

Core Data 应用开发实践指南

[美] Tim Roadley 著 爱飞翔 译

Learning Core Data for iOS
A Hands-On Guide to Building Core Data Applications

- 深度剖析Core Data应用开发的全过程，全面涵盖苹果开发平台的新特性以及一些新的编程范式
- 以Grocery Dude购物管理程序贯穿始终，循序渐进讲解Core Data的各项知识点，包含大量实用开发技巧



Core Data 应用开发实践指南

Learning Core Data for iOS
A Hands-On Guide to Building Core Data Applications

[美] Tim Roadley 著 爱飞翔 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Core Data 应用开发实践指南 / (美) 罗德雷 (Roadley, T.) 著 ; 爱飞翔译 . — 北京 : 机械工业出版社, 2014.11

(iOS/ 苹果技术丛书)

书名原文 : Learning Core Data for iOS: A Hands-On Guide to Building Core Data Applications

ISBN 978-7-111-48226-0

I. C… II. ① 罗… ② 爱… III. 移动终端—应用程序—程序设计—指南 IV. TN929.53-62

中国版本图书馆 CIP 数据核字 (2014) 第 236968 号

本书版权登记号 : 图字 : 01-2014-2025

Authorized translation from the English language edition, entitled *Learning Core Data for iOS: A Hands-On Guide to Building Core Data Applications*, 9780321905765 by Tim Roadley, published by Pearson Education, Inc., Copyright © 2014.

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 China Machine Press Copyright © 2015.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括中国台湾地区和香港、澳门特别行政区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

Core Data 应用开发实践指南

出版发行 : 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 : 100037)

责任编辑 : 关 敏

责任校对 : 董纪丽

印 刷 : 北京市荣盛彩色印刷有限公司

版 次 : 2015 年 1 月第 1 版第 1 次印刷

开 本 : 186mm × 240mm 1/16

印 张 : 25

书 号 : ISBN 978-7-111-48226-0

定 价 : 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线 : (010) 88378991 88361066

投稿邮箱 : (010) 88379604

购书热线 : (010) 68326294 88379649 68995259

读者信箱 : hzjsj@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问 : 北京大成律师事务所 韩光 / 邹晓东

The Translator's Words 译者序

在开发 iOS 应用程序的时候，经常需要考虑怎样管理数据，而 Core Data 正是一种易于使用的数据库管理框架。若想系统地掌握 Core Data 的用法，那不妨花点时间看看这本书。与讲解 Core Data 的其他教材相比，本书有几个特点值得关注。

首先，它用 Grocery Dude 购物管理程序来贯穿 Core Data 的各项知识点，使读者能够以直观流畅的方式学会多种实用技巧，并且可以把在范例项目里学到的经验运用于自己的项目中。其次，它将全过程分解为很多章节，再将这些章节细分为若干步骤。这种循序渐进的讲解形式，令读者能够及时检视自己的学习进度，而且还能够清楚地了解每个知识点对于项目功能所起的作用。你只需把这些步骤稍加改编，即可将其推广到其他项目。最后，每章后面都有数道习题，这些习题不仅有助于培养读者的试验能力，而且还能够为深入研究 Core Data 提供一些线索。

本书内容可以分成三部分。前 7 章可以视为基础篇。一开始我们就会知道 Core Data 的适用场合，并且学会怎样为现有程序添加 Core Data 支持。其后，本书作者从基础知识、迁移方式及扩展方式这三个角度来讲解 Core Data 的托管对象模型。学会它的用法之后，作者将向大家演示怎样用图形界面来操作 Core Data 数据，具体来说，就是怎样用表格视图、视图及选取器视图这三种界面，打造一款简单而易用的 Core Data 程序。对于想要迅速学会 Core Data 的程序员来说，看完这 7 章之后，应该就能初步做出一款得体的 Core Data 程序了。

第 8 ~ 12 章可以看作进阶篇，作者深入讲解了如何为程序配备默认数据、如何更加精细地控制数据迁移、如何实现高效率的数据搜索等主题。其中，第 10 章尤为精彩，作者给程序添加了拍照功能，并以此为例，演示怎样寻找程序的性能瓶颈、怎样用各种工具来分析造成瓶颈的原因，以及怎样解决性能瓶颈。这种处理问题的思路，对于我们制作其他软件产品也是很有启发的。

最后 4 章应该算是扩展篇，通过与各种网络框架相集成，我们能够优雅地实现数据备份、

数据恢复、多台设备间的数据同步以及多人协作等功能，从而进一步提高 Core Data 应用程序的品质。虽说作者是以 Dropbox、iCloud 及 StackMob 为例来讲解的，但大家也可以用类似的方式把 Core Data 程序与其他的云端硬盘或网络服务集成起来。

在翻译过程中，我得到了机械工业出版社华章公司诸位编辑与工作人员的帮助，在此深表谢意，还要感谢 goldlion 及 ChenGe 两位友人对术语翻译工作所提的建议。

由于时间仓促，译者水平有限，错误与疏漏之处在所难免，敬请各位读者批评指正。你可发邮件至 eastarstormlee@gmail.com 与我联系，也可访问 <http://agilemobidev.com/eastarlee/book/learning-core-data-for-ios/?variant=zh-hans> 网页留言。如果对某些术语的翻译有意见或建议，欢迎来 <https://github.com/jeffreybaoshenlee/IT-Terms-EN-CN/issues> 一起讨论。

爱飞翔

Preface 前言

每天都有无数 Apple 设备运行着依赖于 Core Data 的应用程序。这使得 Core Data 成了一个成熟、稳定且非常快速的平台,以供应用程序访问其数据。Core Data 本身并不是数据库,它其实是一个拥有诸多功能的框架,而其中一项功能就是把应用程序同数据库之间的交互过程自动化。有了它之后,就不用再编写 SQL 代码了,而是可以改用 Objective-C 对象来做。这样一来,既能享受到关系型数据库的好处,又无须在 Objective-C 代码中编写、测试并优化 SQL 查询语句。Core Data 会在幕后自动生成 SQL 代码,而 Apple 公司的专业技术人员已经对这种 SQL 代码做了多年的改良与优化。使用 Core Data 不仅能缩短应用程序开发时间,而且还能显著减少开发者所要编写的代码量。

Core Data 的显著特性有:

- ❑ 变更管理(撤销与重做)
- ❑ 关系 (relationship)
- ❑ 数据模型的版本管理及迁移
- ❑ (通过 batching 及 faulting) 高效地获取数据
- ❑ (通过谓词) 高效地过滤数据
- ❑ 数据一致性 & 数据验证

本书将介绍 Core Data 的特性及最佳实践技巧。在学习各章的过程中,你会明白如何从头开始构建一款功能完备的 Core Data iPhone 应用程序。笔者会详细解释每个关键的知识点,使你能够直接把学到的内容付诸实践。本书所展示的范例程序会尽量把 Core Data 的各个方面都纳入其中。同时,它还是个已在 App Store 上架的真实应用程序。这更有助于你把学到的知识与现实工作中的场景联系起来。

随着 iOS 7 的到来,Core Data 与 iCloud 之间的集成在速度、可靠性及简洁程度上都有了大幅改观。对原来放弃了这项技术的人,笔者建议你再试一次,这回肯定能给你带来惊喜。

如果你想对本书内容提供反馈、bug 修复及勘误，或想为本书后续版本出力，请通过电子邮箱 timroadley@icloud.com 联系笔者。最后，感谢你关注此书。笔者花了大量时间来精心编排内容，也衷心希望它能帮助你掌握 Core Data 这项出色的技术。

本书的目标读者

本书写给那些想在 iOS app 中高效管理数据的 Objective-C 程序员。如果原来有数据库方面的经验，那么某些内容学起来可能会快一些，但没有数据库经验的程序员也同样可以阅读本书。某些固守旧习的 SQL 程序员可能很难适应 Core Data 的一些用法。但无论你的技术背景是什么，都无需担心，因为笔者会把每个步骤都解释得非常清楚。

学习本书所需的材料和知识

身为 Objective-C 程序员，你应该装有比较新的 Mac 系统，并运行 Xcode 5 或更新的版本。同时还应该非常熟悉 Xcode 并且拥有一部 iOS 设备，以便测试。iOS 设备对本书第 10 章尤其重要，因为整章都在谈设备性能问题。

你应该知道 Objective-C 中一些术语的含义，例如 *property*（特性）、*method*（方法）、*delegate*（代理）、*class*（类）和 *class instance*（类实例）。若你无法确定本书是否适合自己，建议你先看看下面这些资料：

- ❑ 《iOS Programming: The Big Nerd Ranch Guide》（请搜索 amazon.com 网站）
- ❑ 《The iOS Newbie Tutorial Series》（请搜索 timroadley.com 网站）
- ❑ 《Learning Objective-C: A Primer》^①（请搜索 apple.com 网站）

本书结构

本书将指导你完成 **Grocery Dude** 与 **Grocery Cloud** 的整个构建过程，二者均是在 App Store 上架的应用程序。Grocery Dude 演示了 Core Data 如何与 iCloud 集成，而 Grocery Cloud 则演示了 Core Data 如何与 StackMob 集成。本书的每一章都要用到上一章所讲的知识，而按照各章顺序也会看到所需实现的内容。在这个过程中，我们要构建一些辅助类（helper class），以便更快地重新部署已经完成的这部分应用程序。实际上，第 15 章最后的那个习题就会引导你把这些辅助类重新部署到既有的非 Core Data 应用程序上面。你很快就能拥有一款功能完备的 Core Data app 了，它会与 iCloud 可靠地集成起来。

现将每章内容简述如下：

① 该文档现已更名为《About Objective-C》，网址是：<https://developer.apple.com/library/Mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>。——译者注

第 1 章 初次尝试 Core Data 应用程序——此章是本书的基础，笔者会在其中介绍 Core Data 的基本概念，还会告诉你关于 Core Data 的两个重要方面，就是“它能做什么”以及“它不能做什么”。此外，还会实现 CoreDataHelper 类，用以演示如何将 Core Data 同既有的应用程序相集成。

第 2 章 托管对象模型的基础知识——这一章将对传统数据库模式的设计与 Core Data 之间有何共性，并介绍数据模型。在讨论实体（entity）和属性的时候，你将看到如何配置基本的托管对象模型（managed object model），同时笔者还会给出建议，告诉你如何选择合适的数据类型。此外还会讲解怎样插入、获取、过滤、排序及删除“托管对象”，其后还会介绍获取请求（fetch request）模板。

第 3 章 托管对象模型的迁移——本章将会讲解三种迁移方式，分别是：轻量级迁移、默认迁移以及采用迁移管理器来迁移，其中，迁移管理器可以显示迁移的进度。你将学到如何在各种迁移方式之间做出明智的抉择，同时还将学会适应 Core Data 的模型版本管理（model-versioning）功能。

第 4 章 托管对象模型的扩展——笔者会解释各种关系并将其添加到 Grocery Dude 程序中，此时关系数据模型的强大之处就能体现出来了。模型的其他特性，诸如抽象实体及父实体也会在这一章里讲到，同时笔者还会告诉你一些处理数据验证错误的技术。

第 5 章 表格视图——用 Core Data 及获取结果（fetched result）控制器来驱动表格视图（table view），既可节省内存，又能提升效率，而且这么做也会令应用程序初具规模。当然，大部分的例行任务都是由 CoreDataTVC 来完成的，这是个可以复用的子类，继承自 UITableViewController。只需把这个子类放到你自己的应用程序里，就能轻松地部署好一份由 Core Data 所驱动的表格视图了。

第 6 章 视图——本章将展示怎样在应用程序里传递托管对象，由此可以学会如何操作它们。在表格视图里选定的对象可以传给下一个视图，以供编辑。本章会给 Grocery Dude 加上编辑用的界面，以演示如何操作传过来的对象，并且还会演示如何将其存回持久化存储区里面。

第 7 章 选取器视图——本章会把由 Core Data 所驱动的选取器视图（Picker View）添加到编辑视图里面，这样可以令应用程序更加精致。用户可以通过选取器视图将某种计量单位、家庭住址或商铺地址快速地设定到现有的货品上面。笔者专门制作了一个可以复用的子类——CoreDataPickerTF，它继承自 UITextField，当用户点击相关的文本框时，这个子类可以用由 Core Data 所驱动的选取器视图来取代默认的输入键盘。

第 8 章 预先加载数据——本章将会解释如何用 XML 中的默认数据来生成持久化存储区，同时还会介绍通用的辅助类 CoreDataImporter。为 Grocery Dude 程序准备好持久

化存储区之后，笔者将演示如何判断是否需要导入默认数据（有时用户可能根本不想导入数据）。

第 9 章 深拷贝——与 `migratePersistentStore` 相比，深拷贝（deep copy）更加灵活也更加精细，它可以从选定的实体中把对象与关系由一个持久化存储区拷贝到另一个存储区中。本章将改进 `CoreDataImporter` 这个辅助类，为其增加深拷贝功能。

第 10 章 性能——编写 Core Data 应用程序的时候，很容易出现一些常见的性能问题，你将会通过本章学到如何用 Instruments 来判断并解决这些问题，并可以由此积累经验。笔者给程序加入照相功能，是为了把性能问题凸现出来，从而使大家明白：要想编出性能优秀的应用程序，就必须把模型设计好。

第 11 章 后台处理——性能一流的程序会把繁重的处理任务转交给后台线程来做。笔者将通过名为 `Thumbnailer` 的辅助类来添加相片缩略图生成功能，由此你会看到在后台处理任务其实是非常简单的。

第 12 章 搜索——本章将会在 `CoreDataTVC` 里实现高效的搜索功能，你将学到如何在同一个表格视图中处理两个获取结果控制器。

第 13 章 与 Dropbox 相结合的备份与恢复——本章将会讲解如何创建备份文件以及如何用 Dropbox 的 Sync API 来同步这些文件。我们还会实现数据恢复功能：用户只需按一下按钮，就能把同一个 Dropbox 账户中的数据恢复到任意 iOS 设备中。

第 14 章 iCloud——本章将会讲解 Core Data 与 iCloud 的集成，这种集成方式目前是最简单、最可靠的。iCloud 可以安全地处理多个账户及各种配置信息，不会有丝毫损失。

第 15 章 iCloud 高级使用技巧——本章将会进一步提升 Core Data 与 iCloud 的集成幅度，以实现实体级别的数据散播（entity-level seeding）及去除重复数据（de-duplication）的功能（该功能可以保证每个对象只存储一份）。此外，你还会学到如何通过正确的方式来准确地模拟用户第一次使用 iCloud 时的情形，这种方式可以把相关的内容全部重置。

第 16 章 与 Web Service 相集成——本章通过 StackMob 来介绍如何在多个用户之间进行跨平台的数据分享及协作。StackMob 是个极为优秀的免费 BaaS（Backend-as-a-Service 的缩写），它直接提供了针对 Core Data 的 iOS API。StackMob 允许笔者在书中使用其美术资源，并且对本书第 16 章有所帮助，笔者在此对 StackMob 表示感谢。

附录 A 为第 1 章的 Grocery Dude 程序所做的准备工作——本书第 1 章要以 Grocery Dude 程序为起点展开讲解，为了更完整地演示此程序的制作过程，笔者把程序里面与 Core Data 无关的步骤都放在了附录中。

附录 B 为第 16 章的 Grocery Cloud 程序所做的准备工作——本书第 16 章要以 Grocery Cloud 程序为起点展开讲解，为了更完整地演示此程序的制作过程，笔者把程序里

面与 Core Data 无关的步骤都放在了附录中。

范例代码获取方式

本书范例代码均可从 timroadley.com 网站下载。每一章都会给出相关的链接，也可以参考表 1，该表按照实现的先后顺序列出了各范例代码的下载链接。

表 1 Grocery Dude 代码下载链接汇总

| 最终代码 | 下载链接 |
|--------------------------|---|
| 附录A | http://timroadley.com/LearningCoreData/GroceryDude-AfterAppendixA.zip |
| 第1章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter01.zip |
| 第2章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter02.zip |
| 第3章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter03.zip |
| 第4章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter04.zip |
| 第5章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter05.zip |
| 第6章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter06.zip |
| 第7章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter07.zip |
| 第8章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter08.zip |
| 第9章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter09.zip |
| 第10章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter10.zip |
| 第11章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter11.zip |
| 第12章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter12.zip |
| 第13章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter13.zip |
| 第14章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter14.zip |
| 第15章 | http://timroadley.com/LearningCoreData/GroceryDude-AfterChapter15.zip |
| 第15章 Mini项目 | http://timroadley.com/LearningCoreData/EasyiCloud.zip |
| 辅助类，可供 你在自己的 项目中使用 | http://timroadley.com/LearningCoreData/Generic%20Core%20Data%20Classes.zip |
| 附录B | http://timroadley.com/LearningCoreData/GroceryCloud-AfterAppendixB.zip |
| 第16章 | http://timroadley.com/LearningCoreData/GroceryCloud-AfterChapter16.zip |

请注意，有时候一行代码会比较长，从而超出了书的宽度。在这种情况下，笔者用“代码接续箭头”（➡）来表示换行。例如：

```
[[NSURL URLWithString:[self applicationDocumentsDirectory]]
➡URLByAppendingPathComponent:@"Stores"];
```


致谢

首先感谢 Trina MacDonald 给了我写作本书的机会，她在成书过程中亦对笔者提供了莫大的帮助，而 Rich Warren、Carl Brown、Mark Granoff 及 Ricky O’Sullivan 这四位技术评审也做得相当出色。笔者熬夜写书时肯定出了一些差错，你们都帮着找了出来，并提供了一些独到的见解和代码编写技巧。此外还要特别感谢 Betsy Gratner、Olivia Basegio、Bart Reed、Sheri Cain、Chris Zahn 及 Matt Vaznaian 在成书过程中对笔者的协助。

——Tim Roadley (Twitter 用户名 : @TimRoadley)

Contents 目 录

译者序

前 言

| | |
|--|----|
| 第 1 章 初次尝试 Core Data 应用程序 | 1 |
| 1.1 Core Data 是什么 | 1 |
| 1.2 Core Data 的适用场合 | 4 |
| 1.3 创建 Grocery Dude 项目 | 5 |
| 1.4 为现有的应用程序添加 Core Data 支持 | 6 |
| 1.5 小结 | 15 |
| 1.6 习题 | 15 |
| 第 2 章 托管对象模型的基础知识 | 17 |
| 2.1 托管对象模型是什么 | 17 |
| 2.2 添加托管对象模型 | 18 |
| 2.3 实体 | 18 |
| 2.4 属性 | 20 |
| 2.5 Integer 16、Integer 32 与 Integer 64 | 21 |
| 2.6 单精度浮点数与双精度浮点数 | 22 |
| 2.7 属性的各种设置选项 | 25 |
| 2.8 创建 NSManagedObject 的子类 | 27 |

| | | |
|--------------|---|-----------|
| 2.9 | Scalar Properties for Primitive Data Types 选项 | 28 |
| 2.10 | 代码片段: demo 方法 | 29 |
| 2.11 | 创建托管对象 | 29 |
| 2.12 | 后端 SQL 的可见性 | 31 |
| 2.13 | 获取托管对象 | 34 |
| 2.14 | 删除托管对象 | 39 |
| 2.15 | 小结 | 40 |
| 2.16 | 习题 | 40 |
| 第 3 章 | 托管对象模型的迁移 | 42 |
| 3.1 | 修改托管对象模型 | 42 |
| 3.2 | 添加模型版本 | 43 |
| 3.3 | 轻量级的迁移方式 | 45 |
| 3.4 | 默认的迁移方式 | 48 |
| 3.5 | 通过迁移管理器来迁移数据 | 52 |
| 3.6 | 小结 | 62 |
| 3.7 | 习题 | 63 |
| 第 4 章 | 托管对象模型的扩展 | 64 |
| 4.1 | 关系 | 64 |
| 4.2 | Delete 规则 | 69 |
| 4.3 | 数据验证错误 | 73 |
| 4.4 | 实体继承 | 77 |
| 4.5 | 小结 | 81 |
| 4.6 | 习题 | 81 |
| 第 5 章 | 表格视图 | 82 |
| 5.1 | 表格视图基础 | 82 |
| 5.2 | 由 Core Data 所驱动的表格视图 | 83 |
| 5.3 | 创建 CoreDataTVC | 84 |
| 5.4 | DELEGATE: NSFetchedResultsController | 88 |
| 5.5 | AppDelegate 的 CoreDataHelper 实例 | 93 |

| | |
|--|------------|
| 5.6 创建 PrepareTVC | 94 |
| 5.7 创建 ShopTVC | 105 |
| 5.8 小结 | 110 |
| 5.9 习题 | 110 |
| 第 6 章 视图 | 111 |
| 6.1 概述 | 111 |
| 6.2 范例程序所需的视图层级 | 112 |
| 6.3 创建 ItemVC | 113 |
| 6.4 DELEGATE: UITextField | 121 |
| 6.5 货品的计量单位、在家中的位置以及在商店中的位置 | 127 |
| 6.6 小结 | 141 |
| 6.7 习题 | 141 |
| 第 7 章 选取器视图 | 143 |
| 7.1 概述 | 143 |
| 7.2 创建 CoreDataPickerTF | 144 |
| 7.3 DELEGATE+DATASOURCE:UIPickerView | 146 |
| 7.4 创建 UnitPickerTF | 151 |
| 7.5 创建 LocationAtHomePickerTF | 158 |
| 7.6 创建 LocationAtShopPickerTF | 160 |
| 7.7 使选取器不遮住文本框 | 167 |
| 7.8 小结 | 170 |
| 7.9 习题 | 170 |
| 第 8 章 预先加载数据 | 171 |
| 8.1 默认的数据 | 171 |
| 8.2 判断应用程序是否需要导入数据 | 172 |
| 8.3 从 XML 中导入数据 | 174 |
| 8.4 创建导入默认数据所需的上下文 | 177 |
| 8.5 防止重复导入默认数据 | 178 |

| | | |
|---------------|------------------------|------------|
| 8.6 | 触发导入默认数据的操作 | 178 |
| 8.7 | 创建 CoreDataImporter | 180 |
| 8.8 | 选定各实体的 Unique 属性 | 185 |
| 8.9 | 把 XML 中的数据映射到实体的属性 | 186 |
| 8.10 | 从持久化存储区中导入数据 | 189 |
| 8.11 | 小结 | 192 |
| 8.12 | 习题 | 193 |
| 第 9 章 | 深拷贝 | 194 |
| 9.1 | 概述 | 194 |
| 9.2 | 配置拷贝源数据所用的 Core Data 栈 | 197 |
| 9.3 | 增强 CoreDataImporter 类 | 200 |
| 9.4 | 触发深拷贝 | 210 |
| 9.5 | 小结 | 214 |
| 9.6 | 习题 | 214 |
| 第 10 章 | 性能 | 216 |
| 10.1 | 发现性能问题 | 216 |
| 10.2 | 实现拍照功能 | 217 |
| 10.3 | 生成测试数据 | 222 |
| 10.4 | 用 SQLDebug 测量性能 | 225 |
| 10.5 | 用 Instruments 测量性能 | 227 |
| 10.6 | 改善程序性能 | 229 |
| 10.7 | 清理 | 237 |
| 10.8 | 小结 | 237 |
| 10.9 | 习题 | 237 |
| 第 11 章 | 后台处理 | 239 |
| 11.1 | 后台保存 | 239 |
| 11.2 | 后台处理 | 244 |
| 11.3 | 建立 Faulter 类 | 245 |
| 11.4 | 建立 Thumbnailer 类 | 249 |

| | |
|--|------------|
| 11.5 小结..... | 254 |
| 11.6 习题..... | 254 |
| 第 12 章 搜索 | 256 |
| 12.1 修改 CoreDataTVC 类..... | 257 |
| 12.2 修改 PrepareTVC 类..... | 264 |
| 12.3 小结..... | 268 |
| 12.4 习题..... | 268 |
| 第 13 章 与 Dropbox 相结合的备份与恢复 | 270 |
| 13.1 与 Dropbox 相集成..... | 271 |
| 13.2 在 CoreDataHelper 类中准备相关代码..... | 279 |
| 13.3 构建 DropboxHelper 类..... | 280 |
| 13.4 构建 DropboxTVC 类..... | 287 |
| 13.5 小结..... | 297 |
| 13.6 习题..... | 298 |
| 第 14 章 iCloud | 300 |
| 14.1 概述..... | 300 |
| 14.2 启用 iCloud..... | 302 |
| 14.3 为 CoreDataHelper 类添加 iCloud 功能..... | 303 |
| 14.4 Debug Navigator..... | 310 |
| 14.5 禁用 iCloud..... | 310 |
| 14.6 小结..... | 316 |
| 14.7 习题..... | 316 |
| 第 15 章 iCloud 高级使用技巧 | 318 |
| 15.1 去除重复数据..... | 318 |
| 15.2 散播数据..... | 327 |
| 15.3 打造干净的开发环境..... | 333 |
| 15.4 Core Data 程序的配置..... | 335 |
| 15.5 收尾工作..... | 336 |

| | |
|--|------------|
| 15.6 小结 | 337 |
| 15.7 习题 | 337 |
| 第 16 章 与 Web 服务相集成 | 343 |
| 16.1 StackMob 简介 | 343 |
| 16.2 StackMob SDK | 345 |
| 16.3 创建 StackMob 应用程序 | 346 |
| 16.4 准备托管对象模型 | 347 |
| 16.5 配置 StackMob 客户端 | 349 |
| 16.6 SAVING | 351 |
| 16.7 响应底层数据的变更 | 353 |
| 16.8 自动生成 Schema | 354 |
| 16.9 Schema 的权限 | 356 |
| 16.10 认证 | 358 |
| 16.11 使程序保持响应 | 369 |
| 16.12 小结 | 370 |
| 16.13 习题 | 371 |
| 附录 A 为第 1 章的 Grocery Dude 程序所做的准备工作 | 372 |
| 附录 B 为第 16 章的 Grocery Cloud 程序所做的准备工作 | 377 |



