetadataKeyArtist;
NSString *albumKey = AVMetadataiTunesMetadataKeyAlbum;

NSArray *artistMetadata = [AVMetadataItemmetadataItemsFromArray:metadata
                                                         withKey:artistKey
                                                       keySpace:keySpace];

NSArray *albumMetadata = [AVMetadataItemmetadataItemsFromArray:metadata
                                                         withKey:albumKey
                                                       keySpace:keySpace];

AVMetadataItem *artistItem, *albumItem;
if (artistMetadata.count> 0) {
    artistItem = artistMetadata[0];
}

if (albumMetadata.count> 0) {
    albumItem = albumMetadata[0];
}
```

例子中使用了AVMetadataItem的metadataItemsFromArray:withKey:keySpace:方法来对集合进行筛选，得出那些匹配键和键空间标准的对象。这个方法的返回值是一个NSArray，但通常只包含单个AVMetadataItem实例。

3.5.2 使用 AVMetadataItem

AVMetadataItem最基本的形式其实是一个封装键值对的封装器。可通过它查询key或commonKey，查询其是否存在于Common键空间中，最重要的是它对应的value。value属性被定义成id<NSObject, NSCopying>形式，不过它可能是NSString、NSNumber、NSData或其他一些情况，比如NSDictionary。如果开发者已经提前知道value的类型，AVMetadataItem还会提供三个类型强制属性——stringValue、numberValue和dataValue——这简化了输入相应返回值的过程。

大部分开发者在第一次接触AVMetadataItem时都会碰到的一个常见问题是，如何理解该对象的key属性。commonKey被定义为一个字符串，并且很容易估算它在AVMetadataFormat.h中所对应的key，不过key属性是以id<NSObject, NSCopying>值的形式定义的。虽然这个类型可以保存NSString，不过很少这么使用。比如，如果通过下面的代码循环访问嵌入到库中的M4A文件内的元数据，就会得到错误的键值：

```
NSURL *url = // Song URL
AVAsset *asset = // Asset that has its properties loaded

NSArray *metadata =
    [assetmetadataForFormat:AVMetadataFormatiTunesMetadata];
```

```
for (AVMetadataItem *item in metadata) {
    NSLog(@"%@: %@", item.key, item.value);
}
```

执行这段代码将生成如下清单：

```
-1452383891: Have A Drink On Me
-1455336876: AC/DC
-1451789708: A. Young - M. Young - B. Johnson
-1453233054: Back In Black
-1453039239: 1980
```

虽然我们通过具体的结果可以看出元数据信息所描述的歌曲是Back in Black唱片中的AC/DC，不过对于key属性返回的整数形式无法知道它的含义。我们看到的是不同的key字符串所对应的整数值。在对这些值进行解释前，需要将它们转换为相应的字符串。由于这是一类常见问题，因此可在AVMetadataItem上添加一个名为keyString的分类方法，这样就可以很容易地获取NSString所对应的内容了。这个分类方法的实现如代码清单3-1所示。

代码清单 3-1　AVMetadataItem keyString 分类方法

```
#import "AVMetadataItem+THAdditions.h"

@implementation AVMetadataItem (THAdditions)
- (NSString *)keyString {
    if ([self.key isKindOfClass:[NSString class]]) {                    // 1
        return (NSString *)self.key;
    }
    else if ([self.key isKindOfClass:[NSNumber class]]) {

        UInt32 keyValue = [(NSNumber *) self.key unsignedIntValue];     // 2

        // Most, but not all, keys are 4 characters. ID3v2.2 keys are
        // only be 3 characters long. Adjust the length if necessary.

        size_t length = sizeof(UInt32);                                 // 3
        if ((keyValue >> 24) == 0) --length;
        if ((keyValue >> 16) == 0) --length;
        if ((keyValue >> 8) == 0) --length;
        if ((keyValue >> 0) == 0) --length;

        long address = (unsigned long)&keyValue;
        address += (sizeof(UInt32) - length);

        // keys are stored in big-endian format, swap
        keyValue = CFSwapInt32BigToHost(keyValue);                      // 4

        char cstring[length];                                           // 5
        strncpy(cstring, (char *) address, length);
        cstring[length] = '\0';

        // Replace '©' with '@' to match constants in AVMetadataFormat.h
        if (cstring[0] == '\xA9') {                                     // 6
            cstring[0] = '@';
```

```
        }

        return [NSString stringWithCString:(char *) cstring                    // 7
        encoding:NSUTF8StringEncoding];
    }
    else {
         return @"<<unknown>>";
    }
}

@end
```

(1) 如果key属性已经是一个字符串，则原样返回，但这并不常见。

(2) 请求key的无符号整型值。返回值是一个稍后将提取出的表示4字符代码的32位big endian数字。

(3) 大部分情况下，value值都是一个4字符代码，比如©gen或TRAK，不过对于使用ID3v2.2标签的MP3文件，键值只有3个字符的长度。代码会移动每个字节来确定length值是否应该截短。

(4) 由于数字是big endian格式，因此使用CFSwapInt32BigToHost()函数将其转换为符合主CPU的顺序的little endian格式，Intel和ARM处理器都是这样要求的。

(5) 创建一个字符数组，并使用strncpy函数将字符字节填充到该数组中。

(6) 大量QuickTime用户数据和iTunes key的前缀都带有一个©符号。不过AVMetadataFormat.h中定义key所使用的前缀符号为@。所以为了进行key常量字符串比较，需要先将©替换为@。

(7) 最后，使用tringWithCString:encoding初始化器将字符数组转换为一个NSString。

如果我们导入这个分类并将前一示例修改为使用这个新方法，就会得到下面的结果，结果已不再那么难懂。

```
@nam: Have A Drink On Me
@ART: AC/DC
@wrt: A. Young - M. Young - B. Johnson
@alb: Back In Black
@day: 1980
```

注意：

除了通过键和键空间获取资源的元数据之外，Mac OS X 10.10 和 iOS 8 还引进了使用标识符(identifier)获取元数据的方法。标识符将键和键空间统一成单独的字符串，以一个更简单的模型来获取资源的元数据。本章使用键和键空间的方法，是因为这种方法可以兼容大部分 OS 版本，不过如果只针对 Mac OS X 10.10 或 iOS 8 进行开发，完全可以考虑使用标识符。

现在我们基本理解AVMetadataItem，下面让我们利用学到的知识编写一个名为MetaManager的Mac元数据编辑器应用程序。

3.6　创建 MetaManager 应用程序

MetaManager应用程序提供了一个简单界面，让用户可以查看并编辑元数据(如图3-7所示)。该程序可以让用户查看AV Foundation支持的所有媒体类型的元数据信息，还可以对除了MP3之外的所有资源写入元数据信息。该应用程序很好地展示了如何使用这些元数据类，同时也对刚刚接触该框架的开发者经常遇到的一些问题给出了很好的解决方案。用户界面创建完毕后，你只需把注意力放在如何构建基础AV Foundation框架上。可在Chapter 3目录中找到名为MetaManager_starter的项目代码。

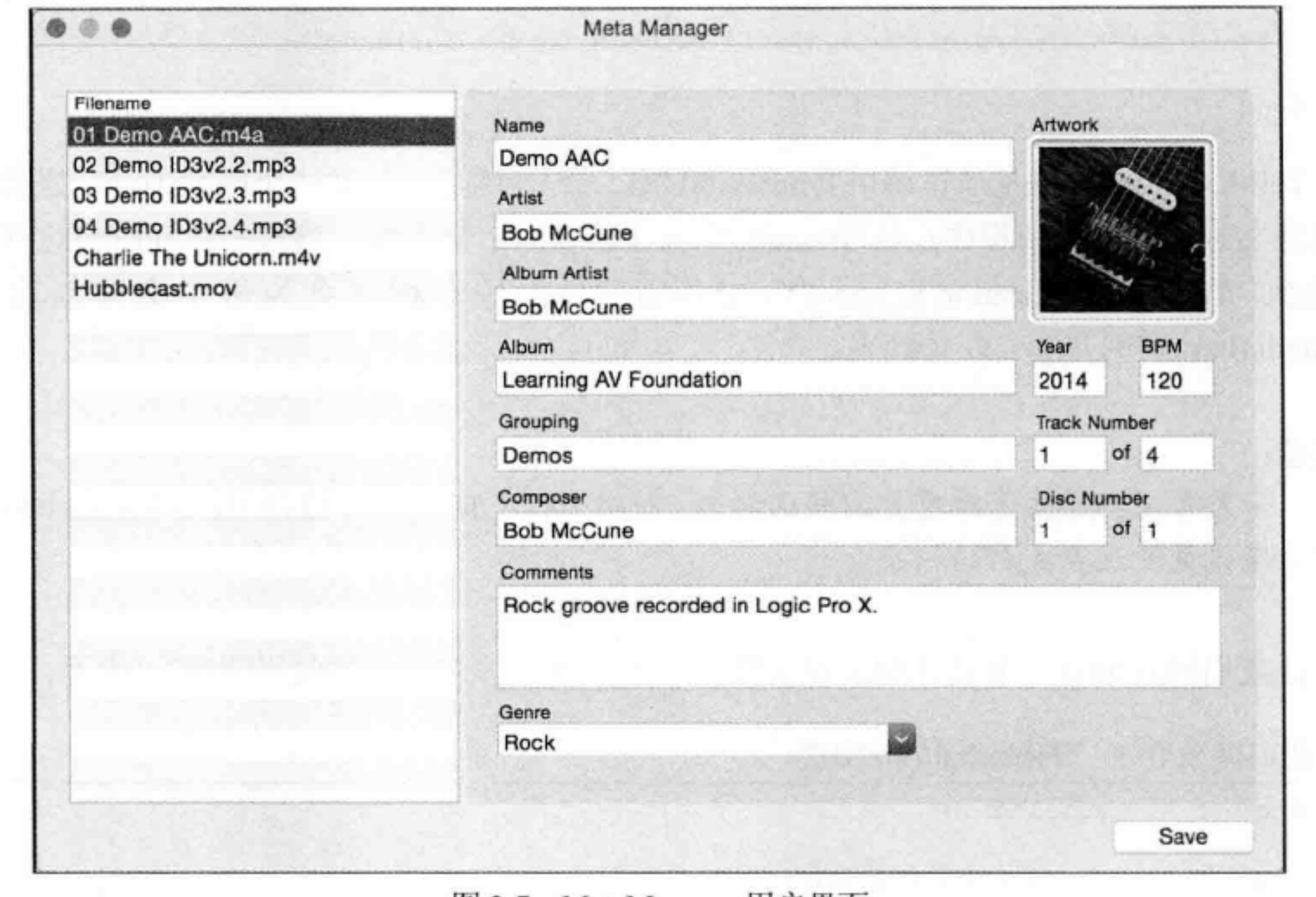

图 3-7　MetaManager 用户界面

虽然AV Foundation的元数据类可在一定程度上抽象因基础元数据格式不同而带来的区别，不过要对多种媒体格式的元数据进行统一管理还是很有挑战的。要对基础元数据提供一个统一界面，应该采用的策略是将特定格式的元数据映射到一个标准化的键值集合中。这就为基础元数据提供了一个简单、一致的用户界面，并使得管理这些元数据的逻辑方法可以集中在单个类中。让我们首先实现第一个类THMediaItem。

3.6.1　THMediaItem

THMediaItem定义了应用程序管理媒体最主要的接口。它封装了基础AVAsset实例并负责管理相关元数据的载入和解析。首先看一下这个类的接口，如代码清单3-2所示。

代码清单 3-2　THMediaItem 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THMetadata.h"
```

```
typedef void(^THCompletionHandler)(BOOL complete);

@interface THMediaItem : NSObject

@property (copy, readonly) NSString *filename;
@property (copy, readonly) NSString *filetype;
@property (strong, readonly) THMetadata *metadata;
@property (assign, readonly, getter = isEditable) BOOL editable;

- (id)initWithURL:(NSURL *)url;

- (void)prepareWithCompletionHandler:(THCompletionHandler)handler;
- (void)saveWithCompletionHandler:(THCompletionHandler)handler;

@end
```

THMediaItem实例是通过调用其initWithURL:方法创建的，传递一个磁盘中给定媒体文件的本地文件URL。应用程序在程序bundle中包含需要的示例文件，当程序启动时将这些文件复制到~/Library/Application Support/MetaManager目录中。程序会为每个文件创建一个THMediaItem实例并填充媒体列表。

> **注意：**
>
> 如果希望将媒体资源重置回默认状态，可以选择 Edit 菜单并选择 Reset Media Item，即可将媒体重新设置成初始状态。

让我们转过来看一下具体的实现文件并开始实现这些方法，如代码清单3-3所示。

代码清单 3-3 THMediaItem 实现

```
#import "THMediaItem.h"

#import "AVMetadataItem+THAdditions.h"
#import "NSFileManager+DirectoryLocations.h"

#define COMMON_META_KEY     @"commonMetadata"
#define AVAILABLE_META_KEY          @"availableMetadataFormats"

@interface THMediaItem ()
@property (strong) NSURL *url;
@property (strong) AVAsset *asset;
@property (strong) THMetadata *metadata;
@property (strong) NSArray *acceptedFormats;
@property BOOL prepared;
@end

@implementation THMediaItem

- (id)initWithURL:(NSURL *)url {
    self = [super init];
    if (self) {
        _url = url;                                                    // 1
        _asset = [AVAsset assetWithURL:url];
```

```
    _filename = [url lastPathComponent];
    _filetype = [self fileTypeForURL:url];                          // 2
    _editable = ![_filetype isEqualToString:AVFileTypeMPEGLayer3];  // 3
    _acceptedFormats = @[                                           // 4
        AVMetadataFormatQuickTimeMetadata,
        AVMetadataFormatiTunesMetadata,
        AVMetadataFormatID3Metadata
    ];
}
return self;
}
- (NSString *)fileTypeForURL:(NSURL *)url {
    NSString *ext = [[self.url lastPathComponent] pathExtension];
    NSString *type = nil;
    if ([ext isEqualToString:@"m4a"]) {
        type = AVFileTypeAppleM4A;
    } else if ([ext isEqualToString:@"m4v"]) {
        type = AVFileTypeAppleM4V;
    } else if ([ext isEqualToString:@"mov"]) {
        type = AVFileTypeQuickTimeMovie;
    } else if ([ext isEqualToString:@"mp4"]) {
        type = AVFileTypeMPEG4;
    } else {
        type = AVFileTypeMPEGLayer3;
    }
    return type;
}
```

(1) 在initWithURL:方法中设置类的内部状态，并基于传入的URL创建一个AVAsset实例。应用程序的表视图中显示的每个条目都是该类的一个实例。

(2) 为基础资源确定文件类型。虽然AV Foundation提供了一些高级方法用于确定文件类型，但基于文件扩展名来确定类型的方法已经可以满足需求了。当实现程序的保存功能时，我们会用到filetype属性。

(3) 程序会基于媒体文件的类型来设置editable标志。虽然AV Foundation允许我们读取ID3元数据，但我们无法向其中写入数据，所以需要设置editable标志，使得程序可以正确设置Save按钮的enabled状态。

(4) 还需要设置一个包含支持的所有元数据格式的数组。这么做是为了排除那些可能出现在AVMetadataKeySpaceQuickTimeUserData和AVMetadataKeySpaceISOUserData键空间中的外部重复值。

让我们继续研究prepareWithCompletionHandler:方法的实现，如代码清单3-4所示。

代码清单 3-4　prepareWithCompletionHandler:方法的实现

```
- (void)prepareWithCompletionHandler:(THCompletionHandler)completionHandler {
    if (self.prepared) {                                            // 1
        completionHandler(self.prepared);
        return;
    }
```

```
    self.metadata = [[THMetadata alloc] init];                          // 2

    NSArray *keys = @[COMMON_META_KEY, AVAILABLE_META_KEY];

    [self.asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{

        AVKeyValueStatus commonStatus =
            [self.asset statusOfValueForKey:COMMON_META_KEY error:nil];
        AVKeyValueStatus formatsStatus =
            [self.asset statusOfValueForKey:AVAILABLE_META_KEY error:nil];
        self.prepared = (commonStatus == AVKeyValueStatusLoaded) &&     // 3
                        (formatsStatus == AVKeyValueStatusLoaded);

        if (self.prepared) {
            for (AVMetadataItem *item in self.asset.commonMetadata) {   // 4
            //NSLog(@"%@: %@", item.keyString, item.value);
            [self.metadata addMetadataItem:item withKey:item.commonKey];
        }

        for (id format in self.asset.availableMetadataFormats) {        // 5
            if ([self.acceptedFormats containsObject:format]) {
                NSArray *items = [self.asset metadataForFormat:format];
                for (AVMetadataItem *item in items) {
                    //NSLog(@"%@: %@", item.keyString, item.value);
                    [self.metadata addMetadataItem:item
                                           withKey:item.keyString];
                }
            }
        }

        completionHandler(self.prepared);                               // 6
    }];
}
```

(1) 用户每次在应用的表视图中选择一个媒体项时，都会调用prepareWithCompletionHandler:方法。只需将这个项准备一次，所以如果已经准备了这个项，则直接调用completion块并跳出。

(2) 创建一个新的THMetadata实例。这个我们即将实现的类会对保存在单独AVMetadataItem实例中的值进行管理。

(3) 通过查看commonMetadata和availableMetadataFormats属性的状态来确定prepared状态。这两个属性一定要成功载入用于后续的进程。

(4) 循环访问所有由通用键空间返回的AVMetadataItem实例，并将每一个添加到THMetadata实例。

(5) 请求资源的availableMetadataFormats会返回一个数组，其中包含用于确定媒体元素中可用格式的字符串数组。如果supportedFormats集合中包含了特定格式，就需要获取相关的AVMetadataItem实例并将其添加到内部元数据存储中。

(6) 最后调用completion handler块使用户界面可以将获取的元数据呈现在屏幕上。

继续运行该应用程序。仅仅展示元数据的状态还不够，当选择单独的媒体项时，可以看

到控制台将键值信息输出了。当验证完这一行为后，就可以将注释代码清除了，或者将NSLog语句删除掉。

下面将注意力转到THMetadata类，重点研究单个AVMetadataItem实例中返回值相关的细节问题。

3.6.2　THMetadata 的实现

THMetadata类实现了这个应用程序中大部分重要的功能。它负责从相关元数据元素中提取值，并将它们保存供日后使用。在这个类中，我们仍需对映射到用户界面上特殊字段的所有不同键值做标准化操作。让我们先看一下代码清单3-5所示的THMetadataItem接口。

代码清单 3-5　THMetadataItem 接口

```
#import <AVFoundation/AVFoundation.h>

@class THGenre;

@interface THMetadata : NSObject

@property (copy) NSString *name;
@property (copy) NSString *artist;
@property (copy) NSString *albumArtist;
@property (copy) NSString *album;
@property (copy) NSString *grouping;
@property (copy) NSString *composer;
@property (copy) NSString *comments;
@property (strong) NSImage *artwork;
@property (strong) THGenre *genre;

@property NSString *year;
@property NSNumber *bpm;
@property NSNumber *trackNumber;
@property NSNumber *trackCount;
@property NSNumber *discNumber;
@property NSNumber *discCount;

- (void)addMetadataItem:(AVMetadataItem *)item withKey:(id)key;
- (NSArray *)metadataItems;

@end
```

该接口定义了多种可在屏幕上呈现的数据值。每个属性都对应用户接口中的一个特殊字段。之后程序定义了addMetadataItem:方法，向其内部集合添加一个AVMetadataItem。最后定义了一个metadataItems方法用于获取已经更新的元数据，这样用户就可以对所发生的变更进行保存。让我们转到具体的实现中(如代码清单3-6所示)。

代码清单 3-6　THMetadata 方法的实现

```
#import "THMetadata.h"
#import "THMetadataConverterFactory.h"

@interface THMetadata ()
```

```
@property (strong) NSDictionary *keyMapping;
@property (strong) NSMutableDictionary *metadata;
@property (strong) THMetadataConverterFactory *converterFactory;
@end

@implementation THMetadata

- (id)init {
    self = [super init];
    if (self) {
        _keyMapping = [self buildKeyMapping];                          // 1
        _metadata = [NSMutableDictionary dictionary];                  // 2
        _converterFactory = [[THMetadataConverterFactory alloc] init]; // 3
    }
    return self;
}

- (NSDictionary *)buildKeyMapping {
    return @{
        // Name Mapping
        AVMetadataCommonKeyTitle : THMetadataKeyName,

        // Artist Mapping
        AVMetadataCommonKeyArtist : THMetadataKeyArtist,
        AVMetadataQuickTimeMetadataKeyProducer : THMetadataKeyArtist,

        // Album Artist Mapping
        AVMetadataID3MetadataKeyBand : THMetadataKeyAlbumArtist,
        AVMetadataiTunesMetadataKeyAlbumArtist : THMetadataKeyAlbumArtist,
        @"TP2" : THMetadataKeyAlbumArtist,

        // Continued mappings...
        ...
    };
}
```

(1) 要完成对不同键空间的键和格式的标准化工作，需要创建一个从指定格式键到标准化键的映射。标准化的键常量在THMetadataKeys.h中定义。代码清单3-6给出了buildKeyMappings方法的缩略版本。要了解所有键值的映射关系，一定要查看源代码中程序的完整版本。

(2) 创建一个NSMutableDictionary来保存从AVMetadataItem值提取到的展示内容。保存在这个字典对象中的值就是最后在屏幕上呈现的值。

(3) 创建一个THMetadataConverterFactory实例来声明THMetadataConverter实例。我们要创建的这个转换对象用于在保存于AVMetadataItem中的数据和呈现在屏幕上的值之间进行转换。

接下来看一下addMetadataItem:方法的实现(如代码清单3-7所示)。

代码清单 3-7 addMetadataItem:的实现

```
- (void)addMetadataItem:(AVMetadataItem *)item withKey:(id)key {

    NSString *normalizedKey = self.keyMapping[key];                    // 1

    if (normalizedKey) {
```

```
id <THMetadataConverter> converter =                                    // 2
    [self.converterFactory converterForKey:normalizedKey];

// Extract the value from the AVMetadataItem instance and
// convert it into a format suitable for presentation.
id value = [converter displayValueFromMetadataItem:item];

// Track and Disc numbers/counts are stored as NSDictionary
if ([value isKindOfClass:[NSDictionary class]]) {                       // 3
     NSDictionary *data = (NSDictionary *)value;
             for (NSString *currentKey in data) {
                  [self setValue:data[currentKey] forKey:currentKey];
             }
             } else {
                [self setValue:value forKey:normalizedKey];
             }

             // Store the AVMetadataItem away for later use
             self.metadata[normalizedKey] = item;                       // 4
    }
}
```

(1) 在这个方法中，我们首先为传进来的键找到标准化的键。也就是将指定格式的键映射为标准化的键。如果返回得到标准化的值，则继续处理AVMetadataItem。

(2) 之后在converterFactory中请求能够处理当前AVMetadataItem的THMetadataConverter对象。通过转换器获取当前元数据项对应的展示值，即将元数据的值通过提取和转换使其成为能够用于展示的格式。

(3) 当得到一个正确的返回值之后，需要判断它的类型。唱片和曲目编号是一种特殊情况，需要其他一些操作才能提取它们的值。最终使用键值编码方法setValue:forKey来设置每个独立的属性值。

(4) 最后，将AVMetadataItem保存在metadata字典中以备日后使用。

如果还对这个流程感到不是很理解也不必担心，开始创建转换器后，我们会将这些复杂的步骤分解成一个个小的部分来实现。让我们开始创建这些对象吧。

3.6.3　数据转换器

在处理AVMetadataItem时，其中最具有挑战性及最难懂的部分就是理解保存在其value属性中的数据。当value是一个简单字符串时，比如艺术家名字或唱片标题，或是数字类型，比如录制年份或心律值(BPM)，这样的数据很容易理解并且不需要转换。不过我们经常会遇到一些看起来很混乱的值，或者仅仅是一些晦涩的内容，这就意味着需要通过一些额外努力将这些值转换为可以展示的格式。可在THMetadata类中直接加入转换逻辑，但是这样的话代码量会快速增加，导致不易维护。所以正确的做法是将这些逻辑语句分离到几个转换类中，当然这些转换类都要遵循THMetadataConverter协议，如代码清单3-8所示。

代码清单 3-8 THMetadataConverter 协议

```
#import <AVFoundation/AVFoundation.h>

@protocol THMetadataConverter <NSObject>

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item;

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                withMetadataItem:(AVMetadataItem *)item;
@end
```

THMetadataConverter协议定义了两个方法。其中displayValueFromMetadataItem:方法用于解析AVMetadataItem实例中的数据并将其转换为可展示的格式。它的同级方法metadataItemFromDisplayValue:withMetadataItem:可将展示内容转换为之前的AVMetadataItem格式，使其始终可以保存到基础媒体中。转换对象的类接口全部遵循代码清单3-9所示的模式，所以接下来的代码清单只给出类的实现部分。

代码清单 3-9 THMetadataConverter 接口模板

```
#import "THMetadataConverter.h"

@interface ClassName : NSObject <THMetadataConverter>

@end
```

3.6.4 简单转换

上述协议的默认实现方法很简单，仅提供了一个到AVMetadataItem值的通道。这个类主要用于转换简单的字符串和数字值。代码清单3-10给出了这个转换器的具体实现。

代码清单 3-10 THDefaultMetadataConverter 的实现

```
# import "THDefaultMetadataConverter.h"

@implementation THDefaultMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    return item.value;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                withMetadataItem:(AVMetadataItem *)item {
    AVMutableMetadataItem *metadataItem = [item mutableCopy];
    metadataItem.value = value;
    return metadataItem;
}

@end
```

将AVMetadataItem值转换为对应的展示内容时，需要返回项的value属性。这对那些value属性为简单字符串或数字值的元数据项是适用的。当由展示内容转换回AVMetadataItem值时，首先需要创建一个原始AVMetadataItem的可变副本，这将返回一个AVMutableMetadataItem

值。当创建新的元数据或修改已存在的元数据时，需要使用AVMetadataItem的可变子类。它具有与父类一样的基本接口，但它重新将其中的一些只读属性定义为读写属性。通过复制原始AVMetadataItem所创建的AVMutableMetadataItem是一个新实例，实例中所有重要的属性都和原始项的值一样，只需要将它的value属性设置为传入方法的值即可。

这只是我们所说的简单转换，在后面马上会接触到很多复杂的场景。下面让我们看一下如何转换元数据项的Artwork。

3.6.5　转换 Artwork

媒体Artwork元数据会以多种不同的格式返回，比如唱片的封面和电影的海报等。不过最终Artwork数据的字节都保存在一个NSData实例中，但是定位NSData实例有时候需要完成一些额外工作。代码清单3-11给出了THArtworkMetadataConverter的实现。

代码清单 3-11　THArtworkMetadataConverter 的实现

```
#import "THArtworkMetadataConverter.h"

@implementation THArtworkMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    NSImage *image = nil;
    if ([item.value isKindOfClass:[NSData class]]) {                    // 1
        image = [[NSImage alloc] initWithData:item.dataValue];
    }
    else if ([item.value isKindOfClass:[NSDictionary class]]) {         // 2
        NSDictionary *dict = (NSDictionary *)item.value;
        image = [[NSImage alloc] initWithData:dict[@"data"]];
    }
    return image;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                          withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    NSImage *image = (NSImage *)value;
    metadataItem.value = image.TIFFRepresentation;                      // 3

    return metadataItem;
}

@end
```

(1) 如果AVMetadataItem返回一个NSData类型的值，我们就需要根据保存在该对象中的字节创建一个新的NSImage。NSImage和UIImage都可以提供初始化器，使我们可从NSData直接创建一个图片实例。

(2) 如果选中的对象是MP3，就需要对它进行深入地挖掘以得到图片字节。这种情况下的value属性可能是一个NSDictionary，所以我们适当地转换值并检索对应的data键，这样就可以得到保存图片字节的NSData。

(3) 将展示内容转换回AVMutableMetadataItem实例的实现过程相当简单，这是由于AV Foundation无法写入ID3元数据。可认为这个值就是以NSData格式保存的并向NSImage请求TIFFRepresentation。如果需要存储PNG或JPG格式的图片数据，可以使用NSBitmapImageRep、UIImage方法或Quartz框架将该数据转换为所需的类型。

3.6.6 转换注释

获取媒体项注释的方法相当直接。如果处理的对象是MPEG-4或QuickTime内容，可以获取AVMetadataItem的stringValue属性。MP3同样需要一些额外方法来获取注释数据，如代码清单3-12所示。

代码清单 3-12 THCommentMetadataConverter 的实现

```
#import "THCommentMetadataConverter.h"

@implementation THCommentMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSString *value = nil;
    if ([item.value isKindOfClass:[NSString class]]) {                    // 1
        value = item.stringValue;
}

    else if ([item.value isKindOfClass:[NSDictionary class]]) {           // 2
        NSDictionary *dict = (NSDictionary *)item.value;
        if ([dict[@"identifier"] isEqualToString:@""]) {
            value = dict[@"text"];
        }
    }
    return value;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                            withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];             // 3
    metadataItem.value = value;
    return metadataItem;
}

@end
```

(1) 如果条目的value属性为NSString，可以获取它的stringValue属性。这种情况适用于MPEG-4和QuickTime媒体。

(2) MP3的注释保存在一个定义ID3 COMM帧的NSDictionary中(如果处理的是ID3v2.2，则为COM)。所有类型的值都保存在这个帧中。比如，iTunes在这个帧中保存音频标准化和无缝播放设置等，意味着当请求ID3元数据时需要接收多个COMM帧。包含实际注释内容的特定COMM帧被存储在一个带有空字符串标识符的帧中。找到需要的条目后，通过请求text键来检索注释。

(3) 将展示内容转换回AVMetadataItem很简单，因为这个过程不涉及ID3数据，所以只需复制原始AVMetadataItem并设置value属性为传入方法的值即可。

3.6.7　转换音轨数据

音轨通常包含一首歌曲在整个唱片中的编号位置信息(比如："12首歌曲中的第4首"这样的信息)。MP3文件以一种简单易懂的方式保存此信息，但是M4A就有些复杂了，并且需要一些处理才能生成用于展示的内容。代码清单3-13给出了THTrackMetadataConverter类的实现。

代码清单 3-13　THTrackMetadataConverter 的实现

```
#import "THTrackMetadataConverter.h"
#import "THMetadataKeys.h"

@implementation THTrackMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSNumber *number = nil;
    NSNumber *count = nil;

    if ([item.value isKindOfClass:[NSString class]]) {                    // 1
        NSArray *components =
            [item.stringValue componentsSeparatedByString:@"/"];
        number = @([components[0] integerValue]);
        count = @([components[1] integerValue]);
    }
    else if ([item.value isKindOfClass:[NSData class]]) {                 // 2
        NSData *data = item.dataValue;
        if (data.length == 8) {
            uint16_t *values = (uint16_t *) [data bytes];
            if (values[1] > 0) {
                number = @(CFSwapInt16BigToHost(values[1]));              // 3
            }
            if (values[2] > 0) {
                count = @(CFSwapInt16BigToHost(values[2]));               // 4
            }
        }
    }

    NSMutableDictionary *dict = [NSMutableDictionary dictionary];         // 5
    [dict setObject:number ?: [NSNull null] forKey:THMetadataKeyTrackNumber];
    [dict setObject:count ?: [NSNull null] forKey:THMetadataKeyTrackCount];

    return dict;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];
```

```
    NSDictionary *trackData = (NSDictionary *)value;
    NSNumber *trackNumber = trackData[THMetadataKeyTrackNumber];
    NSNumber *trackCount = trackData[THMetadataKeyTrackCount];

    uint16_t values[4] = {0};                                          // 6

    if (trackNumber && ![trackNumber isKindOfClass:[NSNull class]]) {
        values[1] = CFSwapInt16HostToBig([trackNumber unsignedIntValue]);   // 7
    }

    if (trackCount && ![trackCount isKindOfClass:[NSNull class]]) {
        values[2] = CFSwapInt16HostToBig([trackCount unsignedIntValue]);    // 8
    }

    size_t length = sizeof(values);
    metadataItem.value = [NSData dataWithBytes:values length:length];      // 9

    return metadataItem;
}

@end
```

(1) MP3音轨信息以“xx/xx” 字符串的格式返回。比如在一个包含10首歌曲的唱片中的第8首歌曲，得到的返回字符串值应该为“8/10”。这种情况下，我们需要使用NSString的componentsSeparatedByString:方法将值分开，得到单独值并将它们转换为相应的数值对象，将值打包为一个NSNumber实例。

(2) 对于M4A文件，这个值有些神秘。如果在控制台打印输出元数据条目的值，所得到的结果是NSData，使用诸如<00000008 000a0000>的值。这是4个16位big endian数字数组的十六进制表现形式。数组中的第2个和第3个元素分别包含音轨编号和音轨计数值。

(3) 如果音轨编号不等于0，使用CFSwapInt16BigToHost()函数进行endian变换将其转换为little endian格式，并将这个值打包为一个NSNumber实例。

(4) 同样，如果音轨计数值不等于0，则获取该值并执行endian转换，最后将结果值打包为一个NSNumber实例。

(5) 将得到的编号和计数值保存在一个NSDictionary实例中，将它返回给调用者。需要检查每个值是否为nil，如果有的值为nil，则使用NSNull实例替换。

(6) 将展示内容转换回AVMetadataItem所需的格式意味着需要将之前的步骤反过来，即把在displayValueFromMetadataItem:方法中提取这些值的过程反过来。需要创建一个包含4个uint16_t值的数组来保存音轨编号和计数值。

(7) 如果得到一个有效的音轨编号，将字节转换为big endian格式并将其保存在数组的第二个位置。

(8) 如果得到一个有效的音轨总数，将字节转换为big endian格式并将其保存在数组的第三个位置。

(9) 最后，将这些values数组打包为一个NSData实例，并将其设置为元数据项的值。

现在我们知道了处理音轨计数信息并不是那么简单，不过现在我们已经知道其中的奥妙了！

3.6.8　转换唱片数据

唱片计数信息用于表示一首歌曲所在的CD是所有唱片中的第几张唱片。大部分情况下这个值都是1/1，即只有一张唱片，不过如果这首歌曲所属的唱片属于一个唱片集合的一部分，则这个值可能就会发生变化。比如，如果你拥有Led Zeppelin的Complete Studio Recordings(很值得拥有的唱片)，则Led Zeppelin IV这张唱片的唱片值就应该是4/10。好消息是我们已经知道如何获取这个数据，因为它与上一节中的音轨编号和音轨计数数据的保存方法几乎一样，所以它的类实现也十分相似。代码清单3-14给出了THDiscMetadataConverter类的实现。

代码清单 3-14　THDiscMetadataConverter 的实现

```
#import "THDiscMetadataConverter.h"
#import "THMetadataKeys.h"

@implementation THDiscMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSNumber *number = nil;
    NSNumber *count = nil;

    if ([item.value isKindOfClass:[NSString class]]) {                    // 1
        NSArray *components =
            [item.stringValue componentsSeparatedByString:@"/"];
        number = @([components[0] integerValue]);
        count = @([components[1] integerValue]);
    }
    else if ([item.value isKindOfClass:[NSData class]]) {                 // 2
        NSData *data = item.dataValue;
        if (data.length == 6) {
            uint16_t *values = (uint16_t *)[data bytes];
            if (values[1] > 0) {
                number = @(CFSwapInt16BigToHost(values[1]));              // 3
            }
            if (values[2] > 0) {
                count = @(CFSwapInt16BigToHost(values[2]));               // 4
            }
        }
    }

    NSMutableDictionary *dict = [NSMutableDictionary dictionary];         // 5
    [dict setObject:number ?: [NSNull null] forKey:THMetadataKeyDiscNumber];
    [dict setObject:count ?: [NSNull null] forKey:THMetadataKeyDiscCount];

    return dict;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];
```

```
    NSDictionary *discData = (NSDictionary *)value;
    NSNumber *discNumber = discData[THMetadataKeyDiscNumber];
    NSNumber *discCount = discData[THMetadataKeyDiscCount];

    uint16_t values[3] = {0};                                          // 6

    if (discNumber && ![discNumber isKindOfClass:[NSNull class]]) {
        values[1] = CFSwapInt16HostToBig([discNumber unsignedIntValue]);   // 7
    }

    if (discCount && ![discCount isKindOfClass:[NSNull class]]) {
        values[2] = CFSwapInt16HostToBig([discCount unsignedIntValue]);    // 8
    }

    size_t length = sizeof(values);
    metadataItem.value = [NSData dataWithBytes:values length:length];      // 9

    return metadataItem;
}

@end
```

(1) MP3的唱片信息以“xx/xx”格式的字符串返回。如果一首歌曲所属唱片是10个唱片集合中的第4张唱片，那么得到的字符串值就为4/10。这时我们使用NSString的components-SeparatedByString:方法将值分开，并将它们转换为数值类型，打包成一个NSNumber实例。

(2) iTunes M4A文件的唱片信息保存在一个NSData中，NSData包含3个16位big endian数字。数组中的第2个和第3个元素分别保存唱片编号和唱片计数值。

(3) 如果唱片编号不等于0，则获取该值并使用CFSwapInt16BigToHost()函数执行endian转换，转换为little endian，并打包成一个NSNumber。

(4) 同样，如果唱片计数值不等于0，则获取该值并在字节上执行endian转换，将得到的值打包为一个NSNumber。

(5) 将所得到的编号和计数值保存在一个NSDictionary中，将它返回到调用者。需要检查每个值是否为nil，如果有的值为nil，则使用NSNull实例替换。

(6) 将展示内容转换回AVMetadataItem所需的格式意味着需要将之前的步骤反过来，即把在displayValueFromMetadataItem:方法中提取这些值的过程反过来。需要创建一个包含3个uint16_t值的数组来保存唱片编号和唱片计数值。

(7) 如果得到一个有效的唱片编号，就将字节转换为big endian格式并将其保存在数组的第二个位置。

(8) 如果得到一个有效的唱片计数值，将字节转换为big endian格式并将其保存在数组的第三个位置。

(9) 最后，将这些values数组打包为一个NSData实例，并将其设置为元数据项的值。

3.6.9 转换风格数据

处理风格(genre)数据具有一定挑战。困难并非来自于数据本身的复杂性，而是风格具有多种不同的格式。比如预定义风格、用户风格、风格ID、音乐风格、电影风格等。另一个

挑战在于风格信息的保存不止一种方法，即使对于文件类型也同样如此。我们看一下相关的基础知识。

对数字音频使用的标准风格分类最初来自于MP3。ID3规范定义了80个默认的风格类型及另外46个WinAmp扩展，一共126个风格。你可能还会找到各种工具所支持的特有风格类型，不过这些都不属于正式格式。

由于在这一点上MP3的主导地位明显，iTunes没有创建一套自己的音乐风格划分，而是基本遵循了ID3的风格分类，不过做了点小变化。iTunes音乐风格的标号比相应的ID3标识符大1。比如表3-1给出了ID3集合中的前5种风格以及相应的iTunes值。

表 3-1　音乐风格

ID3 风格	iTunes 风格
0: Blues	1: Blues
1: Classic Rock	2: Classic Rock
2: Country	3: Country
3: Dance	4: Dance
4: Disco	5: Disco

虽然iTunes使用了ID3集合中的预定义音乐风格类型，不过iTunes对电视、电影和有声读物等定义了自己的风格集。可在Apple’s Genre IDs Appendix[3]中找到完整的iTunes风格列表。

要简化风格数据的处理(节省代码的输入)，示例程序包含一个名为THGenre的类，它定义了标准的音乐风格和电视节目的风格。可以对应用程序中的音频及视频内容使用这些风格类型。

让我们学习一下有关风格的具体细节。代码清单3-15给出了THGenreMetadata- Converter类的实现。

代码清单 3-15　THGenreMetadataConverter 的实现

```
#import "THGenreMetadataConverter.h"
#import "THGenre.h"

@implementation THGenreMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    THGenre *genre = nil;

    if ([item.value isKindOfClass:[NSString class]]) {                    // 1
        if ([item.keySpace isEqualToString:AVMetadataKeySpaceID3]) {
            // ID3v2.4 stores the genre as an index value
            if (item.numberValue) {                                       // 2
                NSUInteger genreIndex = [item.numberValue unsignedIntValue];
                genre = [THGenre id3GenreWithIndex:genreIndex];
```

3. Genre IDs Appendix—http://www.apple.com/itunes/affiliates/resources/documentation/genre-mapping.html。

```
            } else {
                genre = [THGenre id3GenreWithName:item.stringValue];        // 3
            }
        } else {
            genre = [THGenre videoGenreWithName:item.stringValue];          // 4
        }
    }
    else if ([item.value isKindOfClass:[NSData class]]) {                   // 5
        NSData *data = item.dataValue;
        if (data.length == 2) {
            uint16_t *values = (uint16_t *)[data bytes];
            uint16_t genreIndex = CFSwapInt16BigToHost(values[0]);
            genre = [THGenre iTunesGenreWithIndex:genreIndex];
        }
    }
    return genre;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    THGenre *genre = (THGenre *)value;

    if ([item.value isKindOfClass:[NSString class]]) {                      // 6
        metadataItem.value = genre.name;
    }
    else if ([item.value isKindOfClass:[NSData class]]) {                   // 7
        NSData *data = item.dataValue;
        if (data.length == 2) {
            uint16_t value = CFSwapInt16HostToBig(genre.index + 1);
            size_t length = sizeof(value);
            metadataItem.value = [NSData dataWithBytes:&value length:length];
        }
    }

    return metadataItem;.
}

@end
```

(1) 有些格式将风格数据保存为一个NSString。有时该字符串的内容就是实际风格的名字，有时是风格在预定义风格列表中对应的索引值。

(2) 如果元数据项来自于ID 3键空间并且这个值可以强制转换为NSNumber，在使用ID3v2.4的情况下，可以取到该值的无符号整型值并用它作为索引来查找正确的THGenre实例。

(3) ID3的另一个变化是使用风格实际的名称进行保存，比如Jazz、Rock、Country及类似的情况。如果值为风格的名称，可以使用这个名称找到相应的THGenre实例。

(4) 对于其他以NSString类型保存的风格信息，可以将它们视为QuickTime电影或MPEG-4视频文件，这两种情况都直接保存风格的实际名称。可以从THGenre类查找到相应的

视频风格。

(5) 当使用其中一个预定义风格时，iTunes M4A音频会返回一个保存在NSData中的16位big endian数字。取到该值并将其转换为little endian格式，之后调用iTunesGenreWithIndex:方法返回相应的风格实例。

(6) 当转换回AVMetadataItem格式时，如果原始AVMetadataItem中保存的值是一个字符串，可以简单地将输入值赋给条目的value属性。

(7) 如果原始AVMetadataItem保存的值是一个NSData，通过将字节转换回big endian格式，并将结果包装为一个NSData值，将展示的内容转换回相应的格式。另外注意THGenre的索引属性基于0，因此为调整回iTunes索引，需要加1。

恭喜，有关转换的方法我们全部学习完了。现在可以运行程序，从列表中选择一个条目并观察它在用户界面中展示的内容。我们还剩下一个功能需要实现，就是让用户对这些变更进行保存。让我们回到上面的THMetadata类并完成其实现。

3.6.10　完成 THMetadata

我们还需要实现THMetadata的metadataItems方法。这个方法会获取所有当前保存的展示内容，并使用我们上面几节中创建的转换类将它们转换回AVMetadataItem格式。代码清单3-16给出了这个方法的实现。

代码清单 3-16　metadataItems 方法的实现

```
- (NSArray *)metadataItems {

    NSMutableArray *items = [NSMutableArray array];                    // 1

    // Add track number/count if applicable
    [self addMetadataItemForNumber:self.trackNumber                    // 2
                            count:self.trackCount
                       numberKey:THMetadataKeyTrackNumber
                        countKey:THMetadataKeyTrackCount
                         toArray:items];

    // Add disc number/count if applicable
    [self addMetadataItemForNumber:self.discNumber
                            count:self.discCount
                       numberKey:THMetadataKeyDiscNumber
                        countKey:THMetadataKeyDiscCount
                         toArray:items];

    NSMutableDictionary *metaDict = [self.metadata mutableCopy];       // 5
    [metaDict removeObjectForKey:THMetadataKeyTrackNumber];
    [metaDict removeObjectForKey:THMetadataKeyDiscNumber];

    for (NSString *key in metaDict) {

        id <THMetadataConverter> converter =
            [self.converterFactory converterForKey:key];

        id value = [self valueForKey:key];                             // 6
```

```
        AVMetadataItem *item =                                          // 7
            [converter metadataItemFromDisplayValue:value
                                  withMetadataItem:metaDict[key]];
        if (item) {
            [items addObject:item];
        }
    }

    return items;
}

- (void)addMetadataItemForNumber:(NSNumber *)number
                          count:(NSNumber *)count
                      numberKey:(NSString *)numberKey
                       countKey:(NSString *)countKey
                        toArray:(NSMutableArray *)items {

    id <THMetadataConverter> converter =
        [self.converterFactory converterForKey:numberKey];

    NSDictionary *data = @{numberKey : number ?: [NSNull null],         // 3
                           countKey : count ?: [NSNull null]};

    AVMetadataItem *sourceItem = self.metadata[numberKey];

    AVMetadataItem *item =                                              // 4
        [converter metadataItemFromDisplayValue:data
                                  withMetadataItem:sourceItem];
    if (item) {
        [items addObject:item];
    }

}
```

(1) 创建一个NSMutableArray实例，保存在本方法创建的元数据集合中。

(2) 音轨编号/计数及唱片编码/计数都需要额外的处理来将它们的值转换回AVMetadataItem格式，所以具体处理逻辑应该分解为不同方法，如应用程序中的编号3所示。

(3) 将成对的number和count封装到一个NSDictionary，并将原始AVMetadataItem传递到转换方法中。

(4) 从转换器获取元数据项，如果返回一个有效的元数据实例，则将它添加到items数组中。

(5) 创建一个包含不同显示值的内部metadata字典的副本，将音轨和唱片编号键从数组中删除，因为这些都是已经处理过的。遍历剩下的键值来创建相应的元数据项。

(6) 使用键/值编码方法valueForKey:查找当前键对应的值。

(7) 最后从当前转换方法中检索AVMetadataItem实例，如果返回一个有效实例，则将它添加到items集合中。

THMetadata类现在就全部完成了，我们也基本上完成了MetaManager应用程序。唯一剩下的任务就是实现THMediaItem上的saveWithCompletionHandler:方法。下面就开始实现它吧。

3.7　保存元数据

我们已经创建了THMetadata类及相关的转换对象，现在就可以读取元数据并将用户输入的内容转换回AVMetadataItem实例。不过还有一个重要的问题没有解决，由于AVAsset是一个不可变类，我们如何才能应用这些元数据的改变呢？答案是我们不直接修改AVAsset，而是使用一个名为AVAssetExportSession的类来导出一个新的资源副本以及元数据改动。在具体学习这个方法的实现之前，先看一下如何配置和使用AVAssetExportSession。

应用 AVAssetExportSession

AVAssetExportSession用于将AVAsset内容根据导出预设条件进行转码，并将导出资源写到磁盘中。AVAssetExportSession提供了多个功能来实现将一种格式转换为另一种格式、修订资源的内容、修改资源的音频和视频行为，当然还有我们最感兴趣的功能，即写入新的元数据。

创建一个AVAssetExportSession实例需要提供资源和导出预设。导出预设用于确定导出内容的质量、大小等属性。创建一个导出会话后，并且需要指定一个outputURL用于声明导出内容将要写入的地址，并且需要给出一个outputFileType值来表示将要写入的导出格式。最后调用exportAsynchronouslyWithCompletionHandler:方法开始导出过程。

在此并不对AVAssetExportSession的一些抽象概念进行讲述，而是直接来看saveWithCompletetionHandler:方法的实现。代码清单3-17给出了这个方法的具体实现。

代码清单 3-17　THMediaItem 的 saveWithCompletionHandler:的实现

```
- (void)saveWithCompletionHandler:(THCompletionHandler)handler {

    NSString *presetName = AVAssetExportPresetPassthrough;                    // 1
    AVAssetExportSession *session =
        [[AVAssetExportSession alloc] initWithAsset:self.asset
                                         presetName:presetName];

    NSURL *outputURL = [self tempURL];                                        // 2
    session.outputURL = outputURL;
    session.outputFileType = self.filetype;
    session.metadata = [self.metadata metadataItems];                         // 3

    [session exportAsynchronouslyWithCompletionHandler:^{
        AVAssetExportSessionStatus status = session.status;
        BOOL success = (status == AVAssetExportSessionStatusCompleted);
        if (success) {                                                        // 4
            NSURL *sourceURL = self.url;
            NSFileManager *manager = [NSFileManager defaultManager];
            [manager removeItemAtURL:sourceURL error:nil];
            [manager moveItemAtURL:outputURL toURL:sourceURL error:nil];
            [self reset];                                                     // 5
        }
        if (handler) {
            handler(success);
```

```
        }
    }];
}

- (NSURL *)tempURL {
    NSString *tempDir = NSTemporaryDirectory();
    NSString *ext = [[self.url lastPathComponent] pathExtension];
    NSString *tempName = [NSString stringWithFormat:@"temp.%@", ext];
    NSString *tempPath = [tempDir stringByAppendingPathComponent:tempName];
    return [NSURL fileURLWithPath:tempPath];
}

- (void)reset {
    _prepared = NO;
    _asset = [AVAsset assetWithURL:self.url];
}
```

(1) 首先使用AVAssetExportPresetPassthrough预设值创建一个AVAssetExportSession。AVAssetExportSession具有多个预设选项，不过AVAssetExportPresetPassthrough预设值可以让我们在不需要重新对媒体编码的前提下实现写入元数据的功能。在将实际媒体内容进行转码时，“passthrough”导出过程的时间很短。

(2) 为临时文件创建一个写入磁盘的URL，该URL基于当前url属性值，只不过在文件名后面附上temp字样。

(3) 取得THMetadata实例的metadataItems属性，这一步会返回一个包含用户接口值状态的AVMutableMetadataItem实例数组。

(4) 在导出会话的completion handler中，首先要判断导出状态。如果导出状态为AVAssetExportSessionCompleted，意味着导出成功，需要删除旧资源，改用新导出的版本。本例中我们做的有些快而忽略了对于文件操作的错误处理程序。在实际的产品中需要对各种可能出现的问题进行处理。

(5) 最后调用私有的reset方法来重置媒体项的prepared状态并重新初始化基础资源。

MetaManager应用程序终于全部完成了！现在可以运行该程序并对已有的元数据项进行修改，并且还可以将这些改动保存到磁盘中。

注意：

AVAssetExportPresetPassthrough 预设值在一些特定场景中是有用的，并且对于演示类应用程序非常适合。不过要注意它有一定的限制，它确实允许修改 MPEG-4 和 QuickTime 容器中存在的元数据信息，不过它不可以添加新的元数据。添加元数据的唯一方法是使用转码预设值。此外，它不能用于修改 ID3 标签。框架不支持写入 MP3 数据，这也是为什么在演示程序中 MP3 文件是只读类型的原因。

3.8 小结

本章我们已经介绍了很多基础知识！本章的内容很多，不过最重要的是我们为之后内容的学习做好了充分准备。主要学习了AVAsset和AVAssetTrack类并理解了使用AVAsynchronous-

KeyValueLoading协议异步获取属性的重要性。深入研究AV Foundation基于AVMetadataItem和AVMutableMetadataItem类的元数据功能，并讨论了更底层的数据操作方法。最后，我们第一次接触AVAssetExportSession类并将它很好地用在MetaManager应用程序中。这绝对不是我们最后一次使用这些类，并且随着我们学习的深入还会涉及这些类在其他方面的应用。

现在我们已经为这趟AV Foundation学习之旅做好了充分准备。

3.9 挑战

本章对如何读取或写入最常见的元数据进行了详细介绍，但这还远远不够。复制媒体库中的一首歌曲或一部电影并在iTunes中对它进行完整的标注。使用十六进制编辑器查看数据是如何保存及保存在什么位置。当你开始接触更高级的AV Foundation用例时，掌握不同容器格式中的数据存储方式会大有帮助。

第4章

视频播放

视频是一种非常强大的媒体介质，主要用于通知、传播、教育和娱乐等方面。由于诸如YouTube和Vimeo这类共享视频网站的发展，近十年来人们使用视频的习惯发生了显著变化，尤其是最近随着iPhone和iPad等设备的推出这一变化更加迅速。电视机的地位很快就被这些流媒体视频所取代；在线教育网站和视频类虚拟教室随处可见，在Facebook和Twitter上每一分钟都进行着大量的视频分享。显然，为了迎合这一日益增长的需求，学习如何开发出功能强大、动态的视频应用程序变得非常重要。本章将深入研究AV Foundation的视频播放功能，学习创建一个自定义视频播放器所需的核心组件，并通过添加一些额外的功能提升产品的播放体验。

4.1 播放功能综述

当开发一个自定义播放器时会用到大量的对象。本节从一个较高层级的介绍入手，通过探究其所扮演的角色和所含类之间的关系来学习AV Foundation的播放功能。后面还会继续深入分析具体的API，并通过实际开发一个自定义视频播放器来实际使用这些类。图4-1概要显示了用到的类及其关系。

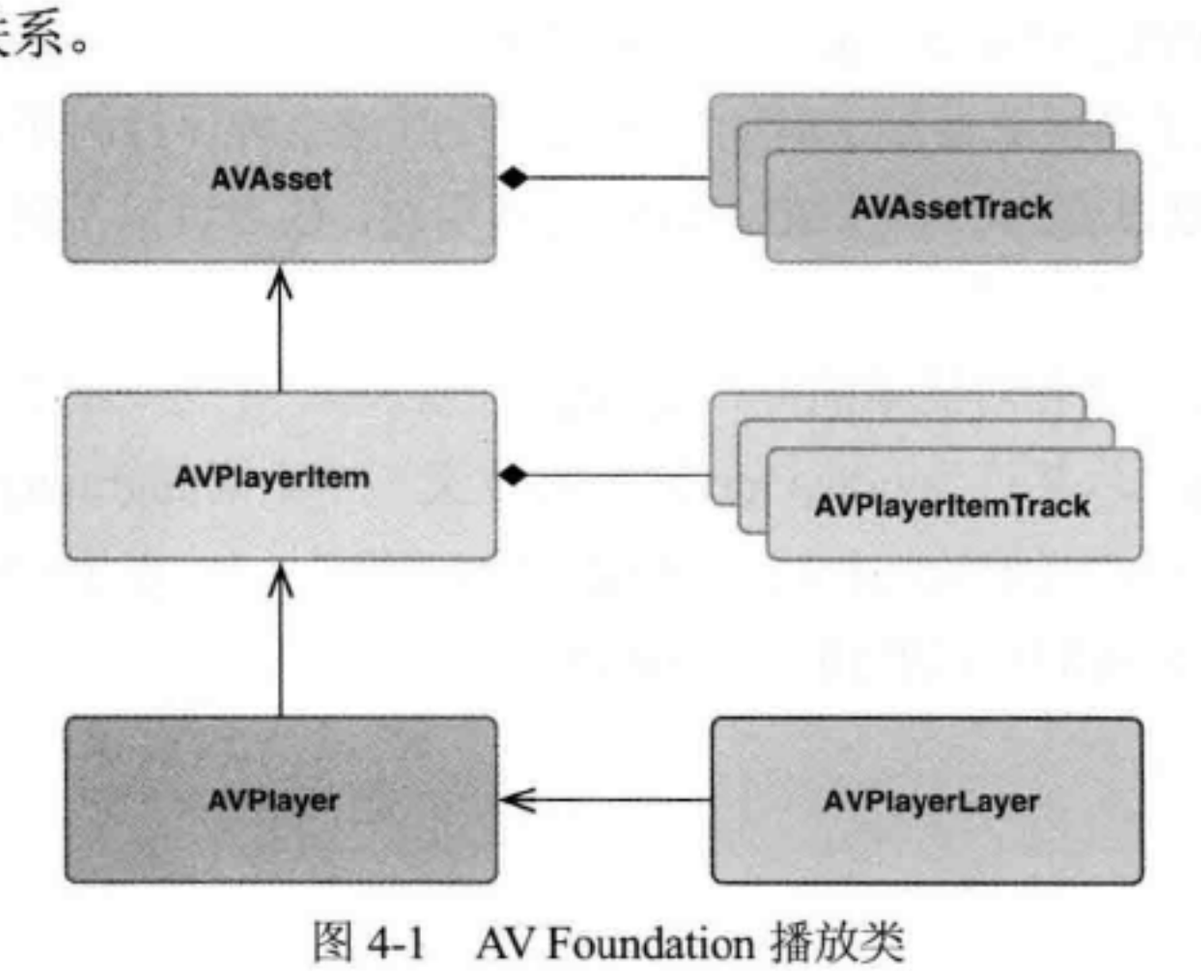

图 4-1　AV Foundation 播放类

4.1.1 AVPlayer

AV Foundation的播放都围绕AVPlayer类展开，AVPlayer是一个用来播放基于时间的视听媒体的控制器对象。支持播放从本地、分步下载或通过HTTP Live Streaming协议得到的流媒体，并在多种播放场景中播放这些视频资源。需要说明的是，当我们说“控制器”时，是指我们通常的理解，它不是一个视图或窗口控制器，而是一个对播放和资源时间相关信息进行管理的对象。开发者通过框架提供的应用程序接口来开发控制播放基于时间的媒体的用户界面。

AVPlayer是一个不可见组件。如果播放MP3或AAC音频文件，那么没有可视化的用户界面也不会有什么问题。不过如要播放一个QuickTime电影或一个MPEG－4视频，会导致非常不好的用户体验。要将视频资源导出到用户界面的目标位置，需要使用AVPlayerLayer类。

> **注意：**
>
> AVPlayer 只管理一个单独资源的播放，不过框架还提供了 AVPlayer 的一个子类 AVQueue-Player，可以用来管理一个资源队列。当你需要在一个序列中播放多个条目或者为音频、视频资源设置播放循环时可使用该子类。

4.1.2 AVPlayerLayer

AVPlayerLayer构建于Core Animation之上，是AV Foundation中能找到的为数不多的可见组件。Core Animation是Mac和iOS平台上负责图形渲染与动画的基础框架，主要用于这些平台资源的美化和动画流畅度提升。Core Animation本身具有基于时间的属性，并且由于它基于OpenGL，所以具有很好的性能，能非常好地满足AV Foundation的各种需要。

AVPlayerLayer扩展了Core Animation的CALayer类，并通过框架在屏幕上显示视频内容。这一图层并不提供任何可视化控件或其他附件(根据开发者需求搭建的)，但是它用作视频内容的渲染面。创建AVPlayerLayer需要一个指向AVPlayer实例的指针，这就将图层和播放器紧密绑定在一起，保证了当播放器基于时间的方法出现时使二者保持同步。AVPlayerLayer与其他CALayer一样，可以设置为UIView或NSView的备用层，或者可以手动添加到一个已有的层继承关系中。

AVPlayerLayer是一个相对简单的类，使用起来也简单。在这一层中开发者可以自定义的领域只有video gravity。总共可为videoGravity属性定义三个不同的gravity值，用来确定在承载层的范围内视频可以拉伸或缩放的程度。图4-2、图4-3和图4-4给出了一个16:9的视频置于4:3矩形范围内的情况，使我们可以看到不同gravity值。

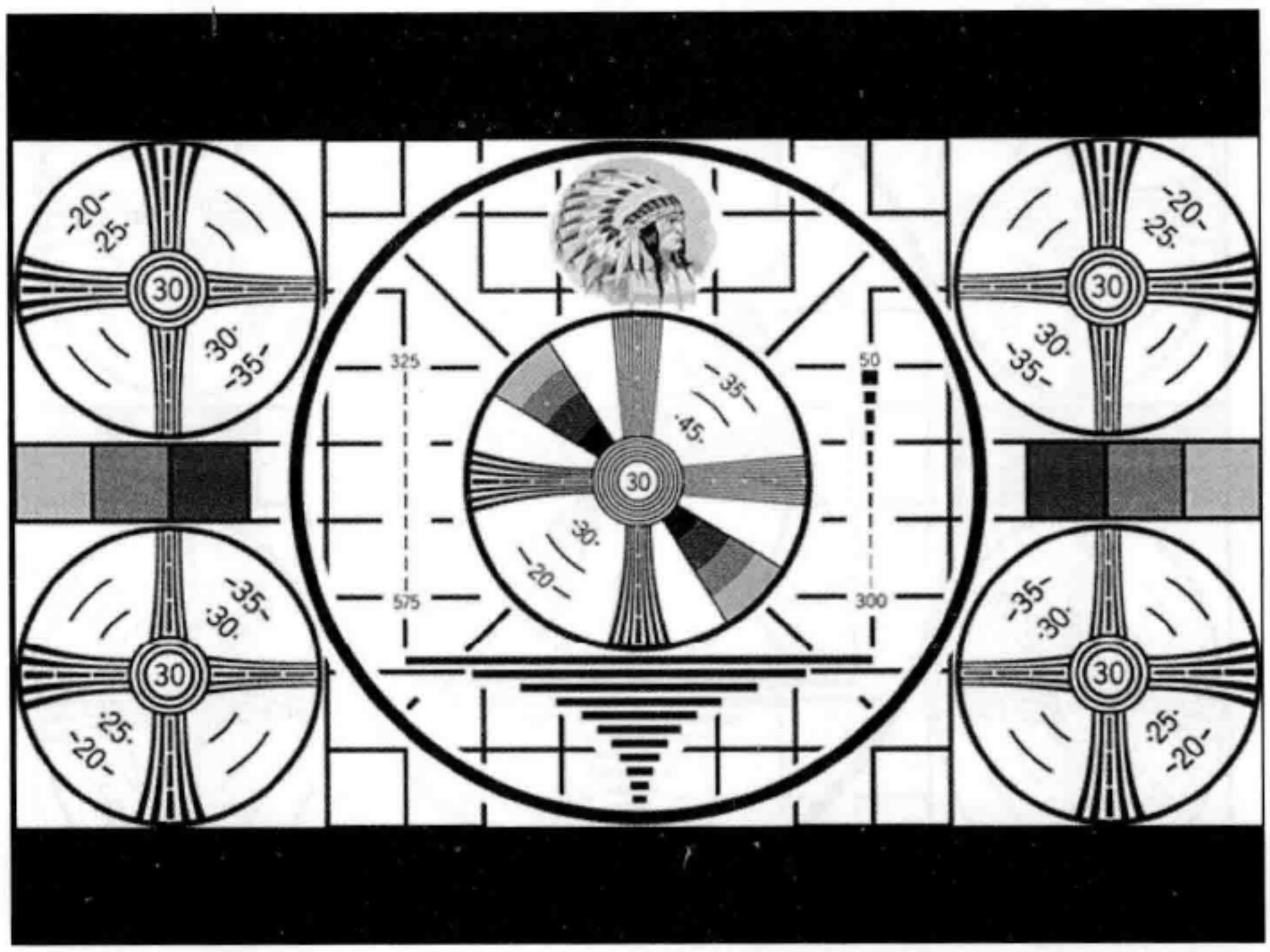

图 4-2 AVLayerVideoGravityResizeAspect 会在承载层的范围内缩放视频大小来保持视频的原始宽高比。这是在没有其他设置情况下的默认值，适用于大部分情况

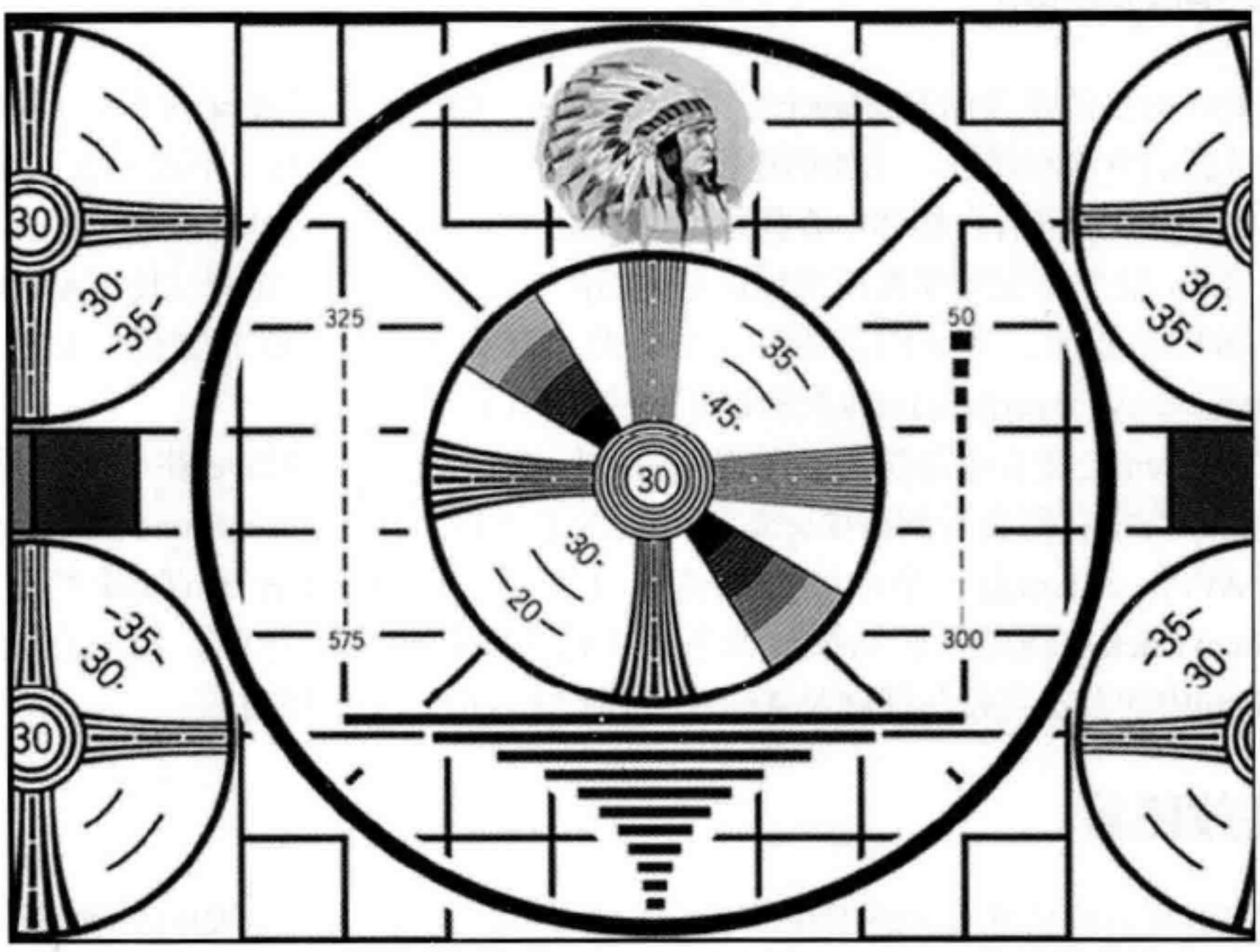

图 4-3 AVLayerVideoGravityResizeAspectFill 将保留视频的宽高比，并使其通过缩放填满层的范围区域，通常会导致视频图片被部分裁剪

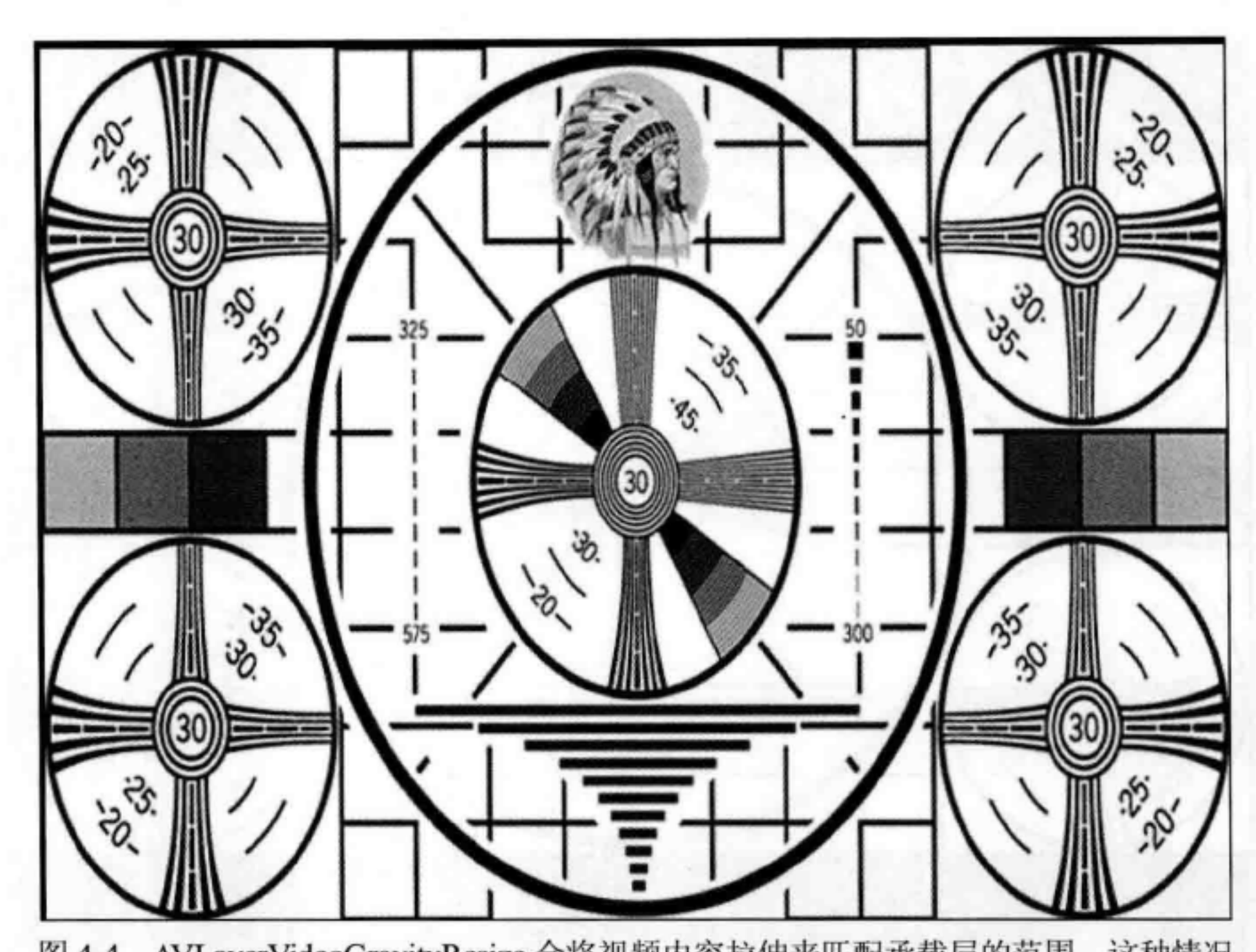

图 4-4 AVLayerVideoGravityResize 会将视频内容拉伸来匹配承载层的范围。这种情况最不常用，因为它通常会导致图片扭曲而导致的 funhouse effect 效应

4.1.3 AVPlayerItem

我们最终的目的是使用AVPlayer来播放AVAsset。如果查看AVAsset文档，可以找到一些用来获取数据的方法和属性，比如创建日期、元数据和时长等信息。不过无法查到如何获取当前时间的方法，也没有在媒体中查找特定位置的方法。这是因为AVAsset模型只包含媒体资源的静态信息，这些不变的属性用来描述对象的静态状态。这就意味着仅使用AVAsset对象是无法实现播放功能的。当我们需要对一个资源及其相关曲目进行播放时，首先需要通过AVPlayerItem和AVPlayerItemTrack类构建相应的动态内容。

AVPlayerItem会建立媒体资源动态视角的数据模型并保存AVPlayer在播放资源时的呈现状态。在这个类中我们会看到诸如seekToTime:的方法以及访问currentTime和presentationSize的属性。AVPlayerItem由一个或多个媒体曲目组成，由AVPlayerItemTrack类建立模型。AVPlayerItemTrack实例用于表示播放器条目中的类型统一的媒体流，比如音频或视频。AVPlayerItem中的曲目直接与基础AVAsset中的AVAssetTrack实例相对应。

4.2 播放秘籍

仅掌握这些类的简单概念还不够，下面通过一小段代码来看一下如何设置播放栈来播放保存在应用程序bundle中的视频。

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // 1. Define the asset URL
```

```
    NSURL *assetURL =
        [[NSBundle mainBundle] URLForResource:@"waves" withExtension:@"mp4"];

    // 2. Create an instance of AVAsset
    AVAsset *asset = [AVAsset assetWithURL:assetURL];

    // 3. Create an AVPlayerItem with a pointer to the asset to play
    AVPlayerItem *playerItem = [AVPlayerItem playerItemWithAsset:asset];

    // 4. Create an instance of AVPlayer with a pointer to the player item
    self.player = [AVPlayer playerWithPlayerItem:playerItem];

    // 5. Create a player layer to direct the video content
    AVPlayerLayer *playerLayer =
        [AVPlayerLayer playerLayerWithPlayer:self.player];

    // 6. Attach layer into layer hierarchy
    [self.view.layer addSublayer:playerLayer];
}
```

该示例中对播放视频文件所需的基础架构进行了设置。不过在实际播放视频内容前还需要一个额外步骤，这是因为播放器的播放控件还没有为播放动作做好准备。AVPlayerItem没有准备播放的界面，不过取而代之的是基于“主动发起请求”(“don’t call me, I’ll call you”)的机制。

AVPlayerItem具有一个名为status的AVPlayerItemStatus类型的属性。在对象创建之初，播放条目由AVPlayerItemStatusUnknown状态开始，该状态表示当前媒体还未载入并且还不在播放队列中。将AVPlayerItem与一个AVPlayer对象进行关联就开始将媒体放入队列中，但是在具体内容可以播放前，需要等待对象的状态由AVPlayerItemStatusUnknown变为AVPlayerItemStatusReadyToPlay。开发者可通过Key-Value Observing (KVO)机制监视status属性的值来跟踪这一变化过程。

KVO是由Foundation框架提供的Observer模式的由苹果公司给出的解决方案。可以让开发者注册一个对象作为其他对象状态的观察者。当被观察的对象状态发生变化时，观察对象就会得到通知并采取相应的动作。在将AVPlayerItem与AVPlayer关联之前，开发者需要将代码设置为status属性的观察者，如下面的示例所示。

```
static const NSString *PlayerItemStatusContext;

- (void)viewDidLoad {

    ...