初次尝试 Core Data 应用程序

如果不能把一件事用简单的话说清楚，那就表明你理解得还不够透彻。

——阿尔伯特·爱因斯坦

“体验式学习” (kinesthetic learning) 或者说“从实践中学习” (learning by doing)，是接收并记住信息的绝佳手段。即便对于许多有经验的程序员来说，Core Data 也是个相当棘手的话题，于是，笔者就适时地编写了你手中的这本书，它以实践的方式来讲解 Core Data。本书不会过早地讲解一些比较难懂的话题，本章只是会提到后面章节中的一些概念而已。首先，我们要讲解 Core Data 的入门知识，然后直接演示如何为范例程序添加 Core Data 支持。本书后续章节会依次介绍更为复杂的话题，而这个范例程序也会随之不断扩充。

1.1 Core Data 是什么

Core Data 是个框架，它使得开发者可以把数据当成对象来操作，而不必在乎数据在磁盘中的存储方式。对于 Objective-C 程序员来说，这很有用，因为他们已经可以通过代码非常熟练地操作对象了。由 Core Data 所提供的数据对象叫做托管对象 (managed object)，而 Core Data 本身则位于你的应用程序和持久化存储区 (persistent store) 之间。持久化存储区是个通用的术语，指的是像 SQLite 数据库、XML 文件 (iOS 不支持用 XML 文件作为持久化存储区) 或 Binary store (又名 atomic store) 这种数据文件。由于这些文件在底层硬件重启之后还会保留下来，所以它们是持久的。还有一种持久化存储方式，它的名字非常奇怪，叫做 “In-Memory store”。虽说 In-Memory store 并不是“持久的”，但开发者在用它管理数据时却可以享受 Core Data 的所有优点，诸如变更管理与数据验证等，另外，其效率自然也是相当高的。

为了把数据从托管对象映射到持久化存储区中，Core Data 需要使用托管对象模型，而开发者则可以通过对象图（object graph）来配置应用程序的数据结构。可以把对象图想象成一系列“饼干模型切割刀”（cookie cutter），而托管对象正是用这些切割刀切出来的。对象图里的对象指的是实体，每个实体就好比一把“饼干模型切割刀”，用于制作自定义的托管对象。有了托管对象之后，就可以直接在 Objective-C 里面操作它们，而无需再编写 SQL 代码了（笔者假定你使用 SQLite 作为持久化存储区，因为这是最常用的一种持久化存储方式）。当把数据保存到磁盘的时候，Core Data 显然会把这些托管对象映射回持久化存储区里面。

托管对象持有一份对持久化存储区里相关数据的拷贝。如果用数据库作为持久化存储区，那么托管对象可能对应于数据库里某张数据表中的一行。如果用 XML 文件作为持久化存储区（此方式只有 Mac 系统支持），那么托管对象可能对应于某个数据元素（data element）里面的一份数据。托管对象可以是 `NSManagedObject` 类的实例，但一般情况下，它都是某个 `NSManagedObject` 子类的实例。这个问题将在第 2 章中详细讨论。

所有托管对象都必须位于托管对象上下文（managed object context）里面，而托管对象上下文又位于高速的易失性存储器里面，也就是位于 RAM 中。为什么需要有托管对象上下文呢？原因之一就是磁盘与 RAM 之间传输数据时会有开销。磁盘读写速度比 RAM 慢得多，所以不应该频繁访问它。而有了托管对象上下文之后，对于原来需要读取磁盘才能获取到的数据，现在只需访问这个上下文，就可以非常迅速地获取到了。但它的缺点在于，开发者必须在托管对象上下文上面定期调用 `save:` 方法，以将变更后的数据写回磁盘。托管对象上下文的另一个功能是记录开发者对托管对象所做的修改，以提供完整的撤销与重做支持。



提示 “如果不能把一件事用简单的话说清楚，那就表明你理解得还不够透彻。”这是先贤阿尔伯特·爱因斯坦的一句名言。本书每章均以爱因斯坦的名言开头。Core Data 是个比较难学的技术，但这并不是说我们不能把它分解成多个易于理解的小知识点。笔者在编写技术教程和文档的时候，都会遵照爱因斯坦的教诲，尽量用比较好懂的方式把它们写出来，同时也尽量会把内容写得丰富一些。

图 1-1 直观地描述了 Core Data 的几个主要概念。

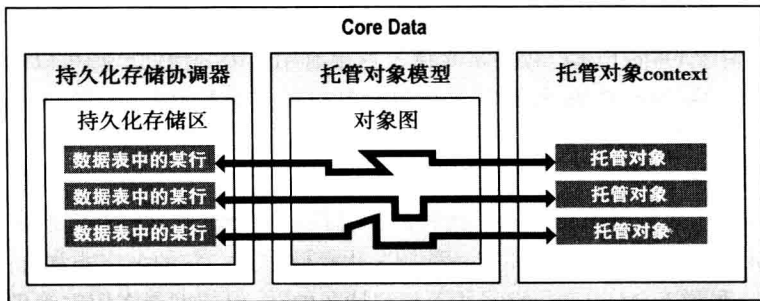


图 1-1 Core Data 概念

1.1.1 持久化存储协调器

图 1-1 左侧的持久化存储协调器 (persistent store coordinator) 里面包含一份持久化存储区, 而存储区里面又含有数据表里的若干行数据。设置持久化存储协调器的时候, 我们通常选用 SQLite 数据库作为持久化存储区。另外, 也可以选用 Binary、XML 或 In-Memory 等形式的持久化存储区。但要注意, Binary 和 XML 格式的存储区是“原子的”(atomic), 也就是说, 即便你只想修改少量数据, 在保存的时候也依然需要把整个文件都写入磁盘。首次将原子的存储区 (atomic store) 读入内存时当然也会有这个问题。如果数据很多, 那么使用这种存储区的问题就比较严重了, 因为它会占据宝贵的内存空间。

与原子存储不同, SQLite 数据库会在用户提交变更日志时进行增量更新, 变更日志也叫做事务日志。由于采用了这种更新方式, 所以 SQLite 数据库的内存占用量相对来说非常小。有鉴于此, 开发者一般都会选用 SQLite 数据库, 尤其在把 Core Data 集成到 iCloud 的时候, 更应该如此。



提示 持久化存储区只应该由 Core Data 来创建。不应该让 Core Data 去使用不是由它所创建的数据库。假如需要使用既有的数据, 那么应该将其导入 Core Data。这个问题放在第 8 章讨论。

同一个持久化存储协调器可以有多个持久化存储区。把 Core Data 与 iCloud 相集成的时候, 就可能会出现这种情况。我们可以把不属于 iCloud 的数据放在一个存储区里, 而把属于 iCloud 的数据放在另一个存储区里。这样既能节省网络带宽, 又能节省 iCloud 存储空间。即便你有两个持久化存储区, 也不意味着必须使用两种对象图。Core Data 的模型配置允许开发者使用多个独立的存储区, 但却采用同一套对象图。在设定 Core Data 的模型配置选项时, 可以指明对象图里的某一部分属于哪个持久化存储区。假如确实想使用多个持久化存储区, 那么就不能为这些存储区之间的数据建立“关系”了。Core Data 的配置问题放在第 15 章讨论。

要想创建持久化存储区, 需生成 `NSPersistentStore` 类的实例; 要想创建持久化存储协调器, 需生成 `NSPersistentStoreCoordinator` 类的实例。

1.1.2 托管对象模型

图 1-1 的中部是托管对象模型, 它位于持久化存储协调器和托管对象上下文之间。顾名思义, 托管对象模型是描述数据结构的模型或图示 (graphical representation), 而托管对象正是以它为基础产生出来的。它与数据库模式 (database schema) 相似, 有时也叫做对象图。要想创建托管对象模型, 可以用 Xcode 来配置实体及实体之间的关系。实体类似于数据库中的数据表模式 (table schema)。实体本身并不包含数据, 它们只是规定了基于该实体的托管对象应该具有何种特性。实体就是刚才提到的那种“饼干模型切割刀”, 正如数据库里的数据表有字段 (field) 一样, 实体也有属性 (attribute)。属性的数据类型

可以是整数（integer）、字符串（string）或日期（date）。第2章与第4章将会详述这些问题。

要想创建托管对象模型，需生成 `NSManagedObjectModel` 类的实例。

1.1.3 托管对象上下文

图1-1的右侧是托管对象上下文，其中包含多个托管对象。托管对象上下文负责管理其中对象的生命期（lifecycle），并且负责提供许多强大的功能，诸如 faulting、变更追踪（change tracking）、验证（validation）等。所谓 faulting，意思就是用户从持久化存储区中获取数据时，系统只会把需要用到的一部分获取过来。第10章将详细讨论 faulting。变更追踪用于支持撤销及重做功能。验证机制用来确保由托管对象模型所订立的规则。比方说，可以针对实体的单个属性来限定其最小值或最大值，这将在第2章中讨论。

持久化存储区可以有很多个，与之类似，托管对象上下文也可以不止一个。有时我们需要在后台处理任务（比方说把数据保存到磁盘或导入数据），这种情况下可以采用多个上下文。假如在前台上下文（foreground context）上面调用 `save:`，那么用户界面就可能会有“卡顿”（lag）现象，尤其当数据变化较大的时候更是如此。要想避免这个问题，有个简单的办法就是只在用户按下手机 Home 键时才去调用 `save:`，这时应用程序会转入后台。还有个稍微复杂但却更加灵活的办法，就是采用两个托管对象上下文。请记住，托管对象上下文是存放在高速内存里面的。你可以配置其中一个上下文，令其把数据保存到另一个上下文里。一旦把前台上下文中的数据保存到后台上下文，那么就可将后台上下文中的数据异步地（asynchronously）存入磁盘。这种分段式的做法可以确保磁盘写入操作不会影响用户界面的流畅度。

从 iOS 5 开始就可以配置多个上下文之间的上下级关系了。子上下文会将它的父上下文视为持久化存储区，而这个父上下文实际上是用来处理各项繁重操作的（例如在后台保存数据等）。这个问题放在第11章深入讨论。

要想创建托管对象上下文，需生成 `NSManagedObjectContext` 类的实例。

1.2 Core Data 的适用场合

如果应用程序要保存的设置数据太多，以致 `NSUserDefaults` 及“特性列表”（property list）这种简单的存储方案无法应付，那么就会出现内存占用量方面的问题。解决办法是直接使用数据库或通过 Core Data 来间接操作数据库。选用 Core Data 的好处是，不用再花时间编写数据库接口的代码了。此外，你还将享受性能方面的优势，而且可以使用诸如撤销及验证等强大的功能。假如选择直接使用数据库，那就要花时间去开发及测试工作，也就是通常所说的“重新发明轮子”（reinventing the wheel），而使用 Core Data 则无须操心这些事情，开发者可以把精力放在应用程序中更为重要的事情上面。

你可能在想：我只是要把一些数据保存到磁盘中而已，用得着这么麻烦吗？其实只要理解了 Core Data 的几个要点，你就会发现这一点都不麻烦。笔者相信你自己确实能够写出一套数据库接口，而且这套接口在短时间内的效果也许还很好。但是，当需求有了变化，或有了新需求时，比方说，现在要支持多个设备之间的数据同步功能，该怎么办呢？在不影响用户界面的前提下，你有没有把握写出多线程环境里的数据导入例程呢？你能不能写出既支持撤销与验证功能，又能在老式 iPhone 上面高速运行，而且内存占用量还很小的程序呢？

对于上面这些工作，其实 Core Data 框架早就做好了，而且已经测试过了。即便你的应用程序所要处理的数据量特别少，也依然值得使用 Core Data，因为这样可以令应用程序能够适应将来的需求变化，同时又不会影响到性能。

一旦用上 Core Data，你就会见识到它的健壮与流畅程度。每天都有很多人在使用集成了 Core Data 的应用程序，而这也使得 Core Data 的各项功能日趋成熟，同时其性能也令人满意。简言之：要是武断地抛弃 Core Data 不用，那就要自己编写数据库接口；反之，若学会了 Core Data，则能节省大量开发时间，而且还能自动享受到由 Core Data 所提供的许多附加功能。



提示 在继续往下阅读之前，请你先确认 Mac 中安装的 Xcode 版本不低于 5。本书范例代码是针对 iOS 7 编写的，所以无法在低版本的 Xcode 里面使用。另外，笔者也建议你注册成为“iOS Developer Program”的会员，这样你可以根据需要在相关设备上运行范例程序了。入会详情可参阅：<http://developer.apple.com>。

1.3 创建 Grocery Dude 项目

Grocery Dude 是个运行在 iPhone 上的范例程序，在学习本书的过程中，你将了解到它的制作流程。学会了 Core Data 中的某个特性或某项开发技巧之后，你可以将其运用在 Grocery Dude 程序上面。到了本书收尾的时候，你将会制作好一款功能完备而且运行速度很快的 Code Data 程序，它能够同 iCloud 紧密地集成在一起。假如你现在就想直接看看成品，那可以去 App Store 下载 Grocery Dude。请注意，Grocery Dude 是专门为 iPhone 编写的。无论要把数据显示在多大的屏幕上，Core Data 的使用理念都是一致的。好了，言归正传，我们现在就来编写这款程序！

站在冰箱、储藏室、碗柜前面，或待在家里其他地方的时候，你有没有觉得自己忘了买什么东西？到了商店之后，是不是又忘了要买的东西放在哪排货架上？更麻烦的是，当你走入第 8 条过道（aisle）时，发现要买的东西在第 2 条过道旁的货架上，于是绕了半天跑过去，等拿了那件货品之后，却又发现下一个要买的东西竟然就在第 8 条过道附近，于是又得折回来！

有了 Grocery Dude 之后，就不用担心这些问题了：

- ❑ 它可以分类显示家里各个位置的东西，以此提示你该购买哪些生活用品了。
- ❑ 在超市购物时，它可以告诉你某件货品摆在哪条过道旁的货架上。
- ❑ 它可以待买物品按照过道编组，这样的话，每条过道只需走一遍，即可拿完所需采购的货品。
- ❑ 它可以通过 iCloud 在各设备之间同步数据。
- ❑ 它还能帮你学习 Core Data ！



提示 附录 A 按步骤讲解了怎样从头开始创建本书的主项目 Grocery Dude。你可以自己照着做，也可以直接从 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterAppendixA.zip> 把这个起始项目下载下来。下载好之后，需要用 Xcode 5 或更高版本的 Xcode 打开它。

1.4 为现有的应用程序添加 Core Data 支持

在 Xcode 中创建 iOS 应用程序项目时，可以使用各种起始模板（starting-point template）。假如要根据 Master-Detail、Utility Application 或 Empty Application 等模板来创建项目，那么只需勾选 **Use Core Data**，即可在项目中使用 Core Data。不过，Grocery Dude 项目是根据 **Single View Application** 模板创建的，它起初并没有包含 Core Data，笔者想通过这种方式来演示如何手工为项目添加 Core Data 支持，这样做更有意义。为了使用 Core Data Framework，我们需要将它与项目相链接。

请按如下步骤修改 Grocery Dude，以便将该项目与 Core Data Framework 相链接：

1. 如图 1-2 所示，选中 **Grocery Dude Target**。
2. 在 **General** 分页中，点击 **Linked Frameworks and Libraries** 区域中的“+”按钮，然后把项目链接到 **CoreData.framework**，如图 1-2 所示。

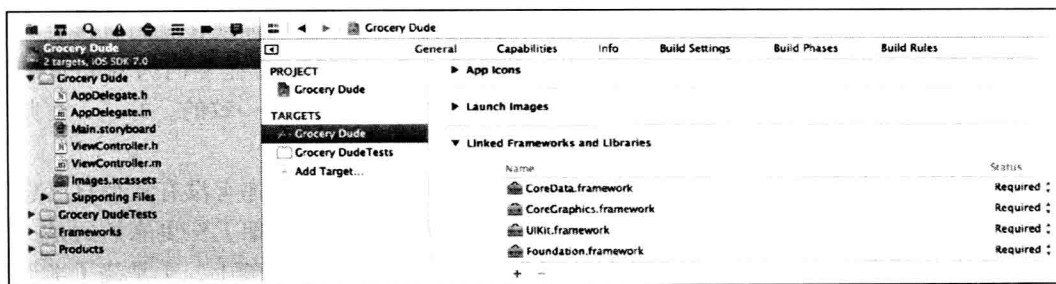


图 1-2 将项目与 Core Data Framework 相链接

1.4.1 Core Data Helper 简介

试着查看一下支持 Core Data 的那些内置模板，你也许会发现：Core Data 是在应用程序委托（application delegate）里面设置的。本书所用的办法是通过辅助类来设置 Core Data，这样的话，你就可以将这种办法运用到自己的项目中了。这也使得 Core Data 组件变得模块化，而且易于移植。我们将通过应用程序委托来惰性地（lazily）创建 CoreDataHelper 类的实例。这种实例可用来完成下列事项：

- ❑ 初始化托管对象模型
- ❑ 根据托管对象模型来创建持久化存储区，并据此初始化持久化存储协调器
- ❑ 根据持久化存储协调器来初始化托管对象上下文

请根据下列步骤修改 Grocery Dude，以便在新的 Xcode 组（group）里面创建 CoreDataHelper 类：

1. 右击 Xcode 中的 **Grocery Dude** 组，然后创建名为 **Generic Core Data Classes** 的新组，如图 1-3 所示。

2. 选定刚创建好的 **Generic Core Data Classes** 组。

3. 点击 **File > New > File...**。

4. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。

5. 将 **Subclass of** 设为 **NSObject**，并将 **Class** 的名称改为 **CoreDataHelper**，然后点击 **Next** 按钮。

6. 在 **Targets** 中勾选“**Grocery Dude**”，然后点击 **Create** 按钮，这样就会在 Grocery Dude 项目的目录中创建 CoreDataHelper 类了。

程序清单 1-1 列出了需要添加到 CoreDataHelper 头文件中的新代码。

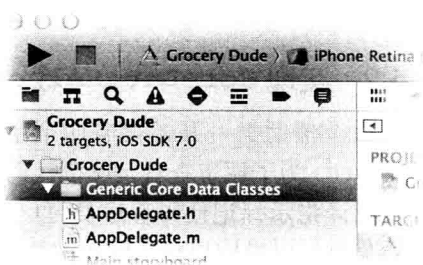


图 1-3 在 Xcode 中创建名为 Generic Core Data Classes 的组

程序清单 1-1 CoreDataHelper.h

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface CoreDataHelper : NSObject

@property (nonatomic, readonly) NSManagedObjectContext *context;
@property (nonatomic, readonly) NSManagedObjectModel *model;
@property (nonatomic, readonly) NSPersistentStoreCoordinator *coordinator;
@property (nonatomic, readonly) NSPersistentStore *store;

- (void)setupCoreData;
- (void)saveContext;
@end
```

作为 Objective-C 程序员，你应该已经熟悉头文件（也就是 .h 文件）的用途了。CoreData-

Helper.h 用来声明 CoreDataHelper 中的 context、model、coordinator 及 store 等属性。当应用程序委托创建 CoreDataHelper 实例的时候，系统会调用 setupCoreData 方法。若要把托管对象上下文里发生的变更保存到持久化存储区，则可调用 saveContext 方法。假如要写入磁盘的数据比较多，那么该方法会导致用户界面卡顿。第 11 章将给程序添加后台保存功能，而在这之前，笔者建议你只从 AppDelegate.m 的 applicationDidEnterBackground 及 applicationWillTerminate 方法里面调用它。

请按照下述步骤修改 Grocery Dude，以配置 CoreDataHelper 的头文件：

1. 用程序清单 1-1 中的代码把 CoreDataHelper.h 文件里现有的全部代码都替换掉。假如现在查看 CoreDataHelper.m 文件，那么 Xcode 会提示你 setupCoreData 及 saveContext 方法还没实现，不过目前这无关紧要。

1.4.2 实现 CoreDataHelper 类

这个辅助类一开始会有四个主要的部分，它们分别叫做 FILES、PATHS、SETUP 及 SAVING。为了便于查看及阅读代码，笔者用编译指示标记（pragma mark）将这些区域分隔开。编译指示标记使得程序员可以按照逻辑来组织源码，而且 Xcode 还会据此生成一份漂亮的菜单，以供用户在不同的部分中浏览。

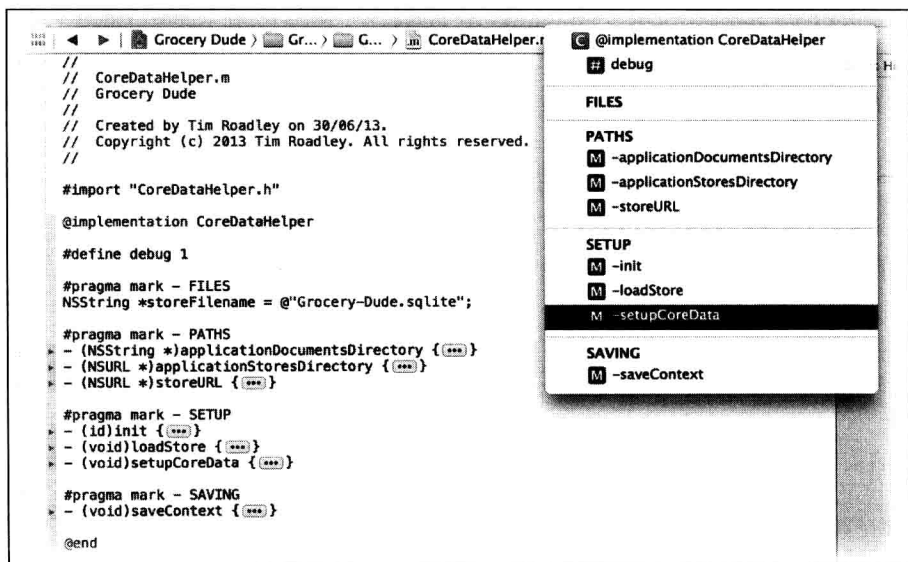


图 1-4 由编译指示标记所生成的菜单

1.4.3 FILES 部分

CoreDataHelper.m 的 FILES 部分中会有一个 NSString，用于存储持久化存储区的文件名。如果稍后还要添加其他持久化存储区，那么也应该在这里写下它们的文件名。

程序清单 1-2 中的代码包含一条 #define 语句，Grocery Dude 中的绝大多数类都要使用这行语句，它是为了协助调试工作而设的。如果 debug 是 1，那么该类的 debug logging（调试日志记录）功能就会开启。本书大部分的 NSLog 命令都包裹在 if (debug==1) 语句里，所以只有开启了调试功能之后，那些命令才会有效果。

程序清单1-2 CoreDataHelper.m文件的FILES部分

```
#define debug 1

#pragma mark - FILES
NSString *storeFilename = @"Grocery-Dude.sqlite";
```

请按下述步骤修改 Grocery Dude，以便添加 FILES 部分：

1. 把程序清单 1-2 中的代码添加到 CoreDataHelper.m 文件底部，但是要将其置于 @end 语句之前。

1.4.4 PATHS

为了把数据以持久化的形式写入磁盘，Core Data 需要知道持久化存储文件在文件系统中的位置。我们可以分别编写 3 个方法来向 Core Data 提供这一信息。程序清单 1-3 列出了第一个方法，也就是 applicationDocumentsDirectory 方法，该方法返回 NSString，而这个 NSString 就代表应用程序文档目录的路径。你会注意到我们这是首次把调试代码包裹在 if (debug==1) 语句里面，这种调试代码用于显示当前运行的究竟是哪个方法。NSLog 语句很适合用来打印应用程序里各个方法的执行顺序，而这对调试工作很有帮助。

程序清单1-3 CoreDataHelper.m文件的PATHS部分

```
#pragma mark - PATHS
- (NSString *)applicationDocumentsDirectory {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }
    return [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES)
        lastObject];
}
```

请按下列步骤修改 Grocery Dude，以便添加 PATHS 部分：

1. 将程序清单 1-3 中的代码添加到 CoreDataHelper.m 文件尾部，并放在 @end 语句之前。

接下来这个方法叫做 applicationStoresDirectory，它会向应用程序文档目录中添加名为 Stores 的子目录，并且将其路径放在 NSURL 中返回。若 Stores 目录尚未建立，则程序清单 1-4 中的相关代码会将其创建出来。

程序清单1-4 CoreDataHelper.m文件中的applicationStoresDirectory方法

```

- (NSURL *)applicationStoresDirectory {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}

NSURL *storesDirectory =
[[NSURL fileURLWithPath:[self applicationDocumentsDirectory]
    URLByAppendingPathComponent:@"Stores"];

NSFileManager *fileManager = [NSFileManager defaultManager];
if (![fileManager fileExistsAtPath:[storesDirectory path]]) {
    NSError *error = nil;
    if ([fileManager createDirectoryAtURL:storesDirectory
        withIntermediateDirectories:YES
        attributes:nil
        error:&error]) {
        if (debug==1) {
            NSLog(@"Successfully created Stores directory");
        }
        else {NSLog(@"FAILED to create Stores directory: %@", error);}
    }
    return storesDirectory;
}
}

```

请按下列步骤修改 Grocery Dude，以便将 applicationStoresDirectory 方法添加到 PATHS 部分中：

1. 将程序清单 1-4 中的代码添加到 CoreDataHelper.m 文件尾部，并放在 @end 语句之前。

最后一个方法如程序清单 1-5 所示，它只是把持久化存储区的文件名添加到 Stores 目录的路径中。调用完该方法之后，我们就可以知道持久化存储文件的完整路径了。

程序清单1-5 CoreDataHelper.m文件中的storeURL方法

```

- (NSURL *)storeURL {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}
return [[self applicationStoresDirectory]
    URLByAppendingPathComponent:storeFilename];
}

```

请按下列步骤修改 Grocery Dude，以便将 storeURL 方法添加到 PATHS 部分中：

将程序清单 1-5 中的代码添加到 CoreDataHelper.m 文件尾部，并放在 @end 语句之前。

1.4.5 SETUP

处理完文件和路径之后，现在该实现初始化 Core Data 所用的三个方法了。程序清单 1-6 列出了第一个方法，它的名字叫做 `init`，程序在创建 `CoreDataHelper` 实例的时候，会自动运行该方法。

程序清单1-6 CoreDataHelper.m文件的SETUP部分

```
#pragma mark - SETUP
- (id)init {
    if (debug==1) {
        NSLog(@"Running %@", NSStringFromClass(_cmd));
    }

    self = [super init];
    if (!self) {return nil;}

    _model = [NSManagedObjectModel mergedModelFromBundles:nil];
    _coordinator = [[NSPersistentStoreCoordinator alloc]
                    initWithManagedObjectModel:_model];
    _context = [[NSManagedObjectContext alloc]
                initWithConcurrencyType:NSMainQueueConcurrencyType];
    [_context setPersistentStoreCoordinator:_coordinator];
    return self;
}
```

`_model` 实例变量指向 `NSManagedObjectModel` 对象。这个对象是我们在 `NSManagedObjectModel` 类上以 `nil` 为参数调用 `mergedModelFromBundles` 方法而得来的，该方法会用 `main bundle` 中的全部数据模型文件（`data model file`，也就是对象图）来初始化此对象。目前项目里还没有模型文件，但是第 2 章就会加进来一个。如果有多个模型需要合并，那么可以把元素类型为 `NSBundle` 的 `NSArray` 数组传给 `mergedModelFromBundles`，但一般来说不用担心这个问题。



提示 还有一种办法也能初始化托管对象模型，就是明确写出需要使用的模型文件。与直接合并 `bundle` 相比，这种写法的代码量多了一倍。可以用这条语句来手工指定模型：`_model=[[NSManagedObjectModel alloc] initWithContentsOfURL:[NSBundle mainBundle] URLForResource:@"Model" withExtension:@"momd"]];`。

`_coordinator` 实例变量指向 `NSPersistentStoreCoordinator` 对象。刚才我们曾经创建了托管对象模型，并令 `_model` 指向该模型，而现在我们就根据这个 `_model` 指针来初始化 `NSPersistentStoreCoordinator`。持久化存储协调器里面目前还没有持久化存储文件，这些文件稍后将由 `setupCoreData` 方法加入。

`_context` 实例变量指向 `NSManagedObjectContext` 对象。在初始化该对象的时候，我们把并发类型设为 `NSMainQueueConcurrencyType`，意思是令这个上下文在主

线程队列中运行。凡是要编写数据驱动型的用户界面，就需要将上下文放在主线程中。把上下文初始化好之后，我们用刚才那个指向 `NSPersistentStoreCoordinator` 的 `_coordinator` 指针来配置它。第 8 章将会演示如何使用多个托管对象上下文，而且还会介绍与后台运行相对应的 `NSPrivateQueueConcurrencyType`，不过目前我们把上下文放在主线程里就可以了。

请按下列步骤修改 `Grocery Dude`，以便添加 `SETUP` 部分：

1. 将程序清单 1-6 中的代码添加到 `CoreDataHelper.m` 文件底部，并放在 `@end` 语句之前。

`SETUP` 部分里还需要另外一个方法——`loadStore` 方法，其代码如程序清单 1-7 所示。

程序清单1-7 CoreDataHelper.m文件中的loadStore方法

```
- (void)loadStore {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }

    if (_store) {return;} // Don't load store if it's already loaded
    NSError *error = nil;
    _store = [_coordinator addPersistentStoreWithType:NSSQLiteStoreType
                                                configuration:nil
                                                URL:[self storeURL]
                                                options:nil error:&error];
    if (!_store) {NSLog(@"Failed to add store. Error: %@", error);abort();}
    else        {if (debug==1) {NSLog(@"Successfully added store: %@", _store);}}
}
```

`loadStore` 方法很简单。首先判断 `_store` 是不是已经加载好了，如果还没有，那就声明一个指向 `NSError` 实例的 `error` 指针，并将其当前值设为 `nil`。稍后在设置 `_store` 实例变量的时候，我们会用该指针来捕获设置过程中所发生的错误。假如在执行完配置代码之后，`_store` 是 `nil`，那就表明配置操作失败了，我们把发生的错误及其内容记录到控制台。

通过 `addPersistentStoreWithType` 方法将 `SQLite` 持久化存储区添加到 `_coordinator` 之后，`_store` 变量的值就是指向这个持久化存储区的指针。而调用 `addPersistentStoreWithType` 方法时所使用的 `storeURL` 参数则是由早前创建的 `storeURL` 方法所返回的。

请按下列步骤修改 `Grocery Dude`，以便将 `loadStore` 方法添加到 `SETUP` 部分中：

1. 把程序清单 1-7 中的代码添加到 `CoreDataHelper.m` 文件底部，并放在 `@end` 语句之前。

最后我们来创建 `setUpCoreData` 方法。由于其他的辅助方法已经就位，所以这个方法写起来非常简单。程序清单 1-8 列出了这个新方法的代码，目前它只需调用 `loadStore` 即可。本书稍后将为程序加入更多的功能，到那时，该方法也会随之扩充。

程序清单1-8 CoreDataHelper.m文件中的setupCoreData方法

```

- (void)setupCoreData {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}
    [self loadStore];
}

```

请按下列步骤修改 Grocery Dude，以便将 setupCoreData 方法添加到 SETUP 部分中：

1. 把程序清单 1-8 中的代码添加到 CoreDataHelper.m 文件底部，并放在 @end 语句之前。

1.4.6 SAVING

接下来我们要实现的这个方法可以把 _context 中所发生的变更保存到 _store 里。具体做法很简单：只需像程序清单 1-9 这样，给上下文发送 save: 消息即可。我们将新建名为 SAVING 的部分，并把 saveContext 方法置于其中。

程序清单1-9 CoreDataHelper.m文件的SAVING部分

```

#pragma mark - SAVING
- (void)saveContext {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}
    if ([_context hasChanges]) {
        NSError *error = nil;
        if ([_context save:&error]) {
            NSLog(@"_context SAVED changes to persistent store");
        } else {
            NSLog(@"Failed to save _context: %@", error);
        }
    } else {
        NSLog(@"SKIPPED _context save, there are no changes!");
    }
}

```

请按下列步骤修改 Grocery Dude，以便添加 SAVING 部分：

1. 把程序清单 1-9 中的代码添加到 CoreDataHelper.m 文件底部，并放在 @end 语句之前。

现在我们应该来试用一下这个 CoreDataHelper 了！为了使用该类，首先需要在应用程序委托的头文件（header）中添加新的属性。另外，还需引入 CoreDataHelper，使头文件能够知道我们刚写好的这个类。程序清单 1-10 用粗体标出了应用程序委托的头文件中所需修改的代码。

程序清单1-10 AppDelegate.h

```
#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
@interface AppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong, readonly) CoreDataHelper *coreDataHelper;
@end
```

请按下列步骤修改 Grocery Dude，以便将 CoreDataHelper 添加到应用程序委托中：

1. 用程序清单 1-10 中的代码替换 AppDelegate.h 的原有内容。

下一步是修改应用程序委托的实现文件，把名为 cdh 的小方法放入其中，该方法会返回“非 nil”的 CoreDataHelper 实例。另外，为了便于调试，我们还添加了一行“#define debug 1”语句，如程序清单 1-11 所示。

程序清单1-11 CoreDataHelper.m文件中的cdh方法

```
#define debug 1

- (CoreDataHelper*)cdh {
if (debug==1) {
    NSLog(@"Running %@ '%@'", self.class, NSStringFromSelector(_cmd));
}
    if (!_coreDataHelper) {
        _coreDataHelper = [CoreDataHelper new];
        [_coreDataHelper setupCoreData];
    }
    return _coreDataHelper;
}
```

请按下列步骤修改 Grocery Dude，以便将 cdh 方法添加到应用程序委托之中：

1. 把程序清单 1-11 里的代码添加到 AppDelegate.m 文件中的“@implementation AppDelegate”这一行下面。

本小节的最后一步需要确保当应用程序转入后台或终止时，上下文里面的内容能够得以保存。这是个将数据写入磁盘的好机会——此时写入数据，不会导致用户界面反应迟缓，因为用户界面本身已经隐藏起来了。程序清单 1-12 列出了与上下文的保存操作有关的代码。

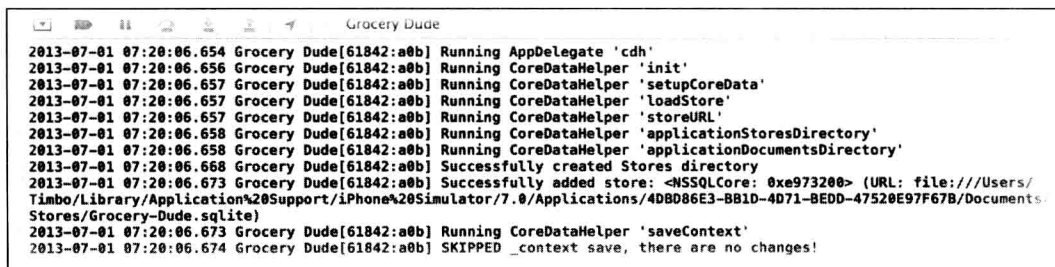
程序清单1-12 AppDelegate.m文件中的applicationDidEnterBackground方法

```
- (void)applicationDidEnterBackground:(UIApplication *)application {
    [[self cdh] saveContext];
}
- (void)applicationWillTerminate:(UIApplication *)application {
    [[self cdh] saveContext];
}
```

请按下列步骤修改 Grocery Dude，以确保应用程序在转入后台或终止时上下文中的数据能够得以保存：

1. 把 `[[self cdh] saveContext];` 这行语句添加到 AppDelegate.m 文件的 `applicationDidEnterBackground` 方法里。
2. 把 `[[self cdh] saveContext];` 这行语句添加到 AppDelegate.m 文件的 `applicationWillTerminate` 方法里。

在 iOS 仿真器中运行 Grocery Dude 程序，然后按下 home 键（可以通过“**Shift + ⌘ + H**”组合键或 iOS 仿真器的 **Hardware > Home** 菜单项来模拟“按下 home 键”这一操作），并同时查看调试日志窗口。只有真正用到 Core Data 的时候，应用程序才会通过应用程序委托的 `cdh` 方法设置它，于是，在刚开始运行程序的时候，日志里面并没有内容。首次使用 Core Data 是在调用 `save:` 的时候，也就是应用程序转入后台的时候。本书在后续章节中还会不断地完善这个应用程序，而 `cdh` 方法的调用时机也会有所提前。图 1-5 演示了按下 Home 键之后各方法的执行顺序。



```

Grocery Dude
2013-07-01 07:20:06.654 Grocery Dude[61842:a0b] Running AppDelegate 'cdh'
2013-07-01 07:20:06.656 Grocery Dude[61842:a0b] Running CoreDataHelper 'init'
2013-07-01 07:20:06.657 Grocery Dude[61842:a0b] Running CoreDataHelper 'setupCoreData'
2013-07-01 07:20:06.657 Grocery Dude[61842:a0b] Running CoreDataHelper 'loadStore'
2013-07-01 07:20:06.657 Grocery Dude[61842:a0b] Running CoreDataHelper 'storeURL'
2013-07-01 07:20:06.658 Grocery Dude[61842:a0b] Running CoreDataHelper 'applicationStoresDirectory'
2013-07-01 07:20:06.658 Grocery Dude[61842:a0b] Running CoreDataHelper 'applicationDocumentsDirectory'
2013-07-01 07:20:06.668 Grocery Dude[61842:a0b] Successfully created Stores directory
2013-07-01 07:20:06.673 Grocery Dude[61842:a0b] Successfully added store: <NSSQLCore: 0xe973200> (URL: file:///Users/Timbo/Library/Application%20Support/iPhone%20Simulator/7.0/Applications/4DB086E3-B81D-4D71-BE0D-47520E97F67B/Documents.Stores/Grocery-Dude.sqlite)
2013-07-01 07:20:06.673 Grocery Dude[61842:a0b] Running CoreDataHelper 'saveContext'
2013-07-01 07:20:06.674 Grocery Dude[61842:a0b] SKIPPED _context save, there are no changes!
  
```

图 1-5 显示在调试日志窗口中的方法的执行顺序

1.5 小结

本章介绍了 Core Data 的关键组件。Grocery Dude 范例程序包含了 SQLite 格式的持久化存储区、持久化存储协调器、托管对象模型以及托管对象上下文。但由于我们没有配置数据模型，所以目前的应用程序还不怎么好玩。第 2 章将会介绍数据模型，那时这个程序就会变得比较有意思了。即便你现在还不理解 Core Data 中某些部件的用途，也无需担心，等把每个组件都用熟了之后，自然就能明白它们是如何组织起来的了。

1.6 习题

你可以试着在已经构建好的 Grocery Dude 项目上面做试验：

1. 修改应用程序委托中每个方法的代码，将下列语句添加到代码顶部，以利于调试：