    AVPlayerItem *playerItem = [AVPlayerItem playerItemWithAsset:asset];

    [playerItem addObserver:self
                forKeyPath:@"status"
                   options:0
                   context:&PlayerItemStatusContext];

    self.player = [AVPlayer playerWithPlayerItem:playerItem];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
```

```
                    ofObject:(id)object
                     change:(NSDictionary *)change
                    context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        AVPlayerItem *playerItem = (AVPlayerItem *)object;
        if (playerItem.status == AVPlayerItemStatusReadyToPlay) {
            // proceed with playback
        }
    }
}
```

当观察到播放控件的status变为AVPlayerItemStatusReadyToPlay时，就可以开始播放了。

4.3 处理时间

AVPlayer和AVPlayerItem都是基于时间的对象，但是在我们使用它们的功能前，需要了解在AV Foundation框架中呈现时间的方式。

人们倾向于用日子、小时、分钟和秒的方式表示时间。开发人员经常将时间进一步精确到毫秒和纳秒。所以用一个双精度浮点型数据表示时间也合情合理。实际上，回顾第2章中介绍的AVAudioPlayer，可以看到时间是以NSTimeInterval表示的，其实就是简单地对double值进行了typedef定义。不过使用浮点型数据类型表示时间存在一定问题，因为浮点型数据的运算会导致不精确的情况。当进行多时间计算累加时这些不精确的情况就会特别严重，经常导致时间的明显偏移，使得媒体的多个数据流几乎无法实现同步。此外，以浮点型数据呈现时间信息无法做到自我描述，这就导致在使用不同时间轴进行比较和运算时比较困难。AV Foundation使用一种可靠性更高的方法来展示时间信息，这就是基于CMTime数据结构。

CMTime

AV Foundation是基于Core Media的高层封装。Core Media是基于C的底层框架，提供了许多处理Mac和iOS媒体栈的关键功能。虽然这个框架通常都在后台工作，不过其中一个我们经常能够接触到的部分就是它的数据结构CMTime。CMTime为时间的正确表示给出了一种结构，即分数值的方式。具体定义如下：

```
typedef struct {
  CMTimeValue value;
    CMTimeScale timescale;
    CMTimeFlags flags;
    CMTimeEpoch epoch;
} CMTime;
```

这个结构最关键的两个值是value和timescale。value是一个64位整数值，timescale是一个32位整数值，在时间呈现样式中分别作为分子和分母。

建立以分数的格式处理时间数据的思维方式可能开始不太习惯，不过当开发者多使用几

次这种方式之后就会慢慢习惯。下面看几个示例，了解如何使用CMTimeMake函数创建时间。

```
// 0.5 seconds
CMTime halfSecond = CMTimeMake(1, 2);

// 5 seconds
CMTime fiveSeconds = CMTimeMake(5, 1);

// One sample from a 44.1 kHz audio file
CMTime oneSample = CMTimeMake(1, 44100);

// Zero time value
CMTime zeroTime = kCMTimeZero;
```

除对CMTime进行定义外，CMTime.h头文件还定义了大量实用的函数用于简化时间的处理。与大部分苹果公司的底层C框架一样，最好的参考资料就是头文件，所以这里建议大家仔细阅读CMTime.h头文件，了解其中定义的函数的功能。

4.4 创建视频播放器

本节通过创建一个iOS视频播放器(如图4-5所示)来深入学习AV Foundation播放API的细节。应用程序能播放本地和远程媒体，支持播放、暂停和拖动媒体时间轴。完成基本功能后，还需要对应用程序进一步优化以改善用户体验。可以在Chapter 4目录中找到名为VideoPlayer_Starter的示例项目。

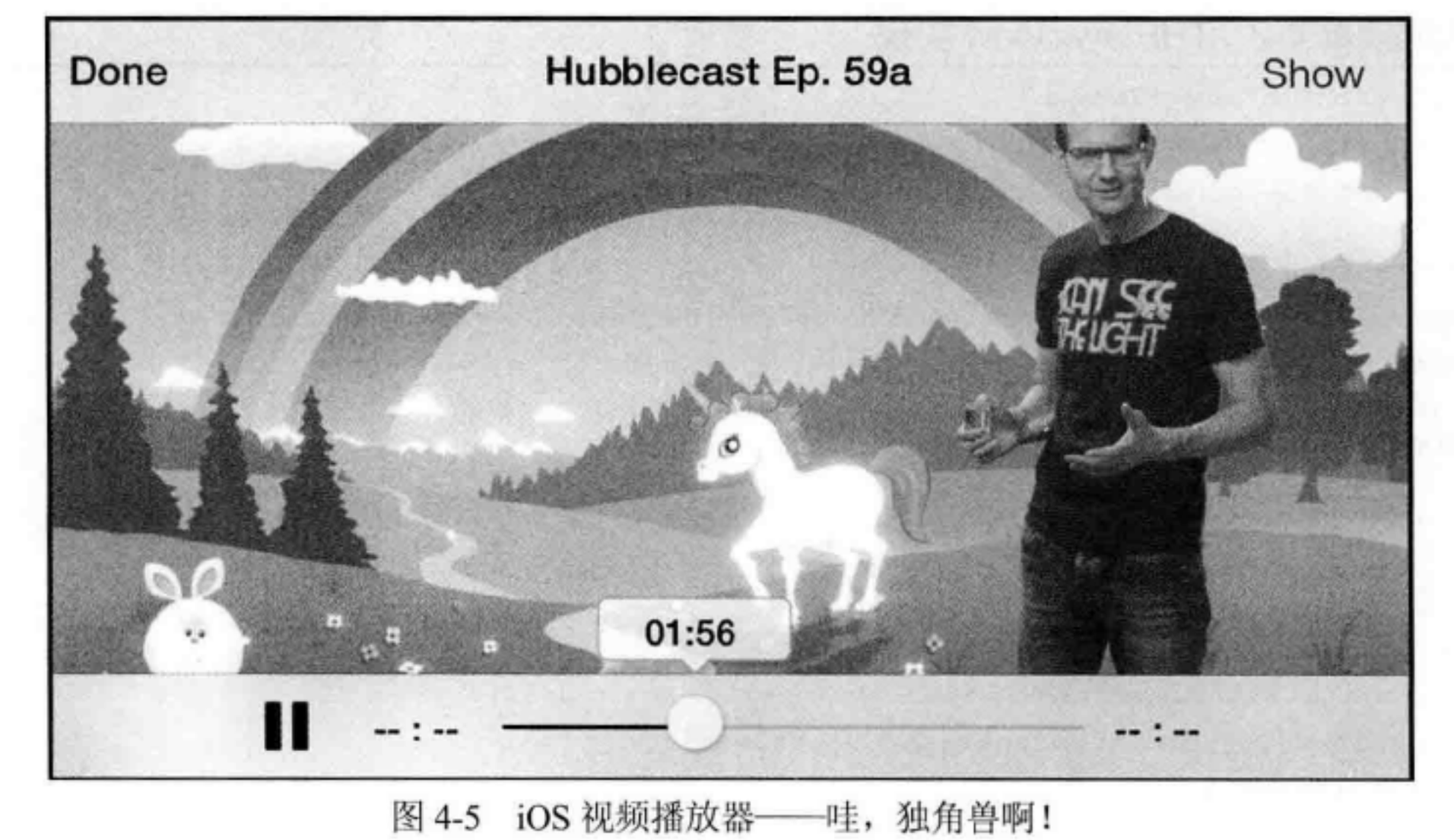

图 4-5 iOS 视频播放器——哇，独角兽啊！

4.4.1 创建视频视图

第一步需要创建一个用来在屏幕上展示视频内容的视图。在示例项目中的THVideoPlayer/Views文件组下面，可以找到一个名为THPlayerView的类。这个类就是用来展示视频内容并为操作视频播放提供用户界面的类。下面看一下这个类的接口，如代码清单4-1所示。

代码清单 4-1 THPlayerView 接口

```
#import "THTransport.h"

@class AVPlayer;

@interface THPlayerView : UIView

- (id)initWithPlayer:(AVPlayer *)player;

@property (nonatomic, readonly) id <THTransport> transport;

@end
```

这是一个仅带有几个方法的简单类。通过调用其initWithPlayer:初始化方法，并传递当前AVPlayer实例的引用进行实例化。这样就可以将播放器输出的视频直接展示在这个视图中，只读属性transport负责管理展示在视图中的可视化控件。在讲解应用程序播放控制器类的实现时会看到它是如何工作的。下面我们来看这个类的具体实现。

视图本身并不是视频输出的目标，相反，开发者需要将播放器输出指向一个AVPlayerLayer实例。可以手动创建层，并将它添加到视图的层继承关系中，但是在iOS平台下有一种更便捷的方法。UIView视图都受Core Animation层的支持，默认情况下，就是CALayer的通用实例，不过你可以通过在UIView中重写layerClass方法自定义支持层的类型，以便在实例化一个视图的时候返回一个要使用的自定义CALayer。在使用AVPlayerLayer对象时上述方法更加方便，因为不需要手动创建和操作层以及层继承关系。代码清单4-2给出了THPlayerView类的实现。

代码清单 4-2 THPlayerView 实现

```
#import "THPlayerView.h"
#import "THOverlayView.h"
#import <AVFoundation/AVFoundation.h>

@interface THPlayerView ()
@property (strong, nonatomic) THOverlayView *overlayView;                    // 1
@end

@implementation THPlayerView

+ (Class)layerClass {                                                         // 2
    return [AVPlayerLayer class];
}

- (id)initWithPlayer:(AVPlayer *)player {
    self = [super initWithFrame:CGRectZero];                                  // 3
    if (self) {
        self.backgroundColor = [UIColor blackColor];
        self.autoresizingMask = UIViewAutoresizingFlexibleHeight |
                                UIViewAutoresizingFlexibleWidth;

        [(AVPlayerLayer *)[self layer] setPlayer:player];                     // 4

        [[NSBundle mainBundle] loadNibNamed:@"THOverlayView"                  // 5
                                     owner:self
                                   options:nil];
```

```
        [self addSubview:_overlayView];
    }
    return self;
}

- (void)layoutSubviews {
    [super layoutSubviews];
    self.overlayView.frame = self.bounds;
}

- (id <THTransport>)transport {
    return self.overlayView;
}

@end
```

(1) 创建一个类扩展来定义一个私有属性用于保存指向THOverlayView视图实例的指针。这个类提供用户界面中操作视频播放的控件。

(2) 重写layerClass类方法返回一个AVPlayerLayer类。每当创建THPlayerView实例时，就会使用AVPlayerLayer作为它的支持层。

(3) 创建时没有给出默认的尺寸大小，所以开发者需要调用带有zero-sized框架的超类初始化方法。展示视图的视图控制器负责设置合适的框架。

(4) 这是该类中最关键的一行代码。我们希望获得传入初始化方法的AVPlayer实例并在AVPlayerLayer上对其进行设置。这一步将从AVPlayer输出的视频指向AVPlayerLayer实例。

(5) 在NIB中定义覆盖视图，通过调用loadNibNamed:owner:options方法创建视图实例。当视图创建完成并赋给overlayView属性后，将其作为子视图进行添加。

完成THPlayerView的实现后，下面将注意力转移到THPlayerController类。

4.4.2 创建视频控制器

在项目的THVideoPlayer/Controllers组下面，可找到THPlayerController类的具体实现代码。这个类为应用程序完成了很多功能，也是我们处理核心播放API方法的地方。代码清单4-3给出了这个类的接口。

代码清单 4-3 THPlayerController 接口

```
@interface THPlayerController : NSObject

- (id)initWithURL:(NSURL *)assetURL;

@property (strong, nonatomic, readonly) UIView *view;

@end
```

创建一个THPlayerController实例时，需要调用其initWithURL:方法，并传递需要播放的媒体的NSURL。AVPlayer可用来播放本地或流媒体，所以这个URL可以是本地文件URL，也可以是远程HTTP URL。该类还为相关视图提供了一个只读属性，以便客户端UIViewController可将视图添加到视图继承关系中。返回的视图是一个THPlayerView实例，不过由于这些细节

需要对客户端隐藏，所以返回一个通用UIView即可。

转过来看类的具体实现，首先创建一个类扩展来定义控制器的内部属性(如代码清单4-4所示)。

代码清单 4-4 THPlayerController 类扩展

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>
#import "THTransport.h"
#import "THPlayerView.h"
#import "AVAsset+THAdditions.h"
#import "UIAlertView+THAdditions.h"

// AVPlayerItem's status property
#define STATUS_KEYPATH @"status"

// Refresh interval for timed observations of AVPlayer
#define REFRESH_INTERVAL 0.5f

// Define this constant for the key-value observation context.
static const NSString *PlayerItemStatusContext;

@interface THPlayerController () <THTransportDelegate>

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) AVPlayerItem *playerItem;
@property (strong, nonatomic) AVPlayer *player;
@property (strong, nonatomic) THPlayerView *playerView;

@property (weak, nonatomic) id <THTransport> transport;

@property (strong, nonatomic) id timeObserver;
@property (strong, nonatomic) id itemEndObserver;
@property (assign, nonatomic) float lastPlaybackRate;

@end
```

在这个类的实现中，首先创建了一个类扩展来定义对象需要的存储属性。注意该扩展遵循THTransportDelegate协议并定义了一个transport属性。该类和THOverlayView之间会有很多交互操作，用来定义管理视频播放的用户界面。虽然这些类需要沟通，不过它们不必直接了解彼此。要断开这个关联，需要用到THTransport和THTransportDelegate协议(如代码清单4-5所示)。

代码清单 4-5 THTransport.h

```
@protocol THTransportDelegate <NSObject>

- (void)play;
- (void)pause;
- (void)stop;

- (void)scrubbingDidStart;
- (void)scrubbedToTime:(NSTimeInterval)time;
- (void)scrubbingDidEnd;
```

```
- (void)jumpedToTime:(NSTimeInterval)time;

@end

@protocol THTransport <NSObject>

@property (weak, nonatomic) id <THTransportDelegate> delegate;

- (void)setTitle:(NSString *)title;
- (void)setCurrentTime:(NSTimeInterval)time duration:(NSTimeInterval)duration;
- (void)setScrubbingTime:(NSTimeInterval)time;
- (void)playbackComplete;

@end
```

THOverlayView遵循THTransport协议，它可以为与覆盖视图进行通信提供正式接口。当播放栏(transport)发生变化时，比如用户改变时间轴位置或点击Play/Pause按钮，控制器对象会执行相应的委托回调。稍后将看到具体的实现过程，代码清单4-6给出了THPlayerController的实现。

代码清单 4-6 THPlayerController 实现

```
@implementation THPlayerController

#pragma mark - Setup

- (id)initWithURL:(NSURL *)assetURL {
    self = [super init];
    if (self) {
        _asset = [AVAsset assetWithURL:assetURL];                          // 1
        [self prepareToPlay];
    }
    return self;
}

- (void)prepareToPlay {
    NSArray *keys = @[@"tracks", @"duration", @"commonMetadata"];
    self.playerItem = [AVPlayerItem playerItemWithAsset:self.asset         // 2
                       automaticallyLoadedAssetKeys:keys];

    [self.playerItem addObserver:self                                      // 3
                  forKeyPath:STATUS_KEYPATH
                     options:0
                     context:&PlayerItemStatusContext];

    self.player = [AVPlayer playerWithPlayerItem:self.playerItem];         //4

    self.playerView = [[THPlayerView alloc] initWithPlayer:self.player];   //5
    self.transport = self.playerView.transport;
    self.transport.delegate = self;
}

// More methods to follow ...

@end
```

(1) 首先将URL传递给初始化方法来创建一个AVAsset。资源创建完成后，调用控制器的prepareToPlay方法来设置播放该资源所需的基础结构。

(2) 框架会自动载入资源的tracks属性，省去了通过AVAsynchronousKeyValueLoading协议手动载入该属性的过程。不过在以前，开发者仍然需要执行loadValuesAsynchronously-ForKeys:completionHandler:方法来载入需要访问的其他资源属性。iOS 7和Mac OS 10.9在AVPlayerItem的处理上有了大幅改进，通过使用新的初始化方法initWithAsset:automatically-LoadedAssetKeys:或playerItemWithAsset:automaticallyLoadedAssetKeys:创建一个AVPlayerItem实例，将任意属性集的载入委托给该框架。两种方式都将NSArray用作第二个参数，包含了随着AVPlayerItem在初始化队列中的载入过程所需的资源键。使用这个方法自动载入tracks、duration和commonMetadata属性。

(3) 添加self作为AVPlayerItem的status属性监听器。回顾一下创建过程，播放项开始时的status状态为AVPlayerItemStatusUnknown，播放项直到状态变为AVPlayerItemStatusReadyToPlay才可以开始播放。对status属性的键值观察可以让你监听变化。

(4) 为新创建的AVPlayerItem对象创建一个AVPlayer实例。AVPlayer会立即开始媒体队列化的过程。

(5) 最后，创建一个THPlayerView实例，传递给它一个指向AVPlayer实例的指针。开发者还需要为THPlayerController和THTransport设置关系。

4.4.3 监听状态改变

我们已经将THPlayerController设置为播放项的status属性的监听器。在对该属性监听前，需要实现observeValueForKeyPath:ofObject:change:context方法，如代码清单4-7所示。

代码清单 4-7 监听 status 属性

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        dispatch_async(dispatch_get_main_queue(), ^{                          // 1

            [self.playerItem removeObserver:self forKeyPath:STATUS_KEYPATH];

            if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {

                // Set up time observers.                                     // 2
                [self addPlayerItemTimeObserver];
                [self addItemEndObserverForPlayerItem];

                CMTime duration = self.playerItem.duration;

                // Synchronize the time display                               // 3
```

```
            [self.transport setCurrentTime:CMTimeGetSeconds(kCMTimeZero)
                                duration:CMTimeGetSeconds(duration)];

            // Set the video title.
            [self.transport setTitle:self.asset.title];                    // 4

            [self.player play];                                            // 5

        } else {
            [UIAlertView showAlertWithTitle:@"Error"
                                    message:@"Failed to load video"];
        }
    });
  }
}
```

(1) AV Foundation没有指定在哪个线程执行status改变通知，所以在采取下一步动作前，需要通过dispatch_async确保应用程序返回到主线程，向其传递一个主队列的引用。

(2) 通过调用私有方法addPlayerItemTimeObserver和addItemEndObserverForPlayerItem设置播放器的时间监听器。下面的小节会讨论这些方法和时间监听器。

(3) 在transport对象上设置当前时间和总长。将用户界面上展示的时间与播放的媒体进行同步。transport对象无法识别CMTime，只能处理以秒为单位的NSTimeInterval类型的时间。我们使用CMTimeGetSeconds函数将CMTime值转换为秒。Core Media定义了常量kCMTimeZero，开发者可以将它作为开头的currentTime参数，使用播放条目的duration属性值作为第二个参数。

(4) 向播放栏传递一个标题字符串，来展示资源的标题(如果资源的元数据中存在标题信息)。AVAsset没有title属性，这是我们加入AVAsset中的一个分类方法，目的是增加代码的可读性。这个分类方法用到了上一章介绍的元数据API，具体地讲，从资源的commonMetadata得到AVMetadataCommonKeyTitle值。具体细节参考AVAsset+THAdditions。

(5) 现在就准备调用AVPlayer的play方法进行播放了。最后，在完成对status关键路径的监听后，我们希望将作为监听器的self移除。

现在可以启动应用程序并开始播放其中一个视频。虽然视频已经播放，不过用户界面上的控件还没有提供任何功能，并且随着时间的推移用户界面也没有相应的反馈信息。这就又回到了addPlayerItemTimeObserver方法上，我们需要在该方法上实现相关的功能，不过在此之前我们需要先学习如何得知AVPlayer的时间变化。

4.5 时间监听

我们已经讨论过并了解到如何使用KVO来观察播放条目的status属性。KVO对于常见的状态监控表现得很出色，并且可以监听AVPlayerItem和AVPlayer的许多属性。不过KVO也有不能胜任的场景，比如需要监听AVPlayer的时间变化。这些监听类型都是自身具有明显的动态特性并需要非常高的精确度，这一点要比标准的键值监听要求高。为满足这一需求，AVPlayer提供了两种基于时间的监听方法，让应用程序可以对时间变化进行精准的监听。下面分别看一下这两个方法。

4.5.1 定期监听

通常情况下，我们希望以一定的时间间隔获得通知。如果需要随着时间的变化移动播放头位置或更新时间显示，这非常重要。利用AVPlayer的addPeriodicTimeObserverForInterval:queue:usingBlock:方法可以很容易地监听到此类变化。这个方法需要传递如下参数：

- **interv：** 一个用于指定通知周期间隔的 CMTime 值。
- **queue：** 通知发送的顺序调度队列。大多数时候，我们希望这些通知消息发生在主队列，在如果没有明确指定的情况下则默认为主队列。需要重点注意的是不可以使用并行调度队列，因为 API 没有处理并行队列的方法，否则会导致一些不可知的问题。
- **block：** 一个在指定的时间间隔中将会在队列上调用的回调块。这个块传递一个 CMTime 值用于指示播放器的当前时间。

4.5.2 边界时间监听

AVPlayer还提供了一种更有针对性的方法来监听时间，应用程序可以得到播放器时间轴中多个边界点的遍历结果。这一方法主要用于同步用户界面变更或随着视频播放记录一些非可视化数据。比如，可以定义25%、50%和75%边界的标记，以此判断用户播放进度。要使用这个功能，需要用到addBoundaryTimeObserverForTimes:queue:usingBlock:方法，并提供如下参数：

- times：CMTime 值组成的一个 NSArray 数组定义了需要通知的边界点。
- queue：与定期监听类似，为方法提供一个用来发送通知的顺序调度队列。指定 NULL 等同于明确设置主队列。
- block：每当正常播放中跨越一个边界点时就会在队列中调用这个回调块。有趣的是，该块不提供遍历的 CMTime 值，所以开发者需要为此执行一些额外计算进行确定。

本示例应用程序没有用到边界时间监听，不过定期监听对应用程序的功能非常重要。下面通过addPlayerItemTimeObserver方法的实现看一下如何在实际中使用定期监听法，如代码清单4-8所示。

代码清单 4-8 定期监听法

```
- (void)addPlayerItemTimeObserver {

    // Create 0.5 second refresh interval - REFRESH_INTERVAL == 0.5
    CMTime interval =
        CMTimeMakeWithSeconds(REFRESH_INTERVAL, NSEC_PER_SEC);          // 1

    // Main dispatch queue
    dispatch_queue_t queue = dispatch_get_main_queue();                 // 2

    // Create callback block for time observer
    __weak THPlayerController *weakSelf = self;                         // 3
    void (^callback)(CMTime time) = ^(CMTime time) {
        NSTimeInterval currentTime = CMTimeGetSeconds(time);
        NSTimeInterval duration = CMTimeGetSeconds(weakSelf.playerItem.duration);
        [weakSelf.transport setCurrentTime:currentTime duration:duration];  // 4
```

```
    };

    // Add observer and store pointer for future use
    self.timeObserver =                                                    // 5
        [self.player addPeriodicTimeObserverForInterval:interval
                                                  queue:queue
                                             usingBlock:callback];

}
```

> 注意：
>
> AV Foundation 使用较长的类名和方法名。与块连在一起，一行应用程序就会显得非常多。这个方法还可以写得更简洁，不过除非出版社想要这本书达到 14 英尺宽，否则我还是按上面格式撰写应用程序吧。不过这里建议大家在实际项目代码中采用更简洁的代码风格。

(1) 首先创建一个用于定义通知时间间隔的CMTime值。这里将间隔定义为0.5秒，这个时间粒度足以更新播放器的时间显示。

(2) 定义发送回调通知的调度队列。大多数情况下，由于我们所要更新的用户界面处于主线程，所以一般使用主队列。

(3) 定义一个回调块，在前面定义的时间周期内会调用该代码块。非常重要的一点是代码块要获取self的弱引用。不这样做会出现难以诊断的内存泄漏。

(4) 在回调块内部，我们希望通过CMTimeGetSeconds函数将代码块的CMTime值转换成一个NSTimeInterval。同样，还需要将播放条目的duration进行转换。传递这个duration信息看起来是多余的，因为我们已经在KVO回调中传递duration到transport中，不过transport会随着媒体的载入而改变，所以要保持用户界面的同步，最好还是传递最新的值。

(5) 最后调用addPeriodicTimeObserverForInterval:queue:usingBlock:方法并传递定义好的参数。调用这个方法会返回一个隐含id类型指针。对这些回调必须保持一个强引用。这个指针还会用于移除监听器。

4.5.3 条目结束监听

另一常见的需要监听的事件就是条目播放完毕的时间，虽然这不同于上面介绍的基于时间的监听，不过我们倾向于认为二者有着类似的原理。当播放完成时，AVPlayerItem会发送一个AVPlayerItemDidPlayToEndTimeNotification通知。THPlayerController实例应该注册为该通知的监听器，这样就可以采取相应的动作。代码清单4-9给出了addItemEndObserverForPlayerItem方法的实现。

代码清单 4-9 条目结束监听

```
- (void)addItemEndObserverForPlayerItem {

    NSString *name = AVPlayerItemDidPlayToEndTimeNotification;
```

```
    NSOperationQueue *queue = [NSOperationQueue mainQueue];

    __weak THPlayerController *weakSelf = self;                          // 1
    void (^callback)(NSNotification *note) = ^(NSNotification *notification) {
        [weakSelf.player seekToTime:kCMTimeZero                          // 2
                completionHandler:^(BOOL finished) {
            [weakSelf.transport playbackComplete];                       // 3
        }];
    };

    self.itemEndObserver =                                               // 4
        [[NSNotificationCenter defaultCenter] addObserverForName:name
                                                          object:self.playerItem
                                                           queue:queue
                                                       usingBlock:callback];
}

- (void)dealloc {
    if (self.itemEndObserver) {                                          // 5
        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
        [nc removeObserver:self.itemEndObserver
                    name:AVPlayerItemDidPlayToEndTimeNotification
                  object:self.player.currentItem];
        self.itemEndObserver = nil;
    }

}
```

(1) 在定义代码块之前，首先需要定义一个到self的弱引用。与定期监听使用的回调块类似，如果没有建立对self的弱引用将会导致内存泄漏。这些基于块的计数循环诊断起来非常难。

(2) 当播放完毕时，需要通过调用播放器实例的seekToTime:kCMTimeZero方法重新定位播放头光标回到0位置。

(3) 当#2的搜索调用完成时，通知播放栏播放已经完成了，这样就可以重新设置展示时间和搓擦条。

(4) 通过注册NSNotificationCenter来添加itemEndObserver作为通知的监听器，并将定义好的参数传递给它。

(5) 最后重写dealloc方法，当控制器被释放时移除作为监听器的itemEndObserver。

运行应用程序。可以看到在视频播放过程中，随着时间的变动，当前时间和剩余时间标签中的值不断更新，并且可以看到时间搓擦条相应地更新播放头的位置。

下面继续实现其他委托回调方法，使播放栏控件正常工作。

4.5.4 播放栏委托回调

我们先来看一下THTransportDelegate协议提供的简单播放栏回调的实现。代码清单4-10给出了这些方法的实现。

代码清单 4-10 播放栏委托回调

```
- (void)play {
    [self.player play];
}

- (void)pause {
    self.lastPlaybackRate = self.player.rate;
    [self.player pause];
}

- (void)stop {
    [self.player setRate:0.0f];
    [self.transport playbackComplete];
}

- (void)jumpedToTime:(NSTimeInterval)time {
    [self.player seekToTime:CMTimeMakeWithSeconds(time, NSEC_PER_SEC)];

}
```

play的实现不需要过多解释，因为它委托给播放器的同名方法。同样，pause方法委托播放器的pause方法，不过为了条理清晰，仍获取lastPlaybackRate。stop方法调用setRate:并传递参数0，相当于调用了pause，只是采用不同方法实现同一效果。还对播放栏调用了playbackComplete来更新搓擦条的位置。jumpedToTime:方法利用播放器的seekToTime:方法跳转到时间轴上的任意位置。这个方法的使用会在本章后面看到。

接下来，看一下如何实现搓擦条相关的方法。一共有三个方法需要实现，分别对应着当用户与UISlider控件交互时产生的三个事件，如代码清单4-11所示。

代码清单 4-11 Scrubbing 方法

```
- (void)scrubbingDidStart {                                          // 1
    self.lastPlaybackRate = self.player.rate;
    [self.player pause];
    [self.player removeTimeObserver:self.timeObserver];
}

- (void)scrubbedToTime:(NSTimeInterval)time {                        // 2
    [self.playerItem cancelPendingSeeks];
    [self.player seekToTime:CMTimeMakeWithSeconds(time, NSEC_PER_SEC)];
}

- (void)scrubbingDidEnd {                                            // 3
    [self addPlayerItemTimeObserver];
    if (self.lastPlaybackRate > 0.0f) {
        [self.player play];
    }

}
```

(1) 触控事件(UIControlEventTouchDown)会调用scrubbingDidStart方法。在这个方法中开发者将获取当前播放率并暂停播放器。获取当前播放率是为了在搓擦进度结束时恢复播放。此外，

还需要移除当前定期监听器，因为我们不希望在用户直接控制媒体进度时触发此类事件。

(2) 当UISlider实例的值发生变化时(UIControlEventValueChanged)会调用scrubbedToTime方法。由于这个方法在用户移动滑动条位置时会迅速触发，所以首先应该在播放条目上调用cancelPendingSeeks。这是经过性能优化的，如果前一个搜索请求没有完成，则避免出现搜索操作堆积情况的出现。开发者可调用seekToTime:发起一个新的搜索，并将NSTimeInterval值转换为CMTime。

(3) 区域内触控事件(UIControlEventTouchUpInside)会调用scrubbingDidEnd方法，用来表示用户已经完成了搓擦操作。在这个方法中，需要调用addPlayerItemTimeObserver重新添加定期监听器。之后查看lastPlaybackRate值，如果该值大于0，则表示视频已经播放过了，需要重新播放该视频。

通过上述过程，最主要的视频播放功能都已经完成了！运行应用程序，现在可以播放、暂停和调整视频播放进度。完成了这些核心的播放功能，下面就需要对播放中涉及的各功能进行优化，通过添加一些功能提高视频播放的用户体验。

4.6 创建可视搓擦条

你可能已经注意到了播放器右上角有一个带有Show标签的按钮。如果点击这个按钮，会发现在主导航栏下面出现了一个黑色的栏。目前它还没有实际的功能，不过下面看一下能否把这个地方有效利用起来。

可在AV Foundation中找到一个名为AVAssetImageGenerator的工具类。这个类可用来从一个AVAsset视频曲目中提取图片。这样可以生成一个或多个缩略图，用来提升应用程序用户界面的效果。

AVAssetImageGenerator定义了两个方法实现从视频资源中检索图片，分别为：

- copyCGImageAtTime:actualTime:error:允许在指定时间点捕捉图片。如果开发者希望捕捉一张图片那么这个方法是最适合的，可能用于在视频列表中展示视频缩略图。
- generateCGImagesAsynchronouslyForTimes:completionHandler:允许按照第一个参数所指定的时间段生成一个图片序列。该方法具有很高的性能，只需要调用这一个方法就可以生成一组图片。

> **注意：**
>
> AVAssetImageGenerator 既可以生成本地图片，也可以生成持续下载的资源。不过它不能从 HTTP Live Stream 生成图片。

由此实现的一个优秀功能就是创建可视搓擦条。不同于在工具栏底部展示的标准搓擦条，这里创建一个可视化的搓擦条，这样用户可以更简单地在时间轴中指定位置并立即跳转到指定位置。下面看一下如何实现这个功能(如代码清单4-12所示)。

代码清单 4-12 生成图片

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>
```

```
#import "THTransport.h"
#import "THPlayerView.h"
#import "AVAsset+THAdditions.h"
#import "UIAlertView+THAdditions.h"
#import "THThumbnail.h"

...

@interface THPlayerController () <THTransportDelegate>

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) AVPlayerItem *playerItem;
@property (strong, nonatomic) AVPlayer *player;
@property (strong, nonatomic) THPlayerView *playerView;

@property (weak, nonatomic) id <THTransport> transport;

@property (strong, nonatomic) id timeObserver;
@property (strong, nonatomic) id itemEndObserver;
@property (assign, nonatomic) float lastPlaybackRate;

@property (strong, nonatomic) AVAssetImageGenerator *imageGenerator;

@end
```

将导入THThumbnail.h头文件。THThumbnail类是项目中的一个简单模型对象，用来保存我们捕捉到的图片及其相关的时间。还需要添加一个AVAssetImageGenerator类型的新属性。

接下来添加一个新方法generateThumbnails并在status监听器回调方法中调用这个方法，如代码清单4-13所示。

代码清单 4-13 调用 generateThumbnails

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        dispatch_async(dispatch_get_main_queue(), ^{

            [self.playerItem removeObserver:self forKeyPath:STATUS_KEYPATH];

            if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
                // Set up time observers.
                [self addPlayerItemTimeObserver];
                [self addItemEndObserverForPlayerItem];

                CMTime duration = self.playerItem.duration;

                // Synchronize the time display
                [self.transport setCurrentTime:CMTimeGetSeconds(kCMTimeZero)
                                      duration:CMTimeGetSeconds(duration)];

                // Set the video title.
```

```
        [self.transport setTitle:self.asset.title];
106
        [self.player play];
        [self generateThumbnails];
    } else {
        [UIAlertView showAlertWithTitle:@"Error"
                                message:@"Failed to load video"];
    }
        });
    }
}
- (void)generateThumbnails {
}
```

构建基础结构后，下面具体实现该方法。代码清单4-14给出了generateThumbnails方法的实现。

代码清单 4-14 generateThumbnails 的实现

```
- (void)generateThumbnails {

    self.imageGenerator =                                                    // 1
        [AVAssetImageGenerator assetImageGeneratorWithAsset:self.asset];

    // Generate the @2x equivalent
    self.imageGenerator.maximumSize = CGSizeMake(200.0f, 0.0f);              // 2

    CMTime duration = self.asset.duration;

    NSMutableArray *times = [NSMutableArray array];                          // 3
    CMTimeValue increment = duration.value / 20;
    CMTimeValue currentValue = kCMTimeZero;
    while (currentValue <= duration.value) {
        CMTime time = CMTimeMake(currentValue, duration.timescale);
        [times addObject:[NSValue valueWithCMTime:time]];
        currentValue += increment;
    }

    __block NSUInteger imageCount = times.count;                             // 4
    __block NSMutableArray *images = [NSMutableArray array];

    AVAssetImageGeneratorCompletionHandler handler;                          // 5

    handler = ^(CMTime requestedTime,
                CGImageRef imageRef,
                CMTime actualTime,
                AVAssetImageGeneratorResult result,
                NSError *error) {

        if (result == AVAssetImageGeneratorSucceeded) {                      // 6
```

```
            UIImage *image = [UIImage imageWithCGImage:imageRef];
            id thumbnail =
                [THThumbnail thumbnailWithImage:image time:actualTime];
            [images addObject:thumbnail];
        } else {
            NSLog(@"Failed to create thumnail image.");
        }

        // If the decremented image count is at 0, we're all done.
        if (--imageCount == 0) {                                        // 7
            dispatch_async(dispatch_get_main_queue(), ^{
                NSString *name = THThumbnailsGeneratedNotification;
                NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
                [nc postNotificationName:name object:images];
            });
        }
    };

    [self.imageGenerator generateCGImagesAsynchronouslyForTimes:times     // 8
                                          completionHandler:handler];

}
```

(1) 首先创建一个新的AVAssetImageGenerator实例，为其传递一个对控制器asset属性的引用。保持对该对象的强引用非常关键。如果没有注意到这一点将遇到麻烦，因为会导致无法调用回调。

(2) AVAssetImageGenerator为配置图片生成定义了一些属性。虽然为大部分属性提供了合理的默认值，不过有一个属性在每次使用时都需要明确配置，就是maximumSize属性。默认情况下，捕捉的图片都保持原始维度。如果处理720p或1080p视频的话，则创建的图片会非常大。设置maximumSize属性会自动对图片的尺寸进行缩放并显著提高性能。指定一个width值为200、height值为0的CGSize。这样可以确保生成的图片都遵循一定宽度，并且会根据视频的宽高比自动设置高度值。

(3) 下面需要做的是执行一些计算来生成CMTime值的集合，这些值用来指定视频中的捕捉位置。代码中将视频时间轴平均分成20个CMTime值。循环遍历视频的duration，使用CMTimeMake函数创建了一个新的时间，之后将结果CMTime封装成一个NSValue保存在times数组中。

(4) 基于times数组中元素的个数，定义一个名为imageCount的_block变量。这用于确定所有图片处理完成的时间。还定义一个__block变量，类型为NSMutableArray，名为images。用于保存生成图片的集合。__block修饰词用来确保回调block操作直接发生在这些指针上而非副本上。

(5) 接下来定义了一个AVAssetImageGeneratorCompletionHandler类型的回调块。这是其中一个较长的代码块定义，下面看一下它的参数：

- **requestedTime**：请求的最初时间。它对应于生成图像的调用中指定的 times 数组中的值。
- **imageRef**：生成的 CGImageRef，如果在给定的时间点没有生成图片则赋值 NULL。

- **actualTime**：图片实际生成的时间。基于实际效率，这个值可能与请求时间不同。可以在生成图片前通过在 AVAssetImageGenerator 实例设置 requestedTimeToleranceBefore 和 requestedTimeToleranceAfter 值来调整 requestedTime 和 actualTime 的接近程度。
- **result**：AVAssetImageGeneratorResult 用来表示图片是成功生成、失败还是取消。
- **error**：一个 NSError 指针，如果收到 AVAssetImageGeneratorFailed 的 AVAssetImage-GeneratorResult，可以通过这个 NSError 指针诊断问题。

(6) 如果result值为AVAssetImageGeneratorSucceeded，则表示图片已经成功生成了，基于返回的CGImageRef创建一个新的UIImage。接下来创建一个新的THThumbnail实例将图片和时间信息打包，并添加到数组中。

(7) 在回调块的每次调用中，使imageCount属性减1并判断其是否等于0，如果等于0则表明所有图片都处理完成了。之后发送一个新的名为THThumbnails- GeneratedNotification的应用程序专用通知消息，将图片集合作为object参数传递。视图层会接收该通知并用它生成可视化搓擦条。

再次运行该应用程序。现在当我们点击Show按钮时会看到黑色的条被一串缩略图所替代，缩略图对应于视频文件中的不同时间点。点击一张图片会调用我们之前实现的委托的jumpedToTime:方法。注意AVAssetImageGenerator可为本地资源和远程资源生成图片，不过可以预料的是，当为远程资源生成图片会消耗比较长的时间。这种情况下可以使用效率更好的方法以提升用户体验，比如为每个返回的图片创建可视化布局，或将其写入图片缓存并让视图定期轮询缓存。

> **注意：**
>
> 大部分播放用例都可以在 iOS 模拟器中进行测试。不过在实际设备上测试时性能表现更好。

4.7 显示字幕

使应用程序被尽可能多的用户接受是一件非常重要的事，这就意味着我们需要让用户可以使用本国母语访问我们的应用程序，同时还要考虑存在听觉障碍或有其他辅助功能需求的用户。视频播放器在这一点上提高用户体验常用的方法就是随时提供字幕。AV Foundation在展示字幕或隐藏式字幕方面提供了可靠方法。AVPlayerLayer会自动渲染这些元素，并且可以让开发者告诉应用程序哪些元素需要渲染。完成这些操作要用到AVMediaSelectionGroup和AVMediaSelectionOption两个类。

AVMediaSelectionOption表示AVAsset中的备用媒体呈现方式。一个资源可能包含备用媒体呈现方式，比如备用音频、视频或文本轨道。这些轨道可能是指定语言的音频轨道、备用相机角度或此刻我们所感兴趣的指定语言的字幕。确定存在哪些备用轨道要用到一个名为availableMediaCharacteristicsWithMediaSelectionOptions的AVAsset属性(我之前就说过AV Foundation团队喜欢这种长名字)。这个属性会返回一个包含字符串的数组，这些字符串用于表示保存在资源中可用选项的媒体特征。具体来说，返回数组所包含的字符串值为AVMediaCharacteristicVisual(视频)、AVMediaCharacteristicAudible(音频)、AVMediaCharacteristicLegible (字幕或隐藏式字幕)。

请求可用媒体特性数据后，调用AVAsset的mediaSelectionGroupForMediaCharacteristic:方法，为其传递要检索的选项的特定媒体特性。这个方法会返回一个AVMediaSelectionGroup，它作为一个或多个互斥的AVMediaSelectionOption实例的容器。下面看一个简单示例：

```
NSArray *mediaCharacteristics =
    self.asset.availableMediaCharacteristicsWithMediaSelectionOptions;
for (NSString *characteristic in mediaCharacteristics) {
    AVMediaSelectionGroup *group =
        [self.asset mediaSelectionGroupForMediaCharacteristic:characteristic];
    NSLog(@"[%@]", characteristic);
    for (AVMediaSelectionOption *option in group.options) {
        NSLog(@"Option: %@", option.displayName);
    }
}
```

为包含一个或多个字幕的资源运行这段代码所生成的输出内容如下所示：

```
[AVMediaCharacteristicLegible]
Option: English
Option: Italian
Option: Portuguese
Option: Russian
[AVMediaCharacteristicAudible]
Option: English
```

在示例中可以看到多个字幕轨道以及一个English音频轨道。

当我们载入正确的AVMediaSelectionGroup并定义好需要的AVMediaSelectionOption之后，下一步就是付诸实际行动了。通过在激活的AVPlayerItem上调用selectMediaOption:inMediaSelectionGroup:来实现这一功能。比如，如果需要显示俄文字幕，如下编码：

```
AVMediaSelectionGroup *group =
    [self.asset mediaSelectionGroupForMediaCharacteristic:characteristic];
NSLocale *russianLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"ru_RU"];
NSArray *options =
    [AVMediaSelectionGroup mediaSelectionOptionsFromArray:group.options
                                                withLocale:russianLocale];
AVMediaSelectionOption *option = [options firstObject];
[self.playerItem selectMediaOption:option inMediaSelectionGroup:group];
```

下面在Video Player应用程序中具体实施，在THPlayerController类中添加几个新的方法。首先如代码清单4-15所示进行一些修改。

代码清单 4-15　loadMediaOptions 设置

```
- (void)prepareToPlay {
    NSArray *keys = @[
        @"tracks",
        @"duration",
        @"commonMetadata",
        @"availableMediaCharacteristicsWithMediaSelectionOptions"
```

```
    ];
    ...
}
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {
    if (context == &PlayerItemStatusContext) {
        dispatch_async(dispatch_get_main_queue(), ^{
            if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
                ...
                [self loadMediaOptions];
            } else {
                [UIAlertView showAlertWithTitle:@"Error"
                                        message:@"Failed to load video"];
            }
        });
    }
}
- (void)loadMediaOptions {
}
```

在prepareToPlay方法中，我们希望在它的自动载入属性列表中加入availableMediaCharacteristicsWithMediaSelectionOptions属性。在调用任何媒体选择API前载入该属性很有必要，这样会避免主线程拥堵。当播放器条目准备播放就绪时调用loadMediaOptions方法。

loadMediaOptions方法的实现如代码清单4-16所示。

代码清单 4-16 loadMediaOptions 实现

```
- (void)loadMediaOptions {
    NSString *mc = AVMediaCharacteristicLegible;                          // 1
    AVMediaSelectionGroup *group =
        [self.asset mediaSelectionGroupForMediaCharacteristic:mc];        // 2
    if (group) {
        NSMutableArray *subtitles = [NSMutableArray array];               // 3
        for (AVMediaSelectionOption *option in group.options) {
            [subtitles addObject:option.displayName];
        }
        [self.transport setSubtitles:subtitles];                          // 4
    } else {
        [self.transport setSubtitles:nil];
    }
}
```

(1) 我们只对查找资源中的字幕选项感兴趣，所以只定义了一个媒体特性字符串，并令它的值为AVMediaCharacteristicLegible。

(2) 请求与已定义媒体特性对应的AVMediaSelectionGroup。

(3) 假设找到一组数据(应用程序本地资源包含字幕、远程资源不包含)，创建一个包含要传递给视图层的用户可呈现字符串的数组，做法是请求每个选项的displayName属性。

(4) 最后，设置播放栏上的字幕字符串集合，使其可在字幕选择界面中呈现。在else条件中，传递nil，表示没有可呈现的界面。

当用户选择一个字幕时，需要一个方法来处理该选择，并在当前播放器条目上激活相应的AVMediaSelectionOption。代码清单4-17给出了这个方法的实现。

代码清单 4-17 处理字幕选择

```
- (void)subtitleSelected:(NSString *)subtitle {
    NSString *mc = AVMediaCharacteristicLegible;
    AVMediaSelectionGroup *group =
        [self.asset mediaSelectionGroupForMediaCharacteristic:mc];          // 1
    BOOL selected = NO;
    for (AVMediaSelectionOption *option in group.options) {
        if ([option.displayName isEqualToString:subtitle]) {
            [self.playerItem selectMediaOption:option                       // 2
                        inMediaSelectionGroup:group];
            selected = YES;
        }
    }
    if (!selected) {
        [self.playerItem selectMediaOption:nil                              // 3
                    inMediaSelectionGroup:group];
    }

}
```

(1) 为资源中包含的有效选项检索AVMediaSelectionGroup。

(2) 循环遍历所有组中的选项，并找到与传递给subtitleSelected:方法的字幕字符串匹配的AVMediaSelectionOption。找到正确选项后，在播放器条目上调用selectMediaOption:inMediaSelectionGroup:方法激活它。这样选中的字幕就会立即出现在AVPlayerLayer上。

(3) 如果用户在字幕选项列表中选择None，则为选中媒体选项设置nil，以便移除展示中的字幕。

最后需要做的一件事是打开VideoPlayer-Prefix.pch文件并将ENABLE_SUBTITLES的定义由0改为1。如果当前媒体中有可用字幕，则播放视图会展示合适的字幕选择界面。

再次运行应用程序，在播放栏右下角会看到一个新的按钮。选中按钮查看可用的字幕，选择一个选项，瞧！神奇的一幕出现了。

4.8 Airplay

最后一个需要讨论的优化问题是在Video Player应用程序中整合AirPlay功能。AirPlay是苹果公司推出的一项技术，旨在用无线方式将流媒体音频和视频内容在Apple TV上播放，或者将纯音频内容在多种第三方音频系统中播放。如果用户拥有Apple TV或其他音频系统中的一个，就会知道这个功能简直太神奇了。好消息是将这个功能整合到应用程序非常容易实现。

AVPlayer有一个属性allowsExternalPlayback，允许启用或禁用AirPlay播放功能。该属性的默认值为YES，即在不做任何额外编码的情况下，播放器应用程序也会自动支持AirPlay功能。虽然通常AirPlay功能是需要的，不过如果由于某些强制的原因要禁用该功能，可以通过设置allowsExternalPlayback属性为NO来实现。

线路选择功能

iOS为选择AirPlay线路提供了一个整体的界面。具体的用户界面和手势动作取决于iOS的版本。在iOS 6及早期版本中，用户双击Home键启动dock，向右滑动找到选择AirPlay线路的界面，如图4-6所示。

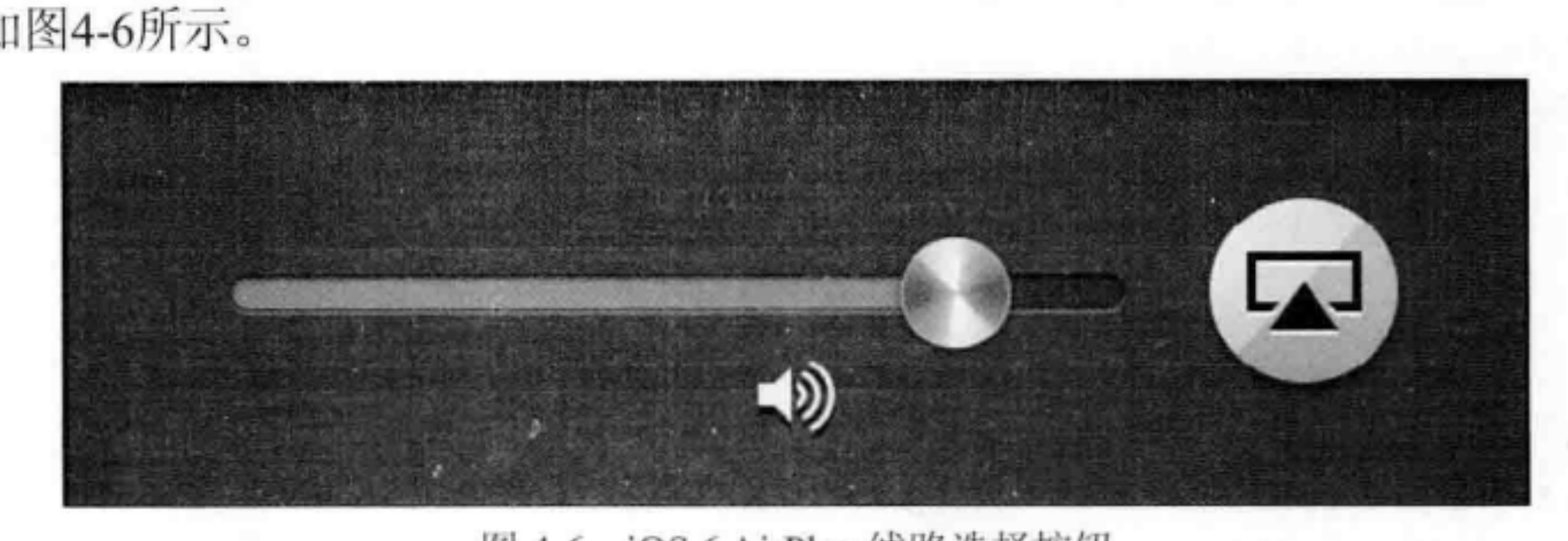

图 4-6 iOS 6 AirPlay 线路选择按钮

iOS 7及之后的版本提供了一种更方便的方法访问该界面。在屏幕底端向上滑动打开控制中心(Control Center)，选择AirPlay按钮即可，如图4-7所示。

图 4-7 iOS 7 AirPlay 线路按钮

虽然使用整体线路选择方法可以帮助用户实现期望的功能，不过其在用户体验方面做的还不够理想。尤其是iOS 6版本中需要用户跳出应用程序，会中断应用程序的工作流。另一个重要的需要注意的是很多用户并不知道这个整体界面，很可能完全忽略这个强大实用的功能。所以开发者应该以比较明显的方式在应用程序内部提供AirPlay线路选择界面。有趣的是，iOS并没有AirPlay框架或API供开发者使用，取而代之的是我们使用Media Player框架中的MPVolumeView类来实现这个功能。

使用这个组件时，需要关联和导入MediaPlayer框架(<MediaPlayer/MediaPlayer.h>)并创建一个MPVolumeView实例，如下面代码所示：

```
CGRect rect = // desired frame
MPVolumeView *volumeView = [[MPVolumeView alloc] initWithFrame:rect];
[self.view addSubview:volumeView];
```

默认的MPVolumeView实例提供两个用户界面元素。顾名思义，其中一个元素是控制系统音量的滑动条。它所提供的功能等同于iOS设备侧面的硬音量调节按钮(硬件)。如果用户网络中存在可用的AirPlay设备，则会额外显示一个AirPlay路线选择按钮。点击按钮会显示所有可用AirPlay线路的列表。

如果只需要展示线路选择按钮，可以对应用程序做如下修改。

```
MPVolumeView  *volumeView = [[MPVolumeView alloc] init];
volumeView.showsVolumeSlider = NO;
[volumeView sizeToFit];
[transportView addSubview:volumeView];
```

有一点需要明确，只有当用户具有可用AirPlay目标而且WiFi网络启用时才会显示线路选择按钮。这两个条件只要有一个不满足，MPVolumeView就会自动隐藏按钮。

> **注意：**
>
> MPVolumeView 只有在 iOS 设备上才可以显示，iOS 模拟器是不可以显示的。

我们不准备对实现过程进行详细讲解，因为与同前面的示例一样简单。如果可能的话应用程序已经创建好了线路选择按钮，不过我们还是需要打开VideoPlayer-Prefix.pch，将ENABLE_AIRPLAY定义由0修改为1。如果你有一个Apple TV或一个支持AirPlay的音频系统，当运行应用程序时就会看到AirPlay线路选择按钮了。

要了解更多关于AirPlay及其用法的高级技术，可到Apple Developer Center中查找AirPlay Overview文档(https:developer.apple.com/Library/ios/documentation/AudioVideo/Conceptual/AirPlay Guide/Introduction/Introduction.html)。

4.9 小结

本章深入探讨了AV Foundation的视频播放功能。现在我们知道了如何通过AVPlayer播放AVPlayerItem实例，并直接将视频输出为AVPlayerLayer实例。我们还第一次接触了AVAssetImageGenerator，用它来创建播放器的可视化搓擦条。开发者会发现这是在AV

Foundation中的不同场景都有用的类。最后我们通过整合AirPlay功能提高视频播放的体验，并使用AVMediaSelectionGroup和AVMediaSelectionOption来展示字幕。本章构建的示例应用程序对开发者编写任何视频播放解决方案都是一个好起点。

4.10 挑战

本章提出两各方面的挑战：

(1) 我们已经了解了AVPlayer、AVPlayerItem和AVPlayerLayer的一些重要方面，不过还有一些其他有趣的功能没有介绍。可以尝试一些额外的搜索操作，研究更多的属性用法及获取更多的日志信息。示例应用程序可以提供一个非常好的测试环境对这些额外的功能进行实验。

(2) 多花一些时间让自己熟悉CMTime。它在AV Foundation所提供的多种功能中扮演着重要角色，所以开发者一定要精通它的使用。打开CMTime.h头文件查看它所提供的功能。头文件还是一个最好的文档资源用于了解各种结构、宏和功能。建议大家创建一个关联Core Media框架的简单命令行应用程序，并花时间进行实验。学习有关CMTime值的创建、比较、相加、相减等操作，这么做绝对物有所值。

第5章

AV Kit 用法

第4章中介绍了使用AVPlayer和AVPlayerItem创建一个自定义视频播放器的方法。创建一个自定义视频播放器在很多情况下是需要的，因为这样才可以对所有播放器行为和用户界面进行控制。不过我们是否希望既能利用AV Foundation强大的功能，又可以提供给用户熟悉的界面和优秀的体验呢，就如同使用iOS上的Video应用程序或Mac OS X上的QuickTime播放器一样的感觉。要取得同样的效果和体验还需要进行大量的工作，并且还需要开发者为iOS或Mac OS X的不同版本编写多个用户界面。电视广告会这样说：我们一定会找到更好的办法！幸运的是得益于AV Kit框架，还真有更好的办法。

AV Kit可以简化基于AV Foundation框架且满足默认操作系统视觉效果和体验的视频播放器的创建过程。AV Kit框架第一次出现是在Mac OS X Mavericks版本中，并从iOS 8开始被引入到iOS平台。本章我们集中讨论AV Kit在OS X上的应用，因为它比iOS版本的功能更多，不过下面先简单看一下AV Kit的iOS版本都能做些什么。

5.1 针对 iOS 平台的 AV Kit 框架

iOS Media Player 框架最初就定义了 MPMoviePlayerController 和 MPMoviePlayerViewController两个类，它们提供一种简单的方法将完整的视频播放功能整合到应用程序中。MPMoviePlayerController定义了一些标准的播放控件，这些控件可以以子视图或全屏的方式内置于应用程序中，并支持通过AirPlay连接的音频流和视频流，同时还包含很多其他实用的功能。已经有这么多的功能，为什么我们还需要其他功能？MPMoviePlayerController的一个关键问题是它是一个极度的黑盒组件，它基于AV Foundation之上，但是不幸的是它将所有的基础功能都隐藏了。这就导致开发者无法使用AVPlayer和AVPlayerItem提供的一些更高级功能。从iOS 8版本开始，AV Kit框架的引入带来了更多强大的功能。

针对iOS平台的AV Kit是一个简单的标准框架——只包含一个AVPlayerViewController类。它是UIViewController的子类，用于展示并控制AVPlayer实例的播放。AVPlayerViewController具有一个很小的界面，提供以下几个属性：

- **player**：用来播放媒体内容的 AVPlayer 实例。

- **showsPlaybackControls**：一个用来表示播放控件是否显示或隐藏的布尔类型的值。
- **videoGravity**：对内部 AVPlayerLayer 实例的 video gravity 进行设置的一个 NSString。如果需要对 AVPlayerLayer 的 video gravity 概念进行复习，可参阅第 4 章。
- **readyForDisplay**：通过观察这个布尔类型的值来确定视频内容是否已经准备好进行展示。

虽然界面非常朴实，不过这个类可以提供许多实用值。要证明这一点，请查看下面给出的示例。

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    NSURL *url =
        [[NSBundle mainBundle] URLForResource:@"video" withExtension:@"m4v"];

    AVPlayerViewController *controller = [[AVPlayerViewController alloc] init];
    controller.player = [AVPlayer playerWithURL:url];

    self.window.rootViewController = controller;

    return YES;
}

@end
```

该示例中创建了一个新的AVPlayerViewController，为它设置了一个AVPlayer实例，并将其控制器设置为窗口的rootViewController。虽然我们只写了短短几行代码，不过运行这些代码就会显示一个播放器，如图5-1所示。

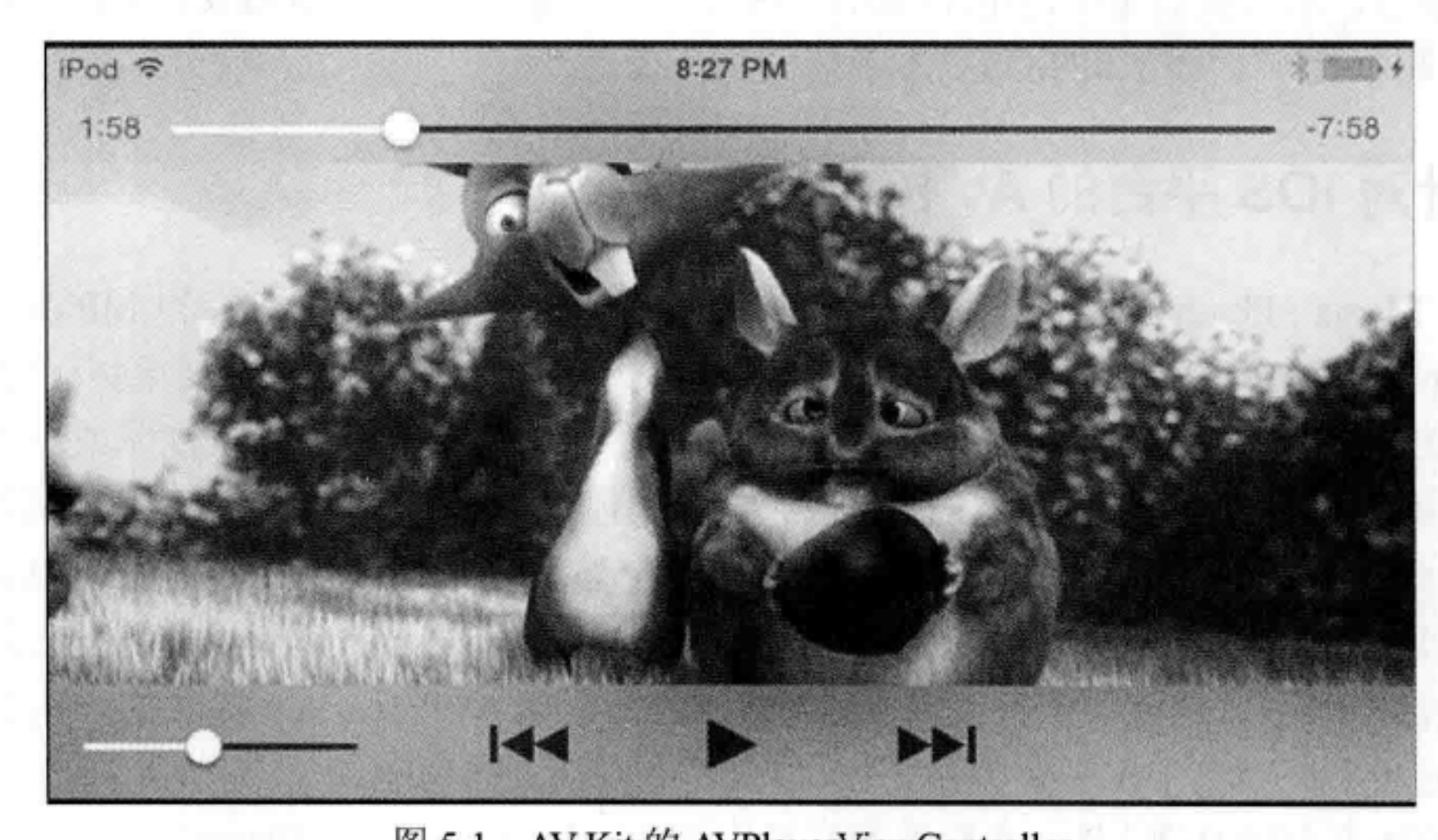

图 5-1 AV Kit 的 AVPlayerViewController

如图5-1所示，AVPlayerViewController在没有编写过多代码的前提下实现了许多功能。创建了一个带有与iOS视频播放器同样用户界面和体验的功能齐全的播放器。由于这个类是

UIViewController子类，所以很容易将它作为子视图控制器嵌入到其他视图中，或向其他视图控制器一样呈现。

> **注意：**
>
> 示例中使用 playerWithURL:便捷初始化方法创建了一个 AVPlayer。当然这的确是一个比较简单的情况，不过在开发者需要更多控件时，可以像上一章中介绍的那样设置完整的播放栈。

与MPMoviePlayerController不同，AVPlayerViewController没有定义controlsStyle属性用于表示所使用的播放控件。取而代之的是，它提供了一种动态播放控件，可以动态更新，为播放的内容呈现所需的用户界面。这意味着播放器总是带给用户最好的体验，无论是通过章节标签播放本地视频还是播放带有字幕的流媒体视频。图5-2给出了实际应用中的几种备用控件。

图 5-2　AVPlayerViewController：动态播放控件

AVPlayerViewController是iOS 8给出的一个简单但功能强大的新类。它提供了一个非常强大的替代方法，即使用MPMoviePlayerController，因为其具有全部AV Foundation栈的特性，支持开发者使用更高级的功能。要了解AVPlayerViewController类的其他信息，可以参考WWDC 2014的Session 503: Mastering Modern Media Playback。

5.2　针对 Mac OS X 平台的 AV Kit 框架

Mac平台的AV Kit框架首先在Mac OS X Mavericks版本中引入。该框架定义了一个名为AVPlayerView的类，通过这个类可以简单地将完整的视频播放功能整合到Mac应用程序中。它所定义的界面和使用体验同QuickTimePlayer X一样，能够提供Mac用户熟悉的功能和用户界面效果。

AVPlayerView是一个NSView子类，用于展示和控制AVPlayer实例的播放。在使用AVPlayerView对象的过程中上一章我们所学到的知识同样适用，不过开发一款带有标准化播放控件和功能的视频播放应用程序会变得更加快速和简单。它还自动支持所有标准的最新OS X功能，比如本地化、状态恢复、全屏播放、高分辨率展示和辅助功能等。

5.3　迈出第一步

本节通过创建一个基于AV Kit框架的视频播放器来学习如何使用AVPlayerView。在Chapter 5目录中可以找到名为KitTimePlayer_starter的示例项目。虽然本书的大部分示例应用

程序都将AV Foundation有关的代码分解出来到放到自己的类集合中，不过由于我们现在讨论的是针对Mac系统的开发，所以我们需要在主NSDocument实例中开发这个应用程序。现在打开项目开始编码吧！

首先需要配置的是对KitTime Player的目标Document Types进行设置。选中Project Navigator中的项目根节点使其处于高亮状态，并选择KitTime Player目标。选择Info选项卡并展开Document Types部分，如图5-3所示进行修改。

图 5-3 配置 KitTime Player 的 Document Types

- 将 mydoc 扩展从 Extensions 字段中移除，因为应用程序将不会创建自定义文档类型。
- 设置 Identifier 字段为 public.audiovisual-content。选择 Uniform Type Identifier 将允许应用程序打开所有视听媒体。
- 将 Role 设置为 Viewer，这样当启动应用程序时不会创建一个新的文档窗口。

接下来，在Project Navigator中选择THDocument.xib。NSWindow已经被设置为640×360来适配即将播放的视频内容的宽高比，不过开发者可以根据需求自定定义该尺寸。在Object Library的搜索框中，输入AVPlayerView直到看见AV Player View组件。拖曳一个它的实例到窗口中并将它置于窗口区域的中间，如图5-4所示。

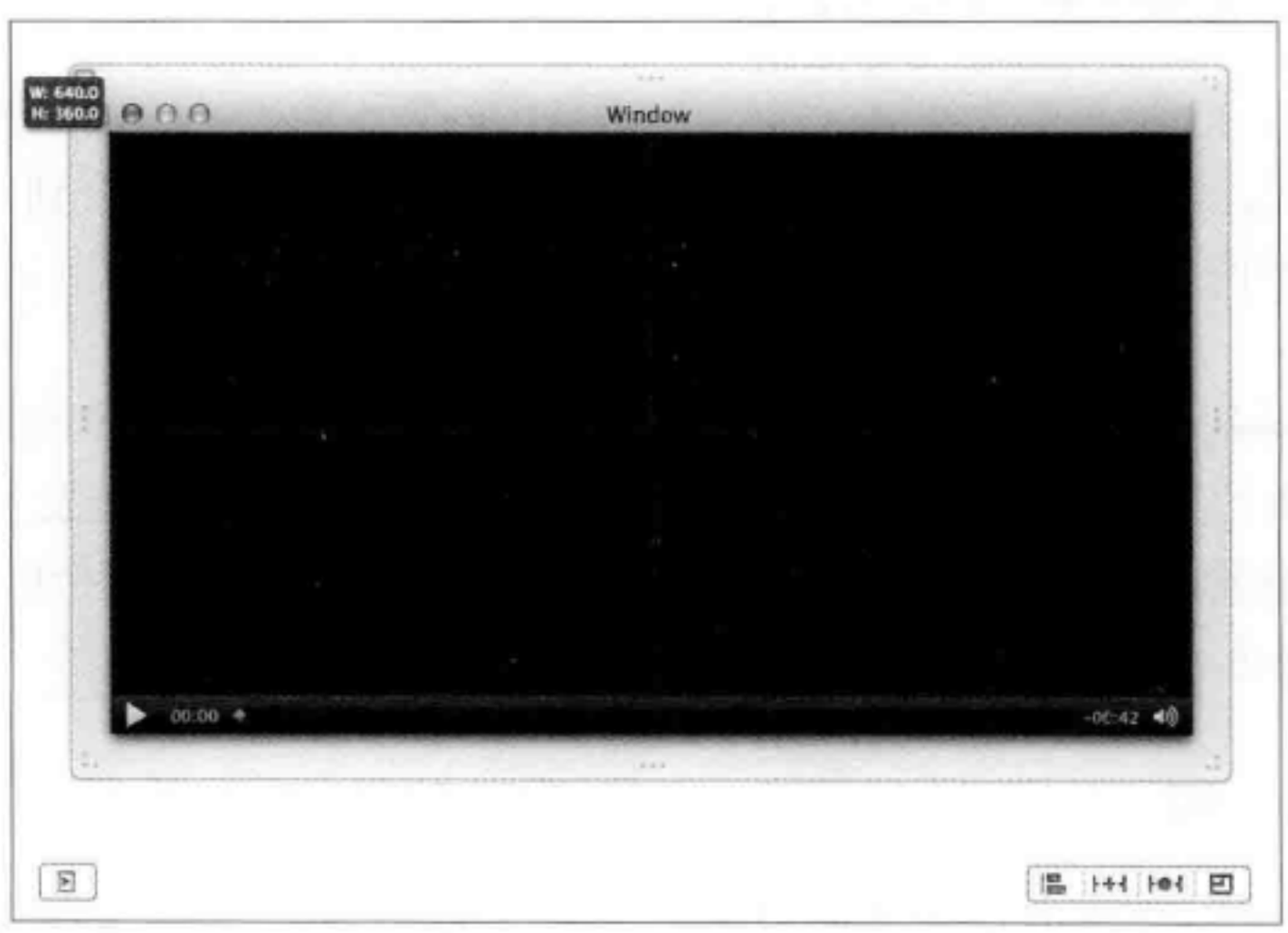

图 5-4 向窗口添加一个 AVPlayerView

当用户改变窗口大小时需要确保AVPlayerView也进行相应的变化，开发者需要为其添加一个合适的Auto Layout约束。这种情况下最简单的方法就是选择Resolve Auto Layout Issues按钮并选择Add Missing Constraints inWindow，如图5-5所示。

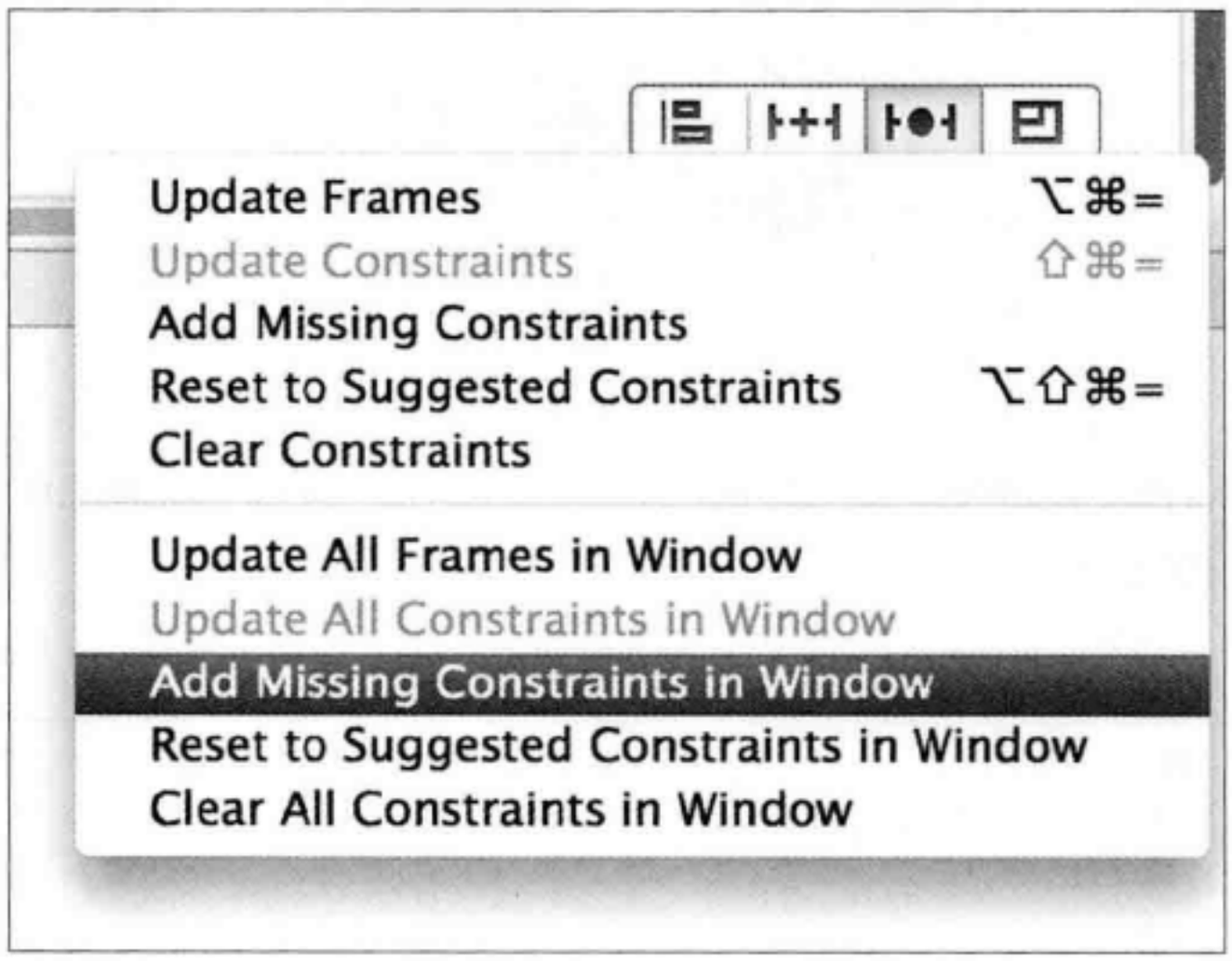

图 5-5　设置 Auto Layout 约束

找到播放视图的属性查看器，设置Controls Style属性为Floating，如图5-6所示。

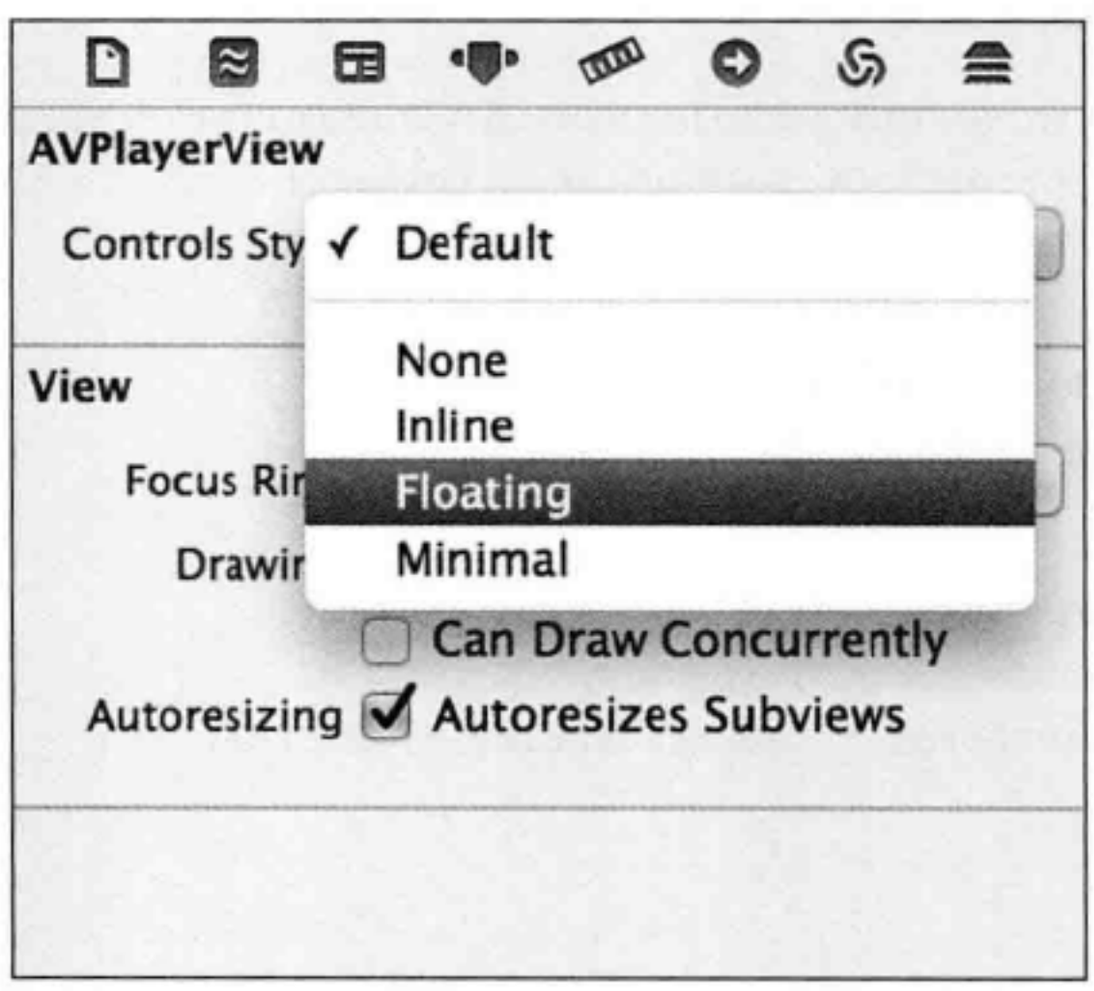

图 5-6　设置 Controls Style 属性

上述方法所创建的用户界面与QuickTime Player界面具有同样的视觉效果。

THDocument实例已经为播放器视图定义了一个IBOutlet，不过我们还是需要将播放器关联到IBOutlet。按住Control从File's Owner代理拖曳到AVPlayerView实例，并选择playerView Outlet，如图5-7所示。

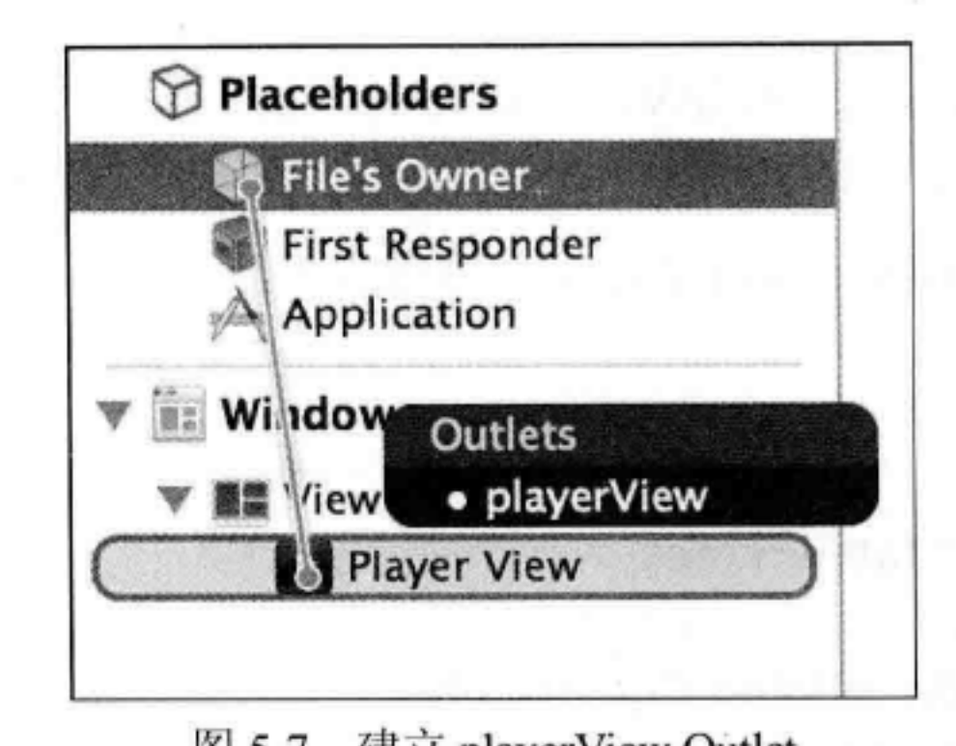

图 5-7 建立 playerView Outlet

目前所需的Interface Builder已经配置完成了，下面介绍THDocument.m文件。代码清单5-1给出了这个类的最初实现。

代码清单 5-1 Document 实现

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>

@interface THDocument ()
@property (weak) IBOutlet AVPlayerView *playerView;
@end

@implementation THDocument

- (void)windowControllerDidLoadNib:(NSWindowController *)aController {
    [super windowControllerDidLoadNib:aController];

}

- (NSString *)windowNibName {
    return @"THDocument";
}

- (BOOL)readFromURL:(NSURL *)url
            ofType:(NSString *)typeName
             error:(NSError *__autoreleasing *)outError {
    return YES;
}

@end
```

这只是NSDocument子类的梗概，不过它具有创建视频播放器应用程序所需的全部核心元素。唯一欠缺的就是AV Foundation代码。将下面两行代码添加到windowControllerdLoadNib:方法的结尾处。