```
if (debug==1) {  
    NSLog(@"Running %@ '%@'", self.class, NSStringFromSelector(_cmd));  
}
```

2. 在真实设备与 iOS 仿真器中分别运行应用程序，然后对比日志中打印出来的持久化存储区文件的路径有何不同。如果想在排解程序故障的时候打开持久化存储区，那么这一路径信息将会很有用处。

3. 修改 CoreDataHelper.m 文件中的 loadStore 方法，将持久化存储区的类型从 NSSQLStoreType 改为 NSXMLStoreType，然后试着运行程序。你会发现程序无法运行，因为 iOS 系统并不支持这种存储格式。



托管对象模型的基础知识

知识只能得自经验。

——阿尔伯特·爱因斯坦

第1章把 Core Data 中的一些基本内容添加到了 Grocery Dude 范例程序里面。现在我们已经配置好持久化存储区、持久化存储协调器、托管对象模型以及托管对象上下文了，但是对象图还空着。这就好比已经把做饼干的全部材料都准备好了，但却发现还没有饼干模型切割刀一样！本章将要讲解托管对象模型的基础知识，并告诉你配置范例程序对象图的全过程。

2.1 托管对象模型是什么

托管对象模型是一种数据结构。数据结构、纲要、对象图、数据模型、托管对象模型这些术语其实可以互换，因为它们的意思差不多。比方说，你要重新设计一个不使用 Core Data 的数据库，那么可能会配置一套数据库模式，并把它称作数据模型。而 Core Data 关注的是（托管）对象。于是，我们不把这个模式称为数据模型，而是把它叫做托管对象模型。尽管笔者采用的是这套称呼，但你完全可以把它叫做对象模型、对象图、模式或数据结构。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter01.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使

用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项，这样可以清除掉同名项目所残留的缓存。

2.2 添加托管对象模型

在第 1 章中，我们通过 CoreDataHelper.m 文件里的 mergedModelFromBundles 方法初始化了托管对象模型。然而现在的问题是：项目里根本就没有模型可用！如果连模型都没有的话，那 Core Data 就彻底失去意义了，所以，我们这个时候应该创建模型文件。模型文件一般会含有“对象图”，而对象图则用来表示应用程序的数据结构以及其他一些可以简化应用程序开发的东西，我们稍后再来解释。

请按下列步骤修改 Grocery Dude，以便添加数据模型文件：

1. 在现有的 **Grocery Dude** 组上点击鼠标右键，然后选择 **New Group** 菜单项。
2. 将新的组取名为 **Data Model**。
3. 选中 **Data Model** 组。
4. 点击 **File > New > File...** 菜单项。
5. 选择 **iOS > Core Data > Data Model**，然后点击 **Next** 按钮。
6. 确保 Targets 中的“Grocery Dude”处于勾选状态，并保持默认的文件名 **Model** 不变，然后点击 **Create** 按钮。
7. 选定 **Model.xcdatamodeld**，如图 2-1 所示。

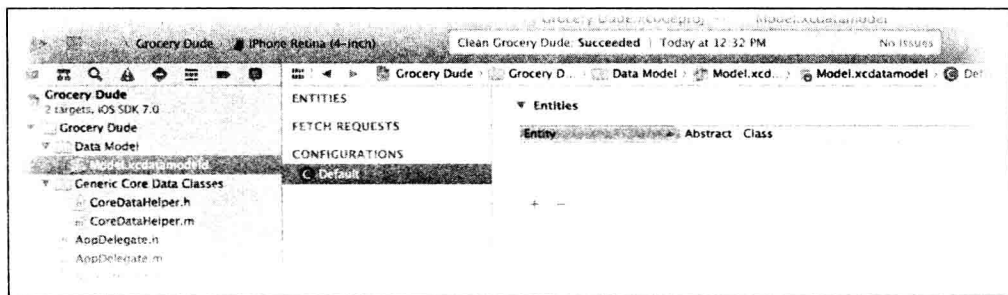


图 2-1 Xcode 的 Data Model Designer 界面

图 2-1 就是 Xcode 的 Data Model Designer 界面，该界面用于配置数据模型。初次看到该界面时，对有些内容可能会觉得比较陌生，而且你可能还想知道它们的含义。本章稍后会讨论界面中的实体（Entities）与 Fetch Request（获取请求），而配置（Configurations）则会放在第 15 章讨论。

2.3 实体

托管对象模型由一系列实体描述对象构成，这种对象就叫做实体。实体就好比一把饼

干模型切割刀，用于创建托管对象。有了托管对象之后，我们就可以用 Objective-C 代码来操作其中的数据了。

托管对象模型可以拥有一个或多个实体，每个应用程序的实体数量会有所差别。在制作托管对象之前，首先要把每个“饼干模型切割刀”（也就是实体）设计好。实体的设计与传统数据库中数据表的设计是相似的。

在设计数据库中的数据表时，你需要完成下列内容：

- ❑ 配置数据表名称 (table name)。
- ❑ 配置字段 (field) 并为每个字段设定“数据类型”(data type)。

而在设计实体时，你需要做的是：

- ❑ 配置实体名称 (entity name)。
- ❑ 配置属性，并为每个属性设定数据类型。
- ❑ 根据实体来配置 `NSManagedObject` 的子类 (该项可选)。

正如数据库中的表有字段一样，实体也有属性。属性必须有特定的数据类型（比方说字符串或整数）。如果想从实体中创建托管对象，那我们通常会根据实体来创建 `NSManagedObject` 的子类，但这并不是强制性的。采用 `NSManagedObject` 的子类确实有好处，比如可以在托管对象后面使用“点符号” (.) 访问相关属性，这样可以令代码更易阅读。无论是从 `NSManagedObject` 类还是从 `NSManagedObject` 的子类创建实例，我们都可以通过托管对象这种形式来操作数据。用数据库领域的术语来说，托管对象的实例类似于数据库中某张数据表里的一行 (row)。实体的名称与根据该实体创建出来的 `NSManagedObject` 子类的名称通常是一样的。根据实体来创建托管对象时，在实体中配置好的那些属性也会变成托管对象里的特性。图 2-2 演示了实体是如何在持久化存储区中的数据库与托管对象之间建立映射关系的。

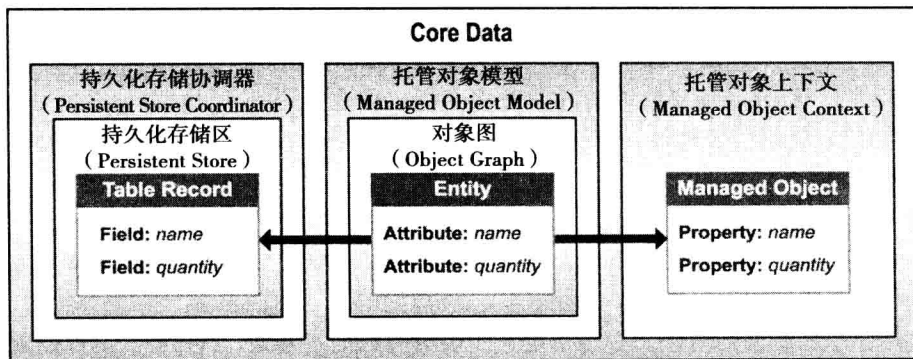


图 2-2 由 Core Data 实体在数据库与托管对象之间所建立的映射关系

实体是托管对象模型的基础，因为它可以把同一个范围内的数据从逻辑上组织起来。设计托管对象模型的时候，至关重要的一件事就是给实体起名字。给实体所起的名字应该由一两个英文单词构成，它要能描述出实体所表示的数据。Grocery Dude 程序的用户要能

够把待购买的东西放到购物清单中。用户外出购物时，可能会把“苹果”和“橘子”添加到购物清单里。而想到这一点之后，我们就会觉得：这个用来表示购物清单里待购物品的实体可能起名叫做“水果”会比较好。

在配置实体的时候一定要谨慎：为实体起的名字需要稍微通用一些，以便适应将来的变化，同时还必须足够具体，以便明确描述出它所表示的数据。有时候名字很好起，而有时候则必须反复权衡，需要根据应用程序当前和以后的功能来精心选择恰当的实体名称。由于购物清单里面不是只有水果，所以用“Item”（货品）作为实体名称，应该比“水果”更能准确地描述出清单中的待买物品。

请按下列步骤修改 Grocery Dude，以便添加 Item 实体：

1. 选定 **Model.xcdatamodeld**。
2. 点击 **Add Entity**。
3. 把新实体的名称改为 **Item**。

2.4 属性

属性（attribute）是实体的特征（property）。在本书范例程序中，Item 实体代表可以添加到购物清单里的东西。为了给 Item 实体拟定出合适的属性，我们需要考虑购物清单里所有货品的共性。一开始，你可能会拟定出下面这两个属性：

- ❑ **Item name** （货品名称）
- ❑ **Item quantity** （货品数量）

属性的名称必须以小写字母开头，而且不应该与 NSObject 或 NSObject 方法重名。Xcode 不允许开发者违背这条规则，如果违背了，它会给出警告，比方说，把实体的属性名设为“description”就是非法的。

根据 Item 实体来创建 NSObject 子类的时候，类中会出现与实体的属性同名的各项特性。与 Objective-C 中的其他对象一样，你也可以在 NSObject 子类上面用“点”（.）来引用类特性。使用 item.name 及 item.quantity 来获取特性值可以令代码更易理解。

请按下列步骤修改 Grocery Dude，以便把两个新的属性加入其中：

1. 在选定 **Item** 实体的前提下，点击 **Add Attribute** 按钮，新增名为 **name** 及 **quantity** 的属性。添加完成后的效果如图 2-3 所示。

向实体中添加属性的时候，必须指定它所表示的数据“类型”（type）。属性的默认类型是 Undefined（未定义）。你可以为每个属性指定不同的数据类型，而且有时可能还要预想一下这个属性在未来的用法。可供选择的属性类型有很多，而且作为 Objective-C 程序员，其中某些类型你可能已经比较熟悉了。[⊖]

⊖ 从数据类型的角度来说，下面的 2.5 节、2.6 节均应从属于 2.4 节，但为了与英文版保持一致，译文仍然按照英文原书来翻译。——译者注

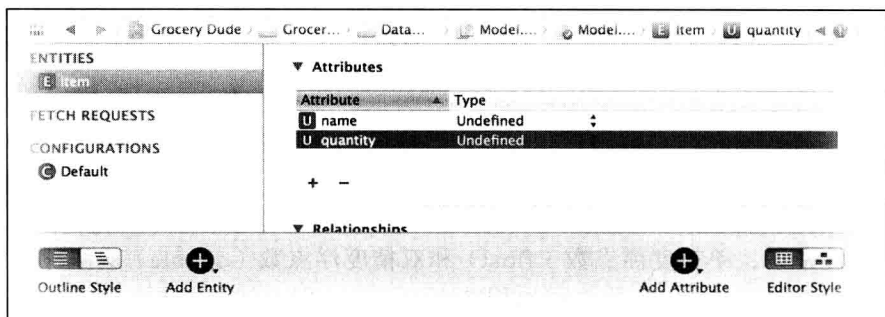


图 2-3 添加两个类型为 undefined 的属性

2.5 Integer 16、Integer 32 与 Integer 64

对于属性来说，这三种数据类型是比较相似的，它们都表示没有小数点的整数，唯一区别就在于能够表示多大或多小的数。由于 Core Data 使用“带符号的整数”（signed integer），所以取值范围从某个负数开始，而不是从 0 开始：

1. **Integer 16** 的取值范围是 -32 768 至 32 767
2. **Integer 32** 的取值范围是 -2 147 483 648 至 2 147 483 647
3. **Integer 64** 的取值范围是 -9 223 372 036 854 775 808 至 9 223 372 036 854 775 807

数字的值越大，所占的内存就越多。在这三种整数类型之间选择时，你需要思考当前属性的最小取值和最大取值。如果不能确定的话，那么通常可以选用 Integer 32。要是程序出错了，那就说明需要选用一种取值范围更广的数据类型，此时可以把属性的类型提升到 Integer 64。我们需要升级托管对象模型才能完成这种修改操作，第 3 章将会讨论此话题。

Integer 使用以 2 为底的数制，更通俗的说法是二进制。计算机执行整数运算的速度要比执行浮点数运算更快，因为它无需考虑运算所产生的余数。比方说，如果计算 10 除以 3 的话，那么只要算出结果是 3 就好了，余下的那个 1 可以丢弃。这种运算有个专门的术语，叫做低精度（low precision）运算。如果要用整数来表示货币，那么笔者强烈建议你用“1”表示“1 分钱”。这样的话，在执行财务计算的时候就不会出现舍入误差了。



提示 标准整数的最小取值与最大取值可以从 `stdint.h` 文件里看到：在 Xcode 中打开任意类文件，输入 `INT32_MAX`，然后用鼠标右击这几个字符，选择 **Jump to Definition**，你会看到 `stdint.h` 文件里定义了各种整数的最小取值与最大取值。你也会注意到：无符号整数的最大取值要比带符号整数的大，这是因为它们的值不会低于 0。Core Data 只使用带符号的整数，这样做的优点是既能表示负值，又能表示正值，缺点则是最大取值要比无符号整数的小。

根据实体来创建 `NSManagedObject` 子类时，如果实体中某个属性的类型为 `Integer 16`、`Integer 32` 或 `Integer 64`，那么在创建好的子类里，相关特性的类型就会是 `NSNumber`。

2.6 单精度浮点数与双精度浮点数

对于属性来说，单精度浮点数（`float`）和双精度浮点数（`double`）这两种数据类型可以看作带小数的非整数。它们都可以用来表示实数，但也都有一定的限制。单精度浮点数与双精度浮点数都使用以 2 为底的数制（也叫二进制），对 CPU 来说，这是一种原生的数制，它容易引起舍入误差。以 $1/5$ 这个分数为例，如果采用十进制，那我们可以精确地将其写为 0.2，但如果改用二进制，则只能表示出它的近似值。小数点后面的数位越多，精度就越高，表示出来的近似值也就越准确。高精度的数会占用更多内存，以保持其准确性。

与单精度浮点数相比，双精度浮点数所包含的二进制位（`bit`）的个数是它的两倍。单精度浮点数用 32 个二进制位来保存其数据，而双精度浮点数则占用 64 个二进制位。两者都采用科学计数法，也就是说，整个浮点数是由尾数和指数（表示 2 的多少次幂）组成的。双精度浮点数有 64 个二进制位，这意味着它的取值范围比单精度浮点数广，精度也比单精度浮点数高。

在 iOS 中，最大的单精度浮点数是 340 282 346 638 528 859 811 704 183 484 516 925 440.000 000。单精度浮点数与双精度浮点数都有符号位（`sign bit`），所以，iOS 中数值最小的单精度浮点数是 -340 282 346 638 528 859 811 704 183 484 516 925 440.000 000，而数值最大的双精度浮点数则比数值最大的单精度浮点数还要大出许多。本章最后有道习题，题中给出了一段代码，可以打印出各种数值数据类型的最小值与最大值。

在单精度浮点数和双精度浮点数之间取舍时，需要考虑正在配置的这个属性有何特点：它的最小取值和最大取值是多少？是不是真的需要超过 7 位的精度（单精度浮点数所提供的精度大约是 7 位）？如果不需要的话，那么在 iOS 平台上还是应该选用单精度浮点数，因为在 64 位的 iPhone 5S 出品之前，单精度浮点数这种数据类型更能够同底层的处理器相匹配。虽说使用一大批双精度浮点数特性看上去有些不太合理，但是要注意：数据库可能会导致程序在存储量方面的需求变得比想象中更大，因为它们可能要包含巨量的数据行。目前设备的能力和容量都很大，所以在大多数情况下，使用双精度浮点数其实也可以。如果追求浮点运算的速度，同时又不太关心精度，那么选用单精度浮点类型会更加合适。但在涉及“元”或“分”等货币单位的财务计算中，则不应该使用单精度浮点数或双精度浮点数，因为“舍入误差”会导致钱数出错！

根据实体来创建 `NSManagedObject` 子类时，如果实体中某个属性的类型为单精度浮点类型或双精度浮点类型，那么在创建好的子类里，相关特性的类型就会是 `NSNumber`。

2.6.1 小数

在涉及货币或其他十进制运算的场合中，建议把属性的数据类型设为小数（decimal）。与二进制不同，对于 CPU 来说，十进制并不是原生的数制，这就意味着以小数来运算时，处理器会有比较大的开销。与单精度浮点数和双精度浮点数一样，小数也是由尾数（该尾数是个整数）、指数及符号组成的。虽说内存占用量和处理时间都比较多，但是小数的计算精度却很高。在这种数制里，0.1 这个数就可以精确地表示出来了。如果属性的数据类型是小数，那么它就会把 0.1 存储为“1/10¹”。

与值最大的双精度浮点数值相比，值最大的小数其实并不算大，但它的精度却比双精度浮点数高出许多，而且在有些时候甚至是完全准确的。本章最后有道习题，题中给出的那段代码会以 1/3 这个数为例，打印出每一种“数值数据类型”所能达到的精度。

根据实体来创建 `NSManagedObject` 子类时，如果实体中某个属性的类型是小数类型，那么在创建好的子类里，相关特性的类型就会是 `NSDecimalNumber`。在 `NSDecimalNumber` 上面执行计算时要注意：若想保留精度，则只能使用 `NSDecimalNumber` 内置的方法。

2.6.2 字符串

对于属性来说，字符串这种数据类型可以存放字符数组（array of character）或普通文本（plain old text）。作为 Objective-C 程序员，你应该已经对字符串相当熟悉了。根据实体来创建 `NSManagedObject` 子类时，如果实体中某个属性的数据类型是字符串，那么在创建好的子类里，相关特性的类型就会是 `NSString`。

2.6.3 Boolean

对于属性来说，Boolean 这种数据类型可用来存放“是”或“否”这两种值。根据实体来创建 `NSManagedObject` 子类时，如果实体中某个属性的数据类型是 Boolean，那么在创建好的子类中，相关特性的类型就是 `NSNumber`。若想从 `NSNumber` 中获取 Boolean 值，只需向该实例发送 `boolValue` 消息即可。而若想将 `NSNumber` 设置为某个 Boolean 值，则可使用 `numberWithBool` 方法。

2.6.4 日期类型

顾名思义，日期（date）这种数据类型就是用来在属性中保存日期和时间的。根据实体来创建 `NSManagedObject` 子类时，如果实体中某个属性的类型是日期类型，那么在创建好的子类中，相关特性的类型就是 `NSDate`。

2.6.5 二进制数据类型

如果要保存照片、音频或其他由“0”、“1”二进制位所组成的连续 BLOB，那么就

应该把属性的类型设为二进制数据类型 (Binary Data)。根据实体来创建 `NSManagedObject` 子类时, 如果某个属性的类型是二进制数据, 那么在创建好的子类中, 相关特性的类型就是 `NSData`。至于如何在数据和 `NSData` 之间转换, 那要依照具体存储的数据来定。二进制数据较常见的用途就是存储照片。存储照片时, 可以通过 `UIImagePNGRepresentation()` 或 `UIImageJPEGRepresentation()` 来把 `UIImage` 转换成 `NSData`。而获取照片时, 则可以通过 `UIImage` 的类方法 `imageWithData` 把 `NSData` 转换为 `UIImage`。二进制数据这种数据类型对于大文件来说比较合适, 因为在属性的设置选项中, 我们可以开启 **Allows External Storage**, 将其“无缝地”存储在数据库之外。启用了这个选项之后, **Core Data** 就会自行判断是把文件存放在数据库内的效率高还是存放到数据库外的效率高。

2.6.6 可变类型

可变 (Transformable) 数据类型很适合用来把 Objective-C 对象存放到属性里。这种属性类型比较灵活, 它可以存放任意类的实例。比方说, `UIColor` 类的实例就可以保存在类型为可变类型的属性里。在根据实体来创建 `NSManagedObject` 子类时, 如果实体中某个属性的类型是可变类型, 那么在创建好的子类中, 相关特性的类型就是 `id`。若想把 `id` 对象放入存储区 (或将其从存储区里取出来), 则需借助 `NSValueTransformer` 类的实例或 `NSValueTransformer` 子类的实例。`NSValueTransformer` 类可以在属性与 `NSData` 之间“透明地”执行转换。转换过程也比较简单, 尤其当待存储的类本身已经实现了 `NSCoding` 协议时更是如此。假如实现了该协议, 那么系统就会提供默认的 transformer, 而这个 transformer 自己知道如何“压缩” (archive) 或“解压缩” (un-archive) 相关的对象。

请按下列步骤修改 **Grocery Dude**, 以便配置其中的属性:

1. 将 **name** 属性的 Type 设为 **String**。
2. 将 **quantity** 属性的 Type 设为 **Float**。
3. 在 **Item** 实体中添加名为 **photoData** 的属性, 并将其 Type 设为 **Binary Data**。这个属性用来存放货品照片的图像信息。(注意: 目前并不启用 **Allows External Storage**, 本书后面再讲解何时启用它。)
4. 在 **Item** 实体中添加名为 **listed** 的属性, 并将其 Type 设为 **Boolean**。这个属性用来表示货品是否已出现在购物清单中。
5. 在 **Item** 实体中添加名为 **collected** 的属性, 并将其 Type 设为 **Boolean**。如果用户已经拿到了所要购买的货品, 那么这个属性就是“真”, 这表示该货品已经可以从购物清单中勾掉了。

执行完上述步骤之后的数据模型如图 2-4 所示, 现在每个属性的数据类型都已经配置好了。

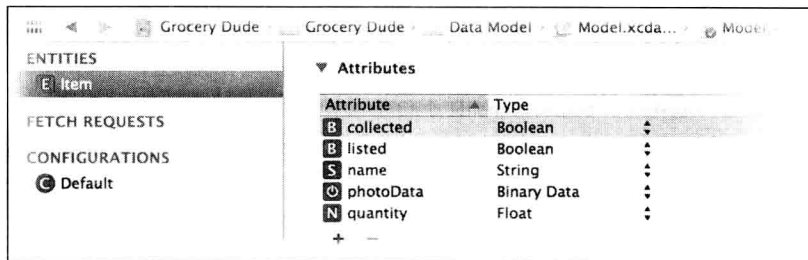


图 2-4 在 Table 风格的编辑器中显示的 Grocery Dude 数据模型

为了给以后更加复杂的数据模型做准备，笔者在这里告诉你如何切换到图形化的编辑界面。要改变编辑器的显示模式，只需点击图 2-5 正下方的 **Editor Style** 按钮。图 2-5 左侧演示了编辑器在 Graph 风格下的样貌。

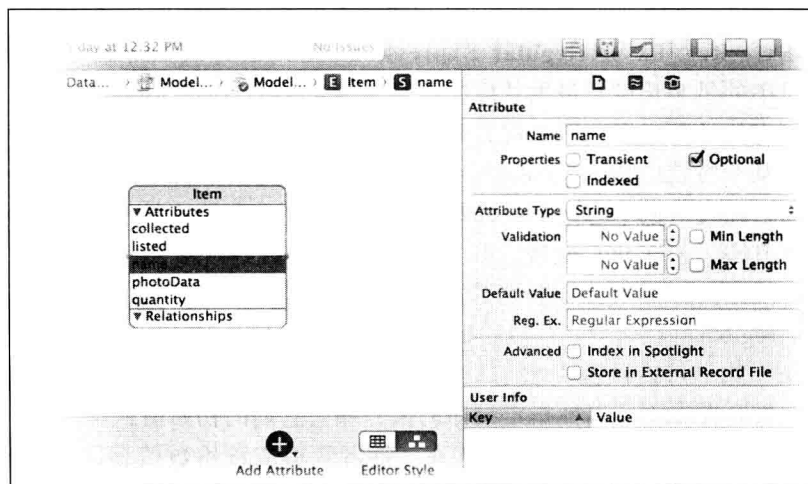


图 2-5 在 Graph 风格的编辑器中显示的 Grocery Dude 数据模型

2.7 属性的各种设置选项

图 2-5 右侧是 Data Model Inspector，开发者可以在这个界面中配置类型之外的其他属性选项。选中某个属性之后，按“**Option + ⌘ + 3**”组合键，即可显示该界面。可供配置的选项根据属性的类型而有所变化。并不是每一种属性都能配置下列选项：

- ❑ **Transient** 如果在 Properties 中勾选了这一项，那么该特性就不会写入持久化存储区了。“不写入持久化存储区的特性”听上去有些奇怪，但有的时候，只需把特性留在托管对象上下文里面就行了。比方说，你需要计算某个临时的值，而这种值就可以放在 transient 特性中，此特性会留在上下文里，从而能够得益于撤销或重做等功能。

- ❑ **Optional** optional 特性并不一定要有值。所有的特性在刚创建出来的时候都是 optional 特性。如果某特性不是 optional 特性，那么在把这个非 optional 特性放回存储区的时候，它必须具备有效的值才行。
- ❑ **Indexed** 系统会优化 Indexed 特性以提升搜索效率，但代价是要在底层的持久化存储区中占用额外的空间。这些额外空间的大小要根据待索引的数据量来定。如果不打算搜索某个属性，那么就不要再勾选 Indexed，这样可以节省一些空间。
- ❑ **Validation** 你可以使用 Validation 中的各个选项来阻止不合理的数据进入持久化存储区。每一种数值型的属性类型都支持相同的 validation（验证）选项，也就是可以规定其最小值与最大值。同理，对于字符串类型或日期类型的属性来说，也可以限定其字符串长度或日期范围。无效的值其实也可以出现在托管对象上下文里面，只要在调用 save：之前能把这个问题解决就行。一般来说，应该在用户试图将焦点从输入控件（比如 UITextField）中移开的时候验证数据。
- ❑ **Reg. Ex.** Reg. Ex. 是 **Regular Expression**（正则表达式）的缩写，它不仅能够限定字符串的最小长度及最大长度，而且还能实现很多验证功能。当然可以用它来限定字符串的长度，但一般来说，我们会用正则表达式来判断属性中的字符串值是不是能与某个特定的模式（pattern）相匹配。如果为某个属性配置了 Reg. Ex. 验证功能，那么托管对象中与之对应的特性值就必须能和给定的模式相匹配才行，否则无法将其写入持久化存储区。开发者所配置的待匹配模式必须符合 ICU Reg Ex 规范。该规范的详情以及可供使用的配置选项请参考：<http://userguide.icu-project.org/strings/regexp>。
- ❑ **Default** 除了可变数据类型与二进制数据之外，其余类型的属性都可以具备默认值。如果开发者不给属性指定具体的值，那么它们的初始值就是默认值。考虑到后端 SQLite 数据库处理 null 值的方式，笔者觉得应该给数值型的属性设定默认值。对于字符串类型的属性来说，默认值要依照具体情况来定，也就是说，得根据自己的需求来为这种属性选择适当的默认值。而对于日期类型的属性来说，开发者则无法在 Model Editor 中把它的默认值设为“now”（当前时间）^①。
- ❑ **Allows External Storage** 开启了该选项之后，类型为二进制数据的属性就可以把大量数据保存在持久化存储区之外了。假如要保存照片、音频、视频等数据量非常大的媒体文件，那么笔者推荐你启用该选项。启用之后，Core Data 会自动把数据量超过 1MB 的属性值保存在 SQLite 持久化存储区之外。但如果底层的持久化存储区是 XML 格式（注意，iOS 不支持这种格式的存储区），那么该选项不起作用。
- ❑ **Index in Spotlight** 这个选项不会影响 iOS 应用程序，它的用途是把基于 Core Data 的 Mac 应用程序同 Spotlight 集成起来。Spotlight 是一种搜索机制。在 Mac 操作系统中，屏幕右上角会有个“放大镜”图标，用户点击该图标之后，即可利用该机制

① 在 Xcode 5.1.1 中，如果在 Default 文本框中输入 now，那么当焦点离开该文本框之后，Xcode 就会自动把 now 替换成当前的时间。——译者注

来搜索。Mac 应用程序的某个 Core Data 属性如果启用了 Index in Spotlight 选项，那么它的值就会出现在 Spotlight 的搜索结果中。Core Data 会创建一种长度为 0 的隐藏文件，用以表示持久化存储区中的记录，而 Spotlight 在执行搜索的时候，则会寻找这种文件。如果持久化存储区里某个属性的值变了，而这个属性又启用了 Index in Spotlight 选项，那么存储区外对应的那个文件也会随之自动更新。

❑ **Store in External Record File** 启用了该选项之后，系统会把持久化存储区里的数据复制成 XML 格式，并保存在存储区之外。该选项如果和 Index in Spotlight 选项一起启用，那么在创建供 Spotlight 所用的“索引文件”（index file）时，文件里面就会填有一些值。除非有特殊需要（比如为了调试），否则笔者不建议开启此选项。假如想通过“external records”（外部记录）给其他应用程序提供数据，那么请注意：包含 records 的目录其结构可能会改变。

❑ **Name** 如果某个属性的类型是可变类型，那么名称这一栏中填写的名称将会用作 NSValueTransformer 子类的名称，而这个子类会知道如何在任意的类与 NSData 之间相互转换。

请按下列步骤修改 Grocery Dude，为相关的属性启用 Indexed 选项并配置其默认值：

1. 勾选 **name** 属性的 **Indexed** 选项。
2. 将 **name** 属性的 **Default Value**（默认值）设为 **New Item**。
3. 将 **quantity** 属性的 **Default Value** 设为 **1**。
4. 将 **listed** 属性的 **Default Value** 设为 **YES**。这样一来，新创建的 item 就会出现在购物清单中了。
5. 将 **collected** 属性的 **Default Value** 设为 **NO**，这么做是为了使购物清单中的新 item 不会处于“打上对勾”（ticked off）的状态。

2.8 创建 NSManagedObject 的子类

托管对象模型就位之后，我们就该根据 Item 实体来创建 NSManagedObject 的子类了。有了这些子类文件，就可以在对象上面直接用“点”（.）来操作数据了，而不用再编写 SQL 查询语句。如果将来模型变了，那么就需要按照下面所讲的流程重新生成这些文件。尽管开发者也可以在这些生成的文件里自行添加方法，但笔者觉得不应该这么做，因为重新生成之后，原来所做的修改就会丢失。假如确实需要添加自定义的方法，那么可以从继承子类，或是针对生成的文件创建类。

请按下列步骤修改 Grocery Dude，以便生成与 NSManagedObject 子类相关的文件：

1. 选中 **Item** 实体。
2. 点击 **Editor > Create NSManagedObject Subclass...** 菜单项。
3. 确保 **Model** 处于勾选状态，然后点击 **Next** 按钮。

4. 勾选 **Item** 实体，然后点击 **Next** 按钮。
5. 确保 **Targets** 中的 **Grocery Dude** 已处于勾选状态。
6. 不要勾选 **Use scalar properties for primitive data types**。
7. 确保文件保存在 Grocery Dude 项目的目录之下，然后点击 **Create** 按钮。

执行完上述步骤之后，Xcode 项目里会多出来两个新文件，分别是 `Item.h` 和 `Item.m`。这两个文件都是根据 Item 实体生成的，程序清单 2-1 列出了 `Item.h` 的代码。各特性的出现顺序可能与书中不同。

程序清单2-1 Item.h

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface Item : NSManagedObject

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSNumber * listed;
@property (nonatomic, retain) NSNumber * quantity;
@property (nonatomic, retain) NSNumber * collected;
@property (nonatomic, retain) NSData * photoData;

@end
```

请注意，实体中各个属性的类型与生成的类中各个特性的类型是稍有区别的。下面总结了实体的属性与托管对象的特性是如何对应起来的：

- ❑ 实体中的 **Date** 属性会成为类里的 `NSDate` 特性。
- ❑ 实体中的 **String** 属性会成为类里的 `NSString` 特性。
- ❑ 实体中的 **Decimal** 属性会成为类里的 `NSDecimalNumber` 特性，而其他各种数值数据类型会成为类里的 `NSNumber` 特性。
- ❑ 实体中的 **Binary Data** 属性会成为类里的 `NSData` 特性。
- ❑ 实体中的 **Transformable** 属性，会成为类里的 `id` 特性。

查看一下 `Item.m` 文件就会发现，实现文件只是给各个特性都添加了 `@dynamic` 修饰符而已。Core Data 用这种方式告诉大家：获取及设置特性值所需的方法都会动态地生成，不用开发者自己去实现。

2.9 Scalar Properties for Primitive Data Types 选项

根据 Item 实体来创建 `NSManagedObject` 子类时，会注意到 **Use scalar properties for primitive data types**（用 scalar 特性来表示原始数据类型）这个选项。启用了该选项之后，`NSManagedObject` 子类只会在没有其他途径可选时才去使用对象类型的特性。下面列出了该选项开启之后实体的属性与托管对象的特性之间的对应关系：

- ❑ 实体中的 Date 属性会成为类里的 `NSTimeInterval` 特性。
- ❑ 实体中的 Double 属性会成为类里的 `double` 特性。
- ❑ 实体中的 Float 属性会成为类里的 `float` 特性。
- ❑ 实体中的 Integer 16/32/64 属性分别会成为类里的 `int16_t/int32_t/int64_t` 特性。
- ❑ 实体中的 Boolean 属性会成为类里的 `BOOL` 特性。

这个选项对于字符串、小数、二进制数据或可变类型类型的属性没有影响，与这些属性相对应的特性仍然是“对象指针”（object pointer）。启用了 `Use scalar properties for primitive data types` 选项之后，`NSManagedObject` 子类文件会生成另一套 `getter` 方法，这样的话，开发者在使用这些 `scalar` 值之前就无需代码执行 `unbox`（数值解包）操作了。

2.10 代码片段：demo 方法

本书会用一些代码来示范某些知识点，而这些代码无须包含在最终的应用程序项目里。程序清单 2-2 新建了名为 `demo` 的方法，而且还修改了 `applicationDidBecomeActive` 方法，该方法通过 `[self cdh]` 把 Core Data 准备好，然后再调用 `demo` 方法。

程序清单2-2 AppDelegate.m文件中的demo方法

```
- (void)demo {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
}
- (void)applicationDidBecomeActive:(UIApplication *)application
{
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    [self cdh];
    [self demo];
}
```

请按下列步骤修改 `Grocery Dude`，以便实现 `demo` 方法：

1. 把程序清单 2-2 中的 `demo` 方法添加到 `AppDelegate.m` 文件顶部的 `#define debug 1` 这一行下面。
2. 用程序清单 2-2 中的 `applicationDidBecomeActive` 方法来替换 `AppDelegate.m` 文件里原有的 `applicationDidBecomeActive` 方法。

2.11 创建托管对象

所有事情都准备好之后，现在就可以新建一些托管对象了。新对象是由 `NSEntity-`

Description 按照指定的名称并根据某个特定的实体而创建出来的。除了要指定对象所依据的实体之外, 还需提供指向托管对象上下文的指针, 创建好的托管对象将会放在那个上下文里面。在 application delegate 中, 可以通过 [self cdh] 或 _coreDataHelper 的 context 特性来获得这个上下文。

程序清单 2-3 演示了如何根据实体来新建托管对象, 并将其插入上下文。要完成这项操作其实很简单, 只需要调用 NSEntityDescription 类的 insertNewObjectForEntityForName 方法, 并把适当的实体名称及指向上下文的指针传进去即可。

根据 Item 实体创建好托管对象之后, 就可以直接用代码来操作它的值了。程序清单 2-3 底部的 NSLog 命令就说明了这一点: 我们能够把 newItem.name 当作字符串变量传给 NSLog。在操作对象的时候, 通过“点”(.) 来访问其特性是一种很清晰的写法, 可以令代码更易读懂。

程序清单2-3 AppDelegate.m中的demo方法(用来演示如何向上下文中插入托管对象)

```
NSArray *newItemNames =
[NSArray arrayWithObjects:
 @"Apples", @"Milk", @"Bread", @"Cheese", @"Sausages", @"Butter",
 @"Orange Juice", @"Cereal", @"Coffee", @"Eggs", @"Tomatoes", @"Fish",
 nil];

for (NSString *newItemName in newItemNames) {
    Item *newItem =
    [NSEntityDescription insertNewObjectForEntityForName:@"Item"
     inManagedObjectContext:_coreDataHelper.context];
    newItem.name = newItemName;
    NSLog(@"Inserted New Managed Object for '%@'", newItem.name);
}
```

请按下列步骤修改 Grocery Dude, 以便向上下文中插入托管对象:

1. 把 #import "Item.h" 语句添加到 AppDelegate.m 文件顶部。
2. 修改 AppDelegate.m 中的 demo 方法, 把程序清单 2-3 中的代码置于方法体的底部。

运行应用程序之后, 就应该能在控制台中看到这些托管对象的名字了。Core Data 已经成功运作起来了, 你是不是该鼓励一下自己呢? 程序运行效果如图 2-6 所示。

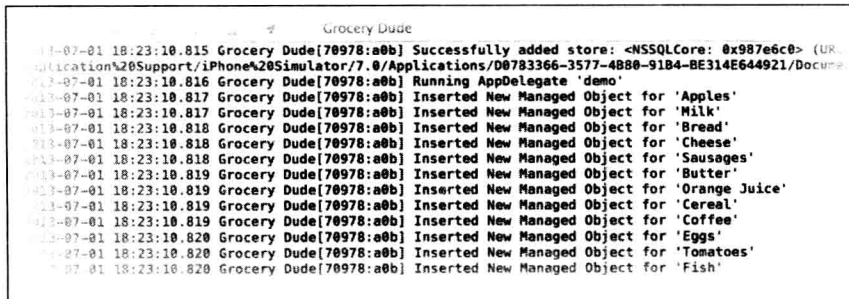


图 2-6 向上下文中插入托管对象



在给项目添加了某些属性之后，下次启动程序时可能会出现“The model used to open the store is incompatible with the one used to create the store.”（打开存储区时所用的模型与创建存储区时所用的不符）错误。假如真的碰到了这个错误，请在模拟器或手机中删除该应用程序，然后重新运行项目，这样就能装好一份新的程序了。如果还是报错，那么可以点击 Xcode 的 **Product>Clean** 菜单项。到了第 3 章，我们会讲解一套优雅的模式升级方案，并把它实现出来。

2.12 后端 SQL 的可见性

如果只在控制台的日志中查看 Core Data 所输出的结果，那么意义并不算太大。你并不知道这些事情背后究竟发生了什么？Core Data 对持久化存储区中的数据到底进行了哪些操作？这些操作是否恰当？为了提供无缝的 Core Data 体验，系统都生成了哪些 SQL 查询语句？每次在模拟器中运行程序的时候，是不是会插入重复的对象？

有个极其详尽的调试选项可以提供足够的信息，告诉你这些操作背后所发生的事情，从而令你知道上述那些问题的答案。这个调试选项会把系统自动生成的 SQL 查询语句打印出来，使开发者深刻认识到 Core Data 的工作原理。

请按下列步骤修改 Grocery Dude，以便开启 SQL Debug 模式：

1. 点击 **Product > Scheme > Edit Scheme...** 菜单项。
2. 点击 **Run Grocery Dude**，并切换到 **Arguments** 分页。
3. 点击 **Arguments Passed On Launch** 区域中的“+”按钮，以新增参数。
4. 输入新参数 `-com.apple.CoreData.SQLDebug 3`，然后点击 **OK** 按钮。

现在我们已经开启了第三级（level 3）的 SQL Debug 模式，然后重新运行应用程序。按下手机的 home 键（如果是在模拟器中运行，那就按“**Shift + ⌘ + H**”组合键或点击 **Hardware > Home** 菜单项），并观察控制台中的日志。你会看到，系统自动生成了一些 INSERT 语句，开发者无须手工编写这些语句。图 2-7 显示了其中一部分输出信息。