```
self.playerView.player = [AVPlayer playerWithURL:[self fileURL]];
self.playerView.showsSharingServiceButton = YES;
```

将这些代码添加好就可以运行应用程序了。当应用程序启动后，选择File菜单，点击Open，

然后选择Chapter 5目录下的hubblecast.m4v文件并点击应用程序的Play按钮。我们只花了几分钟时间和短短几行代码就实现了一个播放器应用程序，如图5-8所示。

图 5-8　KitTime 播放器

AVPlayerView会自动提供标准的播放控件，一个视频搓擦条、一个音量控制器、动态的章节和字幕菜单，以及一个允许与Mac OS X提供的标准目标进行分享的分享服务菜单。显然，AV Kit对于Mac平台是一个非常好的补充。虽然在之后我们会学到更多方法，不过这已经是一个令人激动的开端了。现在把视线转移到AVPlayerView提供的各种控件类型上。

5.4　控件类型

AVPlayerView定义了大量的控件类型供开发者选用。可在Interface Builder中直接修改这些属性，我们之前就是这样做的，或以编程方式修改controlsStyle属性进行编辑。下面介绍该类定义的各个类型。

5.4.1　内嵌类型

内嵌类型(AVPlayerViewControlsStyleInline)是AVPlayerView使用的默认类型，如图5-9所示。这个类型与QTKit Framework的QTMovieView界面非常相似，并定义了标准播放栏、搓擦条和音量控制器。当媒体具有动态章节和字幕数据时，它还可以支持动态章节和字幕菜单的展示。

图 5-9　带有内嵌控件的 AVPlayerView

5.4.2 浮动类型

浮动类型(AVPlayerViewControlsStyleFloating)的效果与当前版本的QuickTime Player效果(如图5-10所示)一致。实际上QuickTime Player的Mavericks版本就是使用的AVPlayerView，所以它所使用的界面对于大部分Mac用户而言很快就会熟悉。与内嵌类型一样，浮动类型也定义了标准的播放栏、搓擦条和音量控件，如果视频包含章节分段和字幕，同样会自动显示章节信息和字幕菜单。

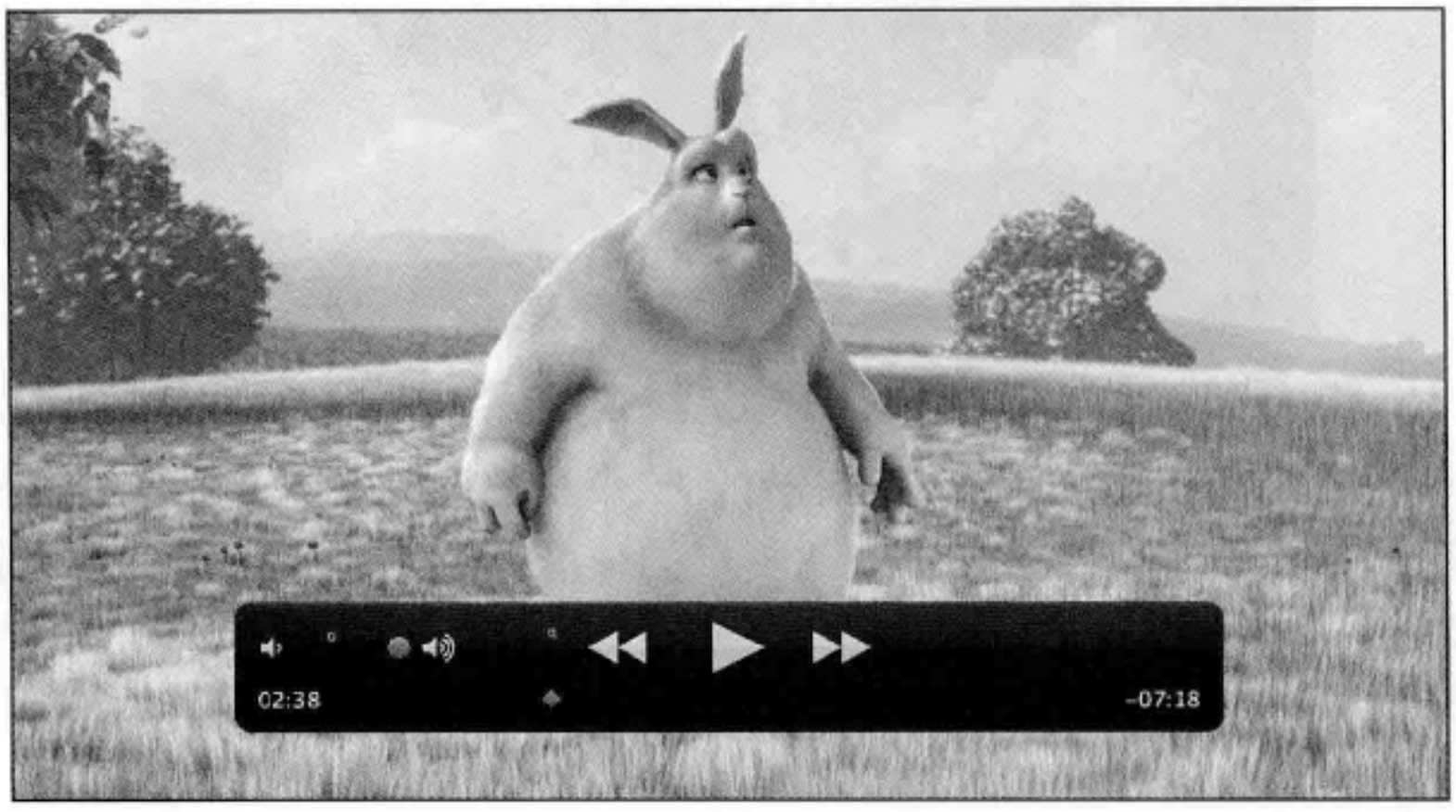

图 5-10 带有浮动类型控件的 AVPlayerView

5.4.3 最小化类型

最小化类型(AVPlayerViewControlsStyleMinimal)在屏幕中间定义一个圆形浮动按钮，该按钮用来展示带有圆形进度指示图标的播放或暂停按钮，如图5-11所示。当播放需要最小化控件的短视频时，可以选择该方案。

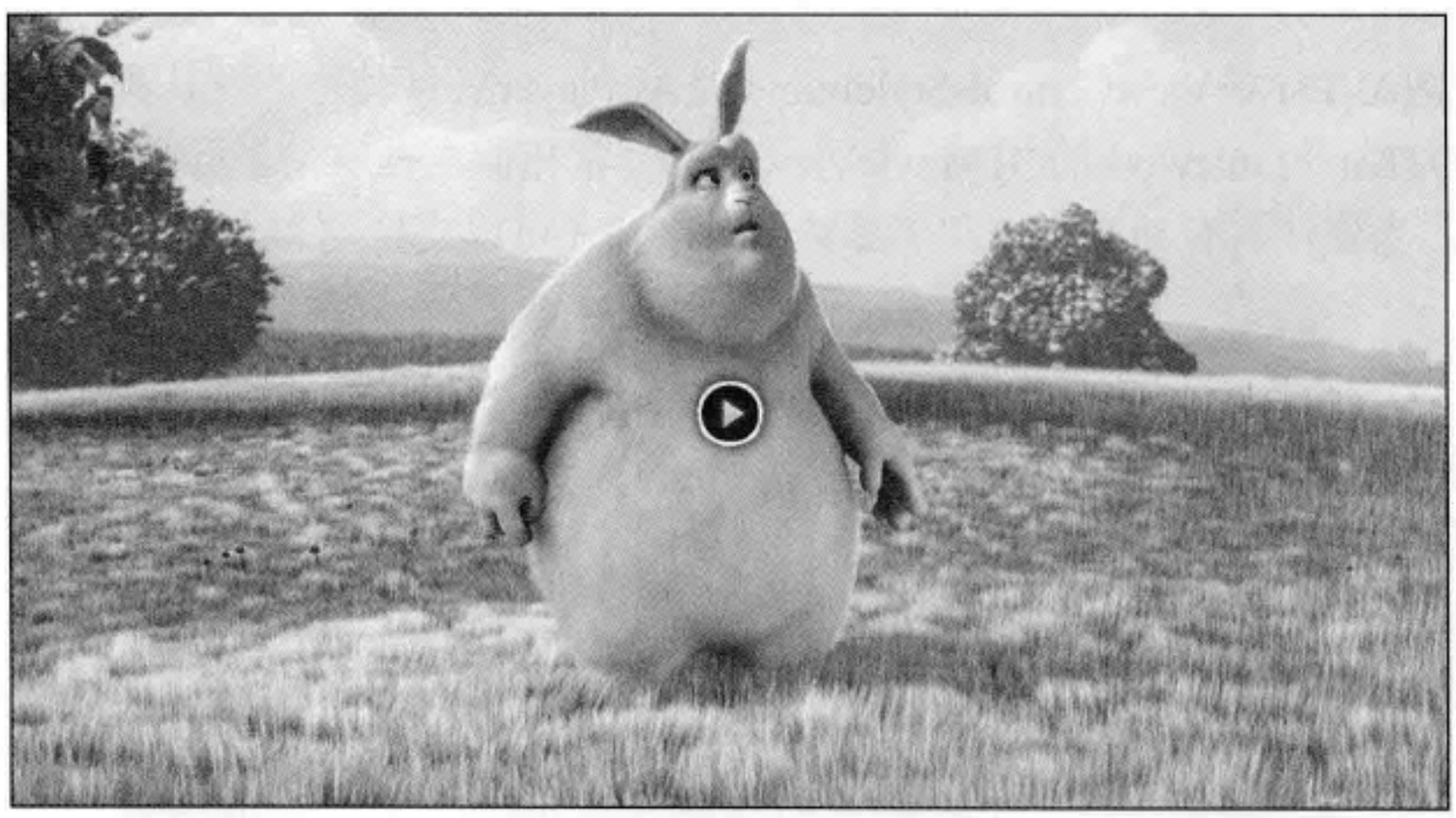

图 5-11 带有最小化类型控件的 AVPlayerView

5.4.4　None 类型

None(AVPlayerViewControlsStyleNone)类型实际上就是没有类型。选择这一类型则不会出现播放控件，只简单展示视频的内容，如图5-12所示。如果开发者希望使用自定义的播放控件或在一些没有特定控件需求的场景下，使用None类型很有用。

图 5-12　不带控件的 AVPlayerView

不管选择了哪个类型，AVPlayerView都会对标准键盘集指令进行反馈。空格键可以对视频进行播放和暂停。左右箭头可以逐帧调整视频。此外，播放器视图支持J-K-L导航，即J键对应回放，L键对应快进和K键对应停止播放。

> **注意：**
>
> 如果正在播放一个 HTTP Live Streaming 视频，AVPlayerView 会根据控件类型自动切换为备用控件，以满足流媒体的播放要求。

> **注意：**
>
> 只有浮动类型和内嵌类型可以通过设置 showsSharingServiceButton 属性为 YES，来支持分享服务的菜单。

5.5　拓展学习

我们现在已经学习了如何通过AV Kit快速创建并运行应用程序，不过我希望进一步深入了解AV Kit框架，并在应用程序中加入一些更高级的功能。要实现这些功能，需要从媒体栈开始进行一些修改。AVPlayer包含了一个playerWithURL:方法，可以快速创建基础AVAsset和AVPlayerItem实例并为这些对象执行相关的准备工作。使用这个方法虽然很简单，不过通常当开发者需要直接使用这些对象时，会发现还是明确创建和准备这些对象用起来更加简单。开发者将会用到与上一章设置AV Foundation播放栈类似的基本技巧，如代码清单5-2所示。

代码清单 5-2 设置播放栈

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>
#define STATUS_KEY @"status"

@interface THDocument ()

@property (strong) AVAsset *asset;                                    // 1
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;

@property (weak) IBOutlet AVPlayerView *playerView;

@end

@implementation THDocument

- (void)windowControllerDidLoadNib:(NSWindowController *)controller {
    [super windowControllerDidLoadNib:controller];

    [self setupPlaybackStackWithURL:[self fileURL]];                  // 2
}

- (void)setupPlaybackStackWithURL:(NSURL *)url {

    self.asset = [AVAsset assetWithURL:url];

    NSArray *keys = @[@"commonMetadata", @"availableChapterLocales"];  // 3

    self.playerItem = [AVPlayerItem playerItemWithAsset:self.asset     // 4
                        automaticallyLoadedAssetKeys:keys];

    [self.playerItem addObserver:self                                  // 5
                   forKeyPath:STATUS_KEY
                      options:0 context:NULL];

    self.playerView.player = [AVPlayer playerWithPlayerItem:self.playerItem];
    self.playerView.showsSharingServiceButton = YES;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                   ofObject:(id)object
                     change:(NSDictionary *)change
                    context:(void *)context {

    if ([keyPath isEqualToString:STATUS_KEY]) {

        if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
            NSString *title = [self titleForAsset:self.asset];         // 6
            if (title) {
                self.windowForSheet.title = title;
            }
            self.chapters = [self chaptersForAsset:self.asset];        // 7
        }
```

```
        [self.playerItem removeObserver:self forKeyPath:STATUS_KEY];          // 8
    }
}

- (NSString *)titleForAsset:(AVAsset *)asset {
    return nil;
}

- (NSArray *)chaptersForAsset:(AVAsset *)asset {
    return nil;
}
```

(1) 为AVAsset、AVPlayerItem和NSArray添加三个新属性，用于保存即将捕捉的章节数据。

(2) 删除简单的AVPlayer创建代码，并将设置代码提取到一个新方法中，该方法名为setupPlaybackStackWithURL:，这个方法会在windowControllerDidLoadNib:方法中被调用。

(3) 创建一个NSArray来保存commonMetadata和availableChapterLocales键。这些都是我们希望载入的AVAsset键，因为它们能使开发者对资源进行更深入的调查。

(4) 使用新的playerItemWithAsset:automaticallyLoadedAssetKeys:便捷初始化方法创建一个AVPlayerItem。这是Mac OS X 10.9和iOS 7.0版本中新增的实用功能，简化了载入AVAsset键值的过程。

(5) 添加self作为播放条目status属性的监听器，这样当播放条目处于准备播放状态时应用程序就会得到通知。不过前提是被请求的资源属性先载入进来并且播放通道已经准备好。

(6) 如果播放条目的状态为AVPlayerItemStatusReadyToPlay，可以放心地调用请求载入的AVAsset属性。

(7) 调用titleForAsset:方法获取播放资源的标题。如果有可用的标题内容返回，将它设置为窗口的title属性。如果没有，则以资源的文件名作为标题展示。后面会给出titleForAsset:方法的实现，不过现在我们只给出一个存根实现令其返回nil。

(8) 调用chaptersForAsset:方法获取当前资源中保存章节书签的对象数组。稍后会讨论如何获取章节数据，不过现在我们仅给出chaptersForAsset:方法的存根实现。

我们所做的修改并没有改变应用程序的功能，只是为下面将要介绍的功能做准备。作为完整性检查，再次运行应用程序并确保在功能方面同之前的应用程序保持一致。

在为应用程序添加更多功能之前，下面看一下如何实现titleForAsset:方法，如代码清单5-3所示。这一方法用到了前两章所介绍的AV Foundation标准元数据功能。

代码清单 5-3　titleForAsset:的实现

```
- (NSString *)titleInMetadata:(NSArray *)metadata {
    NSArray *items =                                                          // 1
        [AVMetadataItem metadataItemsFromArray:metadata
                                    withKey:AVMetadataCommonKeyTitle
                                   keySpace:AVMetadataKeySpaceCommon];

    return [[items firstObject] stringValue];                                 // 2
}

- (NSString *)titleForAsset:(AVAsset *)asset {
```

```
    NSString *title = [self titleInMetadata:asset.commonMetadata];          // 3
    if (title && ![title isEqualToString:@""]) {
        return title;
    }
    return nil;

}
```

(1) 添加一个新的titleInMetadata:方法，将标题信息从AVMetadataItem实例集合中解析出来。通过调用metadataItemsFromArray:withKey:keySpace:类方法在通用键空间中获取标题对应的键。

(2) 获取这个数组中的firstObject并得到对应的NSString值。通用键空间中的标题值类型都是字符串，所以可以对它们使用stringValue，强制转换为NSString类型。

(3) 调用titleInMetadata:方法，并将资源对象的commonMetadata作为参数。如果找到有效的标题值，则返回给调用函数，否则返回nil。

再次运行应用程序，Chapter 5目录中的视频文件包含标题元数据，所以我们可以看到窗口的标题栏中会显示资源的标题。

5.6 章的处理

如果使用浮动控件类型或内嵌控件类型的话，AVPlayerView在视频文件具有可以展示的章数据情况下会自动显示章菜单。虽然这一做法非常方便，但是如果需要直接使用章数据实现一些额外功能，或应用程序所使用的控件类型不支持自动章菜单该怎么办呢？幸运的是，AV Foundation可以通过AVTimedMetadataGroup类直接处理章数据。

时间相关的元数据与第3章介绍的静态元数据同样重要，不过和将其作为一个整体应用于资源不同的是，时间相关元数据只用于资源时间轴内特定的时间范围。AVAsset定义了两个方法可以获取这个数据：

```
chapterMetadataGroupsWithTitleLocale:containingItemsWithCommonKeys:
chapterMetadataGroupsBestMatchingPreferredLanguages:
```

这两个方法返回AVTimedMetadataGroup对象的NSArray数组，数组中的对象为资源中包含的章元数据。从方法签名就可以推断出，章数据取决于位置。在调用这两个方法前，首先需要确保资源的availableChapterLocales键已经载入，如下面代码所示。

```
NSURL *url = // asset URL;
AVAsset *asset = [AVAsset assetWithURL:url];
NSString *key = @"availableChapterLocales";

[asset loadValuesAsynchronouslyForKeys:@[key] completionHandler:^{

    AVKeyValueStatus status = [asset statusOfValueForKey:key error:nil];

    if (status == AVKeyValueStatusLoaded) {

        NSArray *langs = [NSLocale preferredLanguages];
        NSArray *chapterMetadata =
```

```
            [asset chapterMetadataGroupsBestMatchingPreferredLanguages:langs];

        // Process AVTimeMetadataGroup objects
    }

}];
```

AVTimedMetadataGroup包含两个属性：timeRange和items。timeRange属性保存着一个CMTimeRange结构，该结构包含用于表示时间范围起点的CMTime值和定义资源时长的CMTime值。这允许确定章元数据所对应的资源时间轴上的时间范围。章的标题和作为可选项的缩略图可在items属性中找到，items属性包含了一个由来自于通用键空间的AVMetadataItem对象组成的NSArray。

代码清单5-4给出了chaptersForAsset:方法在具体实践中的实现。

代码清单 5-4　chaptersForAsset:方法的实现

```
- (NSArray *)chaptersForAsset:(AVAsset *)asset {

    NSArray *languages = [NSLocale preferredLanguages];                    // 1

    NSArray *metadataGroups =                                              // 2
        [asset chapterMetadataGroupsBestMatchingPreferredLanguages:languages];

    NSMutableArray *chapters = [NSMutableArray array];

    for (NSUInteger i = 0; i < metadataGroups.count; i++) {
        AVTimedMetadataGroup *group = metadataGroups[i];

        CMTime time = group.timeRange.start;
        NSUInteger number = i + 1;
        NSString *title = [self titleInMetadata:group.items];

        THChapter *chapter =                                               // 3
            [THChapter chapterWithTime:time number:number title:title];

        [chapters addObject:chapter];

    }
    return chapters;

}
```

(1) 首先查询NSLocale的preferredLanguages数组，这将返回根据用户语言首选项排序的语言编码。我们的示例中，第一个元素就是对应英文语言的en。

(2) 获取资源中与用户首选语言最匹配的章元数据组。

(3) 遍历每个AVTimedMetadataGroup对象并提取对应的相关数据。具体来说我们需要获取开始时间timeRange并通过我们之前定义的titleInMetadata:方法获取标题信息。同时还要基于当前循环索引为每章创建章号。将这个数据保存在一个自定义对象THChapter中，之后使用该数据时就会很方便。

我们已经成功获取章节元数据并将它妥善封装到THChapter对象集合中，那么如何使用这些信息呢？通过利用AVPlayerView给出的另一个自定义点来实际使用这些数据。AVPlayerView

定义了一个actionPopUpButtonMenu方法让开发者向播放器控件添加自定义NSMenu。我们创建一个菜单用来实现跳转到下一章和前一章的功能。如代码清单5-5所示，首先构建NSMenu。

代码清单 5-5 创建一个 Action Menu

```
- (void)setupActionMenu {

    NSMenu *menu = [[NSMenu alloc] init];                                    // 1
    [menu addItem:[[NSMenuItem alloc] initWithTitle:@"Previous Chapter"
                                          action:@selector(previousChapter:)
                                     keyEquivalent:@""]];
    [menu addItem:[[NSMenuItem alloc] initWithTitle:@"Next Chapter"
                                          action:@selector(nextChapter:)
                                     keyEquivalent:@""]];

    self.playerView.actionwPopUpButtonMenu = menu;                           // 2
}

- (void)previousChapter:(id)sender {

}

- (void)nextChapter:(id)sender {

}
```

(1) 创建一个新的NSMenu实例，添加“Previous Chapter”(上一章)和“Next Chapter”(下一章)菜单项，分别取名previousChapter: 和nextChapter:选择器。现在暂时提供这两个方法的存根实现。

(2) 将这个菜单设置到播放视图的actionPopUpButtonMenu属性。当使用浮动类型控件或内嵌类型控件时就会显示该菜单。

在添加菜单前，首先需要调用setupActionMenu方法。修改observeValueForKeyPath:方法，如代码清单5-6所示。

代码清单 5-6 修改 observeValueForKeyPath:方法

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {

    if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
        NSString *title = [self titleForAsset:self.asset];
        if (title) {
            self.windowForSheet.title = title;
        }
        self.chapters = [self chaptersForAsset:self.asset];

        // Create action menu if chapters are available
        if ([self.chapters count] > 0) {
            [self setupActionMenu];
        }
```

```
    }
    [self.playerItem removeObserver:self forKeyPath:STATUS_KEY];
}
```

我们希望在资源中存在章节数据时才显示这个菜单，如果chapters数组非空，就调用setupActionMenu方法。

再次运行应用程序并确认菜单是否显示。点击菜单项，不过由于我们还没有实现这些方法，所以什么也不会发生。在为这些方法添加具体实现前，首先需要添加一些额外的辅助代码来找到具体的上一章和下一章(如代码清单5-7所示)。下面的代码就开始变得相对复杂一些，也是我们第一次看到Core Media类型和函数的高级用法，所以我们仔细研究一下这些方法。

代码清单 5-7　查找章节

```
- (THChapter *)findPreviousChapter {

    CMTime playerTime = self.playerItem.currentTime;
    CMTime currentTime = CMTimeSubtract(playerTime, CMTimeMake(3, 1));          // 1
    CMTime pastTime = kCMTimeNegativeInfinity;

    CMTimeRange timeRange = CMTimeRangeMake(pastTime, currentTime);             // 2

    return [self findChapterInTimeRange:timeRange reverse:YES];                 // 3
}

- (THChapter *)findNextChapter {

    CMTime currentTime = self.playerItem.currentTime;                           // 4
    CMTime futureTime = kCMTimePositiveInfinity;

    CMTimeRange timeRange = CMTimeRangeMake(currentTime, futureTime);           // 5

    return [self findChapterInTimeRange:timeRange reverse:NO];                  // 6
}

- (THChapter *)findChapterInTimeRange:(CMTimeRange)timeRange
                              reverse:(BOOL)reverse {

    __block THChapter *matchingChapter = nil;

    NSEnumerationOptions options = reverse ? NSEnumerationReverse : 0;
    [self.chapters enumerateObjectsWithOptions:options                          // 7
                                     usingBlock:^(id obj,
                                                  NSUInteger idx,
                                                  BOOL *stop) {

        if ([(THChapter *)obj isInTimeRange:timeRange]) {                       // 8
            matchingChapter = obj;
            *stop = YES;
        }
    }];

    return matchingChapter;                                                     // 9

}
```

(1) 要查找前一章，首先定义两个CMTime值。第一个值为播放控件当前时间减掉3秒，第二个值是使用kCMTimeNegativeInfinity常量定义一个当前时间之前的无限大时间。如果视频正处于播放状态，则时间会不断向前增加，所以当用户选择菜单项时应该留一些时间余地。如果不这么做，用户就会陷入再次回到章起始时间的循环中。当计算currentTime时我们对它使用CMTimeSubtract函数减掉3秒，并以此结果值作为currentTime。

(2) 使用CMTimeRangeMake函数创建CMTimeRange，将pastTime作为时间范围的start值，将currentTime作为时间范围的duration值。这个时间范围用于在THChapter对象集合中查找。

(3) 调用findChapterInTimeRange:reverse:方法为reverse:参数传递YES值。表示我们希望在chapters数组中后向进行搜索。

(4) 与findPreviousChapter方法类似，获取播放控件的当前时间。不需要进行任何特别的计算，因为当时间前进时不需要考虑时间相关的问题。我们还使用kCMTimePositiveInfinity常量创建一个当前时间之后的无限大时间，用来标记时间范围的上限。

(5) 创建一个CMTimeRange，使用当前时间作为start，使用预计时间作为duration。

(6) 调用findChapterInTimeRange:reverse:方法，这次将reverse:参数设为NO，因为我们希望向前查找chapters数组。

(7) 枚举chapters数组中的对象，按照reverse:参数指定的顺序遍历集合。

(8) 调用章的isInTimeRange:方法判断章的起点时间是否在时间范围之内。如果在，则可以找到匹配并停止对元素的处理。

(9) 最后，返回匹配的THChapter。如果没有发现匹配则返回nil，比如在时间轴开始时向后导航和在时间轴结尾处向前导航这两种情况。

我们不对完整的THChapter类实现展开讲解，因为它是一个简单保存数据的对象，不过我们一定要注意isInTimeRange:方法，因为它使用了Core Media框架中的宏。

```
- (BOOL)isInTimeRange:(CMTimeRange)timeRange {
    return CMTIME_COMPARE_INLINE(_time, >, timeRange.start) &&
           CMTIME_COMPARE_INLINE(_time, <, timeRange.duration);
}
```

Core Media定义了大量有用的函数和宏。其中一个最常用的宏就是CMTIME_COMPARE_INLINE。这个宏包含两个CMTime值和一个比较运算符，并返回一个布尔值用来表示比较结果，在isInTimeRange:方法中，我们会判断章的时间是否处于start和duration时间范围内。

现在最难的部分已经完成了，我们准备实现这个功能。代码清单5-8给出了剩下几个方法的实现。

代码清单 5-8 previousChapter:和 nextChapter:方法的实现。

```
- (void)previousChapter:(id)sender {
    [self skipToChapter:[self findPreviousChapter]];                    // 1
}

- (void)nextChapter:(id)sender {
    [self skipToChapter:[self findNextChapter]];                        // 2
}

- (void)skipToChapter:(THChapter *)chapter {                           // 3
```

```
    [self.playerItem seekToTime:chapter.time completionHandler:^(BOOL done) {
        [self.playerView flashChapterNumber:chapter.number
                                chapterTitle:chapter.title];
    }];

}
```

(1) 当用户选择Previous Chapter菜单，就会调用findPreviousChapter方法，将结果传递给skipToChapter:方法。

(2) 同样，当用户选择Next Chapter菜单，就会调用findNextChapter方法，将结果传递给skipToChapter:方法。

(3) 最后，在skipToChapter:方法中，调用播放条目的seekToTime:completionHandler:方法，将章时间作为第一个参数。在completionHandler:回调中，调用播放器视图的flashChapterNumber:chapterTitle:方法，将章节编号和标题显示在播放器视图中。

再次运行应用程序，打开视频，尝试跳过各场景。

5.7　启用修剪

除了AVPlayerView定义的即用型播放体验外，还有一种实用方式。如果你用过目前版本的QuickTime Player，可能会注意到Edit菜单中的Trim选项，选择该选项可以对资源进行修剪，如图5-13所示。

图 5-13　QuickTime Player 的修剪界面

AVPlayerView定义了相同的功能和界面，最主要的是它非常容易添加到应用程序中。示例应用程序已经定义了一个Trim选项控件和startTrimming:存根方法，不过我们需要为这个方法提供具体的实现，如代码清单5-9所示。

代码清单 5-9　启用修剪

```
- (IBAction)startTrimming:(id)sender {
    [self.playerView beginTrimmingWithCompletionHandler:NULL];

}
```

读者可能会想代码这么简单是不是我们没有写全啊，那你就亲自试试吧。运行应用程序并打开示例视频，进入Edit菜单并选择Trim菜单项。我们所看到的修剪界面与QuickTime Player的一样，Trim和Cancel按钮的功能都和预期的一样。

在修剪前，首先需要查询播放器视图的canBeginTrimming属性。有一些情况下该属性会返回NO。第一种情况是如果修剪界面已经处于展示中，因为这时打开修剪界面是没有意义的。第二种情况是资源已明确不能被修剪。比如不能对从iTunes Store上购买的电影或电视节目进行修剪。查询该属性最常见的位置是在validateUserInterfaceItem:方法中，以便启用或禁用相应的菜单项(如代码清单5-10所示)。

代码清单 5-10 实现方式

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)item {

    SEL action = [item action];

    if (action == @selector(startTrimming:)) {
        return self.playerView.canBeginTrimming;
    }

    return YES;

}
```

现在运行应用程序，当修剪界面打开时，Trim菜单项处于禁用状态。

读者朋友可能会好奇资源的修剪是如何完成的。开发者不需要添加任何修剪资源的代码，如果回顾之前的内容，就应该知道AVAsset是不可变对象，也就是说它是不能被修改的。那点击Trim按钮会发生什么呢？是的，会产生错觉。当我们点击Trim按钮，播放器视图会修改当前AVPlayerItem的两个属性。左侧的修剪控件会设置播放条目的reversePlaybackEndTime属性，右边的控件设置forwardPlaybackEndTime。这些属性定义了资源的有效时间轴。如果修剪视频之后调用Trim菜单项，会发现所有内容保持不变并且修剪界面处于已修剪的部分。这并不是AVPlayerView的限制，而仅是为了保持AV Foundation的不可变设计理念，确实在QuickTime Player中也是这样。所以如果资源不能被修改，那如何才能保存这些变更呢？这就带来了下面有关资源导出的主题。

5.8 导出

要保存修剪操作的结果，需要使用AVAssetExportSession类将当前资源导出为一个新的资源。我们已经见过这个类了，它是AV Foundation框架最常用的类之一。

应用程序已经在File菜单中加入了Export菜单项，并使它同对应的startExporting:方法建立了关联，现在需要的就是具体的实现代码了。首先为文档对象的类扩展添加一组新的属性，如代码清单5-11所示。

代码清单 5-11 添加导出属性

```
#import "THExportWindowController.h"

@interface THDocument () <THExportWindowControllerDelegate>
```

```
@property (strong) AVAsset *asset;
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;
@property (strong) AVAssetExportSession *exportSession;
@property (strong) THExportWindowController *exportController;

@property (weak) IBOutlet AVPlayerView *playerView;

@end
```

配置好之后，继续看startExporting:方法的实现，如代码清单5-12所示。

代码清单 5-12　startExporting:方法的实现

```
- (IBAction)startExporting:(id)sender {

    [self.playerView.player pause];                                          // 1

    NSSavePanel *savePanel = [NSSavePanel savePanel];

    [savePanel beginSheetModalForWindow:self.windowForSheet
                    completionHandler:^(NSInteger result) {

        if (result == NSFileHandlingPanelOKButton) {
            // Order out save panel as the export window will be shown
            [savePanel orderOut:nil];

            NSString *preset = AVAssetExportPresetAppleM4V720pHD;
            self.exportSession =                                             // 2
            [[AVAssetExportSession alloc] initWithAsset:self.asset
                                        presetName:preset];

            CMTime startTime = self.playerItem.reversePlaybackEndTime;
            CMTime endTime = self.playerItem.forwardPlaybackEndTime;
            CMTimeRange timeRange = CMTimeRangeMake(startTime, endTime);     // 3

            // Configure the export session                                  // 4
            self.exportSession.timeRange = timeRange;
            self.exportSession.outputFileType =
                [self.exportSession.supportedFileTypes firstObject];
            self.exportSession.outputURL = savePanel.URL;

            self.exportController = [[THExportWindowController alloc] init];
            self.exportController.exportSession = self.exportSession;
            self.exportController.delegate = self;
            [self.windowForSheet beginSheet:self.exportController.window     // 5
                        completionHandler:nil];

            [self.exportSession exportAsynchronouslyWithCompletionHandler:^{
                // Tear down                                                 // 6
                [self.windowForSheet endSheet:self.exportController.window];
                self.exportController = nil;
                self.exportSession = nil;
            }];
        }
```

```
    }];
}

- (void)exportDidCancel {
    [self.exportSession cancelExport];                                    // 7

}
```

(1) 如果视频处于播放状态则需要先暂停，因为当视频导出时用户不能与播放栏控件互动。

(2) 创建一个新的AVAssetExportSession实例，传递资源对象和导出配置。这里使用AVAssetExportPresetAppleM4V720pHD参数，它将创建一个720p MPEG-4 视频文件，不过也可以任意选择其他预设配置。比如，如果正在导出一个修剪操作，可以考虑使用AVAssetExport-PresetPassthrough，不需要对媒体进行转码。

(3) 基于反向和正向播放的结束时间创建一个时间范围。如果没有执行修剪操作，反向和前向结束时间就等于kCMTimeInvalid，这样会得到一个等于kCMTimeRangeInvalid的时间范围。在导出会话上设置kCMTimeRangeInvalid范围值，就会导出全部视频。

(4) 为导出会话配置时间范围、一个与会话兼容的输出文件类型以及从NSSavePanel获取的输出URL。

(5) 示例应用程序提供了一个THExportWindowController类，它是一个简单的NSWindow-Controller子类，用于展示导出进度窗口。创建一个新的实例并设置self作为委托，在新的页面展示该窗口。

(6) 开始导出资源，当回调函数被调用时就可以关掉导出页面和控制器实例。

(7) 如果用户在导出进度窗口中点击Cancel按钮，就会取消导出会话。

是时候体验所有功能了，运行应用程序并打开示例视频，修剪其中感兴趣的部分并导出一个视频副本到硬盘。要验证导出的结果是否可用，应该使用什么？当然是KitTimePlayer!

5.9 传统资源的兼容

本章最后一个要解决的问题并不是AV Foundation范畴的，不过它肯定与我们的主题相关，对于基于QuickTime的应用程序尤其重要。

QuickTime支持很多编解码方式和媒体类型，这也是QuickTime被认为是一个功能强大且灵活的平台的原因之一。不过大部分它所支持的编解码都过时了，并且它支持的曲目类型也被最新的技术所超过。AV Foundation不支持这些传统的特性，而是将注意力放在苹果公司所认为的未来最流行的编解码和曲目类型处理上。

对于当今的媒体环境，这一重点的编解码集合和媒体类型完全有理由更进一步；不过Macintosh多媒体环境不是今天开始的。用户可能已经使用这些不被支持的编码格式许多年了，所以放弃这些格式也不是用户所期望的。那么我们应该怎么办呢？

少有的是，苹果公司在Mac OS X 10.9中为这些过时的框架给出了一个新的类。确切地讲，为QTKit框架加入了一个新类QTMovieModernizer。该类可将过时的媒体内容转换为AV Foundation支持的格式。实际上，如果你使用Mavericks系统下的QuickTime Player播放器打开一个早期QuickTime文件，就会发现其实已经实现了这个转变功能。当用户打开文件时，展

示内容前会看到一条消息显示“转换中...”。 QuickTime Player使用这个类转换媒体，开发者也可在自己的应用程序中使用它。

运行该应用程序，选择File，再选择Open…，找到Chapter 5目录。在里面可以找到一个子目录Legacy，其中包含一些使用传统编解码的QuickTime文件。打开其中一个文件并点击Play按钮。可以听到有声音传出，不过没有视频影像展示，这是因为AV Foundation无法对该视频曲目解码。下面看一下如何使用QTMovieModernizer类解决这一问题。首先对之前的代码进行一些修改，如代码清单5-13所示。

代码清单 5-13　Movie Modernization 准备

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>
#import "THChapter.h"
#import <QTKit/QTKit.h>
#import "NSFileManager+THAdditions.h"
#import "THWindow.h"

#import "THExportWindowController.h"

#define STATUS_KEY @"status"

@interface THDocument () <THExportWindowControllerDelegate>

@property (strong) AVAsset *asset;
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;
@property (strong) AVAssetExportSession *exportSession;
@property (strong) THExportWindowController *exportController;

@property BOOL modernizing;

@property (weak) IBOutlet AVPlayerView *playerView;

@end

@implementation THDocument

#pragma mark - NSDocument Methods

- (NSString *)windowNibName {
    return @"THDocument";
}

- (void)windowControllerDidLoadNib:(NSWindowController *)controller {
    [super windowControllerDidLoadNib:controller];

    if (!self.modernizing) {
        [self setupPlaybackStackWithURL:[self fileURL]];
    } else {
        [(id)controller.window showConvertingView];
    }
}
```

修改windowControllerDidLoadNib:方法，根据modernizing属性的状态酌情执行相应的设置，modernizing属性用于表示QTMovieModernizer当前是否正在运行一个会话。如果modernizing为false，则运行正常设置，如果modernizing为true，在窗口对象中调用showConvertingView，展示一个进度条。

完成代码清单5-13的修改后，下面看一下实际的现代化过程。按照下面的应用程序对readFromURL:ofType:error:方法进行修改，如代码清单5-14所示。

代码清单 5-14 运行现代化

```
- (BOOL)readFromURL:(NSURL *)url
            ofType:(NSString *)typeName
             error:(NSError *__autoreleasing *)outError {

    NSError *error = nil;

    if ([QTMovieModernizer requiresModernization:url error:&error]) {          // 1

        self.modernizing = YES;

        NSURL *destURL = [self tempURLForURL:url];                             // 2

        if (!destURL) {
            self.modernizing = NO;
            NSLog(@"Error creating destination URL, skipping modernization.");
            return NO;
        }

        QTMovieModernizer *modernizer =                                        // 5
            [[QTMovieModernizer alloc] initWithSourceURL:url
                                        destinationURL:destURL];

        modernizer.outputFormat = QTMovieModernizerOutputFormat_H264;          // 6

        [modernizer modernizeWithCompletionHandler:^{
            if (modernizer.status ==                                           // 7
                    QTMovieModernizerStatusCompletedWithSuccess) {

                dispatch_async(dispatch_get_main_queue(), ^{
                    [self setupPlaybackStackWithURL:destURL];                  // 8
                    [(id)self.windowForSheet hideConvertingView];
                });
            }
        }];
    }

    return YES;
}

- (NSURL *)tempURLForURL:(NSURL *)url {

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *dirPath =                                                        // 3
        [fileManager temporaryDirectoryWithTemplateString:@"kittime.XXXXXX"];
```

```
    if (dirPath) {                                                    ``////// 4
        NSString *filePath =
            [dirPath stringByAppendingPathComponent:[url lastPathComponent]];
        return [NSURL fileURLWithPath:filePath];
    }

    return nil;
}
```

(1) 首先在QTMovieModernizer上调用requiresModernization:error:类方法来判断媒体是否需要转换。这个方法会确定是否需要一个转换过程。要知道这是一个同步调用，所以在实际应用程序中通常是在后台线程中执行该查找，这样就不会阻塞用户界面。如果该方法返回YES，则继续执行会话。有时返回值为YES，错误指针会包含一些在转换过程中可能出现的错误信息描述，所以在实际应用程序中对这一点要格外注意。

(2) 在NSTemporaryDirectory()中获取一个写入转换文件的NSURL地址。如果没有找到有效地址，会终止该方法，弹出一个默认提示框，告诉用户无法读取文件。

(3) temporaryDirectoryWithTemplateString:是我们为NSFileManager添加的一个分类方法。这个方法基于传递给方法的模板字符串并利用mkdtemp函数在NSTemporaryDirectory()之下创建一个唯一的目录。

(4) 如果临时目录创建好了，再基于源URL的文件名创建一个新的NSURL并将其返回给调用方法。

(5) 创建一个新的QTMovieModernizer实例，传递源地址和目的地址URL。

(6) 为转换媒体资源指定一个输出格式，这里我们使用h.264编码，不过你也可以使用QTMovieModernizerOutputFormat_AppleProRes422常量或QTMovieModernizerOutputFormat_AppleProRes4444常量分别指定使用Apple ProRes 422或4444编码格式。

(7) 通过调用modernizeWithCompletionHandler:方法开始现代化过程。当调用completion handler块时，需要查看它的状态来判断转换过程是否成功。如果成功，则调度回主线程继续处理。

(8) 调用setupPlaybackStackWithURL:方法，并传递转换资源的URL。最后，将之前的转换视图删除。

再次运行应用程序，重新打开一个示例传统格式的文件。这次会看到一个带有转换效果的spinner视图。当该视图消失后，就可以点击Play按钮观看视频内容了，多亏了QTMovie-Modernizer！

5.10　小结

本章我们对AV Kit框架的使用有了比较深入的了解。iOS版本的AV Kit定义了AVPlayer-ViewController类，为使用AV Foundation创建iOS最新视频播放应用程序提供了一种简单方法。Mac OS X版本的AV Kit定义了AVPlayerView类，为Mac平台创建视频播放应用程序提供了一种简单且功能强大的方法，并填补了无法播放QTMovieView传统格式的空白。使用AV Kit可让开发者在两个平台上创建具有相同功能、特色和用户界面的视频播放器。我们还学习了使用AVTimeMetadataGroup类来处理存储在AVAsset中的基于时间的章元数据。虽然本章使用

Mac平台讨论这些问题，不过这些功能和方法在iOS平台也是适用的。最后，我们学习了QTMovieModernizer类。虽然这个类是QTKit框架的一部分，不过它可以使Mac平台的AV Foundation应用程序拥有更完善的功能。

5.11 挑战

KitTime Player实现了QuickTime Player播放器的几个核心功能。用所学到的知识，试着模仿QuickTime Player的其他功能。比如，尝试用AVAssetExportSession实现一些QuickTime Player更高级的导出功能。将章跳转功能移到View菜单中并添加正确的键绑定。添加Open Location菜单项支持远程媒体资源播放。甚至可以尝试使用第2章中介绍的音频类实现QuickTime Player的音频录制功能。

第 II 部分

媒体捕捉

第 6 章　捕捉媒体
第 7 章　高级捕捉功能
第 8 章　读取和写入媒体

第6章

捕捉媒体

2007年推出的iPhone手机极大颠覆了移动手机产业，并完全重新定义了我们对移动计算的理解。不过随着iPhone的推出并没有立即显现的是它对数码相机产业的影响。iPhone成了市场上最受欢迎的数码相机之一(http://www.flickr.com/cameras)，对于大部分人而言，iPhone已经完全取代了传统傻瓜相机和便携式摄像机。对传统相机的摒弃同时还改变了我们使用数码照片和视频的方式。我们的照片不再仅是相册中的照片，视频也不仅仅只能在客厅中播放。现在只需要点击手指就可以捕捉、编辑和分享全世界的媒体资源。本章将学习AV Foundation捕捉功能的知识，并利用学到的知识创建一个优秀的照片或视频应用程序。

6.1 捕捉功能综述

AV Foundation的照片和视频捕捉功能从框架搭建之初就是它的一个强项。从iOS 4版本开始，开发者就可以直接访问iOS设备的摄像头和摄像头生成的数据，定义一个新的用于照片和视频应用的类。框架的捕捉功能仍然是苹果公司媒体工程师最关注的领域，每个新版本的发布都带来强大的新功能和提升。虽然核心的捕捉类在iOS和OS X上是一致的，但你会发现不同平台下的框架还是有一些区别，这些区别一般都是针对平台的定制功能。比如，Mac OS X为截屏功能定义了AVCaptureScreenInput类，而iOS由于沙盒的限制就没有这个类。我们对于AV Foundation捕捉功能的讲解主要以iOS版本为例，但讨论的大部分功能都适用于OS X。

当开发一个带有捕捉功能的应用程序时会用到许多类，学习框架的功能第一步就是了解其中包含的各个类及每个类所扮演的角色和职责。图6-1给出了开发捕捉功能应用程序时可能用到的一些类。

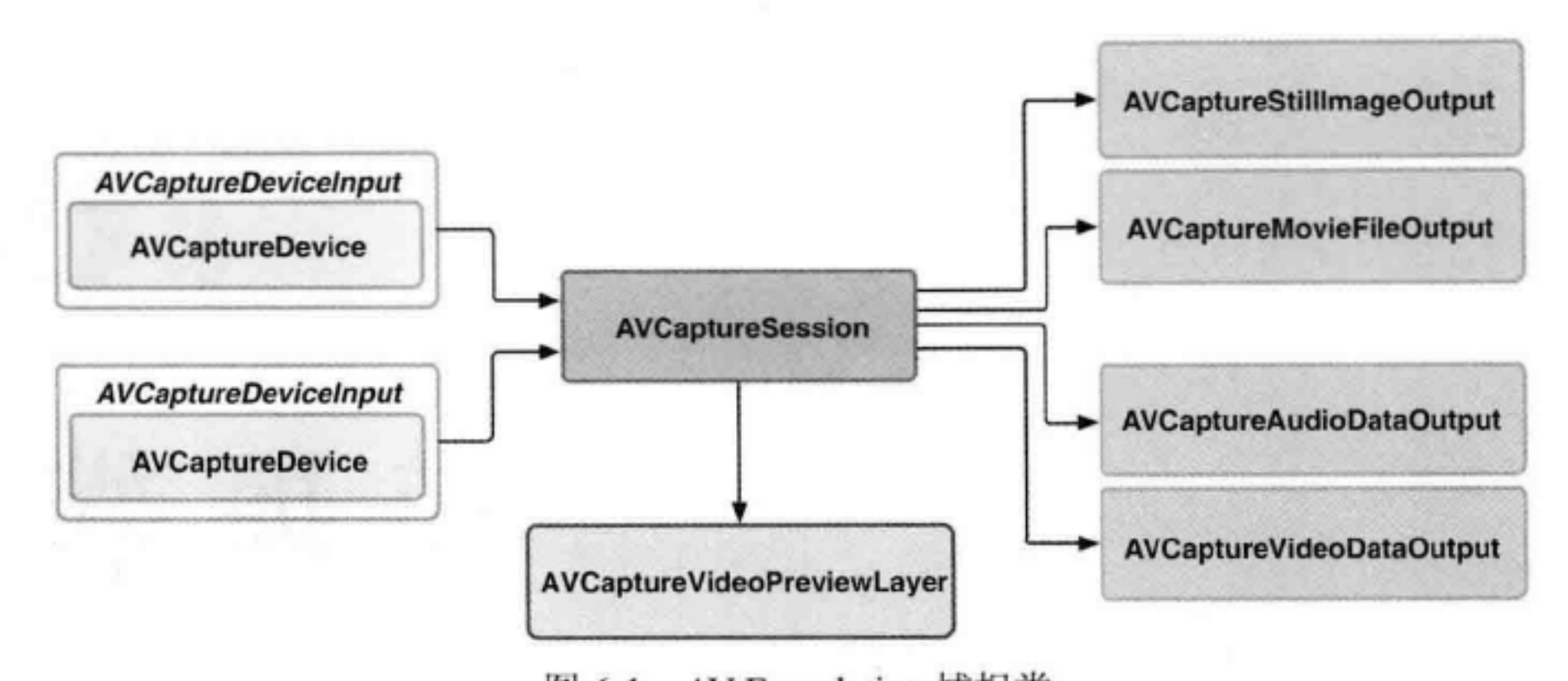

图 6-1 AV Foundation 捕捉类

6.1.1 捕捉会话

AV Foundation捕捉栈的核心类是AVCaptureSession。一个捕捉会话相当于一个虚拟的“插线板”，用于连接输入和输出的资源。捕捉会话管理从物理设备得到的数据流，比如摄像头和麦克风设备，输出到一个或多个目的地。可以动态配置输入和输出的线路，让开发者能够在会话进行中按需重新配置捕捉环境。

捕捉会话还可以额外配置一个会话预设值(session preset)，用来控制捕捉数据的格式和质量。会话预设值默认为AVCaptureSessionPresetHigh，它适用于大多数情况，不过框架仍然提供了多个预设值对输出进行定制，以满足应用程序的特殊需求。

6.1.2 捕捉设备

AVCaptureDevice为诸如摄像头或麦克风等物理设备定义了一个接口。大多数情况下，这些设备都内置于Mac、iPhone或iPad中，不过也可能是外部数码相机或便携式摄像机。AVCaptureDevice针对物理硬件设备定义了大量的控制方法，比如控制摄像头的对焦、曝光、白平衡和闪光灯等。

AVCaptureDevice定义了大量类方法用于访问系统的捕捉设备，最常用的一个方法是defaultDeviceWithMediaType:，它会根据给定的媒体类型返回一个系统指定的默认设备。如下例所示：

```
AVCaptureDevice *videoDevice =
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
```

示例中我们请求一个默认的视频设备，在包含前置摄像头和后置摄像头的iOS系统下，这个方法会返回后置摄像头，因为它是系统默认的摄像头。在带有摄像头的Mac机器上，会返回内置的FaceTime摄像头。

6.1.3 捕捉设备的输入

在使用捕捉设备进行处理前，首先需要将它添加为捕捉会话的输入。不过一个捕捉设备不能直接添加到AVCaptureSession中，但是可以通过将它封装在一个AVCaptureDeviceInput实

例中来添加。这个对象在设备输出数据和捕捉会话间扮演接线板的作用。使用deviceInputWithDevice:error:方法创建AVCaptureDeviceInput，如下所示：

```
NSError *error;

AVCaptureDeviceInput *videoInput =
    [AVCaptureDeviceInput deviceInputWithDevice:videoDevice error:&error];
```

需要为这个方法传递一个有效的NSError指针，因为输入创建中会遇到的错误描述信息都会在这里体现。

6.1.4 捕捉的输出

AV Foundation定义了AVCaptureOutput的许多扩展类。AVCaptureOutput是一个抽象基类，用于为从捕捉会话得到的数据寻找输出目的地。框架定义了一些这个抽象类的高级扩展类，比如AVCaptureStillImageOutput和AVCaptureMovieFileOutput，使用它们可以很容易地实现捕捉静态照片和视频的功能。还可以在这里找到底层扩展，比如AVCaptureAudioDataOutput和AVCaptureVideoDataOutput，使用它们可以直接访问硬件捕捉到的数字样本。使用这些底层输出类需要对捕捉设备的数据渲染有更好的理解，不过这些类可以提供更强大的功能，比如对音频和视频流进行实时处理。

6.1.5 捕捉连接

在上面的插图中有一个类没有明确的名字，而是由一个连接不同组件的黑色箭头所表示，这就是AVCaptureConnection类。捕捉会话首先需要确定由给定捕捉设备输入渲染的媒体类型，并自动建立其到能够接收该媒体类型的捕捉输出端的连接。比如，AVCaptureMovieFileOutput可以接受音频和视频数据，所以会话会确定哪些输入产生视频，哪些输入产生音频，并正确地建立该连接。对这些连接的访问可以让开发者对信号流进行底层的控制，比如禁用某些特定的连接，或在音频连接中访问单独的音频轨道。

6.1.6 捕捉预览

如果不能在影像捕捉中看到正在捕捉的场景，那么应用程序就不会那么好用。幸运的是，框架定义了AVCaptureVideoPreviewLayer类来满足该需求。预览层是一个Core Animation的CALayer子类，对捕捉视频数据进行实时预览。这个类所扮演的角色类似于AVPlayerLayer，不过还是针对摄像头捕捉的需求进行了定制。像AVPlayerLayer一样，AVCaptureVideoPreviewLayer也支持视频重力的概念，可以控制视频内容渲染的缩放和拉伸效果，如图6-2、图6-3和图6-4所示。

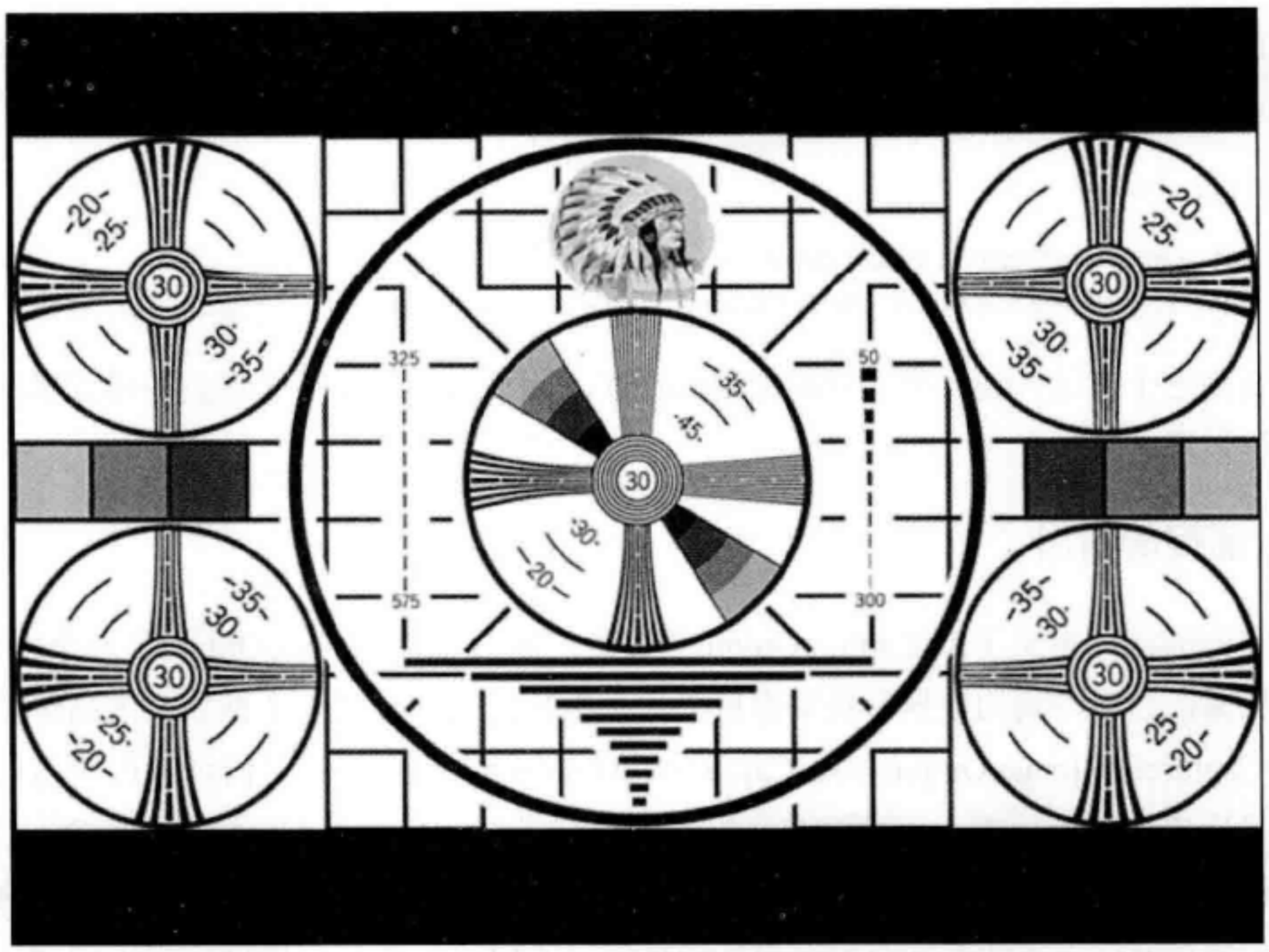

图 6-2 AVLayerVideoGravityResizeAspect 在承载层区域内缩放视频，保持视频原始宽高比。这个值为默认值，适用于大多数情况

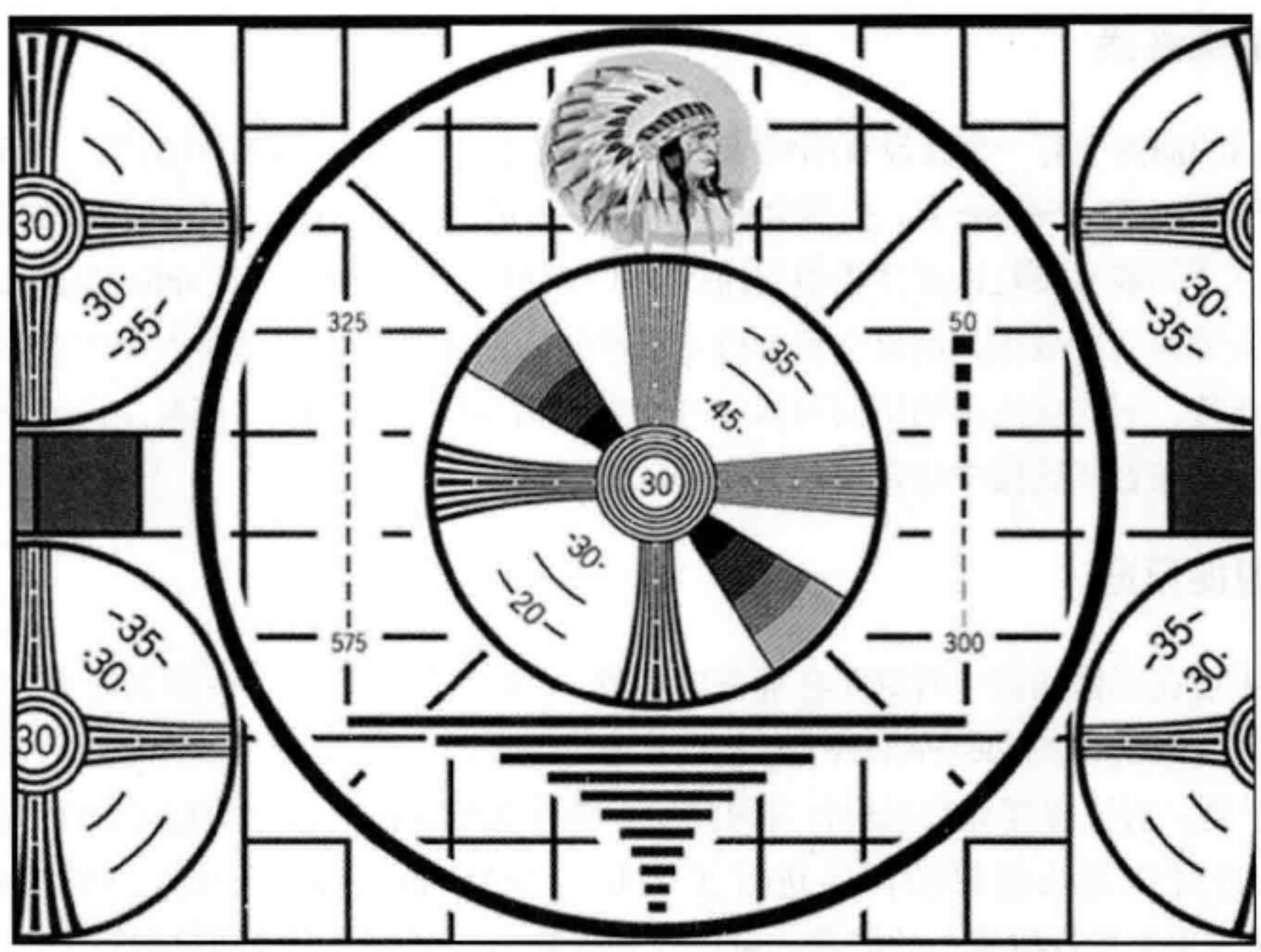

图 6-3 AVLayerVideoGravityResizeAspectFill 按照视频的宽高比将视频拉伸填满整个图层，通常会导致视频图片被裁剪

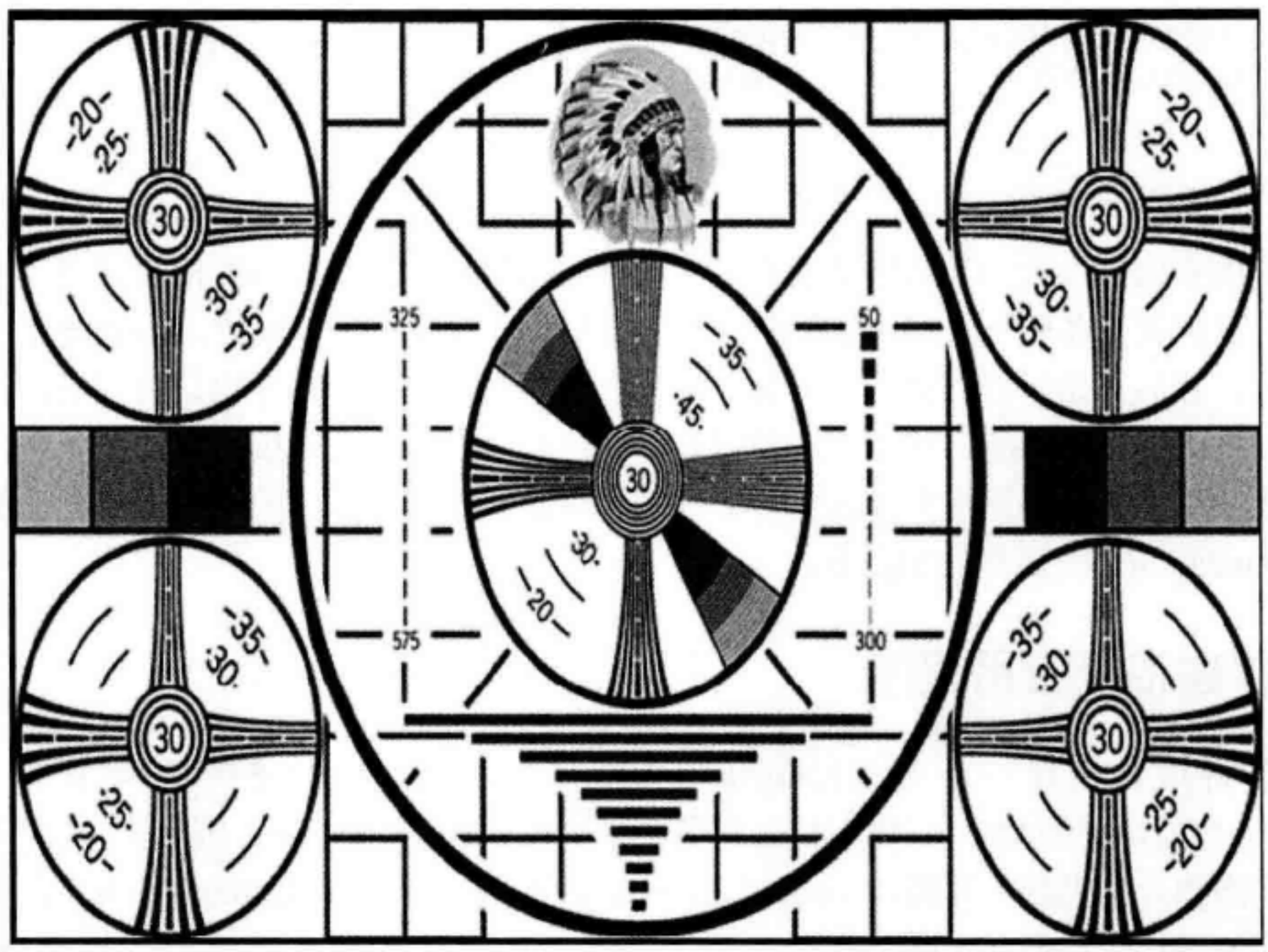

图 6-4 AVLayerVideoGravityResize 拉伸视频内容以匹配承载层大小。这是最不常用的情况，因为会导致由视频图片扭曲带来的 funhouse effect 效应

6.2 简单的秘籍

我们需要站在更高层面理解捕捉类，那么先看一下如何为简单的相机应用程序创建一个捕捉会话。

```
//1. Create a capture session.
AVCaptureSession *session = [[AVCaptureSession alloc] init];

//2. Get a reference to the default camera.
AVCaptureDevice *cameraDevice =
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];

//3. Create a device input for the camera.
NSError *error;
AVCaptureDeviceInput *cameraInput =
    [AVCaptureDeviceInput deviceInputWithDevice:cameraDevice error:&error];

//4. Connect the input to the session.
if ([session canAddInput:cameraInput]) {
    [session addInput:cameraInput];
}

//5. Create an AVCaptureOutput to capture still images.
AVCaptureStillImageOutput *imageOutput =
    [[AVCaptureStillImageOutput alloc] init];

//6. Add the output to the session.
```

```
if ([session canAddOutput:imageOutput]) {
    [session addOutput:imageOutput];
}

//7. Start the session and begin the fl
```

上面的示例中，我们为从默认摄像头捕捉静态图片建立了基础的架构。创建了一个捕捉会话，经过捕捉设备输入将默认摄像头添加到会话中，将一个捕捉输出添加到会话用于输出静态图片，然后开始运行该会话，视频数据流就可以开始在系统中传输。典型的会话创建过程会比示例中的代码复杂一些，不过我们的示例已经很好地将这些核心组件组合在一起了。

现在我们对核心的捕捉类都已经有所了解，那么开始通过实际的应用程序更加深入的了解AV Foundation有关捕捉的功能吧。

6.3 创建相机应用程序

本节将通过创建一个名为Kamera的相机应用程序(如图6-5所示)来详细学习AV Foundation有关捕捉的API。示例项目是模仿苹果手机内置的Camera应用程序，支持用户拍摄高质量静态图片和视频，并通过Assets Library框架将它们写入iOS Camera Roll。在实现这个应用程序的过程中，你会实现完成这一核心功能所需的大量实用方法，会对AV Foundation有关捕捉的类有更深入的理解。可以在Chapter 6目录中找到名为Kamera_Starter的初始项目。

图 6-5 Kamera 应用程序用户界面

> **注意：**
>
> 开发 Kamera 应用程序需要开发者使用真机进行编译和测试。大多数 AV Foundation 的功能可以用 iOS 模拟器进行测试，不过捕捉相关 API 只能在真机上测试。

6.3.1 创建预览视图

图6-6给出了Kamera应用程序用户界面的组成图示。我们把重点放在中间层THPreviewView的实现上，因为它直接包含AV Foundation的用户界面。

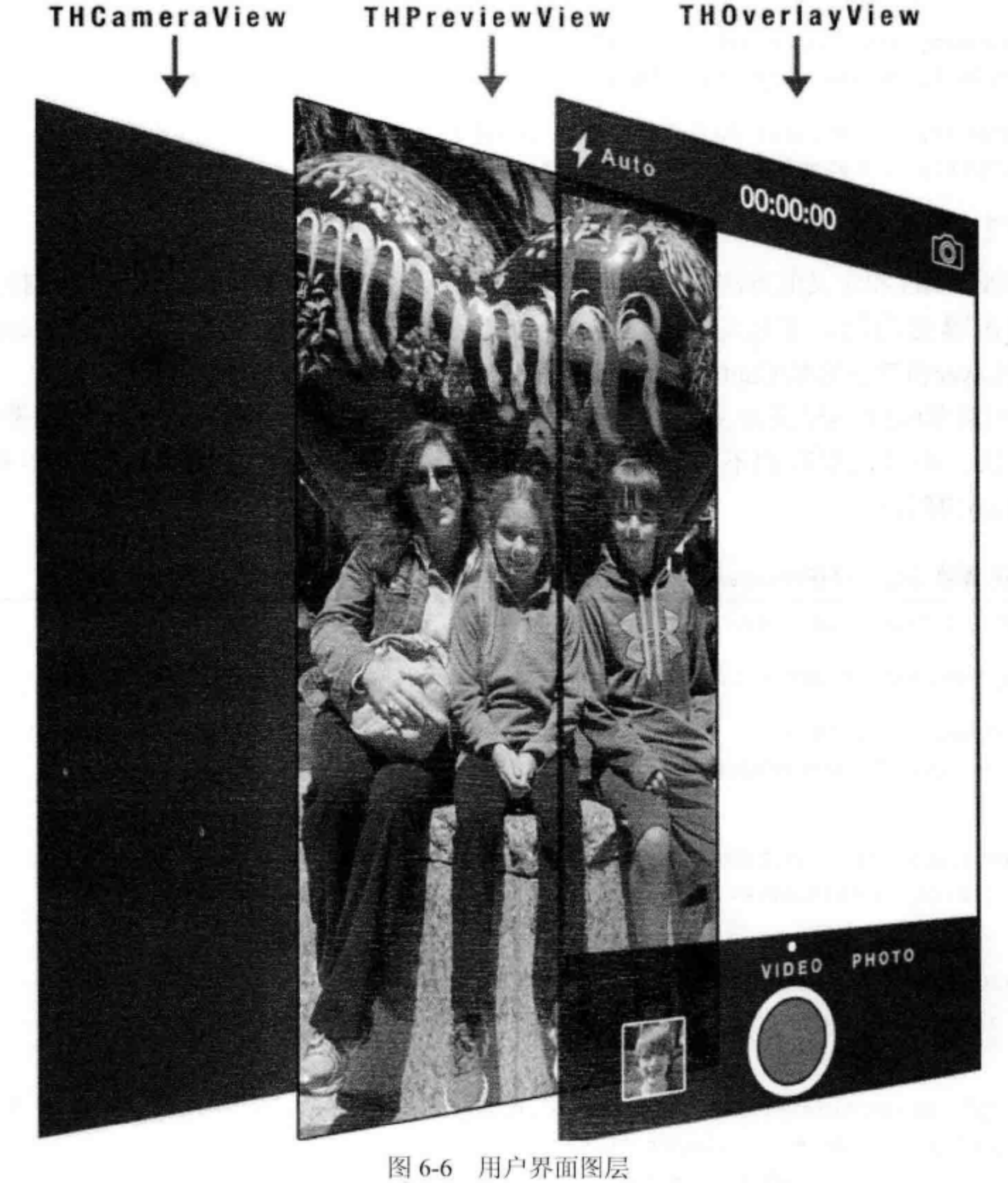

图 6-6 用户界面图层

图6-6中所示的THPreviewView类提供给用户一个摄像头当前拍摄内容的实时预览视图。我们将使用AVCaptureVideoPreviewLayer方法实现这个行为。首先通过认识THPreviewView的接口开始了解如何实现，如代码清单6-1所示。

代码清单 6-1 THPreviewView 接口

```
#import <AVFoundation/AVFoundation.h>

@protocol THPreviewViewDelegate <NSObject>
- (void)tappedToFocusAtPoint:(CGPoint)point;
- (void)tappedToExposeAtPoint:(CGPoint)point;
- (void)tappedToResetFocusAndExposure;
@end

@interface THPreviewView : UIView

@property (strong, nonatomic) AVCaptureSession *session;
@property (weak, nonatomic) id<THPreviewViewDelegate> delegate;

@property (nonatomic) BOOL tapToFocusEnabled;
@property (nonatomic) BOOL tapToExposeEnabled;

@end
```

大部分属性和定义的方法都需要与多种点击手势一起使用。Kamera应用程序支持点击对焦和点击曝光功能。不过这个类定义的关键属性是session，用来关联AVCaptureVideo-PreviewLayer和激活的AVCaptureSession。让我们继续看这个类的具体实现。

代码清单6-2为这个类定义了一个缩写列表。这个项目版本的类中大部分代码都带有触控处理方法，不过这里我们不对触控动作进行讨论。下面让我们重点学习代码中有关AV Foundation的部分。

代码清单 6-2 THPreviewView 的实现

```
#import "THPreviewView.h"

@implementation THPreviewView

+ (Class)layerClass {                                                    // 1
    return [AVCaptureVideoPreviewLayer class];
}

- (void)setSession:(AVCaptureSession *)session {                        // 2
    [(AVCaptureVideoPreviewLayer*)self.layer setSession:session];
}

- (AVCaptureSession*)session {
    return [(AVCaptureVideoPreviewLayer*)self.layer session];
}

- (CGPoint)captureDevicePointForPoint:(CGPoint)point {                  // 3
    AVCaptureVideoPreviewLayer *layer =
        (AVCaptureVideoPreviewLayer *)self.layer;
    return [layer captureDevicePointOfInterestForPoint:point];
}

@end
```

(1) CALayer实例通常都会支持UIView。一般都是一个泛型CALayer实例，不过重写layerClass类方法可以让开发者在创建视图实例时自定义图层类型。可在UIView上重写layerClass方法并返回AVCaptureVideoPreviewLayer类对象。

(2) 重写session属性的访问方法。在setSession:方法中访问视图的layer属性，它是一个AVCaptureVideoPreviewLayer实例，并为它设置AVCaptureSession。这会将捕捉数据直接输出到图层中，并确保与会话状态同步。如果需要的话还要重写session方法来返回捕捉会话。

(3) captureDevicePointForPoint:方法是一个私有方法，用于支持该类定义的不同触控处理方法。该方法将屏幕坐标系上的触控点转换为摄像头坐标系上的点。

坐标空间转换

这里我们将代码清单6-2中的captureDevicePointForPoint:方法重点拿出来引起你对AVCaptureVideoPreviewLayer类定义的几个重要方法的注意。当使用AV Foundation的捕捉API时，一定要理解屏幕坐标系和捕捉设备坐标系的不同。

iPhone 5或iPhone 5s的屏幕坐标系左上角为(0, 0)，垂直模式右下角坐标为(320, 568)，水平模式右下角坐标为(568, 320)。作为一名iOS开发者一定要对这一坐标空间非常熟悉。捕捉设备坐标系有着不同的定义。它们通常是基于摄像头传感器的本地设置，水平方向不可旋转，并且左上角为(0, 0)右下角为(1, 1)，如图6-7所示。

图 6-7 屏幕坐标系(左图)；设备坐标系(右图)

iOS 6之前的版本，要在这两个坐标空间间进行转换非常困难。要精确地将屏幕坐标点转换为摄像头坐标点(或者相反的变换)，开发者必须考虑诸如视频重力、镜像、图层变换和方向等因素进行综合计算。幸运的是，AVCaptureVideoPreviewLayer现在定义了一个转换方法让这一过程变得简单多了。

AVCaptureVideoPreviewLayer定义了两个方法用于在两个坐标系间进行转换：

- **captureDevicePointOfInterestForPoint:** 获取屏幕坐标系的 CGPoint 数据，返回转换得到的设备坐标系 CGPoint 数据。

- **pointForCaptureDevicePointOfInterest**：获取摄像头坐标系的 CGPoint 数据，返回转换得到的屏幕坐标系 CGPoint 数据。

THPreviewView使用captureDevicePointOfInterestForPoint:方法将用户触控点信息转换为摄像头设备坐标系中的点。在Kamera应用程序点击对焦和点击曝光功能的实现中会用到这个转换坐标点。

实现THPreviewView后，让我们继续讨论核心的捕捉代码。

6.3.2 创建捕捉控制器

捕捉会话的代码会包含在一个名为THCameraController的类里面。这个类用于配置和管理不同的捕捉设备，同时也对捕捉的输出进行控制和交互。首先来看THCameraController类的接口，如代码清单6-3所示。

代码清单 6-3 THCameraController 的接口

```
#import <AVFoundation/AVFoundation.h>

extern NSString *const THThumbnailCreatedNotification;

@protocol THCameraControllerDelegate <NSObject>                          // 1
- (void)deviceConfigurationFailedWithError:(NSError *)error;
- (void)mediaCaptureFailedWithError:(NSError *)error;
- (void)assetLibraryWriteFailedWithError:(NSError *)error;
@end

@interface THCameraController : NSObject

@property (weak, nonatomic) id<THCameraControllerDelegate> delegate;
@property (nonatomic, strong, readonly) AVCaptureSession *captureSession;

// Session Configuration                                                 // 2
- (BOOL)setupSession:(NSError **)error;
- (void)startSession;
- (void)stopSession;

// Camera Device Support                                                 // 3
- (BOOL)switchCameras;
- (BOOL)canSwitchCameras;
@property (nonatomic, readonly) NSUInteger cameraCount;
@property (nonatomic, readonly) BOOL cameraHasTorch;
@property (nonatomic, readonly) BOOL cameraHasFlash;
@property (nonatomic, readonly) BOOL cameraSupportsTapToFocus;
@property (nonatomic, readonly) BOOL cameraSupportsTapToExpose;
@property (nonatomic) AVCaptureTorchMode torchMode;
@property (nonatomic) AVCaptureFlashMode flashMode;

// Tap to * Methods                                                      // 4
- (void)focusAtPoint:(CGPoint)point;
- (void)exposeAtPoint:(CGPoint)point;
- (void)resetFocusAndExposureModes;
```

```
/** Media Capture Methods **/                                          // 5

// Still Image Capture
- (void)captureStillImage;

// Video Recording
- (void)startRecording;
- (void)stopRecording;
- (BOOL)isRecording;

@end
```

这次的接口是我们至今为止见过的代码最多的，所以让我们将它分解成几个小的部分来学习。

(1) THCameraControllerDelegate定义了当有错误事件发生时需要在对象委托上调用的方法。

(2) 这些方法都用于配置和控制捕捉会话。

(3) 这些方法能够在不同摄像头中切换以测试摄像头的不同功能，保证用户可以在界面上进行正确选择。

(4) 这些方法可实现点击对焦和点击曝光功能，允许开发者通过多点触控设置焦点和曝光参数。

(5) 这些方法中还包括实现捕捉静态图片和视频的功能。

让我们转到实现文件开始学习具体的实现过程。

6.3.3 设置捕捉会话

首先实现THCameraController类，先从setupSession:方法开始，如代码清单6-4所示。

代码清单 6-4 setupSession:方法

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
#import <AssetsLibrary/AssetsLibrary.h>
#import "NSFileManager+THAdditions.h"

@interface THCameraController ()

@property (strong, nonatomic) dispatch_queue_t videoQueue;
@property (strong, nonatomic) AVCaptureSession *captureSession;
@property (weak, nonatomic) AVCaptureDeviceInput *activeVideoInput;

@property (strong, nonatomic) AVCaptureStillImageOutput *imageOutput;
@property (strong, nonatomic) AVCaptureMovieFileOutput *movieOutput;
@property (strong, nonatomic) NSURL *outputURL;

@end

@implementation THCameraController

- (BOOL)setupSession:(NSError **)error {

   self.captureSession = [[AVCaptureSession alloc] init];                // 1
```

```
    self.captureSession.sessionPreset = AVCaptureSessionPresetHigh;

    // Set up default camera device
    AVCaptureDevice *videoDevice =                                          // 2
        [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];

    AVCaptureDeviceInput *videoInput =                                      // 3
        [AVCaptureDeviceInput deviceInputWithDevice:videoDevice error:error];
    if (videoInput) {
        if ([self.captureSession canAddInput:videoInput]) {                 // 4
            [self.captureSession addInput:videoInput];
            self.activeVideoInput = videoInput;
        }
    } else {
        return NO;
    }

    // Setup default microphone
    AVCaptureDevice *audioDevice =                                          // 5
        [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeAudio];

    AVCaptureDeviceInput *audioInput =                                      // 6
        [AVCaptureDeviceInput deviceInputWithDevice:audioDevice error:error];
    if (audioInput) {
        if ([self.captureSession canAddInput:audioInput]) {                 // 7
            [self.captureSession addInput:audioInput];
        }
    } else {
        return NO;
    }

    // Set up the still image output
    self.imageOutput = [[AVCaptureStillImageOutput alloc] init];            // 8
    self.imageOutput.outputSettings = @{AVVideoCodecKey : AVVideoCodecJPEG};

    if ([self.captureSession canAddOutput:self.imageOutput]) {
        [self.captureSession addOutput:self.imageOutput];
    }

    // Set up movie file output
    self.movieOutput = [[AVCaptureMovieFileOutput alloc] init];             // 9

    if ([self.captureSession canAddOutput:self.movieOutput]) {
        [self.captureSession addOutput:self.movieOutput];
    }

    self.videoQueue = dispatch_queue_create("com.tapharmonic.VideoQueue", NULL);

    return YES;

}
```

(1) 创建的第一个对象就是会话本身。AVCaptureSession是捕捉场景各活动的中心枢纽，也是输入和输入数据需要添加的对象。一个捕捉会话可以配置会话预设值。本例中我们为其

设置AVCaptureSessionPresetHigh预设值，因为它是默认选择且满足Kamera的需求。其他会话预设值可在有关AVCaptureSession的文档中找到。

(2) 得到一个指向默认视频捕捉设备的指针。在几乎所有iOS系统中，AVCaptureDevice都会返回手机后置摄像头。

(3) 在将捕捉设备添加到AVCaptureSession前，首先需要将它封装成一个AVCaptureDeviceInput对象。尤其重要的是开发者需要传递一个有效的指向这个方法的NSError指针，用于捕捉尝试创建输入数据时出现的任何错误信息。

(4) 当返回一个有效AVCaptureDeviceInput时，首先希望调用会话的canAddInput:方法测试其是否可以被添加到会话中，如果可以，则调用addInput:方法将其添加到会话并给它传递捕捉设备的输入信息。

(5) 与查找设备获得默认视频捕捉设备类似，选择默认的音频捕捉设备也是同样的，即返回一个指向内置麦克风的指针。

(6) 为这个设备创建一个捕捉设备输入。同样，务必留意调用该方法时遇到的错误。

(7) 测试设备是否可以被添加到会话，如果可以，将其添加到捕捉会话。

(8) 创建一个AVCaptureStillImageOutput实例。它是AVCaptureOutput子类，用于从摄像头捕捉静态图片。可为对象的outputSettings属性配置一个字典来表示希望捕捉JPEG格式的图片。创建并配置完毕后，就可以调用addOutput:方法将它添加到捕捉会话。与刚才添加设备输入信息一样，开发者同样希望在此之前测试该输出是否可以被添加到捕捉会话。如果不经过测试盲目添加输出内容可能会抛出异常错误，导致应用程序崩溃。

(9) 创建一个新的AVCaptureMovieFileOutput实例。它是一个AVCaptureOutput子类，用于将QuickTime电影录制到文件系统。开发者也需要测试并将这个输出内容添加到会话。最后，返回YES，让调用函数知道会话配置已经完成了。

6.3.4 启动和停止会话

捕捉会话的对象图会通过调用setupSession:方法被妥善设置，不过在使用捕捉会话前，首先需要启动会话。启动会话第一步是启动数据流并使它处于准备捕捉图片和视频的状态，让我们看一下startSession方法和stopSession方法的实现，如代码清单6-5所示。

代码清单 6-5 启动和停止捕捉会话

```
- (void)startSession {
    if (![self.captureSession isRunning]) {                                    // 1
        dispatch_async(self.videoQueue, ^{
            [self.captureSession startRunning];
        });
    }
}

- (void)stopSession {
    if ([self.captureSession isRunning]) {                                     // 2
        dispatch_async(self.videoQueue, ^{
            [self.captureSession stopRunning];
        });
    }

}
```

(1) 检查并确保捕捉会话没有处于准备运行状态。如果没有准备好，则调用捕捉会话的startRunning方法，这是一个同步调用并会消耗一定时间，所以你要以异步方式在videoQueue排队调用该方法，这样就不会阻碍主线程。

(2) stopSession的实现基本相同。在捕捉会话上调用stopRunning方法会停止系统中的数据流，这也是一个同步调用，所以你要采用异步方式调用这个方法。

运行该应用程序，在启动时立即会出现一个或两个对话框，如图6-8所示。下面继续讨论为什么会显示这些对话框及如何对其进行处理。

图 6-8 捕捉设备隐私保护提示

6.3.5 处理隐私需求

在iOS 7和iOS 8版本中有关隐私保护的功能又得到了进一步的改进，让应用程序试图使用设备硬件的动作变得更加透明化。当应用程序试图访问麦克风和相机时就会弹出一个对话框询问用户是否授权。这一有关隐私保护的改进对于iOS平台非常受欢迎，只是我们在创建捕捉应用程序时可能要更加麻烦一点了。

> **注意：**
>
> iOS 7 版本中只有特地区有法律规定时才会询问用户是否可以访问设备的相机。从 iOS 8 开始，所有地区的用户都要在应用程序中取得授权才可以访问相机。

这一提醒的触发是通过创建AVCaptureDeviceInput实现的，当这些对话框出现时，系统并没有停止来等待用户响应，而是立即返回一个设备，比如调用音频设备则返回静音的设备，如果是相机则返回黑白帧。直到用户回答并同意了对话框中的内容，才会实际开始捕捉音频或视频内容。如果用户回答“不允许”，则在这个会话期间不会记录任何内容。如果用户不同意访问，下次应用程序启动时的AVCaptureDeviceInput创建情况就会返回nil，创建方法会收到NSError产生的错误代码AVErrorApplicationIsNotAuthorizedToUseDevice和失败原因，如下所示：

```
NSLocalizedFailureReason=This app is not authorized to use iPhone Microphone.
```

如果收到这个错误信息，唯一的办法就是弹出一个错误消息，如图6-9所示，告诉用户需要在“设置”应用程序中授权访问所需的硬件。

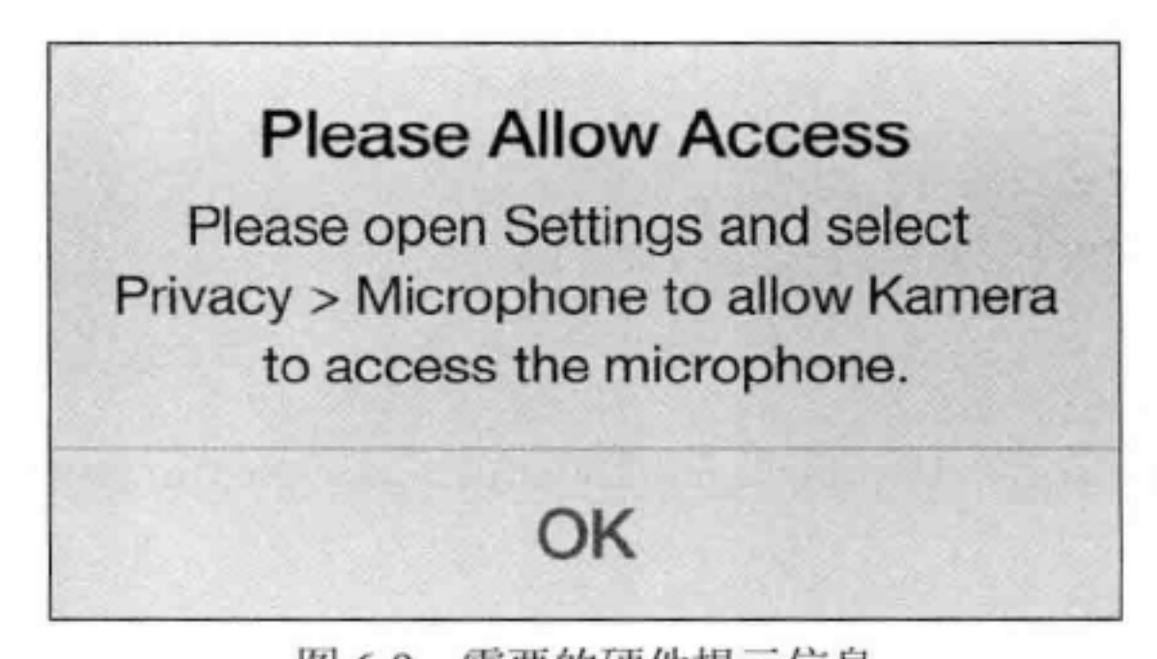

图 6-9 需要的硬件提示信息

注意：

用户可随时在“设置”应用程序中修改隐私设置，所以在创建 AVCaptureDeviceInput 对象时一定要检查任何返回的错误信息。

会话配置完成后，让我们继续开始学习Kamera应用程序具体功能的实现方法。

6.3.6 切换摄像头

基本上目前所有的iOS设备都具有前置和后置两个摄像头。Kamera应用程序会用到这两个摄像头，所以首先要开发的功能就是让用户能够在摄像头之间进行切换。让我们先从几个支撑方法开始学习，这些方法可以简化功能的实现，如代码清单6-6所示。

代码清单 6-6 摄像头的支撑方法

```
- (AVCaptureDevice *)cameraWithPosition:(AVCaptureDevicePosition)position {  // 1
    NSArray *devices = [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in devices) {
        if (device.position == position) {
            return device;
        }
    }
    return nil;
}

- (AVCaptureDevice *)activeCamera {                                          // 2
    return self.activeVideoInput.device;
}

- (AVCaptureDevice *)inactiveCamera {                                        // 3
    AVCaptureDevice *device = nil;
    if (self.cameraCount > 1) {
        if ([self activeCamera].position == AVCaptureDevicePositionBack) {
            device = [self cameraWithPosition:AVCaptureDevicePositionFront];
        } else {
            device = [self cameraWithPosition:AVCaptureDevicePositionBack];
        }
    }
    return device;
```

```
}

- (BOOL)canSwitchCameras {                                                    // 4
    return self.cameraCount > 1;
}

- (NSUInteger)cameraCount {                                                   // 5
    return [[AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo] count];

}
```

(1) cameraWithPosition:方法返回指定位置的AVCaptureDevice，有效的位置为AVCapturePositionFront或AVCapturePositionBack。遍历可用的视频设备并返回position参数对应的值。

(2) activeCamera方法返回当前捕捉会话对应的摄像头，返回激活的捕捉设备输入的device属性。

(3) inactiveCamera方法返回当前未激活的摄像头，通过查找当前激活摄像头的反向摄像头实现。如果应用程序运行的设备只有一个摄像头，则返回nil。

(4) canSwitchCameras方法返回一个BOOL值用于表示是否有超过一个摄像头可用。

(5) 最后，cameraCount会返回可用视频捕捉设备的数量。

实现了这些方法后，我们继续完成摄像头切换功能。切换前置和后置摄像头需要重新配置捕捉会话。幸运的是，可以动态重新配置AVCaptureSession，所以不必担心停止会话和重启会话带来的开销。不过我们对会话进行的任何改变，都要通过beginConfiguration和commitConfiguration方法进行单独的、原子性的变化。如代码清单6-7所示。

代码清单 6-7 切换摄像头

```
- (BOOL)switchCameras {

    if (![self canSwitchCameras]) {                                           // 1
        return NO;
    }

    NSError *error;
    AVCaptureDevice *videoDevice = [self inactiveCamera];                     // 2

    AVCaptureDeviceInput *videoInput =
        [AVCaptureDeviceInput deviceInputWithDevice:videoDevice error:&error];

    if (videoInput) {
        [self.captureSession beginConfiguration];                             // 3

        [self.captureSession removeInput:self.activeVideoInput];              // 4

        if ([self.captureSession canAddInput:videoInput]) {                   // 5
            [self.captureSession addInput:videoInput];
            self.activeVideoInput = videoInput;
        } else {
            [self.captureSession addInput:self.activeVideoInput];
        }
```

```
        [self.captureSession commitConfiguration];                          // 6
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];           // 7
        return NO;
    }
    return YES;
}
```

(1) 首先要确认是否可以切换摄像头。如果不可以，则返回NO，并退出方法。

(2) 接下来，获取指向未激活摄像头的指针并为它创建一个新的AVCaptureDeviceInput。

(3) 在会话中调用beginConfiguration，并标注原子配置变化的开始。

(4) 移除当前激活的AVCaptureDeviceInput。该当前视频捕捉设备输入信息必须在新的对象添加前移除。

(5) 执行标准测试来检查是否可以添加新的AVCaptureDeviceInput，如果可以，将它添加到会话并设置为activeVideoInput。为确保安全，如果新的输入不能被添加，需要重新添加之前的输入。

(6) 配置完成后，对AVCaptureSession调用commitConfiguration，会分批将所有变更整合在一起，得出一个有关会话的单独的、原子性的修改。

(7) 当创建新的AVCaptureDeviceInput时如果出现错误，需要通知委托来处理该错误。

再次运行该应用程序。假设iOS设备具有两个摄像头，点击屏幕右上角的摄像头图标就可以切换前置和后置摄像头了。

6.3.7 配置捕捉设备

AVCaptureDevice定义了很多方法让开发者控制iOS设备上的摄像头。尤其是可以独立调整和锁定摄像头的焦距、曝光和白平衡。对焦和曝光还可以基于特定的兴趣点进行设置，使其在应用程序中实现点击对焦和点击曝光的功能。AVCaptureDevice还可以让你控制设备的LED作为拍照的闪光灯或手电筒使用。

每当修改摄像头设备时，一定要先测试修改动作是否能够被设备支持。并不是所有摄像头(即使单摄像头的iOS设备)都能支持所有功能。比如，前置摄像头不支持对焦操作，因为它和目标之间不会超过一个臂长的距离；但是大部分的后置摄像头就可以支持全尺寸对焦。尝试应用一个不被支持的修改动作可能会带来异常，导致应用程序崩溃，所以在进行修改前对其进行测试非常有必要。比如，在将对焦模式设置为自动之前，首先就要检查这一模式是否被支持，如下面的代码所示：

```
AVCaptureDevice *device = // Active video capture device

if ([device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {
    // Perform configuration

}
```

当验证这一个配置的修改可以支持时，就可以执行实际的设备配置了。修改捕捉设备的

基本技巧包括先锁定设备准备配置，执行所需的修改，最后解锁设备。比如，在确定摄像头支持自定对焦模式后，按如下方式配置focusMode属性：

```
AVCaptureDevice *device = // Active video capture device
if ([device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {
    NSError *error;
    if ([device lockForConfiguration:&error]) {
        device.focusMode = AVCaptureFocusModeAutoFocus;
        [device unlockForConfiguration];
    } else {
        // handle error
    }
}
```

Mac、iPhone或iPad上的设备都是系统通用的，所以在进行设备修改前，AVCaptureDevice要求开发者获取设备上的一个排它锁，不这样做会导致应用程序抛出一个异常。虽然不要求配置完成后立即释放这个排它锁，不过如果没有释放则会对其他使用同一资源的应用程序产生副作用，所以大多数时候我们都要做平台的好市民，每当配置完成后就释放这个排它锁。

学习了上述知识后，让我们继续研究如何实现Kamera的各种设备配置功能。

6.3.8 调整焦距和曝光

iOS设备上的大多数后置摄像头都支持基于给定的兴趣点设置对焦和曝光数据。在界面上直观地利用这个功能是允许用户在摄像头界面点击一个位置，就会在这个点自动对焦或曝光。还可以对这些兴趣点锁定焦距和曝光，确保用户可以稳定地点击拍照按钮。我们首先实现点击对焦功能，如代码清单6-8所示。

代码清单 6-8 点击对焦方法的实现

```
- (BOOL)cameraSupportsTapToFocus {                                    // 1
    return [[self activeCamera] isFocusPointOfInterestSupported];
}

- (void)focusAtPoint:(CGPoint)point {                                 // 2

    AVCaptureDevice *device = [self activeCamera];

    if (device.isFocusPointOfInterestSupported &&                     // 3
        [device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) {                   // 4
            device.focusPointOfInterest = point;
            device.focusMode = AVCaptureFocusModeAutoFocus;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }

}
```

(1) 首先实现cameraSupportsTapToFocus方法。第一步询问激活中的摄像头是否支持兴趣点对焦。客户端利用这个方法按需更新用户界面。

(2) 传递一个CGPoint结构给focusAtPoint:方法。传进来的点已经从屏幕坐标转换为捕捉设备坐标，这一转换是在之前实现的THPreviewView类中进行的。

(3) 在得到激活摄像头设备的指针后，测试它是否支持兴趣点对焦并确认它是否支持自动对焦模式。这一模式会使用单独扫描的自动对焦，并将focusMode设置为AVCaptureFocusModeLocked。

(4) 锁定设备准备配置。如果获得了锁，将focusPointOfInterest属性设置为传进来的CGPoint值，设置对焦模式为AVCaptureFocusModeAutoFocus。最后调用AVCaptureDevice的unlockForConfiguration释放该锁定。

运行应用程序，找到一个你希望设置对焦和执行单击的特定对象。应用程序会在屏幕上显示一个蓝色的矩形区域，摄像头的焦点就在其中。现在焦点锁定了，移动摄像头找到一个新的兴趣点并点击确定新的对象。点击对焦功能完成后，让我们看点击曝光的实现。

大部分设备的默认曝光模式都是AVCaptureExposureModeContinuousAutoExposure，即根据场景的变化自动调整曝光度。我们可以用与点击对焦类似的方法实现“点击曝光并锁定”功能；不过这一过程有点难度。虽然AVCaptureExposureMode枚举定义了AVCaptureExposureModeAutoExpose值，不过当前的iOS设备都不支持这个值。这意味着开发者需要一点创造力以便采用点击对焦的同样方式实现这个功能。让我们看一下具体的实现过程，如代码清单6-9所示。

代码清单 6-9 “点击曝光”方法

```
- (BOOL)cameraSupportsTapToExpose {                                    // 1
    return [[self activeCamera] isExposurePointOfInterestSupported];
}

// Define KVO context pointer for observing 'adjustingExposure" device property.
static const NSString *THCameraAdjustingExposureContext;

- (void)exposeAtPoint:(CGPoint)point {

    AVCaptureDevice *device = [self activeCamera];

    AVCaptureExposureMode exposureMode =
        AVCaptureExposureModeContinuousAutoExposure;

    if (device.isExposurePointOfInterestSupported &&                   // 2
    [device isExposureModeSupported:exposureMode]) {

    NSError *error;
    if ([device lockForConfiguration:&error]) {                        // 3

        device.exposurePointOfInterest = point;
        device.exposureMode = exposureMode;

        if ([device isExposureModeSupported:AVCaptureExposureModeLocked]) {
            [device addObserver:self                                   // 4
                    forKeyPath:@"adjustingExposure"
```

```
                                options:NSKeyValueObservingOptionNew
                                context:&THCameraAdjustingExposureContext];
                }

                [device unlockForConfiguration];
            } else {
                [self.delegate deviceConfigurationFailedWithError:error];
            }
        }
    }

    - (void)observeValueForKeyPath:(NSString *)keyPath
                        ofObject:(id)object
                          change:(NSDictionary *)change
                         context:(void *)context {

        if (context == &THCameraAdjustingExposureContext) {                    // 5

            AVCaptureDevice *device = (AVCaptureDevice *)object;

            if (!device.isAdjustingExposure &&                                 // 6
                [device isExposureModeSupported:AVCaptureExposureModeLocked]) {

                [object removeObserver:self                                    // 7
                          forKeyPath:@"adjustingExposure"
                            context:&THCameraAdjustingExposureContext];

                dispatch_async(dispatch_get_main_queue(), ^{                   // 8
                    NSError *error;
                    if ([device lockForConfiguration:&error]) {
                        device.exposureMode = AVCaptureExposureModeLocked;
                        [device unlockForConfiguration];
                    } else {
                        [self.delegate deviceConfigurationFailedWithError:error];
                    }
                });
            }

        } else {
            [super observeValueForKeyPath:keyPath
                                ofObject:object
                                  change:change
                                 context:context];
        }

    }
```

(1) cameraSupportsTapToExpose方法的实现几乎与cameraSupportsTapToFocus的实现一样。即询问激活设备是否支持对一个兴趣点进行曝光。

(2) 执行设备配置测试来确保需要的配置是被支持的。本例中需要验证是否支持AVCaptureExposureModeContinuousAutoExposure模式。

(3) 锁定设备准备配置，设置exposurePointOfInterest和exposureMode属性为期望的值。

(4) 判断设备是否支持锁定曝光模式。如果支持，使用KVO来确定设备adjustingExposure属性的状态。观察该属性可以知道曝光调整何时完成，让我们有机会在该点上锁定曝光。

(5) 通过测试上下文参数是否为&THCameraAdjustingExposureContext指针来判断监听回调是否对应我们期望的变更操作。

(6) 判断设备是否不再调整曝光等级，确认设备的exposureMode是否可以设置为AVCaptureExposureModeLocked。如果可以，则继续下面的流程。

(7) 移除作为adjustingExposure属性监听器的self，这样就不会得到后续变更的通知了。

(8) 最后，以异步方式调度回主队列，定义一个块来设置exposureMode属性为AVCaptureExposureModeLocked。将exposureMode更改转移到下一个事件循环运行非常重要，这样上一步中的removeObserver:调用才有机会完成。

的确，这比点击对焦的实现要复杂一些；不过你会发现最终效果是一样的。再次运行应用程序，在预览视图中找一些较暗的区域双击。会看到屏幕上出现一个橙色的矩形框，曝光会锁定这个区域。在明亮区域指向摄像头(电脑屏幕可以很好地工作)，再次双击，会看到曝光调整为合理的程度。

添加了点击设置锁定焦点和曝光区域功能对Kamera应用程序来说增光不少。不过开发者还需要定义一种方法让用户可以切换回连续对焦和曝光模式。我们现在就来实现。代码清单6-10给出了resetFocusAndExposureModes方法的实现。

代码清单 6-10 重新设置对焦和曝光

```
- (void)resetFocusAndExposureModes {
    AVCaptureDevice *device = [self activeCamera];

    AVCaptureFocusMode focusMode = AVCaptureFocusModeContinuousAutoFocus;

    BOOL canResetFocus = [device isFocusPointOfInterestSupported] &&          // 1
                         [device isFocusModeSupported:focusMode];

    AVCaptureExposureMode exposureMode =
       AVCaptureExposureModeContinuousAutoExposure;

    BOOL canResetExposure = [device isExposurePointOfInterestSupported] &&    // 2
                            [device isExposureModeSupported:exposureMode];

    CGPoint centerPoint = CGPointMake(0.5f, 0.5f);                            // 3

    NSError *error;
    if ([device lockForConfiguration:&error]) {

        if (canResetFocus) {                                                  // 4
            device.focusMode = focusMode;
            device.focusPointOfInterest = centerPoint;
        }

        if (canResetExposure) {                                               // 5
            device.exposureMode = exposureMode;
            device.exposurePointOfInterest = centerPoint;
        }
```

```
        [device unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}
```

(1) 首先确认对焦兴趣点和连续自动对焦模式是否被支持。

(2) 执行类似的测试，确保曝光度可以通过相关的功能测试被重设。

(3) 创建一个CGPoint，x和y的值均为0.5f。回顾一下，捕捉设备空间的左上角为(0, 0)，右下角为(1, 1)。创建一个(0.5, 0.5)的CGPoint会从摄像头空间的中点开始扫描。

(4) 锁定设备准备配置，如果焦点可以重设，就进行期望的修改。

(5) 同样，如果曝光度可以重设，就设置为期望的曝光模式。

运行应用程序，执行一些焦点和曝光的调整。在预览视图中两个手指双击任何位置，会发现对焦和曝光被重新设置为连续模式。

6.3.9 调整闪光灯和手电筒模式

AVCaptureDevice类可以让开发者修改摄像头的闪光灯和手电筒模式。设备后面的LED灯当拍摄静态图片时作为闪光灯，而当拍摄视频时用作连续灯光(手电筒)。捕捉设备的flashMode和torchMode属性可以被设置为以下3个值中的一个：

- AVCapture(Torch|Flash)ModeOn：总是开启。
- AVCapture(Flash|Torch)ModeOff：总是关闭。
- AVCapture(Flash|Torch)ModeAuto：系统会基于周围环境光照情况自动关闭或打开LED。

代码清单6-11给出了闪光灯和手电筒模式方法的实现。

代码清单 6-11 闪光灯和手电筒方法

```
- (BOOL)cameraHasFlash {
    return [[self activeCamera] hasFlash];
}

- (AVCaptureFlashMode)flashMode {
    return [[self activeCamera] flashMode];
}

- (void)setFlashMode:(AVCaptureFlashMode)flashMode {

    AVCaptureDevice *device = [self activeCamera];

    if ([device isFlashModeSupported:flashMode]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) {
            device.flashMode = flashMode;
            [device unlockForConfiguration];
        } else {
```

```
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }
}

- (BOOL)cameraHasTorch {
    return [[self activeCamera] hasTorch];
}

- (AVCaptureTorchMode)torchMode {
    return [[self activeCamera] torchMode];
}

- (void)setTorchMode:(AVCaptureTorchMode)torchMode {

    AVCaptureDevice *device = [self activeCamera];

    if ([device isTorchModeSupported:torchMode]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) {
            device.torchMode = torchMode;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }

}
```

我们这里不对上述代码逐步讲解了，因为这段代码现在看起来应该非常熟悉了。与之前一样，在更改配置前，开发者需要确认我们尝试修改的功能是否被支持。

Kamera应用程序的主视图控制器会正确调用setFlashMode:或setTorchMode:方法，这取决于用户界面上Video/Photo模式选择器的位置。运行应用程序，点击屏幕左上角的闪电图标可在不同模式之间切换。

6.3.10 拍摄静态图片

在setupSession:方法的实现过程中，我们将一个AVCaptureStillImageOutput实例添加到捕捉会话。这个类是AVCaptureOutput的子类，用于捕捉静态图片。AVCaptureStillImageOutput类定义了captureStillImageAsynchronouslyFromConnection:completionHandler:方法来执行实际的拍摄。让我们看下面这个简单示例：

```
AVCaptureConnection *connection = // Active video capture connection
id completionHandler = ^(CMSampleBufferRef buffer, NSError *error){
    // Handle image capture
};
[imageOutput captureStillImageAsynchronouslyFromConnection:connection
                                    completionHandler:completionHandler];
```

这个方法包含了一些新的对象类型需要我们讨论；第一个就是AVCaptureConnection。当

创建一个会话并添加捕捉设备输入和捕捉输出时，会话自动建立输入和输出的连接，按需选择信号流线路。访问这些连接在一些情况下是非常实用的功能，因为可以让开发者更好地对发送到输出端的数据进行控制。另一个新的对象类型是CMSampleBuffer。CMSampleBuffer是由Core Media框架定义的一个Core Foundation对象。下一章会对这个对象类型进行详细讲述，不过现在我们只需要知道这个对象用来保存捕捉到的图片数据。由于我们在创建静态图片输出对象时指定了AVVideoCodecJPEG作为编解码的键，所以该对象包含的字节就会被压缩成JPEG格式。

让我们看一下Kamera应用程序拍摄静态图片时是如何使用这些对象的(如代码清单6-12所示)。

代码清单 6-12 捕捉静态图片

```
- (void)captureStillImage {
    AVCaptureConnection *connection =                                    // 1
        [self.imageOutput connectionWithMediaType:AVMediaTypeVideo];

    if (connection.isVideoOrientationSupported) {                        // 2
        connection.videoOrientation = [self currentVideoOrientation];
    }

    id handler = ^(CMSampleBufferRef sampleBuffer, NSError *error) {
        if (sampleBuffer != NULL) {

            NSData *imageData =                                          // 4
                [AVCaptureStillImageOutput
                    jpegStillImageNSDataRepresentation:sampleBuffer];

            UIImage *image = [[UIImage alloc] initWithData:imageData];   // 5

        } else {
            NSLog(@"NULL sampleBuffer: %@", [error localizedDescription]);
        }
    };
    // Capture still image                                               // 6
    [self.imageOutput captureStillImageAsynchronouslyFromConnection:connection
                                                completionHandler:handler];
}

- (AVCaptureVideoOrientation)currentVideoOrientation {

    AVCaptureVideoOrientation orientation;

    switch ([UIDevice currentDevice].orientation) {                      // 3
        case UIDeviceOrientationPortrait:
            orientation = AVCaptureVideoOrientationPortrait;
            break;
        case UIDeviceOrientationLandscapeRight:
            orientation = AVCaptureVideoOrientationLandscapeLeft;
            break;
        case UIDeviceOrientationPortraitUpsideDown:
            orientation = AVCaptureVideoOrientationPortraitUpsideDown;
```

```
        break;
    default:
        orientation = AVCaptureVideoOrientationLandscapeRight;
        break;
    }

    return orientation;

}
```

(1) 首先通过调用connectionWithMediaType:方法获取AVCaptureStillImageOutput对象使用的当前AVCaptureConnection的指针。当查找AVCaptureStillImageOutput连接时一般会传递AVMediaTypeVideo媒体类型。

(2) Kamera应用程序本身只支持垂直方向，所以当旋转设备时，用户界面保持不变。不过我们希望满足横向使用手机的用户需求，所以需要相应地调整结果图片的方向。要确定连接是否支持设置视频方向；如果是，则设置其为currentVideoOrientation方法返回的AVCaptureVideoOrientation。

(3) 取UIDevice的orientation，切换该值确定相应的AVCaptureVideoOrientation。重要的一点是注意左侧和右侧的AVCaptureVideoOrientation值是和它们的UIDeviceOrientation值相反的。

(4) 在completion handler块中，如果接收一个有效的CMSampleBuffer，则调用AVCaptureStillImageOutput类的jpegStillImageNSDataRepresentation方法，会返回一个表示图片字节的NSData。

(5) 用NSData创建一个新的UIImage实例。

现在我们已经成功捕捉到一个图片并创建了相应的UIImage。可以把它呈现在用户界面的相应位置或写入应用程序沙盒的指定位置。不过大部分用户都希望将通过Kamera应用程序拍摄的图片写入Photos应用程序中的Camera Roll中。要添加这个功能，需要用到Assets Library框架。

6.3.11 使用 Assets Library 框架

Assets Library框架可以让开发者通过编程方式访问iOS Photos应用程序所管理的用户相册和视频库。该框架在许多AV Foundation应用程序中都是默认添加的，所以掌握它的用法非常重要。

这个框架的核心类是ALAssetsLibrary。ALAssetsLibrary类的实例定义了与用户库进行交互的接口。该对象具有多个“写入”方法，可以让开发者这将照片或视频写入到自己的库中。应用程序第一次试图与库进行交互时，会弹出一个如图6-10所示的对话框。

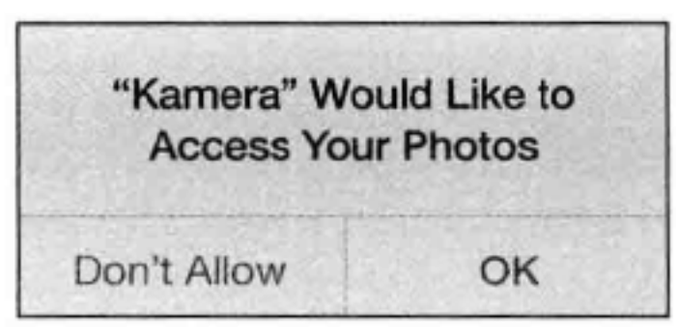

图 6-10 访问相片隐私提示框

在实际使用前，用户必须明确允许应用程序访问该库。如果用户不同意访问，则任何向库中的写入操作都会失败，并返回一个表示用户拒绝访问的错误信息。如果访问用户库是应用程序的核心功能，开发者首先就需要确定应用程序的授权状态，如下面的示例所示。

```
ALAuthorizationStatus status = [ALAssetsLibrary authorizationStatus];
if (status == ALAuthorizationStatusDenied) {
    // Show prompt indicating the application won't function
    // correctly without access to the library
} else {
    // Perform authorized access to the library
}
```

让我们看一下如何将捕捉到的图片写入Assets Library，如代码清单6-13所示。

代码清单 6-13 捕捉静态图片

```
- (void)captureStillImage {
    AVCaptureConnection *connection =
        [self.imageOutput connectionWithMediaType:AVMediaTypeVideo];

    if (connection.isVideoOrientationSupported) {
        connection.videoOrientation = [self currentVideoOrientation];
    }

    id handler = ^(CMSampleBufferRef sampleBuffer, NSError *error) {
        if (sampleBuffer != NULL) {

            NSData *imageData =
                [AVCaptureStillImageOutput
                    jpegStillImageNSDataRepresentation:sampleBuffer];

            UIImage *image = [[UIImage alloc] initWithData:imageData];
            [self writeImageToAssetsLibrary:image];                    // 1

        } else {
            NSLog(@"NULL sampleBuffer: %@", [error localizedDescription]);
        }
    };
    // Capture still image
    [self.imageOutput captureStillImageAsynchronouslyFromConnection:connection
                                                  completionHandler:handler];
}

- (void)writeImageToAssetsLibrary:(UIImage *)image {

    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];         // 2

    [library writeImageToSavedPhotosAlbum:image.CGImage                // 3
                        orientation:(NSInteger)image.imageOrientation  // 4
                    completionBlock:^(NSURL *assetURL, NSError *error) {
                        if (!error) {
                            [self postThumbnailNotification:image];    // 5
```

```
                    } else {
                        id message = [error localizedDescription];
                        NSLog(@"Error: %@", message);
                    }
                }];
}

- (void)postThumbnailNotification:(UIImage *)image {
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc postNotificationName:THThumbnailCreatedNotification object:image];
}
```

(1) 在捕捉completion handler中，调用新方法writeImageToAssetsLibrary:，并将用图片数据创建的UIImage对象传递给它。

(2) 创建一个新的ALAssetsLibrary实例，用于通过编程方式向用户的Camera Roll写入。

(3) 调用库的writeImageToSavedPhotosAlbum:orientation:completion-Block:方法。图片参数一定是CGImageRef，所以我们需要获取UIImage的CGImage表示。

(4) 方向参数是一个ALAssetOrientation枚举值。这些值直接对应于由图片的imageOrientation返回的UIImageOrientation值，所以将这个值转换为一个NSInterger。

(5) 如果写入成功，发送一个带有捕捉图片的通知。用于绘制Kamera应用程序用户界面中的缩略图。

运行应用程序，切换为Photos模式，拍摄几张图片。第一次拍照时会弹出一个提示框询问是否授权访问“照片库”，本例中一定要回答“同意”。在应用程序中点击拍照按钮左边的图片图标会得到一个用户Camera Roll的简单视图；不过要查看图片细节，还需要转到Photos应用程序。

拍摄静态图片的功能已经完成，让我们继续学习有关视频捕捉的功能。

6.3.12 视频捕捉

结束本章学习之前还有最后一件需要讨论的事，即视频内容的捕捉。当设置捕捉会话时，添加一个名为AVCaptureMovieFileOutput的输出。这个类定义了一个简单实用的方法将QuickTime影片捕捉到磁盘。这个类的大多数核心功能继承于超类AVCaptureFileOutput。这个抽象超类定义了许多实用功能，比如录制到最长时限或录制到特定文件大小时为止。还可以配置成保留最小可用磁盘空间，这一点在存储空间有限的移动设备上录制视频时非常重要。

通常当QuickTime影片准备发布时，影片头的元数据处于文件的开始位置。这样可以让视频播放器快速读取头包含的信息，来确定文件的内容、结构和其包含的多个样本的位置。不过当录制一个QuickTime影片时，直到所有的样本都完成捕捉后才能创建信息头。当录制结束时，创建头数据并将它附在文件结尾处，如图6-11所示。

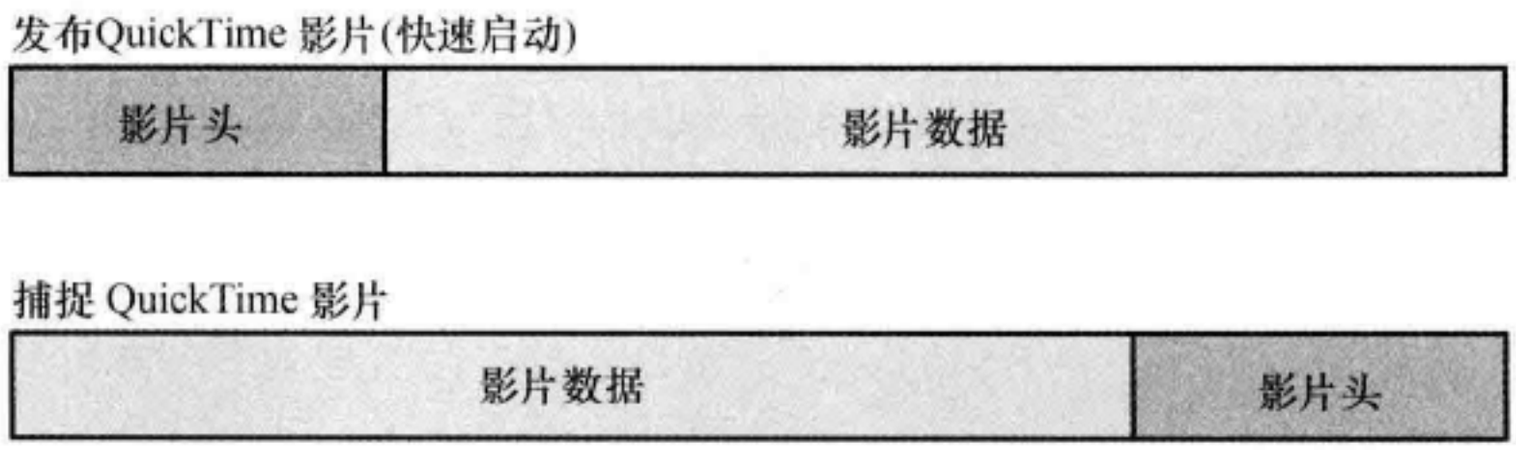

图 6-11 发布和捕捉 QuickTime 影片

将创建头的过程放在所有影片样本完成捕捉之后存在一个问题，尤其是在移动设备的情况下。如果遇到崩溃或其他中断，比如有电话拨入，则影片头就不会被正确写入，会在磁盘生成一个不可读的影片文件。AVCaptureMovieFileOutput提供的一个核心功能就是分段捕捉QuickTime影片，如图6-12所示。

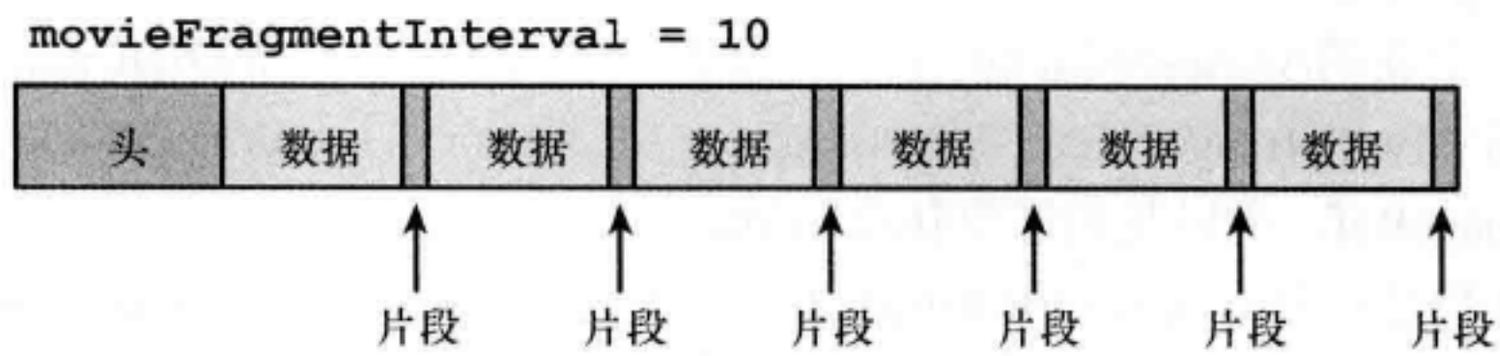

图 6-12 捕捉影片片段

当录制开始时，在文件最前面会写入一个最小化的头信息，随着录制的进行，片段按照一定的周期写入，创建完整的头信息。默认状态下，每10秒写入一个片段，不过这个时间间隔可以通过修改捕捉输出的movieFragmentInterval属性来改变。写入片段的方式可以逐步创建完整的QuickTime影片头，这样就确保了当遇到应用程序崩溃或中断时，影片仍然会以最后一个写入的片段为终点进行保存。默认的片段间隔足以满足Kamera应用程序的要求，不过可以在你自己的应用程序中修改这个值。

让我们从用于启动和停止录制的传输方法开始实现视频录制功能，如代码清单6-14所示。

代码清单 6-14 视频录制的传输方法

```
- (BOOL)isRecording {                                                    // 1
    return self.movieOutput.isRecording;
}

- (void)startRecording {

    if (![self isRecording]) {

        AVCaptureConnection *videoConnection =                           // 2
            [self.movieOutput connectionWithMediaType:AVMediaTypeVideo];

        if ([videoConnection isVideoOrientationSupported]) {             // 3
            videoConnection.videoOrientation = self.currentVideoOrientation;
        }

        if ([videoConnection isVideoStabilizationSupported]) {           // 4
            videoConnection.enablesVideoStabilizationWhenAvailable = YES;
        }
```

```
    AVCaptureDevice *device = [self activeCamera];

    if (device.isSmoothAutoFocusSupported) {                                    // 5
        NSError *error;
        if ([device lockForConfiguration:&error]) {
            device.smoothAutoFocusEnabled = YES;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }

    self.outputURL = [self uniqueURL];                                          // 6
    [self.movieOutput startRecordingToOutputFileURL:self.outputURL              // 8
                                     recordingDelegate:self];

}
}

- (NSURL *)uniqueURL {                                                          // 7

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *dirPath =
        [fileManager temporaryDirectoryWithTemplateString:@"kamera.XXXXXX"];

    if (dirPath) {
        NSString *filePath =
            [dirPath stringByAppendingPathComponent:@"kamera_movie.mov"];
        return [NSURL fileURLWithPath:filePath];
    }

    return nil;
}

- (void)stopRecording {                                                         // 9
    if ([self isRecording]) {
        [self.movieOutput stopRecording];
    }

}
```

(1) 此处提供了一个isRecording方法用于获取AVCaptureMovieFileOutput实例的状态。这是一个支撑方法，主要为外部客户端提供控制器的当前状态。

(2) 在startRecording方法中，获取处理当前视频捕捉连接的信息。用于对捕捉的视频数据配置一些核心属性。

(3) 判断是否支持设置videoOrientation属性；如果支持，将其设置为当前视频方向。设置视频的方向不会物理上旋转像素缓存，不过会应用QuickTime文件相应的矩阵变化。

(4) 如果可以设置enablesVideoStabilizationWhenAvailable，则设置它为YES。并不是所有摄像头和设备都支持该功能，所以需要对其进行测试。支持视频稳定可以显著提升捕捉到的视频质量，尤其是在iPhone这样的设备上。其中一点需要重点关注的是视频稳定只在录制视频文件时才会涉及。这一稳定效果也不会在视频预览屏幕中看到。

(5) 摄像头可以进行平滑对焦模式的操作，即减慢摄像头镜头对焦的速度。通常情况下，当用户移动拍摄时摄像头会尝试快速自动对焦，这会在捕捉的视频中出现脉冲式效果。当平滑对焦时，会降低对焦操作的速率，从而提供更加自然的视频录制效果。

(6) 查找写入捕捉视频的唯一文件系统URL。开发者需要保持对地址的强引用，因为这个地址在后面处理视频时会用到。

(7) 唯一URL查找代码看起来眼熟，因为我们之前就使用过这个方法。这个方法用到了一个我们添加到NSFileManager的分类方法，叫temporaryDirectoryWithTemplateString。它可以为文件将要写入的目的地创建一个唯一命名的目录。

(8) 最后在捕捉输出上调用startRecordingToOutputFileURL:recordingDelegate:方法，传递outputURL并将self设置为委托。这样就实际开始录制了。

(9) 添加一个方法用于停止录制，可以对捕捉输出调用stopRecording方法。

如果现在编译项目，会注意到编辑器有警告，意思是我们指定的录制委托self没有遵循AVCaptureFileOutputRecordingDelegate协议。我们需要修改一下控制器类的扩展，如代码清单6-15所示。

代码清单 6-15 遵循 AVCaptureFileOutputRecordingDelegate 协议

```
@interface THCameraController () <AVCaptureFileOutputRecordingDelegate>

@property (strong, nonatomic) AVCaptureSession *captureSession;
@property (weak, nonatomic) AVCaptureDeviceInput *activeVideoInput;

@property (strong, nonatomic) AVCaptureStillImageOutput *imageOutput;
@property (strong, nonatomic) AVCaptureMovieFileOutput *movieOutput;
@property (strong, nonatomic) NSURL *outputURL;

@end
```

遵循这些协议意味着还需要实现其中一个必要的方法。该方法用于获取最终文件并将它写入到Camera Roll中。代码清单6-16给出了这些方法的实现。

代码清单 6-16 写入捕捉到的视频

```
- (void)captureOutput:(AVCaptureFileOutput *)captureOutput
didFinishRecordingToOutputFileAtURL:(NSURL *)outputFileURL
     fromConnections:(NSArray *)connections
               error:(NSError *)error {
   if (error) {                                                          // 1
      [self.delegate mediaCaptureFailedWithError:error];
   } else {
      [self writeVideoToAssetsLibrary:[self.outputURL copy]];
   }
   self.outputURL = nil;
}

- (void)writeVideoToAssetsLibrary:(NSURL *)videoURL {

   ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];            // 2

   if ([library videoAtPathIsCompatibleWithSavedPhotosAlbum:videoURL]) {  // 3
```

```
    ALAssetsLibraryWriteVideoCompletionBlock completionBlock;

    completionBlock = ^(NSURL *assetURL, NSError *error){                    // 4
        if (error) {
            [self.delegate assetLibraryWriteFailedWithError:error];
        } else {
            [self generateThumbnailForVideoAtURL:videoURL];
        }
    };

    [library writeVideoAtPathToSavedPhotosAlbum:videoURL                     // 8
                                  completionBlock:completionBlock];
    }
}

- (void)generateThumbnailForVideoAtURL:(NSURL *)videoURL {

    dispatch_async(self.videoQueue, ^{

        AVAsset *asset = [AVAsset assetWithURL:videoURL];

        AVAssetImageGenerator *imageGenerator =                              // 5
            [AVAssetImageGenerator assetImageGeneratorWithAsset:asset];
        imageGenerator.maximumSize = CGSizeMake(100.0f, 0.0f);
        imageGenerator.appliesPreferredTrackTransform = YES;

        CGImageRef imageRef = [imageGenerator copyCGImageAtTime:kCMTimeZero      // 6
                                                      actualTime:NULL
                                                           error:nil];
        UIImage *image = [UIImage imageWithCGImage:imageRef];
        CGImageRelease(imageRef);

        dispatch_async(dispatch_get_main_queue(), ^{                         // 7
            [self postThumbnailNotification:image];
        });
    });

}
```

(1) 在委托回调中，如果返回错误信息则只需通知委托，让委托对错误进行处理。如果没有遇到错误，则尝试将视频写入用户的Camera Roll，这通过调用writeVideoToAssetsLibrary:方法来实现。

(2) 创建一个ALAssetsLibrary实例，它会提供写入视频的接口。

(3) 向资源库写入前，应调用videoAtPathIsCompatibleWithSavedPhotosAlbum:方法检查视频是否可被写入。本例中这个方法会返回YES，不过开发者要养成在写入资源库时调用该方法的习惯。

(4) 创建一个completion handler块，当向资源库的写入完成时调用该块。如果操作中出现错误，调用委托来通知错误信息；否则就是写入操作成功，所以接下来生成用于界面展示的视频缩略图。

(5) 在videoQueue上，为新建的视频建立一个新的AVAsset和AVAssetImageGenerator。设

置maximumSize属性为宽100和高0，根据视频的宽高比来设置图片高度。还需要设置appliesPreferredTrackTransform为YES，这样当捕捉缩略图时会考虑视频的变化(如视频方向的变化)。如果不这样设置，则会出现错误的缩略图方向。

(6) 因为只需要捕捉一张图片，所以可以使用copyCGImageAtTime:actualTime:error:方法。这是一个同步方法，这就是为什么我们将这个操作移到主线程之外。根据返回的CGImageRef数据，创建一个用户界面中展示用的UIImage。当调用copyCGImageAtTime:actualTime:error:方法时，开发者需要负责释放创建的图片，所以需要调用CGImageRelease(imageRef)方法避免出现内存泄漏。

(7) 调度回主线程，并发送一个通知传递最新创建的UIImage。

(8) 执行实际写入资源库的动作，传递videoURL和completionBlock。

运行应用程序，录制一段几秒钟的视频。点击Stop会生成一个为刚才拍摄的场景创建的新缩略图。可以点击这个缩略图打开浏览框并查看最新创建的视频。

6.4 小结

现在你应该已经很好地掌握了核心AV Foundation捕捉API的知识。我们学习如何配置和控制一个AVCaptureSession，通过一些示例学习了如何直接控制和操作捕捉设备，还学习了如何利用AVCaptureOutput的子类来捕捉静态图片和视频。使用这些核心功能，创建了一个与苹果内置的“相机”应用程序的功能类似的一个应用程序。虽然本章的讲解都是针对iOS平台的，不过我们用到方法和技术在Mac平台创建摄像头和视频应用程序时同样适用。我们已经对AV Foundation捕捉API的学习有了一个好的开端，在下一章中我们会继续学习一些更高级的捕捉特性，将摄像头和视频应用程序提升到一个新高度。

6.5 挑战

在Mac平台上创建一个简单的摄像头应用程序。几乎所有的Mac机器都有内置摄像头，现在需要的就是用捕捉代码将它们唤醒。我们为THCameraController对象撰写的大部分代码都可以在Mac上使用，所以就从这个类开始并按照实际需求进行修改吧。

第7章

高级捕捉功能

在前一章中，第一次见到了AV Foundation框架的捕捉类并学习了如何使用这些类完成相应的功能。精通这些核心功能非常重要，因为它们是所有的照片或视频捕捉应用程序的基础。不过AV Foundation的本领远不止这些，最近发布的iOS版本引入了一些强大的新功能。本章将在基础知识之上深入学习AV Foundation的高级捕捉功能。还等什么，现在就开始吧！

7.1 视频缩放

在iOS 7之前的版本中，苹果公司通过AVCaptureConnection的videoScaleAndCropFactor属性对摄像头缩放进行了有限制的支持。开发者可以调整连接缩放的值从默认的1.0增加到videoMaxScaleAndCropFactor属性定义的最大值。这一功能在一些特定的场景非常实用，不过带有一些明显的限制。第一个就是videoScaleAndCropFactor是一个连接级的设置，意思就是要想在用户界面正确获取它的状态值，开发者必须对AVCaptureVideoPreviewLayer应用一个正确的缩放变换。这是一个额外的工作量，并且当应用高缩放因子进行变换时会导致预览层的图片质量降低。第二个也是最大的限制就是这个属性只能在AVCaptureStillImageOutput连接中设置，这就让视频捕捉类应用程序无法利用这一特性。幸运的是，iOS 7版本为缩放功能给出了一个更好的方法。这个方法可以让你直接对AVCaptureDevice对象使用缩放参数，这意味着所有会话的输出，包括预览层，都会自动响应这一设置的状态。让我们现在就看一下如何使用这个新功能吧。

AVCaptureDevice提供了名为videoZoomFactor的属性，用于控制捕捉设备的缩放等级。这个属性的最小值为1.0，即不能进行缩放的图片。最大值由捕捉设备的activeFormat值确定，它是新类型AVCaptureDeviceFormat的一个实例。这个类定义了活动捕捉格式功能的细节，其中就包括videoMaxZoomFactor。

设备执行缩放效果是通过居中裁剪由摄像头传感器捕捉到的图片实现。当设置了一个低缩放因子时，一般低于1.5，图片等于或大于输出尺寸，这就在不用放大的情况下进行了一个适度的缩放，图片质量得以全部保存。在哪个点开始放大图像需要由AVCaptureDeviceFormat的videoZoomFactorUpscaleThreshold值确定。

让我们看一下示例代码来学习如何使用这个功能。在Chapter 7目录下可以找到一个名为ZoomKamera_Starter的示例项目，如图7-1所示。

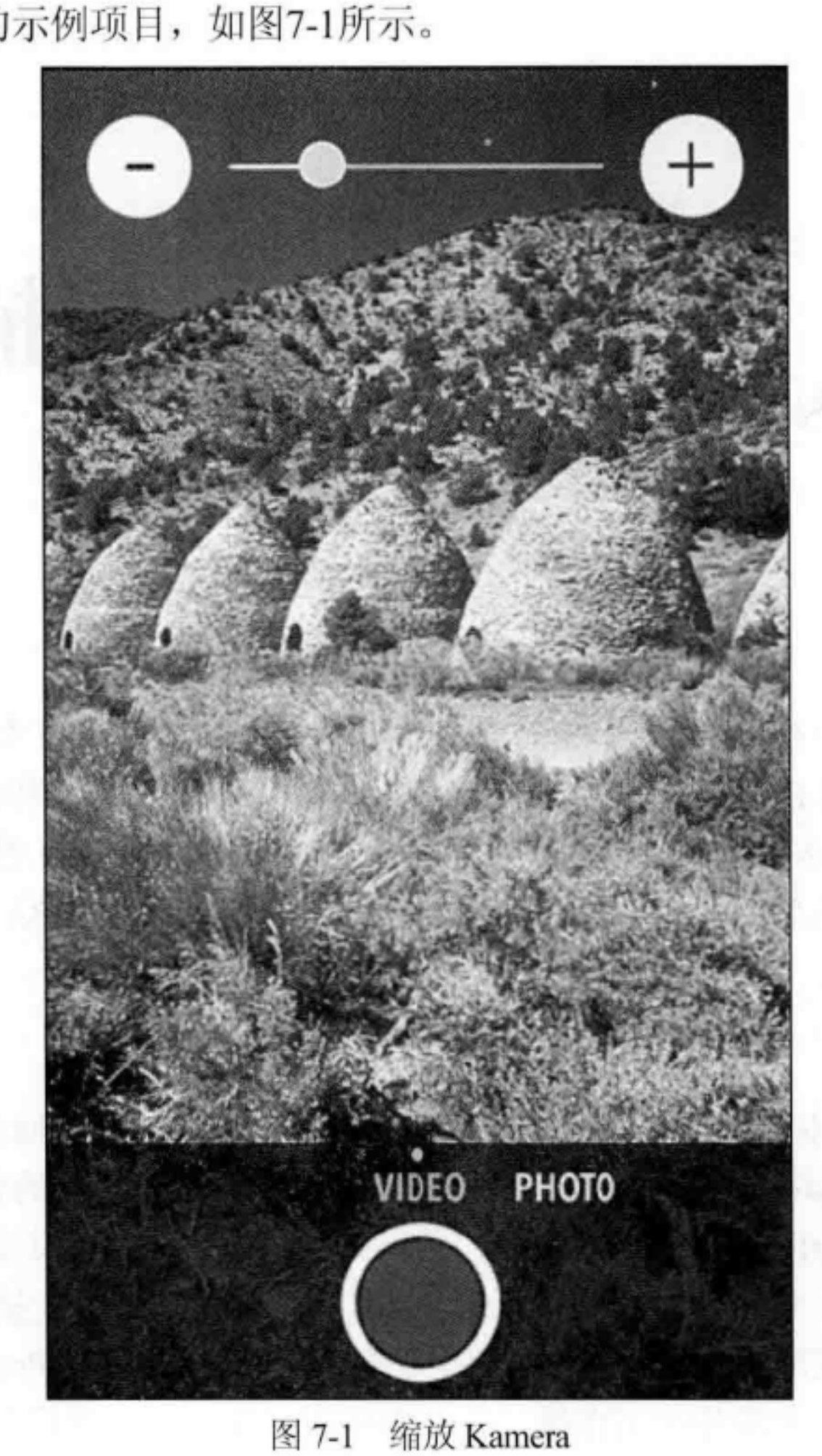

图 7-1 缩放 Kamera

注意：

本章中的示例项目用到一个基类 THBaseCameraController。这个类本质上就是第 6 章中创建的摄像头控制器，只不过增加了一些扩展点。尤其是它定义了一些可以重写的方法来配置会话输入、输出和会话预设。这样就能把注意力集中在本章需要介绍的新功能上，而避免被前一章内容的干扰。

在启动项目中，可以看到一个THCameraController类的存根实现。这个类扩展了THBaseCameraController并添加了视频缩放功能。代码清单7-1是这个类的接口。

代码清单 7-1 THCameraController 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THCameraZoomingDelegate <NSObject>                    // 1
```

```
- (void)rampedZoomToValue:(CGFloat)value;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id<THCameraZoomingDelegate> zoomingDelegate;

- (BOOL)cameraSupportsZoom;                                          // 2

- (void)setZoomValue:(CGFloat)zoomValue;
- (void)rampZoomToValue:(CGFloat)zoomValue;
- (void)cancelZoom;

@end
```

(1) THCameraZoomingDelegate提供了一个rampedZoomToValue:方法，用于在用户界面保持缩放滑动条控件与当前缩放状态的同步。

(2) 由于并不是所有的硬件都支持这个功能，所以需要添加一个cameraSupportsZoom方法，这样客户端可以进行查询，以确定是否展示缩放控件。

让我们转到类的实现中，开始学习如何实现具体的缩放功能，如代码清单7-2所示。

代码清单 7-2　THCameraController 的实现

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

const CGFloat THZoomRate = 1.0f;

// KVO Contexts
static const NSString *THRampingVideoZoomContext;
static const NSString *THRampingVideoZoomFactorContext;

@implementation THCameraController

- (BOOL)cameraSupportsZoom {
    return self.activeCamera.activeFormat.videoMaxZoomFactor > 1.0f;        // 1
}

- (CGFloat)maxZoomFactor {
    return MIN(self.activeCamera.activeFormat.videoMaxZoomFactor, 4.0f);    // 2
}

- (void)setZoomValue:(CGFloat)zoomValue {                                   // 3
    if (!self.activeCamera.isRampingVideoZoom) {

        NSError *error;
        if ([self.activeCamera lockForConfiguration:&error]) {              // 4

            // Provide linear feel to zoom slider
            CGFloat zoomFactor = pow([self maxZoomFactor], zoomValue);      // 5
            self.activeCamera.videoZoomFactor = zoomFactor;

            [self.activeCamera unlockForConfiguration];                     // 6

        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
```

```
        }
    }
}

...

@end
```

(1) 首先是cameraSupportsZoom方法的实现。通过当前选中的AVCaptureDevice (activeCamera)获取它的活动AVCaptureDeviceFormat。如果格式的videoMaxZoomFactor值大于1.0，则捕捉设备支持缩放功能。

(2) 要确定最大允许缩放因子，找到活动格式videoMaxZoomFactor的MIN值和4.0f。4.0f这个值是随意的，不过需要定义一个合理的缩放范围。通常我们不希望将内容缩放至videoMaxZoomFactor大小，因为太大就不实用了，不过在计算最大值时要时刻留意该值。如果试图设置缩放因子超过允许的最大值，就会出现异常。

(3) 当用户移动图7-1中的缩放条时，就会调用setZoomValue:方法。滑动条的范围从0.0f到1.0f。

(4) 如果捕捉设备当前没有将视频缩放，需要锁定设备进行配置。这一配置与其他的类似，如果在修改配置前没有进行锁定则会抛出一个异常。

(5) 应用程序提供的缩放范围是1x到4x。这一增长是指数形式的，所以要提供范围线性增长的感觉，需要通过计算最大缩放因子的zoomValue次幂(0到1)来计算zoomFactor值。计算后，设置捕捉设备的videoZoomFactor属性的值。

(6) 当完成配置后，结束设备锁定。

运行应用程序并移动滑动条，可以看到缩放级别不断在捕捉照片和视频的预览层中反映出来。随意调整maxZoomFactor方法的返回上限来查看设置更高缩放级别的效果。

videoZoomFactor属性会立即调整缩放级别。这对一些连续类型的控制比较实用，比如滑动条；不过，我们还希望在一个时间段内将缩放数值从一个值逐渐调整为另一个值。如果希望创建一个飞梭摇杆类型的控制，这个方法就非常适合。让我们为这些按钮添加一些功能来实现按时间调整缩放级别，如代码清单7-3所示。

代码清单 7-3 Zoom Ramping

```
- (void)rampZoomToValue:(CGFloat)zoomValue {                          // 1
    CGFloat zoomFactor = pow([self maxZoomFactor], zoomValue);
    NSError *error;
    if ([self.activeCamera lockForConfiguration:&error]) {
        [self.activeCamera rampToVideoZoomFactor:zoomFactor            // 2
                                        withRate:THZoomRate];
        [self.activeCamera unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}

- (void)cancelZoom {                                                  // 3
    NSError *error;
```

```
    if ([self.activeCamera lockForConfiguration:&error]) {
        [self.activeCamera cancelVideoZoomRamp];
        [self.activeCamera unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }

}
```

(1) 在用户界面中按下加号或减号按钮就会调用rampZoomToValue:方法。如果用户按下减号(-)按钮，传递到方法中的值为0.0f，如果用户按下加号(+)按钮，值则为1.0f。就像之前做的那样，基于zoomValue参数计算一个正确的zoomFactor值。

(2) 调用rampToVideoZoomFactor:withRate:方法并传递计算的zoomFactor和常量THZoomRate，即等于1.0f。这个方法的效果是每秒增加缩放因子一倍。相应的速率值如何设置取决于开发者自己，不过通常都是在1.0f到3.0f范围之内，可以提供一个比较舒服的缩放控制效果。

(3) 当用户在按钮上发起一个touch up inside动作，会调用cancelZoom方法。这会取消当前缩放进程并设置zoomFactor作为当前的状态。与所有设备配置变化一样，都需要先锁定设备，再应用期望的变更，完成后解除锁定。

再次运行应用程序并尝试这一新功能。会发现加号和减号按钮已经加入了期望的功能并平滑地对视频进行缩放；不过注意一个问题，即用户界面中的缩放条没有与当前缩放级别同步变化。让我们按照代码清单7-4解决这个问题。

代码清单 7-4　添加缩放状态监听器

```
- (BOOL)setupSessionInputs:(NSError **)error {
    BOOL success = [super setupSessionInputs:error];                    // 1
    if (success) {
        [self.activeCamera addObserver:self
                          forKeyPath:@"videoZoomFactor"
                             options:0
                             context:&THRampingVideoZoomFactorContext];
        [self.activeCamera addObserver:self
                          forKeyPath:@"rampingVideoZoom"
                             options:0
                             context:&THRampingVideoZoomContext];

    }
    return success;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {

    if (context == &THRampingVideoZoomContext) {
        [self updateZoomingDelegate];                                   // 2
    } else if (context == &THRampingVideoZoomFactorContext) {
```

```
        if (self.activeCamera.isRampingVideoZoom) {
            [self updateZoomingDelegate];                                   // 3
        }
    } else {
        [super observeValueForKeyPath:keyPath
                              ofObject:object
                                change:change
                               context:context];
    }
}

- (void)updateZoomingDelegate {
    CGFloat curZoomFactor = self.activeCamera.videoZoomFactor;
    CGFloat maxZoomFactor = [self maxZoomFactor];
    CGFloat value = log(curZoomFactor) / log(maxZoomFactor);            // 4
    [self.zoomingDelegate rampedZoomToValue:value];                     // 5

}
```

(1) 重写超类的setupSessionInputs:方法的实现，这样就可以在捕捉会话配置好之后访问activeCamera。添加self作为捕捉设备videoZoomFactor和rampingVideoZoom两个关键路径的监听器。

(2) 如果上下文为&THRampingVideoZoomContext，则调用updateZoomingDelegate方法。当缩放开始时监听方法开始出现，当缩放结束时再次出现。

(3) 如果上下文为&THRampingVideoZoomFactorContext并且有正在运行的视频缩放动作，则再次调用updateZoomingDelegate方法。

(4) 将当前缩放级别转换为滑动条的0到1比例，对当前videoZoomFactor值取对数，除以maxZoomFactor值取对数。

(5) 最后将结果赋给zoomingDelegate，这样用户界面就可以相应地进行更新了。

现在再次运行该应用程序，可以看到当与飞梭摇杆控件进行交互时，滑动条的值也相应地更新了。

AV Foundation新的缩放功能是平台的一大补充，对使用捕捉连接的videoScaleAndCropFactor提供了极大帮助。直接对AVCaptureDevice设置videoZoomFactor说明现在可以使用这个缩放来控制一切了。

7.2 人脸检测

如果你用过iOS内置的相机应用，会注意到这样一个功能，视图中有新的人脸进入时会自动建立相应的焦点。一个黄色的矩形框会显示在新检测到的人脸位置，并以矩形的中点完成自动对焦。这是一个有用的功能，因为它不需要手动操作就可以简捷地完成对焦拍摄。幸运的是，使用AV Foundation的实时人脸检测功能可在应用程序中实现相同的功能。

苹果公司首次向iOS开发者提供人脸检测功能是在Core Image框架中给出的。Core Image框架定义了CIDetector和CIFaceFeature两个对象，它们使用起来非常简单但提供了非常强大的人脸检测功能。不过这些方法并没有针对实时性进行优化，导致在现代摄像头和视频应用程

序要求的帧率之下很难应用。在iOS 6版本中，一个新的硬件加速特性被直接加入到AV Foundation中，这次支持对10个人脸进行实时检测。通过一个特定的AVCaptureOutput类型AVCaptureMetadataOutput可以实现上述功能。它的输出同之前见过的类似，不过不同于输出一个静态图片或QuickTime影片，它输出的是元数据。这个元数据来自于一个AVMetadata-Object抽象类的形式，该类定义了用来处理多种元数据类型的接口。当使用人脸检测时，会输出一个具体的子类类型AVMetadataFaceObject。

AVMetadataFaceObject实例定义了多个用于描述被检测到人脸的属性，最重要的一个属性就是人脸的边界(bounds)。它是一个设备标量坐标格式的CGRect。第6章中介绍过设备标量坐标系的范围是摄像头原始朝向从左上角的(0,0)到右下角的(1,1)。

除了边界(bounds)，AVMetadataFaceObject示例还给出了用于定义检测人脸斜倾角和偏转角的参数。斜倾角(roll angle)表示人的头部向肩膀方向的侧倾角度，偏转角(yaw angle)表示人脸绕y轴旋转的角度。

让我们创建一个应用程序来看具体是如何使用这些功能的。可在Chapter 7目录中找到一个名为FaceKamera_Starter的示例项目。我们从创建应用程序的摄像头控制器类开始，如代码清单7-5所示。

代码清单 7-5　THCameraController 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THFaceDetectionDelegate <NSObject>
- (void)didDetectFaces:(NSArray *)faces;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id <THFaceDetectionDelegate> faceDetectionDelegate;

@end
```

与本章中的其他类一样，该类扩展于THBaseCameraController，THBaseCameraController定义了核心的摄像头操作方法。我们在该接口中定义一个新的协议THFaceDetectionDelegate，并在THCameraController类中为它定义一个属性。这一委托会在摄像头捕捉新的元数据时得到通知。转到类的具体实现代码中，开始具体添加这些功能，如代码清单7-6所示。

代码清单 7-6　THCameraController 的实现

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController ()
@property (strong, nonatomic) AVCaptureMetadataOutput *metadataOutput;          // 1
@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    self.metadataOutput = [[AVCaptureMetadataOutput alloc] init];               // 2
```

```
    if ([self.captureSession canAddOutput:self.metadataOutput]) {
        [self.captureSession addOutput:self.metadataOutput];

        NSArray *metadataObjectTypes = @[AVMetadataObjectTypeFace];                    // 3
        self.metadataOutput.metadataObjectTypes = metadataObjectTypes;

        dispatch_queue_t mainQueue = dispatch_get_main_queue();
        [self.metadataOutput setMetadataObjectsDelegate:self                            // 4
                                                  queue:mainQueue];

        return YES;

    } else {                                                                            // 5
        if (error) {
            NSDictionary *userInfo = @{NSLocalizedDescriptionKey:
                                       @"Failed to still image output."};
            *error = [NSError errorWithDomain:THCameraErrorDomain
                                         code:THCameraErrorFailedToAddOutput
                                     userInfo:userInfo];
        }
        return NO;
    }
}

@end
```

(1) 首先在类扩展中创建一个属性用来保存这个类使用的AVCaptureMetadataOutput实例。

(2) 完全重写setupSessionOutputs:方法，因为超类中关于这个方法的实现在我们的示例中是不需要的。取而代之的是创建一个新的AVCaptureMetadataOutput示例并将它添加为捕捉会话的输出。

(3) 当配置AVCaptureMetadataOutput对象时，重要的一点是要通过设置metadataObjectTypes属性指定对象输出的元数据类型。限制检测到的元数据类型集合的做法是一种优化处理方法，可以减少我们实际感兴趣的对象数量。AV Foundation支持多种元数据类型，但是由于当前我们只对人脸元数据感兴趣，所以定义一个NSArray用于保存单独的带有AVMetadataObjectTypeFace常量值的元素。

(4) 当有新的元数据被检测到时，AVCaptureMetadataOutput需要有一个委托对象可以被调用。我们标记了self作为委托并在可能出现的回调中定义了一系列调度队列。也可以使用自定义系列的调度队列，不过由于人脸检测用到硬件加速，而且许多重要任务都在主线程中执行，所以需要为这个参数指定主队列。

(5) 最后，如果输出没有被添加，需要使用适当的NSError填充NSError并返回NO。

如果现在编译项目，会看到编译器告警显示self不是一个有效的委托。这是因为它未遵循AVCaptureMetadataOutputObjectsDelegate协议。在新的元数据被检测并通知控制器之前，需要应用这个协议并实现captureOutput:didOutputMetadataObjects:fromConnection:方法。让我们添加一些内容到THCameraController类中来解决这一问题，如代码清单7-7所示。

代码清单 7-7　接受 AVCaptureMetadataOutputObjectsDelegate 协议

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController ()<AVCaptureMetadataOutputObjectsDelegate>     // 1
...
@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {
...
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
      fromConnection:(AVCaptureConnection *)connection {

   for (AVMetadataFaceObject *face in metadataObjects) {                    // 2
      NSLog(@"Face detected with ID: %li", (long)face.faceID);
      NSLog(@"Face bounds: %@", NSStringFromCGRect(face.bounds));
   }

   [self.faceDetectionDelegate didDetectFaces:metadataObjects];             // 3

}

@end
```

(1) 令类扩展接受AVCaptureMetadataOutputObjectsDelegate协议。

(2) 在协议的方法实现中，添加一些临时的调试代码来检查检测是否如期出现。遍历每一个检测到的人脸，for循环输出它们的faceID和bounds。当确定配置完成后就可以移除这些调试代码了。

(3) 最后调用委托对象的didDetectFaces:方法，传递给它由AVCaptureMetadataOutput示例输出的AVMetadataObject实例数组。

运行该应用程序。当有人脸进入到摄像头区域时，你会看到调试信息被发送到控制台。记住，边界数据是以设备坐标系给出的。稍后将讨论如何将这些数据转换为更有用的坐标空间。

创建人脸检测委托

本书中的大部分示例代码都遵循这样的原则，即尽量不对用户界面层做过多干涉。不过在添加人脸检测功能时我们需要使用一些UIKit和Core Animation代码，目的是为了更好地了解元数据的用法。本应用程序中实现了THFaceDetectionDelegate协议的对象是THPlayerView类。让我们看一下代码清单7-8给出的接口。

代码清单 7-8　THPreviewView 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THFaceDetectionDelegate.h"

@interface THPreviewView : UIView <THFaceDetectionDelegate>
```

```
@property (strong, nonatomic) AVCaptureSession *session;

@end
```

这个类的接口很简单，一个有趣的地方是它实现了THFaceDetectionDelegate协议。这个类将成为元数据的目标并用于提供可视化呈现方式。

转到具体实现开始创建这个类，视图实现包含了大量代码，所以我们先从基础类结构开始分块创建它，如代码清单7-9所示。

代码清单 7-9 THPreviewView 的实现

```
#import "THPreviewView.h"

@interface THPreviewView ()                                                    // 1
@property (strong, nonatomic) CALayer *overlayLayer;
@property (strong, nonatomic) NSMutableDictionary *faceLayers;
@property (nonatomic, readonly) AVCaptureVideoPreviewLayer *previewLayer;
@end

@implementation THPreviewView

+ (Class)layerClass {                                                          // 2
    return [AVCaptureVideoPreviewLayer class];
}

- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setupView];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        [self setupView];
    }
    return self;
}

- (void)setupView {
    //
}

- (AVCaptureSession*)session {
    return self.previewLayer.session;
}

- (void)setSession:(AVCaptureSession *)session {                              // 3
    self.previewLayer.session = session;
}

- (AVCaptureVideoPreviewLayer *)previewLayer {
```

```
    return (AVCaptureVideoPreviewLayer *)self.layer;
}

- (void)didDetectFaces:(NSArray *)faces {
    //
}

@end
```

(1) 创建一个类扩展并定义类所使用的内部属性。

(2) 同上一章一样，重写layerClass方法并返回一个AVCaptureVideoPreviewLayer实例。这一技巧会在新实例创建时自动将AVCaptureVideoPreviewLayer设为该类的备用层。

(3) 要建立AVCaptureSession和AVCapturePreviewLayer之间的关联，需要重写setSession:方法并设置AVCaptureSession实例为预览层的session属性。

创建好基本的类结构之后，继续完成setupView方法的实现，如代码清单7-10所示。

代码清单 7-10　setupView 方法的实现

```
@implementation THPreviewView

...

- (void)setupView {
    self.faceLayers = [NSMutableDictionary dictionary];                    // 1
    self.previewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;

    self.overlayLayer = [CALayer layer];                                   // 2
    self.overlayLayer.frame = self.bounds;
    self.overlayLayer.sublayerTransform = THMakePerspectiveTransform(1000);
    [self.previewLayer addSublayer:self.overlayLayer];
}

static CATransform3D THMakePerspectiveTransform(CGFloat eyePosition) {     // 3
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / eyePosition;
    return transform;
}

...

@end
```

(1) 初始化faceLayers属性，赋给它一个NSMutableDictionary来保存对应检测人脸的层实例。此外，设置videoGravity为AVLayerVideoGravityResizeAspectFill使其填满整个预览层的边界范围。可以在第6章中查找有关视频gravity值的介绍。

(2) 初始化overlayLayer属性，赋给它一个新的CALayer实例并设置其边界等于视图的边界，这样它就可以填满整个预览层。设置层的sublayerTransform属性为CATransform3D，可以对所有子层应用视角转换。

(3) 创建一个返回CATransform3D类型的函数。它是Core Animation中所使用的transformation matrix类型，用于进行缩放和旋转等转换。设置m34元素让开发者可以应用视角

转换，即让子层绕Y轴旋转。

如果你之前没有用过Core Animation框架，上面的代码可能看起来比较难懂，不过不必担心，稍后就会看到这一配置的效果了。要学习Core Animation框架，推荐阅读Nick Lockwood撰写的*iOSCore Animation* (2014, Boston: Addison-Wesley)一书。

AVCaptureMetadataOutput对象捕捉的元数据位于设备空间，要使用这一元数据，首先需要将该数据转换到视图坐标系空间。幸运的是这一变换很容易，因为AVCaptureVideoPreviewLayer提供了一个方法可以完成这个艰难的任务。如代码清单7-11所示。

代码清单 7-11 元数据转换

```
@implementation THPreviewView

...

- (void)didDetectFaces:(NSArray *)faces {

    NSArray *transformedFaces = [self transformedFacesFromFaces:faces];          // 1

    // Process transformed faces. To be implemented in Listing 7.12.
}

- (NSArray *)transformedFacesFromFaces:(NSArray *)faces {                        // 2
    NSMutableArray *transformedFaces = [NSMutableArray array];
    for (AVMetadataObject *face in faces) {
        AVMetadataObject *transformedFace =                                      // 3
            [self.previewLayer transformedMetadataObjectForMetadataObject:face];
        [transformedFaces addObject:transformedFace];
    }
    return transformedFaces;
}

...

@end
```

(1) 创建一个本地NSArray来保存转换后的人脸数据。

(2) 创建一个新方法，将设备坐标空间的人脸对象转换为视图空间对象集合。

(3) 遍历传入方法的人脸集合，在for循环中调用预览层的transformedMetadataObjectForMetadataObject:方法，并将它添加到transformedFaces数组。

现在我们有一个AVMetadataFaceObject实例组成的集合，其中具有创建用户界面所需的坐标点。让我们继续学习预览视图的实现，如代码清单7-12所示。

代码清单 7-12 将检测到的人脸进行可视化

```
@implementation THPreviewView

...

- (void)didDetectFaces:(NSArray *)faces {

    NSArray *transformedFaces = [self transformedFacesFromFaces:faces];

    NSMutableArray *lostFaces = [self.faceLayers.allKeys mutableCopy];           // 1
```

```
for (AVMetadataFaceObject *face in transformedFaces) {

    NSNumber *faceID = @(face.faceID);                                  // 2
    [lostFaces removeObject:faceID];

    CALayer *layer = self.faceLayers[faceID];                           // 3
    if (!layer) {
        // no layer for faceID, create new face layer
        layer = [self makeFaceLayer];                                   // 4
        [self.overlayLayer addSublayer:layer];
        self.faceLayers[faceID] = layer;
    }

    layer.transform = CATransform3DIdentity;                            // 6
    layer.frame = face.bounds;

  }

  for (NSNumber *faceID in lostFaces) {                                 // 7
    CALayer *layer = self.faceLayers[faceID];
    [layer removeFromSuperlayer];
    [self.faceLayers removeObjectForKey:faceID];
  }

}

- (CALayer *)makeFaceLayer {                                            // 5
  CALayer *layer = [CALayer layer];
  layer.borderWidth = 5.0f;
  layer.borderColor =
    [UIColor colorWithRed:0.188 green:0.517 blue:0.877 alpha:1.000].CGColor;
  return layer;
}

...

@end
```

(1) 创建一个保存在faceLayers字典中键值的可变副本。这个数组用于确定哪些人脸移出了视图并将其对应的图层移出用户界面。

(2) 遍历每个转换的人脸对象并捕捉其关联的faceID，这个属性唯一标识一个检测到的人脸。将对象从lostFaces数组中移除，在这个方法最后面它所对应的图层不会从用户界面移出。

(3) 在faceLayers字典中查找CALayer实例当前的faceID值。这一字典作为CALayer对象的一个临时缓存。

(4) 如果对于给定的faceID没有找到对应的层，调用makeFaceLayer方法创建一个新的人脸图层并将其添加到overlayLayer。最后，将它添加到字典中就可以被之前didDetectFaces:方法的调用重新使用了。

(5) makeFaceLayer方法创建了一个新的CALayer，带有5个像素的浅蓝色加粗边框，并将这个图层返回到调用函数。

(6) 设置图层的transform属性为CATransform3DIdentity。标识变换是图层的默认变换，本

质的意思就是不可变状态。这会重新设置我们之前刚刚应用的所有变换。还需要基于人脸对象的bounds属性设置图层的框架。

(7) 最后，遍历lostFaces数组中剩下的人脸ID集合并将它们从上一图层和faceLayers字典中移出。

现在我们准备运行应用程序实际看一下这个功能的效果。应用程序默认使用前置摄像头，所以我们从摄像头中会看到在你的脸周围出现一个蓝色的方框，并实时跟踪脸部的移动。这很酷吧，我们还没有使用AVMetadataFaceObject提供的其他相关数据。我们已经讨论过人脸对象包含了斜倾角(roll angle)和偏转角(yaw angle)。让我们看一下如何在用户界面中对其进行描述，如代码清单7-13所示。

代码清单 7-13 可视化斜倾角和偏转角

```
@implementation THPreviewView

...

- (void)didDetectFaces:(NSArray *)faces {

    NSArray *transformedFaces = [self transformedFacesFromFaces:faces];

    NSMutableArray *lostFaces = [self.faceLayers.allKeys mutableCopy];

    for (AVMetadataFaceObject *face in transformedFaces) {

        NSNumber *faceID = @(face.faceID);
        [lostFaces removeObject:faceID];

        CALayer *layer = self.faceLayers[faceID];
        if (!layer) {
            // no layer for faceID, create new face layer
            layer = [self makeFaceLayer];
            [self.overlayLayer addSublayer:layer];
            self.faceLayers[faceID] = layer;
        }

        layer.transform = CATransform3DIdentity;                          // 1
        layer.frame = face.bounds;

        if (face.hasRollAngle) {
            CATransform3D t = [self transformForRollAngle:face.rollAngle]; // 2
            layer.transform = CATransform3DConcat(layer.transform, t);
        }

        if (face.hasYawAngle) {
            CATransform3D t = [self transformForYawAngle:face.yawAngle];   // 4
            layer.transform = CATransform3DConcat(layer.transform, t);
        }
    }

    for (NSNumber *faceID in lostFaces) {                                 // 6
        CALayer *layer = self.faceLayers[faceID];
```

```
        [layer removeFromSuperlayer];
        [self.faceLayers removeObjectForKey:faceID];
    }

}

// Rotate around Z-axis
- (CATransform3D)transformForRollAngle:(CGFloat)rollAngleInDegrees {        // 3
    CGFloat rollAngleInRadians = THDegreesToRadians(rollAngleInDegrees);
    return CATransform3DMakeRotation(rollAngleInRadians, 0.0f, 0.0f, 1.0f);
}

// Rotate around Y-axis
- (CATransform3D)transformForYawAngle:(CGFloat)yawAngleInDegrees {        // 5
    CGFloat yawAngleInRadians = THDegreesToRadians(yawAngleInDegrees);

    CATransform3D yawTransform =
        CATransform3DMakeRotation(yawAngleInRadians, 0.0f, -1.0f, 0.0f);

    return CATransform3DConcat(yawTransform, [self orientationTransform]);
}

- (CATransform3D)orientationTransform {                                    // 6
    CGFloat angle = 0.0;
    switch ([UIDevice currentDevice].orientation) {
        case UIDeviceOrientationPortraitUpsideDown:
            angle = M_PI;
            break;
        case UIDeviceOrientationLandscapeRight:
            angle = -M_PI / 2.0f;
            break;
        case UIDeviceOrientationLandscapeLeft:
            angle = M_PI / 2.0f;
            break;
        default: // as UIDeviceOrientationPortrait
            angle = 0.0;
            break;
    }
    return CATransform3DMakeRotation(angle, 0.0f, 0.0f, 1.0f);
}

static CGFloat THDegreesToRadians(CGFloat degrees) {
    return degrees * M_PI / 180;
}

...

@end
```

(1) 对每个人脸图层，先将它的transform属性设置为CATransform3DIdentity。这样就可以重新设置之前应用过的变换。

(2) 通过检查hasRollAngle属性判断人脸对象是否具有有效的斜倾角。如果hasRollAngle返回NO，在获取对象的rollAngle属性将会产生异常。如果人脸对象具有rollAngle，则获取相应的CATransform3D，将它与标识变换关联在一起，并设置图层的transform属性。

(3) 从人脸对象得到的rollAngle单位是度，不过我们需要使用Core Animation的转换函数将它转换为所需的弧度值。将转换结果赋给CATransform3DMakeRotation函数，x、y、z轴的参数分别为0、0、1。这样会得到一个绕Z轴的斜倾角旋转转换.