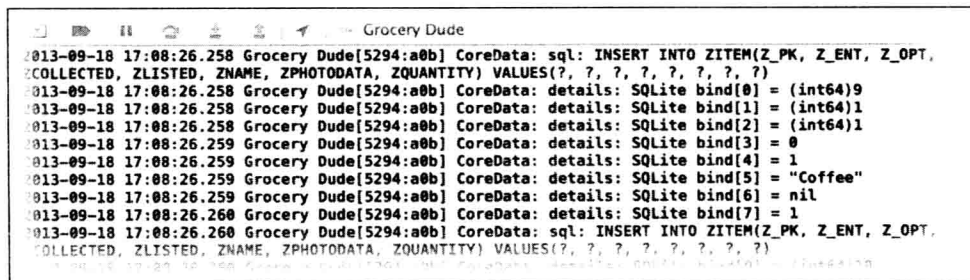


图 2-7 由 Core Data 所生成的 SQL 查询语句

在图 2-7 中，SQLite bind[0] 至 SQLite bind[7] 等变量都是用来拼合 INSERT 语句的，

而这种 INSERT 语句的用途则是把托管对象的特性值插入到 ZITEM 表的行中, 这张表位于持久化存储区里。ZITEM 表是同 Item 实体相关联的。通过名称可以看出实体的属性与数据库中的 field (字段) 是怎样对应起来的。Core Data 使用前缀字母“z”作为其标准的命名约定。

为了验证保存在 SQLite 持久化存储区里的托管对象, 你可能需要借助第三方工具来查看其内容。请注意: 笔者并不建议直接修改数据库。图 2-8 演示了如何寻找包含 Grocery-Dude.sqlite 文件的 iOS 仿真器工作目录。由于 Library 目录是隐藏的, 所以需要右击 **Finder**, 选择 **Go to Folder** 菜单项, 然后手工输入目录名。具体的目录位置是: `/Users/Username/Library/Application Support/iPhone Simulator/`, 当然了, 你需要把其中的 Username 部分替换为自己的用户名。Applications 目录下面的子目录名称可能会和书中有所不同, 因为这是个随机生成的 GUID。

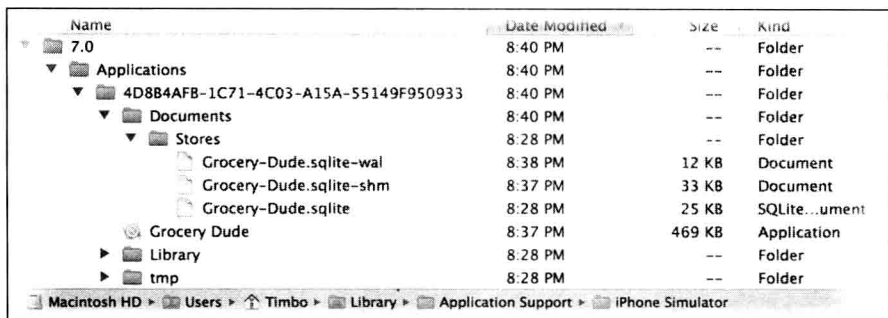


图 2-8 用 iOS 仿真器运行 Grocery Dude 程序之后, 其 SQLite 数据库文件所在的目录

除了 Grocery-Dude.sqlite 文件以外, 在同一目录下还有两个分别以 `-wal` 及 `-shm` 结尾的文件。iOS 7 默认会采用一种新的“数据库日志记录模式” (database journaling mode), 而这两个文件正是由这种记录模式所生成的。第 8 章将会详细讨论这一模式。对于目前来说, 为了查看 Grocery-Dude.sqlite 的内容, 我们必须先禁用这种模式。要想禁用 iOS 7 默认的日志记录模式, 就必须在添加持久化存储区的时候传入一种新的选项。

请按下列步骤修改 Grocery Dude, 以便禁用 iOS 7 默认的日志记录模式:

1. 修改 CoreDataHelper.m 文件的 loadStore 方法, 把下面这行代码添加到 NSError*error = nil 这一行之上:

```
NSDictionary *options =
    @{@"NSSQLitePragmasOption": @{@"journal_mode": @"DELETE"}};
```

2. 在 CoreDataHelper.m 文件的 loadStore 方法中找到调用的 addPersistentStoreWithType 那行语句, 把其中的 options:nil 换成 options:options。

3. 再次运行应用程序, 然后 `-wal` 文件应该就会消失了, 这表明所有数据都已存放到 Grocery-Dude.sqlite 文件里面了。可将 `-shm` 文件删除, 也可忽略该文件。

通过 Google 可以搜到很多能够打开 SQLite 数据库文件的数据库浏览工具。在 Sourceforge 网站上有一款好工具，名字叫做 **SQLite Database Browser**^①，笔者建议你现在就把它下载下来，并安装在电脑上试试看。由于该程序未经签名认证，所以需要在 **System Preferences > Security & Privacy > General** 中开启 Allow applications downloaded from Anywhere 选项，然后才能使用。如果不想为了使用该程序而开启这个选项，那么可以在 Mac App Store 中搜索 **extension:sqlite**，这也是一款能够打开 .sqlite 文件的程序，而且它通过了 Mac App Store 的认证。

虽说查看数据库的内容对程序调试工作很有帮助，但是对于这种由 Core Data 所管理的数据库来说，不应该在编写代码时依赖它内部的私有模式，因为苹果公司可能会在不通知开发者的前提下自行改变其结构。



提示 由于 Library 文件夹是隐藏的，所以若想查看 Grocery-Dude.sqlite 文件的内容，方便一些的办法是：直接在这个文件上面点击鼠标右键，选择 **Open With** 菜单，然后指定用 **SQLite Database Browser** 来打开它。请注意，在 Xcode 中正在运行 Grocery Dude 程序的时候，不要用数据库浏览工具打开 SQLite 文件，否则应用程序在尝试打开数据库时会超时。

图 2-9 演示了持久化存储区中的托管对象所具备的各种特性值。

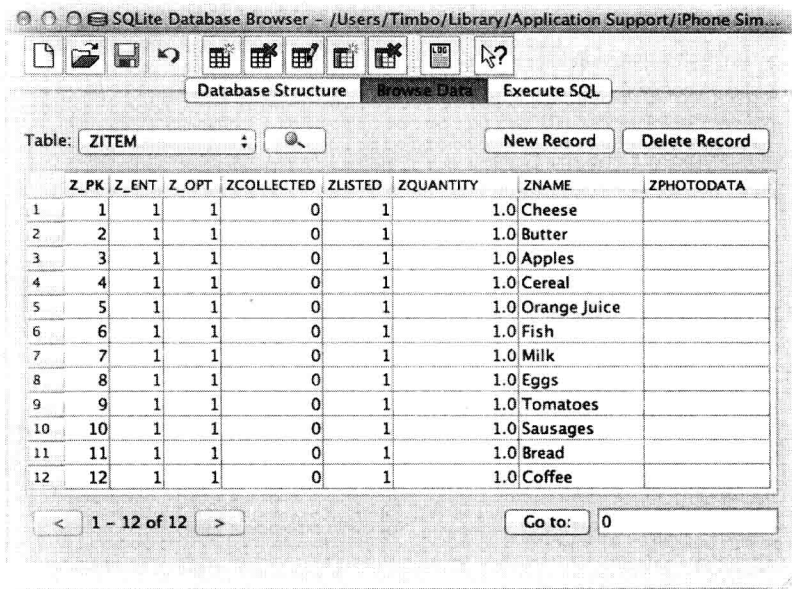


图 2-9 查看 Grocery Dude 程序持久化存储区中的 SQLite 数据库文件

① 网址是：<http://sourceforge.net/projects/sqlitebrowser/>。——译者注

假如发现了重复的条目，那就说明上下文保存了不止一次。目前来看，只有当用户按下手机的 home 键时，程序才会保存上下文。而程序每次启动的时候，demo 方法都会插入新对象，于是就产生了重复。但现在不用担心此问题，因为稍后我们就会讲解如何删除托管对象。

请按下列步骤修改 Grocery Dude，以免再产生重复的数据：

1. 修改 AppDelegate.m 文件的 demo 方法，把 NSLog 以外的所有代码都删掉。

2.13 获取托管对象

想操作托管对象上下文中的现有数据，就必须先把它获取（fetch）过来。假如待获取的数据没有放在上下文里，那么 Core Data 会从底层的持久化存储区里把它拿出来，这个过程对开发者来说是透明的。要执行获取操作，就得有 NSFetchedRequest 实例，该实例会返回 NSArray，这个数组里面的元素都是托管对象。在执行获取操作的时候，NSFetchedRequest 会根据特定的实体，把每个托管对象都放在 NSArray 这个数组中，并将其返回给调用者。用 SQL 数据库领域的术语来说，获取操作类似于 SELECT 语句。相关代码如程序清单 2-4 所示。

程序清单2-4 AppDelegate.m文件的demo方法（演示如何用NSFetchedRequest获取数据）

```
NSFetchedRequest *request =
[NSFetchedRequest fetchRequestWithEntityName:@"Item"];
[_coreDataHelper.context executeFetchRequest:request error:nil];
```

请按照下列步骤修改 Grocery Dude，以便获取 Item 实体的所有实例：

把程序清单 2-4 中的代码添加到 AppDelegate.m 文件的 demo 方法里面。

运行程序并注意控制台中输出的日志，你将会看到 12 行非常相似的信息，每一行信息都代表从数据库中取出的一个托管对象。假如你的数据库里有重复数据，那么具体的行数可能和书中列出的有所不同。图 2-10 演示了正常运行程序时所产生的结果。

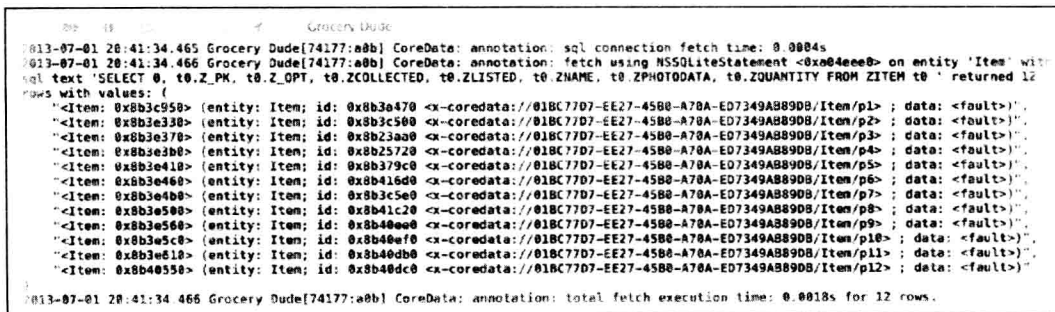


图 2-10 获取到的托管对象

如果想查看每个托管对象的特性值，那么可以用 for 循环来遍历 NSArray，并用 NSLog 把每个 item 的 name 特性值打印出来。相关代码如程序清单 2-5 所示。

程序清单2-5 AppDelegate.m文件中的demo方法（在获取到的托管对象中查看name特性）

```
NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Item"];
NSArray *fetchedObjects =
[_coreDataHelper.context executeFetchRequest:request error:nil];
for (Item *item in fetchedObjects) {
    NSLog(@"Fetched Object = %@", item.name);
}
```

请按下列步骤修改 Grocery Dude，以便在控制台的日志中显示出每个 item 的 name：

1. 把 AppDelegate.m 文件的 demo 方法改为程序清单 2-5 的样子。方法开头的 NSLog 语句可以保留，也可以删掉。

重新运行应用程序，并观察控制台中的日志，现在应该可以看到每个托管对象的 name 特性值了。正常的运行效果如图 2-11 所示。

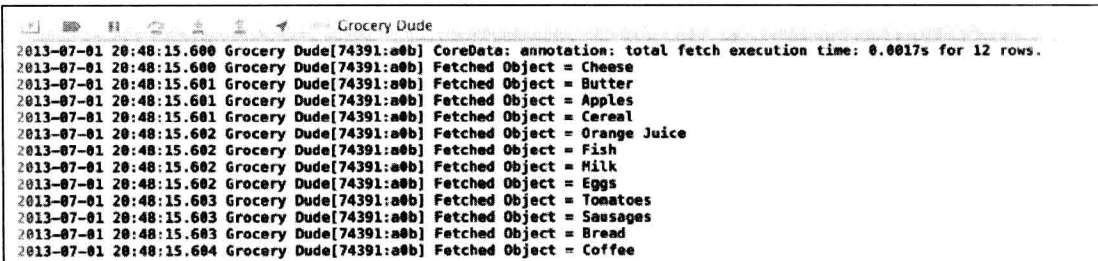


图 2-11 获取到的各托管对象所具备的 name 特性值

2.13.1 对获取请求的结果进行排序

NSFetchRequest 执行完毕之后，会返回 NSArray，而 NSArray 本身就支持对其中的元素进行排序。另外，我们也可以换一种办法，就是给 NSFetchRequest 配置排序描述符（sort descriptor），这样的话，NSFetchRequest 就可以直接按特定方式对获取到的托管对象进行排序了。这个排序描述符是作为 NSSortDescriptor 的实例传给 NSFetchRequest 的。用 SQL 数据库的术语来说，排序描述符就类似于 ORDER BY 语句。程序清单 2-6 列出了相关代码。

程序清单2-6 AppDelegate.m文件中的demo方法（演示排序功能）

```
NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Item"];
```

```

NSSortDescriptor *sort =
    [NSSortDescriptor sortDescriptorWithKey:@"name" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sort]];

NSArray *fetchedObjects =
    [_coreDataHelper.context executeFetchRequest:request error:nil];
for (Item *item in fetchedObjects) {
    NSLog(@"Fetched Object = %@", item.name);
}

```

请按下列步骤修改 Grocery Dude，以便对获取到的托管对象进行排序：

1. 修改 AppDelegate.m 文件的 demo 方法，用程序清单 2-6 中的粗体代码把原有的相关代码替换掉。

再次运行应用程序，并查看控制台中的日志，你会发现托管对象已经按照其名称的字母顺序排列好了。正常的运行效果如图 2-12 所示。

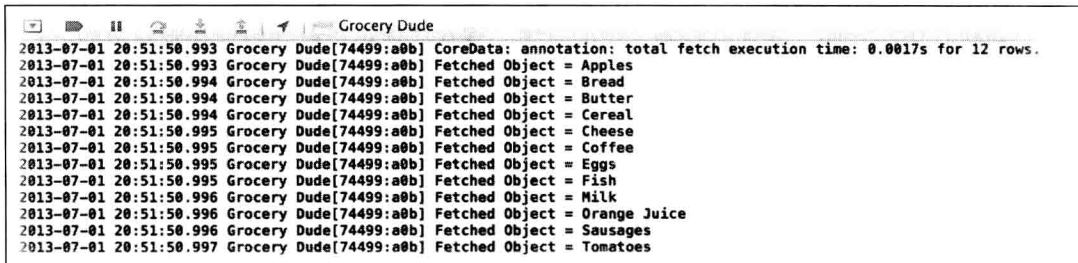


图 2-12 获取到的托管对象已按名称排好顺序

2.13.2 对获取请求的结果进行筛选

有时我们并不想把与某个实体有关的全部对象都获取过来，这时可以通过谓词来筛选。我们采用 NSPredicate 实例来定义谓词，并将其传给 NSFetchedRequest 实例。有了谓词之后，获取请求就会根据谓词中的标准来限定获取到的托管对象的数量。由于谓词与具体的持久化存储区无关，所以不论后端采用何种存储区，都可以使用相同的谓词来筛选它。但是，在特殊情况下，某些谓词无法适用于特定格式的存储区。比方说，**matches** 操作符可以对 in-memory 存储区执行筛选，却不能用在 SQLite 格式的存储区上。以 SQL 数据库的术语来说，谓词类似于 WHERE 子句。

在获取请求的执行过程中，系统要根据每个托管对象来对谓词分别求值。谓词的求值结果是个 Boolean 值。如果是 YES，那就表明该托管对象符合谓词中的标准，于是也就可以留在最终的获取结果中；但若是 NO，则表示该对象不符合谓词中的标准，于是就会从最终的获取结果中剔除。

执行完 NSFetchedRequest 之后，获取到的结果会存放在 NSArray 里面，此时就可以按照自己的需要对这个数组里面的托管对象进行筛选了。可以用 NSArray 的

filteredArrayUsingPredicate 方法来筛选,也可以用 NSMutableArray 的 filterUsingPredicate 方法执行就地筛选。

以 Grocery Dude 程序为例,假设我们现在要把名称为 Coffee 的货品排除掉。在创建传递给 NSFetchedRequest 的 NSPredicate 时,我们可以指明 **name** 不等于字符串 Coffee。这个谓词写成代码就是 `name!=@"Coffee"`。由于谓词支持变量替换,所以我们可以像程序清单 2-7 中的粗体代码那样,等到运行期再向谓词传入字符串。构建谓词所需的逻辑有时比较简单,有时却相当复杂,这要根据需求来定。有关谓词的更多信息,请访问 <http://developer.apple.com/> 网站并搜索 **Predicate Programming Guide**。

程序清单2-7 AppDelegate.m文件中的demo方法（演示筛选功能）

```
NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Item"];

NSSortDescriptor *sort =
[NSSortDescriptor sortDescriptorWithKey:@"name" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sort]];

NSPredicate *filter =
[NSPredicate predicateWithFormat:@"name != %@", @"Coffee"];
[request setPredicate:filter];

NSArray *fetchedObjects =
[_coreDataHelper.context executeFetchRequest:request error:nil];
for (Item *item in fetchedObjects) {
    NSLog(@"Fetched Object = %@", item.name);
}
```

请按下列步骤修改 Grocery Dude,向其中添加谓词,以便筛选获取到的托管对象:

1. 修改 AppDelegate.m 文件的 demo 方法,用程序清单 2-7 中的粗体代码把原有的相关代码替换掉。

运行应用程序,并观察控制台中的日志,你应该会发现:在列出的货品名称中,已经没有 Coffee 这一项了。正常的运行结果如图 2-13 所示。

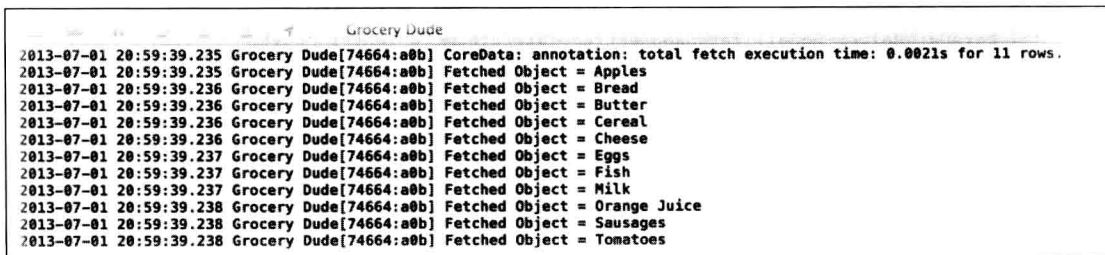


图 2-13 对获取到的托管对象进行排序和筛选,然后打印其名称

2.13.3 获取请求模板

假如每次获取托管对象时都要手工编写谓词格式确实很累人，幸好 Xcode 的 Data Model Designer 有预定义获取请求的功能。这些可复用的模板比谓词更容易配置，而且还能减少重复代码。只需根据应用程序的模型来操作一系列下拉列表框及文本框，即可配置好一份获取请求模板。但如果要自定义 AND、OR 这样的逻辑组合，那么这个模板就无法满足要求了，此时仍然需要通过代码来指定谓词。

请按下列步骤修改 Grocery Dude，以创建获取请求模板：

1. 选中 **Model.xcdatamodeld**。
2. 点击 **Editor > Add Fetch Request** 菜单项。
3. 把获取请求模板的名称设为 **Test**。
4. 点击 “+” 按钮来配置名为 **Test** 的获取请求模板，如图 2-14 所示。

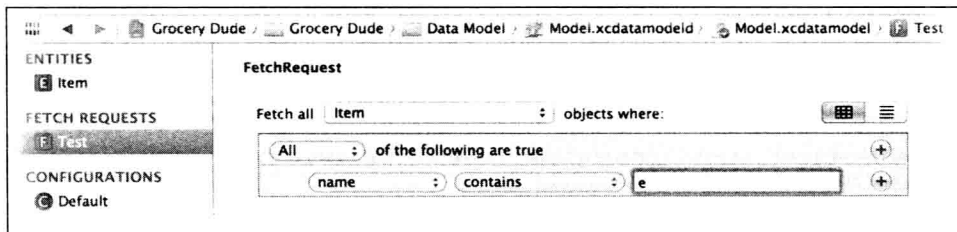


图 2-14 获取请求模板的配置界面

要想使用获取请求模板，需要先给托管对象模型发送消息，告诉它将要使用的模板叫什么名字。发送完消息之后，就可以在返回的 `NSFetchRequest` 上面操作了。由于这种获取请求是根据模板创建出来的，所以开发者无需通过向其发送谓词来执行筛选操作。假如想修改这种获取请求（比方说，要对其进行排序），那么必须先制作它的一份拷贝，这是因为获取请求模板是根据不可变的模型（immutable model 或 unchangeable model）创建出来的。相关代码如程序清单 2-8 所示。

程序清单 2-8 AppDelegate.m 文件中的 demo 方法（演示获取请求模板的用法）

```
NSFetchRequest *request =
[[[_coreDataHelper model] fetchRequestTemplateForName:@"Test"] copy];

NSSortDescriptor *sort =
[NSSortDescriptor sortDescriptorWithKey:@"name" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sort]];

NSArray *fetchedObjects =
[_coreDataHelper.context executeFetchRequest:request error:nil];
for (Item *item in fetchedObjects) {
    NSLog(@"Fetched Object = %@", item.name);
}
```

请按下列步骤修改 Grocery Dude，以便使用名为 Test 的获取请求模板：

1. 修改 AppDelegate.m 文件中的 demo 方法，用程序清单 2-8 中的代码替换原有的代码。程序清单 2-8 中的那行粗体代码是新加进来的，另外，修改过的代码中是没有谓词的。

运行应用程序，并观察控制台中的日志，你会发现这些托管对象均已按照名称排过序，而且只有名称中包含字母 e 的对象才会出现在控制台中，这些效果都是通过配置获取请求模板而实现出来的。正常的运行结果如图 2-15 所示。

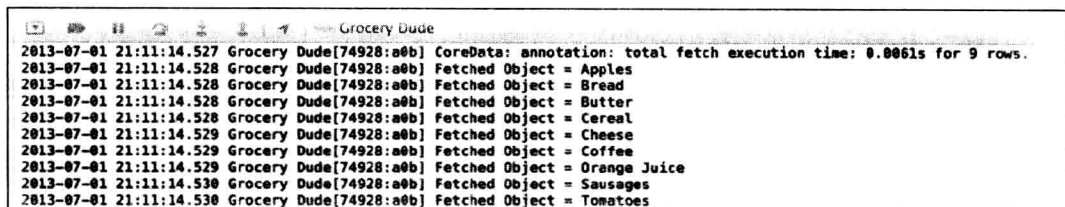


图 2-15 通过模板来筛选获取到的托管对象

不知你是否注意到了系统为这次 fetch 操作所生成的 SQL 语句。这条 SQL 语句的意思就是对获取到的各个 Item 进行筛选与排序（“获取”对应于语句中的“SELECT”；“筛选”对应于语句中的“WHERE”；“排序”对应于语句中的“ORDER BY”）。正常的运行效果如图 2-16 所示。

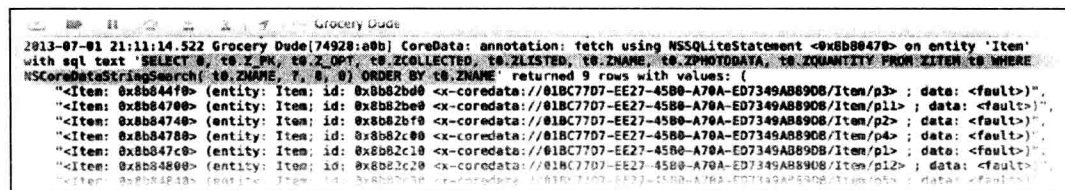


图 2-16 系统自动生成的 SQL 语句

2.14 删除托管对象

若想删除托管对象，只需在包含该对象的上下文中调用 deleteObject 或 deleteObjects 即可。请注意，此时对象并未永久删除，必须调用上下文的 save: 方法才能将其永久删去。相关代码如程序清单 2-9 所示。

程序清单2-9 AppDelegate.m文件中的demo方法（演示如何删除托管对象）

```
NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Item"];

NSArray *fetchedObjects =
[_coreDataHelper.context executeFetchRequest:request error:nil];
```

```
for (Item *item in fetchedObjects) {
    NSLog(@"Deleting Object '%@'", item.name);
    [_coreDataHelper.context deleteObject:item];
}
```

请按下列步骤修改 Grocery Dude，以删除所有对象：

1. 修改 demo 方法，用程序清单 2-9 中的代码替换掉原有代码。
2. 运行应用程序。
3. 按 home 键（如果是在 iOS 仿真器中运行，可以通过“**Shift + ⌘ + H**”组合键或 **Hardware > Home** 菜单项来模拟按键），以便将修改后的数据保存到上下文中。

按下 Home 键之后，就会在上下文上触发 save 操作，而此时请注意：控制台的日志里面会出现 SQL 语句，这些语句通过 DELETE 来删除数据库里的相关数据。相信你已经能体会到 Core Data 是如何自动操作后端 SQL 的了。在开始学习第 3 章之前，请先关闭 SQLDebug 调试选项，并把 demo 方法内的所有代码删掉。

2.15 小结

读者现在已经知道了如何按步骤来配置简单的托管对象模型。我们还讨论了实体及属性，笔者也给出了一些建议，告诉你如何为属性选定合适的数据类型。接下来，我们又讲解了如何插入、获取、筛选、排列及删除托管对象，并且介绍了获取请求模板的用法。为了使你能够更深刻地理解 Core Data 的工作原理，笔者在讲解的过程中还揭示了 Core Data 在后台所执行的一些操作。

2.16 习题

请你通过下列习题来尝试运用本章所学到的内容：

1. 插入一些新的托管对象，并设定其名称（name）与数量（quantity）。
2. 修改名为 **Test** 的获取请求模板，尝试一下其他各种筛选选项。
3. 用 **SQLite Database Browser** 打开 Grocery-Dude.sqlite 文件，并点击 **Database Structure** 按钮。注意到 ZITEM_ZNAME_INDEX 了吗？由于我们为名为 name 的属性勾选了 **Indexed** 选项，所以数据库里会保存这份索引。
4. 临时修改 demo 方法，把程序清单 2-10 中的代码放进去运行一下。看看各种数值数据类型的取值范围。

程序清单2-10 各种数值数据类型的取值范围

```
NSLog(@"Integer 16 Range: %d to %d", INT16_MIN, INT16_MAX);
NSLog(@"Integer 32 Range: %d to %d", INT32_MIN, INT32_MAX);
NSLog(@"Integer 64 Range: %lld to %lld", INT64_MIN, INT64_MAX);
```

```
NSLog(@"Float Range = %f to %f", -FLT_MAX, FLT_MAX);
NSLog(@"Double Range = %f to %f", -DBL_MAX, DBL_MAX);
NSLog(@"Decimal Range = %@ to %@",
      [NSDecimalNumber minimumDecimalNumber],
      [NSDecimalNumber maximumDecimalNumber]);

NSLog(@" Float 1/3 = %@", [NSNumber numberWithFloat:1.0f/3]);
NSLog(@" Double 1/3 = %@", [NSNumber numberWithDouble:1.0/3]);
NSLog(@"Decimal 1/3 = %@",
      [[NSDecimalNumber one] decimalNumberByDividingBy:
       [NSDecimalNumber decimalNumberWithString:@"3"]]);
```

托管对象模型的迁移

不创新者永不犯错。

——阿尔伯特·爱因斯坦

第2章介绍了托管对象模型的基础知识，但我们把内容局限在了一个实体及几个属性上面。按理说接下来就该向模型里面添加更多的内容了，但在执行修改之前，还必须完成一些准备步骤，以防应用程序因为这些改动而崩溃。本章就来介绍如何添加模型版本及模型映射，同时也会演示几种不同的迁移技术，供你在升级模型时选用。

3.1 修改托管对象模型

在应用程序的进化过程中，其托管对象模型也可能需要改变。对于一些比较简单的修改，诸如设定属性的默认值、设定验证规则、使用获取请求模板等，是可以直接实施的。而对于另外一些更为结构化的（structural）修改，则需先把持久化存储区迁移到新的模型版本才行。假如没有提供迁移数据所需的映射与设定，那么应用程序就会崩溃。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter02.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项，这样可以清除掉同名项目所残留的缓存。在学习本章的过程中，最好能用 iOS 仿真器来运行程序，因为这样更容易查看到 SQLite 数据库文件里的内容。

请按下列步骤修改 Grocery Dude，以引发模型不兼容错误：

1. 运行 Grocery Dude，确保程序用现有模型创建持久化存储区。你应该会在控制台日志里看到 Successfully added store 字样。
2. 在 Xcode 界面里选中 **Model.xcdatamodeld**。
3. 添加名为 **Measurement** 的新实体。
4. 选中 **Measurement** 实体，添加名为 **abc** 的属性，并将其类型设为 **String**。
5. 重新运行应用程序，并观察控制台输出的日志。你将会看到图 3-1 里的这个错误，它应该算是 Core Data 开发中最为常见的错误之一。假如没有发生该错误，请删除应用程序，并点击 **Product>Clean** 菜单项，然后从第 1 步开始再试一次。

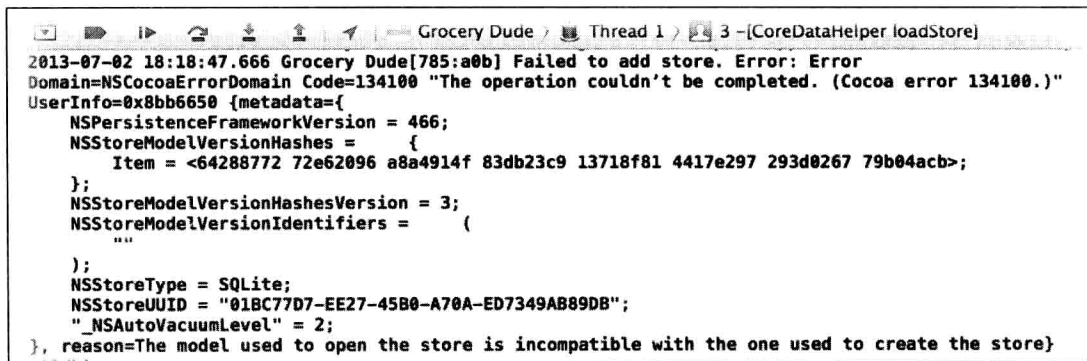


图 3-1 由于模型发生改变而导致与持久化存储区不兼容

对于处在开发初期的应用程序来说，这种崩溃算不上什么大问题，我们只需把程序删除了并重新运行一遍就好。删除之后，再次运行应用程序时，它会按照最新的模型来创建持久化存储区。这样一来，存储区就可以和模型相兼容了，于是应用程序也就不再崩溃了。但是，这样做也会失去存储区里原有的数据。对于已经在 App Store 上架的程序来说，这让人无法接受。有好几种办法都可以迁移现有的持久化存储区，而迁移路径则是由变更的复杂程度以及是否使用 iCloud 等因素来决定的。无论采用哪种迁移办法，我们都必须首先熟悉“模型版本控制”(model versioning)。

请按下列步骤修改 Grocery Dude，以便将模型恢复到修改前的状态：

1. 选定 **Model.xcdatamodeld**。
2. 删除 **Measurement** 实体。
3. 重新运行应用程序，这次应该不会崩溃了。

3.2 添加模型版本

为了不使应用程序像图 3-1 那样崩溃，我们需要在修改模型之前先创建新的模型版本。

添加新模型之后，就不应该再删除旧版的模型了。旧的模型有助于把原来的持久化存储区迁移到当前的模型版本。假如用户的设备上原来就没有持久化存储区，那么可以先不考虑模型版本控制问题，等到应用程序在 App Store 上架之后再说。

请按下列步骤修改 Grocery Dude，以便添加模型版本：

1. 选中 **Model.xcdatamodeld**。
2. 点击 **Editor > Add Model Version...** 菜单项。
3. 点击 **Finish** 按钮，将 **Model 2** 用作版本名称。

现在项目中应该会有两个版本的模型了，如图 3-2 所示。

Model 2.xcdatamodel 这个新模型的内容一开始便与 **Model.xcdatamodel** 完全相同，而开发者不经意间就会在错误的模型版本上进行修改。所以，为了防止这一情况，在编辑模型之前，应再三检查你所选定的模型是不是自己要编辑的那个版本。应该养成抓取快照的习惯，甚至可以在编辑模型之前把整个项目都备份起来。

请按下列步骤修改 Grocery Dude，以便重新引入 **Measurement** 实体：

1. 可以在执行修改之前先抓取快照或备份整个 Grocery Dude 项目。
2. 选定 **Model 2.xcdatamodel**。
3. 创建名为 **Measurement** 的新实体。
4. 选定 **Measurement** 实体，创建名叫 **abc** 的属性，并将其类型设为 **String**。

添加了新版模型之后，必须将其设为当前版本（current version），然后才能使应用程序使用它。

请按下列步骤修改 Grocery Dude，以便修改当前的模型版本：

1. 选定 **Model.xcdatamodeld**。
2. 点击 **View > Utilities > Show File Inspector** 菜单项（或按“**Option + ⌘ + 1**”组合键）。
3. 将 **Current Model Version** 设为 **Model 2**，如图 3-3 所示。

如果想正常运行应用程序，那么我们还必须配置好迁移选项，告诉 Core Data 应该如何迁移。要是现在就去运行应用程序的话，那自然还是会发生 Store is incompatible（存储区不兼容）错误。

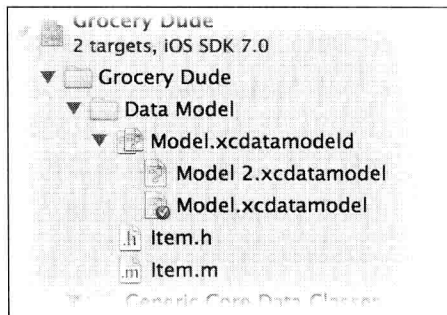


图 3-2 项目里有多个模型版本

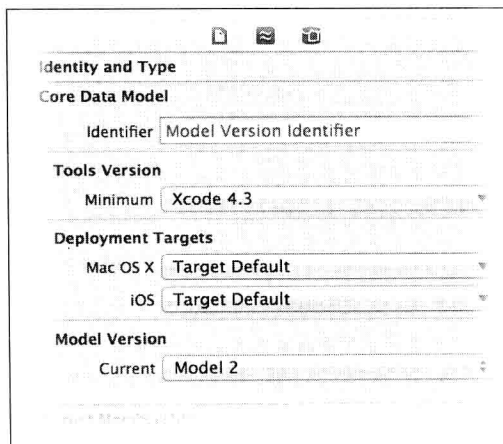


图 3-3 设定当前模型版本

3.3 轻量级的迁移方式

把新模型设为当前版本之后，必须迁移现有的持久化存储区，只有这样，才能正常使用新模型。这是因为，持久化存储区协调器会试着用新版的模型来打开原有的存储区，但由于原有的存储区是用旧版模型创建的，所以该操作会失败。在向 `NSPersistentStoreCoordinator` 添加存储区的时候，只需将下列选项放在 `NSDictionary` 里传过去，即可自动完成存储区的迁移工作：

- ❑ 如果传给 `NSPersistentStoreCoordinator` 的 `NSMigratePersistentStoresAutomaticallyOption` 是 YES，那么 Core Data 就会试着把低版本的（也就是与当前模型不兼容的）持久化存储区迁移到最新版本的模型。
- ❑ 如果传给 `NSPersistentStoreCoordinator` 的 `NSInferMappingModelAutomaticallyOption` 是 YES，那么 Core Data 就会试着以最为合理的方式自动推断出源模型实体（source model entity）中的某个属性到底对应于“目标模型实体”（destination model entity）中的哪一个属性。

把上述两个选项都打开并传给 `NSPersistentStoreCoordinator`，这种迁移方式就叫做轻量级迁移（**lightweight migration**），程序清单 3-1 中的粗体代码演示了该方式。我们修改了 `CoreDataHelper.m` 文件中的 `loadStore` 方法，在里面设定了这两个选项。请注意，假如在开发 Core Data 程序时还使用了 iCloud，那么只能采用这种迁移方式。

程序清单3-1 CoreDataHelper.m文件中的loadStore方法

```
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (_store) {return;} // Don't load store if it's already loaded
NSDictionary *options =
@{
    NSMigratePersistentStoresAutomaticallyOption:@YES
    ,NSInferMappingModelAutomaticallyOption:@YES
    ,NSSQLitePragmasOption: @{@"journal_mode": @"DELETE"}
};
NSError *error = nil;
_store = [_coordinator addPersistentStoreWithType:NSSQLiteStoreType
                                           configuration:nil
                                           URL:[self storeURL]
                                           options:options error:&error];

if (!_store) {NSLog(@"Failed to add store. Error: %@", error);abort();}
else        {NSLog(@"Successfully added store: %@", _store);}
```

请按下列步骤修改 Grocery Dude，以启用轻量级迁移：

1. 修改 `CoreDataHelper.m` 文件中的 `loadStore` 方法，用程序清单 3-1 里的代码把原有代码替换掉。笔者用粗体标出了改动的部分，以示强调。

2. 重新运行应用程序，这次应该就不会崩溃了。

从现在开始，只需把新模型设为当前版本并启用轻量级迁移，Core Data 就会无缝地完成迁移过程。

在演示其他迁移方式之前，我们需要生成一些测试用的数据。程序清单 3-2 里的代码可以根据 **Measurement** 实体来生成托管对象。

程序清单3-2 AppDelegate.m文件的demo方法（插入测试所需的Measurement数据）

```
- (void)demo {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}
    for (int i = 1; i < 50000; i++) {
        Measurement *newMeasurement =
            [NSEntityDescription insertNewObjectForEntityForName:@"Measurement"
                inManagedObjectContext:_coreDataHelper.context];

        newMeasurement.abc =
            [NSString stringWithFormat:@"-->> LOTS OF TEST DATA x%i",i];
        NSLog(@"Inserted %@",newMeasurement.abc);
    }
    [_coreDataHelper saveContext];
}
```

请按下列步骤修改 Grocery Dude，以便生成测试用的数据：

1. 根据 **Measurement** 实体创建 `NSManagedObject` 子类。创建步骤在第 2 章中讲过：首先选中实体，然后点击 **Editor > Create NSManagedObject Subclass...** 菜单项，并按提示操作。在保存类文件这个步骤中，别忘了勾选 targets 里的“Grocery Dude”。

2. 把 `#import "Measurement.h"` 添加到 AppDelegate.m 顶部。

3. 修改 AppDelegate.m 文件的 demo 方法，用程序清单 3-2 中的代码替换掉原有代码。

4. 运行一遍应用程序。这次它会向持久化存储区里插入大量测试数据，你可以从控制台的日志中观察到这一操作。根据电脑执行程序的快慢，该操作可能要花些时间。请耐心等待这些数据插入完毕。为了于稍后演示迁移的速度，我们现在必须在持久化存储区中放入大量数据。

只要运行一遍应用程序，持久化存储区里就会有测试数据了，所以现在无需再次启动应用程序，以免它又向存储区中重复添加数据。请注意，Items 这个表格视图里面还没有内容，因为我们还没有把它要显示的内容配置好。

下一步是重新配置 demo 方法，令其显示出持久化存储区里的一部分内容。程序清单 3-3 中的代码可以获取少量 **Measurement** 样例数据。注意，我们通过新的选项把获取到的结果数量限制为 50。从大的数据集中获取数据时，这个限制选项很有用处，如果能和排序操作结合起来就更好了，比方说，可以按某项标准列出排名前 50 的数据。

程序清单3-3 AppDelegate.m文件的demo方法（获取测试用的Measurement数据）

```

- (void)demo {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:@"Measurement"];
    [request setFetchLimit:50];
    NSError *error = nil;
    NSArray *fetchedObjects =
    [_coreDataHelper.context executeFetchRequest:request error:&error];

    if (error) {NSLog(@"%@", error);}
    else {
        for (Measurement *measurement in fetchedObjects) {
            NSLog(@"Fetched Object = %@", measurement.abc);
        }
    }
}
}

```

请按下列步骤修改 Grocery Dude，以防应用程序再次插入测试数据：

1. 修改 AppDelegate.m 文件中的 demo 方法，用程序清单 3-3 替换掉方法里原有的代码。
2. 运行应用程序。
3. 用 **SQLite Database Browser** 工具查看 Grocery-Dude.sqlite 文件的内容，第 2 章曾讲过该工具的用法。图 3-4 演示了正常的操作结果。

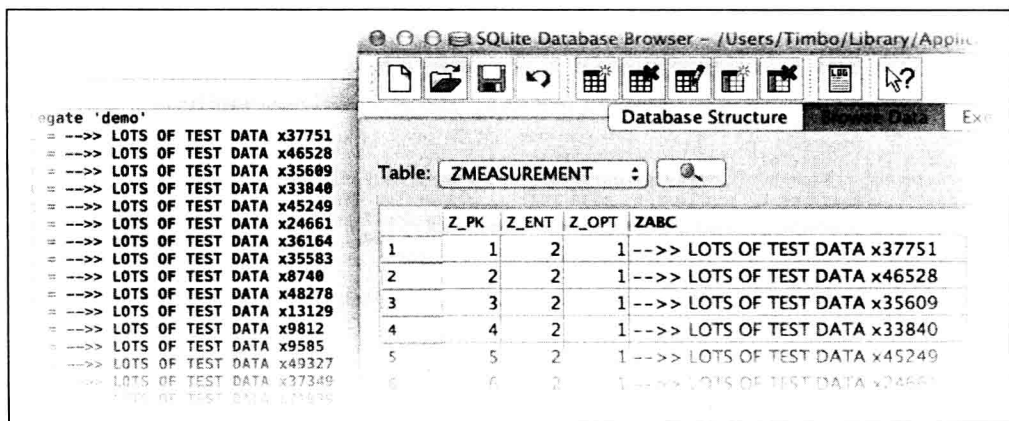


图 3-4 为本章下一节而准备的测试数据

在学习下一节之前，一定要先关掉 SQLite Database Browser。

3.4 默认的迁移方式

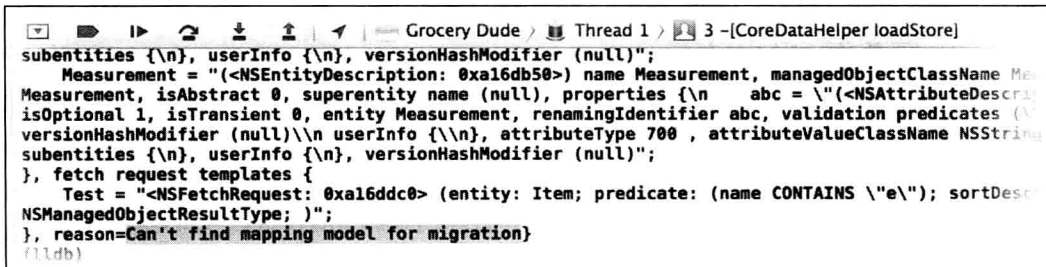
有时候我们需要比轻量级迁移更为精细的控制手段。比方说，我们要把 **Measurement** 实体替换成另外一个名叫 **Amount** 的实体，并且还想把 **Measurement** 实体中名叫 **abc** 的那个属性迁移到 **Amount** 实体中的 **xyz** 属性上面。**abc** 中已有的数据也要迁移到 **xyz** 属性。为了完成这些需求，开发者需要创建模型映射，以便手工指明映射关系。在添加持久化存储区时，即便 `NSInferMappingModelAutomaticallyOption` 选项设为 `YES`，Core Data 也还是会先检测有没有文件，如果有的话，那么在执行自动推断之前，它会先试着使用这个文件来迁移。在测试映射模型之前，建议先禁用该选项，这样才可以确定映射模型是不是已经付诸使用并且能够正常运作了。

请按下列步骤修改 Grocery Dude，以禁用自动化模型映射功能：

修改 `CoreDataHelper.m` 文件中的 `loadStore` 方法，把 `NSInferMappingModelAutomaticallyOption` 设为 `@NO`。

请按下列步骤修改 Grocery Dude，以便添加新模型，为从 **Measurement** 实体迁移到 **Amount** 实体做准备：

1. 可以先抓取一份快照或备份整个项目。
2. 根据 **Model 2** 版本来创建新版模型，将其命名为 **Model 3**。
3. 选中 **Model 3.xcdatamodel**。
4. 删除 **Measurement** 实体。
5. 新建 **Amount** 实体，并向其中添加类型为 `String` 的 **xyz** 属性。
6. 根据 **Amount** 实体创建 `NSManagedObject` 子类。在保存类文件这个步骤中，别忘了勾选 `targets` 中的 “Grocery Dude”。
7. 将 **Model 3** 设为当前模型版本。
8. 运行应用程序，目前它应该出错并崩溃。错误信息如图 3-5 所示。



```

Grocery Dude > Thread 1 > 3 -[CoreDataHelper loadStore]
subentities {\n}, userInfo {\n}, versionHashModifier (null)";
Measurement = "<NSEntityDescription: 0x16db50> name Measurement, managedObjectClassName Measurement, isAbstract 0, superentity name (null), properties {\n abc = \"<NSAttributeDescription: 0x16db50> name Measurement, isOptional 1, isTransient 0, entity Measurement, renamingIdentifier abc, validation predicates {\n versionHashModifier (null)\n userInfo {\n}, attributeType 700 , attributeValueClassName NSString\n}, fetch request templates {\n}, fetch request templates {\n}, userInfo {\n}, versionHashModifier (null)";
Test = "<NSFetchRequest: 0x16ddc0> (entity: Item; predicate: (name CONTAINS \"e\"); sortDescriptors: ());";
NSManagedObjectResultType; );";
}, reason=Can't find mapping model for migration)
(11db)

```

图 3-5 在不开启自动推断功能时，必须要有映射模型方能完成迁移工作

为了解决图 3-5 中的错误，我们需要创建映射模型，以指明字段之间的映射关系。具体到本例来说，就是要把旧模型中 **Measurement** 实体的 **abc** 属性迁移为新模型中 **Amount** 实体的 **xyz** 属性。

请按下列步骤修改 Grocery Dude，以添加新的映射模型：

1. 确保 **Data Model** 组处于选中状态[⊖]。
2. 点击 **File > New > File...** 菜单项。
3. 选择 **iOS > Core Data > Mapping Model**，并点击 **Next** 按钮。
4. 把 **Model 2.xcdatamodel** 选为 **Source Data Model**，并点击 **Next** 按钮。
5. 把 **Model 3.xcdatamodel** 选为 **Target Data Model**，并点击 **Next** 按钮。
6. 将 mapping model 的名称设为 **Model2toModel3**，并将其保存。
7. 确保 Targets 中的“Grocery Dude”处于勾选状态，然后点击 **Create** 按钮。
8. 选中 **Model2toModel3.xcmappingmodel**。

现在你将看到如图 3-6 所示的 model-mapping editor 界面。

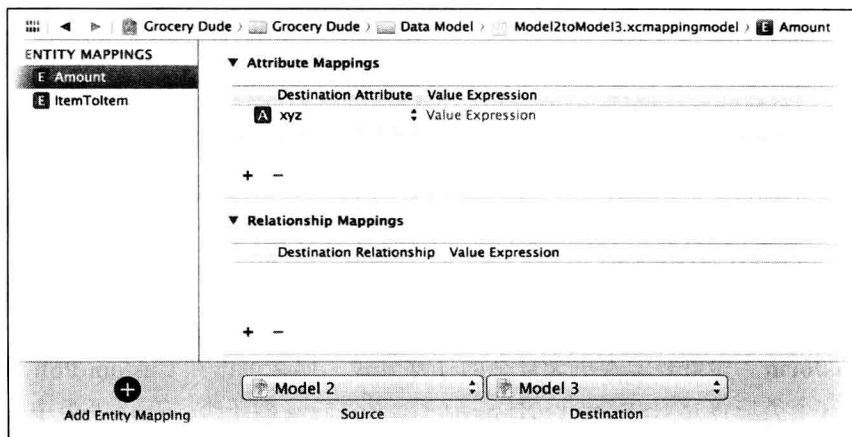


图 3-6 Xcode 的 model-mapping editor 界面

Xcode 目前呈现的这套映射是 Core Data 以最合理的方式推断出来的。在界面左方，应该会看到 **ENTITY MAPPINGS** 字样，它下面列出了源实体与目标实体之间的映射。通过图 3-6 我们应该可以看到，Core Data 已经推断出源 **Item** 实体对应于目标 **Item** 实体，而这个推断是合理的。实体映射时所采用的命名标准是 **SourceToDestination**（源实体名到目标实体名）。明白了这一点之后，我们就会发现，Amount 实体并没有与之对应的源实体，因为 Amount 没有出现在源模型里面。

请按下列步骤修改 Grocery Dude，以便将旧版模型的 Measurement 实体映射到新版模型的 Amount 实体：

1. 确保 **Model2toModel3.xcmappingmodel** 处于选中状态。
2. 在 ENTITY MAPPINGS 中选定 **Amount**。

⊖ “确保某物处于选中状态”或“确保某物受选”的意思就是“用鼠标点击某物，以将其选中”。为了尊重原著，译文将原书中的某些“ensure...is selected”直接对译为“确保……处于选中状态”。——译者注

3. 点击 **View > Utilities > Show Mapping Model Inspector** 菜单项（假如菜单里没有这一项，可以按“**Option + ⌘ + 3**”组合键），然后应该就会看到如图 3-7 所示的面板了。

4. 在 Entity Mapping 区域中，把 **Amount** 实体的 **Source** 设置成 **Measurement**。设置好的结果如图 3-7 所示。

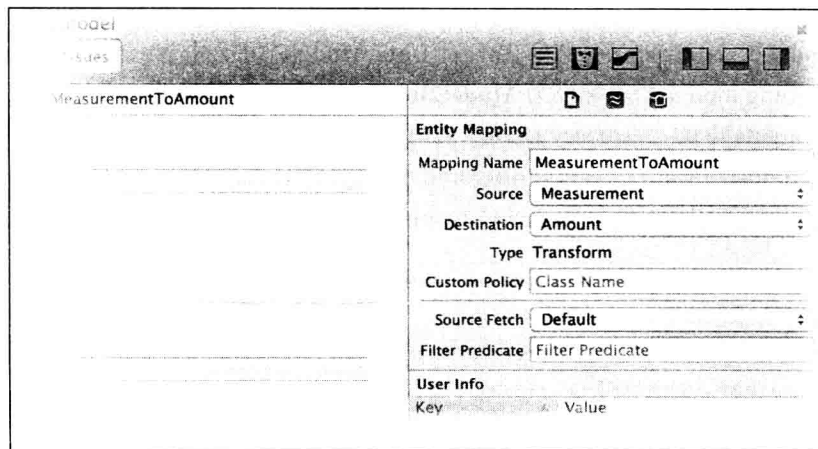


图 3-7 手动配置从 Measurement 实体到 Amount 实体的映射

由于我们把 **Measurement** 选为源实体，而把 **Amount** 选为目标实体，所以 Mapping Name 这一栏就自动变成了 **MeasurementToAmount**。此外，映射的 Type（类型）也从 **Add** 变成了 **Transform**。如果要实现更为复杂的迁移方式，那么可以在 Custom Policy 文本框中输入类名，这个类应该是 `NSEntityMigrationPolicy` 的子类。在该子类中，可以通过覆写 `createDestinationInstancesForSourceInstance` 方法而操作待迁移的数据。比方说，可以拦截 **abc** 这个属性的值，将其中每个单词的首字母改为大写，然后再把修改过的值迁移到 **xyz** 属性。