(4) 与计算rollAngle时我们所做的一样，首先需要询问人脸对象是否具有偏转角。如果有，则获取这个值并计算相应的转换，将它与图层现存的转换串联在一起，并将它设置为图层的transform属性。

(5) 构建一个关于偏转角的转换与创建斜倾角的方法类似。首先将角度转换为弧度并创建一个旋转转换，不过这次我们是绕Y轴旋转。由于overlayLayer需要应用sublayerTransform，所以图层会投影到Z轴，人脸从一侧移动到另一侧时就会出现一个3D效果。

(6) 应用程序的用户界面固定为垂直方向，不过我们需要为设备方向计算一个相应的旋转变换。如果不这样做会导致人脸图层的偏转效果不正确。这一转换会同其他变换关联。

再次运行应用程序，现在将头左右摆动，会看到矩形框发生旋转并跟踪着目标。同样，当我们将头转到肩膀后面，会看到矩形区域跟踪这一偏转并映射到3D空间。

现在我们已经深入理解了如何使用AV Foundation进行人脸检测，我们才刚刚创建了一个比较粗糙的有关人脸识别的用户界面。我希望像帽子、眼睛和胡须都出现在脑袋上。结合Core Animation图层和视频图片是完全可以捕捉到静态图片的，不过这需要用到非常重要的Quartz框架，它不在本书的讨论范围之内。我建议有兴趣的朋友可以从Apple Developer Connection网站找到Apple's SquareCam示例，或从WWDC 2013中找到StacheCam示例来学习如何实现这一结合。

7.3 机器可读代码识别

AV Foundation的一个重要新功能是可以识别机器可读的代码，这其实就是条码扫描的官方叫法。AV Foundation框架定义了对多种条码符号进行实时识别的方法，前置及后置摄像头都可以，并且只要能运行iOS 7和iOS 8的设备都支持这个功能。在学习这一技术的详细内容前，我们先来看一下框架所支持的几种不同的条码类型。

7.3.1 一维条码

一维条码是我们所知的最流行的条码类型。大量用于商店、生产制造业和零售业，在大多数商品控制系统中都扮演着举足轻重的角色。AV Foundation支持的一维条码如表7-1所示。

表 7-1　支持的一维条码

代　　码	示　　例
UPC-E	0 012345 7
EAN-8	0123 4565
EAN-13	0 123456 789104
Code 39(包含以及不包含校验和)	0123456789
Code 93	1 INFINITE LOOP
Code 128	1 Infinite Loop
交错式 2 of 5 码(仅限 iOS 8)	12345678901231
ITF(仅限 iOS 8)	01234567890128

AV Foundation还支持3种二维码。QR码主要用于移动营销，不过我们会发现在移动领域中QR码还有许多创造性的用法。Aztec码广泛用于航天领域内登机牌。PDF417通常用于商品运输应用程序。表7-2给出了这3种条码的示例图。

表 7-2 支持的二维条码

代码	示例
QR	
Aztec	
PDF-417	
Data Matrix(仅限 iOS 8)	

对条码细节的讨论不在本书的范围内，不过可以在以下网址找到这一概念的详细总结和用法讲解：http://makebarcode.com/。

7.3.2 创建条码扫描器

让我们创建一个示例应用程序来使用这些功能。在Chapter 7目录中可以找到一个名为CodeKamera_Starter的示例项目。我们从创建摄像头控制器对象开始，如代码清单7-14所示。

代码清单 7-14 THCameraController 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THCodeDetectionDelegate <NSObject>
- (void)didDetectCodes:(NSArray *)codes;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id <THCodeDetectionDelegate> codeDetectionDelegate;

@end
```

THCameraController的接口与我们为FaceKamera应用程序创建的接口几乎一样；唯一的

不同是定义了一个新的委托协议THCodeDetectionDelegate。这个委托只定义了一个didDetectCodes:方法，当有新的条码被识别时会调用该方法。让我们转到它的实现中开始编写这个功能，如代码清单7-15所示。

代码清单 7-15　设置摄像头

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
@interface THCameraController () <AVCaptureMetadataOutputObjectsDelegate>   // 1
@property (strong, nonatomic) AVCaptureMetadataOutput *metadataOutput;
@end

@implementation THCameraController

- (NSString *)sessionPreset {                                                 // 2
    return AVCaptureSessionPreset640x480;
}

- (BOOL)setupSessionInputs:(NSError *__autoreleasing *)error {
    BOOL success = [super setupSessionInputs:error];
    if (success) {
        if (self.activeCamera.autoFocusRangeRestrictionSupported) {          // 3

            if ([self.activeCamera lockForConfiguration:error]) {

                self.activeCamera.autoFocusRangeRestriction =
                            AVCaptureAutoFocusRangeRestrictionNear;

                [self.activeCamera unlockForConfiguration];
            }
        }
    }
    return success;
}

...

@end
```

(1) 首先创建一个类扩展来保存到AVCaptureMetadataOutput的引用。与我们在FaceKamera应用程序中做的一样，这个类需要接受AVCaptureMetadataOutputObjectsDelegate协议，这样才能在得到新元数据时收到通知。

(2) 重写sessionPreset方法返回一个其他会话预设类型来使用。可以任意选择最适合应用程序的捕捉预设类型，不过苹果公司建议开发者使用最低合理解决方案以提高性能。

(3) 捕捉设备的自动对焦功能通常在任何距离都可以进行扫描，这一功能也是我们在典型的相机或视频捕捉应用程序中所需要的。不过，iOS 7版本增加了一个新属性，允许我们使用一个范围约束对这个功能进行定制。我们扫描的大部分条码距离都不远，所以可以通过缩小扫描区域来提升识别成功率。记得检测是否可以支持该功能，如果可以，则设置autoFocusRangeRestriction属性为AVCaptureAutoFocusRangeRestrictionNear。

在完成捕捉设备输入配置后，让我们看一下捕捉会话输出的配置，如代码清单7-16所示。

代码清单 7-16 配置会话输出

```
@implementation THCameraController

...

- (BOOL)setupSessionOutputs:(NSError **)error {
    self.metadataOutput = [[AVCaptureMetadataOutput alloc] init];

    if ([self.captureSession canAddOutput:self.metadataOutput]) {
        [self.captureSession addOutput:self.metadataOutput];

        dispatch_queue_t mainQueue = dispatch_get_main_queue();
        [self.metadataOutput setMetadataObjectsDelegate:self
                                                 queue:mainQueue];

        NSArray *types = @[AVMetadataObjectTypeQRCode,                    // 1
                           AVMetadataObjectTypeAztecCode];

        self.metadataOutput.metadataObjectTypes = types;

    } else {
        NSDictionary *userInfo = @{NSLocalizedDescriptionKey:
                                   @"Failed to add metadata output."};
        *error = [NSError errorWithDomain:THCameraErrorDomain
                                     code:THCameraErrorFailedToAddOutput
                                 userInfo:userInfo];
        return NO;
    }

    return YES;
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
      fromConnection:(AVCaptureConnection *)connection {

    [self.codeDetectionDelegate didDetectCodes:metadataObjects];          // 2

}

@end
```

(1) setupSessionOutputs:方法的实现与我们在FaceKamera应用程序中所用到的实现方法几乎一样。唯一的不同是metadataObjectTypes会被AVCaptureMetadataOutput对象使用。本例中我们指定感兴趣的扫描对象是QR码和Aztec码。

(2) 在委托回调中调用THCodeDetectionDelegate，并传递由我们识别到的元数据对象组成的数组。

处理条码时出现的AVMetadataCaptureOutput对象是AVMetadataMachineReadableCodeObject实例。这个对象定义了一个stringValue属性用于提供条码实际数据值，还定义了两个属性用来表示条码的几何特征。bounds属性提供了识别码的按坐标轴对齐的矩形边界，corners属性提供角点字典表示的NSArray。后一个属性更实用，因为使用它可以让我们构建一个与条码的角点坐标紧密对齐的Bezier路径。

7.3.3　创建条码识别的委托

本应用程序中的THCodeDetectionDelegate是THPreviewView。让我们看它是如何实现的，如代码清单7-17所示。

代码清单 7-17　THPreviewView 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THCodeDetectionDelegate.h"

@interface THPreviewView : UIView <THCodeDetectionDelegate>

@property (strong, nonatomic) AVCaptureSession *session;

@end
```

这个接口看起来很熟悉，唯一需要注意的是它接受了THCodeDetectionDelegate协议，使它可以成为元数据的目标。让我们看一下代码清单7-18给出的实现过程。与我们在FaceKamera应用程序中讨论的预览视图一样，我们分节来学习，先从基础类结构开始。

代码清单 7-18　THPreviewView 的实现

```
#import "THPreviewView.h"

@interface THPreviewView ()                                          // 1
@property (strong, nonatomic) NSMutableDictionary *codeLayers;
@end

@implementation THPreviewView

+ (Class)layerClass {                                                // 2
  return [AVCaptureVideoPreviewLayer class];
}

- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setupView];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        [self setupView];
    }
    return self;
}

- (void)setupView {                                                 // 3
    _codeLayers = [NSMutableDictionary dictionary];
    self.previewLayer.videoGravity = AVLayerVideoGravityResizeAspect;
}
```

```
- (AVCaptureSession*)session {
  return self.previewLayer.session;
}

- (void)setSession:(AVCaptureSession *)session {                    // 4
    self.previewLayer.session = session;
}

- (AVCaptureVideoPreviewLayer *)previewLayer {
    return (AVCaptureVideoPreviewLayer *)self.layer;
}

- (void)didDetectCodes:(NSArray *)codes {

}

@end
```

(1) 创建一个类扩展并定义类所使用的内部属性。

(2) 重写layerClass方法并返回一个AVCaptureVideoPreviewLayer实例，就像上一节中做的一样。重写该方法可以让视图的备用层成为AVCaptureVideoPreviewLayer的实例。

(3) 用NSMutableDictionary实例来初始化codeLayers属性。它用来保存一组表示识别编码的几何信息的图层。还需要设置图层的videoGravity属性为AVLayerVideoGravityResizeAspect，保证宽高比在边界范围之内。

(4) 要建立AVCaptureSession和AVCapturePreviewLayer之间的关联，可重写setSession:方法并设置AVCaptureSession实例作为预览层的session属性。

基础的类结构创建完成后，我们继续实现didDetectCodes:方法，如代码清单7-19所示。第一步需要做的是将设备坐标系中的元数据转换为视图坐标系的数据。

代码清单 7-19　元数据转换

```
@implementation THPreviewView

...

- (void)didDetectCodes:(NSArray *)codes {

    NSArray *transformedCodes = [self transformedCodesFromCodes:codes];     // 1

}

- (NSArray *)transformedCodesFromCodes:(NSArray *)codes {                   // 2
    NSMutableArray *transformedCodes = [NSMutableArray array];
    for (AVMetadataObject *code in codes) {
        AVMetadataObject *transformedCode =
          [self.previewLayer transformedMetadataObjectForMetadataObject:code];
        [transformedCodes addObject:transformedCode];
    }
    return transformedCodes;
}

@end
```

(1) 基于传递给委托方法的AVMetadataMachineReadableCodeObject实例，创建一个本地NSArray来保存转换完成的元数据对象。

(2) 创建一个新的方法，将设备坐标空间元数据对象转换为视图坐标空间对象。这个方法会遍历传进方法中的元数据对象集合，对每一对象调用预览图层的transformedMetadata-ObjectForMetadataObject:方法，并将它添加到transformedCodes数组。

现在我们有一组AVMetadataMachineReadableCodeObject实例，它们都具有组成用户界面的有效坐标。让我们继续实现这个方法，如代码清单7-20所示。

代码清单 7-20　创建图层

```
@implementation THPreviewView

...

- (void)didDetectCodes:(NSArray *)codes {

    NSArray *transformedCodes = [self transformedCodesFromCodes:codes];

    NSMutableArray *lostCodes = [self.codeLayers.allKeys mutableCopy];          // 1

    for (AVMetadataMachineReadableCodeObject *code in transformedCodes) {

        NSString *stringValue = code.stringValue;                                // 2
        if (stringValue) {
            [lostCodes removeObject:stringValue];
        } else {
            continue;
        }

        NSArray *layers = self.codeLayers[stringValue];                          // 3

        if (!layers) {
            // no layers for stringValue, create new code layers
            layers = @[[self makeBoundsLayer], [self makeCornersLayer]];

            self.codeLayers[stringValue] = layers;
            [self.previewLayer addSublayer:layers[0]];
            [self.previewLayer addSublayer:layers[1]];
        }

        CAShapeLayer *boundsLayer = layers[0];                                   // 4
        boundsLayer.path  = [self bezierPathForBounds:code.bounds].CGPath;

        NSLog(@"String: %@", stringValue);                                       // 5
    }

    for (NSString *stringValue in lostCodes) {                                   // 6
        for (CALayer *layer in self.codeLayers[stringValue]) {
            [layer removeFromSuperlayer];
        }
        [self.codeLayers removeObjectForKey:stringValue];
    }
}
```

```
- (UIBezierPath *)bezierPathForBounds:(CGRect)bounds {
    return [UIBezierPath bezierPathWithRect:bounds];
}

- (CAShapeLayer *)makeBoundsLayer {
    CAShapeLayer *shapeLayer = [CAShapeLayer layer];
    shapeLayer.strokeColor =
      [UIColor colorWithRed:0.95f green:0.75f blue:0.06f alpha:1.0f].CGColor;
    shapeLayer.fillColor = nil;
    shapeLayer.lineWidth = 4.0f;
    return shapeLayer;
}

- (CAShapeLayer *)makeCornersLayer {
    CAShapeLayer *cornersLayer = [CAShapeLayer layer];
    cornersLayer.lineWidth = 2.0f;
    cornersLayer.strokeColor =
      [UIColor colorWithRed:0.172 green:0.671 blue:0.428 alpha:1.000].CGColor;
    cornersLayer.fillColor =
      [UIColor colorWithRed:0.190 green:0.753 blue:0.489 alpha:0.500].CGColor;

    return cornersLayer;
}

@end
```

(1) 从codeLayers字典中创建键的可变副本。这个数组用来判断哪个图层应该在方法结尾处移除。

(2) 查找代码的stringValue。如果返回一个有效的字符串对象，则将它从lostCodes数组中移除。如果stringValue为nil，则执行continue语句继续下一个迭代，因为应该跳过不具有合法值的任何代码。

(3) 为当前的stringValue查找已有的图层数组，如果没有对应的条目，则创建两个新的CAShapeLayer对象。CAShapeLayer是具体化的CALayer子类，用于绘制Bezier路径。第一个对象绘制bounds矩形，另一个对象用于绘制之后构建的corners路径。在字典中为这些图层添加一个条目，并将每个图层加到previewLayer中。

(4) 对于边界图层，创建一个和对象的bounds关联的UIBezierPath。Core Animation只支持Quartz类型，所以我们获取UIBezierPath的CGPath属性，会得到一个CGPathRef用来赋给图层的path属性。

(5) 为达到应用程序的目的，我们将stringValue记录到控制台。在实际应用程序中，开发者会以其他有意义的方式将这个值展示给用户。

(6) 最后，遍历剩下的lostCodes条目，将每一个条目的图层从previewLayer中移除，并将数组条目从字典中移除。

现在可以运行应用程序了，我们可以使用表7-2中的条码来测试这个功能。对于每个识别码，都会在屏幕上看到一个矩形方框。如果摄像头视图与条码垂直，矩形的边界就会对齐。不过当以一定的角度进行扫描时就不能很好对齐。这时就会用到corners属性了，大部分情况下，corners属性都可以提供较好的资源来绘制代码的几何图形，因为它提供了包含代码角点

的字典。使用这些点很容易构建出与条码角落紧密对齐的Bezier路径。让我们看如何应用corners属性，如代码清单7-21所示。

代码清单 7-21　使用 corners 属性

```
@implementation THPreviewView

...

- (void)didDetectCodes:(NSArray *)codes {

    NSArray *transformedCodes = [self transformedCodesFromCodes:codes];

    NSMutableArray *lostCodes = [self.codeLayers.allKeys mutableCopy];

    for (AVMetadataMachineReadableCodeObject *code in transformedCodes) {

        NSString *stringValue = code.stringValue;
        if (stringValue) {
            [lostCodes removeObject:stringValue];
        } else {
            continue;
        }

        NSArray *layers = self.codeLayers[stringValue];

        if (!layers) {
            // no layers for stringValue, create new code layers
            layers = @[[self makeBoundsLayer], [self makeCornersLayer]];

            self.codeLayers[stringValue] = layers;
            [self.previewLayer addSublayer:layers[0]];
            [self.previewLayer addSublayer:layers[1]];
        }

        CAShapeLayer *boundsLayer  = layers[0];
        boundsLayer.path  = [self bezierPathForBounds:code.bounds].CGPath;

        CAShapeLayer *cornersLayer = layers[1];                                  // 1
        cornersLayer.path = [self bezierPathForCorners:code.corners].CGPath;

        NSLog(@"String: %@", stringValue);
    }

    for (NSString *stringValue in lostCodes) {
        for (CALayer *layer in self.codeLayers[stringValue]) {
            [layer removeFromSuperlayer];
        }
        [self.codeLayers removeObjectForKey:stringValue];
    }
}

- (UIBezierPath *)bezierPathForCorners:(NSArray *)corners {
    UIBezierPath *path = [UIBezierPath bezierPath];
    for (int i = 0; i < corners.count; i++) {
        CGPoint point = [self pointForCorner:corners[i]];                        // 2
```

```
        if (i == 0) {                                                    // 4
            [path moveToPoint:point];
        } else {
            [path addLineToPoint:point];
        }
    }
    [path closePath];                                                    // 5
    return path;
}

- (CGPoint)pointForCorner:(NSDictionary *)corner {                      // 3
    CGPoint point;
    CGPointMakeWithDictionaryRepresentation((CFDictionaryRef)corner, &point);
    return point;
}

...


@end
```

(1) 对于cornersLayer，基于元数据对象的corners属性构建一个CGPath。

(2) 在bezierPathForCorners:方法中，首先创建一个空的UIBezierPath。遍历corners数组中的条目，为每个条目构建一个CGPoint。

(3) 包含角点对象的字典具有x值的条目和y值的条目。开发者可将这些值取出来手动创建CGPoint值。不过Quartz框架为这个过程提供了一个便捷函数CGPointMakeWithDictionary-Representation，只需要给要填充的CGPoint结构传递字典和指针即可。

(4) 基于这些点使用UIBezierPath方法构建Bezier路径。对第一个点调用moveToPoint:方法开始绘制路径，对后面的点调用addLineToPoint:连到其余的点。

(5) 最后，关闭路径并返回UIBezierPath实例给调用函数。

再次运行该应用程序，现在我们移动设备，可以看到图层紧密与条码的边角对齐了。

识别机器可读码的功能是AV Foundation框架非常受欢迎的功能之一，并且我们认为这个功能将会大量应用于应用程序的开发中。我们建议大家对其他几种类型的条码都进行实验，以更好地理解该功能。还可以参阅苹果公司的Tech Note TN2325，那里对使用这个功能给出了一个实用的建议和观点。

7.4 使用高帧率捕捉

在iOS 7版本引入的几个功能中，对于最新的iOS设备，最有趣的一个功能就是高帧率视频的捕捉。以高帧率(FPS)捕捉视频内容带来了很多好处。首先高FPS视频通常具有更逼真的效果和更好的清晰度，因为由于增加了瞬时采样率而使对于移动物体的捕捉更加精确。对于细节方面的加强和动作流畅度的提升非常明显，尤其是当录制快速移动的内容时更为明显，比如运动场景。另一个应用更广泛的优势在于可以支持高质量的慢动作视频效果。最新发布的iOS设备支持以60FPS的速率捕捉视频(iPhone 5s可以支持120FPS)，这意味着可以使用播放速度一半的帧率还可以达到以30FPS速率进行播放，播放体验更加流畅。没有什么比加入慢

动作更好的了，现在这个功能已经可以被我们使用了。

高帧率捕捉是开发者在一些场景中希望使用的技术。幸运的是，苹果公司并没有单独推出这一个功能，而是通过框架为这个功能给出了强大的支撑。

- **捕捉**：框架支持 60FPS 帧率捕捉 720p 视频，或在 iPhone 5s 中将帧率提高到 120FPS，并带有视频稳定技术。此外还可以支持启用 droppable P-frames 的 h.264 特性，保证高帧率内容可以在旧设备上流畅播放。
- **播放**：AVPlayer 已经支持以多种播放帧率播放资源内容，不过 AVPlayerItem 给出了一个显著的音频处理过程的优化，即在降低播放率的时候如何控制音频内容的播放。AVPlayerItem 现在带有一个 audioTimePitchAlgorithm 属性，可以为降速或加速播放率设置算法。查看 AVPlayerItem 的 API 文档了解可用的选项。
- **编辑**：框架的编辑功能为在可变组合中执行缩放编辑提供全面支持。我们将在第 8 章开始讨论有关媒体资源组合及编辑的内容。
- **导出**：AV Foundation 提供了保存原始帧率的功能，所以高 FPS 内容可以被导出或进行帧率转换，这样就可以保证所有内容都以标准的 30FPS 进行输出。

7.4.1　高帧率捕捉综述

在上一章中，我们学习了如何通过指定会话预设值来设置捕捉会话的服务质量，开发者可能希望找到一些新的预设来支持高帧率捕捉。但是，支持这个功能并没有这么简单。无法实际扩展这一预设机制，因为要满足所有的帧率和尺寸组合会导致预设参数暴增。所以这一功能通过在iOS 7版本加入的并行配置机制来实现。这个新方法并没有替代预设模型，而是在需要对内容进行更细致配置时给出的另一种配置会话的方法。

本章前面介绍了使用AVCaptureDeviceFormat来判断激活捕捉设备格式所支持的最大缩放因子。在视频缩放示例中，用到了设备的activeFormat，会根据选中的会话预设自动设置它。除了active格式外，还可以通过获取formats属性得到设备所有支持的格式。AVCaptureDeviceFormat实例具有videoSupportedFrameRateRanges属性，它包含了一个AVFrameRateRange对象数组，其中带有格式所支持的最小帧率、最大帧率和时长信息。使用高帧率捕捉的一个基本秘诀就是找到设备的最高质量格式，找到它相关的帧时长，之后手动设置捕捉设备的格式和帧时长。当我们借用一些示例来看会更清楚一些，打开Chapter 7目录中的SlowKamera_Starter项目，开始实现这个功能。

7.4.2　支持高帧率捕捉

要支持这一功能现在看来有些麻烦，不过我们可以通过将这一功能封装到AVCapture-Device上的分类来简化这一过程，如代码清单7-22所示。我们先来创建这个分类。

代码清单 7-22　高帧率捕捉分类

```
#import <AVFoundation/AVFoundation.h>

@interface AVCaptureDevice (THAdditions)

- (BOOL)supportsHighFrameRateCapture;
```

```
- (BOOL)enableHighFrameRateCapture:(NSError **)error;

@end
```

我们在AVCaptureDevice创建分类，添加两个方法：一个用来判断当前设备是否支持高帧率捕捉，另一个用来实现这个功能。下面看一下具体的实现。

在给出分类方法的实现前，首先需要在AVCaptureDevice+THAdditions.m文件中创建一个私有类THQualityOfService，它可以简化分类方法的实现，如代码清单7-23所示。

代码清单 7-23 私有类 THQualityOfService

```
#import "AVCaptureDevice+THAdditions.h"
#import "THError.h"

@interface THQualityOfService : NSObject

@property(strong, nonatomic, readonly) AVCaptureDeviceFormat *format;
@property(strong, nonatomic, readonly) AVFrameRateRange *frameRateRange;
@property(nonatomic, readonly) BOOL isHighFrameRate;

+ (instancetype)qosWithFormat:(AVCaptureDeviceFormat *)format
            frameRateRange:(AVFrameRateRange *)frameRateRange;

- (BOOL)isHighFrameRate;

@end

@implementation THQualityOfService

+ (instancetype)qosWithFormat:(AVCaptureDeviceFormat *)format
            frameRateRange:(AVFrameRateRange *)frameRateRange {

   return [[self alloc] initWithFormat:format frameRateRange:frameRateRange];
}

- (instancetype)initWithFormat:(AVCaptureDeviceFormat *)format
            frameRateRange:(AVFrameRateRange *)frameRateRange {
   self = [super init];
   if (self) {
      _format = format;
      _frameRateRange = frameRateRange;
   }
   return self;
}

- (BOOL)isHighFrameRate {
   return self.frameRateRange.maxFrameRate > 30.0f;
}

@end
```

要启用高帧率捕捉，需要查看捕捉设备的可用格式，并找到所支持的最高AVCaptureDeviceFormat及其最高AVFrameRateRange。THQualityOfService用来保存这些值并简化公共分类方法的实现。让我们看一下第一个分类方法的实现，如代码清单7-24所示。

代码清单 7-24　确定高 FPS 支持

```
@implementation AVCaptureDevice (THAdditions)

- (BOOL)supportsHighFrameRateCapture {
    if (![self hasMediaType:AVMediaTypeVideo]) {                          // 1
        return NO;
    }
    return [self findHighestQualityOfService].isHighFrameRate;            // 2
}

- (THQualityOfService *)findHighestQualityOfService {

    AVCaptureDeviceFormat *maxFormat = nil;
    AVFrameRateRange *maxFrameRateRange = nil;

    for (AVCaptureDeviceFormat *format in self.formats) {

        FourCharCode codecType =                                          // 3
            CMVideoFormatDescriptionGetCodecType(format.formatDescription);

        if (codecType == kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange) {

            NSArray *frameRateRanges = format.videoSupportedFrameRateRanges;

            for (AVFrameRateRange *range in frameRateRanges) {            // 4
                if (range.maxFrameRate > maxFrameRateRange.maxFrameRate) {
                    maxFormat = format;
                    maxFrameRateRange = range;
                }
            }
        }
    }

    return [THQualityOfService qosWithFormat:maxFormat                    // 6
                        frameRateRange:maxFrameRateRange];
}

...

@end
```

(1) 通过查看AVCaptureDevice实例是否具有AVMediaTypeVideo媒体类型值来判断它是不是视频设备。如果不是，则从该方法中返回NO。

(2) 调用findHighestQualityOfService方法来确定这个摄像头设备所支持的最大format和帧率。通过THQualityOfService对象查看其是否支持高帧率捕捉并返回结果值。

(3) 遍历所有捕捉设备的支持formats并对每一个元素从formatDescription获取相应的codecType。CMFormatDescriptionRef是一个Core Media的隐含类型，提供了格式对象的许多相关信息。这里只需要codeType值等于420YpCbCr8BiPlanarVideoRange的格式，筛选出视频格式。

(4) 遍历由videoSupportedFrameRateRanges属性返回的format的AVFrameRateRange对象集。对于每一个，都判断其maxFrameRate是否大于当前的最大值。最终的目的是找到这个摄

像头设备所提供的最高的format和帧率。

(5) 最后，返回一个新的内部THQualityOfService的实例，捕捉所支持的最高format和帧率范围。

现在我们知道如何确定活动摄像头所支持的最高级别了，下面看一下如何实际使用这个功能，如代码清单7-25所示。

代码清单 7-25 高帧率捕捉

```
- (BOOL)enableMaxFrameRateCapture:(NSError **)error {

    THQualityOfService *qos = [self findHighestQualityOfService];

    if (!qos.isHighFrameRate) {                                          // 1
        if (error) {
            NSString *message = @"Device does not support high FPS capture";
            NSDictionary *userInfo = @{NSLocalizedDescriptionKey : message};

            NSUInteger code = THCameraErrorHighFrameRateCaptureNotSupported;

            *error = [NSError errorWithDomain:THCameraErrorDomain
                                         code:code
                                     userInfo:userInfo];
        }
        return NO;
    }

    if ([self lockForConfiguration:error]) {                             // 2

        CMTime minFrameDuration = qos.frameRateRange.minFrameDuration;

        self.activeFormat = qos.format;                                  // 3
        self.activeVideoMinFrameDuration = minFrameDuration;             // 4
        self.activeVideoMaxFrameDuration = minFrameDuration;

        [self unlockForConfiguration];
        return YES;
    }
    return NO;

}
```

(1) 首先找到设备支持的最高质量的服务。如果不支持高帧率捕捉，则生成一个错误指针并返回NO。

(2) 在修改捕捉设备前，需要为配置操作锁住设备。

(3) 将设备的activeFormat设置为检索到的AVCaptureDeviceFormat。

(4) 将最小帧时长和最大帧时长设置为AVFrameRateRange定义的值。AV Foundation通常处理帧时长数据，使用CMTime实例而不是帧率。minFrameDuration值为maxFrameRate的倒数值。比如帧率为60FPS，则duration为1/60秒。

完成分类之后，让我们在应用程序的THCameraController类中使用它，如代码清单7-26所示。

代码清单 7-26　使用 AVCaptureDevice+THAdditions

```
#import "THCameraController.h"
#import "AVCaptureDevice+THAdditions.h"

@implementation THCameraController

- (BOOL)cameraSupportsHighFrameRateCapture {
    return [self.activeCamera supportsHighFrameRateCapture];
}

- (BOOL)enableHighFrameRateCapture {
    NSError *error;
    BOOL enabled = [self.activeCamera enableMaxFrameRateCapture:&error];
    if (!enabled) {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
    return enabled;
}

@end
```

如果你有一个最新的设备，就可以实际看下这个功能。启动应用程序并录制几秒视频。当点击Stop按钮时，视频显示在一个带有帧率控件的播放器中。尝试选择不同的帧率选项来体验一下高帧率视频的效果。

高帧率捕捉已经成为广泛需求的功能了，从iOS 7开始开发者就可以自己对其进行处理。除了捕捉外，AV Foundation对于在各个方面使用高帧率内容都进行了完全的支持，它成为了我们创建捕捉应用程序新类时的一个非常有创造力的工具。

7.5　视频处理

在第6章，使用AVCaptureMovieFileOutput类捕捉QuickTime影片，这个类定义了从内置摄像头捕捉视频数据的简单方法，不过它无法同视频数据进行交互，而交互又是许多场景所需要的功能。思考一下苹果产品Photo Booth的需求，它对视频流应用了实时的效果，或者思考一下Sphero公司的增强现实类应用程序Sharky the Beaver，它将交互元素与摄像头捕捉到的环境内容融合在一起。这两个示例都需要对捕捉到的视频数据进行比AVCapture-MovieFileOutput更加底层的控制。当需要使用底层控制时，就会用到框架提供的最底层的视频捕捉输出AVCaptureVideoDataOutput。

AVCaptureVideoDataOutput是一个AVCaptureOutput子类，可以直接访问摄像头传感器捕捉到的视频帧。这是一个强大的功能，因为这样我们就完全控制了视频数据的格式、时间和元数据，可以按照需求操作视频内容。大部分情况下，处理过程都是使用OpenGL ES或Core Image，不过有时Quartz也可以满足一些简单的处理要求。在我们即将创建的示例应用程序中，我们会整合OpenGL ES，因为它在视频数据处理中应用更为广泛，不过下一章将介绍如何整合Core Image并学习其强大的筛选器集合。

> **注意：**
>
> AV Foundation 为处理音频数据提供了一个底层捕捉输出 AVCaptureAudioDataOutput。本章仅介绍 AVCaptureVideoDataOutput 的用法，不过在下一章讨论 AVAssetReader 和 AVAssetWriter 时，会用到音频处理的这个兄弟类。

使用AVCaptureVideoDataOutput与本章前面使用AVCaptureMetadataOutput的方法类似，最明显的区别在于它们各自的委托回调。与输出AVMetadataObject实例不同，AVCaptureVideoDataOutput输出的对象需要通过AVCaptureVideoDataOutputSampleBufferDelegate协议包含视频数据。

AVCaptureVideoDataOutputSampleBufferDelegate定义了下面两个方法：

- **captureOutput:didOutputSampleBuffer:fromConnection:** 每当有一个新的视频帧写入时该方法就会被调用。数据会基于视频数据输出的 videoSettings 属性进行解码或重新编码。
- **captureOutput:didDropSampleBuffer:fromConnection:** 每当一个迟到的视频帧被丢弃时就会调用该方法。通常是因为在 didOutputSampleBuffer:调用中消耗了太多处理时间就会调用该方法。开发者应该尽量提高处理效率，否则将收不到缓存数据。

这两个方法中最重要的参数与ample buffer相关。sample buffer以CMSampleBuffer对象的形式提供，在第6章讨论AVCaptureStillImageOutput时曾简单介绍过。深入理解这个类型非常重要，所以让我们看一下这个对象的含义和作用。

7.6 学习 CMSampleBuffer

CMSampleBuffer是一个由Core Media框架提供的Core Foundation风格的对象，用于在媒体管道中传输数字样本。CMSampleBuffer的角色是将基础的样本数据进行封装并提供格式和时间信息，还会加上所有在转换和处理数据时用到的元数据。让我们先看一下CMSampleBuffer提供的样本数据吧。

7.6.1 样本数据

在使用AVCaptureVideoDataOutput时，sample buffer会包含一个CVPixelBuffer，它是一个带有单个视频帧原始像素数据的Core Video对象。下面的示例展示了如何直接操作CVPixelBuffer的内容为捕捉到的图片buffer应用一个灰度效果。

```
const int BYTES_PER_PIXEL = 4;

CMSampleBufferRef sampleBuffer = // obtained sample buffer

CVPixelBufferRef pixelBuffer =                                          // 1
   CMSampleBufferGetImageBuffer(sampleBuffer);

CVPixelBufferLockBaseAddress( pixelBuffer, 0);                          // 2

size_t bufferWidth = CVPixelBufferGetWidth(pixelBuffer);                // 3
size_t bufferHeight = CVPixelBufferGetHeight(pixelBuffer);
```

```
unsigned char *pixel =                                                    // 4
    (unsigned char *)CVPixelBufferGetBaseAddress(pixelBuffer);
unsigned char grayPixel;

for (int row = 0; row < bufferHeight; row++) {                            // 5
    for(int column = 0; column < bufferWidth; column++) {
        grayPixel = (pixel[0] + pixel[1] + pixel[2]) / 3;
        pixel[0] = pixel[1] = pixel[2] = grayPixel;
        pixel += BYTES_PER_PIXEL;
    }
}

CVPixelBufferUnlockBaseAddress(pixelBuffer, 0);                           // 6

// Process grayscale video frame
```

示例中的新内容很多，让我们具体来分析一下吧。

(1) 首先使用CMSampleBufferGetImageBuffer函数从CMSampleBufferRef中获取基本的CVPixelBuffer。CVPixelBuffer在主内存中保存像素数据，提供了操作内容的机会。

(2) 在与CVPixelBuffer数据交互前，必须调用CVPixelBufferLockBaseAddress获取一个相应内存块的锁。

(3) 使用CVPixelBufferGetWidth函数和CVPixelBufferGetHeight函数确定像素buffer的宽和高，这样就可以迭代它的行和列。

(4) CVPixelBufferGetBaseAddress函数可以得到像素buffer的基址指针。现在就可以在buffer中进行索引并迭代其数据。

(5) 迭代buffer中像素的行和列，执行简单的RGB像素灰度平均。

(6) 最后，调用CVPixelBufferUnlockBaseAddress函数释放第2步中的锁。

现在，我们可以把buffer转换为CGImageRef或UIImage，或执行任何所需的附加图像处理。我们建议还可以额外看一下CVPixelBuffer.h中的CVPixelBuffer函数。

7.6.2　格式描述

除了原始媒体样本本身之外，CMSampleBuffer还提供了以CMFormatDescription对象的形式来访问样本的格式信息。CMFormatDescription.h定义了大量函数用于访问媒体样本的更多细节。在头文件中带有CMFormatDecription前缀的函数一般可以用于所有的媒体类型，还有前缀为CMVideoFormatDescription和CMAudioFormatDescription的函数分别适用于获取视频和音频的细节。

让我们看一下使用CMFormatDescription来区别音频和视频数据的一个示例。

```
CMFormatDescriptionRef formatDescription =
    CMSampleBufferGetFormatDescription(sampleBuffer);

CMMediaType mediaType = CMFormatDescriptionGetMediaType(formatDescription);

if (mediaType == kCMMediaType_Video) {
    CVPixelBufferRef pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer);
    // Process the frame of video
```

```
} else if (mediaType == kCMMediaType_Audio) {
    CMBlockBufferRef blockBuffer = CMSampleBufferGetDataBuffer(sampleBuffer);
    // Process audio samples
}
```

7.6.3 时间信息

CMSampleBuffer还定义了关于媒体样本的时间信息。可以分别使用CMSampleBuffer-GetPresentationTimeStamp函数和CMSampleBufferGetDecodeTimeStamp函数提取时间信息来得到原始的表示时间戳和解码时间戳。下一章讨论AVAssetReader和AVAssetWriter时会更多地用到这些函数。

7.6.4 附加的元数据

Core Media还在CMAttachment.h中定义了一个CMAttachment形式的元数据协议。API提供了读取和写入底层元数据的基础架构，比如可交换图片文件格式(Exif)标签。比如，下面的示例中我们可以从一个给定的CMSampleBuffer中获取Exif元数据。

```
CFDictionaryRef exifAttachments =
    (CFDictionaryRef)CMGetAttachment(sampleBuffer,
                                     kCGImagePropertyExifDictionary,
                                     NULL);
```

如果将这个字典数据打印到控制台，可以看到类似下例的输出信息。

```
{
    ApertureValue = "2.526068811667587";
    BrightnessValue = "-0.4554591284958377";
    ExposureMode = 0;
    ExposureProgram = 2;
    ExposureTime = "0.04166666666666666";
    FNumber = "2.4";
    Flash = 32;
    FocalLenIn35mmFilm = 35;
    FocalLength = "2.18";
    ISOSpeedRatings =    (
        800
    );
    LensMake = Apple;
    LensModel = "iPhone 5 front camera 2.18mm f/2.4";
    LensSpecification =    (
        "2.18",
        "2.18",
        "2.4",
        "2.4"
    );
    MeteringMode = 5;
    PixelXDimension = 640;
    PixelYDimension = 480;
```

```
    SceneType = 1;
    SensingMethod = 2;
    ShutterSpeedValue = "4.584985584026477";
    WhiteBalance = 0;
}
```

CMSampleBuffer和它相关的类型在处理高级AV Foundation用例时扮演着重要的作用。稍后会实际用到它们，不过关于这个类型对象的一些额外用法将在下一章中介绍。

7.6.5 AVCaptureVideoDataOutput

让我们通过示例项目来学习如何使用AVCaptureVideoDataOutput。在Chapter 7目录中可以找到一个名为CubeKamera_Starter的示例项目。这个项目用到了一些将AV Foundation和OpenGL ES相结合的技术。CubeKamera应用程序从前置摄像头捕捉视频并将这些视频帧映射为一个旋转方块的OpenGL贴图。

从项目的THCameraController类接口开始学习吧，如代码清单7-27所示。

代码清单 7-27　THCameraController 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THTextureDelegate <NSObject>
- (void)textureCreatedWithTarget:(GLenum)target name:(GLuint)name;
@end

@interface THCameraController : THBaseCameraController

- (instancetype)initWithContext:(EAGLContext *)context;
@property (weak, nonatomic) id <THTextureDelegate> textureDelegate;

@end
```

这个类的接口与本章之前的几个类类似，只不过多了几个有关OpenGL ES类型的定义。THCameraController的创建带有一个EAGLContext示例。这个对象提供了用于管理状态的渲染上下文和使用OpenGL ES进行绘制所需的资源。头文件还定义了一个THTextureDelegate协议，每当新的贴图(不可变OpenGL ES图片)被创建时就会调用这个协议。让我们看一下它的实现方法，如代码清单7-28所示。

代码清单 7-28　捕捉输出的配置

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate>

@property (weak, nonatomic) EAGLContext *context;
@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;

@end

@implementation THCameraController
```

```
- (instancetype)initWithContext:(EAGLContext *)context {
    self = [super init];
    if (self) {
        _context = context;
    }
    return self;
}

- (NSString *)sessionPreset {                                          // 1
    return AVCaptureSessionPreset640x480;
}

- (BOOL)setupSessionOutputs:(NSError **)error {

    self.videoDataOutput = [[AVCaptureVideoDataOutput alloc] init];

    self.videoDataOutput.videoSettings =                               // 2
        @{(id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA)};

    [self.videoDataOutput setSampleBufferDelegate:self                 // 3
                                          queue:dispatch_get_main_queue()];

    if ([self.captureSession canAddOutput:self.videoDataOutput]) {     // 4
        [self.captureSession addOutput:self.videoDataOutput];
        return YES;
    }

    return NO;
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
       fromConnection:(AVCaptureConnection *)connection {

}

@end
```

(1) 这个应用程序不需要高分辨率的视频，所以我们重写sessionPreset方法来返回AVCaptureSessionPreset640x480。

(2) 创建一个新的AVCaptureVideoDataOutput实例并自定义其videoSettings字典。摄像头初始格式为双平面 420v。如果回顾第1章关于色度子抽样的讨论，这个格式分为亮度和色度，并且在垂直和水平方法对颜色进行子抽样。虽然直接只用这个原始格式肯定可以，不过在使用OpenGL ES时经常会选用BGRA。要注意这一格式的转换会稍微牺牲一点性能。

(3) 设置输出委托为self，因为类实现了AVCaptureVideoDataOutputSampleBufferDelegate协议，还在应该调用的委托方法上指定了调度队列。为queue参数指定的调度队列必须是一个序列队列。本例中，我们使用主队列，因为主队列最适合应用程序，不过在许多其他案例中，都是指定专门的视频处理队列。

(4) 最后，执行标准测试来确认输出是否可以被添加到会话，如果可以，则添加到会话并返回YES。如果测试失败，则返回NO。

目前为止，这和我们之前的示例看起来很像。让我们继续并开始关注有关OpenGL ES整合点的一些知识。有关OpenGL ES的知识虽然不在本书谈论的范围内，不过作为AV Foundation开发人员，会发现在很多高性能的视频应用程序案例中OpenGL ES都是唯一提供控制和功能的可行解决方案。所以我们将AV Foundation基础框架所提供的工具视作与OpenGL ES之间的桥梁。想要找到完整的OpenGL ES 2.0介绍，建议阅读Erik Buck撰写的*Learning OpenGL ES for iOS*一书(2012, Boston: Addison-Wesley)。

Core Video提供了一个对象类型CVOpenGLESTextureCache，作为Core Video像素buffer和OpenGL ES贴图之间的桥梁。缓存的目的是减少数据从CPU转移到GPU(也可能是反向)的开销。让我们从创建贴图缓存开始吧，如代码清单7-29所示。

代码清单 7-29　创建 OpenGLESTextureCache

```
@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate>

@property (weak, nonatomic) EAGLContext *context;
@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;

@property (nonatomic) CVOpenGLESTextureCacheRef textureCache;              // 1
@property (nonatomic) CVOpenGLESTextureRef cameraTexture;

@end

@implementation THCameraController

- (instancetype)initWithContext:(EAGLContext *)context {
    self = [super init];
    if (self) {
        _context = context;
        CVReturn err = CVOpenGLESTextureCacheCreate(kCFAllocatorDefault,     // 2
                                                    NULL,
                                                    _context,
                                                    NULL,
                                                    &_textureCache);
        if (err != kCVReturnSuccess) {                                       // 3
            NSLog(@"Error creating texture cache. %d", err);
        }
    }
    return self;

}
```

(1) 添加两个新属性。第一个是为CVOpenGLESTextureCacheRef自己添加，第二个是为在AVCaptureVideoDataOutput委托回调中创建的CVOpenGLESTextureRef 对象添加。

(2) 使用CVOpenGLESTextureCacheCreate函数创建一个新的缓存实例。这个函数的关键参数是后备EAGLContext和textureCache指针。

(3) 检查这个函数的返回值是个好主意。本例中，如果返回值不等于kCVReturnSuccess，则在控制台打印错误信息。在实际的应用程序中我们建议这一块的处理应该更加完善。

创建好贴图缓存后，我们就可以继续学习实际创建一些贴图了。使用CVPixelBuffer创建

贴图的函数称为CVOpenGLESTextureCacheCreateTextureFromImage。这个函数将在委托回调中使用，为每个视频帧创建一个新的贴图。如代码清单7-30所示。

代码清单 7-30 创建 OpenGL ES 贴图

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
      fromConnection:(AVCaptureConnection *)connection {

   CVReturn err;
   CVImageBufferRef pixelBuffer =                                          // 1
      CMSampleBufferGetImageBuffer(sampleBuffer);

   CMFormatDescriptionRef formatDescription =                              // 2
      CMSampleBufferGetFormatDescription(sampleBuffer);
   CMVideoDimensions dimensions =
      CMVideoFormatDescriptionGetDimensions(formatDescription);

   err = CVOpenGLESTextureCacheCreateTextureFromImage(kCFAllocatorDefault, // 3
                                                      _textureCache,
                                                      pixelBuffer,
                                                      NULL,
                                                      GL_TEXTURE_2D,
                                                      GL_RGBA,
                                                      dimensions.height,
                                                      dimensions.height,
                                                      GL_BGRA,
                                                      GL_UNSIGNED_BYTE,
                                                      0,
                                                      &_cameraTexture);

   if (!err) {
      GLenum target = CVOpenGLESTextureGetTarget(_cameraTexture);          // 4
      GLuint name = CVOpenGLESTextureGetName(_cameraTexture);
      [self.textureDelegate textureCreatedWithTarget:target name:name];    // 5
   } else {
      NSLog(@"Error at CVOpenGLESTextureCacheCreateTextureFromImage %d", err);
   }

   [self cleanupTextures];
}

- (void)cleanupTextures {                                                  // 6
   if (_cameraTexture) {
      CFRelease(_cameraTexture);
      _cameraTexture = NULL;
   }
   CVOpenGLESTextureCacheFlush(_textureCache, 0);

}
```

(1) 首先从捕捉到的CMSampleBuffer获取基础CVImageBuffer(CVImageBuffer是CVPixelBufferRef的类型定义)。

(2) 获取 CMSampleBuffer 的 CMFormatDescription。用CMVideoFormatDescriptionGetDimensions函数通过格式描述信息来获取视频帧的维度。它会返回一个带有宽和高信息的CMVideoDimensions结构。

(3) 使用CVOpenGLESTextureCacheCreateTextureFromImage函数从CVPixelBuffer创建一个OpenGL ES贴图。会注意到我们将dimensions.height作为宽和高两个参数的值。这并不是印刷错误，不过的确有点欺骗性，因为我们希望在水平方向上剪辑视频，所以这是一个完美矩形。还有另一种方法可以实现这样的裁剪，不过我们所用的这个方法既简单又合适。

(4) 从CVOpenGLESTextureRef获取target和name的值。它们用来将贴图对象与旋转的小方块的表面进行合适的绑定。

(5) 调用委托的textureCreatedWithTarget:name方法，这里委托函数会执行实际的GL贴图绑定。

(6) 最后，调用私有cleanupTextures来释放贴图并刷新贴图缓存。

应用程序已经全部开发完毕并准备试运行。运行应用程序会看到出现一个不断旋转的小方块，它的表面图案就是捕捉到的视频帧。这虽然是一个简单应用程序，不过它很好地将AV Foundation和OpenGL ES整合在一起，开发者会发现很多时候这两个技术都在同时使用。

AVCaptureVideoDataOutput定义了在视频帧被捕捉时访问该帧的接口。它提供了对呈现的数据和处理中的数据全方面的控制，让开发者可以创建出非常强大的视频应用。下一章在介绍如何记录自定义视频处理结果时还会提到这个话题。

7.7 小结

本章我们学习了许多功能和特性来润色和改进我们的应用程序。虽然我们都是孤立地在使用这些功能，不过在实际中会发现这些功能效果不错，可以很好地用在单一应用程序中。第6章和第7章介绍的这些功能和特性都是我们创建下一代精彩捕捉类应用程序的工具。

7.8 挑战

将本章学到的一些功能整合到第6章的Kamera应用程序。使用iOS的Camera应用程序作为参照来查看如何使用缩放和人脸检测功能，同时为了得到更好的效果开启高帧率捕捉功能。同样，勇于实验并放飞自己的创意吧！

第8章

读取和写入媒体

AV Foundation为处理专门的A/V任务定义了许很高级功能(比如播放、捕捉和编辑等任务)。这些高级特性也是AV Foundation成为一个强大框架的原因；开发者不需要了解媒体处理的底层知识就可以创建出功能强大的媒体应用程序。不过有时只使用框架的高级功能不能满足开发者的需求，我们还是希望能够对这一功能进行更深入的挖掘。本章介绍该框架提供的底层读写功能，并讨论在创建AV Foundation应用程序时的一些常见使用场景。

8.1 综述

AV Foundation定义了一组功能可以用于创建媒体应用程序时遇到的大部分用例场景。在使用框架时，开发者首选的方法还是使用专门的高级功能，不过随着我们开发更复杂的媒体应用程序，就可能遇到一些用例，它们需要的功能并不受AV Foundation框架的内置支持。这意味着我们不够幸运并需要到其他地方寻找解决方案吗？不，这时需要使用框架的AVAssetReader和AVAssetWriter类提供的低级功能(如图8-1所示)。这些类可以让开发者直接处理媒体样本，打开了机会之窗。

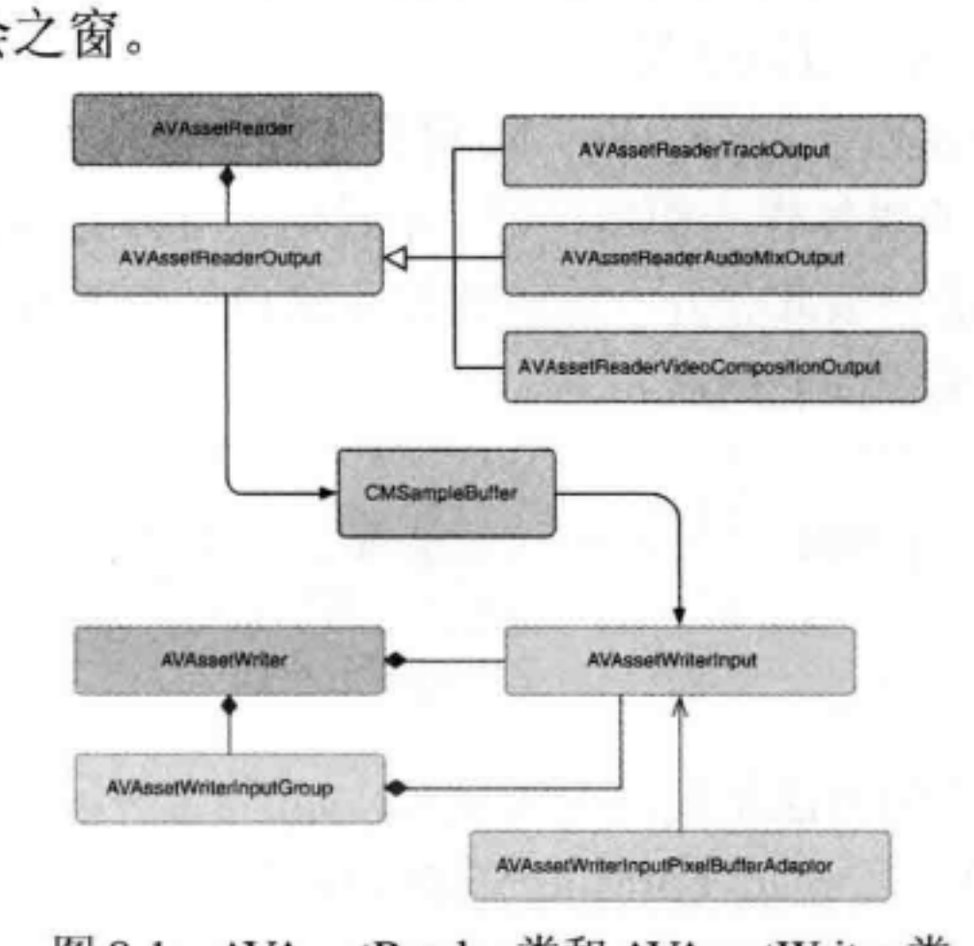

图 8-1　AVAssetReader 类和 AVAssetWriter 类

8.1.1 AVAssetReader

AVAssetReader用于从AVAsset实例中读取媒体样本。通常会配置一个或多个AVAsset-ReaderOutput实例，并通过copyNextSampleBuffer方法可以访问音频样本和视频帧。AVAssetReaderOutput是一个抽象类，不过框架定义了3个具体实例来从指定的AVAssetTrack中读取解码的媒体样本，从多音频轨道中读取混合输出，或者从多视频轨道中读取组合输出。一个资源读取器的内部通道都是以多线程的方式不断提取下一个可用样本的，这样可以在系统请求资源时最小化时延。尽管提供了低时延的检索操作，还是不倾向于实时操作，比如播放。

> **注意：**
>
> AVAssetReader只针对带有一个资源的媒体样本。如果需要同时从多个基于文件的资源中读取样本，可将它们组合到一个AVAsset子类AVComposition中，下一章我们会讲到相关内容。

8.1.2 AVAssetWriter

AVAssetWriter是AVAssetReader对应的兄弟类，它用于对媒体资源进行编码并将其写入到容器文件中，比如一个MPEG-4文件或一个QuickTime文件。它由一个或多个AVAssetWriterInput对象配置，用于附加将包含要写入容器的媒体样本的CMSampleBuffer对象。AVAssetWriterInput被配置为可以处理指定的媒体类型，比如音频或视频，并且附加在其后的样本会在最终输出时生成一个独立的AVAssetTrack。当使用一个配置了处理视频样本的AVAssetWriterInput时，开发者会经常用到一个专门的适配器对象AVAssetWriterInput-PixelBufferAdaptor。这个类在附加被包装为CVPixelBuffer对象的视频样本时提供最优性能。输入信息也可以通过使用AVAssetWriterInputGroup组成互斥的参数。这就让开发者能够创建特定资源，其中包含在播放时使用AVMediaSelectionGroup和AVMediaSelectionOption类选择的指定语言媒体轨道，第4章介绍过这些类。

AVAssetWriter可以自动支持交叉媒体样本。将样本写入磁盘的一种方式是按照顺序写入，如图8-2所示。需要将所有的媒体样本都捕捉好，不过这会导致数据的低效率排列，因为本应该整体呈现的样本数据可能会彼此分开。这就使得存储设备更难有效地读取数据，并在播放和寻找资源时产生负面效果和性能问题。

图 8-2 不交错的样本布局

一种更好安排这些样本的方法是使用交错模式，如图8-3所示。为保持一个合适的交错模式，AVAssetWriterInput提供一个readyForMoreMediaData属性来指示在保持所需的交错情况下输入信息是否还可以附加更多数据。只有在这个属性值为YES时才可以将一个新的样本添加

到写入输入信息中。

图 8-3　交错的样本布局

AVAssetWriter可用于实时操作和离线操作两种情况，不过对于每个场景都有不同的方法将样本buffer添加到写入对象的输入中。

- **实时**：当处理实时资源时，比如从 AVCaptureVideoDataOutput 写入捕捉的样本时，AVAssetWriterInput 应该令 expectsMediaDataInRealTime 属性为 YES 来确保 readyForMoreMediaData 值被正确计算。从实时资源写入数据优化了写入器，这样一来，与维持理想交错效果相比，快速写入样本具有更高的优先级。这一优化效果不错，视频和音频样本以大致相同的速率捕捉，传入数据自然交错。
- **离线**：当从离线资源中读取媒体资源时，比如从 AVAssetReader 读取样本 buffer，在附加样本前仍然需要观察写入器输入的 readyForMoreMediaData 属性的状态，不过可以使用 requestMediaDataWhenReadyOnQueue:usingBlock:方法控制数据的提供。传到这个方法中的代码块会随写入器输入准备附加更多的样本而不断被调用，添加样本时开发者需要检索数据并从资源中找到下一个样本进行添加。

8.1.3　读写示例

下面通过一个基础示例学习在一个离线场景下如何使用AVAssetReader和AVAssetWriter。在示例中，我们用AVAssetReader从资源的视频轨道读取样本，并使用AVAssetWriter将它们写入到一个新的QuickTime电影文件中。虽然这是一个虚构的示例，不过它展示了在同时使用这些类时所涉及的一些基本步骤。下面首先来设置和配置AVAssetReader。

```
AVAsset *asset = // Asynchronously loaded video asset
AVAssetTrack *track =
    [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

self.assetReader =
    [[AVAssetReader alloc] initWithAsset:asset error:nil];

NSDictionary *readerOutputSettings = @{
    (id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA)
};

AVAssetReaderTrackOutput *trackOutput =
    [[AVAssetReaderTrackOutput alloc] initWithTrack:track
                                    outputSettings:readerOutputSettings];

[self.assetReader addOutput:trackOutput];

[self.assetReader startReading];
```

首先创建一个新的AVAssetReader，传递读取的AVAsset实例。创建一个AVAssetReader-

TrackOutput从资源的视频轨道中读取样本，将视频帧解压缩为BGRA格式。添加输出到读取器，并调用startReading方法开始读取过程。

接下来创建并配置AVAssetWriter。

```
NSURL *outputURL = // Destination output URL

self.assetWriter = [[AVAssetWriter alloc] initWithURL:outputURL
                                        fileType:AVFileTypeQuickTimeMovie
                                          error:nil];

NSDictionary *writerOutputSettings = @{
    AVVideoCodecKey: AVVideoCodecH264,
    AVVideoWidthKey: @1280,
    AVVideoHeightKey: @720,
    AVVideoCompressionPropertiesKey: @{
        AVVideoMaxKeyFrameIntervalKey: @1,
        AVVideoAverageBitRateKey: @10500000,
        AVVideoProfileLevelKey: AVVideoProfileLevelH264Main31,
    }
};

AVAssetWriterInput *writerInput =
    [[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeVideo
                                outputSettings:writerOutputSettings];

[self.assetWriter addInput:writerInput];

[self.assetWriter startWriting];
```

示例中创建了一个新的AVAssetWriter对象，并传递了一个新文件写入目的地的输出URL和所希望的文件类型。创建了一个新的AVAssetWriterInput，它带有相应的媒体类型和输出设置，以便创建一个720p H.264格式的视频。将输入添加到写入器并调用startWriting方法。

注意：

与 AVAssetExportSession 相比，AVAssetWriter 明显的优势就是它对输出进行编码时能够进行更加细致的压缩设置控制。可以让开发者指定诸如关键帧间隔、视频比特率、H.264配置文件、像素宽高比和纯净光圈等设置。

在完成AVAssetReader和AVAssetWriter对象的设置后，是时候创建一个新的写入会话来从资源中读取样本并将它们写入到新位置。示例中使用的是拉模式(pull model)，即当写入器输入准备附加更多的样本时从资源中拉取样本。这是当我们从一个非实时资源中写入样本时所使用的模式。

```
// Serial Queue
dispatch_queue_t dispatchQueue =
    dispatch_queue_create("com.tapharmonic.WriterQueue", NULL);

[self.assetWriter startSessionAtSourceTime:kCMTimeZero];
[writerInput requestMediaDataWhenReadyOnQueue:dispatchQueue usingBlock:^{
```

```
BOOL complete = NO;

while ([writerInput isReadyForMoreMediaData] && !complete) {

    CMSampleBufferRef sampleBuffer = [trackOutput copyNextSampleBuffer];

    if (sampleBuffer) {
        BOOL result = [writerInput appendSampleBuffer:sampleBuffer];
        CFRelease(sampleBuffer);
        complete = !result;
    } else {
        [writerInput markAsFinished];
        complete = YES;
    }
}

if (complete) {
    [self.assetWriter finishWritingWithCompletionHandler:^{
        AVAssetWriterStatus status = self.assetWriter.status;
        if (status == AVAssetWriterStatusCompleted) {
            // Handle success case
        } else {
            // Handle failure case
        }
    }];
}
}];
```

示例中首先使用startSessionAtSourceTime:方法创建了一个新的写入会话，并传递kCMTimeZero参数作为资源样本的开始时间。传给requestMediaDataWhenReadyOnQueue:usingBlock:方法的代码块在写入器输入准备好添加更多样本时会不断被调用。在每次调用期间，输入准备添加更多数据时，再从轨道的输出中复制可用的样本，并将其附加到输入中。当所有样本都从轨道输出中复制后，需要标记AVAssetWriterInput已结束并指明添加操作已经完成。最后，调用finishWritingWithCompletionHandler:关闭写入会话。资源写入器的status属性可在completion handler中查询，来确定写入会话是否成功完成、失败或取消。

现在我们对AVAssetReader和AVAssetWriter这两个类有了更多了解。上述示例中已经给出了我们使用这些类处理离线资源时所使用的基础模式。下面继续学习并讨论一些更加具体、真实的情况，这样会更好地理解AVAssetReader和AVAssetWriter的价值。

8.2　创建音频波形视图

对于大部分音频和视频应用程序，一个常见的需求就是提供图像化显示的音频波形(waveform)，如图8-4所示。这个功能可以让用户更简单地查看音频轨道，也就更容易操作滑动条或选定希望的位置进行编辑。本节将讨论如何使用AV Foundation来实现这个功能。

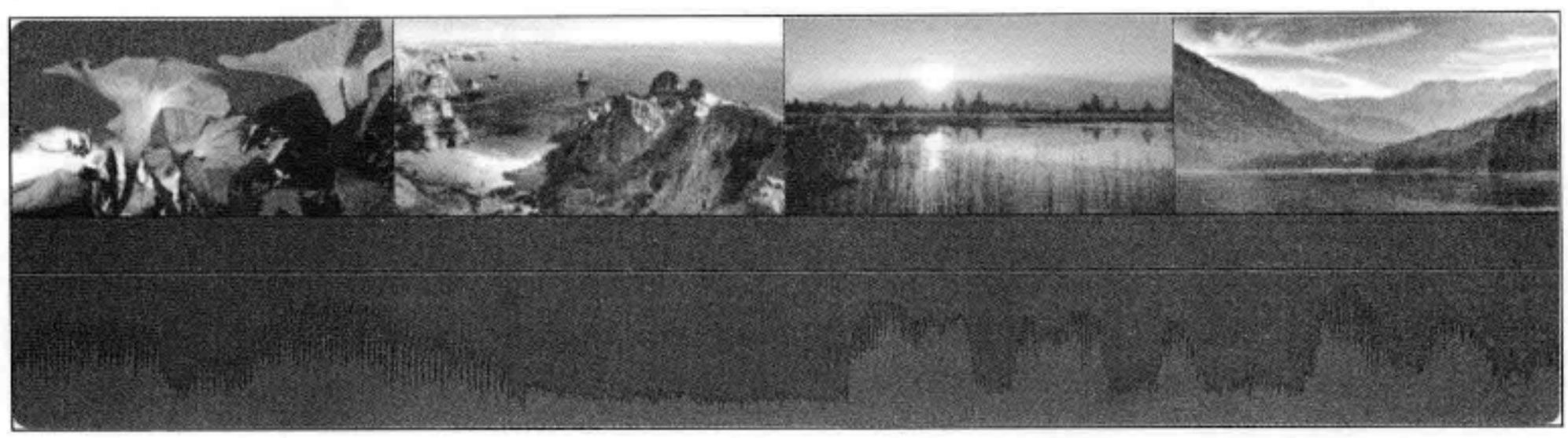

图 8-4 Final Cut Pro X 示波形展示

绘制一个波形的基本技巧包括以下三个步骤：

(1) **读取**：第一步是读取音频样本进行渲染。需要读取或可能解压音频数据，比如对于线性PCM资源。回顾第1章所讲到的，线性PCM是一种未压缩的音频样本格式。

(2) **缩减**：实际读取到的样本数量要远比我们在屏幕上渲染的多。考虑到单声道音频文件是以44.1kHz比率进行采样的，所得到的样本要比我们具有的像素多得多。缩减的过程必须作用于这个样本集。这一过程通常包括将样本总量分为小的样本块，并在每个样本块上找到最大的样本、所有样本的平均值或min/max值。

(3) **渲染**：在这一步我们将缩减后的样本呈现在屏幕上。通常会用到Quartz框架，不过也可以使用任何苹果公司支持的绘图框架。如何绘制这些数据的类型取决于开发者是如何缩减样本的。如果采用min/max对，则为它的每一对绘制一条垂线。如果使用每个样本块的平均值或最大值，会发现使用Quartz Bezier路径绘制波形是最合适的。

在Chapter 8目录下可以找到名为THWaveformView_Starter的示例项目。这里我们创建了一个UIView的子类来展示如图8-5所示的波形效果。下面从第一步开始吧。

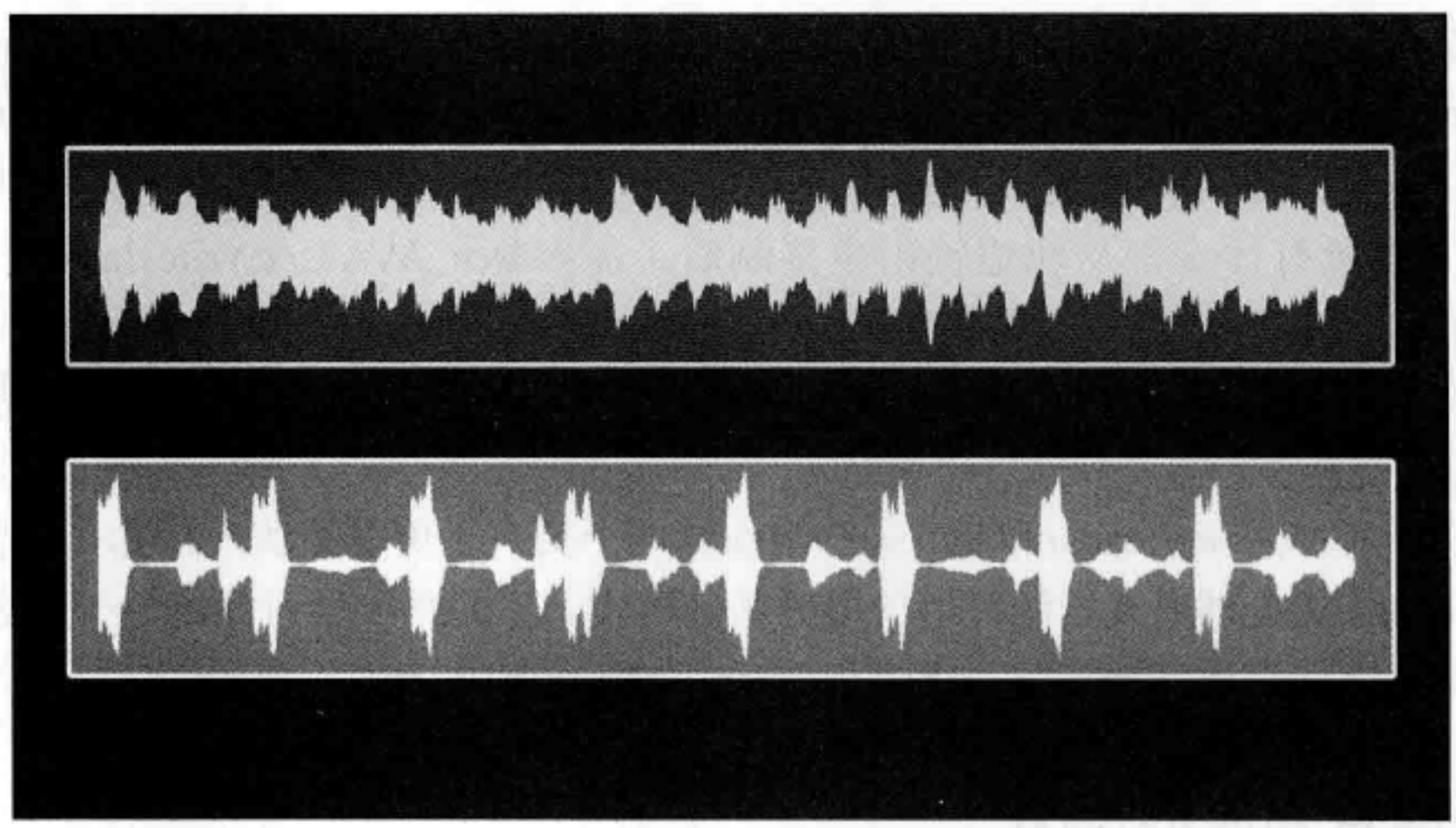

图 8-5 THWaveformView 的实际应用

8.2.1 读取音频样本

要创建的第一个类称为THSampleDataProvider，这个类使用AVAssetReader实例从AVAsset中读取音频样本并返回一个NSData对象。代码清单8-1给出了这个类的接口。

代码清单 8-1　THSampleDataProvider 接口

```
#import <AVFoundation/AVFoundation.h>

typedef void(^THSampleDataCompletionBlock)(NSData *);

@interface THSampleDataProvider : NSObject

+ (void)loadAudioSamplesFromAsset:(AVAsset *)asset
                 completionBlock:(THSampleDataCompletionBlock)completionBlock;

@end
```

这个类的接口最直接的关注点就是loadAudioSamplesFromAsset:completionBlock:类方法，调用这个方法可以读取音频样本。下面看一下该类的具体实现，如代码清单8-2所示。

代码清单 8-2　THSampleDataProvider 实现

```
#import "THSampleDataProvider.h"

@implementation THSampleDataProvider

+ (void)loadAudioSamplesFromAsset:(AVAsset *)asset
                 completionBlock:(THSampleDataCompletionBlock)completionBlock {

    NSString *tracks = @"tracks";

    [asset loadValuesAsynchronouslyForKeys:@[tracks] completionHandler:^{   // 1

        AVKeyValueStatus status = [asset statusOfValueForKey:tracks error:nil];

        NSData *sampleData = nil;

        if (status == AVKeyValueStatusLoaded) {                             // 2
            sampleData = [self readAudioSamplesFromAsset:asset];
        }

        dispatch_async(dispatch_get_main_queue(), ^{                        // 3
            completionBlock(sampleData);
        });
    }];

}

+ (NSData *)readAudioSamplesFromAsset:(AVAsset *)asset {

    // To be implemented

    return nil;

}

@end
```

(1) 首先对资源所需的键执行标准的异步载入操作，这样在访问资源的tracks属性时就不会遇到阻碍。

(2) 如果tracks键成功载入，则调用私有方法readAudioSamplesFromAsset:从资源音频轨道中读取样本。

(3) 由于载入操作可能发生在任意后台队列上，所以我们希望调度回主队列，并调用带有检索到的音频样本的completion block，如果没有读取成功则为nil。

现在我们讨论readAudioSamplesFromAsset:方法的实现。上一章讨论了使用CMSampleBuffer访问AVCaptureVideoDataOutput对象渲染的视频帧。当我们处理未压缩的视频数据时，可以使用CMSampleBufferGetImageBuffer函数检索包含有帧的像素信息的基础CVImageBufferRef。从一个资源中读取音频样本时，会再次用到CMSampleBuffer，不过本例基础数据将会以Core Media类型CMBlockBuffer的形式提供。Block buffer用于在Core Media通道中传送任意字节的数据。根据开发者对音频样本的使用目的，有多种方法可以访问带有音频数据的block buffer。使用CMSampleBufferGetDataBuffer函数得到一个到block buffer的不可保留(unretained)引用，这个方法适用于我们只需要访问数据而不进行后续处理的情况。相反，如果需要对音频数据进行处理，比如把它传递到Core Audio，可以使用CMSampleBufferGetAudioBufferListWithRetainedBlockBuffer函数将数据作为AudioBufferList访问。这会返回Core Audio的AudioBufferList，并带有保留的CMBlockBuffer用于管理所包含的样本的生命周期。由于我们已经检索了样本并将它们复制到NSData中，就可以使用之前的方法检索音频数据了，如代码清单8-3所示。

代码清单 8-3 读取资源的音频样本

```
+ (NSData *)readAudioSamplesFromAsset:(AVAsset *)asset {

    NSError *error = nil;

    AVAssetReader *assetReader =                                          // 1
        [[AVAssetReader alloc] initWithAsset:asset error:&error];

    if (!assetReader) {
        NSLog(@"Error creating asset reader: %@", [error localizedDescription]);
        return nil;
    }

    AVAssetTrack *track =                                                 // 2
        [[asset tracksWithMediaType:AVMediaTypeAudio] firstObject];

    NSDictionary *outputSettings = @{                                     // 3
        AVFormatIDKey             : @(kAudioFormatLinearPCM),
        AVLinearPCMIsBigEndianKey  : @NO,
        AVLinearPCMIsFloatKey      : @NO,
        AVLinearPCMBitDepthKey     : @(16)
    };

    AVAssetReaderTrackOutput *trackOutput =                               // 4
        [[AVAssetReaderTrackOutput alloc] initWithTrack:track
                                    outputSettings:outputSettings];

    [assetReader addOutput:trackOutput];

    [assetReader startReading];
```

```
NSMutableData *sampleData = [NSMutableData data];

while (assetReader.status == AVAssetReaderStatusReading) {

    CMSampleBufferRef sampleBuffer = [trackOutput copyNextSampleBuffer]; // 5

    if (sampleBuffer) {

        CMBlockBufferRef blockBufferRef =                                 // 6
            CMSampleBufferGetDataBuffer(sampleBuffer);

        size_t length = CMBlockBufferGetDataLength(blockBufferRef);
        SInt16 sampleBytes[length];

        CMBlockBufferCopyDataBytes(blockBufferRef,                        // 7
                                   0,
                                   length,
                                   sampleBytes);

        [sampleData appendBytes:sampleBytes length:length];

        CMSampleBufferInvalidate(sampleBuffer);                           // 8
        CFRelease(sampleBuffer);
    }
}

if (assetReader.status == AVAssetReaderStatusCompleted) {                 // 9
    return sampleData;
} else {
    NSLog(@"Failed to read audio samples from asset");
    return nil;
}
}
```

(1) 创建一个新的AVAssetReader实例，并赋给它一个资源来读取。如果在初始化对象时出错，就将这个错误信息打印到控制台并返回nil。

(2) 获取资源中找到的第一个音频轨道。包含在示例项目中的音频文件只含有一个轨道，不过最好总是根据期望的媒体类型获取轨道。

(3) 创建一个NSDictionary来保存从资源轨道读取音频样本时使用的解压设置。样本需要以未压缩的格式被读取，所有我们指定kAudioFormatLinearPCM作为格式键。我们还希望确保以16位、little-endian字节顺序的有符号整型方式读取。这些设置对于示例项目已经足够了，不过我们可以在AVAudioSettings.h文件中找到许多额外的键，它们可以对格式转换进行更详细的控制。

(4) 创建一个新的AVAssetReaderTrackOutput实例，并将上一步我们创建的输出设置传递给它。将其作为AVAssetReader的输出并调用startReading来允许资源读取器开始预收取样本数据。

(5) 调用跟踪输出的copyNextSampleBuffer方法开始每个迭代，每次都返回一个包含音频样本的下一个可用样本buffer。

(6) CMSampleBuffer中的音频样本被包含在一个CMBlockBuffer类型中。使用

CMSampleBufferGetDataBuffer函数可以访问这个block buffer。使用CMBlockBufferGetDataLength函数确定其长度并创建一个16位的带符号整型数组来保存这些音频样本。

(7) 使用CMBlockBufferCopyDataBytes函数生成一个数组，数组中的元素为CMBlockBuffer所包含的数据，并将数组的内容附加在NSData实例后。

(8) 用CMSampleBufferInvalidate函数来指定样本buffer已经处理和不可再继续使用。此外，需要释放CMSampleBuffer副本来释放内容。

(9) 如果资源读取器的status值等于AVAssetReaderStatusCompleted，则数据被成功读取，返回包含音频样本数据的NSData即可。如果出现错误，则返回nil。

第一步就这样完成了，现在我们学会了如何从不同的音频格式中成功读取音频样本。下一步就是对数据进行缩减以满足可在屏幕上进行绘制的要求。

8.2.2 缩减音频样本

THSampleDataProvider将从一个给定的视频资源中提取全部的样本集合。即使是非常小的音频文件，都可能有数十万个样本，远大于在屏幕上进行绘制所需的样本。我们需要定义一个筛选方法来得到最终在屏幕上呈现的值集合。要实现这一缩减操作，需要创建一个THSampleDataFilter对象。这个类的接口如代码清单8-4所示。

代码清单 8-4 THSampleDataFilter 接口

```
@interface THSampleDataFilter : NSObject

- (id)initWithData:(NSData *)sampleData;

- (NSArray *)filteredSamplesForSize:(CGSize)size;

@end
```

用一个带有音频样本信息的NSData来初始化这个类的实例。提供filteredSamplesForSize:方法按照指定的尺寸约束来筛选数据集。

共分两步来处理这个数据。首先将样本分成“箱”，找到每个箱里面的最大样本。当所有箱都处理完成后，对这些与传递给filteredSamplesForSize:方法的尺寸约束有关的样本应用比例因子。下面看下这个类的具体实现，如代码清单8-5所示。

代码清单 8-5 THSampleDataFilter 实现

```
#import "THSampleDataFilter.h"

@interface THSampleDataFilter ()
@property (nonatomic, strong) NSData *sampleData;
@end

@implementation THSampleDataFilter

- (id)initWithData:(NSData *)sampleData {
    self = [super init];
    if (self) {
        _sampleData = sampleData;
    }
```

```
    return self;
}

- (NSArray *)filteredSamplesForSize:(CGSize)size {

    NSMutableArray *filteredSamples = [[NSMutableArray alloc] init];          // 1
    NSUInteger sampleCount = self.sampleData.length / sizeof(SInt16);
    NSUInteger binSize = sampleCount / size.width;

    SInt16 *bytes = (SInt16 *)self.sampleData.bytes;

    SInt16 maxSample = 0;

    for (NSUInteger i = 0; i < sampleCount; i += binSize) {

        SInt16 sampleBin[binSize];

        for (NSUInteger j = 0; j < binSize; j++) {                            // 2
            sampleBin[j] = CFSwapInt16LittleToHost(bytes[i + j]);
        }

        SInt16 value = [self maxValueInArray:sampleBin ofSize:binSize];       // 3
        [filteredSamples addObject:@(value)];

        if (value > maxSample) {                                              // 4
            maxSample = value;
        }
    }

    CGFloat scaleFactor = (size.height / 2) / maxSample;

    for (NSUInteger i = 0; i < filteredSamples.count; i++) {                  // 5
        filteredSamples[i] = @([filteredSamples[i] integerValue] * scaleFactor);
    }

    return filteredSamples;
}

- (SInt16)maxValueInArray:(SInt16[])values ofSize:(NSUInteger)size {
    SInt16 maxValue = 0;
    for (int i = 0; i < size; i++) {
        if (abs(values[i]) > maxValue) {
            maxValue = abs(values[i]);
        }
    }
    return maxValue;
}

@end
```

(1) 首先创建一个NSMutableArray来保存筛选的音频样本数组。我们还确定了要处理的样本总数并计算与传入方法的尺寸约束相应的“箱”尺寸值。一个箱包含一个需要被筛选的样本子集。

(2) 迭代全部音频样本集合，在每个迭代中构建一个需要处理的数据箱。当处理音频样

本时，要时刻记得字节的顺序，所以用到了CFSwapInt16LittleToHost函数来确保样本是按主机内置的字节顺序处理的。

(3) 对于每个箱，调用maxValueInArray:方法找到最大样本。这个方法会迭代箱中的所有样本并找到最大绝对值。结果值被添加到filteredSamples数组。

(4) 当我们遍历所有音频样本时，在筛选结果中计算最大值。这个值作为我们筛选样本所使用的比例因子。

(5) 在返回筛选样本前，需要相对于传递给方法的尺寸约束来缩放值。这会得到一个浮点值的数组，这些值可以在屏幕上呈现。当这些值完成缩放后，就可以将数组返回给调用方法。

THSampleDataFilter类完成了，下面我们准备讨论如何创建视图来渲染这些音频样本。

8.2.3 渲染音频样本

我们需要创建一个UIView子类来渲染结果。下面先来看这个类的接口，如代码清单8-6所示。

代码清单 8-6 THWaveformView 接口

```
@class AVAsset;

@interface THWaveformView : UIView

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) UIColor *waveColor;

@end
```

视图提供了一个简单接口来设置AVAsset和波形绘制时所用的颜色。下面看下这个类的具体实现，如代码清单8-7所示。一些UIView样板代码被我们省略掉了。要查看全部的实现代码，可以去看项目源代码。

代码清单 8-7 实现 setAsset:方法

```
#import "THWaveformView.h"
#import "THSampleDataProvider.h"
#import "THSampleDataFilter.h"
#import <QuartzCore/QuartzCore.h>

static const CGFloat THWidthScaling = 0.95;
static const CGFloat THHeightScaling = 0.85;

@interface THWaveformView ()
@property (strong, nonatomic) THSampleDataFilter *filter;
@property (strong, nonatomic) UIActivityIndicatorView *loadingView;
@end

@implementation THWaveformView

...

- (void)setAsset:(AVAsset *)asset {
   if (_asset != asset) {
```

```
    _asset = asset;

    [THSampleDataProvider loadAudioSamplesFromAsset:self.asset                    // 1
                                  completionBlock:^(NSData *sampleData) {

        self.filter =                                                             // 2
            [[THSampleDataFilter alloc] initWithData:sampleData];

        [self.loadingView stopAnimating];                                         // 3
        [self setNeedsDisplay];
    }];
    }
}

- (void)drawRect:(CGRect)rect {
    // To be implemented
}

@end
```

(1) 首先调用THSampleDataProvider类的loadAudioSamplesFromAsset:completionBlock:方法开始载入音频样本。

(2) 当样本被载入后，构建一个新的THSampleDataFilter实例，为其传递包含音频样本的NSData。

(3) 将视图的载入图标移除并调用setNeedsDisplay来对视图进行清理，此时会调用drawRect:方法。

下面看一下drawRect方法的实现，学习数据是如何在屏幕上绘制的，如代码清单8-8所示。

代码清单 8-8　实现 drawRect:方法

```
- (void)drawRect:(CGRect)rect {

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextScaleCTM(context, THWidthScaling, THHeightScaling);              // 1

    CGFloat xOffset = self.bounds.size.width -
                  (self.bounds.size.width * THWidthScaling);

    CGFloat yOffset = self.bounds.size.height -
                  (self.bounds.size.height * THHeightScaling);

    CGContextTranslateCTM(context, xOffset / 2, yOffset / 2);

    NSArray *filteredSamples =                                                // 2
        [self.filter filteredSamplesForSize:self.bounds.size];

    CGFloat midY = CGRectGetMidY(rect);

    CGMutablePathRef halfPath = CGPathCreateMutable();                        // 3
    CGPathMoveToPoint(halfPath, NULL, 0.0f, midY);

    for (NSUInteger i = 0; i < filteredSamples.count; i++) {
        float sample = [filteredSamples[i] floatValue];
        CGPathAddLineToPoint(halfPath, NULL, i, midY - sample);
```

```
    }

    CGPathAddLineToPoint(halfPath, NULL, filteredSamples.count, midY);

    CGMutablePathRef fullPath = CGPathCreateMutable();                    // 4
    CGPathAddPath(fullPath, NULL, halfPath);

    CGAffineTransform transform = CGAffineTransformIdentity;              // 5
    transform = CGAffineTransformTranslate(transform, 0, CGRectGetHeight(rect));
    transform = CGAffineTransformScale(transform, 1.0, -1.0);
    CGPathAddPath(fullPath, &transform, halfPath);

    CGContextAddPath(context, fullPath);                                  // 6
    CGContextSetFillColorWithColor(context, self.waveColor.CGColor);
    CGContextDrawPath(context, kCGPathFill);

    CGPathRelease(halfPath);                                              // 7
    CGPathRelease(fullPath);
}
```

(1) 我们希望在视图内呈现这个波形，所以首先基于定义的高和宽常量来缩放图像上下文。还要计算x和y偏移量，转换上下文，在缩放上下文中适当调整偏移。

(2) 从THSampleDataFilter实例获取筛选的样本，并传递视图边界的尺寸。在实际代码中，开发者可能希望在drawRect:方法之外执行这一检索操作，这样在筛选样本时会有更好的优化效果。不过在本例中上面的方法已经足以满足要求了。

(3) 创建一个新的CGMutablePathRef，用来绘制波形Bezier路径的上半部。迭代筛选的样本，对每次迭代调用CGPathAddLineToPoint向路径中添加一个点。利用循环索引作为x坐标，样本值作为y坐标。

(4) 创建第二个CGMutablePathRef，传递第4步构建的Bezier路径。使用这个Bezier路径绘制完整的波形。

(5) 要绘制波形的下半部，需要对上半部路径应用translate和scale变化。这会使得上半部路径翻转到下面，填满整个波形。

(6) 将完整的路径添加到图像上下文，根据指定的waveColor值设置填充颜色，并调用CGContextDrawPath(context, kCGPathFill)将填充好的路径绘制到图像上下文。

(7) 每当创建Quartz对象后，开发者都有责任在使用完之后释放相应的内存，所以最后一步就是在创建的路径对象上调用CGPathRelease。

应用程序的视图控制器用来绘制两个视图，如图8-5所示。可以打开应用程序的视图控制器并尝试其他颜色，或在storyboard中修改视图。现在，如果在应用程序中需要用到波形器，就可以重用我们本章的类了。

8.3 捕捉录制的高级方法

上一章最后讨论了将AVCaptureVideoDataOutput捕捉的CVPixelBuffer对象作为OpenGL ES的贴图来呈现。这是一个非常强大的功能，可以使我们开发出很多有趣的应用程序。不过使用AVCaptureVideoDataOutput的一个问题在于会失去AVCaptureMovieFileOutput来记录输出

的便捷性。无论多么神奇的特效，如果应用程序无法记录输出内容，并把它分享到全世界，那都是严重的缺陷。在本节中，我们会学到如何使用AVAssetWriter创建一个与AVCapture-MovieFileOutput类似的可重用类从高级捕捉输出中记录输出。

看一下上一章介绍的有关OpenGL ES的应用。我们可以重用CubeKamera应用程序，不过为了避免重复，我们创建一个新的相机应用程序，使用Core Image框架来处理视频帧应用实时视频效果。

在Chapter 8目录中可以找到名为KameraWriter_Starter的示例项目。KameraWriter是一个带有实时效果的视频录制应用程序。用户可从屏幕最上的筛选器选择可用的效果。如图8-6所示。

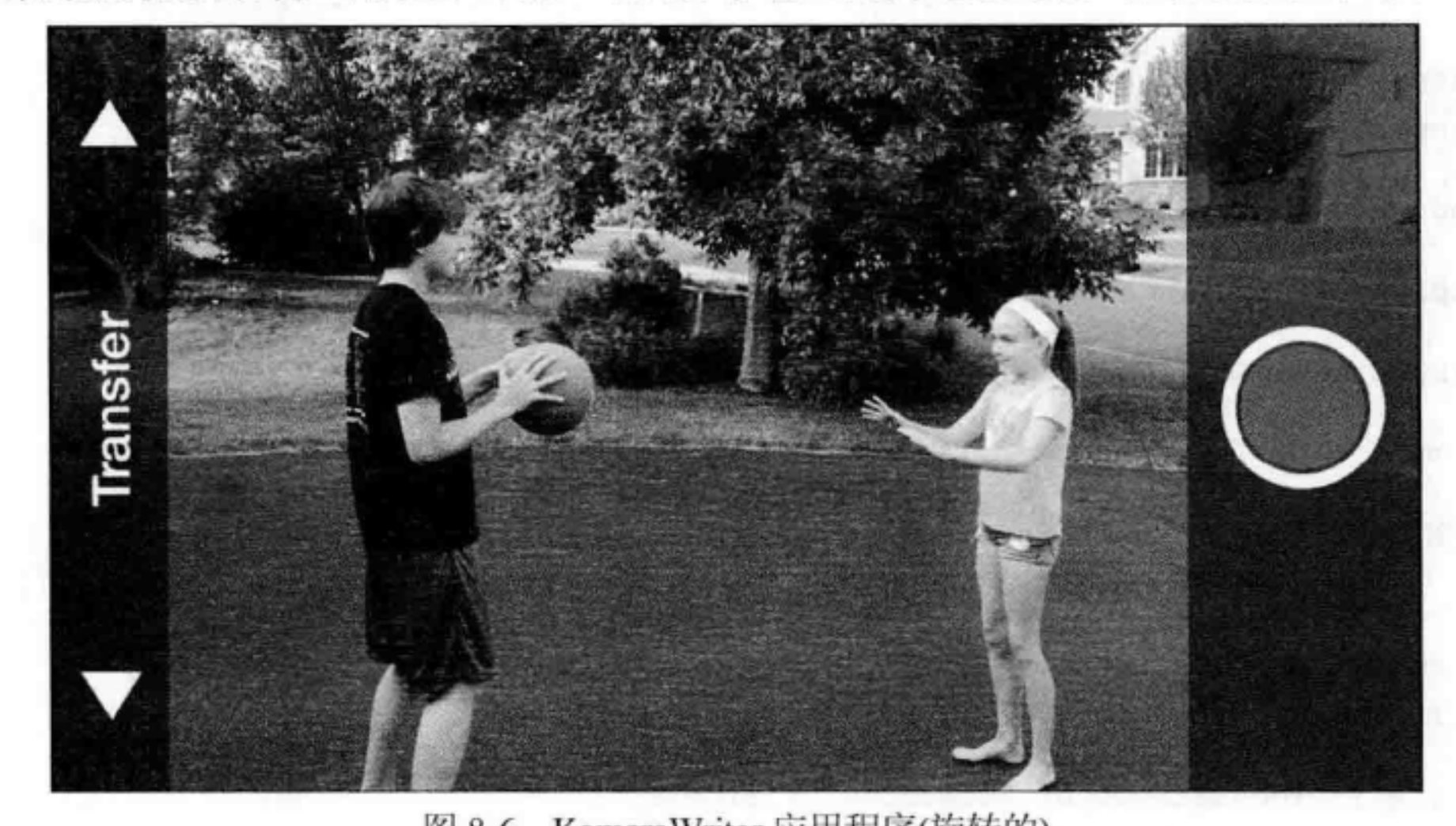

图 8-6　KameraWriter 应用程序(旋转的)

从iOS 7开始，内置的相机应用程序在拍摄静态图片时就可以应用筛选器了。这些筛选器可以在Core Image框架中找到，并且在最新的iOS设备中它们还可用于将实时视频效果添加到应用程序中。Core Image框架的详细内容不在本书的讨论范围内，不过在开发本章这个功能时，也与它进行最小限度的交互。

第一步创建摄像头控制器。我们创建一个与上一章一样的基础结构，这样可以减少在创建过程中样板的代码量。

先从THCameraController接口开始，如代码清单8-9所示。

代码清单 8-9　THCameraController 接口

```
#import "THImageTarget.h"
#import "THBaseCameraController.h"

@interface THCameraController : THBaseCameraController

- (void)startRecording;
- (void)stopRecording;
@property (nonatomic, getter = isRecording) BOOL recording;

@property (weak, nonatomic) id <THImageTarget> imageTarget;

@end
```

这个接口定义了开始录制、停止录制以及确定录制状态的方法。还提供了一个imageTarget属性，作为Core Image的CIImage对象的可视化输出。

下一步就是完成摄像头控制器的实现，首先配置捕捉会话输出，如代码清单8-10所示。

代码清单 8-10 配置会话输出

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate,
                                  AVCaptureAudioDataOutputSampleBufferDelegate>

@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;
@property (strong, nonatomic) AVCaptureAudioDataOutput *audioDataOutput;

@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    self.videoDataOutput = [[AVCaptureVideoDataOutput alloc] init];              // 1

    NSDictionary *outputSettings =
        @{(id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA)};

    self.videoDataOutput.videoSettings = outputSettings;
    self.videoDataOutput.alwaysDiscardsLateVideoFrames = NO;                     // 2

    [self.videoDataOutput setSampleBufferDelegate:self
                                          queue:self.dispatchQueue];

    if ([self.captureSession canAddOutput:self.videoDataOutput]) {
        [self.captureSession addOutput:self.videoDataOutput];
    } else {
        return NO;
    }

    self.audioDataOutput = [[AVCaptureAudioDataOutput alloc] init];              // 3

    [self.audioDataOutput setSampleBufferDelegate:self
                                          queue:self.dispatchQueue];

    if ([self.captureSession canAddOutput:self.audioDataOutput]) {
        [self.captureSession addOutput:self.audioDataOutput];
    } else {
        return NO;
    }

    return YES;
}

- (NSString *)sessionPreset {                                                    // 4
    return AVCaptureSessionPresetMedium;
}
```

```
- (void)startRecording {
   // To be implemented
}

- (void)stopRecording {
   // To be implemented
}

#pragma mark - Delegate methods

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
      fromConnection:(AVCaptureConnection *)connection {

   // To be implemented
}

@end
```

(1) 首先创建一个新的AVCaptureVideoDataOutput。用上一章的方法对它进行配置，设置它的输出格式为kCVPixelFormatType_32BGRA。当结合OpenGL ES和CoreImage时这一格式非常适合。

(2) 设置alwaysDiscardsLateVideoFrames为NO。由于我们要记录输出内容，所以通常我们希望捕捉全部的可用帧。设置这个属性为NO会给委托方法一些额外的时间来处理样本buffer，不过这会增加内存消耗。不过在处理每个样本buffer时都要做到尽可能高效，这样才能保障实时性能。

(3) 还创建了一个新的AVCaptureAudioDataOutput实例。这个类是AVCaptureVideoDataOutput的兄弟类，用于从带有活动AVCaptureSession的音频设备中捕捉音频样本。

(4) 设置会话预设值为AVCaptureSessionPresetMedium。开发这个应用程序时最好从一个简单的预设值开始，等我们熟悉了更多功能后，再一点点增加难度来满足更高级的需求。

捕捉会话的输出配置完毕后，就需要实现委托回调方法了，如代码清单8-11所示。

代码清单 8-11　捕捉输出委托

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
      fromConnection:(AVCaptureConnection *)connection {

   if (captureOutput == self.videoDataOutput) {                          // 1

      CVPixelBufferRef imageBuffer =                                     // 2
         CMSampleBufferGetImageBuffer(sampleBuffer);

      CIImage *sourceImage =                                             // 3
         [CIImage imageWithCVPixelBuffer:imageBuffer options:nil];

      [self.imageTarget setImage:sourceImage];
   }
}
```

(1) captureOutput:didOutputSampleBuffer:fromConnection:方法是两类捕捉输出的委托回调。

目前我们只处理视频样本，所以我们仅限于处理来自AVCaptureVideoDataOutput实例的回调。

(2) 使用CMSampleBufferGetImageBuffer函数从样本buffer中获取基础CVPixelBuffer。

(3) 从CVPixelBuffer中创建一个新的CIImage，并将它传递给需要在屏幕上呈现的图片目标。

基本的捕捉功能已经完成了，现在可以在设备上编译和运行应用程序来实际体验筛选的功能。点击筛选器名称旁边的箭头来进行切换。现在我们已经成功开发了一个应用于视频帧的实时视频筛选器，感谢Core Image框架的帮助！

> **注意：**
>
> 示例应用程序为 AVCaptureVideoDataOutput 和 AVCaptureAudioDataOutput 两个实例使用了一个调度队列。这对于我们的示例应用程序来说是足够的，不过如果希望对数据进行更复杂的处理，可能需要考虑为每一个使用单独的队列。苹果公司有一个示例应用程序称为 RosyWriter(在 ADC 上可用)用的就是这种方法。它还给出了一些供有效处理 CMSample-Buffers 的更高级性能选项。

KameraWriter应用程序看起来还不错，基本功能都也都实现了，不过还不能记录输出内容。要解决这一问题，需要创建一个THMovieWriter对象，该对象和AVCaptureMovieFileOutput的功能类似，不过它使用AVAssetWriter来执行视频编码和文件写入。下面看一下这个对象的接口定义，如代码清单8-12所示。

代码清单 8-12 THMovieWriter 接口

```
#import <AVFoundation/AVFoundation.h>

@protocol THMovieWriterDelegate <NSObject>
- (void)didWriteMovieAtURL:(NSURL *)outputURL;
@end

@interface THMovieWriter : NSObject

- (id)initWithVideoSettings:(NSDictionary *)videoSettings                    // 1
            audioSettings:(NSDictionary *)audioSettings
            dispatchQueue:(dispatch_queue_t)dispatchQueue;

- (void)startWriting;
- (void)stopWriting;
@property (nonatomic) BOOL isWriting;

@property (weak, nonatomic) id<THMovieWriterDelegate> delegate;              // 2

- (void)processSampleBuffer:(CMSampleBufferRef)sampleBuffer;                 // 3

@end
```

(1) THMovieWriter实例带有两个字典，用来描述基础AVAssetWriter实例的配置参数和调度队列。定义了写入进程的开始和停止方法及监听其工作状态的方法。

(2) 这个类定义了一个委托协议THMovieWriterDelegate来表示影片文件什么时候被写入

磁盘，之后委托会收到通知并采取相应的行动。

(3) THMovieWriter的一个关键方法是processSampleBuffer:，每当有新的样本被捕捉输出对象捕捉到时，都会调用这个方法。

对于AVAssetWriter，即使是简单的示例也会比较复杂。所以我们将这一开发过程分解成一些小段。首先我们处理有关生命周期的方法，如代码清单8-13所示。

代码清单 8-13　THMovieWriter 生命周期方法

```
#import "THMovieWriter.h"
#import <AVFoundation/AVFoundation.h>
#import "THContextManager.h"
#import "THFunctions.h"
#import "THPhotoFilters.h"
#import "THNotifications.h"

static NSString *const THVideoFilename = @"movie.mov";

@interface THMovieWriter ()

@property (strong, nonatomic) AVAssetWriter *assetWriter;                    // 1
@property (strong, nonatomic) AVAssetWriterInput *assetWriterVideoInput;
@property (strong, nonatomic) AVAssetWriterInput *assetWriterAudioInput;
@property (strong, nonatomic)
   AVAssetWriterInputPixelBufferAdaptor *assetWriterInputPixelBufferAdaptor;

@property (strong, nonatomic) dispatch_queue_t dispatchQueue;

@property (weak, nonatomic) CIContext *ciContext;
@property (nonatomic) CGColorSpaceRef colorSpace;
@property (strong, nonatomic) CIFilter *activeFilter;

@property (strong, nonatomic) NSDictionary *videoSettings;
@property (strong, nonatomic) NSDictionary *audioSettings;

@property (nonatomic) BOOL firstSample;

@end

@implementation THMovieWriter

- (id)initWithVideoSettings:(NSDictionary *)videoSettings
          audioSettings:(NSDictionary *)audioSettings
          dispatchQueue:(dispatch_queue_t)dispatchQueue {

   self = [super init];
   if (self) {
      _videoSettings = videoSettings;
      _audioSettings = audioSettings;
      _dispatchQueue = dispatchQueue;

      _ciContext = [THContextManager sharedInstance].ciContext;              // 2
      _colorSpace = CGColorSpaceCreateDeviceRGB();

      _activeFilter = [THPhotoFilters defaultFilter];
```

```
        _firstSample = YES;

        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];        // 3
        [nc addObserver:self
              selector:@selector(filterChanged:)
                 name:THFilterSelectionChangedNotification
               object:nil];
    }
    return self;
}

- (void)dealloc {
    CGColorSpaceRelease(_colorSpace);
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

- (void)filterChanged:(NSNotification *)notification {
    self.activeFilter = [notification.object copy];
}

- (void)startWriting {
    // To be implemented
}

- (void)processSampleBuffer:(CMSampleBufferRef)sampleBuffer
                 mediaType:(CMMediaType)mediaType {
    // To be implemented
}

- (void)stopWriting {

    // To be implemented

}

- (NSURL *)outputURL {                                                          // 4
    NSString *filePath =
        [NSTemporaryDirectory() stringByAppendingPathComponent:THVideoFilename];
    NSURL *url = [NSURL fileURLWithPath:filePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:url.path]) {
        [[NSFileManager defaultManager] removeItemAtURL:url error:nil];
    }
    return url;
}

@end
```

(1) 在类扩展中我们为AVAssetWriter创建了一些属性和相关的对象。每当startWriting方法被调用时，就创建对象的图片并在写入会话的持续时间内保持对它们的强引用关系。

(2) 从THContextManager对象得到分享的Core Image上下文。这个对象受OpenGL ES的支持并用于筛选传进来的视频样本，最后得到一个CVPixelBuffer。

(3) 注册THFilterSelectionChangedNotification通知的监听器。当用户切换可用筛选器列表时就会从用户界面发送该通知。每当通知发送时，filterChanged:方法就会被调用，并相应地

更新activeFilter属性。

(4) 定义一个outputURL方法来配置AVAssetWriter实例。这个方法在临时目录中定义了一个NSURL，并将之前的同名文件删除。

有关生命周期的配置完成后，我们再来看一下startWriting方法的实现，如代码清单8-14所示。

代码清单 8-14　设置 AVAssetWriter 图片

```
- (void)startWriting {
  dispatch_async(self.dispatchQueue, ^{                                    // 1

    NSError *error = nil;

    NSString *fileType = AVFileTypeQuickTimeMovie;
    self.assetWriter =                                                     // 2
      [AVAssetWriter assetWriterWithURL:[self outputURL]
                                fileType:fileType
                                  error:&error];
    if (!self.assetWriter || error) {
      NSString *formatString = @"Could not create AVAssetWriter: %@";
      NSLog(@"%@", [NSString stringWithFormat:formatString, error]);
      return;
    }

    self.assetWriterVideoInput =                                           // 3
      [[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeVideo
                                    outputSettings:self.videoSettings];

    self.assetWriterVideoInput.expectsMediaDataInRealTime = YES;

    UIDeviceOrientation orientation = [UIDevice currentDevice].orientation;
    self.assetWriterVideoInput.transform =                                 // 4
      THTransformForDeviceOrientation(orientation);

    NSDictionary *attributes = @{                                          // 5
      (id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA),
      (id)kCVPixelBufferWidthKey : self.videoSettings[AVVideoWidthKey],
      (id)kCVPixelBufferHeightKey : self.videoSettings[AVVideoHeightKey],
      (id)kCVPixelFormatOpenGLESCompatibility : (id)kCFBooleanTrue
    };

    self.assetWriterInputPixelBufferAdaptor =                              // 6
      [[AVAssetWriterInputPixelBufferAdaptor alloc]
        initWithAssetWriterInput:self.assetWriterVideoInput
     sourcePixelBufferAttributes:attributes];

    if ([self.assetWriter canAddInput:self.assetWriterVideoInput]) {       // 7
      [self.assetWriter addInput:self.assetWriterVideoInput];
    } else {
      NSLog(@"Unable to add video input.");
      return;
    }
```

```
        self.assetWriterAudioInput =                                          // 8
            [[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeAudio
                                        outputSettings:self.audioSettings];

        self.assetWriterAudioInput.expectsMediaDataInRealTime = YES;

        if ([self.assetWriter canAddInput:self.assetWriterAudioInput]) {      // 9
            [self.assetWriter addInput:self.assetWriterAudioInput];
        } else {
            NSLog(@"Unable to add audio input.");
            return;
        }

        self.isWriting = YES;                                                 // 10
        self.firstSample = YES;
    });
}
```

(1) 让用户点击Record按钮是为了避免卡顿，以异步方式调度到dispatchQueue队列，以便设置AVAssetWriter对象。

(2) 创建一个新的AVAssetWriter实例，将写入目的文件的输出URL、文件类型常量和一个NSError传递给它。在创建对象时如果出现错误，则将错误信息输出到控制台并返回。

(3) 创建一个新的AVAssetWriterInput，以附加从AVCaptureVideoDataOuput中得到的样本。传递给初始化方法一个AVMediaTypeVideo媒体类型和创建THMovieWriter的视频设置。设置expectsMediaDataInRealTime属性为YES来指明这个输入应该针对实时性进行优化。

(4) 应用程序的用户界面锁定为垂直方向，不过我们希望捕捉应该可以支持任何方向。判断用户界面的方向并使用THTransformForDeviceOrientation函数为输入设置一个合适的转换。在写入会话期间，方向会按照这一设定保持不变。

(5) 定义属性的NSDictionary用于配置将在下一步中创建的AVAssetWriterInput-PixelBufferAdaptor。要保证最大效率，字典中的值应该对应于在配置AVCaptureVideoData-Output时所使用的原像素格式。

(6) 创建一个新的AVAssetWriterInputPixelBufferAdaptor，传递上一步中创建的属性给它。这个对象提供了一个优化的CVPixelBufferPool，使用它可以创建CVPixelBuffer对象来渲染筛选视频帧。

(7) 将视频输入添加到资源写入器，如果输入不能被添加，将错误信息输出到控制台并返回。

(8) 创建AVAssetWriterInput，用于附加来自AVCaptureAudioDataOutput的样本。给初始化方法传递一个AVMediaTypeAudio媒体类型和创建THMovieWriter的音频设置。设置expectsMediaDataInRealTime属性为YES来指明这个输入应该针对实时性进行优化。

(9) 将音频输入添加到资源写入器。如果输入不能被添加，则将错误信息输出到控制台并返回。

(10) 设置isWriting和firstSample属性为YES，就可以开始附加样本了。

接下来，下面看一下processSampleBuffer:方法的实现，这个方法里我们将附加从捕捉输出得到的CMSampleBuffer对象，如代码清单8-15所示。

代码清单 8-15　处理样本 buffer

```
- (void)processSampleBuffer:(CMSampleBufferRef)sampleBuffer {

    if (!self.isWriting) {
        return;
    }

    CMFormatDescriptionRef formatDesc =                                    // 1
        CMSampleBufferGetFormatDescription(sampleBuffer);

    CMMediaType mediaType = CMFormatDescriptionGetMediaType(formatDesc);

    if (mediaType == kCMMediaType_Video) {

        CMTime timestamp =
            CMSampleBufferGetPresentationTimeStamp(sampleBuffer);

        if (self.firstSample) {                                            // 2
            if ([self.assetWriter startWriting]) {
                [self.assetWriter startSessionAtSourceTime:timestamp];
            } else {
                NSLog(@"Failed to start writing.");
            }
            self.firstSample = NO;
        }

        CVPixelBufferRef outputRenderBuffer = NULL;

        CVPixelBufferPoolRef pixelBufferPool =
            self.assetWriterInputPixelBufferAdaptor.pixelBufferPool;

        OSStatus err = CVPixelBufferPoolCreatePixelBuffer(NULL,            // 3
                                                          pixelBufferPool,
                                                          &outputRenderBuffer);
        if (err) {
            NSLog(@"Unable to obtain a pixel buffer from the pool.");
            return;
        }

        CVPixelBufferRef imageBuffer =                                     // 4
            CMSampleBufferGetImageBuffer(sampleBuffer);

        CIImage *sourceImage = [CIImage imageWithCVPixelBuffer:imageBuffer
                                                       options:nil];

        [self.activeFilter setValue:sourceImage forKey:kCIInputImageKey];

        CIImage *filteredImage = self.activeFilter.outputImage;

        if (!filteredImage) {
            filteredImage = sourceImage;
        }

        [self.ciContext render:filteredImage                               // 5
              toCVPixelBuffer:outputRenderBuffer
```

```
                    bounds:filteredImage.extent
                colorSpace:self.colorSpace];

        if (self.assetWriterVideoInput.readyForMoreMediaData) {            // 6
            if (![self.assetWriterInputPixelBufferAdaptor
                        appendPixelBuffer:outputRenderBuffer
                    withPresentationTime:timestamp]) {
                NSLog(@"Error appending pixel buffer.");
            }
        }

        CVPixelBufferRelease(outputRenderBuffer);

    }
    else if (!self.firstSample && mediaType == kCMMediaType_Audio) {       // 7
        if (self.assetWriterAudioInput.isReadyForMoreMediaData) {
            if (![self.assetWriterAudioInput appendSampleBuffer:sampleBuffer]) {
                NSLog(@"Error appending audio sample buffer.");
            }
        }
    }

}
```

(1) 这个方法可以处理音频和视频两类样本，所以我们需要确定样本的媒体类型才能附加到正确的写入器输入。查看样本buffer的CMFormatDescription并使用CMFormatDescriptionGetMediaType方法来判断它的媒体类型。

(2) 如果当用户点击Record按钮后正在处理的是第一个视频样本，则调用资源写入器的startWriting和startSessionAtSourceTime:方法启动一个新的写入会话，将样本的呈现时间作为源时间传递到方法中。

(3) 从像素buffer适配器池中创建一个空的CVPixelBuffer，使用该像素buffer渲染筛选好的视频帧的输出。

(4) 使用CMSampleBufferGetImageBuffer函数获取当前视频样本的CVPixelBuffer。根据像素buffer创建一个新的CIImage并将它设置为活动筛选器的kCIInputImageKey值。通过筛选器得到输出图片，会返回一个封装了CIFilter操作的CIImage对象。如果因为某种原因filteredImage为nil，则设置CIImage的引用为原始的sourceImage。

(5) 将筛选好的CIImage的输出渲染到第3步创建的CVPixelBuffer中。

(6) 如果视频输入的readyForMoreMediaData属性为YES，则将像素buffer连同当前样本的呈现时间都附加到AVAssetWriterPixelBufferAdaptor。现在就完成了对当前视频样本的处理，所以此时应该调用CVPixelBufferRelease函数释放像素buffer。

(7) 如果第一个样本处理完成并且当前的CMSampleBuffer是一个音频样本，则询问音频AVAssetWriterInput是否准备接收更多的数据。如果可以，则将它添加到输入。

processSampleBuffer:方法就完成了，最后一个需要实现的功能是stopWriting方法，如代码清单8-16所示。

代码清单 8-16　完成写入会话

```
- (void)stopWriting {

    self.isWriting = NO;                                                    // 1

    dispatch_async(self.dispatchQueue, ^{

        [self.assetWriter finishWritingWithCompletionHandler:^{             // 2

            if (self.assetWriter.status == AVAssetWriterStatusCompleted) {
                dispatch_async(dispatch_get_main_queue(), ^{                // 3
                    NSURL *fileURL = [self.assetWriter outputURL];
                    [self.delegate didWriteMovieAtURL:fileURL];
                });
            } else {
                NSLog(@"Failed to write movie: %@", self.assetWriter.error);
            }
        }];
    });
}
```

(1) 设置isWriting的标志为NO，这样processSampleBuffer:mediaType:方法就不会再处理更多的样本。

(2) 调用finishWritingWithCompletionHandler:方法来终止写入会话并关闭磁盘上的文件。

(3) 判断资源写入器的状态。如果status等于AVAssetWriterStatusCompleted，则表示文件成功写入，调度回到主线程调用委托的didWriteMovieAtURL:方法。如果status等于其他值，则将资源写入器的错误信息输出到控制台。

THMovieWriter类的实现就全部完成了，最后一步就是将它整合到THCameraController中，如代码清单8-17所示。

代码清单 8-17　应用 THMovieWriter

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
#import "THMovieWriter.h"
#import <AssetsLibrary/AssetsLibrary.h>

@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate,
                                  AVCaptureAudioDataOutputSampleBufferDelegate,
                                  THMovieWriterDelegate>

@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;
@property (strong, nonatomic) AVCaptureAudioDataOutput *audioDataOutput;

@property (strong, nonatomic) THMovieWriter *movieWriter;                   // 1

@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    // AVCaptureVideoDataOutput and AVCaptureAudioDataOutput set up and
```

```
    // configuration previously covered in Listing 8.x

    NSString *fileType = AVFileTypeQuickTimeMovie;

    NSDictionary *videoSettings =                                          // 2
        [self.videoDataOutput
            recommendedVideoSettingsForAssetWriterWithOutputFileType:fileType];

    NSDictionary *audioSettings =
        [self.audioDataOutput
            recommendedAudioSettingsForAssetWriterWithOutputFileType:fileType];

    self.movieWriter =                                                     // 3
        [[THMovieWriter alloc] initWithVideoSettings:videoSettings
                                          audioSettings:audioSettings
                                          dispatchQueue:self.dispatchQueue];
    self.movieWriter.delegate = self;

    return YES;
}

- (NSString *)sessionPreset {
    return AVCaptureSessionPreset1280x720;
}

- (void)startRecording {                                                   // 4
    [self.movieWriter startWriting];
    self.recording = YES;
}

- (void)stopRecording {
    [self.movieWriter stopWriting];
    self.recording = NO;
}

#pragma mark - Delegate methods

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
      fromConnection:(AVCaptureConnection *)connection {

    [self.movieWriter processSampleBuffer:sampleBuffer];                   // 5

    if (captureOutput == self.videoDataOutput) {

        CVPixelBufferRef imageBuffer =
            CMSampleBufferGetImageBuffer(sampleBuffer);

        CIImage *sourceImage =
            [CIImage imageWithCVPixelBuffer:imageBuffer options:nil];

        [self.imageTarget setImage:sourceImage];
    }
}

- (void)didWriteMovieAtURL:(NSURL *)outputURL {                           // 6
```

```
ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

if ([library videoAtPathIsCompatibleWithSavedPhotosAlbum:outputURL]) {

    ALAssetsLibraryWriteVideoCompletionBlock completionBlock;

    completionBlock = ^(NSURL *assetURL, NSError *error){
        if (error) {
            [self.delegate assetLibraryWriteFailedWithError:error];
        }
    };

        [library writeVideoAtPathToSavedPhotosAlbum:outputURL
                                  completionBlock:completionBlock];
    }
}

@end
```

(1) 创建一个新属性，保存关于THMovieWriter对象的强引用。同时需要类遵循THMovie-WriterDelegate协议。

(2) iOS 7版本中引入了一些便捷方法让开发者可以简单地创建带有推荐音频和视频设置的字典对象，这些设置都针对配置AVAssetWriter需要的文件类型。调用这些方法并创建传递到THMovieWriter中的字典。

(3) 创建一个新的THMovieWriter实例，传递配置字典和控制器的dispatchQueue的引用，该引用是在它的超类(THBaseCameraController)中定义的。将该控制器作为影片写入器的委托。

(4) 实现startRecording和stopRecording方法，并在THMovieWriter上调用相应的方法。对每个方法都按照需要更新recording状态。

(5) 调用影片写入器的processSampleBuffer:方法，将当前CMSampleBuffer和媒体类型传递给方法。

(6) 实现委托协议的didWriteMovieAtURL:方法。使用前面介绍过的ALAssetsLibrary框架将最新创建的影片写入到用户的Photos library。

现在可以运行应用程序并录制视频了。打开应用程序并点击Record按钮，当录制进行时，可以点击可用的筛选器。再次点击Record按钮可以停止录制。可以切换到iOS相机应用程序体验这个视频带来的喜悦吧。

8.4 小结

本章主要学习了AVAssetReader和AVAssetWriter所提供的强大功能。这些类都是AV Foundation在处理媒体资源对象时的底层方法，在许多高级用例中都起到了至关重要的作用。虽然在CMSampleBuffer对象一级处理媒体比较复杂，但是它可以提供仅使用框架顶层功能无法实现的一些高级功能。精通AVAssetReader和AVAssetWriter的用法需要一定的时间，不过本章的主题以及示例项目可以为开发者创建自己的应用程序奠定一个良好的基础。

8.5 挑战

学习使用AVAssetReader和AVAssetWriter的关键在于要清楚地知道哪些功能会用到这些类。本章虽然给出了一些重要且常见的用例，不过这仅是众多应用中的九牛一毛。利用本章学到的知识，多研究苹果公司的示例代码和Apple Developer Center网站上分享的其他资源：

- StopNGo for iOS 演示如何创建一个定格动画相机应用程序。
- Sample Photo Editing Extension 介绍了如何创建 iOS 8 可编辑的扩展。
- Technical Note TN2310 介绍了如何使用 AVAssetReader 和 AVAssetWriter 进行时间编码。
- Writing Subtitles to a Movie from the Command Line for OS X。

第 III 部分

媒体创建和编辑

第 9 章　媒体的组合和编辑
第 10 章　混合音频
第 11 章　创建视频过渡效果
第 12 章　动画图层内容

第9章

媒体的组合和编辑

当下是创建数字媒体的黄金时代。著名的电影导演都在使用专业级别的数码SLR相机，实惠且能够达到工作室质量级别的数字音频界面让全世界的音乐家都可以创造出与卓越品质相媲美的音乐，并且iOS和其他便携式设备允许随时创造数字音频和视频。能够拍摄出高质量的数字媒体确实非常有趣和令人激动。不过这只是创造内容的过程，可以说媒体创造的最重要部分其实是创造之后的编辑过程。

编辑其实就是选择、编排和操纵最佳镜头并提供既连续又连贯的内容。能力出众的编辑对于专业电影或视频的创作起着至关重要的作用，不过无论什么规模的媒体编辑都非常重要。无法编辑的播客节目渐渐失去了听众，包含大量错误的视频教程会让学生感到无所适从，将没经过编辑的长达几小时的家庭夏威夷度假录像传到Facebook上是不会受到大家青睐的。

AV Foundation给出了一些API来创建非线性、无损的编辑工具和应用程序。术语“非线性”的意思是可以自由组合、分开、修剪、覆盖和以任何顺序重排媒体片段直到满足我们的期望。最好的一点是这些编辑操作都是无损的，即最初的源媒体资源不会被破坏，可以无约束地进行编辑。本章将研究编辑过程众多关键环节中的一个，那就是如何将多个资源组合在一起。

9.1 组合媒体

想象一下最近如果去一趟旧金山，在那里拍摄了一些金门公园、日本茶园和渔人码头的短片。我们希望将这些视频中的最好镜头组合在一起，配上合适的音乐，做成一个可与朋友和家人一起分享的影片，如图9-1所示。

要完成这一任务，我们首先需要将相关片段从这些源媒体中提取出来，再将它们组成一个临时排列。AV Foundation明确定义了这个功能，可以让开发者简单地将多个音频和视频资源组合成一个新资源。图9-2给出了框架提供的与资源组合相关的几个核心类。

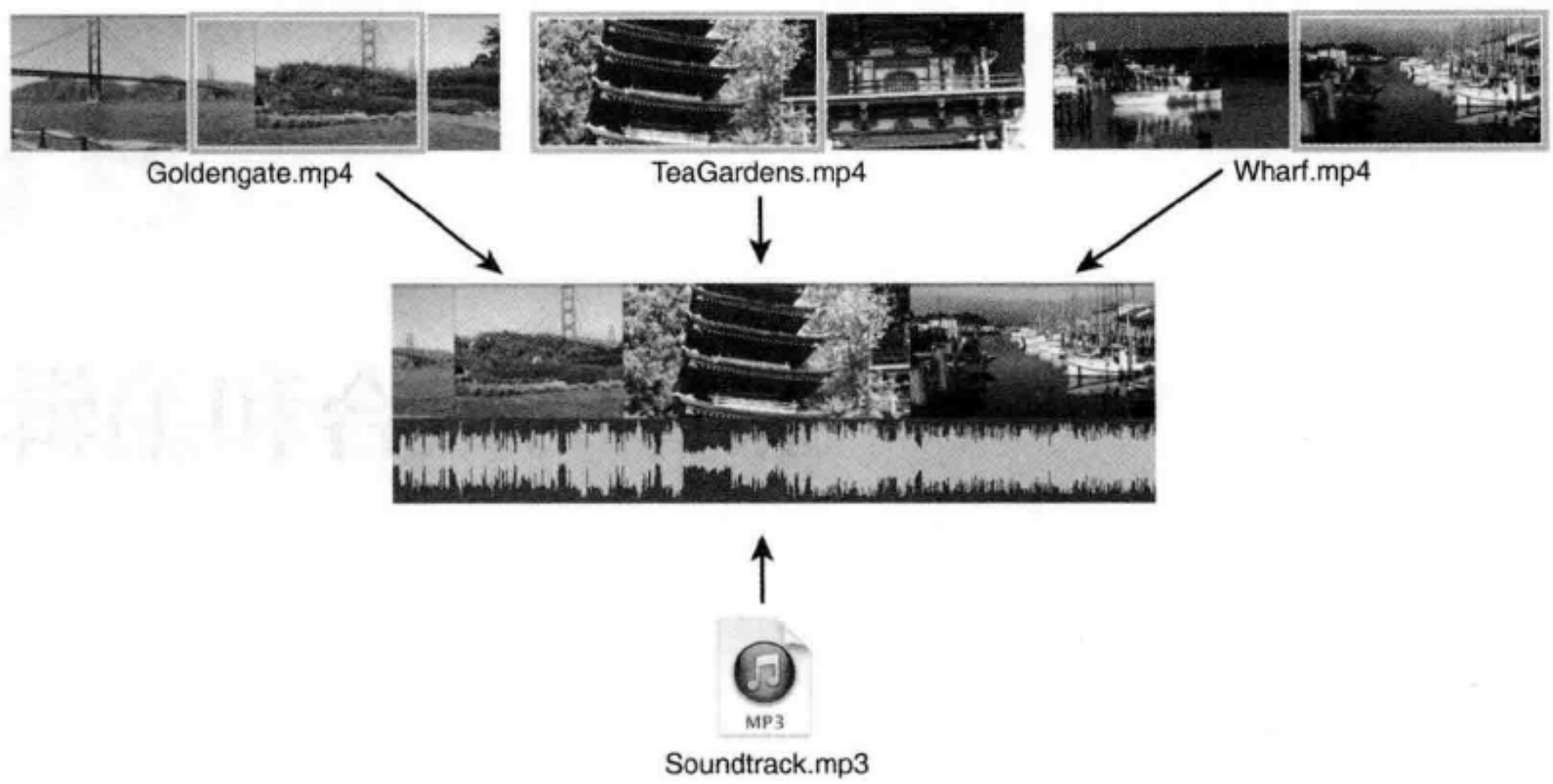

图 9-1 媒体组合

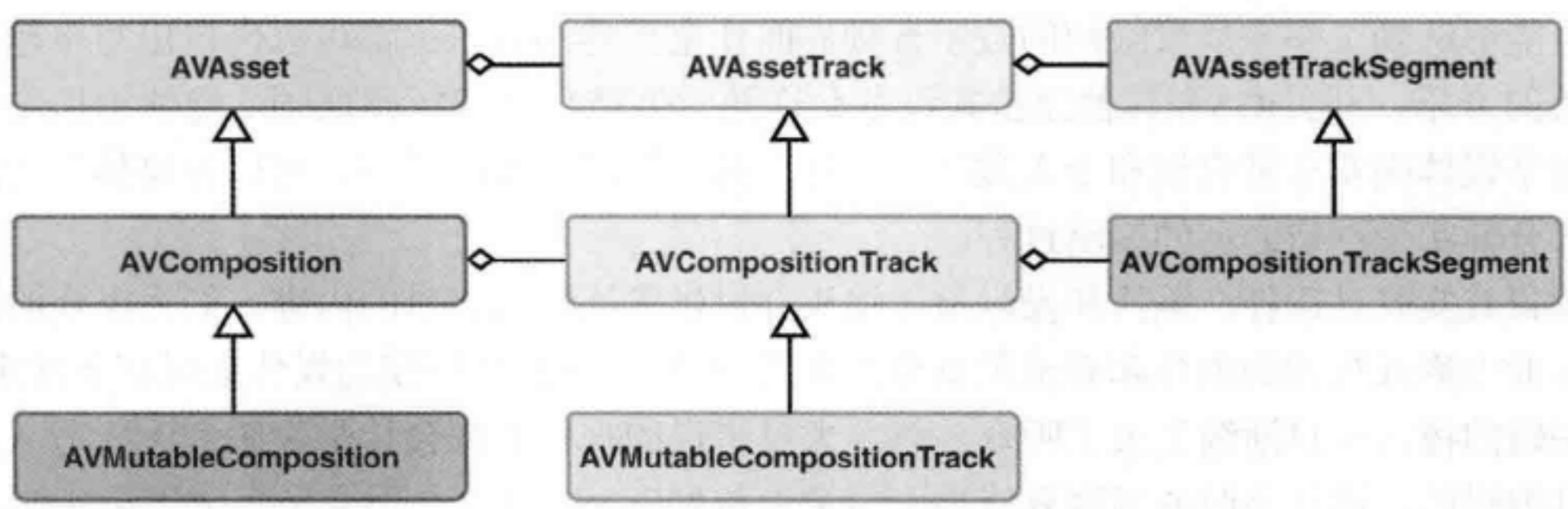

图 9-2 AV Foundation 组合类

AV Foundation有关资源组合的功能源于AVAsset的子类AVComposition。一个组合就是将其他几种媒体资源组合成一个自定义的临时排列，再将这个临时排列视为一个可以呈现或处理的独立媒体项目。就比如AVAsset对象，组合相当于包含了一个或多个给定类型的媒体轨道的容器。AVComposition中的轨道都是AVAssetTrack的子类AVCompositionTrack。一个组合轨道本身由一个或多个媒体片段组成，由AVCompositionTrackSegment类定义，代表这个组合中的实际媒体区域。通常比较容易查看这些对象间的关系，因为它们与诸如iMovie或Logic Pro X等编辑工具中所描绘的情况类似，如图9-3所示。

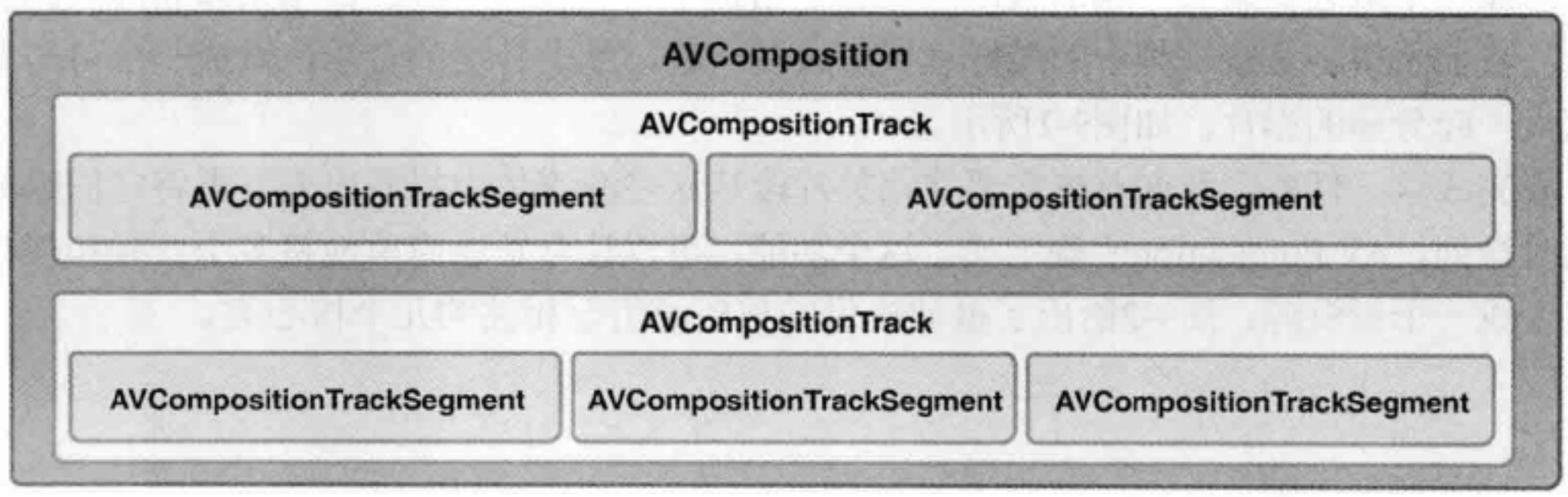

图 9-3 AV Foundation 组合排列

就像AVComposition扩展了AVAsset一样，所有可以使用常见资源的场景都可以使用这个对象，比如播放、图片提取或导出。不过组合被认为是更抽象的术语。不过AVAsset到特殊媒体文件具有直接一对一的映射，组合的概念更像是一组说明，描述多个资源间如何正确地呈现或处理。因此，这些一般都是轻量级的临时对象，我们在应用程序会频繁地创建和处理它们。

> **注意：**
>
> 一个经常被问道的问题是，如何保存一个组合？简单回答就是不保存。AVComposition及其相关类没有遵循 NSCoding 协议，所以不能简单地将一个组合在内存的状态归档到磁盘上。如何我们正创建一个需要具有保存项目文件能力的音频或视频编辑应用程序，需要开发自定义的数据模型类来保存这个状态。

AVComposition和AVCompositionTrack都是不可变对象，提供对资源的只读操作。这些对象提供了一个合适的接口让应用程序的一部分可以进行播放或处理。不过，当创建自己的组合时，就需要使用AVMutableComposition和AVMutableCompositionTrack所提供的可变子类。这些对象提供的接口需要操作轨道和轨道分段，这样我们就可以创建所需的临时排列了。

要创建自定义组合，需指定在将要添加到组合的源媒体的时间范围，还要指定要添加片段的每个轨道的位置。在我们开始学习创建组合内容的技巧前，首先需要讨论时间和时间范围是如何在框架中呈现的。

9.2　时间的处理

第4章中曾简单讨论过有关时间的问题。要创建高效的组合内容，掌握有关时间和时间范围的相关知识非常重要，所以下面更深入地研究这一问题。

9.2.1　CMTime

通常开发者认为时间的呈现格式应该是浮点型数据。有着长期苹果平台开发经验的应用程序员一般都使用NSTimeInterval，它是一个简单的双精度类型，可以表示不同场景中的时间。实际上，AV Foundation在AVAudioPlayer和AVAudioRecorder中处理时间问题时本身也会使用这个类型。虽然很多通用的开发环境使用双精度表示时间已经足够满足要求了，不过其天然的不精确性导致双精度类型无法应用于更多的高级时基媒体的开发中。比如，一个单一舍入错误就会导致丢帧或音频丢失。相反，苹果公司使用Core Media框架定义的CMTime数据类型作为时间格式。C语言结构的类型定义如下所示：

```
typedef struct {
    CMTimeValue value;
    CMTimeScale timescale;
    CMTimeFlags flags;
    CMTimeEpoch epoch;
} CMTime;
```

上面结构中最相关的三个组件是value、timescale和flags。CMTimeValue和CMTimeScale分别是64位和32位有符号整型变量，是CMTime元素的分数形式。CMTimeFlags是一个位掩码用于表示时间的指定状态，比如判断数据是否有效、不确定或是否出现舍入值等。CMTime实例可标记特定的时间点或用于表示持续时间。

有多种方法可以创建CMTime实例，不过最常见的方法是使用CMTimeMake函数，指定一个64位的value参数和一个32位的timescale参数。比如，创建一个代表3秒的CMTime表达式有下面几种不同的方式：

```
CMTime t1 = CMTimeMake(3, 1);
CMTime t2 = CMTimeMake(1800, 600);
CMTime t3 = CMTimeMake(3000, 1000);
CMTime t4 = CMTimeMake(132300, 44100);
```

使用CMTimeShow函数将这些值打印到控制台将会出现如下结果：

```
CMTimeShow(t1); // --> {3/1 = 3.000}
CMTimeShow(t2); // --> {1800/600 = 3.000}
CMTimeShow(t3); // --> {3000/1000 = 3.000}
CMTimeShow(t4); // --> {132300/44100 = 3.000}
```

注意：

在处理视频内容时常见的时间刻度为600，这是大部分常用视频帧率24FPS、25FPS和30FPS的公倍数。音频数据常见的时间刻度就是采样率，比如44 100(44.1kHz)或48 000(48kHz)。

除创建CMTime实例的函数外，我们可以看到在CMTime.h文件中定义了许多函数，这些函数都是用于有关时间的计算的。比如，在创建组合内容时经常会用到的时间相加和时间相减，使用CMTimeAdd和CMTimeSubtract函数可以很简单地完成这些任务。

```
CMTime time1 = CMTimeMake(5, 1);
CMTime time2 = CMTimeMake(3, 1);

CMTime result;

result = CMTimeAdd(time1, time2);
CMTimeShow(result); // --> {8/1 = 8.000}

result = CMTimeSubtract(time1, time2);
CMTimeShow(result); // --> {2/1 = 2.000}
```

通常在进行时间运算时，我们会使用Core Media框架所提供的函数，因为这些函数考虑了不同时间刻度和标志的问题。不过在有些情况下，使用小学数学的知识就够了。比如，可以通过操作其他时间刻度创建一个新的时间值。

```
CMTime time = CMTimeMake(5000, 1000);

CMTime doubledTime = CMTimeMake(time.value, time.timescale / 2);
CMTimeShow(doubledTime); // --> {5000/500 = 10.000}

CMTime halvedTime = CMTimeMake(time.value, time.timescale * 2);
```

```
CMTimeShow(halvedTime); // --> {5000/2000 = 2.500}
```

上例展示了一些最常用的CMTime函数，不过我们还是鼓励开发者参阅有关CMTime的文档，了解其他一些有关创建、操作和比较CMTime数值的方法。除了大量的函数外，还会发现许多常用的常量和宏。

9.2.2　CMTimeRange

Core Media框架还为时间范围提供了一个数据类型，称为CMTimeRange，它在有关资源编辑的API中扮演着重要角色。CMTimeRange由两个CMTime值组成，第一个值定义时间范围的起点，第二个值定义时间范围的持续时间。

```
typedef struct {
   CMTime start;
   CMTime duration;
} CMTimeRange;
```

创建CMTimeRange的一个方法是使用CMTimeRangeMake函数，它的第一个参数是定义时间范围起点的CMTime值，第二个参数是表示范围持续时长的CMTime值。比如，如果我们要创建一个时间范围，从时间轴的5秒钟位置开始，持续时长5秒，则创建方法如下所示：

```
CMTime fiveSecondsTime = CMTimeMake(5, 1);
CMTimeRange timeRange = CMTimeRangeMake(fiveSecondsTime, fiveSecondsTime);
CMTimeRangeShow(timeRange); // --> {{5/1 = 5.000}, {5/1 = 5.000}}
```

另一种创建时间范围的方法是使用CMTimeRangeFromTimeToTime函数。该函数通过表示范围起点和终点的CMTime值来创建一个CMTimeRange。比如，上面的示例还可以这样实现：

```
CMTime fiveSeconds = CMTimeMake(5, 1);
CMTime tenSeconds = CMTimeMake(10, 1);
CMTimeRange timeRange = CMTimeRangeFromTimeToTime(fiveSeconds, tenSeconds);
CMTimeRangeShow(timeRange); // --> {{5/1 = 5.000}, {5/1 = 5.000}}
```

CMTimeRange.h头文件定义了大量实用函数来处理时间范围相关的运算和比较。比如图9-4所示的是两个叠加的时间范围，每个都是5秒的时长，只是开始时间不同。

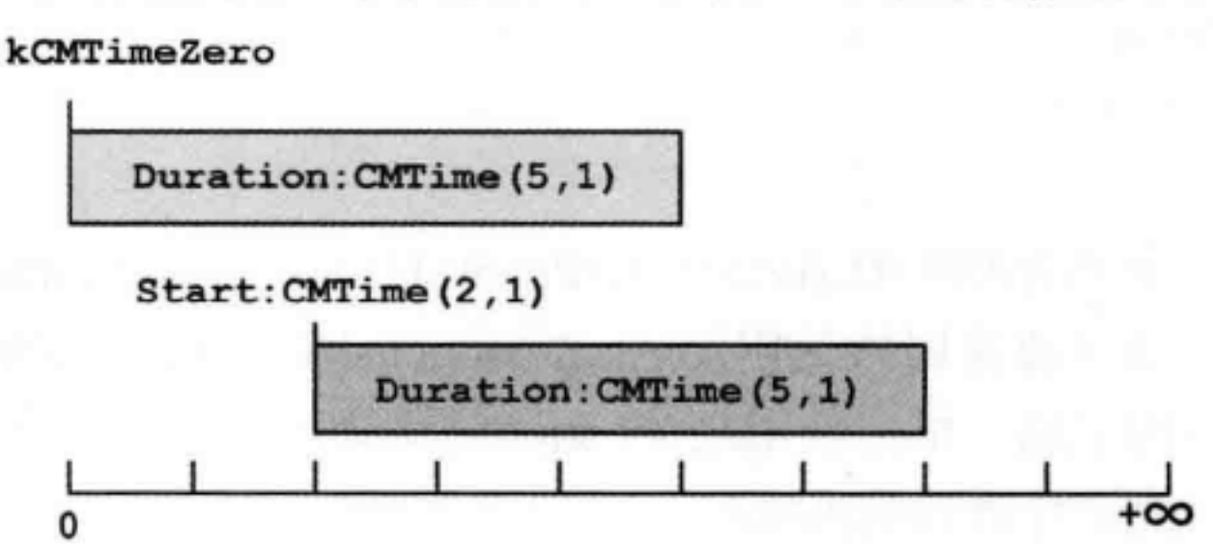

图 9-4　叠加时间范围

如果希望创建一个两个时间交叉的时间范围，或者希望得到两个时间范围的总和，可以

尝试下面给出的实现方法。

```
CMTimeRange range1 = CMTimeRangeMake(kCMTimeZero, CMTimeMake(5, 1));
CMTimeRange range2 = CMTimeRangeMake(CMTimeMake(2, 1), CMTimeMake(5, 1));

CMTimeRange intersectionRange = CMTimeRangeGetIntersection(range1, range2);
CMTimeRangeShow(intersectionRange); // --> {{2/1 = 2.000}, {3/1 = 3.000}}

CMTimeRange unionRange = CMTimeRangeGetUnion(range1, range2);
CMTimeRangeShow(unionRange); // --> {{0/1 = 0.000}, {7/1 = 7.000}}
```

用有理数来表示时间在一开始看起来可能会比较奇怪，这也是框架的编辑功能带来的一个新困惑点。不过在创建自定义组合功能时掌握这一技术至关重要。好消息是使用CMTime和CMTimeRange很快就会习惯。在下面几章的学习中，我们会大量用到这些数据类型，所以不久你就会习惯使用它们了。

9.3 基础方法

了解到CMTime和CMTimeRange的用法后，下面继续讨论实现一个与图9-1类似的组合资源。

创建一个组合资源需要我们拥有一个或多个等待处理的源AVAsset对象。在开始介绍组合资源示例前，了解一下有关资源创建和准备的知识非常重要。本书中大部分的示例都使用assetWithURL:类方法来创建AVAsset实例。这里创建它的一个子类AVURLAsset的实例。当创建资源用于组合时，应该直接使用URLAssetWithURL:options:方法实例化一个AVURLAsset。options:参数允许通过传递一个带有一个或多个初始化选项的NSDictionary来自定义资源初始化的方式。我们看一个载入应用程序bundle中的MP4视频的示例。

```
NSURL *url =
    [[NSBundle mainBundle] URLForResource:@"video" withExtension:@"mp4"];

NSDictionary *options = @{AVURLAssetPreferPreciseDurationAndTimingKey : @YES};
AVAsset *asset = [AVURLAsset URLAssetWithURL:url options:options];

// Asset "keys" to load
NSArray *keys = @[@"tracks", @"duration", @"commonMetadata"];

[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    // Validate loaded status of keys
}];
```

框架定义了一个名为AVURLAssetPreferPreciseDurationAndTimingKey的选项。选项带有@YES值可以确保当资源的属性使用AVAsynchronousKeyValueLoading协议载入时可以计算出准确的时长和时间信息。虽然使用这个option时还会对载入过程增添一些额外开销，不过这可以保证资源正处于合适的编辑状态。

现在就下面继续并开始创建组合资源吧。这一组合操作会将两个视频片段中第一个5秒视频拿出来，并按照组合视频轨道的顺序进行排列。组合资源还会从一个MP3文件中将音频轨道整合到视频中。

```
AVAsset *goldenGateAsset = // prepared golden gate asset
AVAsset *teaGardenAsset  = // prepared tea garden asset
AVAsset *soundtrackAsset = // prepared sound track asset

AVMutableComposition *composition = [AVMutableComposition composition];

// Video Track
AVMutableCompositionTrack *videoTrack =
    [composition addMutableTrackWithMediaType:AVMediaTypeVideo
                          preferredTrackID:kCMPersistentTrackID_Invalid];
// Audio Track
AVMutableCompositionTrack *audioTrack =
    [composition addMutableTrackWithMediaType:AVMediaTypeAudio
                          preferredTrackID:kCMPersistentTrackID_Invalid];
```

上面的示例创建了一个AVMutableComposition并用它的addMutableTrackWithMediaType:preferredTrackID:方法添加了两个轨道对象。当创建组合轨道时，开发者必须指明它所能支持的媒体类型，并给出一个轨道标识符。设置preferredTrackID:参数为CMPersistentTrackID，这是一个32位的整数值。虽然我们可以传递任意标识符作为参数，这个标识符在我们之后需要返回轨道时会用到，不过一般来说都是赋给它一个kCMPersistentTrackID_Invalid常量。这个有着奇怪名字的常量的意思是我们需要将创建一个合适轨道ID的任务委托给框架，标识符会以1…n排列。

现在我们已经实现了一个类似图9-5所示的组合资源了。组合的过程还是比较高效的，不过方法为我们提供了必要的轨道编排来将媒体片段添加到单独轨道中。

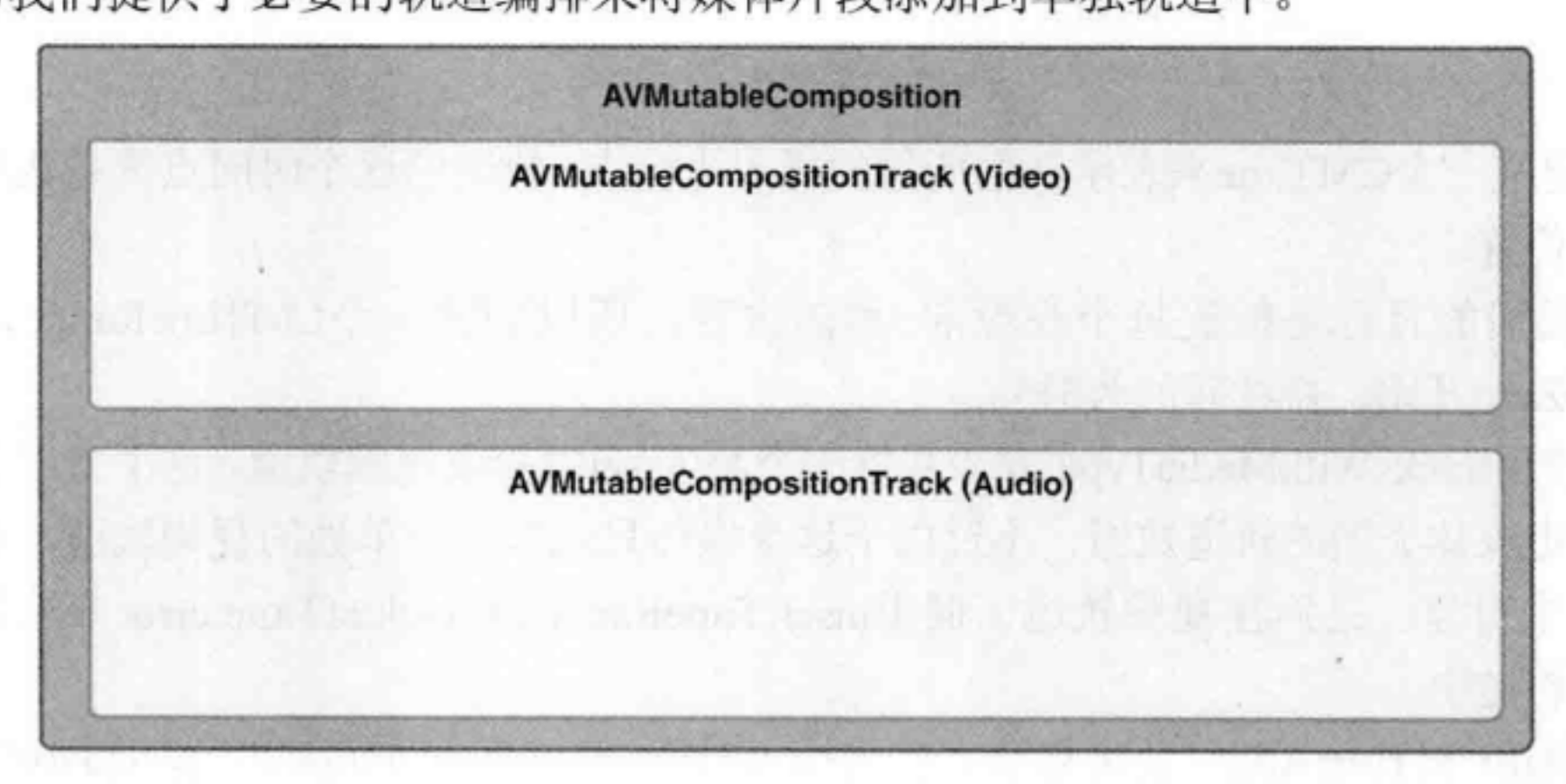

图 9-5　组合状态

下一步就是将独立的媒体片段插入到组合的轨道内，如下面的代码示例所示。这段代码有点复杂，所以我们在相关的步骤周围加入了一些标注。

```
// The "insertion cursor" time
CMTime cursorTime = kCMTimeZero;                                          // 1

CMTime videoDuration = CMTimeMake(5, 1);                                  // 2
CMTimeRange videoTimeRange = CMTimeRangeMake(kCMTimeZero, videoDuration);

AVAssetTrack *assetTrack;
```

```
// Extract and insert Golden Gate Segment
assetTrack =                                                          // 3
   [[goldenGateAsset tracksWithMediaType:AVMediaTypeVideo] firstObject];
[videoTrack insertTimeRange:videoTimeRange
                ofTrack:assetTrack
                 atTime:cursorTime error:nil];

// Increment cursor time
cursorTime = CMTimeAdd(cursorTime, videoDuration);                    // 4

// Extract and insert Tea Garden segment
assetTrack =                                                          // 5
   [[teaGardenAsset tracksWithMediaType:AVMediaTypeVideo] firstObject];
[videoTrack insertTimeRange:videoTimeRange
                ofTrack:assetTrack
                 atTime:cursorTime error:nil];

// Reset cursor time
cursorTime = kCMTimeZero;                                             // 6
CMTime audioDuration = composition.duration;
CMTimeRange audioTimeRange = CMTimeRangeMake(kCMTimeZero, audioDuration);

// Extract and insert Tea Garden segment
assetTrack =                                                          // 7
   [[soundtrackAsset tracksWithMediaType:AVMediaTypeAudio] firstObject];
[audioTrack insertTimeRange:audioTimeRange
                ofTrack:assetTrack
                 atTime:cursorTime error:nil];
```

(1) 定义一个CMTime来表示我们所指的插入光标点。轨道的这个时间点就是我们插入媒体片段的位置。

(2) 我们的目标是捕捉每个视频前5秒的内容，所以创建一个CMTimeRange，令其从kCMTimeZero开始，持续时间为5秒。

(3) 使用tracksWithMediaType:方法从第一个AVAsset中提取视频轨道。这个方法会返回一个匹配给定媒体类型的轨道数组，不过由于这个媒体只包含一个单独的视频轨道，所以我们只取第一个对象。之后在视频轨道上调用insertTimeRange:ofTrack:atTime:error方法将视频片段插入到轨道中。

(4) 调用CMTimeAdd函数来移动光标的插入时间，将videoDuration和当前cursorTime值相加。这会向前移动光标，这样下一段内容就可以在另一段内容最后插入了。

(5) 与第3步一样，提取资源视频轨道并将它插入组合资源的视频轨道上。

(6) 我们希望使音频轨道覆盖整个视频剪辑，所以首先重新设置cursorTime回到kCMTimeZero。之后获取组合资源的总duration值并创建一个CMTimeRange，令它扩展为整个组合时长。

(7) 从音频资源中提取音频轨道并将它插入组合的音频轨道。

现在我们的组合资源如图9-6所示。

图 9-6　完成的组合

这个组合的资源现在与其他任何AVAsset一样，也就是说可以播放、导出或处理。利用多种媒体资源创建一个多轨道的组合是AV Foundation的一个强大且实用的功能。

9.4　15 Seconds 示例应用程序

接下来的几章将创建一个视频编辑应用程序，名为15 Seconds，如图9-7所示。在Chapter 9目录中可以找到一个名为FifteenSeconds_Starter的启动项目。

从名字大概可以推测出，这个应用程序会创建一个15秒的媒体组合。应用程序会将目前为止我们所学到的知识进行整合，包括视频播放、读取元数据信息、图片提取，并为我们学习AV Foundation编辑功能打下坚实基础。下面先来讨论这个应用程序的一些关键功能来了解它的具体功能。

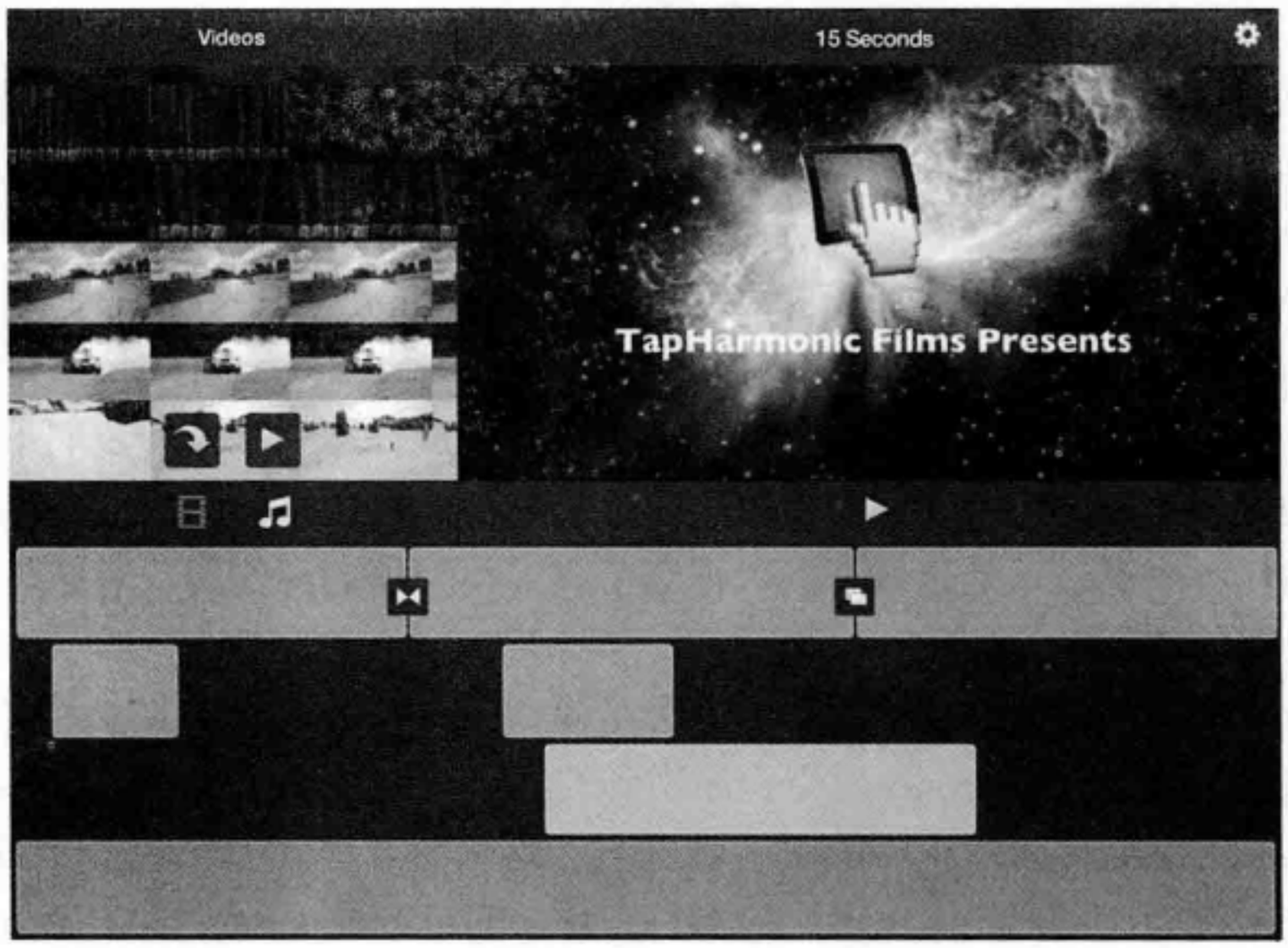

图 9-7　15 Seconds 应用程序界面

视图控制器的组合

应用程序包含3个不同的部分，分别是：一个用于选择和预览音频及视频剪辑的媒体捕捉器，一个视频播放器和一个允许排列组合中媒体剪辑的时间轴区域。图9-8给出了这个应用程序视图控制器对象的示意图。

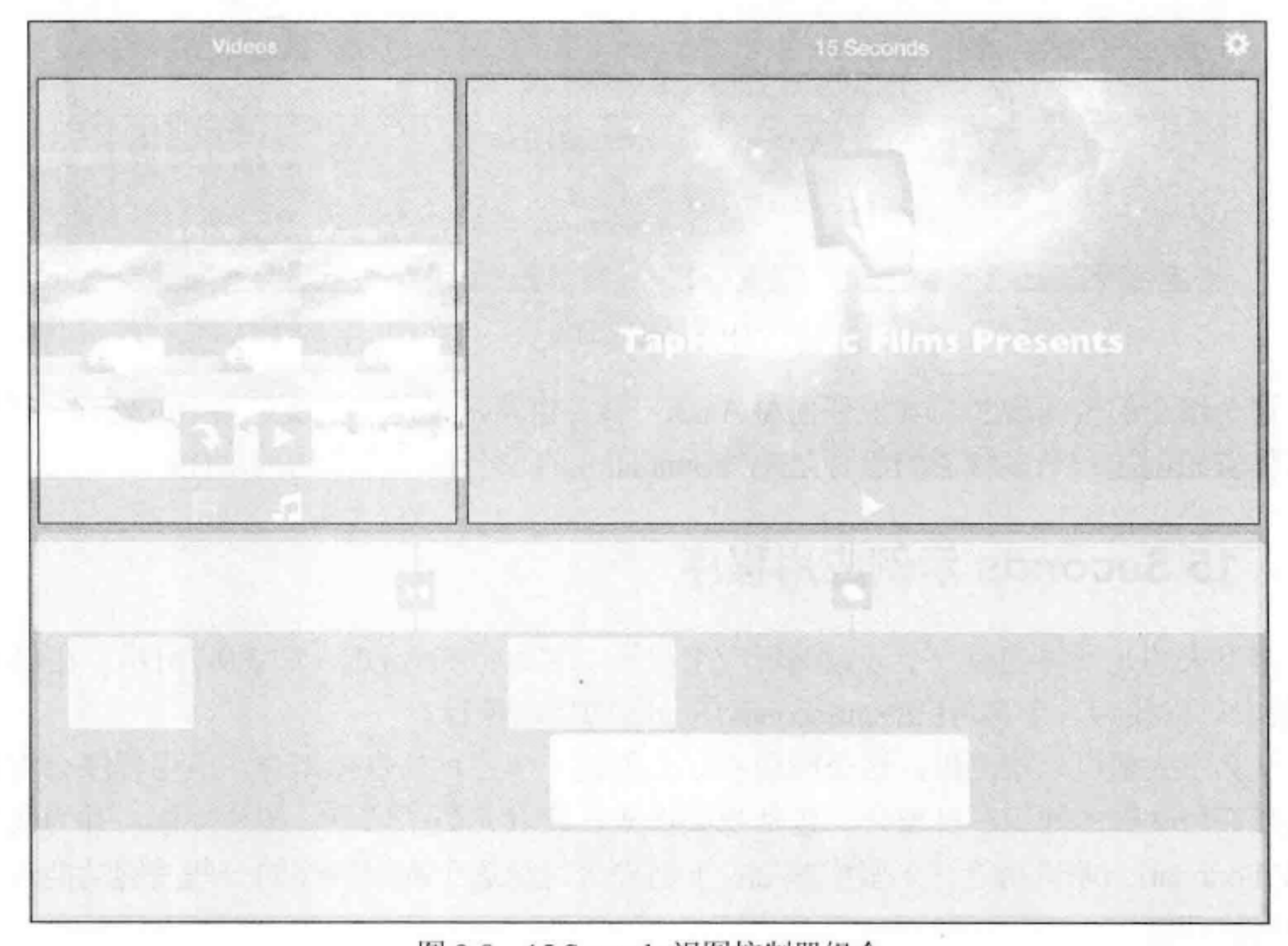

图 9-8 15 Seconds 视图控制器组合

THMainViewController是一个容器视图控制器，负责管理它的子类视图控制器并对应用程序中的请求处理进行集中协调。比如，当从THVideoPickerViewController或THVideoPickerViewController接收到一个预览媒体对象的请求时，选中的媒体对象会传递到主视图控制器，创建相应的AVPlayerItem，并将它传递给THPlayerViewController进行播放。同样，当用户点击播放器上的Play按钮时，主视图控制器会接收到一个请求，获取THTimelineViewController的当前状态，并创建一个可播放格式的时间轴传递回播放视图控制器。我们不需要关心每个视图控制器的责任范围，因为这些视图控制器要么执行前几章中介绍的任务，要么执行现在不需要了解的任务。不过还是建议开发者详细了解应用程序的源代码，对这些代码段如何组织在一起有更深的认识。

对于这个应用程序，我们需要创建的部分是模型，这需要一些有关应用程序数据模型的知识。图9-9给出了一个15 Seconds应用程序用到的组成数据模型的核心对象示意图。

THTimeline对象支持应用程序的时间轴区域。这个对象包含一个或多个THTimelineItem实例，用来表示我们在用户界面上看到的彩色方块。THTimelineItem与AV Foundation没有依赖关系，只是定义了内容的时长和它在时间轴上的位置。当处理音频和视频媒体时，会用到时间轴项的子类THMediaItem。这个类是AVAsset实例的封装并负责为组合内容合理地载入和

准备资源。THMediaItem进一步子类化，分别针对视频和音频的处理给出THVideoItem和THAudioItem两个类。本章中，音频项和视频项的区别并不重要，不过在后续章中就要特殊区别对待它们了。

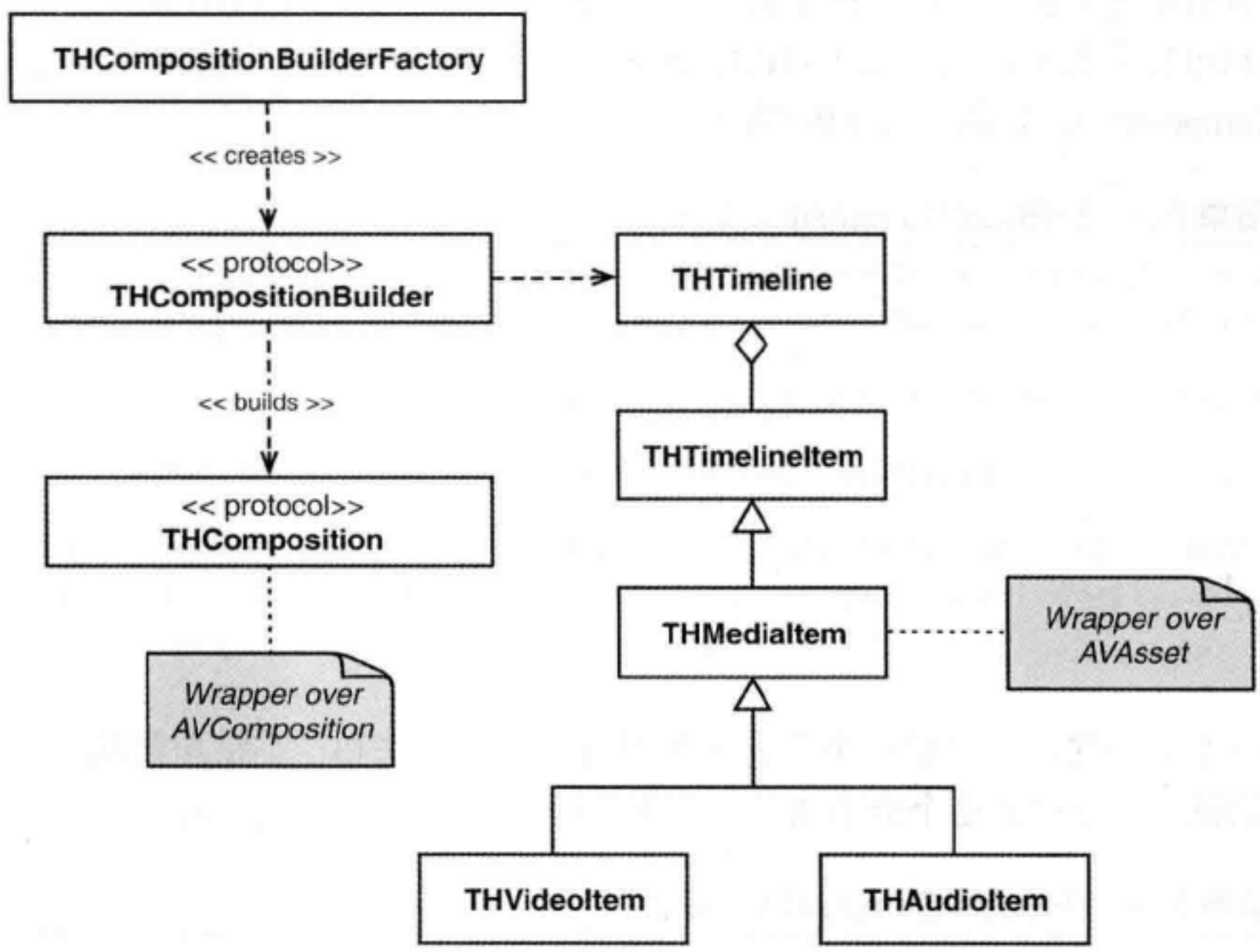

图 9-9　15 Seconds 对象模型

当请求播放当前时间轴状态时，THMainViewController会协同多个对象开始处理该请求。首先从时间轴视图控制器中获取当前的THTimeline实例，从相关的工场对象中获取THCompositionBuilder实例。THCompositionBuilder协议定义了创建THComposition实例的接口。一个具体的组合创建器负责创建实际的AVComposition和相关的轨道和片段，并将它们封装到一个THComposition实例中。这是封装AVComposition实例的又一个协议类型，它提供了一些方法将组合对象转换为可播放或可导出的格式。核心的AV Foundation处理发生在THCompositionBuilder和THComposition对象上，所以本章和后续章节为这些类创建了具体的实现代码。

现在我们已经了解到应用程序和其相关的数据模型对象是如何组合在一起的，可以开始实现应用程序的具体功能了。

9.5　创建一个组合

第一项任务就是创建一个遵循THComposition协议的对象，如代码清单9-1所示。

代码清单 9-1　THComposition 协议