图 3-7 底部的 Source Fetch 选项可通过谓词（在 Filter Predicate 文本框中输入）限定迁移过来的数据量。假如只想把旧数据中的一部分迁移过来，那么这个选项就很有用了。此处的谓词格式与通常代码中编写的谓词相似，只不过要用 `$source` 变量来表示源数据。比方说，如果想把 **abc** 属性为 `nil` 的源数据排除掉，那么可将谓词写成 `$source.abc!=nil`。

在前述的图 3-6 中，选定 ENTITY MAPPINGS 字样下方的 **ItemToItem** 实体，并观察属性映射中的内容，会看到目标实体中的每个属性都设置有对应的 **Value Expression**。现在再来查看 **MeasurementToAmount** 实体的映射，会发现 **xyz** 这个 Destination 属性并没有设置 Value Expression。这就意味着 **xyz** 属性目前还没有对应的 Source 属性，需要按照 ItemToItem 实体映射中的那种格式，给它设置一条 Value Expression。我们一开始提出的需求是把 **abc** 属性映射到 **xyz** 属性，所以接下来就按照这个需求配置 Value Expression。

请按下列步骤修改 Grocery Dude，给名为 **xyz** 的 Destination 属性设置适当的 Value Expression：

1. 在 **MeasurementToAmount** 的实体映射界面中，把 **xyz** 这个 **Destination** 属性的 **Value Expression** 设置为 **\$source.abc**。

迁移模型虽然已经配置好了，但 **demo** 方法仍然会从 **Measurement** 实体获取数据，而在新模型中，是没有这个实体的。

请按下列步骤修改 **Grocery Dude**，令 **demo** 方法使用 **Amount** 实体而非 **Measurement** 实体：

1. 把 **AppDelegate.m** 文件顶部的 **#import "Measurement.h"** 替换为 **#import "Amount.h"**。

2. 修改 **AppDelegate.m** 文件的 **demo** 方法，用程序清单 3-4 中的代码替换原有代码。原来的代码是获取一小部分 **Measurement** 样例数据，而这段代码也与之相似，它是获取一小部分 **Amount** 数据。

3. 运行应用程序。由于要迁移数据，所以加载屏幕的显示时间可能会稍微长一些，具体情况与电脑速度有关。

程序清单3-4 AppDelegate.m文件中的demo方法（获取测试用的Amount数据）

```

NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Amount"];
[request setFetchLimit:50];
NSError *error = nil;
NSArray *fetchedObjects =
[_coreDataHelper.context executeFetchRequest:request error:&error];

if (error) {NSLog(@"%@", error);}
else {
    for (Amount *amount in fetchedObjects) {
        NSLog(@"Fetched Object = %@", amount.xyz);
    }
}

```

只要迁移过程顺利完成，程序就不会崩溃，你应该会在控制台的日志中看到如图 3-8 所示的信息。

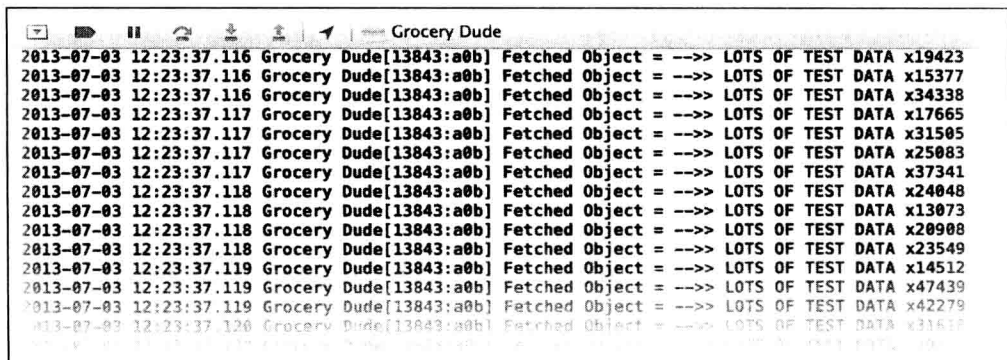


图 3-8 从顺利迁移至新版本的模型中获取到的数据

为了验证迁移后的数据是否已经保存到持久化存储区，我们可以用第2章中讲过的办法来查看 Grocery-Dude.sqlite 文件的内容。正确的结果应该如图3-9所示，其中多出了名为 ZAMOUNT 的表（这张表对应于 Amount 实体），旧的 Measurement 实体里的数据现已出现在这张表中。

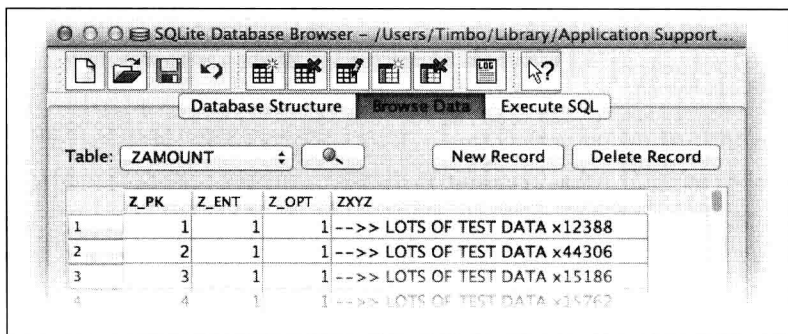


图 3-9 顺利迁移到新版本的模型

在学习下一节之前，一定要先关掉 SQLite Database Browser。

3.5 通过迁移管理器来迁移数据

除了通过 NSPersistentStoreCoordinator 来迁移存储区之外，还可以采用迁移管理器来做。迁移管理器可以使开发者全权掌控迁移过程中创建的文件，从而令他们能够按自己的方式来灵活处理迁移中的各种问题。使用迁移管理器的一个好处就是可以向用户报告迁移进度，使用户知道应用程序哪次会启动得比较慢一些，所以需要耐心等待。虽说迁移过程理应执行得非常快才对，但当数据库比较大、变动比较复杂时，迁移过程就需要耗费一定的时间了。为了使用户界面保持流畅，迁移过程必须在后台线程里执行。只有这样做，用户界面才能反应灵敏，并能把最新动态提供给用户。实现数据迁移的难点在于如何防止用户在迁移过程中操作应用程序。由于此时数据尚未准备好，所以我们必须做这个限制，否则用户就会对着黑屏不知所措。

请按下列步骤修改 Grocery Dude，以便配置 Migration 视图控制器（View Controller）：

1. 选定 **Main.storyboard**。
2. 向故事板（storyboard）中拖放一个新的 **View Controller**，将它摆在现有的 Navigation 控制器上方。
3. 向这个新的视图控制器中拖放 **Label** 与 **Progress View** 控件。
4. 把 **Progress View** 放在视图控制器正中，并将 **Label** 放在 **Progress View** 上方。
5. 按图 3-10 拓宽 **Label** 与 **Progress View** 控件，使其宽度与视图控制器相符。
6. 把 **Label** 的文本改为 **Migration Progress 0%**，并将其居中（**Centered**），如图 3-10 左侧所示。

7. 把 **Progress View** 的进度 (progress) 设为 0。

8. 选中视图控制器，然后在 **Identity Inspector** 界面（可以按 “**Option + ⌘ + 3**” 组合键调出该界面）中把视图控制器的 **Storyboard ID** 设为 migration。

9. 点击 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints in View Controller** 菜单项。最终结果如图 3-10 所示。

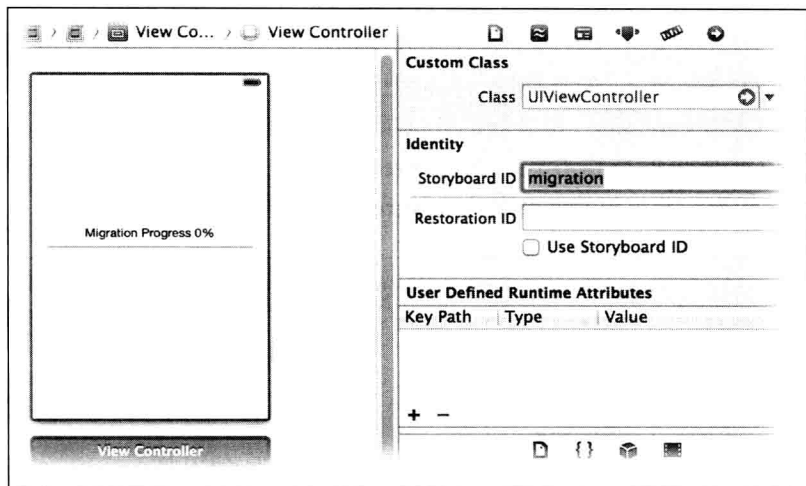


图 3-10 Migration 视图控制器

由于 Migration 视图控制器里的 UILabel 控件及 UIProgressView 控件需要在迁移过程中更新，所以我们需要一种能在代码中使用这两个控件的方式。为此，我们新建 UIViewController 的子类，并将其命名为 MigrationVC。

请按下列步骤修改 Grocery Dude，以便将 MigrationVC 类添加到新的组中：

1. 在现有的 **Grocery Dude** 组上面右击鼠标，选择 **New Group**。

2. 把新组的名字改为 **Grocery Dude View Controllers**。

3. 选中 **Grocery Dude View Controllers** 组。

4. 点击 **File > New > File...** 菜单项。

5. 新建 **iOS > Cocoa Touch > Objective-C class**，并点击 **Next** 按钮。

6. 把 **Subclass of** 设为 UIViewController，并把 **Class** 名称设为 MigrationVC，然后点击 **Next** 按钮。

7. 确保 Targets 中的 “Grocery Dude” 处于勾选状态，然后点击 Create，在 Grocery Dude 项目的文件夹中创建这个类。

8. 选中 **Main.storyboard**。

9. 将 Migration 视图控制器选中，在 **Identity Inspector** 界面（可按 “**Option + ⌘ + 3**” 组合键调出该界面）里把 **Custom Class** 设为 MigrationVC。该选项和刚才设置的 **Storyboard ID** 都位于同一界面中。

10. 点击 **View > Assistant Editor > Show Assistant Editor** 菜单项（或按“**Option + ⌘ + Return**”组合键），把 **Assistant Editor** 界面显示出来。

11. 此时 **Assistant Editor** 界面中应该就会自动显示出 MigrationVC.h 文件了。图 3-11 右上角是该界面的样貌。如果显示的不是这个文件，那可以先把 migration 视图控制器选中，然后点击界面上方的 **Manual** 或 **Automatic**，再选择 MigrationVC.h。

12. 点击 **Control** 键并按住鼠标左键不放，从显示迁移进度的 Label 控件开始沿直线拖到 MigrationVC.h 代码的 @end 上方。松开鼠标左键之后，会弹出一个对话框，该对话框中列出了类型为 UILabel 的新特性，我们把该特性的 **Name** 设为 **label**，并确保 **Storage** 是 **Strong**，然后点击 **Connect**。配置好的各选项如图 3-11 所示。

13. 按第 12 步所讲的操作方式，把 Progress View（进度视图）同 UIProgressView 类型的特性链接起来，并将该特性命名为 **progressView**。

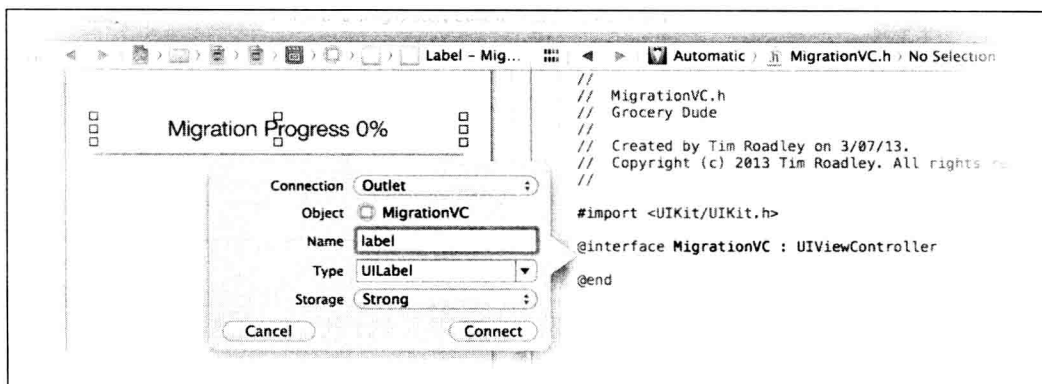


图 3-11 把故事板中的控件同 MigrationVC.h 中的特性链接起来

为了向用户报告迁移进度，我们需要在 CoreDataHelper.h 文件中声明指向 migration 视图控制器的指针。

请按下列步骤修改 Grocery Dude，以添加新特性：

1. 点击 **View > Standard Editor > Show Standard Editor** 菜单项或按“**⌘ + Return**”组合键，把 **Standard Editor** 界面显示出来。

2. 把 #import "MigrationVC.h" 添加到 CoreDataHelper.h 文件顶部。

3. 在 CoreDataHelper.h 文件的现有特性下方添加 @property(n nonatomic, retain) MigrationVC* migrationVC;。

如果采用手动方式迁移数据，那就得在每次启动应用程序时判断数据是否需要迁移。为了做出该判断，我们需要知道存储区的 URL，以便检查系统里是不是有这个存储区。如果有的话，那还要把存储区里的“模型元数据”（model metadata）与新的模型相比较，并根据比较的结果来判断新模型是否与现有的存储区相兼容。假如不兼容，那就要迁移数据了。把刚才说的这段逻辑写成代码，就得到了程序清单 3-5 中的 isMigrationNecessary-

ForStore 方法。

程序清单3-5 CoreDataHelper.m文件中的isMigrationNecessaryForStore方法

```
#pragma mark - MIGRATION MANAGER
- (BOOL)isMigrationNecessaryForStore:(NSURL*)storeUrl {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }
    if (![NSFileManager defaultManager] fileExistsAtPath:[self storeURL].path]) {
        if (debug==1) {NSLog(@"SKIPPED MIGRATION: Source database missing.");}
        return NO;
    }
    NSError *error = nil;
    NSDictionary *sourceMetadata =
        [NSPersistentStoreCoordinator metadataForPersistentStoreOfType:NSSQLiteStoreType
         URL:storeUrl error:&error];
    NSManagedObjectModel *destinationModel = _coordinator.managedObjectModel;
    if ([destinationModel isConfiguration:nil
        compatibleWithStoreMetadata:sourceMetadata]) {
        if (debug==1) {
            NSLog(@"SKIPPED MIGRATION: Source is already compatible");}
        return NO;
    }
    return YES;
}
```

请按下列步骤修改 Grocery Dude，以便实现新的 MIGRATION MANAGER 部分：

1. 将程序清单 3-5 中的代码添加到 CoreDataHelper.m 文件底部，并将其放在 @end 语句之前。

假如已经确定要迁移数据，那么接下来就该执行迁移了。此过程分为三步，程序清单 3-6 中的注释写明了这三个步骤。

程序清单3-6 CoreDataHelper.m文件中的migrateStore方法

```
- (BOOL)migrateStore:(NSURL*)sourceStore {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }

    BOOL success = NO;
    NSError *error = nil;

    // STEP 1 - Gather the Source, Destination and Mapping Model
    NSDictionary *sourceMetadata = [NSPersistentStoreCoordinator
        metadataForPersistentStoreOfType:NSSQLiteStoreType
        URL:sourceStore
        error:&error];

    NSManagedObjectModel *sourceModel =
        [NSManagedObjectModel mergedModelFromBundles:nil
        forStoreMetadata:sourceMetadata];
```

```

NSManagedObjectModel *destinModel = _model;

NSMappingModel *mappingModel =
[NSMappingModel mappingModelFromBundles:nil
                 forSourceModel:sourceModel
                 destinationModel:destinModel];

// STEP 2 - Perform migration, assuming the mapping model isn't null
if (mappingModel) {
    NSError *error = nil;
    NSMigrationManager *migrationManager =
[NSMigrationManager alloc] initWithSourceModel:sourceModel
                             destinationModel:destinModel];
    [migrationManager addObserver:self
                     forKeyPath:@"migrationProgress"
                     options:NSKeyValueObservingOptionNew
                     context:NULL];

    NSURL *destinStore =
[[self applicationStoresDirectory]
 URLByAppendingPathComponent:@"Temp.sqlite"];

    success =
[migrationManager migrateStoreFromURL:sourceStore
                             type:NSSQLiteStoreType options:nil
                             withMappingModel:mappingModel
                             toDestinationURL:destinStore
                             destinationType:NSSQLiteStoreType
                             destinationOptions:nil
                             error:&error];

    if (success) {
        // STEP 3 - Replace the old store with the new migrated store
        if ([self replaceStore:sourceStore withStore:destinStore]) {
            if (debug==1) {
                NSLog(@"SUCCESSFULLY MIGRATED %@ to the Current Model",
                      sourceStore.path);
                [migrationManager removeObserver:self
                          forKeyPath:@"migrationProgress"];
            }
        }
    }
    else {
        if (debug==1) {NSLog(@"FAILED MIGRATION: %@",error);}
    }
}
else {
    if (debug==1) {NSLog(@"FAILED MIGRATION: Mapping Model is null");}
}

return YES; // indicates migration has finished, regardless of outcome
}

```

STEP 1 (第一步) 用于收集执行数据迁移所需的信息, 这些信息分别是:

- ❑ 源模型, 也就是通过 `NSPersistentStoreCoordinator` 的 `metadataForPersistentStoreOfType` 方法从持久化存储区里获取到的元数据。
- ❑ 目标模型, 也就是 `_model` 实例变量。
- ❑ 映射模型, 该模型由系统自动决定, 开发者只需把 `nil` 当做 `mappingModelFromBundles:forSourceModel:destinationModel:` 方法的第一个参数, 并把源模型和目标模型也一并传过去即可。

STEP 2 (第二步) 就是实际的迁移过程。我们先用源模型与目标模型创建 `NSMigrationManager` 实例, 然后在调用 `migrateStoreFromURL` 之前, 还需把目标存储区准备好。该目标存储区只是个为迁移而设的临时存储区。

STEP 3 (第三步) 只有在顺利完成迁移时才会触发。`replaceStore` 方法用于在迁移完成后清理旧的存储区。执行完第二步之后, 目标位置上就会出现一份新的存储区了, 但是, 我们还必须把这个迁移过来的新存储区放回到原来的位置上, 并且要把它文件名起得和旧存储区一样, 唯有如此, `Core Data` 才能使用这个新存储区。为了使用新迁移好的存储区, 我们需要把旧存储区删掉, 并将新存储区放到旧存储区的位置上。当开发自己的项目时, 也可以在删除旧存储区之前先把它备份到某处。是否需要备份, 由你自己决定, 如果真要备份, 那可能得稍微修改一下 `replaceStore` 方法。备份旧存储区会导致应用程序在迁移过程中对存储量的需求翻倍。

当迁移进度有变化时, 系统会调用 `observeValueForKeyPath` 方法, 而我们可以通过该方法把目前的迁移进度告知用户。`migrationManager` 的 `migrationProgress` 特性一旦改变, 我们就可通过该方法来更新 `migrationVC`。

程序清单 3-7 列出了 `observeValueForKeyPath` 及 `replaceStore` 方法的代码。

程序清单3-7 `CoreDataHelper.m`中的`observeValueForKeyPath`与`replaceStore`方法

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

    if ([keyPath isEqualToString:@"migrationProgress"]) {
        dispatch_async(dispatch_get_main_queue(), ^{

            float progress =
                [[change objectForKey:NSKeyValueChangeNewKey] floatValue];
            self.migrationVC.progressView.progress = progress;
            int percentage = progress * 100;
            NSString *string =
                [NSString stringWithFormat:@"Migration Progress: %i%%",
                                             percentage];

            NSLog(@"%@", string);
        });
    }
}
```



```

        self.migrationVC.label.text = string;
    });
}
}
- (BOOL)replaceStore:(NSURL*)old withStore:(NSURL*)new {

    BOOL success = NO;
    NSError *Error = nil;
    if ([[NSFileManager defaultManager]
        removeItemAtURL:old error:&Error]) {

        Error = nil;
        if ([[NSFileManager defaultManager]
            moveItemAtURL:new toURL:old error:&Error]) {
            success = YES;
        }
        else {
            if (debug==1) {NSLog(@"FAILED to re-home new store %@", Error);}
        }
    }
    else {
        if (debug==1) {
            NSLog(@"FAILED to remove old store %@: Error:%@", old, Error);
        }
    }
    return success;
}
}

```

请按下列步骤修改 Grocery Dude，以继续实现 MIGRATION MANAGER 部分：

1. 把程序清单 3-7 里的代码添加到 CoreDataHelper.m 文件 MIGRATION MANAGER 部分的底部，并放在 @end 上方，然后按同样方式把程序清单 3-6 里的代码也加进去。

为了在后台通过 migrationManager 来迁移数据，我们需要使用程序清单 3-8 中的方法。

程序清单3-8 CoreDataHelper.m的performBackgroundManagedMigrationForStore方法

```

- (void)performBackgroundManagedMigrationForStore:(NSURL*)storeURL {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    // Show migration progress view preventing the user from using the app
    UIStoryboard *sb = [UIStoryboard storyboardWithName:@"Main" bundle:nil];
    self.migrationVC =
        [sb instantiateViewControllerWithIdentifier:@"migration"];
    UIApplication *sa = [UIApplication sharedApplication];
    UINavigationController *nc =
        ([UINavigationController]sa.keyWindow.rootViewController);
    [nc presentViewController:self.migrationVC animated:NO completion:nil];
}

```

```

// Perform migration in the background, so it doesn't freeze the UI.
// This way progress can be shown to the user
dispatch_async(
dispatch_get_global_queue(
DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), ^{
    BOOL done = [self migrateStore:storeURL];
    if(done) {
        // When migration finishes, add the newly migrated store
        dispatch_async(dispatch_get_main_queue(), ^{
            NSError *error = nil;
            _store =
            [_coordinator addPersistentStoreWithType:NSSQLiteStoreType
                                             configuration:nil
                                             URL:[self storeURL]
                                             options:nil
                                             error:&error];

            if (!_store) {
                NSLog(@"Failed to add a migrated store. Error: %@",
                    error);abort();}
            else {
                NSLog(@"Successfully added a migrated store: %@",
                    _store);}
            [self.migrationVC dismissViewControllerAnimated:NO
                                                         completion:nil];
            self.migrationVC = nil;
        });
    }
});
}
}

```

performBackgroundManagedMigrationForStore 方法用故事板标识符来实例化视图控制器，并把它展示给用户。用户的操作由这个视图来接受，而我们则可以开始迁移数据了。migrateStore 方法会在后台线程中执行。等迁移完数据，我们就像平常那样通过 _coordinator 来添加存储区，并把显示迁移进度所用的 view 关闭，然后，应用程序就可以照常往下运行了。

请按下列步骤修改 Grocery Dude，以继续实现 MIGRATION MANAGER 部分：

1. 把程序清单 3-8 里的代码添加到 CoreDataHelper.m 文件 MIGRATION MANAGER 部分的底部，并放在 @end 语句上方。

检测是否需要执行数据迁移的最佳时机应该是在把存储区添加到 _coordinator 前的那一刻。为了安排好这项检测，我们需要修改 CoreDataHelper.m 文件中的 loadStore 方法。如果真的要迁移，那么迁移操作就会在此刻触发。相关代码如程序清单 3-9 所示。

程序清单3-9 CoreDataHelper.m文件中的loadStore方法

```

if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

```

```

if (!_store) {return;} // Don't load store if it's already loaded

BOOL useMigrationManager = YES;
if (useMigrationManager &&
    [self isMigrationNecessaryForStore:[self storeURL]]) {
    [self performBackgroundManagedMigrationForStore:[self storeURL]];
} else {
    NSDictionary *options =
    @{
        NSMigratePersistentStoresAutomaticallyOption:@YES
        ,NSInferMappingModelAutomaticallyOption:@NO
        ,NSSQLitePragmasOption: @{@"journal_mode": @"DELETE"}
    };
    NSError *error = nil;
    _store = [_coordinator addPersistentStoreWithType:NSSQLiteStoreType
                                                configuration:nil
                                                URL:[self storeURL]
                                                options:options
                                                error:&error];

    if (!_store) {
        NSLog(@"Failed to add store. Error: %@", error); abort();
    }
    else
        {NSLog(@"Successfully added store: %@", _store);}
}

```

请按下列步骤修改 Grocery Dude，以完成本节范例：

1. 修改 CoreDataHelper.m 文件中的 loadStore 方法，用程序清单 3-9 里的代码把原有代码替换掉。
2. 基于 **Model 3**，添加名为 **Model 4** 的模型版本。
3. 选定 **Model 4.xcdatamodel**。
4. 删除 **Amount** 实体。
5. 新增名为 **Unit** 的实体，并添加名为 **name** 的字符串类型属性。
6. 根据 **Unit** 实体来创建 NSManagedObject 子类。在保存类文件的这一步里，别忘了勾选 Targets 中的 “Grocery Dude”。
7. 将 **Model 4** 设为当前模型。
8. 以 **Model 3** 为源模型，以 **Model 4** 为目标模型，新建映射模型。在保存映射模型文件的这一步里，别忘了勾选 Targets 中的 “Grocery Dude”，然后，把这个模型存为 **Model3toModel4**。
9. 选定 **Model3toModel4.xcmappingmodel**。
10. 选定 ENTITIES MAPPINGS 中的 **Unit**。
11. 将 **Unit** 实体的 **Source** 设为 **Amount**，并给名为 **name** 的 Destination 属性设定 **Value Expression**，将这个 Value Expression 写成 **\$source.xyz**。此时 ENTITIES MAPPINGS 中的 Unit 实体应该会自动改名为 **AmountToUnit**，如图 3-12 所示。

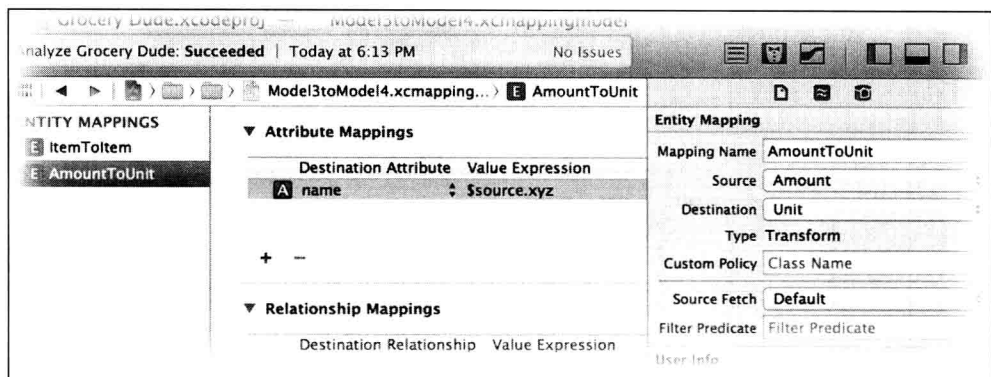


图 3-12 为从 Amount 实体到 Unit 实体的迁移而配置映射模型

现在基本上已经可以开始执行迁移了，不过 demo 方法里的获取请求仍然在使用旧的 Amount 实体。

请按下列步骤修改 Grocery Dude，使 demo 方法不再获取 Amount 实体，而是改为获取 Unit 实体：

1. 把 AppDelegate.m 文件顶部的 #import "Amount.h" 替换成 #import "Unit.h"。
2. 修改 AppDelegate.m 文件的 demo 方法，用程序清单 3-10 中的代码替换原有代码。新的代码只从持久化存储区里获取 50 个 Unit 对象。

程序清单3-10 AppDelegate.m文件中的demo方法（获取测试所用的Unit数据）