```
#import <AVFoundation/AVFoundation.h>

@protocol THComposition <NSObject>

- (AVPlayerItem *)makePlayable;
```

```
- (AVAssetExportSession *)makeExportable;

@end
```

这里为后面几章创建的应用程序定义一个通用的接口。这个接口用来创建一个组合的可播放版本和可导出版本，它将用于THMainViewController类。在这个协议中首先实现的类是THBasicComposition，如代码清单9-2所示。

代码清单 9-2 THBasicComposition 接口

```
#import <AVFoundation/AVFoundation.h>
#import "THComposition.h"

@interface THBasicComposition : NSObject <THComposition>

@property (strong, readonly, nonatomic) AVComposition *composition;

+ (instancetype)compositionWithComposition:(AVComposition *)composition;
- (instancetype)initWithComposition:(AVComposition *)composition;

@end
```

这是一个简单接口，包括一个带有两个可变初始化方法和一个指向基础AVComposition对象的只读指针。下面为这个类添加具体实现代码，如代码清单9-3所示。

代码清单 9-3 THBasic Composition 实现

```
#import "THBasicComposition.h"

@interface THBasicComposition ()
@property (strong, nonatomic) AVComposition *composition;
@end

@implementation THBasicComposition

+ (id)compositionWithComposition:(AVComposition *)composition {
    return [[self alloc] initWithComposition:composition];
}

- (id)initWithComposition:(AVComposition *)composition {
    self = [super init];
    if (self) {
        _composition = composition;
    }
    return self;
}

- (AVPlayerItem *)makePlayable {                                        // 1
    return [AVPlayerItem playerItemWithAsset:[self.composition copy]];
}

- (AVAssetExportSession *)makeExportable {                              // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
    return [AVAssetExportSession exportSessionWithAsset:[self.composition copy]
                                                presetName:preset];
```

```
}

@end
```

(1) makePlayable方法会从基础AVComposition实例创建一个AVPlayerItem。AVComposition遵循NSCopying协议，所以正确的做法是通过调用copy方法得到这个对象的一个不可变副本。这样可以防止在呈现时对象状态发生改变。

(2) 在makeExportable方法内创建一个新的AVAssetExportSession实例，再次得到一个AVMutableComposition的副本，传递给初始化方法exportSessionWithAsset:presetName:。

这样我们就完成了THComposition协议实现。现在需要实现一个构建器来创建对象的实例。

创建一个组合构建器

对于每个THComposition实现，应用程序内都会有一个相应的THCompositionBuilder负责构建实例。这个协议如代码清单9-4所示。

代码清单 9-4　THCompositionBuilder 协议

```
#import <AVFoundation/AVFoundation.h>
#import "THComposition.h"

@protocol THCompositionBuilder <NSObject>

- (id <THComposition>)buildComposition;
@end
```

这个协议的具体实例负责提供buildComposition方法的实现，它会创建AVComposition以及相关的轨道和轨道片段。我们需要创建一个THBasicCompositionBuilder对象实现该协议，代码清单9-5给出了这个类的接口。

代码清单 9-5　THBasicCompositionBuilder 接口

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THBasicCompositionBuilder : NSObject <THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;
@end
```

这个类遵循THCompositionBuilder协议并提供一个initWithTimeline:方法，参数为THTimeline的实例。时间轴对象保存所有关于THTimelineItem实例和其基础AVAsset实例的引用，创建一个AVComposition需要的所有资源都在这里。代码清单9-6给出了这个类的具体实现。

代码清单 9-6　THBasicCompositionBuilder 实现

```
#import "THBasicCompositionBuilder.h"
#import "THBasicComposition.h"
#import "THFunctions.h"

@interface THBasicCompositionBuilder ()
```

```
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@end

@implementation THBasicCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition composition];                    // 1

    [self addCompositionTrackOfType:AVMediaTypeVideo
                  withMediaItems:self.timeline.videos];

    [self addCompositionTrackOfType:AVMediaTypeAudio
                  withMediaItems:self.timeline.voiceOvers];

    [self addCompositionTrackOfType:AVMediaTypeAudio
                  withMediaItems:self.timeline.musicItems];

    // Create and return the basic composition                                // 2
    return [THBasicComposition compositionWithComposition:self.composition];
}

- (void)addCompositionTrackOfType:(NSString *)mediaType
                  withMediaItems:(NSArray *)mediaItems {

    // To be implemented

}

@end
```

(1) 在buildComposition方法中首先创建了一个新的AVMutableComposition实例。为时间轴中的每个元素调用私有方法addCompositionTrackOfType:withMediaItems。这个方法的具体实现稍后会介绍。

(2) 最后，当组合对象和所有轨道都创建完毕后，还需要创建一个我们之前介绍过的THBasicComposition对象的实例，将最新创建的AVMutableComposition对象的引用传递给它。

下面就着手实现私有方法addCompositionTrackOfType:withMediaItems，它在这个类中起到关键作用，如代码清单9-7所示。

代码清单 9-7 创建轨道内容

```
@implementation THBasicCompositionBuilder

...

- (void)addCompositionTrackOfType:(NSString *)mediaType
```

```
                withMediaItems:(NSArray *)mediaItems {

    if (!THIsEmpty(mediaItems)) {                                          // 1

        CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;

        AVMutableCompositionTrack *compositionTrack =                      // 2
            [self.composition addMutableTrackWithMediaType:mediaType
                                    preferredTrackID:trackID];
        // Set insert cursor to 0
        CMTime cursorTime = kCMTimeZero;                                   // 3

        for (THMediaItem *item in mediaItems) {

            if (CMTIME_COMPARE_INLINE(item.startTimeInTimeline,            // 4
                                      !=,
                                      kCMTimeInvalid)) {
                cursorTime = item.startTimeInTimeline;
            }

            AVAssetTrack *assetTrack =                                     // 5
                [[item.asset tracksWithMediaType:mediaType] firstObject];

            [compositionTrack insertTimeRange:item.timeRange               // 6
                                      ofTrack:assetTrack
                                       atTime:cursorTime
                                        error:nil];

            // Move cursor to next item time
            cursorTime = CMTimeAdd(cursorTime, item.timeRange.duration);   // 7
        }
    }
}
@end
```

(1) 使用THIsEmpty函数，首先验证传进方法中的mediaItems数组是否为空。它是一个泛型函数，用于验证Foundation框架的各个类型是否包含有效内容。

(2) 每次调用这个方法就会创建一个带有指定媒体类型的AVMutableCompositionTrack实例。使用常量kCMPersistentTrackID_Invalid来表示AV Foundation应该自动生成一个正确的轨道ID。这样轨道ID的序号就由1到n进行分配。

(3) 创建一个新的带有kCMTimeZero常量的CMTime对象，它作为插入光标时间。

(4) 执行一个快速检查来确定时间轴对象的startTimeInTimeline是否返回一个有效的CMTime值。时间轴视频和音乐轨道中包含的片段将为startTimeInTimeline返回kCMTimeInvalid，因为它们都是按序排列的，这就意味着应用程序不允许剪辑有间隙。不过配音轨道允许在时间轴的任意点上滑动剪辑。本例希望修改cursorTime来在时间轴上显示这个剪辑的用户定义的位置。

(5) 从条目的基础AVAsset实例中提取与请求的mediaType相应的轨道，可能是AVMediaTypeVideo或AVMediaTypeAudio。这个方法返回一个NSArray数组，其中包含了与给定媒体类型匹配的轨道，不过所有示例项目中的媒体都只包含一个单独的给定类型的轨道，所以我们

只取数组中的第一个对象即可。

(6) 使用条目指定的timeRange，在计算的cursorTime值处将媒体剪辑插入到组合轨道中。

(7) 最后，将当前剪辑的时长数据和cursorTime相加，计算得到一个新的cursorTime值，这样就为下一个循环迭代设置了正确的时间点。

终于到了最后的关键时刻，下面运行应用程序并将3个视频剪辑从视频选取器中加入到时间轴，添加一个单独的音乐轨道和一个单独的配音轨道，点击Play按钮来欣赏我们的大作。可以拖拽配音剪辑到时间轴上的任意位置，改变视频剪辑的顺序并再次点击Play按钮。我们实现了一个真正的视频编辑器！

9.6 导出组合

可以创建一个组合一定会很有趣，不过如果不能把这些有创造力的结果分享到全世界岂不是毫无价值了。所以我们还要学习如何将组合导出。如果我们点击屏幕右上角的齿轮图标，可以看到有一个选项为Export Composition(导出组合)。目前点击这个选项不会有任何动作，那下面就下面实现这个功能，这一过程会用到THCompositionExporter对象，如代码清单9-8所示。

代码清单 9-8 THCompositionExporter 接口

```
#import "THTimeline.h"
#import "THComposition.h"

@interface THCompositionExporter : NSObject

@property (nonatomic) BOOL exporting;
@property (nonatomic) CGFloat progress;

- (instancetype)initWithComposition:(id <THComposition>)composition;

- (void)beginExport;

@end
```

这个对象定义的一个核心方法称为beginExport，这个方法负责实际的导出过程。在导出过程中，exporting和progress属性的状态通过THMainViewController监听，进而更新用户界面上的展示状态。代码清单9-9给出了这个类的具体实现。

代码清单 9-9 导出组合

```
#import "THCompositionExporter.h"
#import "UIAlertView+THAdditions.h"
#import <AssetsLibrary/AssetsLibrary.h>

@interface THCompositionExporter ()
@property (strong, nonatomic) id <THComposition> composition;
@property (strong, nonatomic) AVAssetExportSession *exportSession;
@end

@implementation THCompositionExporter
```

```
- (instancetype)initWithComposition:(id <THComposition>)composition {
    self = [super init];
    if (self) {
        _composition = composition;
    }
    return self;
}

- (void)beginExport {
    self.exportSession = [self.composition makeExportable];                    // 1
    self.exportSession.outputURL = [self exportURL];
    self.exportSession.outputFileType = AVFileTypeMPEG4;

    [self.exportSession exportAsynchronouslyWithCompletionHandler:^{          // 2

        // To be implemented

    }];

    self.exporting = YES;                                                      // 3
    [self monitorExportProgress];                                              // 4
}

- (void)monitorExportProgress {
    // To be implemented
}

- (NSURL *)exportURL {
    NSString *filePath = nil;
    NSUInteger count = 0;
    do {
        filePath = NSTemporaryDirectory();
        NSString *numberString = count > 0 ?
            [NSString stringWithFormat:@"-%li", (unsigned long) count] : @"";
        NSString *fileNameString =
            [NSString stringWithFormat:@"Masterpiece-%@.m4v", numberString];
        filePath = [filePath stringByAppendingPathComponent:fileNameString];
        count++;
    } while ([[NSFileManager defaultManager] fileExistsAtPath:filePath]);

    return [NSURL fileURLWithPath:filePath];
}

@end
```

(1) 首先创建一个组合的可导出版本。这会返回一个AVAssetExportSession实例，这个实例有基础AVComposition实例，且具有相应的会话预设值。我们为这个会话动态生成一个唯一的输出URL并设置输出文件类型。

(2) 调用exportAsynchronouslyWithCompletionHandler:方法开始导出过程。我们暂时提供一个空的handler block作为方法的参数。

(3) 设置exporting属性为YES。THMasterViewController会监听该属性的变化，并呈现用

户界面的进展。

(4) 最后，调用monitorExportProgress方法，轮询导出会话的状态来确定当前进度。

继续看一下monitorExportProgress方法的实现，如代码清单9-10所示。

代码清单 9-10 监视导出过程

```
@implementation THCompositionExporter

...

- (void)monitorExportProgress {
    double delayInSeconds = 0.1;
    int64_t delta = (int64_t)delayInSeconds * NSEC_PER_SEC;
    dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, delta);

    dispatch_after(popTime, dispatch_get_main_queue(), ^{                    // 1

        AVAssetExportSessionStatus status = self.exportSession.status;

        if (status == AVAssetExportSessionStatusExporting) {                 // 2

            self.progress = self.exportSession.progress;
            [self monitorExportProgress];

        } else {
            self.exporting = NO;
        }
    });
}

...

@end
```

(1) 首先在一个短暂的延迟之后将一段执行代码放入队列，用来检查导出过程的状态。

(2) 检查导出会话的status属性来确定其当前状态。如果状态返回AVAssetExport-SessionStatusExporting，则用当前导出会话的进度值更新progress属性，并递归调用monitorExportProgress方法。如果导出会话的status属性返回任何其他值，则设置exporting属性为NO，这样用户界面就可以相应地更新了。

最后一点需要实现的功能就是，处理导出语句并将导出的文件写入Assets Library，如代码清单9-11所示。

代码清单 9-11 处理导出语句

```
@implementation THCompositionExporter

...

- (void)beginExport {

    self.exportSession = [self.composition makeExportable];
    self.exportSession.outputURL = [self exportURL];
    self.exportSession.outputFileType = AVFileTypeMPEG4;
```

```
[self.exportSession exportAsynchronouslyWithCompletionHandler:^{
    dispatch_async(dispatch_get_main_queue(), ^{                                    // 1
        AVAssetExportSessionStatus status = self.exportSession.status;
        if (status == AVAssetExportSessionStatusCompleted) {
            [self writeExportedVideoToAssetsLibrary];
        } else {
            [UIAlertView showAlertWithTitle:@"Export Failed"
                                    message:@"The requested export failed."];
        }
    });
}];

self.exporting = YES;
[self monitorExportProgress];
}

- (void)writeExportedVideoToAssetsLibrary {
  NSURL *exportURL = self.exportSession.outputURL;
  ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

  if ([library videoAtPathIsCompatibleWithSavedPhotosAlbum:exportURL]) {    // 2

    [library writeVideoAtPathToSavedPhotosAlbum:exportURL                   // 3
                          completionBlock:^(NSURL *assetURL,
                                            NSError *error) {

      if (error) {                                                          // 4
        NSString *message = @"Unable to write to Photos library.";
        [UIAlertView showAlertWithTitle:@"Write Failed"
                                message:message];
      }

      [[NSFileManager defaultManager] removeItemAtURL:exportURL
                                              error:nil];
    }];
  } else {
    NSLog(@"Video could not be exported to the assets library.");
  }
}

@end
```

(1) 在handler block中，首先调度回主队列并检查导出会话状态。如果导出成功完成，则将导出的视频文件写入Assets Library。如果导出失败，则弹出一个错误提示框给用户。

(2) 创建一个新的ALAssetsLibrary实例。在尝试写入库之前，最好验证一下即将写入的内容是否可以被写入，通过调用videoAtPathIsCompatibleWithSavedPhotosAlbum:方法进行这一验证。

(3) 调用writeVideoAtPathToSavedPhotosAlbum:completionBlock:方法开始写入操作。第一次尝试时，会接收到一个安全提示，询问是否同意向Photos Library写入内容，选择同意。

(4) 最后，在completion handler中判断是否有错误出现。如果有，展示一个相应的错误提

示框给用户。不管写入操作的结果如何，都需要将用户临时目录中的导出文件删除。

运行该应用程序，可手动创建一个新的组合或从菜单选择Load DefaultComposition选项。点击屏幕角落处的齿轮图标并选择Export Composition选项。将会看到视频播放器上出现一个表示导出进展的进度条。当导出完成后，可以切换到相机应用程序欣赏我们的作品。

9.7 小结

本章你第一次接触到AV Foundation的媒体编辑功能。你使用AVComposition创建了一个多轨道的项目，将一些视频和音频资源进行组合创建出一个新的唯一的媒体片段。你还学习了如何像其他任何AVAsset那样，为播放和导出等任务使用组合。虽然我们的讨论集中在创建一个传统的视频编辑应用程序，不过会发现AVComposition其实适用于很多场景。

你已经开了一个好头，不过有趣的部分才刚刚开始！下面3章的内容将讨论如何实现剪切编辑的应用程序并为应用程序进一步添加诸多高级功能。

9.8 挑战

学习有效使用AV Foundation的编辑功能，其中最重要的一点就是熟练掌握有关CMTime和CMTimeRange的知识。当开始学习使用编辑类API时，以有理数表示的时间格式经常会让开发者感到困惑。在Chapter 9目录中可找到一个简单的Mac命令行项目TimeChallenge，你可以用这个项目来学习这些数据类型和功能。该项目给出了一些示例和建议供你尝试。

第10章

混合音频

15 Seconds应用程序进展的还不错，现在已经具备了创建多轨、播放剪辑组合及导出并分享到世界各地等多种能力。核心功能都已经没有问题了，现在我们要把注意力放在基于AV Foundation所提供的各种附加功能上，使用这些附加功能来提升和优化应用程序创建的组合。本章讨论如何通过一些基础的音频混合功能来提升组合音频方面的特性。这一章内容很少，不过所讲述的知识点可以极大地提升应用程序音频输出的质量。

10.1 混合音频

上一章创建的组合媒体看起来还不错，各场景间的切换也很清晰，不过在处理相关音频输出方面确实还有一些问题。首先就是音乐轨道刚开始播放时音量就很大，并且在组合资源结束时又突然停止，用户会觉得声音很震耳。如果开始的时候声音是渐渐增大，结束的时候声音渐渐减小，会带来更好的体验。另一个问题，也可以说更严重的问题是画外音轨道的处理。音乐轨道的音量已经完全覆盖了画外音的声音，几乎听不到画外音。与其说让这两个音频轨道发生冲突，倒不如使用一种名为闪避处理(ducking)的技术在画外音持续期间将音乐声调低，并在画外音持续时间保持这种状态，在画外音完成后再恢复到之前的音量。框架提供了一个类AVAudioMix来解决上面两个问题。

AVAudioMix用来在组合的音频轨道中进行自定义音频的处理。图10-1展示了AVAudioMix类和其相关类的示意图，还给出了框架中用到它的类。

AVAudioMix所具有的音频处理方法是由它的输入参数集定义的，它的参数是AVAudioMixInputParameters类型的对象。AVAudioMixInputParameters的实例关联组合中的单独音频轨道，并在添加到音频混合时定义基于轨道的处理方法。AVAudioMix和其相关联的AVAudioMixInputParameters集合都是不可变对象，意味着它们适用于为AVPlayerItem和AVAssetExportSession之类的客户端提供相关数据，不过它们不能操作其状态。当我们需要创建一个自定义音频混合时，需要改用它们在AVMutableAudioMix和AVMutableAudioMixInput-Parameters中的可变子类。

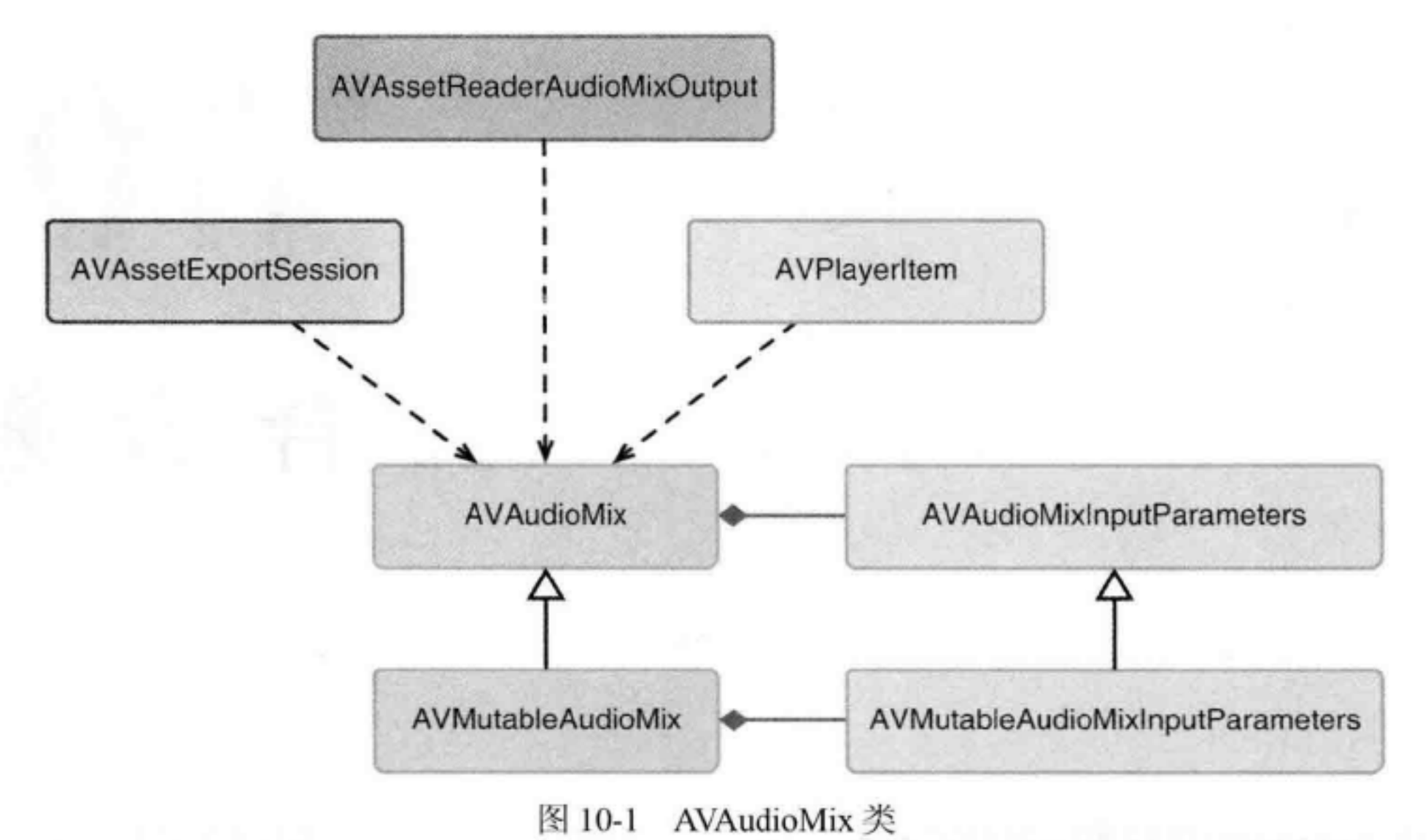

图 10-1 AVAudioMix 类

自动调节音量

在应用音频混合的问题上，最核心的处理就是调整组合音频轨道的音量。当一个组合资源播放或导出时，默认行为是以最大音量或正常音量播放音频轨道。只有一个单音频轨道时这样的方法才可能比较容易接收，不过当一个组合资源包含多个音频源时就会出现问题。对于多音频轨道的情况，每个声音都在争夺空间，这就不可避免会导致一些声音可能无法被听到。一些简单的音频混合可以解决这个问题并帮助我们得到一个清晰悦耳的混合音效。

如果你曾用过音频混合的工具，比如Pro Tools或Logic Pro X，应该比较习惯使用分贝来描述音量。AV Foundation使用一种更简单的模型，把音量定义为一个标准化的浮点型数值，数值范围从0.0～1.0，即0.0等于静音和1.0等于最大音量。音频轨道的默认音量为1.0，不过可以使用AVMutableAudioMixInputParameters实例修改这个值。这个对象允许在一个指定时间点或给定的时间范围自动调节音量，如图10-2所示。

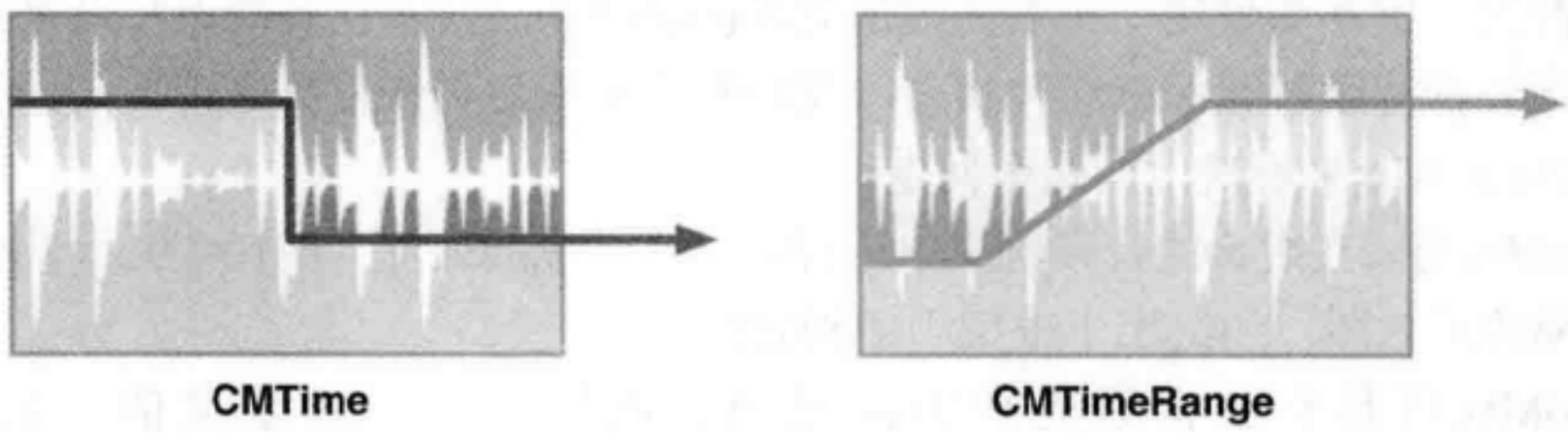

图 10-2 基于时间的音量调节

AVMutableAudioMixInputParameters提供了两个方法来实现图10-2中所示的音量调节。

- **setVolume:atTime:** 在指定时间点立即调节音量。音量在音频轨道持续时间内会保持不变，直到有另一个音量调节出现。
- **setVolumeRampFromStartVolume:toEndVolume:timeRange:** 允许在一个给定时间范围内平滑地将音量从一个值调节到另一个值。当需要在一个时间范围内调整音量

时，音量会立即变为指定值的初始音量并在持续时间之内逐渐调整为指定的结束值。音量会保持一个 toEndVolume:参数所指定的常量，直到接收另一个音量调整为止，这个参数值用于表示音频轨道的持续时间。

下面通过一个示例了解如何使用这些方法创建自动调整音量的功能，如图10-3所示。

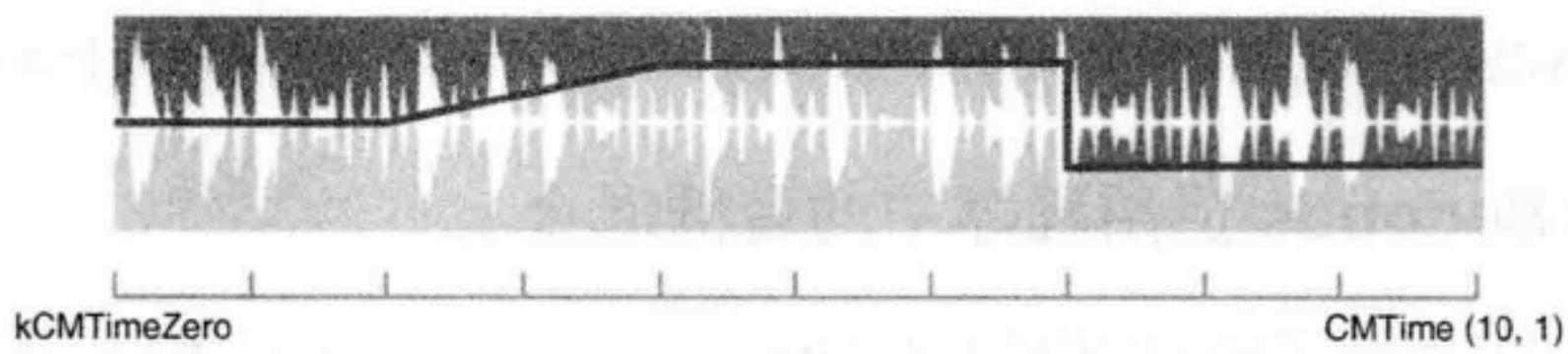

图 10-3 音量自动调节示例

```
AVCompositionTrack *track = // audio track in composition

// Define automation times
CMTime twoSeconds = CMTimeMake(2, 1);
CMTime fourSeconds = CMTimeMake(4, 1);
CMTime sevenSeconds = CMTimeMake(7, 1);

// Create a new parameters object for the given audio track
AVMutableAudioMixInputParameters *parameters =
    [AVMutableAudioMixInputParameters audioMixInputParametersWithTrack:track];

// Set initial volume
[parameters setVolume:0.5f atTime:kCMTimeZero];

// Define time range of the volume ramp
CMTimeRange range = CMTimeRangeFromTimeToTime(twoSeconds, fourSeconds);

// Perform 2 second ramp from 0.5 -> 0.8
[parameters setVolumeRampFromStartVolume:0.5f toEndVolume:0.8f timeRange:range];

// Drop volume to 0.3 at the 7-second mark
[parameters setVolume:0.3f atTime:sevenSeconds];

// Create a new audio mix instance
AVMutableAudioMix *audioMix = [AVMutableAudioMix audioMix];

// Assign the input parameters to the audio mix
audioMix.inputParameters = @[parameters];
```

示例首先创建一个新的与要操作的轨道关联的AVMutableAudioMixInputParameters实例。轨道的默认音量是1.0，按照图10-3所示我们设置开始时的音量为0.5，使用setVolume:atTime:方法并传递kCMTimeZero值来设置初始音量。在轨道中选定2秒，在这2秒期间使用setVolumeRampFromStartVolume:toEndVolume:timeRange:方法应用一个从0.5～0.8的音量渐变。最后，在7秒处使用setVolume:atTime:方法将音量下调到0.3。定义好所有参数后就可以创建AVMutableAudioMix了，将参数添加到数组中，并将数组赋给音频混合对象的inputParameters属性。示例中创建了一个全格式的音频混合，可以被设置为AVPlayerItem或AVAssetExportSession的audioMix属性进行播放或导出。

> **注意：**
>
> 我们讨论的是与在 AVComposition 中混合音频相关的 AVAudioMix 和 AVAudioMixInputParameters。虽然这看起来是一个常见用例，不过对于组合并不单独使用。AVAudioMixInputParameters 的实例和 AVAssetTrack(AVCompositionTrack 的超类)相关，这就意味着我们在一个常规的 AVAsset 上也可以定义音量调整，这就可以用于一些特定的播放和导出场景。

10.2 15 Seconds 应用程序中的音频混合

现在我们已经学会了如何使用AVAudioMix，下面在15 Second应用程序来使用它。首先创建一个新的类THAudioMixComposition，它需要遵循THComposition协议，如代码清单10-1所示。

代码清单 10-1 THAudioMixComposition 接口

```
#import "THComposition.h"

@interface THAudioMixComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVAudioMix *audioMix;
@property (strong, nonatomic, readonly) AVComposition *composition;

+ (instancetype)compositionWithComposition:(AVComposition *)composition
                                 audioMix:(AVAudioMix *)audioMix;

- (instancetype)initWithComposition:(AVComposition *)composition
                            audioMix:(AVAudioMix *)audioMix;

@end
```

这个接口看起来很熟悉，因为它和之前章节中创建的THBasicComposition对象类似。最主要的区别在于添加了AVAudioMix属性和一个带有AVComposition和AVAudioMix参数的初始化方法。下面继续分析代码清单10-2给出的具体实现。

代码清单 10-2 THAudioMixComposition 实现

```
#import "THAudioMixComposition.h"
@interface THAudioMixComposition ()
@property (strong, nonatomic) AVAudioMix *audioMix;
@property (strong, nonatomic) AVComposition *composition;
@end

@implementation THAudioMixComposition

+ (instancetype)compositionWithComposition:(AVComposition *)composition
                                 audioMix:(AVAudioMix *)audioMix {
    return [[self alloc] initWithComposition:composition audioMix:audioMix];
}

- (instancetype)initWithComposition:(AVComposition *)composition
                            audioMix:(AVAudioMix *)audioMix {
```

```
    self = [super init];
    if (self) {
        _composition = composition;
        _audioMix = audioMix;
    }
    return self;
}

- (AVPlayerItem *)makePlayable {                                          // 1
        AVPlayerItem *playerItem =
        [AVPlayerItem playerItemWithAsset:[self.composition copy]];
        playerItem.audioMix = self.audioMix;
        return playerItem;
}

- (AVAssetExportSession *)makeExportable {                               // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
        AVAssetExportSession *session =
        [AVAssetExportSession exportSessionWithAsset:[self.composition copy]
                                          presetName:preset];
        session.audioMix = self.audioMix;
        return session;
}

@end
```

(1) 在makePlayable方法中创建一个带有AVComposition实例的AVPlayerItem。设置audioMix对象作为播放器条目的audioMix属性。这样就可以在应用程序视频播放器中播放音频时应用音频处理。

(2) 在makeExportable方法中创建一个AVAssetExportSession与上一章节中的做法一样。设置audioMix对象作为输出会话的audioMix属性，这样就可以使得在导出这个组合时应用音频处理。

THAudioMixComposition就创建完成了，下面继续讨论它的关联对象THComposition-Builder的实现。首先创建一个新的对象THAudioMixCompositionBuilder，遵循THComposition-Builder协议，如代码清单10-3所示。

代码清单 10-3　THAudioMixCompositionBuilder 接口

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THAudioMixCompositionBuilder : NSObject <THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;
@end
```

THAudioMixCompositionBuilder具有一个简单的接口可以接受THTimeline实例。回顾一下，THTimeline对象包含了应用程序时间区域的状态并提供了创建组合对象所需的数据。下面开始实现这个类，如代码清单10-4所示。

代码清单 10-4 THAudioMixCompositionBuilder 实现

```
#import "THAudioMixCompositionBuilder.h"
#import "THAudioItem.h"
#import "THVolumeAutomation.h"
#import "THAudioMixComposition.h"
#import "THFunctions.h"

@interface THAudioMixCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@end

@implementation THAudioMixCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition composition];                    // 1

    [self addCompositionTrackOfType:AVMediaTypeVideo
                    withMediaItems:self.timeline.videos];

    [self addCompositionTrackOfType:AVMediaTypeAudio
                    withMediaItems:self.timeline.voiceOvers];

    AVMutableCompositionTrack *musicTrack =
        [self addCompositionTrackOfType:AVMediaTypeAudio
                        withMediaItems:self.timeline.musicItems];

    AVAudioMix *audioMix = [self buildAudioMixWithTrack:musicTrack];          // 2

    return [THAudioMixComposition compositionWithComposition:self.composition
                                                    audioMix:audioMix];
}

- (AVMutableCompositionTrack *)addCompositionTrackOfType:(NSString *)type
                                    withMediaItems:(NSArray *)mediaItems {

    if (!THIsEmpty(mediaItems)) {

        CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;

        AVMutableCompositionTrack *compositionTrack =
            [self.composition addMutableTrackWithMediaType:type
                                        preferredTrackID:trackID];
        // Set insert cursor to 0
        CMTime cursorTime = kCMTimeZero;
```

```
    for (THMediaItem *item in mediaItems) {

        if (CMTIME_COMPARE_INLINE(item.startTimeInTimeline,
                                  !=,
                                  kCMTimeInvalid)) {
            cursorTime = item.startTimeInTimeline;
        }

        AVAssetTrack *assetTrack =
            [[item.asset tracksWithMediaType:type] firstObject];

            [compositionTrack insertTimeRange:item.timeRange
                                   ofTrack:assetTrack
                                    atTime:cursorTime
                                     error:nil];

            // Move cursor to next item time
            cursorTime = CMTimeAdd(cursorTime, item.timeRange.duration);
        }

        return compositionTrack;
    }

    return nil;
}

- (AVAudioMix *)buildAudioMixWithTrack:(AVMutableCompositionTrack *)track {

    // To be implemented

    return nil;
}

@end
```

(1) 创建一个新的AVMutableComposition实例并开始添加AVCompositionTrack对象，像上一章中使用addCompositionTrackOfType:withMediaItems:方法一样。这是上一章编写的同一个方法，不过有一点小的改动是返回了一个它创建的AVMutableCompositionTrack的引用。

(2) 调用私有方法buildAudioMixWithTrack:创建一个AVAudioMix实例，我们在之后会实现它。这个方法接收应用音量调整的轨道的引用。最后，创建并返回一个新的THAudioMix-Composition实例，给它传递组合和音频混合。

在学习buildAudioMixWithTrack:方法的实现之前，我们希望简单讨论一下应用程序如何在用户界面表示音量自动调整。包含在THTimeline对象中的音频项都是THAudioItem类型，这一类型的对象封装了基础AVAsset实例并额外定义了一个属性volumeAutomation，它是THVolumeAutomation实例的数组。THVolumeAutomation用于保存音量自动调节数据并在用户界面动态创建相应的值。它提供在AVMutableAudioMixInputParameters实例上设置的创建音量参数需要的所有相关数据。下面看一下buildAudioMixWithTrack方法的实现，了解一下这个对象是如何应用的，如代码清单10-5所示。

代码清单 10-5 创建音频混合

```
- (AVAudioMix *)buildAudioMixWithTrack:(AVCompositionTrack *)track {
    THAudioItem *item = [self.timeline.musicItems firstObject];                // 1
    if (item) {
        AVMutableAudioMix *audioMix = [AVMutableAudioMix audioMix];            // 2

        AVMutableAudioMixInputParameters *parameters =
            [AVMutableAudioMixInputParameters
                audioMixInputParametersWithTrack:track];

        for (THVolumeAutomation *automation in item.volumeAutomation) {        // 3
            [parameters setVolumeRampFromStartVolume:automation.startVolume
                                          toEndVolume:automation.endVolume
                                           timeRange:automation.timeRange];
        }

        audioMix.inputParameters = @[parameters];                              // 4
        return audioMix;
    }
    return nil;
}
```

(1) 首先从时间轴得到音乐轨道的THAudioItem实例。应用程序只允许添加一个单独的音乐轨道，所以从musicItems数组中获取firstObject。

(2) 创建一个新的AVMutableAudioMix实例，用来保存输入参数。还需要创建一个新的AVMutableAudioMixInputParameters实例，将它与传递给方法的AVCompositionTrack进行关联。

(3) 遍历音频条目的volumeAutomation数组中的对象，对每个实例在parameters对象上定义一个音量渐变，传递参数startVolume、endVolume和timeRange。

(4) 将parameters对象封装在NSArray中，并设置它作为音频混合的inputParameters，并返回这个音频混合实例。

运行应用程序，可以手动创建一个自定义的组合或点击Settings菜单选择Load Default Composition。一个新的Audio小节被添加到Settings菜单，现在它包含了两个开关来切换音频处理行为，如图10-4所示。

图 10-4 设置菜单——音频控件

首先启用Fade In & Out切换开关并播放组合。可以看到出现一个音量曲线，它将我们听

到的声音以可视化的效果进行呈现。现在音乐会平缓进入或消失，对用户来说这是一个更好的体验。现在激活Audio Ducking开关，会发现更新了音量自动调节，给出了闪避行为的可视化描述。再次播放组合可以发现画外音变得更响亮和清楚了。音量自动调整的数据是动态创建的，所以我们可以滑动控制画外音的大小来感觉音量是否合适。

10.3 小结

本章我们学习了如何使用AVAudioMix和AVAudioMixInputParameters来提供对于组合音频轨道更好的控制。我们对15Seconds应用程序的改动很小且非常容易实现，不过它对应用程序的提升却很显著。音频混合在所有的音频或视频编辑应用程序中都非常重要，不过可以看出使用AVComposition时几乎都可以利用AVAudioMix类来提升音频输出的效果。

10.4 挑战

尝试对应用程序的音量自动调节功能进行自定义。会在THTimelineViewController中发现一个名为buildVolumeFadesForMusicItem:的方法，它基于用户界面中发生的变化创建音量自动调节数据。改变音频逐渐消失的时间，在音频闪避范围内以不同的音量级别进行播放，从任一个音量开始播放，或剪辑混合的音频。尽可以放手尝试，体验其中的乐趣！

第11章

创建视频过渡效果

15 Seconds应用程序给出了一种简单的方式快速实现cut-only视频编辑。随着视频时间轴的前进，两个场景间快速地过渡或切换，其中没有任何过渡效果。我们经常看到这种不带效果的过渡，不过并不只有这一种过渡方式。通常在时间轴上出现明显的变化时，比如情节或位置的变换，两个场景间会使用一些类型的动画过渡效果。诸如渐隐、溶解和擦除等视频间的过渡被广泛用于各种创意效果，并且可以支持讲述一个视频内容的故事或消息。本章介绍AV Foundation有关创建视频过渡的方法，看看这些方法是如何帮助我们提升视频编辑应用程序的。

11.1 综述

大部分视频编辑应用程序的一个重要功能就是能够创建动态视频过渡效果。AV Foundation对于这一功能的支持具有很高的可靠性，不过同时也被认为是学习媒体编辑API中最具挑战性的一个领域。这个功能是整个框架中文档介绍最少的几个功能之一，这一部分主要包括如何使用这个稍微复杂一点的API(而且较难调试)。本章内容会逐步进行讲解让你对这个问题有一个比较深入的认识，避免一些常见的陷阱。下面从学习几个创建视频过渡的类开始学习吧，如图11-1所示。

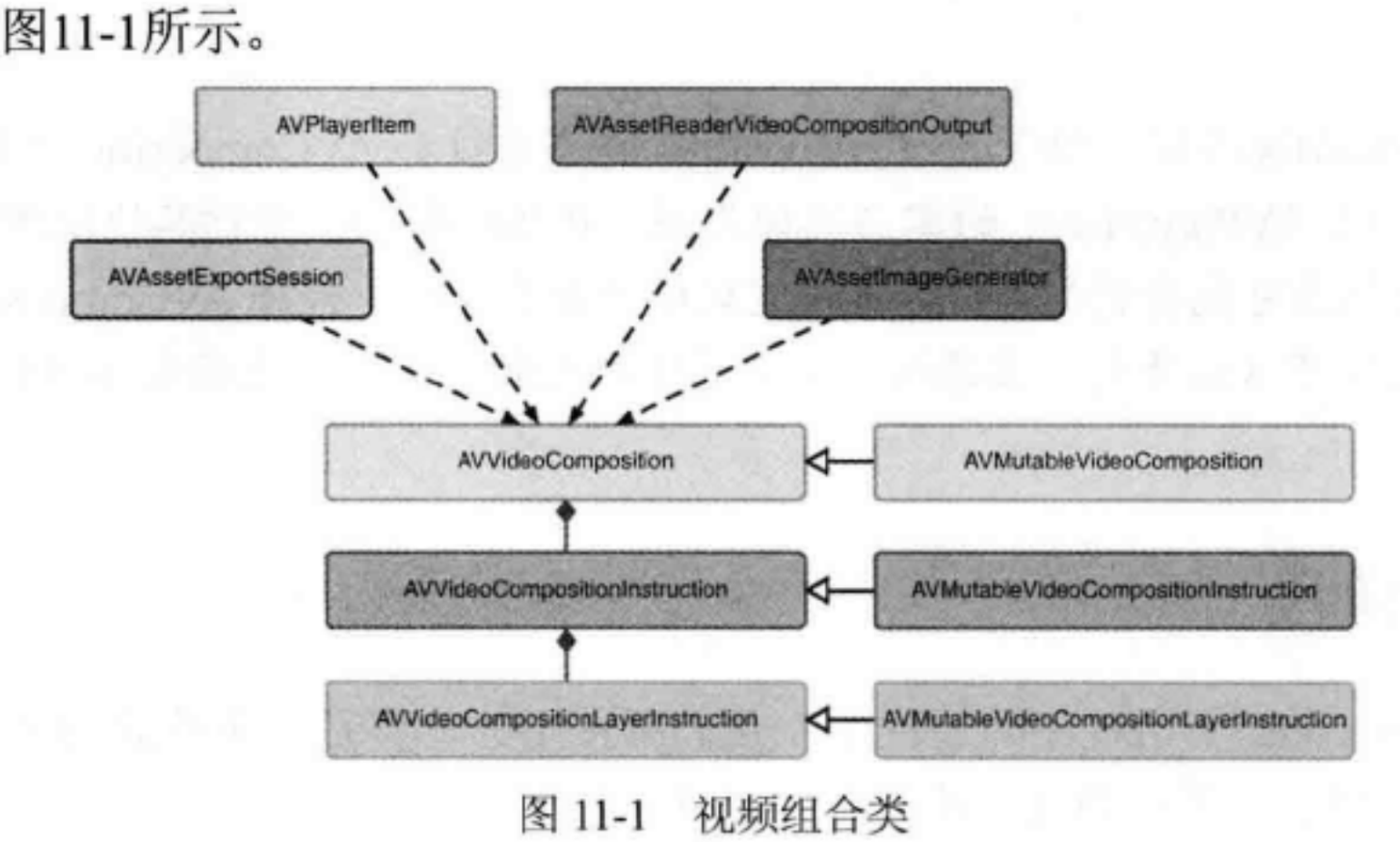

图 11-1　视频组合类

11.1.1 AVVideoComposition

框架的视频过渡类API中最核心的类是AVVideoComposition。这个类对两个或多个视频轨道组合在一起的方法给出了一个总体描述。它由一组时间范围和描述组合行为的介绍内容组成，这些信息出现在组合资源内的任意时间点。除了包含描述输入视频层组合的信息之外，还提供了配置视频组合的渲染尺寸、缩放和帧时长等属性。视频组合的配置确定了委托对象处理时AVComposition的呈现方式，这里的委托对象比如AVPlayer或AVAssetImageGenerator。

> **注意：**
>
> 一个值得重点注意的是，不管名字是如何定义的，AVVideoComposition 并不是AVComposition 的子类，甚至没有直接关联。在视频播放、导出或处理时会用这个类来控制资源视频轨道的视频组合行为。

11.1.2 AVVideoCompositionInstruction

AVVideoComposition是由一组AVVideoCompositionInstruction对象格式定义的指令组成的。这个对象所提供的最关键的一段数据是组合对象时间轴内的时间范围信息，这一时间范围是在某一组合形式出现时的时间范围。要执行的组合特质是通过其layerInstructions集合定义的。

11.1.3 AVVideoCompositionLayerInstruction

AVVideoCompositionLayerInstruction用于定义对给定视频轨道应用的模糊、变形和裁剪效果。它提供了一些方法用于在特定的时间点或在一个时间范围内对这些值进行修改。在一段时间内对这些值应用渐变操作可以让开发者创建出动态的过渡效果，比如溶解和渐淡效果。

与所有的AV Foundation媒体编辑类一样，视频组合API具有不可变和可变两种形式。不可变超类形式适用于客户端对象，比如AVAssetExportSession，不过当我们创建自己的视频组合应用程序时，应该使用不可变子类。

> **注意：**
>
> 与 AVAudioMix 类似，AVVideoComposition 并不直接和 AVComposition 相关。相反，这些对象是和类似 AVPlayerItem 的客户端相关联，在播放组合或进行其他处理时使用这些对象。第一眼看起来可能会觉得奇怪，不过这么设计是有益的。未将 AVComposition 与输出行为强耦合，这样可以让开发者在播放、导出或处理视频时更灵活地确定如何使用这些行为。

11.2 概念阶段

在开发15 Seconds应用程序的过程中，我们所使用的一直是一个单独的视频轨道，其中按时间轴顺序排列了一系列的视频剪辑，如图11-2所示。

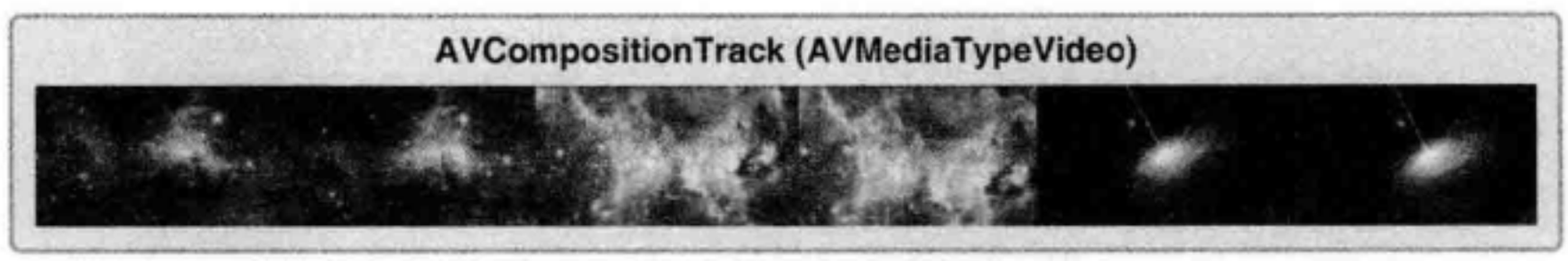

图 11-2　目前的视频轨道布局

这一轨道的排列在创建一个cut-only编辑时很清楚地知道需要什么。不过当需要在独立的视频分段间实现一个动态过渡效果时，这一排列无法满足我们的需要。学习创建过渡效果的有效方法是将这一过程分解为相关的小步骤，这些步骤确实是概念步骤，用于帮助理解这一过程，不过当我们熟悉了这些概念后，会经常将两个或多个概念进行组合变成一个总体的概念了。

11.2.1　部署视频布局

要在剪辑间添加过渡，首先需要将两个轨道间的视频片段重新部署。我们已经知道如何将多个音频轨道组合在一起，那么视频的情况也是一样的。大多数情况下两个轨道就足够了，不过为了满足一些特殊需求而加入更多的轨道也是可以的，要注意同时添加过多的轨道会对性能产生负面影响。首先创建一个包含两个视频轨道的组合，如图11-3所示。

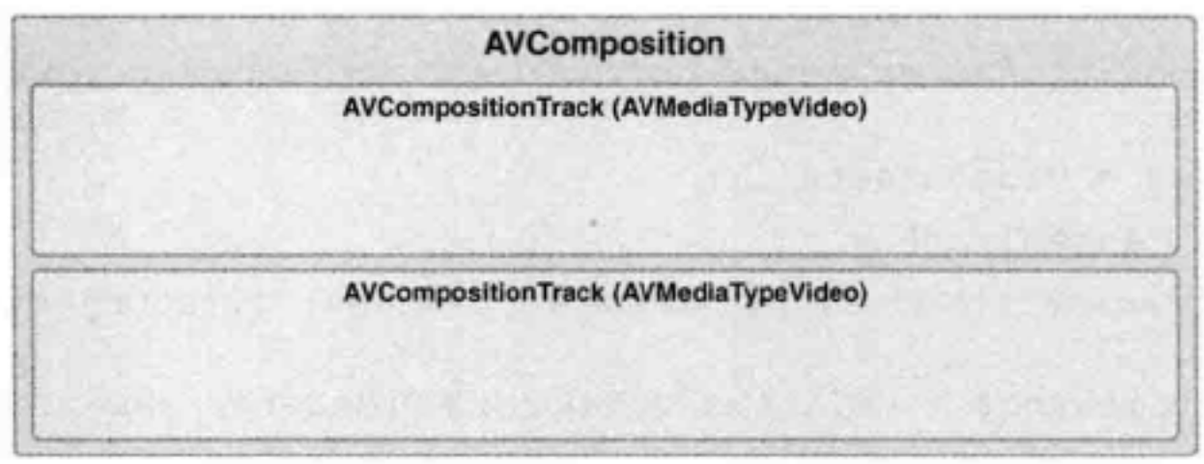

图 11-3　带有两个视频轨道的 AVComposition

构建上图中组合的代码如下所示。

```
AVMutableComposition *composition = [AVMutableComposition composition];

AVMutableCompositionTrack *trackA =
    [composition addMutableTrackWithMediaType:AVMediaTypeVideo
                                preferredTrackID:kCMPersistentTrackID_Invalid];

AVMutableCompositionTrack *trackB =
    [composition addMutableTrackWithMediaType:AVMediaTypeVideo
                                preferredTrackID:kCMPersistentTrackID_Invalid];

NSArray *videoTracks = @[trackA, trackB];
```

上面的示例首先创建了一个新的可变组合并添加了两个AVMediaTypeVideo类型的组合轨道。将组合轨道添加到一个NSArray中备以后使用。

所需的轨道安排好之后，需要错开部署这两个轨道的视频布局，如图11-4所示。

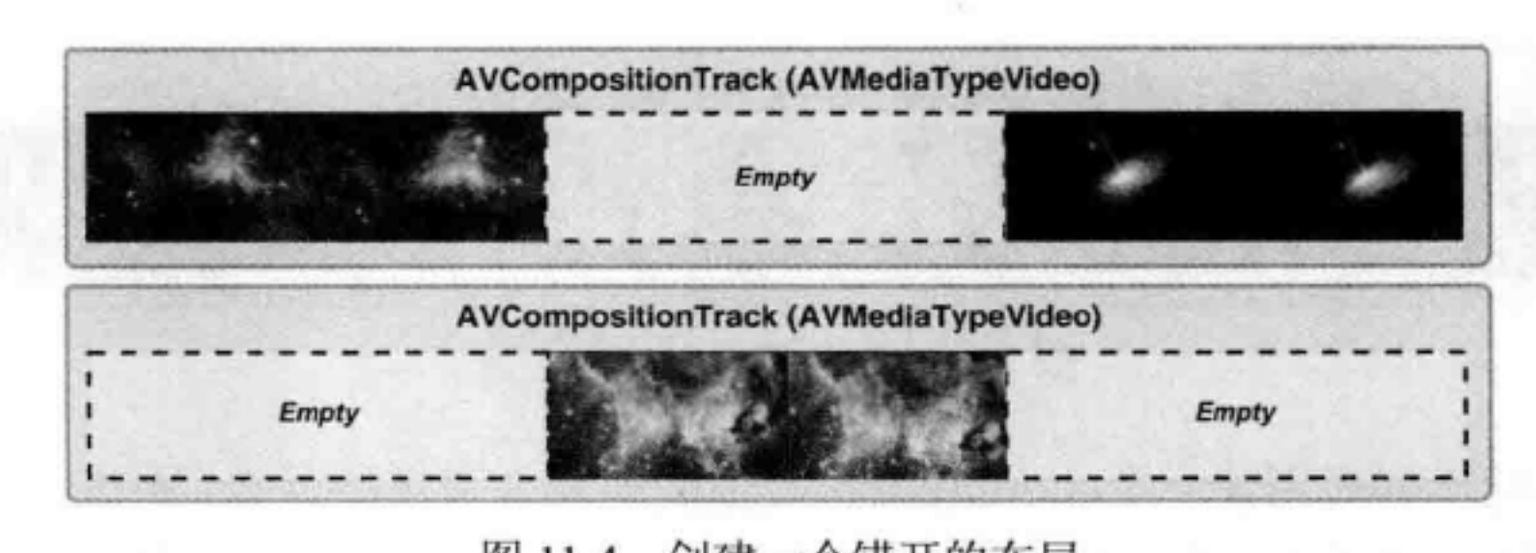

图 11-4 创建一个错开的布局

图11-4展示了一个三段视频的排列，不过在实际组合中的所有视频剪辑都可以遵循相同的A-B模式。当视频片段按照这种方式排列后，视频片段前面或后面的空间都会添加一个空片段。它们都是普通的AVCompositionTrackSegment实例，与视频是一样的，不过它们不包含任何媒体。这个过程的实现如以下代码所示。

```
NSArray *videoAssets = nil;// array of loaded AVAsset instances;

CMTime cursorTime = kCMTimeZero;

for (NSUInteger i = 0; i < videoAssets.count; i++) {

    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = videoTracks[trackIndex];

    AVAsset *asset = videoAssets[i];
    AVAssetTrack *assetTrack =
        [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

    CMTimeRange timeRange = CMTimeRangeMake(kCMTimeZero, asset.duration);

    [currentTrack insertTimeRange:timeRange
                        ofTrack:assetTrack
                         atTime:cursorTime error:nil];

    cursorTime = CMTimeAdd(cursorTime, timeRange.duration);
}
```

示例对所有添加到组合中的视频资源进行遍历。迭代过程首先通过i % 2来计算A-B模式下目标轨道的索引号。将视频轨道从当前资源中提取出来，并将它们插入currentTrack中。最后，更新cursorTime为下一次迭代更新插入点。

轨道和剪辑的布局已经完成了，不过还有一个问题需要解决。知道是什么吗？虽然我们已经以交错方式排列这些轨道，不过每个片段的开始都还紧接着上一片段的结尾。这就意味着我们在两个可见剪辑间没有任何空间，也就无法将它们组合在一起。下一步来解决这一问题。

11.2.2 定义重叠区域

要在两个片段中应用视频过渡，需要根据期望的过渡持续时长来确定片段的重叠情况。解决这一问题很简单，只需对上例中计算cursorTime的方法稍加调整。

```
NSArray *videoAssets = nil;// array of loaded AVAsset instances;

CMTime cursorTime = kCMTimeZero;
CMTime transitionDuration = CMTimeMake(2, 1);

for (NSUInteger i = 0; i < videoAssets.count; i++) {

    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = videoTracks[trackIndex];

    AVAsset *asset = videoAssets[i];
    AVAssetTrack *assetTrack =
        [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

    CMTimeRange timeRange = CMTimeRangeMake(kCMTimeZero, asset.duration);

    [currentTrack insertTimeRange:timeRange
                          ofTrack:assetTrack
                           atTime:cursorTime error:nil];

    cursorTime = CMTimeAdd(cursorTime, timeRange.duration);
    cursorTime = CMTimeSubtract(cursorTime, transitionDuration);
}
```

示例中为期望的过渡持续时长定义了一个CMTime值，这里设为2秒。在循环结束时从cursorTime减去transitionDuration，这样就能考虑到过渡时长适度偏移下一个插入点。图11-5给出了这一变化的结果。

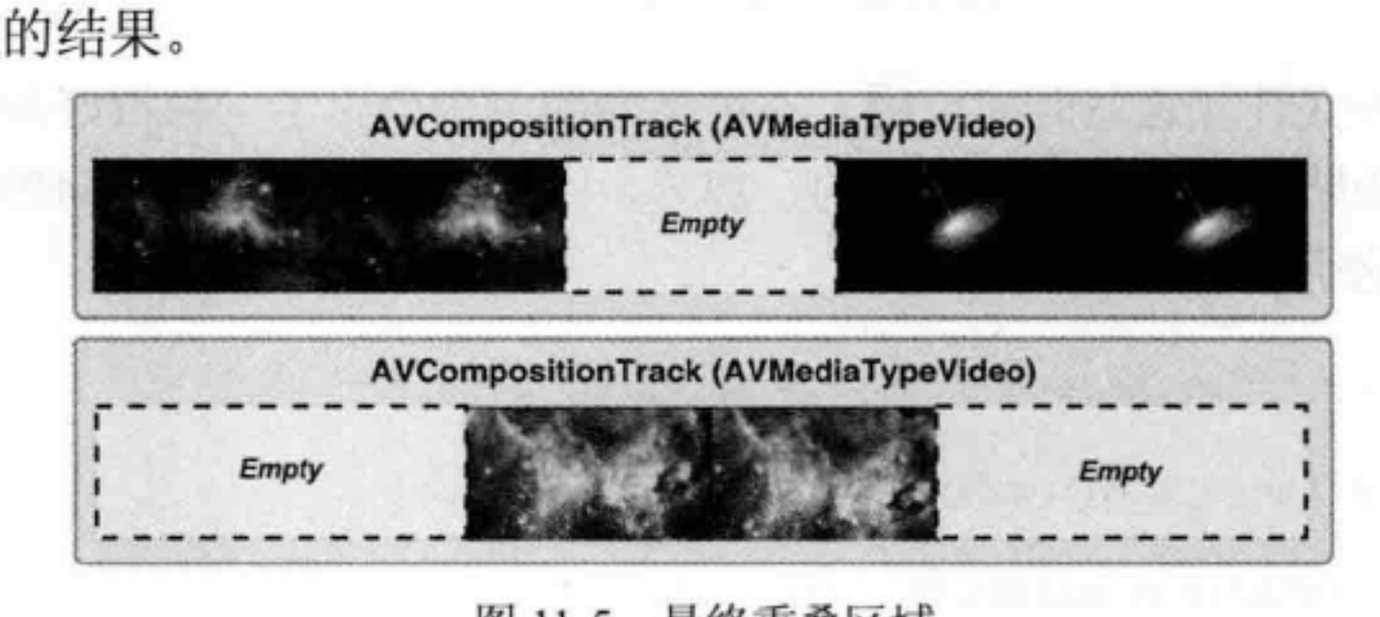

图 11-5　最终重叠区域

此时组合是可以播放的，不过输出结果可能并不理想。视频轨道具有内在的z索引行为，第一个视频轨道和第二个之前，第二个在第三个之前，诸如此类。如果在这种情况下播放组合，会发现第一段视频，第二段视频什么都没有，最后会看到最前面轨道中所包含的第三个片段。在我们看到第二个轨道的内容前，需要定义时间范围并向组合方法解释这两个轨道应该如何组合。下面继续下一个步骤并开始进行处理。

注意：

在创建这些重叠区域时要记住，我们已经将视频时间轴减少了(videoCount-1) * transitionDuration 的大小。如果组合中还有额外的轨道，则需要将它们的时长相应地缩短以满足新的视频时间轴。

11.2.3 计算通过和过渡的时间范围

AVVideoCompostion由一组AVVideoCompositionInstruction对象组成。其中最重要的数据是时间范围，它用来表示某种出现的组合方式持续的时长。在开始创建AVVideoComposition-Instruction实例前，首先需要为组合对象计算一系列时间范围。

需要计算两个类型的时间范围。第一个通常被认为是通过(pass-through)时间范围，在这个时间范围内希望一个轨道的所有帧序列都在不与其他轨道进行混合的情况下通过某一区域。第二个时间范围类型为过渡(transition)时间范围。它定义了在组合中视频片段重叠的区域，并在时间抽上标记出应用过渡效果的区域。图11-6给出了示例组合中的通过时间范围和过渡时间范围。

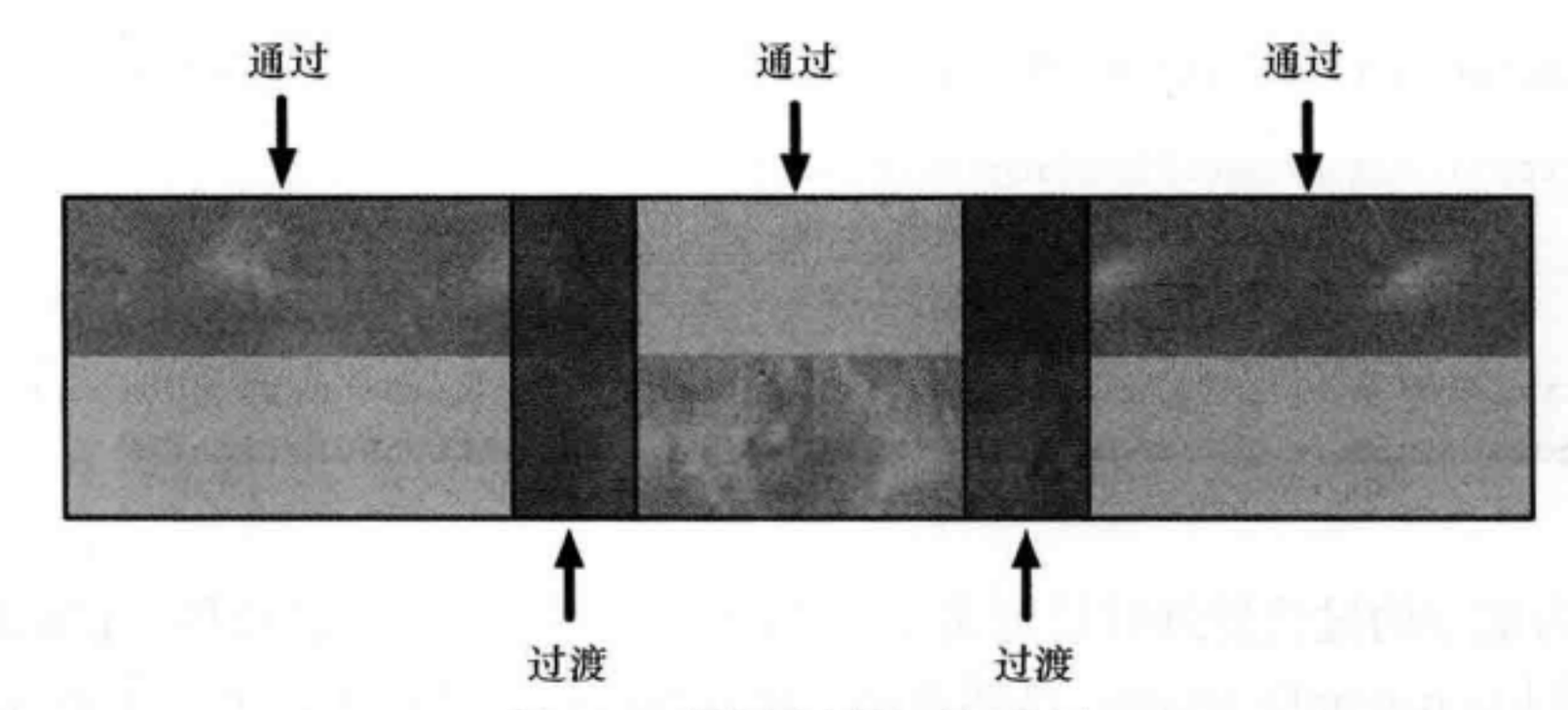

图 11-6 通过和过渡时间范围

可通过多种方式计算这些时间范围，不过下面的示例给出了一种很好的通用方法。这一计算过程通常作为轨道和片段创建过程的一部分，不过为了更好地说明问题，在这个示例中我们单独来实现这个功能。

```
NSMutableArray *videoAssets = // loaded video assets

CMTime cursorTime = kCMTimeZero;

// 2 second transition duration
CMTime transDuration = CMTimeMake(2, 1);

NSMutableArray *passThroughTimeRanges = [NSMutableArray array];
NSMutableArray *transitionTimeRanges = [NSMutableArray array];

NSUInteger videoCount = [videoAssets count];

for (NSUInteger i = 0; i < videoCount; i++) {

    AVAsset *asset = videoAssets[i];

    CMTimeRange timeRange = CMTimeRangeMake(cursorTime, asset.duration);

    if (i > 0) {
        timeRange.start = CMTimeAdd(timeRange.start, transDuration);
        timeRange.duration = CMTimeSubtract(timeRange.duration, transDuration);
    }

    if (i + 1 < videoCount) {
```

```
        timeRange.duration = CMTimeSubtract(timeRange.duration, transDuration);
    }

    [passThroughTimeRanges addObject:[NSValue valueWithCMTimeRange:timeRange]];

    cursorTime = CMTimeAdd(cursorTime, asset.duration);
    cursorTime = CMTimeSubtract(cursorTime, transDuration);

    if (i + 1 < videoCount) {
        timeRange = CMTimeRangeMake(cursorTime, transDuration);
        NSValue *timeRangeValue = [NSValue valueWithCMTimeRange:timeRange];
        [transitionTimeRanges addObject:timeRangeValue];
    }

}
```

示例中对组合的视频集进行了遍历。对于每个视频都创建了一个初始时间范围，之后根据其原始位置，对时间范围的起点和(或)持续时间进行修改。计算出cursorTime值后，基于cursorTime和transDuration创建相关的过渡时间范围。

注意：

CMTime 和 CMTimeRange 都是结构，所以它们不能直接添加到 NSArray 或 NSDictionary 中。取而代之的是，我们使用 AVTime.h 头文件中定义的 category 方法将它们封装成 NSValue 实例。

提示：

上面示例中的效果很难用文字进行描述。我们发现随着代码的步骤在纸上绘制一些矩形框可能会对这一过程的理解有帮助。

创建所需的通过和过渡时间范围被认为是整个处理过程中最重要的一步操作。从面上看，它并不是最难解决的问题，不过它非常容易出错。在进行这一步处理时一定要记住两个关键点：

- 我们计算的时间范围必须没有任何空隙或重叠。必须是紧接上一个片段之后按需排列的最新时间范围。
- 计算必须考虑组合对象持续时间。如果组合中还包含额外的轨道，就需要使它们遵循目前的视频时间轴，或者根据它们的持续时长扩展最终的时间范围。

如果没有注意到这两点，组合对象仍可播放，不过视频内容将不会被渲染，只会显示一个黑屏。这样调试起来会很麻烦，不过苹果公司发布过一个工具类，可以帮助开发者诊断这些问题。在Apple Developer Center中会找到一个项目AVCompositionDebugViewer(支持Mac和iOS版本)，可以帮助我们呈现组合。图11-7给出了这个工具的截屏。

这个类帮助我们呈现组合及其相关的AVVideoComposition和AVAudioMix对象，使开发者可以很容易地找到问题。

现在我们创建了所需的通过和过渡时间范围，进入下一步，创建组合指令吧。

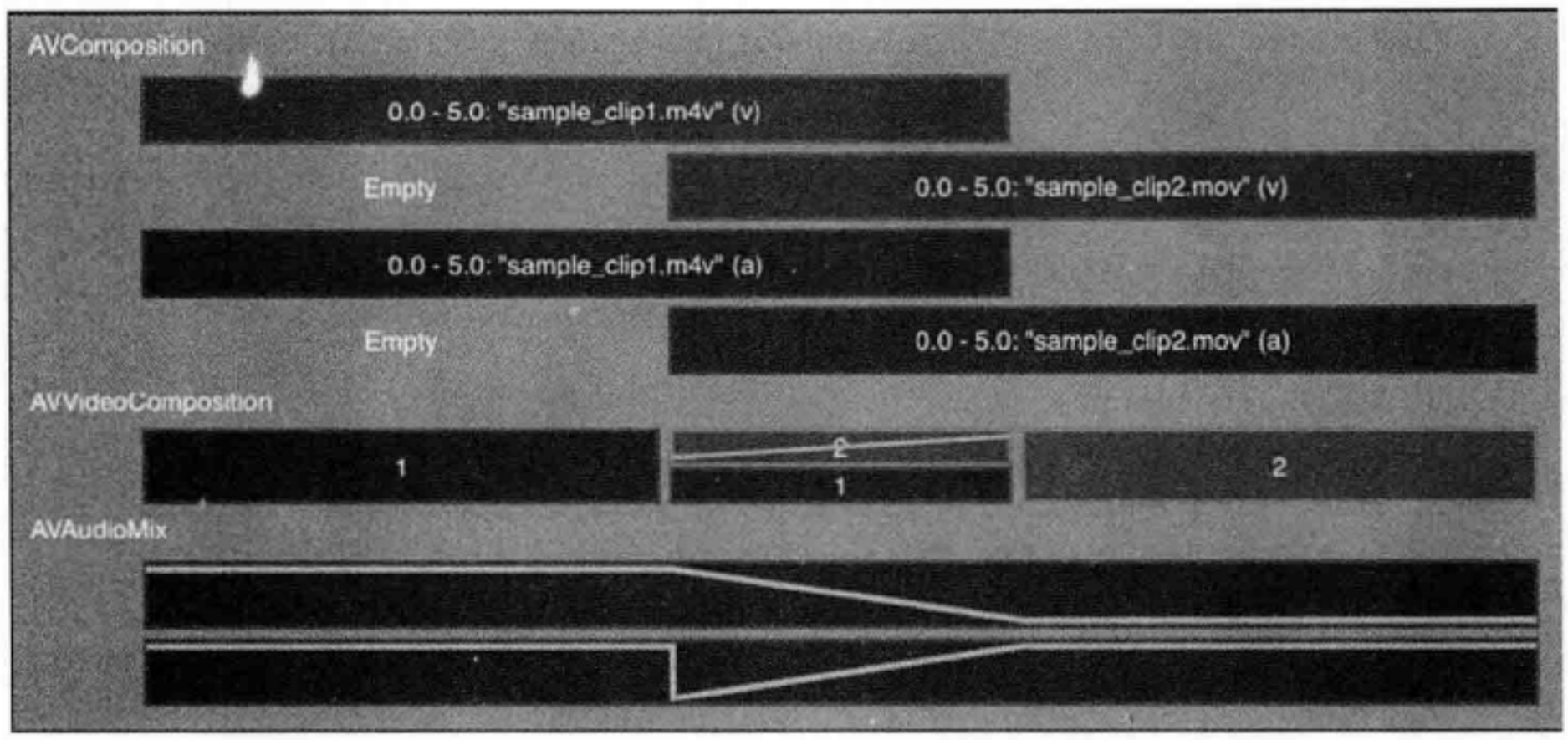

图 11-7 AVCompositionDebugViewer

11.2.4 创建组合和层指令

下一步是创建AVVideoCompositionInstruction和AVVideoCompositionLayerInstruction实例，提供视频组合方法所执行的指令。下面的代码有些复杂，所以我们给出了一些标注。

```
NSMutableArray *compositionInstructions = [NSMutableArray array];

// Look up all of the video tracks in the composition
NSArray *tracks = [composition tracksWithMediaType:AVMediaTypeVideo];

for (NSUInteger i = 0; i < passThroughTimeRanges.count; i++) {                    // 1

    // Calculate the trackIndex to operate upon: 0, 1, 0, 1, etc.
    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = tracks[trackIndex];

    AVMutableVideoCompositionInstruction *instruction =                           // 2
        [AVMutableVideoCompositionInstruction videoCompositionInstruction];

    instruction.timeRange = [passThroughTimeRanges[i] CMTimeRangeValue];

    AVMutableVideoCompositionLayerInstruction *layerInstruction =                 // 3
        [AVMutableVideoCompositionLayerInstruction
            videoCompositionLayerInstructionWithAssetTrack:currentTrack];

    instruction.layerInstructions = @[layerInstruction];

    [compositionInstructions addObject:instruction];

    if (i < transitionTimeRanges.count) {

        AVCompositionTrack *foregroundTrack = tracks[trackIndex];                 // 4
        AVCompositionTrack *backgroundTrack = tracks[1 - trackIndex];

        AVMutableVideoCompositionInstruction *instruction =                       // 5
            [AVMutableVideoCompositionInstruction videoCompositionInstruction];

        CMTimeRange timeRange = [transitionTimeRanges[i] CMTimeRangeValue];
```

```
instruction.timeRange = timeRange;

AVMutableVideoCompositionLayerInstruction *fromLayerInstruction =   // 6
    [AVMutableVideoCompositionLayerInstruction
        videoCompositionLayerInstructionWithAssetTrack:foregroundTrack];

AVMutableVideoCompositionLayerInstruction *toLayerInstruction =
    [AVMutableVideoCompositionLayerInstruction
        videoCompositionLayerInstructionWithAssetTrack:backgroundTrack];

instruction.layerInstructions = @[fromLayerInstruction,             // 7
                                  toLayerInstruction];

        [compositionInstructions addObject:instruction];
    }

}
```

(1) 首先遍历之前计算的所有通过时间范围。循环在两个需要创建所需指令的视频轨道间前后切换。

(2) 创建一个新的AVMutableVideoCompositionInstruction实例并设置当前通过CMTimeRange作为它的timeRange属性。

(3) 之后为活动组合创建一个新的AVMutableVideoCompositionLayerInstruction，将它添加到数组中并设置它作为组合指令的layerInstructions属性。组合的通过时间范围区域只需要一个与要呈现视频帧的轨道相关的单独层指令。

(4) 要创建过渡时间范围指令，需要得到前一个轨道(过渡前的轨道)的引用和后一个轨道(过渡后的轨道)的引用。按这种方式查找轨道可以确保轨道的引用始终顺序正确。

(5) 创建另一个AVMutableVideoCompositionInstruction实例，设置当前过渡时间范围为它的timeRange属性。

(6) 为每个轨道创建一个AVMutableVideoCompositionLayerInstruction实例。在这些层指令上定义从一个场景到另一个场景的过渡效果。本例中没有使用过渡效果。取而代之的是，我们将在本章示例应用程序中讨论这一问题。

(7) 将两个层指令都添加到NSArray中，并设置它们作为当前组合指令的layerInstructions属性。对这一数组中的元素进行排序非常重要，因为它定义了组合输出中视频图层的z轴顺序。

所有需要的组合和层指令都成功创建了，现在继续完成最后一步，创建和配置AVVideoComposition。

11.2.5　创建和配置 AVVideoComposition

最困难的工作已经完成了，剩下的就是创建和配置一个AVVideoComposition实例，如下面的代码所示：

```
AVMutableVideoComposition *videoComposition =
    [AVMutableVideoComposition videoComposition];
videoComposition.instructions = compositionInstructions;
videoComposition.renderSize = CGSizeMake(1280.0f, 720.0f);
videoComposition.frameDuration = CMTimeMake(1, 30);
```

```
videoComposition.renderScale = 1.0f;
```

示例中创建了一个AVMutableVideoComposition实例并设置了4个关键属性：

- instructions 属性用于设置我们在第 4 步中创建的组合指令。这些指令向组合器描述时间范围和执行组合的种类。
- renderSize 属性是一个 CGSize 值，定义组合应该被渲染的尺寸。这个值应该对应于组合中的视频原始大小，比如 720p 视频为 1280 x 720，1080p 视频为 1920 x1080。
- frameDuration 用于设置有效的帧率。回顾一下 AV Foundation 很少处理帧率，而是经常用到帧时长。帧时长是帧率的倒数，所以要设置 30FPS 的帧率，则定义的帧时长应该是 1/30 秒。
- renderScale 定义了视频组合应用的缩放。大部分情况下设置为 1.0。

如何这是第一次在AV Foundation中创建视频过渡，那么可以假设你已经觉得有些头疼或已经晕了。我们知道开发者的痛苦，因为学习视频过渡的确是这个框架有关视频编辑功能中最难的一部分，不过我们还是有好消息告诉你的，继续往下看。

11.2.6 创建视频组合的捷径

在很多情况下有一种比上一节中介绍的方法更简便的创建AVVideoComposition的方法。AVVideoComposition定义了一个便捷初始化方法videoCompositionWithPropertiesOfAsset:，将AVComposition作为资源参数并创建了一个基础AVVideoComposition。这个方法创建了一个带有如下配置的AVVideoComposition对象：

- instructions 属性包含一组完整的基于组合视频轨道(以及其中包含的片段空间布局)的组合和层指令。
- renderSize 属性被设置为 AVComposition 对象的 naturalSize，或者如果没有设置，则使用能够满足组合视频轨道中最大视频维度的尺寸值。
- frameDuration 设置为组合视频轨道中最大 nominalFrameRate 的值。如果所有轨道的 nominalFrameRate 值都为 0，则 frameDuration 设置成默认值 1/30 秒(30FPS)。
- renderScale 始终设置为 1.0。

你可能会问为什么没有一开始就介绍这种方法，而跳过哪些复杂的实现过程。实际上主要有以下两个重要原因：

- 只有理解了前面讨论的步骤，才能发现学习这些对象的使用变容易了，因为我们仍需要对过渡效果配置一些指令。如果不理解如何构建组合及其原因，则很难得到所需的结果，而且几乎不可能在应用程序出现问题时进行调试。
- 创建的可能并不是我们所需要的。以这种方式创建 AVVideoComposition 可能适用于很多场景，不过这并不能解决全部问题。很多时候当需要对组合的时间范围和指令进行更多控制时，就需要从头开始手动创建组合对象。

总的来说，越了解这一过程，离精通AV Foundation的过渡功能更近了一步。

11.3　为 15 Seconds 应用程序添加视频过渡

现在我们对创建视频过渡的概念已经比较了解了，下面讨论如何将这个功能放到15 Seconds应用程序中。在Chapter 10目录中可以找到15 Seconds应用程序项目的启动版本，下面从上一章剩下的工作开始继续完善它。

首先创建一个新的类THTransitionComposition，并令它遵循THComposition协议，如代码清单11-1所示。

代码清单 11-1　THTransitionComposition 接口

```
#import "THComposition.h"

@interface THTransitionComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVComposition *composition;
@property (strong, nonatomic, readonly) AVVideoComposition *videoComposition;
@property (strong, nonatomic, readonly) AVAudioMix *audioMix;

- (id)initWithComposition:(AVComposition *)composition
        videoComposition:(AVVideoComposition *)videoComposition
                audioMix:(AVAudioMix *)audioMix;
@end
```

这个类看起来和之前几章中创建的类类似；唯一的区别是增加了一个新的AVVideoComposition属性和在初始化方法中增加了一个videoComposition参数。下面看一下该类的具体实现，如代码清单11-2所示。

代码清单 11-2　THTransition Composition 实现

```
#import "THTransitionComposition.h"

@implementation THTransitionComposition

- (id)initWithComposition:(AVComposition *)composition
        videoComposition:(AVVideoComposition *)videoComposition
                audioMix:(AVAudioMix *)audioMix {
    self = [super init];
    if (self) {
        _composition = composition;
        _videoComposition = videoComposition;
        _audioMix = audioMix;
    }
    return self;
}

- (AVPlayerItem *)makePlayable {                                        // 1
    AVPlayerItem *playerItem =
        [AVPlayerItem playerItemWithAsset:[self.composition copy]];

    playerItem.audioMix = self.audioMix;
    playerItem.videoComposition = self.videoComposition;

    return playerItem;
```

```
}

- (AVAssetExportSession *)makeExportable {                          // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
        [AVAssetExportSession exportSessionWithAsset:[self.composition copy]
                                          presetName:preset];
    session.audioMix = self.audioMix;
    session.videoComposition = self.videoComposition;

    return session;
}

@end
```

(1) 首先在makePlayable方法中创建一个带有AVComposition实例的新的AVPlayerItem。像上一章中所做的那样设置播放器条目的audioMix和videoComposition属性。这样在播放时可以对播放器应用自定义音频处理和视频组合行为。

(2) 在makeExportable方法中创建一个新的AVAssetExportSession，传递给它一个composition副本和一个预设名。与makePlayable方法类似，对导出会话的相应属性设置audioMix和videoComposition，这样在导出时就可以应用音频和视频处理了。

下一步为对象创建相关的THCompositionBuilder。在示例项目的Models/Builder/Private组中会发现一个名为THTransitionCompositionBuilder的存根类。代码清单11-3给出了这个类的接口。

代码清单 11-3 THTransitionCompositionBuilder 接口

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THTransitionCompositionBuilder : NSObject <THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;

@end
```

不看这个类的名字的话，这个接口内容和我们之前几章创建的其他类接口很相似，继续看这个类的具体实现，如代码清单11-4所示。

代码清单 11-4 THTransitionCompositionBuilder 实现

```
#import "THTransitionCompositionBuilder.h"
#import "THVideoItem.h"
#import "THAudioItem.h"
#import "THVolumeAutomation.h"
#import "THTransitionComposition.h"
#import "THTransitionInstructions.h"
#import "THFunctions.h"

@interface THTransitionCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@property (weak, nonatomic) AVMutableCompositionTrack *musicTrack;
```

```
@end

@implementation THTransitionCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition composition];

    [self buildCompositionTracks];

    AVVideoComposition *videoComposition = [self buildVideoComposition];

    AVAudioMix *audioMix = [self buildAudioMix];

    return [[THTransitionComposition alloc] initWithComposition:self.composition
                                                videoComposition:videoComposition
                                                      audioMix:audioMix];
}

- (void)buildCompositionTracks {

    // To be implemented

}

- (AVVideoComposition *)buildVideoComposition {

    // To be implemented

    return nil;
}

@end
```

buildComposition方法的实现看起来与之前章节创建的实现类似，不过创建轨道的代码已经提取到专门的创建方法中，并添加了一个新方法buildVideoComposition，稍后会实现它们。buildAudioMix方法的实现被略去了，因为它和上一章的内容完全一致。

现在实现buildCompositionTracks方法来提供视频过渡所需的轨道和剪辑布局，如代码清单11-5所示。

代码清单 11-5　实现 buildCompositionTracks 方法

```
- (void)buildCompositionTracks {

    CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;

    AVMutableCompositionTrack *compositionTrackA =                          // 1
        [self.composition addMutableTrackWithMediaType:AVMediaTypeVideo
```

```
                                preferredTrackID:trackID];

AVMutableCompositionTrack *compositionTrackB =
    [self.composition addMutableTrackWithMediaType:AVMediaTypeVideo
                                preferredTrackID:trackID];

NSArray *videoTracks = @[compositionTrackA, compositionTrackB];

CMTime cursorTime = kCMTimeZero;
CMTime transitionDuration = kCMTimeZero;

if (!THIsEmpty(self.timeline.transitions)) {                          // 2
    // 1 second transition duration
    transitionDuration = THDefaultTransitionDuration;
}

NSArray *videos = self.timeline.videos;

for (NSUInteger i = 0; i < videos.count; i++) {

    NSUInteger trackIndex = i % 2;                                    // 3

    THVideoItem *item = videos[i];
    AVMutableCompositionTrack *currentTrack = videoTracks[trackIndex];

    AVAssetTrack *assetTrack =
        [[item.asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

    [currentTrack insertTimeRange:item.timeRange
                         ofTrack:assetTrack
                          atTime:cursorTime error:nil];

    // Overlap clips by transition duration                           // 4
    cursorTime = CMTimeAdd(cursorTime, item.timeRange.duration);
    cursorTime = CMTimeSubtract(cursorTime, transitionDuration);
}

// Add voice overs                                                    // 5
[self addCompositionTrackOfType:AVMediaTypeAudio
                 withMediaItems:self.timeline.voiceOvers];

// Add music track
NSArray *musicItems = self.timeline.musicItems;
self.musicTrack = [self addCompositionTrackOfType:AVMediaTypeAudio
                                   withMediaItems:musicItems];
}
```

(1) 首先创建两个新的AVMutableCompositionTrack对象，二者都是AVMediaTypeVideo类型，提供所需的A-B轨道排列。将它们保存到videoTracks数组，当需要向组合中插入视频时会遍历该集合并取出它们。

(2) 判断THTimeline对象的transitions数组是否被填充。如果视频过渡在用户界面中被设置为激活，则填充描述过渡效果的THVideoTransition对象集合。如果过渡被激活，设置transitionDuration为THDefaultTransitionDuration，这是一个全局常量，是1秒的CMTime值。

(3) 遍历准备添加到组合中的视频集合，首先确定剪辑应该插入的目标轨道。使用i % 2

的算法计算trackIndex值，得到的0,1,0,1模式确保视频片段的排列为相应的A-B模式。

(4) 与之前一样计算cursorTime，不过这里使用CMTimeSubtract函数重新定义它的值，回退插入点时将transitionDuration考虑在内。这样轨道的布局就提供了所需的过渡区域。

(5) 调用前一章中编写的addCompositionTrackOfType: withMediaItems:方法来添加画外音和音乐轨道。出于简洁性的考虑，本章没有给出这个方法的具体实现。

建议现在运行应用程序，这样就可以看到这个新组合排列的结果。compositionTrackA中包含的视频将会照常展现，不过compositionTrackB中包含的视频则不可见。这是因为我们之前提到的当使用多视频轨道时会出现的z-索引行为。要修复这个行为，需要创建一个AVVideoComposition，组合呈现或导出时可以对使用的组合进行控制。首先创建buildVideoComposition方法，如代码清单11-6所示。

代码清单 11-6　实现 buildVideoComposition 方法

```
- (AVVideoComposition *)buildVideoComposition {

    AVVideoComposition *videoComposition =                                  // 1
        [AVMutableVideoComposition
            videoCompositionWithPropertiesOfAsset:self.composition];

    NSArray *transitionInstructions =                                       // 2
        [self transitionInstructionsInVideoComposition:videoComposition];

    for (THTransitionInstructions *instructions in transitionInstructions) {

        CMTimeRange timeRange =                                             // 3
            instructions.compositionInstruction.timeRange;

        AVMutableVideoCompositionLayerInstruction *fromLayer =
            instructions.fromLayerInstruction;

        AVMutableVideoCompositionLayerInstruction *toLayer =
            instructions.toLayerInstruction;

        THVideoTransitionType type = instructions.transition.type;

        // Apply Video Transition Effects                                   // 4
        if (type == THVideoTransitionTypeDissolve) {
            // Listing 11.7
        }

        else if (type == THVideoTransitionTypePush) {
            // Listing 11.8
        }

        else (type == THVideoTransitionTypeWipe) {
            // Listing 11.9
        }

        instructions.compositionInstruction.layerInstructions = @[fromLayer,   // 5
                                                                  toLayer];
    }

    return videoComposition;
```

```
}

- (NSArray *)transitionInstructionsInVideoComposition:(AVVideoComposition *)vc {

    // To be implemented

    return nil;
}
```

(1) 首先使用videoCompositionWithPropertiesOfAsset方法创建一个新的AVVideoComposition实例。这个方法自动创建所需的组合对象和层指令，并设置renderSize、renderScale和frameDuration属性为相应的值。

(2) 调用私有方法transitionInstructionsInVideoComposition:从AVVideoComposition中提取相关的层指令，这样就可以应用期望的视频过渡效果。这个方法返回一个THTransitionInstructions对象数组，这些对象保存着基础数据，用于简化组合和层指令的相关工作。稍后会给出这个方法的实现并创建这些对象。

(3) 为从THTransitionInstructions实例中得到的数据定义本地变量。在应用过渡效果时会用到这些变量。

(4) 定义一个占位符条件语句来判断所执行的视频转换。应用程序的用户界面允许在剪辑之间使用溶解、推入或擦除过渡效果。代码清单11-7到代码清单11-9给出了这些效果的创建过程。

(5) 配置组合指令的layerInstructions，按显示的顺序传递指令。在确保正确应用视频过渡时，这个顺序非常必要。

下一步需要具体实现transitionInstructionsInVideoComposition:方法，如代码清单11-7所示。这个方法负责将组合和层指令从之前创建的AVVideoComposition中提取出来，并将它们与用户在时间轴上配置的THVideoTransition相关联。THVideoTransition定义了需要应用的过渡类型。

代码清单 11-7 提取过渡指令

```
- (NSArray *)transitionInstructionsInVideoComposition:(AVVideoComposition *)vc {

    NSMutableArray *transitionInstructions = [NSMutableArray array];

    int layerInstructionIndex = 1;

    NSArray *compositionInstructions = vc.instructions;                    // 1

    for (AVMutableVideoCompositionInstruction *vci in compositionInstructions) {

        if (vci.layerInstructions.count == 2) {                            // 2

            THTransitionInstructions *instructions =
                [[THTransitionInstructions alloc] init];

            instructions.compositionInstruction = vci;

            instructions.fromLayerInstruction =                            // 3
                vci.layerInstructions[1 - layerInstructionIndex];

            instructions.toLayerInstruction =
                vci.layerInstructions[layerInstructionIndex];

            [transitionInstructions addObject:instructions];

            layerInstructionIndex = layerInstructionIndex == 1 ? 0 : 1;
```

```
    }
}

NSArray *transitions = self.timeline.transitions;

// Transitions are disabled
if (THIsEmpty(transitions)) {                                          // 4
    return transitionInstructions;
}

NSAssert(transitionInstructions.count == transitions.count,
         @"Instruction count and transition count do not match.");

for (NSUInteger i = 0; i < transitionInstructions.count; i++) {        // 5
    THTransitionInstructions *tis = transitionInstructions[i];
    tis.transition = self.timeline.transitions[i];
}

return transitionInstructions;
}
```

(1) 首先遍历从AVVideoComposition中得到的AVVideoCompositionInstruction对象。

(2) 我们只关心包含两个层指令的组合指令，这表示这个指令定义了组合中的过渡区域。

(3) 自动创建的层指令通常保存于layerInstructions数组，第一个轨道的层指令作为数组的第一个元素，接下来是第二个轨道的层指令。代码进行了一些计算来确保过渡之前的层和过渡之后的层保持不变。很容易理解在创建过渡时需要对每个层都应用选定的效果。

(4) 如果transitions数组是空的，则返回transitionInstructions，因为这表示用户界面禁用了过渡。

(5) 如果过渡已启用，则遍历transitionInstructions，并将用户选定的THVideoTransition对象和它进行关联。THVideoTransition对象定义了所应用的过渡类型，比如推入、溶解或擦除等。

应用过渡效果

应用程序支持3种过渡类型：溶解(默认)、推入和擦除。下面看下如何实现每种效果，先从溶解过渡开始。

1. 溶解过渡效果

第一个需要实现的过渡效果是溶解过渡。这个过渡效果需要修改输入层的模糊度来获得两个层逐渐融合的效果，如图11-8所示。

图 11-8　溶解过渡

代码清单11-8是这个方法具体的实现。

代码清单 11-8 应用溶解过渡效果

```
if (type == THVideoTransitionTypeDissolve) {

    [fromLayer setOpacityRampFromStartOpacity:1.0
                                  toEndOpacity:0.0
                                     timeRange:timeRange];
}
```

实现一个简单溶解效果很容易。只需对fromLayer对象设置一个模糊渐变，在过渡时间内将模糊值默认的1.0(完全模糊)调整到0.0(完全透明)。这样就将当前视频以溶解的效果切换到下一个视频。另一个溶解过渡的形式是交叉溶解，对toLayer设置一个模糊渐变，将模糊值从0.0变为1.0。简单模糊效果更加细腻且通常效果很好，不过我们鼓励开发者尝试两种方式的溶解来选择自己喜欢的效果。

2. 推入过渡效果

下一个要实现的过渡是推入过渡，如图11-9所示。这是一个有方向的过渡，即下一段视频将当前视频推出视图区域。

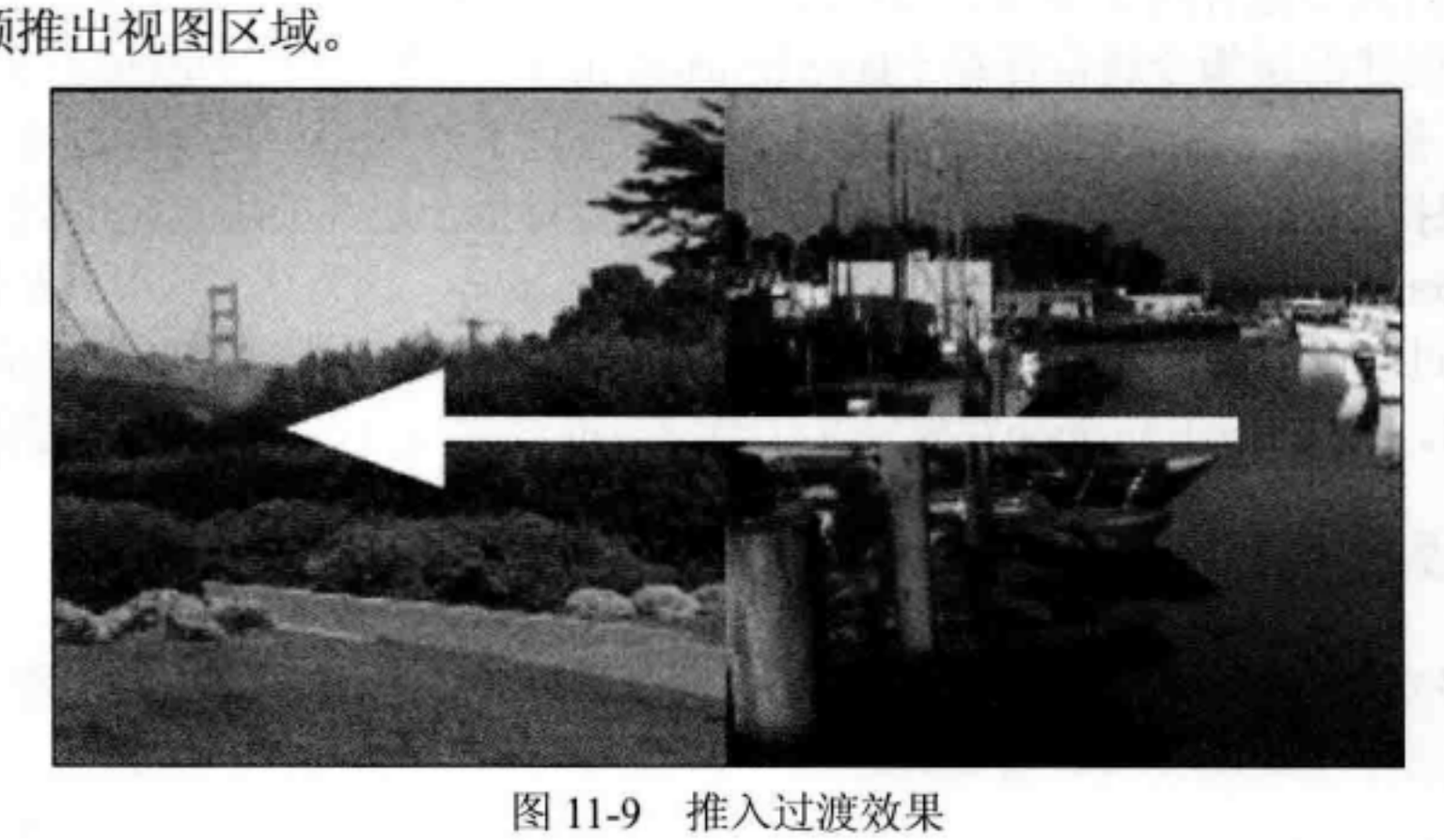

图 11-9 推入过渡效果

代码清单11-9给出了向左推动效果的代码。

代码清单 11-9 应用推入过渡效果

```
else if (type == THVideoTransitionTypePush) {

    // Define starting and ending transforms                          // 1
    CGAffineTransform identityTransform = CGAffineTransformIdentity;

    CGFloat videoWidth = videoComposition.renderSize.width;

    CGAffineTransform fromDestTransform =
        CGAffineTransformMakeTranslation(-videoWidth, 0.0);

    CGAffineTransform toStartTransform =
        CGAffineTransformMakeTranslation(videoWidth, 0.0);
```

```
    [fromLayer setTransformRampFromStartTransform:identityTransform     // 2
                                toEndTransform:fromDestTransform
                                    timeRange:timeRange];

    [toLayer setTransformRampFromStartTransform:toStartTransform        // 3
                                toEndTransform:identityTransform
                                    timeRange:timeRange];
}
```

(1) 首先定义一个对于输入视频层的变换。CGAffineTransform可以修改层的转化、旋转和缩放。对层应用一个渐变的变化可以衍生出许多有趣的效果。这里我们使用转化(translation)转换，修改图层的x、y坐标。fromDestTransform将fromLayer移到左侧，使用视频宽度的负值，使对象完全移出视图。定义一个toStartTransform，将toLayer的起始位置向右移出视图。

(2) 设置fromLayer的渐变效果，初始变换设置为identityTransform(未变换的状态)，终点变换设置为fromDestTransform。这样就会在timeRange时间内实现fromLayer向左侧退出的动画效果。

(3) 采用类似的方式在toLayer上设置一个渐变效果，初始变换设置为toStartTransform，终点变换设置为identityTransform。这样就会在timeRange时间内实现toLayer从右侧进入的动画效果。

3. 擦除过渡效果

最后一个要实现的过渡是擦除效果(如图11-10所示)。擦除过渡有多种形式，不过通常都以一种动画方式将当前视频移出，显示下一个视频。

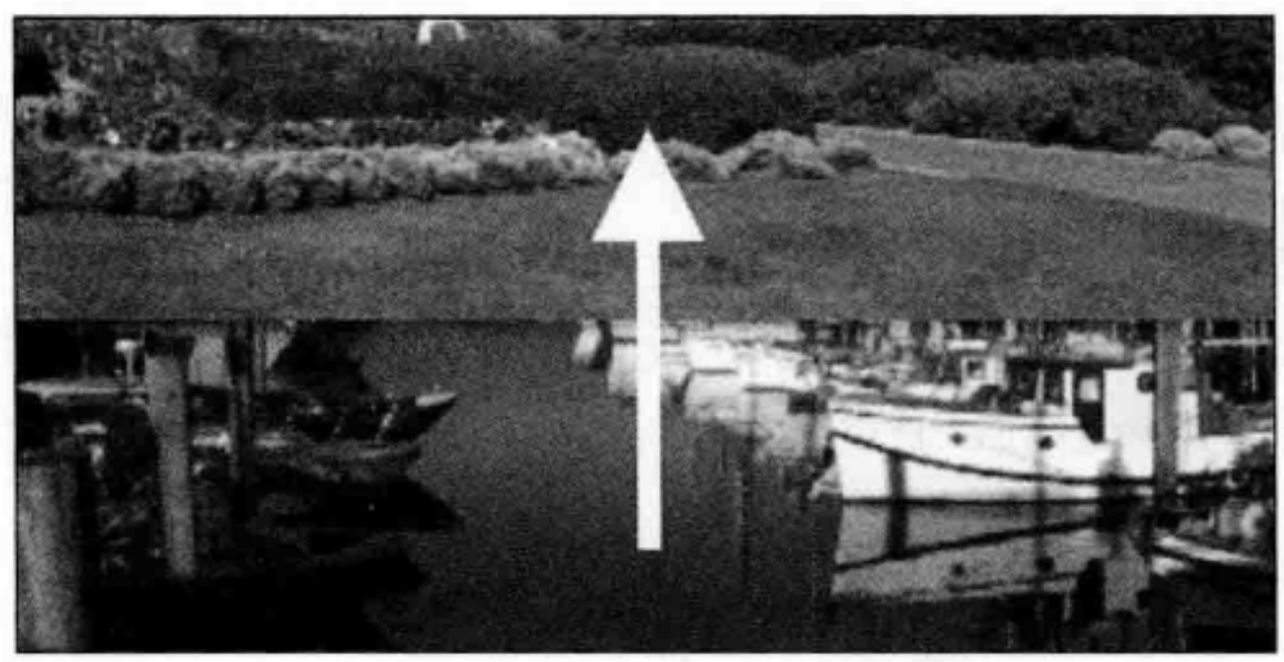

图 11-10　向上擦除的过渡效果

代码清单11-10给出了实现简单向上擦除的实现代码，fromLayer向上划过，显示toLayer层。

代码清单 11-10　应用擦除过渡效果

```
else (type == THVideoTransitionTypeWipe) {

    CGFloat videoWidth = videoComposition.renderSize.width;
    CGFloat videoHeight = videoComposition.renderSize.height;

    CGRect startRect = CGRectMake(0.0f, 0.0f, videoWidth, videoHeight);
    CGRect endRect = CGRectMake(0.0f, videoHeight, videoWidth, 0.0f);

    [fromLayer setCropRectangleRampFromStartCropRectangle:startRect
```

```
                                        toEndCropRectangle:endRect
                                                timeRange:timeRange];
}
```

要实现这个效果，需要从视频组合的renderSize中获取宽和高。这些值用于创建擦除动画效果的开始和结束CGRect值。初始矩形为最大的宽度和高度，最终矩形在高度上有所削减，在fromLayer上生成一个向上擦除的效果。

运行应用程序并点击Settings菜单，可以看到一个新的Video区域出现在菜单中，并带有视频过渡效果的切换开关。激活这个开关可以看到视频轨道更新为一个可见的过渡按钮，如图11-11所示。

图 11-11 用户界面过渡控件

点击这些按钮，尝试这些过渡类型并观察它们的实际效果。

11.4 小结

本章学习了如何使用AVVideoComposition、AVVideoCompositionInstruction和AVVideoCompositionLayerInstruction对象创建视频过渡。深入了解了如何创建一个视频组合并学习了一些规避常见陷阱的方法。学习创建视频过度的过程具有一定的挑战性，不过现在你已经克服重重困难，对创建过渡效果的过程有了比较深入的了解，这样会帮助开发者更好地使用框架的各种特性来优化视频编辑应用程序。

11.5 挑战

你在15 Seconds应用程序中加入了3个常用的过渡效果，不过这还远远不够。可以尝试改变推入或擦除过渡的方向，在视频之间实现一个交叉溶解的过渡效果，甚至可以同时应用多种渐变效果。多多尝试各种特性是学习如何使用它们的最佳方法。

第12章

动画图层内容

15 Seconds应用程序现在看起来很不错，已经可以创建具有非常专业视觉效果的视频组合了，不过在结束本书的学习前还有最后一个问题需要讨论。应用程序的轨道视图已经包含了视频、画外音和音轨，不过到目前为止我们创建的这个组合中还有一个神秘的轨道处于空置状态。本章就是要需要揭开这个神秘的面纱，学习有关组合之上的动画图层内容的相关知识。这一功能让开发者可以为应用程序添加一些叠加效果，比如水印、标题、下沿字幕或其他所需的动画效果。

AV Foundation对于在视频中整合动画叠加层提供了很好的支持。不过，第一次接触这方面的开发可能会感到一些陌生。目前为止我们完成的所有工作都是基于轨道模式的，所以找到一个专门的轨道类型作为整合叠加类型的轨道看起来顺理成章，不过实际上并非如此。取而代之的是，AV Foundation依靠Core Animation框架来创建这些效果并提供一些集成类来实现将效果整合到视频内容中的功能。这就带来了一种比之前几章中介绍的模式更新的思维模式，不过通过本章的讲解会消除这一障碍，通过AV Foundation和Core Animation提供的标准化方式可以很好地实现这些功能。下面先从了解Core Animation的知识开始学习本章的内容。

12.1 Core Animation 应用

Core Animation是OS X和iOS平台提供的用于合成和制作动画的框架，苹果平台上面漂亮、流畅的动画效果都是通过这个框架实现的。它提供一种简单、声明式应用程序模型使得在不需要使用OpenGL或OpenGL ES框架的前提下就可以很容易地创建高性能、基于GPU的动画效果。我们已经在书中见过AV Foundation是如何使用Core Animation框架提供硬件加速视频渲染效果的，其中就用到了AVPlayerLayer和AVVideoCapturePreviewLayer两个类。稍后讨论如何使用这些类为视频播放和导出操作添加动画效果。

从高层次视角看，Core Animation包含两类对象：

- **Layers**：图层对象由 CALayer 类定义，并用于管理屏幕中可视化内容的元素。这里所说的内容一般都是图片或 Bezier 路径，不过图层本身具有可被设置的可视化特征，比如它的背景颜色、透明度和角半径。图层定义了自身的几何属性，比如 bounds 和 position，

并且可以将这些元素组合到图层的层次结构中用于创建更复杂的界面。除了基础CALayer类，框架还定义了很多实用的子类，比如用于渲染贴图内容的CATextLayer类和用于渲染Bezier路径的CAShapeLayer类，这两个类在创建动画叠加效果时都非常重要。

- **Animations：**动画对象是抽象类CAAnimation的实例，定义所有动画类型所共有的一些核心动画行为。该框架定义了CAAnimation的许多具体子类，最常用的就是CABasicAnimation和CAKeyFrameAnimation。这些类将动画状态变为单独的图层属性，以便创建简单和复杂的动画效果。CABasicAnimation可以让你创建简单的单关键帧动画，意味着在一段时间内将属性状态以动画方式由一种状态变为另一种状态。这个类在实现简单动画时非常实用，比如动态调整图层的尺寸、位置或背景色。CAKeyFrameAnimation用于实现更高级的功能，它对动画中的“关键帧”有着更多的控制。比如，当一个图层沿着Bezier路径动态显示，可以用到关键帧动画来指定具体的时间和节奏。

下面看一个有关CALayer和CABasicAnimation的简单示例。将本书的封面图片作为一个图层，放置于父视图的中间，使它绕z轴动态显示，如图12-1所示。

图 12-1 Core Animation 示例

```
CALayer *parentLayer = // parent layer

UIImage *image = [UIImage imageNamed:@"lavf_cover"];

CALayer *imageLayer = [CALayer layer];
// Set the layer contents to the book cover image
imageLayer.contents = (id)image.CGImage;
imageLayer.contentsScale = [UIScreen mainScreen].scale;

// Size and position the layer
CGFloat midX = CGRectGetMidX(parentLayer.bounds);
CGFloat midY = CGRectGetMidY(parentLayer.bounds);

imageLayer.bounds = CGRectMake(0, 0, image.size.width, image.size.height);
imageLayer.position = CGPointMake(midX, midY);

// Add the image layer as a sublayer of the parent layer
[parentLayer addSublayer:imageLayer];

// Basic animation to rotate around z-axis
```

```
CABasicAnimation *rotationAnimation =
    [CABasicAnimation animationWithKeyPath:@"transform.rotation.z"];

// Rotate 360 degrees over a three-second duration, repeat indefinitely
rotationAnimation.toValue = @(2 * M_PI);
rotationAnimation.duration = 3.0f;
rotationAnimation.repeatCount = HUGE_VALF;

// Add and execute animation on the image layer
[imageLayer addAnimation:rotationAnimation forKey:@"rotateAnimation"];
```

Core Animation的效果通过静态代码的展示和描述很难看得出来；在Chapter 12目录中可以找到一个名为CoreAnimationExample的项目，可以实际运行这个应用程序查看效果。

有关Core Animation框架更深入的内容不在本书的讨论范围内，不过建议大家阅读Nick Lockwood撰写的*iOS Core Animation* (2014, Boston: Addison-Wesley)一书来了解如何有效使用Core Animation框架。

12.2　在 AV Foundation 中使用 Core Animation

使用Core Animation为视频应用程序创建叠加效果的方法同使用它在iOS或OS X平台创建实时动画效果的方法几乎一样。最大的区别在于运行动画的时间模型，当创建实时动画时，CAAnimation实例从系统主机时钟获取执行时间，如图12-2所示。

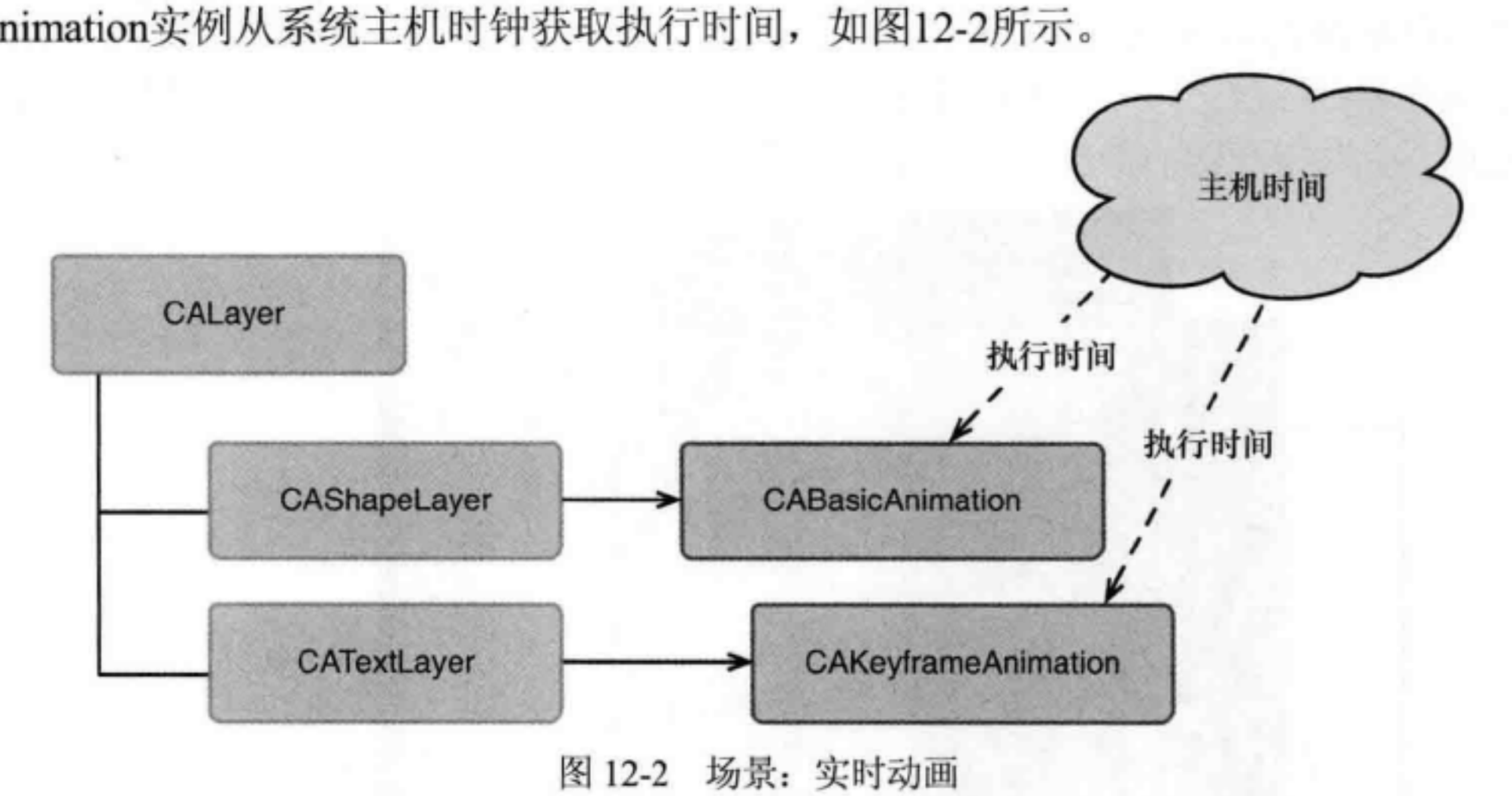

图 12-2　场景：实时动画

主机时间从系统启动开始计算并单向向前推进，将动画执行时间同主机时间相关联在实时动画方面非常适用，不过对于创建视频动画就不合适。视频动画需要基于“影片时间”来操作，开始的时间应该是影片开始的时间直到影片持续时间结束。另外主机时间是一直向前推进，从不会停止，而影片时间可以停止、暂停、回退或快进。因为动画需要紧密地与视频时间轴绑定，所以需要使用不同的执行时间模式。AV Foundation根据不同的用例给出了两种解决方案，下面先看一下播放场景。

12.2.1 使用 AVSynchronizedLayer 播放

AV Foundation提供了一个专门的CALayer子类AVSynchronizedLayer，用于与给定的AVPlayerItem实例同步时间。这个图层本身不展示任何内容，仅用来与图层子树协同时间。这样所有在继承关系中附属于该图层的动画都可以从激活的AVPlayerItem实例中获取相应的执行时间，如图12-3所示。

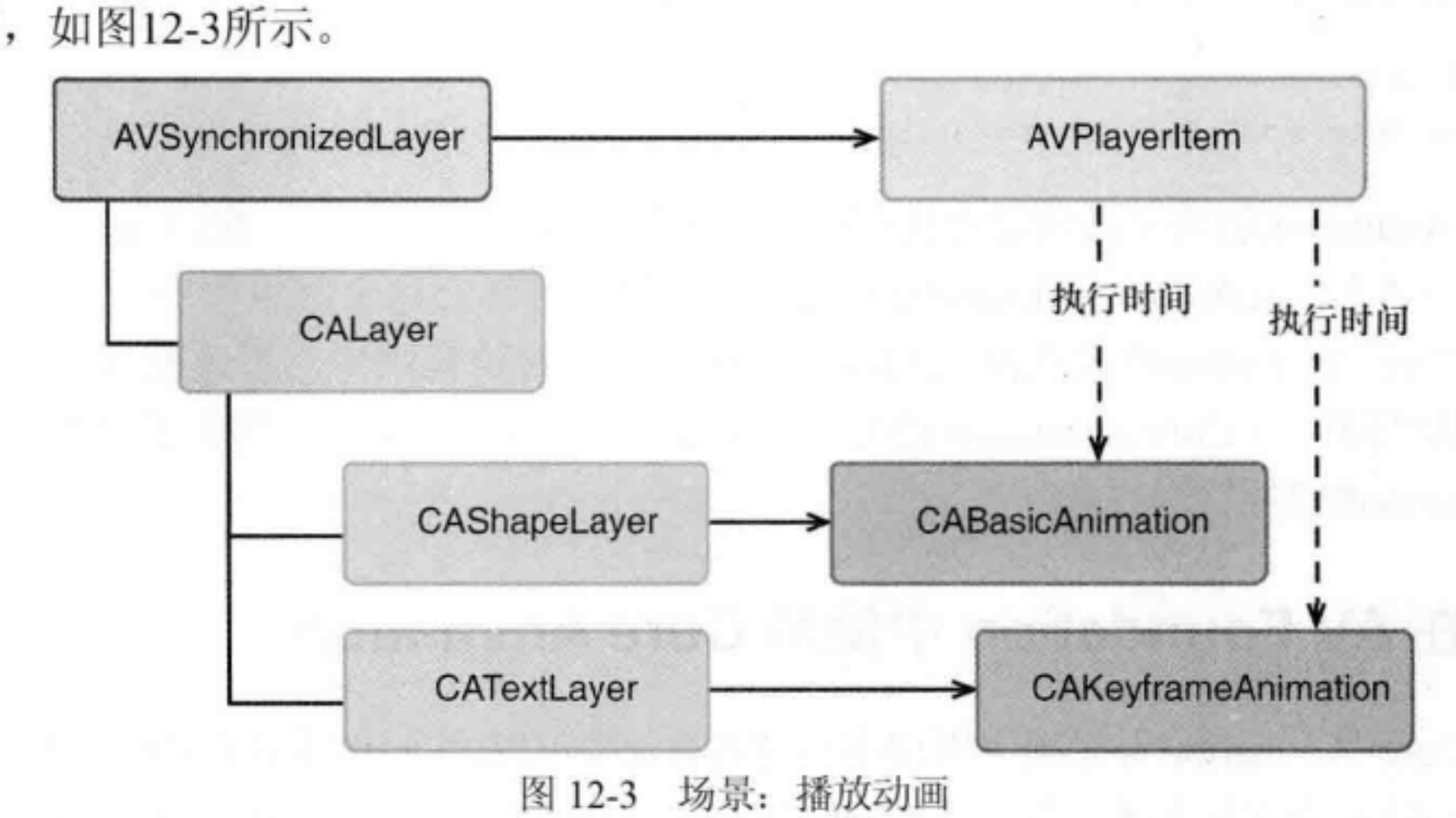

图 12-3 场景：播放动画

通常使用AVSynchronizedLayer时会将其整合到播放器视图的图层继承关系中，同步图层直接呈现在视频图层之上，如图12-4所示。这样就可以添加动画标题、水印或下沿字幕到播放视频中，并与播放器的播放栏行为保持完美同步。

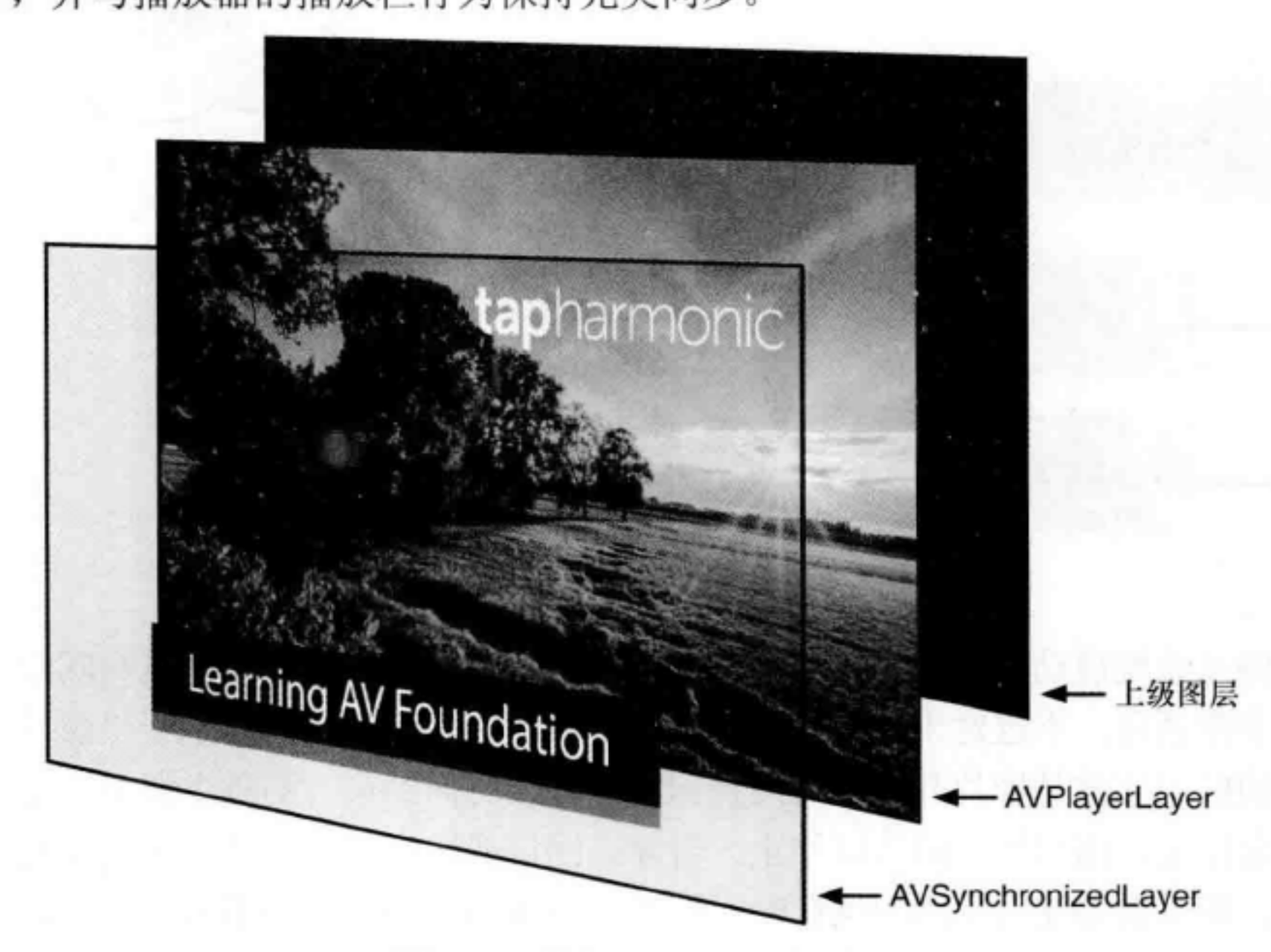

图 12-4 整合 AVSynchronizedLayer

下面来看一个小秘密，我们一直都在使用AVSynchronizedLayer类，虽然AVSynchronized-Layer最初用于在视频内容上加入动画图层内容，不过这并不是唯一的用处。当在15 Seconds

应用程序中播放一个组合时就已经见到过它的用法了，即一个动态的播放头随着视频的播放同步运动。这一功能的实现得益于AVSynchronizedLayer。在项目的Views组中可以找到一个THPlayheadView类。每当有新的AVPlayerItem创建时，应用程序就会调用视图的synchronizeWithPlayerItem:方法。这个方法创建两个CAShapeLayer实例来绘制垂直线条，还有一个CABasicAnimation实例用于在视频播放时动态调整这些图层的x轴位置。这里不对视图进行更深入的分析，不过建议大家去看一下这个类的具体实现代码，更好地理解它的工作原理。

12.2.2　使用 AVVideoCompositionCoreAnimationTool 导出

要将Core Animation图层和动画整合到导出视频中，需要使用AVVideoComposition-CoreAnimationTool类。AVVideoComposition使用这个类将Core Animation效果作为视频组合的后期处理阶段纳入，如图12-5所示。

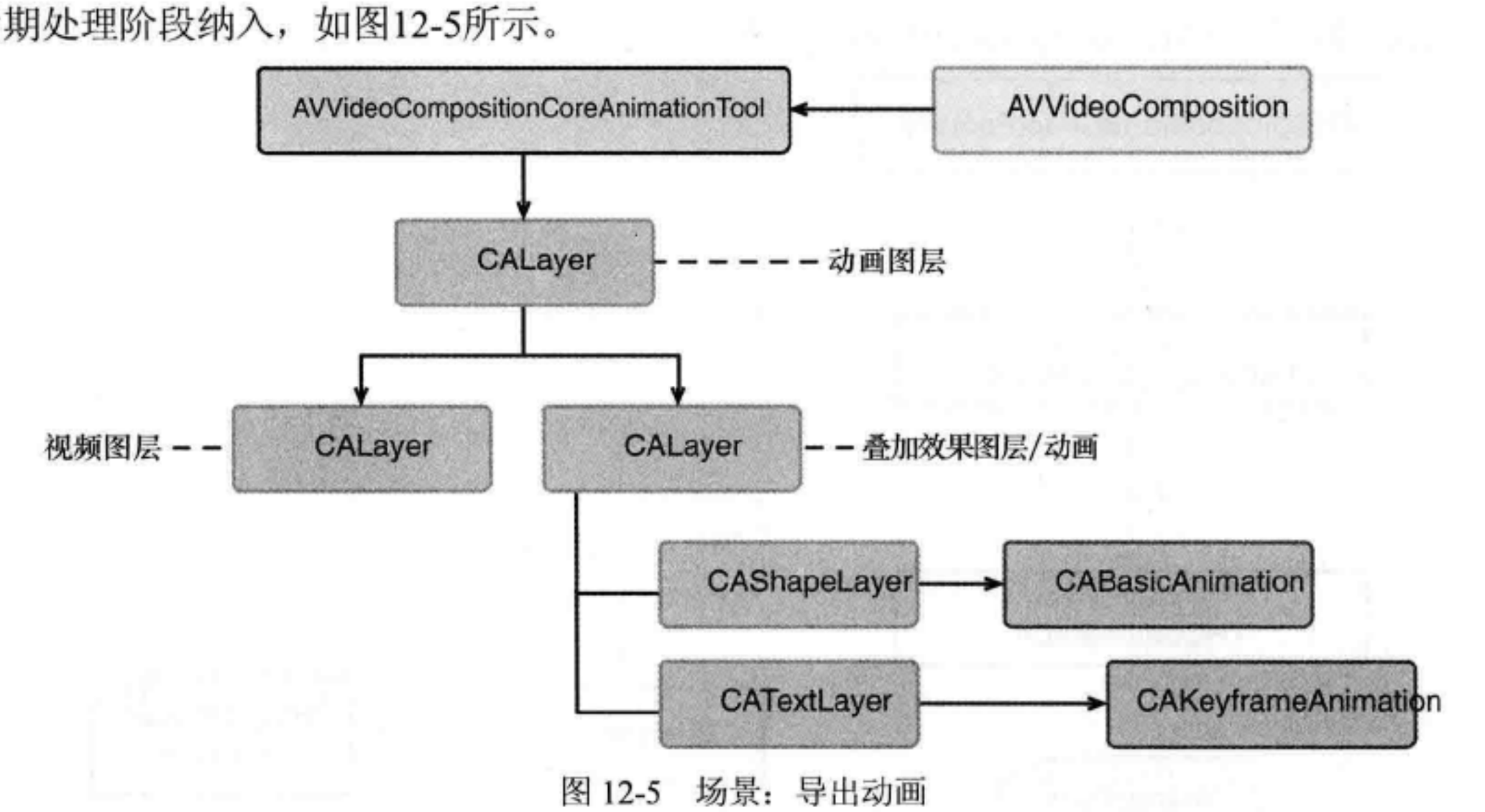

图 12-5　场景：导出动画

AVVideoCompositionCoreAnimationTool就好像一个魔力黑盒，不过幸运的是当我们知道如何使用这个类时，它就会发挥出很好的作用。它可将组合视频帧置于视频图层中，并在之后渲染包含自定义叠加效果的动画图层来生成最终的视频帧。

稍后会讨论AVSynchronizedLayer和AVVideoCompositionCoreAnimationTool的详细用法，不过首先我希望提几点在创建叠加效果时应该注意的问题：

- Core Animation 框架的默认行为是执行动画并在动画行为完成后进行处理。通常这些行为就是我们希望在实时案例中使用的，因为时间一旦过去就没法返回了。不过对于视频动画就会有问题，所以需要设置动画的 removedOnCompletion 属性为 NO 来禁用这一行为。如果没有这样做，则动画效果就是一次性的，如果用户重新播放视频或在时间轴上移动搓擦条也不会再次看到动画。
- 动画的 beginTime 属性被设置为 0.0 的话是不会看到动画效果的。Core Animation 将值为 0.0 的 beginTime 对象转换为 CACurrentMediaTime()，这是当前主机时间，同影

片时间轴中的有效时间没有关系。如果希望在影片开头加入动画，将动画的 beginTime 属性设置成 AVCoreAnimationBeginTimeAtZero 常量。

12.3 15 Seconds 应用程序：添加动画标题

在Core Animation中使用AVComposition的一个挑战就是协调不同的概念和时间模型。在使用AVComposition时，考虑的是轨道以及CMTime和CMTimeRange值。Core Animation没有轨道的概念并使用浮点型数值来表示时间。在一个简单场景中我们可以使用Core Animation自己的概念，不过当需要创建一个更复杂的案例时，最好在两个框架之间定义一个通用的抽象概念来使创建组合资源和动画时具有标准化的模型。下面看一下如何在15 Seconds应用程序中实现这一功能。

在第9章中我们第一次接触到应用程序的数据模型。下面还用之前的示例看看能否将它进行扩展，最终整合Core Animation的功能，如图12-6所示。

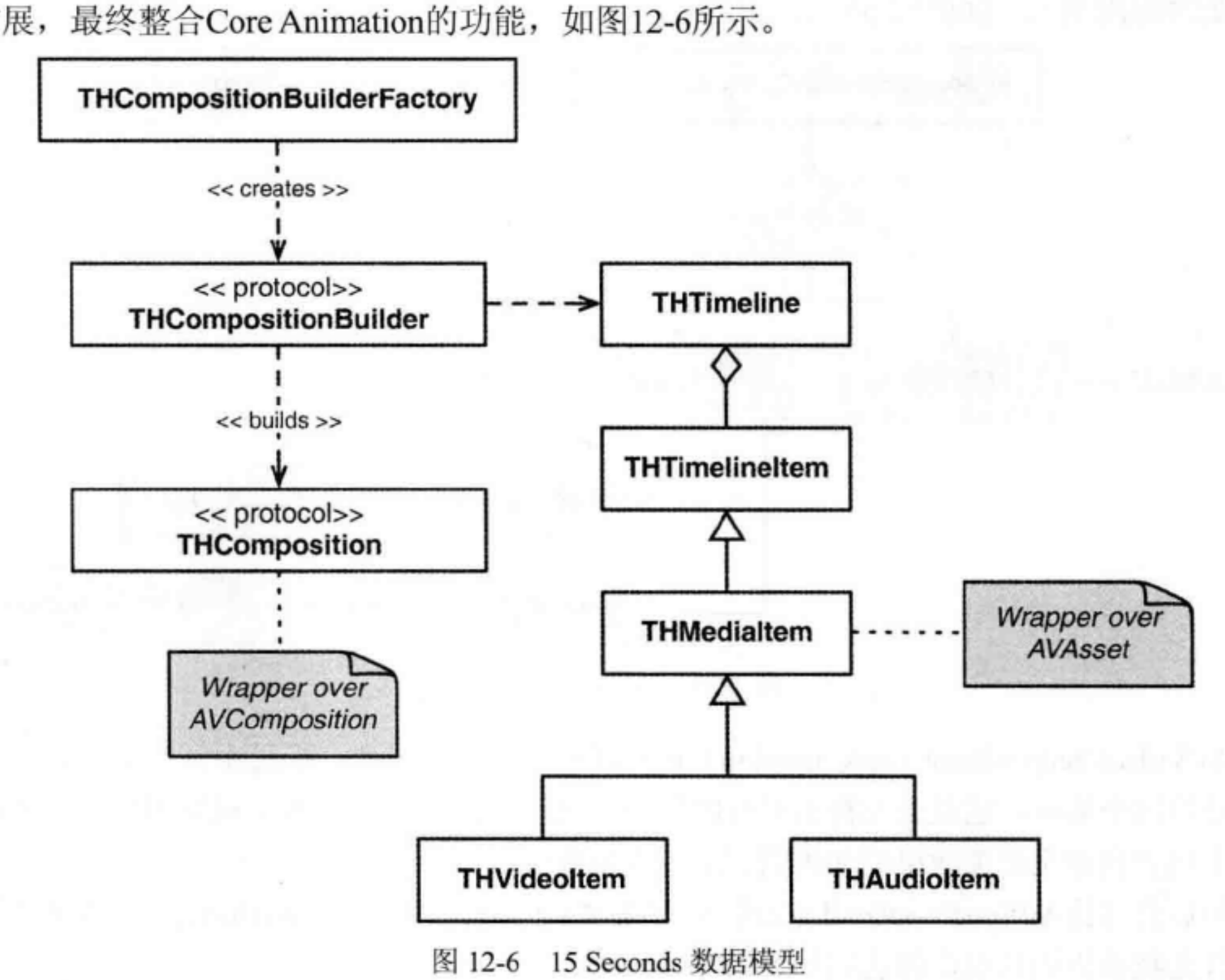

图 12-6 15 Seconds 数据模型

应用程序时间轴区域所呈现的元素都是从一个通用基类THTimelineItem中扩展而来。这个类定义了条目的时间范围和它在时间轴中的起始位置。THMediaItem以及它的子类THVideoItem和THAudioItem都是这个基类的子类，分别用于处理项目的视频和音频资源。用同样的方法处理叠加动画是可行的，因为这样就可以在应用程序的时间轴视图中以可视化的形式描述和排列它们了。应用程序的动画行为可以在THTimelineItem子类中实现。用这种方式封装动画代码可以使相应的动画操作被视为时间轴视图中的一个条目，并且在创建组合时可以为动画代码提供一个通用接口。

从Core Animation框架的角度看，应用程序继续保持简单的特征，所以你不需要成为一个Core Animation框架的专家也能理解这些内容。创建一个简单的THTimelineItem对象THTitleItem，它用于将动画标题添加到项目中。在本章初学者项目的Models组中可以找到这个类的存根实现版本。下面先来看一下它的接口，如代码清单12-1所示。

代码清单 12-1　THTitleItem 接口

```
#import "THTimelineItem.h"
#import <QuartzCore/QuartzCore.h>

@interface THTitleItem : THTimelineItem

+ (instancetype)titleItemWithText:(NSString *)text image:(UIImage *)image;
- (instancetype)initWithText:(NSString *)text image:(UIImage *)image;

@property (copy, nonatomic) NSString *identifier;
@property (nonatomic) BOOL animateImage;
@property (nonatomic) BOOL useLargeFont;

- (CALayer *)buildLayer;

@end
```

该对象的初始化用到了一个测试字符串和一个由展示标题元素组成的图片，并提供了一些属性来自定义呈现效果。该对象的核心方法是buildLayer方法，它负责创建Core Animation图层和动画。下面实现这个功能，第一步要创建一个Core Animation图层，如代码清单12-2所示。

代码清单 12-2　THTitleItem 创建图层

```
#import "THTitleItem.h"
#import "THConstants.h"

@interface THTitleItem ()
@property (copy, nonatomic) NSString *text;
@property (strong, nonatomic) UIImage *image;
@property (nonatomic) CGRect bounds;
@end

@implementation THTitleItem

+ (instancetype)titleItemWithText:(NSString *)text image:(UIImage *)image {
    return [[self alloc] initWithText:text image:image];
}

- (instancetype)initWithText:(NSString *)text image:(UIImage *)image {
    self = [super init];
    if (self) {
        _text = [text copy];
        _image = image;
        _bounds = TH720pVideoRect;                                    // 1
    }
    return self;
}
```

```
- (CALayer *)buildLayer {

    // --- Build Layers

    CALayer *parentLayer = [CALayer layer];                                    // 2
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];

    CALayer *textLayer = [self makeTextLayer];
    [parentLayer addSublayer:textLayer];

    return parentLayer;
}

- (CALayer *)makeImageLayer {                                                  // 3

    CGSize imageSize = self.image.size;

    CALayer *layer = [CALayer layer];
    layer.contents = (id) self.image.CGImage;
    layer.allowsEdgeAntialiasing = YES;
    layer.bounds = CGRectMake(0.0f, 0.0f, imageSize.width, imageSize.height);
    layer.position = CGPointMake(CGRectGetMidX(self.bounds) - 20.0f, 270.0f);

    return layer;
}

- (CALayer *)makeTextLayer {                                                   // 4

    CGFloat fontSize = self.useLargeFont ? 64.0f : 54.0f;
    UIFont *font = [UIFont fontWithName:@"GillSans-Bold" size:fontSize];

    NSDictionary *attrs =
        @{NSFontAttributeName            : font,
          NSForegroundColorAttributeName : (id) [UIColor whiteColor].CGColor};

    NSAttributedString *string =
        [[NSAttributedString alloc] initWithString:self.text attributes:attrs];

    CGSize textSize = [self.text sizeWithAttributes:attrs];

    CATextLayer *layer = [CATextLayer layer];
    layer.string = string;
    layer.bounds = CGRectMake(0.0f, 0.0f, textSize.width, textSize.height);
    layer.position = CGPointMake(CGRectGetMidX(self.bounds), 470.0f);
    layer.backgroundColor = [UIColor clearColor].CGColor;

    return layer;
}

@end
```

(1) 定义一个bounds变量，它的初始化值为TH720pVideoRect。它是THConstants.h中定义

的一个CGRect常量值，用来表示一个原点为(0, 0)，尺寸为(1280, 720)大小的矩形区域。这一尺寸符合应用程序视频的维度要求。

(2) 下一步创建Core Animation层对象。创建一个父图层，它包含相应的图片、文本图层和尺寸。还希望设置初始opacity的值为(0.0)，这样它就是不可见的。之后通过动态调整这个属性值来控制可见性。

(3) 创建一个新的CALayer用于展示图片内容，设置它的contents属性为标题项图片的CGImageRef值。Core Animation只能处理Core Graphics类型，所以需要获取UIImage对象的基础GGImageRef。设置图层的allowsEdgeAntialiasing属性为YES，这样当图片动态显示时边缘就会应用一个抗锯齿效果。最后，根据需求设置图层的bounds和position并返回图层。

(4) 创建一个新的CATextLayer用于渲染字符串内容。处理CATextLayer的最简单方法就是使用NSAttributedString，所以创建一个需要的NSAttributedString属性字典，其中包含期望的字体和前景色参数，并且为标题项的文本内容创建一个NSAttributedString。设置属性字符串作为图层的string属性，调整它的bounds和position属性，并返回图层。

创建完图层后，标题叠加的结构就完成了，不过如果尝试在当前状态下使用这个图层是不会看到任何效果的，因为父图层的模糊值被设置为(0.0)。这是一个正确的初始状态，不过在应用这一图层之前需要添加一些动画来进行融合。首先添加一个淡入淡出动画，如代码清单12-3所示。

代码清单 12-3　添加淡入淡出动画效果

```
@implementation THTitleItem

...

- (CALayer *)buildLayer {

    // --- Build Layers

    CALayer *parentLayer = [CALayer layer];
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];

    CALayer *textLayer = [self makeTextLayer];
    [parentLayer addSublayer:textLayer];

    // --- Build and Attach Animations

    CAAnimation *fadeInFadeOutAnimation = [self makeFadeInFadeOutAnimation];
    [parentLayer addAnimation:fadeInFadeOutAnimation forKey:nil];              // 1

    return parentLayer;
}

...

- (CAAnimation *)makeFadeInFadeOutAnimation {
```

```
    CAKeyframeAnimation *animation =
        [CAKeyframeAnimation animationWithKeyPath:@"opacity"];

    animation.values = @[@0.0f, @1.0, @1.0f, @0.0f];                    // 2
    animation.keyTimes = @[@0.0f, @0.20f, @0.80f, @1.0f];

    animation.beginTime = CMTimeGetSeconds(self.startTimeInTimeline);   // 3
    animation.duration = CMTimeGetSeconds(self.timeRange.duration);

    animation.removedOnCompletion = NO;                                 // 4

    return animation;
}

@end
```

(1) 父图层的模糊度被设置为(0.0)，所以我们希望创建一个动画效果将图层的模糊值在一个相应时间范围内淡出淡入。通过调用makeFadeInFadeOutAnimation创建这个动画，并使用addAnimation:forKey:方法将它添加到父图层中。如果需要在之后识别或取回这个动画则需要指定一个NSString给forKey:参数，不过本例中不需要，所以这里给这个参数赋值nil即可。

(2) 对于每个标题我们都希望快速淡入，在一个给定的时间段完全显示，并在最后渐渐淡出。这是关键帧动画的一个很好的用例，所以我们创建一个新的CAKeyframeAnimation实例操作图层对象的opacity属性。我们定义一个模糊值组成的数组及其相关keyTimes值组成的数组，keyTimes用于表示模糊值被设置的时间。时间值被标准化为同动画持续时间相关的(0.0)到(1.0)区间。可以对这些时间进行更精确的计算，不过对于示例应用程序我们使用20%的时间淡入和20%的时间淡出。

(3) 使用THTimelineItem定义的两个时间相关属性来确定动画的beginTime和duration。使用CMTimeGetSeconds函数将startTimeInTimeline属性转换为浮点型值来设置动画的beginTime属性。同样，取得对象timeRange.duration以秒计算的值并设置它作为动画的duration。

(4) 要记得经常设置动画的removedOnCompletion属性为NO，这样动画不会在执行之后被移除。如果没有明确设置这个属性，则意味着动画只能出现一次。

上述方法实现了所有标题项需要的核心淡入淡出效果。如果希望更有趣一些的话，可以对标题项的图片也使用动画效果。特别地，如果标题项的animateImage属性被设置为YES，则可以对图片应用一个旋转或弹出动画。下面看一下如何实现方法来创建这些动画效果，如代码清单12-4所示。

代码清单 12-4 为标题图片添加动画

```
- (CALayer *)buildLayer {

    // --- Build Layers

    CALayer *parentLayer = [CALayer layer];
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];
```

```
    CALayer *textLayer = [self makeTextLayer];
    [parentLayer addSublayer:textLayer];

    // --- Build and Attach Animations

    CAAnimation *fadeInFadeOutAnimation = [self makeFadeInFadeOutAnimation];
    [parentLayer addAnimation:fadeInFadeOutAnimation forKey:nil];

    if (self.animateImage) {

        parentLayer.sublayerTransform = THMakePerspectiveTransform(1000);    // 1

        CAAnimation *spinAnimation = [self make3DSpinAnimation];

        NSTimeInterval offset =                                              // 2
            spinAnimation.beginTime + spinAnimation.duration - 0.5f;

        CAAnimation *popAnimation =
            [self makePopAnimationWithTimingOffset:offset];

        [imageLayer addAnimation:spinAnimation forKey:nil];                  // 3
        [imageLayer addAnimation:popAnimation forKey:nil];

    }

    return parentLayer;
}

...

static CATransform3D THMakePerspectiveTransform(CGFloat eyePosition) {
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / eyePosition;
    return transform;
}

- (CAAnimation *)make3DSpinAnimation {

    // To be implemented

    return nil;
}

- (CAAnimation *)makePopAnimationWithTimingOffset:(NSTimeInterval)offset {

    // To be implemented

    return nil;
}

@end
```

(1) 应用一个3D动画效果绕y轴旋转。要实现这个效果，需要使用父图层的sublayerTransform属性设置一个透视变化。我们在第7章讨论人脸识别时也用到了类似效果。这个方法将图片的二维平面映射到三维空间。

(2) 创建一个CAAnimation来应用旋转动画并计算一个偏移量，用于创建弹出动画。偏移

量会用到旋转动画的beginTime和duration，之后从这个值减掉半秒。这样就在旋转动作最后半秒加入了一个弹出动画。

(3) 使用addAnimation:forKey:方法将两个动画添加到imageLayer。

> **注意：**
>
> 核心动画提供了一个 CAAnimationGroup 类，用于将多个动画组合在一起同步执行。如果你是一个有经验的 Core Animation 开发者，可能就会问为什么不把旋转和弹出动画放在一起。虽然我们从来没有得到一个明确的说明，指出动画组不能用于 AVSynchronizedLayer，不过还是建议大家在创建视频动画时不要尝试这么做。

下面具体实现这两个动画方法，来完成THTitleItem类，如代码清单12-5所示。

代码清单 12-5 图片动画方法

```
- (CAAnimation *)make3DSpinAnimation {

    CABasicAnimation *animation =                                          // 1
        [CABasicAnimation animationWithKeyPath:@"transform.rotation.y"];

    animation.toValue = @((4 * M_PI) * -1);                                // 2

    animation.beginTime = CMTimeGetSeconds(self.startTimeInTimeline) + 0.2;  // 3
    animation.duration = CMTimeGetSeconds(self.timeRange.duration) * 0.4;

    animation.removedOnCompletion = NO;

    animation.timingFunction =                                             // 4
        [CAMediaTimingFunction
            functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

    return animation;
}

- (CAAnimation *)makePopAnimationWithTimingOffset:(NSTimeInterval)offset {

    CABasicAnimation *animation =                                          // 5
        [CABasicAnimation animationWithKeyPath:@"transform.scale"];

    animation.toValue = @1.3f;                                             // 6

    animation.beginTime = offset;                                          // 7
    animation.duration = 0.35f;

    animation.autoreverses = YES;                                          // 8

    animation.removedOnCompletion = NO;

    animation.timingFunction =
        [CAMediaTimingFunction
            functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

    return animation;
}
```

(1) 首先创建一个CABasicAnimation为图片的变形添加动画效果。Core Animation Programming Guide文档给出了一些Key-Value Coding Extensions的介绍，可以帮助我们很容易地实现上述功能。本例使用transform.rotation.y键路径让图片绕y轴旋转。

(2) 我们要实现的效果是图片以逆时针的方向绕y轴旋转两圈。一圈360度即2π弧度，所以我们设置4 * M_PI实现旋转2周。

(3) 采用与淡入淡出动画类似的方式计算beginTime和duration。需要进行一些小的计算来调整这些时间。对于这些计算没有神奇的公式，只有看起来最适合的方法，所以开发者只需选择自己熟悉的方法即可。

(4) 为动画应用一个时间函数。默认情况下计算插值的时候动画会使用线性曲线。这么做的线条会略显僵硬，通常我们会使用一些柔和的处理方法让曲线看起来更自然。本例中使用缓入缓出曲线。

(5) 对于弹出动画，需要创建一个CABasicAnimation为图片的变形过程应用一个小的缩放效果。

(6) 设置动画的toValue值为@1.3，将图片放大30%。

(7) 由于这个动画是作为前一个动画的一部分，所以设置beginTime为传递给方法的偏移量，并设置持续时间为0.35。duration值可以任意取，所以根据首选项进行设置。

(8) 动画对象具有自动回溯功能。这是我们在执行动画时一个非常方便的方法，可以反向运行动画，使它返回到初始状态。设置autoreverses属性为YES，在动画最后快速缩放图片生成一个小的跳动效果。

THTitleItem类已经完成了，不过在看到这一效果前还需要做一点工作。建议大家运行本章的FifteenSeconds_Final项目来看实际的完成效果。一些新的用户界面元素被加入到这个应用程序中，可以让用户实验多种动画效果。Settings菜单现在包含一个Video Titles开关用于激活动画轨道。在播放栏区域添加了一个UISlider，这样在播放组合时就可以来回调节，并且可以看到动画完美地与时间轴锁定，如图12-7所示。

图 12-7　实际的动画效果

注意：

当在菜单中激活了 Video Titles 开关时，会在时间轴区域添加两个预置标题。构建这些标题项的代码可以在 THTimelineViewController 类的 addTitleItems 方法中找到。

在下一节中，我们会介绍如何在组合上使用THTitleItem。

12.4 准备组合

不需要在本章创建新的THComposition和THCompositionBuilder对象，因为前一章中创建的对象已经可以满足我们的需求了。在示例项目中找到THOverlayComposition和THOverlay CompositionBuilder类，它们都是从之前一章中实现的方法复制过来的，只不过稍加修改，整合了上一节中创建的Core Animation相关的代码，如代码清单12-6所示。

代码清单 12-6 THOverlayComposition 接口

```
@interface THOverlayComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVComposition *composition;
@property (strong, nonatomic, readonly) AVVideoComposition *videoComposition;
@property (strong, nonatomic, readonly) AVAudioMix *audioMix;
@property (strong, nonatomic, readonly) CALayer *titleLayer;

- (id)initWithComposition:(AVComposition *)composition
        videoComposition:(AVVideoComposition *)videoComposition
                audioMix:(AVAudioMix *)audioMix
             titleLayer:(CALayer *)titleLayer;
@end
```

上面的代码同前一章中创建的一样，不过在初始化方法中增加了一个新的CALayer属性和一个新的titleLayer参数。在我们讨论对类实现进行改动来使用THTitleItem内容前，首先看一下在THOverlayCompositionBuilder类中增加的内容。

组合创建方法需要进行一些小改动。最关键的一点是在THTimeline中获取THTitleItem实例，将它们汇总到一个单独的CALayer对象中，并将它添加到THOverlayComposition中，如代码清单12-7所示。

代码清单 12-7 创建视频图层

```
@implementation THOverlayCompositionBuilder

...

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition composition];

    [self buildCompositionTracks];

    AVVideoComposition *videoComposition = [self buildVideoComposition];

    return [[THOverlayComposition alloc]                                    // 1
              initWithComposition:self.composition
                 videoComposition:videoComposition
                        audioMix:[self buildAudioMix]
                       titleLayer:[self buildTitleLayer]];
}
```

```
- (CALayer *)buildTitleLayer {

    if (!THIsEmpty(self.timeline.titles)) {
        CALayer *titleLayer = [CALayer layer];                                  // 2
        titleLayer.bounds = TH720pVideoRect;
        titleLayer.position =  CGPointMake(CGRectGetMidX(TH720pVideoRect),
                                           CGRectGetMidY(TH720pVideoRect));

        for (THTitleItem *compositionLayer in self.timeline.titles) {           // 3
            [titleLayer addSublayer:[compositionLayer buildLayer]];
        }

        return titleLayer;
    }

    return nil;
}

...

@end
```

(1) 创建并返回一个THOverlayComposition实例。与上一章的代码相比，唯一的变化时调用了私有方法buildTitleLayer，用这个方法返回一个包含动画图层效果的CALayer实例。

(2) 创建一个新的CALayer实例并设置它的bounds和position。在计算这些值的时候我们仍然使用TH720pVideoRect常量。15 Seconds中只用到720p视频剪辑，不过在我们自己开发的应用程序中，这一图层的尺寸要根据实际情况设置。

(3) 遍历时间轴包含的所有标题数据，对于每一个标题调用它的buildLayer方法并将返回的图层添加到titleLayer。

THOverlayCompositionBuilder类已经完成了，最后一件事是讨论在THOverlayComposition类中创建AVPlayerItem和AVAssetExportSession时如何使用这些图层。

12.4.1　播放时应用 Core Animation

要将这些层整合到应用程序的播放功能中，需要使用AVSynchronizedLayer类。在组合初始化方法中创建的图层将被添加到同步图层，确保动画的播放与视频时间轴同步。下面看一下THOverlayComposition中的makePlayable方法的具体实现，如代码清单12-8所示。

代码清单 12-8　播放功能中使用 Core Animation

```
- (AVPlayerItem *)makePlayable {

    AVPlayerItem *playerItem =
        [AVPlayerItem playerItemWithAsset:[self.composition copy]];

    playerItem.videoComposition = self.videoComposition;
    playerItem.audioMix = self.audioMix;

    if (self.titleLayer) {                                                      // 1
        AVSynchronizedLayer *syncLayer =
            [AVSynchronizedLayer synchronizedLayerWithPlayerItem:playerItem];
```

```
        [syncLayer addSublayer:self.titleLayer];

        // WARNING: The the 'titleLayer' property is NOT part of AV Foundation
        // Provided by AVPlayerItem+THAdditions category.
        playerItem.syncLayer = syncLayer;                                    // 2
    }

    return playerItem;
}
```

(1) 如果一个有效的titleLayer存在，则使用当前的AVPlayerItem创建一个新的AVSynchronized-Layer实例。添加titleLayer作为子图层，确保所有动画都和AVPlayerItem实例保持同步。

(2) 为AVPlayerItem添加一个分类方法将AVSynchronizedLayer传递到应用程序的THPlayerViewController中。这么做是为了将AVSynchronizedLayer的使用限制在这个类中以简化讨论。在实际应用程序中，AVSynchronizedLayer的创建和管理最好都在THPlayerViewController自身进行处理。

载入默认的组合，在Setting菜单激活Video Titles开关，并播放组合。现在我们已经实现了和视频内容同步播放的动画标题。播放组合时，可以使用播放按钮旁的滑动条调整视频进度来检查动画效果是否和时间轴保持同步。播放场景的工作已经完成了，下面继续讨论导出组合时如何应用这些效果。

12.4.2 导出时应用 Core Animation

在导出场景中整合Core Animation图层，需要用到AVVideoCompositionCoreAnimationTool对象。这个对象允许将Core Animation图层作为视频组合的后期处理阶段纳入。这个对象用起来比较简单，不过并不是特别直观，所以下面看一下如何使用这个工具，如代码清单12-9所示。

代码清单 12-9 导出功能中使用 Core Animation

```
- (AVAssetExportSession *)makeExportable {

    if (self.titleLayer) {

        CALayer *animationLayer = [CALayer layer];                          // 1
        animationLayer.frame = TH720pVideoRect;

        CALayer *videoLayer = [CALayer layer];
        videoLayer.frame = TH720pVideoRect;

        [animationLayer addSublayer:videoLayer];
        [animationLayer addSublayer:self.titleLayer];

        animationLayer.geometryFlipped = YES;                               // 2

        AVVideoCompositionCoreAnimationTool *animationTool =                // 3
            ➥[AVVideoCompositionCoreAnimationTool
➥videoCompositionCoreAnimationToolWithPostProcessingAsVideoLayer:videoLayer
➥inLayer:animationLayer];

        AVMutableVideoComposition *mvc =
            (AVMutableVideoComposition *)self.videoComposition;
```

```
    mvc.animationTool = animationTool;                                    // 4
}

NSString *presetName = AVAssetExportPresetHighestQuality;
AVAssetExportSession *session =
    [[AVAssetExportSession alloc] initWithAsset:[self.composition copy]
                                     presetName:presetName];
session.audioMix = self.audioMix;
session.videoComposition = self.videoComposition;

return session;
}
```

(1) 首先创建两个CALayer实例。组合的视频帧被放在videoLayer中，animationLayer将被渲染并生成导出使用的最终视频帧。

(2) 设置动画图层的geometryFlipped属性为YES来确保标题被正确渲染。如果没有设置这个值将导致图片和文本的位置发生颠倒。

(3) 使用videoCompositionCoreAnimationToolWithPostProcessingAsVideoLayer:inLayer初始化方法创建一个新的AVVideoCompositionCoreAnimationTool实例，为其传递动画和视频图层。这样做的目的是在视频组合的后处理阶段将视频帧和Core Animation图层整合到一起。

(4) 设置AVVideoCompositionCoreAnimationTool作为视频组合的animationTool属性，供在导出时使用。

在设备上运行该应用程序并载入默认的组合。确保在Settings菜单中激活了Video Titles并点击Export Composition菜单项。在一个简短的导出过程之后就可以切换到iOS的相机应用程序，并播放视频。可以看到我们设置的动画标题图层成功地在导出的视频内容中渲染，如图12-8所示。

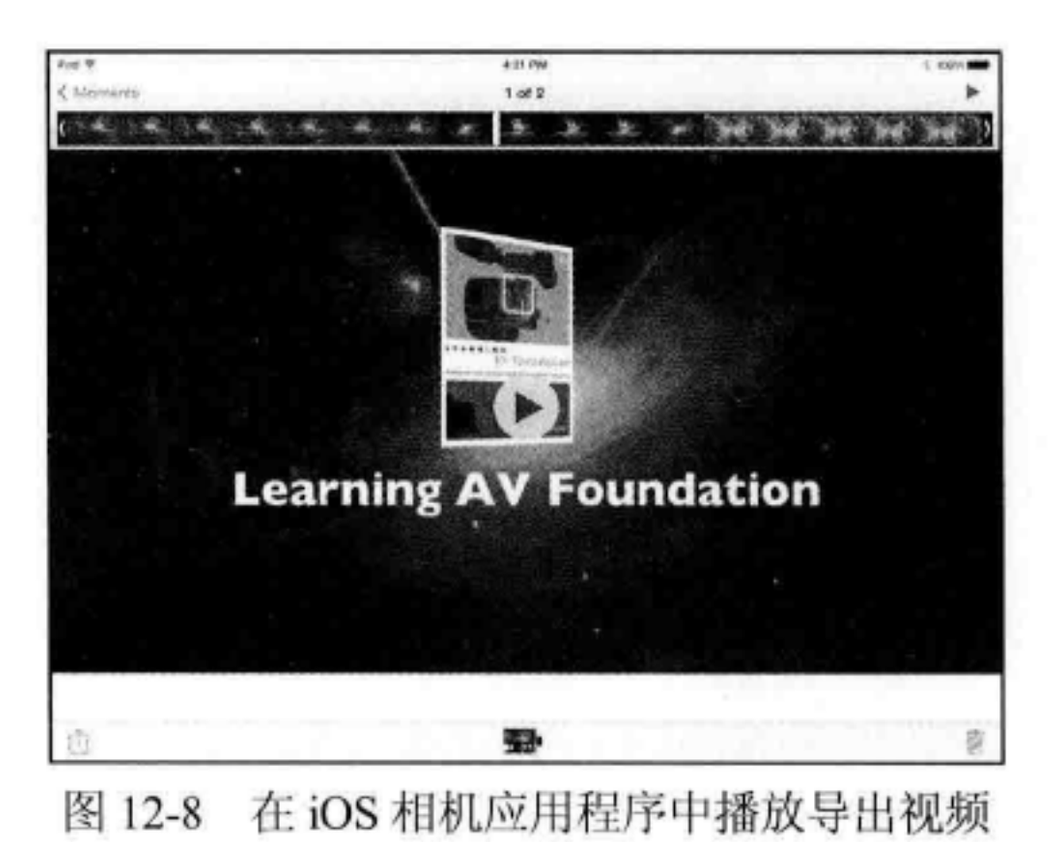

图 12-8　在 iOS 相机应用程序中播放导出视频

12.5　小结

本章我们完成了15 Seconds应用程序的全部内容，结果看起来非常棒！我们学习了如何使用AVSynchronizedLayer在视频播放中添加Core Animation效果，还有如何在导出过程中使用AVVideoCompositionCoreAnimationTool整合这些效果。AV Foundation向视频项目中添加标

题的方法最初看起来比较奇怪，不过当你习惯了这种方法后就会非常适应。本章还大致介绍了一种用于简化AV Foundation和Core Animation整合过程的解决方案。相信类似的设计模式在开发者创建更大或更复杂的视频应用程序时也会提供很大的帮助。

12.6 挑战

我们已经在播放和导出过程中建立了Core Animation的基础架构，所以使用这一架构实现自定义的动画效果，比如添加一个静态水印到视频；或者整合一些交替显示的标题；创建星球大战滑动文本效果。Core Animation有许多有趣的效果，你可以使用这些效果为视频项目创造出非常漂亮的动画重叠效果。