```

NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Unit"];
[request setFetchLimit:50];
NSError *error = nil;
NSArray *fetchedObjects =
[_coreDataHelper.context executeFetchRequest:request error:&error];

if (error) {NSLog(@"%@", error);}
else {
    for (Unit *unit in fetchedObjects) {
        NSLog(@"Fetched Object = %@", unit.name);
    }
}

```

迁移管理器终于实现好了！运行应用程序，仔细观察设备屏幕！你眼前会出现 Migration Progress 界面，它会显示数据的迁移进度。同时，这个进度也会出现在控制台里。

请用第 2 章讲过的办法来查看 Grocery-Dude.sqlite 文件中 ZUNIT 表的内容。正常的结果如图 3-14 所示。假如你已把默认的日志记录模式关闭，但 Stores 目录里却还有 -wal 文件，那么请点击 **Product > Clean** 菜单项并重新运行应用程序，然后再次查看 sqlite 文件。

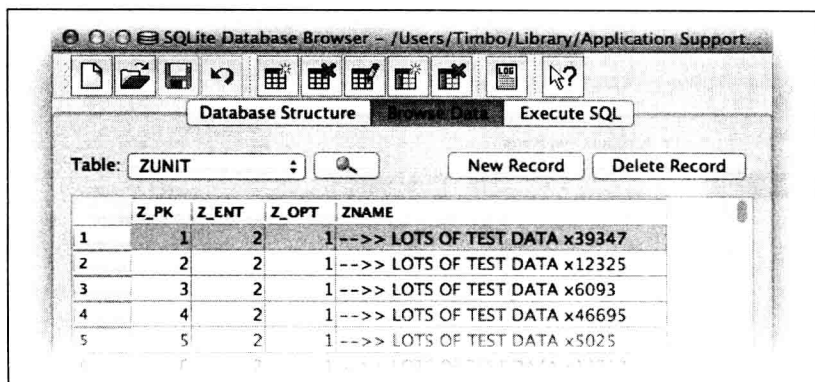


图 3-13 显示在设备屏幕和控制台里的数据迁移进度

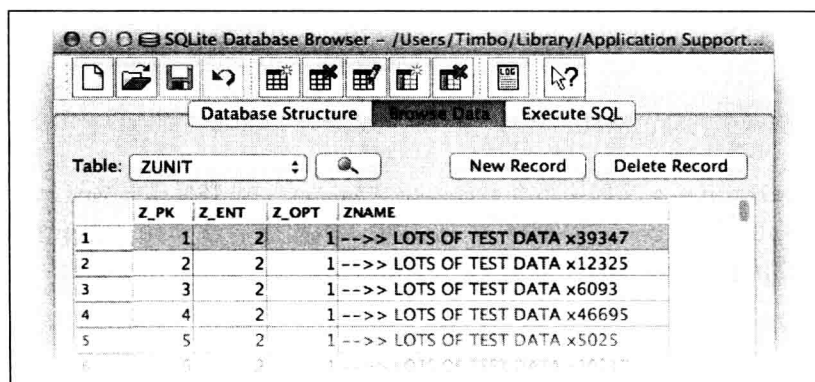


图 3-14 通过迁移管理器迁移过来的新数据

操作结果要是和图 3-14 一样的话，那你真的太棒了，因为你已把三种模型迁移方式全部实现出来了！本书接下来的内容要使用轻量级迁移方式，所以现在必须重新启用它。

请按下列步骤修改 Grocery Dude，以重新启用轻量级迁移方式：

1. 修改 CoreDataHelper.m 文件的 loadStore 方法，把 NSInferMappingModel-AutomaticallyOption 选项改为 @YES。
2. 修改 CoreDataHelper.m 文件的 loadStore 方法，把 useMigrationManager 设为 NO。
3. 删除 AppDelegate.m 文件 demo 方法中的所有代码。

旧的映射模型以及根据旧实体所创建出来的 NSManagedObject 子类现在都已经没有用处了。虽说也可以把它们删掉，但为了便于日后查阅，我们还是将其留在项目之中吧。

3.6 小结

本章讲解了三种迁移方式，它们分别是轻量级迁移方式、默认迁移方式以及使用迁移

管理器来显示进度的迁移方式。现在你应该能够根据自己应用程序的实际需求选出一种恰当的迁移方式了。不过要记住：启用了 iCloud 的 Core Data 应用程序只能使用轻量级迁移方式。在多次修改模型之后，你也应该对添加模型版本的过程非常熟悉了。

3.7 习题

请在所学内容的基础之上完成下列试验：

1. 把当前模型版本设为 **Model 3**，并运行应用程序。这次系统应该不会崩溃，因为它会自动推断数据的降级过程。请注意：系统之所以会自动推断，是因为我们刚才把 `NSInferMappingModelAutomaticallyOption` 重新启用了。但是在实际的程序开发中，为了把属性之间的映射关系处理好，需要配置名为 **Model4toModel3** 的映射模型。

2. 查看 `Grocery-Dude.sqlite` 文件中的 ZAMOUNT 表，你会发现一个严重的问题：原来的数据哪儿去了？由于没配置映射模型，所以 ZUNIT 数据在降级的过程中丢失了！

3. 把当前模型设为 **Model 4**，并修改 `CoreDataHelper.m` 文件的 `loadStore` 方法，把 `useMigrationManager` 设为 YES，以便重新启用迁移管理器。

4. 运行应用程序，这次你又会目睹一遍手动迁移数据的过程，不过它运行得特别快，因为存储区里面没有数据。在继续学习下一章之前，请先把 `useMigrationManager` 设为 NO。



Chapter 4

第 4 章

托管对象模型的扩展

黑洞是上帝拿 0 做除数所制造的神迹。

——阿尔伯特·爱因斯坦

我们在第 3 章中讲解了如何通过版本管理、映射及迁移等技术来改变模型，而这一章将会在现有知识的基础上，再深入介绍一些与模型变更有关的新内容。这次不再局限于一两个孤立的实体了，而是要在实体间建立关系，并且本章末尾还会讲到实体继承。这一章也要讨论 Delete 规则，同时还会讲解某些规则给数据验证所带来的影响。其后，你将学到怎样处理数据验证中所发生的错误，以及在非常糟糕的情况下如何优雅地终止应用程序并告知用户错误码，使用户可将其回报给开发者。

4.1 关系

关系是用来链接实体的。在托管对象模型中使用关系，就相当于引入了一种强大的手段，用以把实体形式的数据中某些逻辑区域连接起来。使用关系可大幅降低数据库对容量的需求。假如不使用关系，那么同样的数据就可能会在多个实体中重复出现，而使用了关系之后，只需在存储区里保存一份数据即可，原有的重复数据可以用关系来取代，该关系就相当于指向这份数据的指针。消除重复确实是关系的一项优势，但其真正强大的地方则在于：它可以在复杂的数据类型之间建立连接。

以 Grocery Dude 模型中的 Item 实体与 Unit 实体为例。Item 实体的实例用来表示可以添加到购物清单里的货品，比方说：

- ❑ Chocolate (巧克力)
- ❑ Potatoes (土豆)
- ❑ Milk (牛奶)

而 Unit 实体的实例则用来表示 g (克)、Kg (千克) 及 ml (毫升) 等计量单位, 这些计量单位及其数值可以添加到 item 里面。比方说:

- ❑ 250g chocolate (250 克巧克力)
- ❑ 4Kg potatoes (4 千克土豆)
- ❑ 500ml milk (500 毫升牛奶)

当然可以在 Item 实体里面添加名为 **unit** 的属性, 并把它类型设为字符串。然后, 针对每个 item 对象都生成一个字符串, 其中写上某种计量单位, 例如 **g**、**Kg**、**ml** 等。但这种做法的问题在于, 它会浪费数据库的存储空间, 因为在这些 Item 的 unit 属性里, 保存着大量的重复数据。除了浪费空间这个缺点之外, 假如要更新所有 Item 的 unit, 那也是件非常麻烦的事。比方说, 现在要将计量单位从 **Kg** 改为 **kilogram**, 那就得遍历所有的 Item, 并逐个修改其中的字符串, 把里面的 **Kg** 改成 **kilogram**。这样做效率很低, 而且还会导致应用程序运行得相当缓慢。解决办法是: 用指向 Unit 实体的“关系”来取代这种字符串, 该“关系”其实就是个指向 **Kg** 对象的指针, 而数据库里只需保存一份这样的 **Kg** 对象就好。这不仅降低了数据所占用的存储空间, 而且还有个好处, 就是若要改动计量单位的名称, 则只需修改这一个对象即可。



提示 为了继续构建范例程序, 需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter03.zip> 下载 ZIP 文件, 并将其解压缩, 这个文件包含了项目到目前为止的全部内容。在学习本章的过程中, 你最好能用 iOS 仿真器来运行程序, 因为这样更容易查看到 SQLite 数据库文件里的内容。

请按下列步骤修改 Grocery Dude, 为本章范例程序做准备:

1. 从 iOS 仿真器里面删掉 Grocery Dude 应用程序。
2. 点击 **Product > Clean** 菜单项, 确保以前的同名项目没有把缓存残留下来。
3. 通过 iOS 仿真器运行一遍程序, 以生成空的持久化存储区。

请按下列步骤修改 Grocery Dude, 以创建关系:

1. 可以先抓取快照或备份整个项目。
2. 用第 3 章所讲的办法, 根据 **Model 4** 来创建新的模型版本, 并将其命名为 **Model 5**。
3. 把 **Model 5** 设为 **Current Model** (当前模型)。
4. 选定 **Model 5.xcdatamodel**。
5. 把界面的 **Editor Style** 改为 **Graph**, 如图 4-1 所示。
6. 如果 **Item** 与 **Unit** 实体相互重叠, 那就通过拖曳鼠标将其分开。

7. 按下 **Control** 键，用鼠标从 **Item** 实体向 **Unit** 实体拖一条线。正确的操作结果如图 4-1 所示。

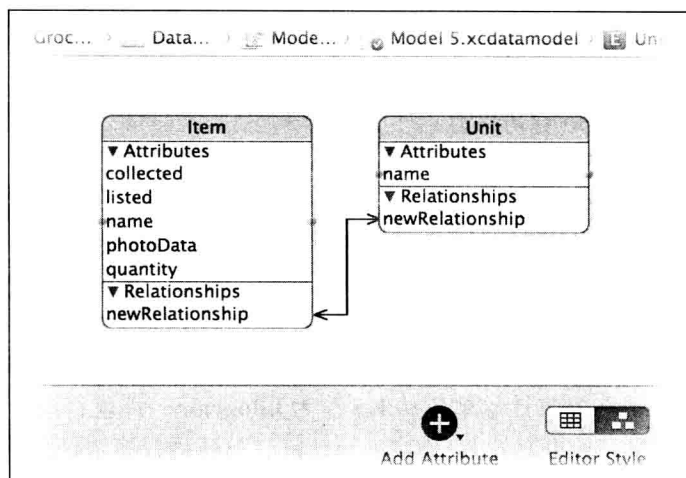


图 4-1 在两实体间创建名为 newRelationship 的关系

当编辑器界面处于 **Graph** 风格时，在两实体间创建的关系是双向关系，也就是说，两个实体之间的这种双向关系其实是由两条方向相反的关系所组成的。在本例中，一条关系是从 **Item** 实体指向 **Unit** 实体，而另外一条关系则是从 **Unit** 实体指向 **Item** 实体。假如在 **Table** 风格的编辑器界面中创建关系，那么创建出来的可能就是单向关系（one-way relationship）了。此时如果需要建立双向关系，那还必须手动添加反向的那一条关系才行。

设置好双向关系之后，可以执行下列操作：

1. 将 **Item** 托管对象关联到 **Unit** 托管对象。
2. 将 **Unit** 托管对象关联到 **Item** 托管对象。

把两个实体关联起来之后，我们就可以通过关系来访问相关实体的属性了（比方说 `item.newRelationship.name`）。接下来需要考虑的问题则是：每个方向上的关系是一对多的，还是一对一的？想清楚这个问题之后，你应该就能给关系起一个更为恰当的名字了。一对多的关系不限制目标对象的数量，而一对一的关系则会把目标对象的数量限定为一个。

现在考虑从 **Item** 到 **Unit** 方向的关系：

- ❑ 假设从 **Item** 实体到 **Unit** 实体的关系是一对多的，那就意味着每个货品（item）都可以拥有无数种计量单位。这不太合适，因为购物清单上的货品只需要一种计量单位就够了，比方说 **Kg**（千克）或 **pound**（磅）。请注意：也可以限定一对多关系所能关联的对象个数上限。
- ❑ 如果从 **Item** 实体到 **Unit** 实体的“关系”是“一对一”的，那就意味着每个货品只能有一种计量单位。这比较合适，因为购物清单上的货品只需一种计量单位即可。由此我们觉得从 **Item** 到 **Unit** 方向的“关系”起名叫做 **unit** 是比较合适的。把

“关系”名称由 `newRelationship` 改为 `unit` 之后，就可以在 `item` 对象上通过 `item.unit.name` 来引用相关计量单位的名称了。

由于本例中的“关系”是“双向关系”，所以我们要把两个方向都考虑到。

接下来，考虑从 `Unit` 到 `Item` 方向的“关系”：

- ❑ 假设从 `Unit` 实体到 `Item` 实体的关系是“一对一”的，那就意味着每一种计量单位只能由一件货品来使用，多个货品无法共用一种计量单位。这不合适，因为购物清单上面的多项货品应该可以共用同一种计量单位才对，比方说，“两千克洋葱”和“一千克玉米”都应该能够使用“千克”这个单位。
- ❑ 如果从 `Unit` 实体到 `Item` 实体的关系是“一对多”的，那就意味着每一种计量单位可以供无数件货品使用。这很合适，因为购物清单上的多件货品确实有可能使用同一种计量单位。由此我们觉得从 `Unit` 到 `Item` 方向的“关系”应该起名叫做 `items`。如果想列出与某种计量单位有关的全部对象，那很简单，只需获取 `unit.items` 就行了，这个 `NSSet` 里面会有许多指针，而每个指针都指向一件使用该计量单位的货品。

请按下列步骤修改 `Grocery Dude`，以便配置刚才创建出来的双向关系：

1. 把从 `Item` 实体指向 `Unit` 实体的 `newRelationship` 改名为 `unit`。
2. 把从 `Unit` 实体指向 `Item` 实体的 `newRelationship` 改名为 `items`。
3. 把 `items` 这条关系的 `Type` 改成 `To-Many`，如图 4-2 所示。

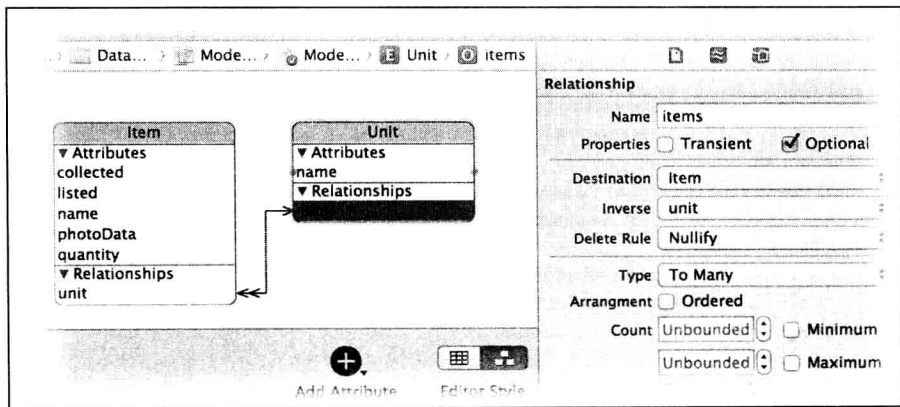


图 4-2 unit 关系和 items 关系

你现在应该已经明白了：关系的名称要和它所提供的某种访问能力相符。配置好 `unit` 及 `items` 这两条关系之后，我们就可以在代码里通过 `item.unit` 及 `unit.items` 来访问关系的对象所具备的各种特性了。但是，在使用“点”（`.`）来访问特性之前，我们必须先为这些实体创建相应的 `NSManagedObject` 子类。

请按下列步骤修改 `Grocery Dude`，以更新现有的 `NSManagedObject` 子类：

1. 确保 `Model 5.xcdatamodel` 处于选定状态。
2. 用前面章节讲过的办法来为 `Item` 及 `Unit` 实体创建 `NSManagedObject` 子类。在

Select the entities you would like to manage 这一步里，记得勾选 Item 与 Unit 实体。保存类文件的时候，别忘了把 Targets 中的“Grocery Dude”选中。Xcode 会提示是否覆盖现有文件，点击 Replace 按钮即可。

查看 Unit.h 文件，你会发现有个 NSSet 类型的 items 特性。这个 items 特性是用来表达“一对多关系”的，而由于是“一对‘多’”，所以特性的类型是 NSSet，以便容纳多个元素，另外，NSSet 还提供了一些辅助方法，用以添加及移除对象。值得注意的是：与 NSOrderedSet 或 NSArray 不同，NSSet 里面的对象是无序的。获取对象时，如果要排序，那么通常会给获取请求传入排序描述符，而获取请求执行完之后所返回的是 NSArray。假如在配置“一对多关系”的时候勾选了 Ordered 选项，那么在 NSManagedObject 子类里面，对应的特性类型就成了 NSOrderedSet。另外，NSSet 与 NSArray 之间还有个差别也值得注意，就是 NSSet 不能包含重复对象。

接下来查看 Item.h 文件，你会发现有个 Unit 类型的 unit 特性。由于从 Item 指向 Unit 的关系是一对一关系，因此系统只需要把这个关系的目标实体所对应的类当作 unit 特性的类型即可。

如程序清单 4-1 所示，我们很容易就能把两个 item 对象的 unit 属性都设为同一种计量单位。

程序清单4-1 AppDelegate.m文件中的demo方法（演示如何使用刚才配置好的关系）

```
if (debug==1) {
    NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
}

Unit *kg =
    [NSEntityDescription insertNewObjectForEntityForName:@"Unit"
     inManagedObjectContext:[self cdh] context]];

Item *oranges =
    [NSEntityDescription insertNewObjectForEntityForName:@"Item"
     inManagedObjectContext:[self cdh] context]];

Item *bananas =
    [NSEntityDescription insertNewObjectForEntityForName:@"Item"
     inManagedObjectContext:[self cdh] context]];

kg.name = @"Kg";
oranges.name = @"Oranges";
bananas.name = @"Bananas";
oranges.quantity = [NSNumber numberWithInt:1];
bananas.quantity = [NSNumber numberWithInt:4];
oranges.listed = [NSNumber numberWithBool:YES];
bananas.listed = [NSNumber numberWithBool:YES];
oranges.unit = kg;
bananas.unit = kg;

NSLog(@"Inserted %@", oranges.quantity, oranges.unit.name, oranges.name);
NSLog(@"Inserted %@", bananas.quantity, bananas.unit.name, bananas.name);
[[self cdh] saveContext];
```

请按下列步骤修改 Grocery Dude，以插入使用同一种计量单位的多件货品：

1. 修改 AppDelegate.m 文件中的 demo 方法，用程序清单 4-1 里的代码替换原有代码。
2. 运行一遍应用程序。控制台里应该会出现如图 4-3 所示的日志。
3. 删掉 AppDelegate.m 文件 demo 方法中的所有代码。

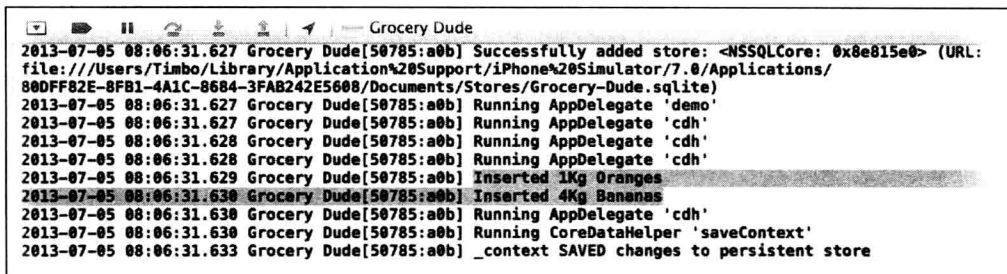


图 4-3 插入多个 item，这些 item 都通过关系指向同一种计量单位

我们采用第 2 章所讲的办法查看 Grocery-Dude.sqlite 文件的内容，以证明数据库里只有一行数据是用来表示 Kg 对象的。正常结果应该如图 4-4 所示。

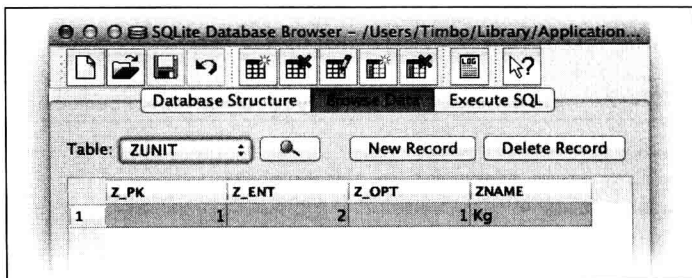


图 4-4 在 SQLite 数据库的 ZUNIT 表中存放的 Unit 实体

在学习下一节之前，请先关掉 SQLite Database Browser。

4.2 Delete 规则

在配置关系的时候，一定要注意 **Delete Rule** (Delete 规则)。当我们删除某个对象时，该规则决定了与之相关的那些对象应该如何处理。可供选择的“Delete 规则”有下面几种：

- ❑ **Nullify** 大多数情况都可以采用这种默认的 Delete 规则。如果删除了某个对象，而该对象与其他对象的“关系”又受制于 Nullify 规则，那么这些对象就会把指向该对象的“关系”清空。比方说，有个名叫 **Kg** 的 **unit** 对象，它关联着一些 **item** 对象。假如 **items** “关系”的 Delete 规则是 **Nullify**，那么当把这个名为 **Kg** 的 **unit** 对象删掉之后，与它相关的那些 **item** 对象就会将其 **unit** 特性设为 **nil**。

- ❑ **Cascade** 这种 Delete 规则会沿着关系来传播删除操作。比方说, 有个名叫 **Kg** 的 **unit** 对象, 它关联着一些 **item** 对象。假如 **items** 关系的 Delete 规则是 **Cascade**, 那么当把这个名为 **Kg** 的 **unit** 对象删掉之后, 与它相关的所有 **item** 对象也会被删除。
- ❑ **Deny** 如果尚有其他对象与某对象相关联, 那么这种 Delete 规则会阻止开发者删除该对象。比方说, 有个名叫 **Kg** 的 **unit** 对象, 它关联着一些 **item** 对象。假如 **items** 关系的 Delete 规则是 **Deny**, 那么当开发者把这个名为 **Kg** 的 **unit** 对象删除并试图将改动后的数据保存到上下文的时候, 系统就会发现目前仍有 **item** 对象与之相关联, 从而引发 validation error (验证错误)。假如把某条关系的 Delete 规则设成了 **Deny**, 那么在删除源对象之前, 开发者需要确保程序里面已经没有与该对象通过这条关系相关联的目标对象。
- ❑ **No Action** 这是一种奇怪的 Delete 规则, 它会导致对象图处于不一致状态 (inconsistent state)。假如运用了这条 Delete 规则, 那么在删除某个对象之后, 开发者必须手动设定反向的关系, 以确保它们都指向有效的对象。只有在极个别的情况下才需要使用这种 Delete 规则。

为了测试删除对象之后的效果, 我们需要添加一个新方法, 用以显示持久化存储区里 **unit** 对象及 **item** 对象的个数。相关代码如程序清单 4-2 所示。

程序清单4-2 AppDelegate.m文件中的showUnitAndItemCount方法

```
- (void)showUnitAndItemCount {
    // List how many items there are in the database
    NSFetchRequest *items =
    [NSFetchRequest fetchRequestWithEntityName:@"Item"];
    NSError *itemsError = nil;
    NSArray *fetchedItems =
    [[[self cdh] context] executeFetchRequest:items error:&itemsError];
    if (!fetchedItems) {NSLog(@"%@", itemsError);}
    else {NSLog(@"Found %lu item(s) ", (unsigned long) [fetchedItems count]);}

    // List how many units there are in the database
    NSFetchRequest *units =
    [NSFetchRequest fetchRequestWithEntityName:@"Unit"];
    NSError *unitsError = nil;
    NSArray *fetchedUnits =
    [[[self cdh] context] executeFetchRequest:units error:&unitsError];
    if (!fetchedUnits) {NSLog(@"%@", unitsError);}
    else {NSLog(@"Found %lu unit(s) ", (unsigned long) [fetchedUnits count]);}
}
```

请按下列步骤修改 Grocery Dude, 为测试 Delete 规则做准备:

1. 把程序清单 4-2 中的 showUnitAndItemCount 方法添加到文件 AppDelegate.m

里，并放在现有的 demo 方法上方。

2. 修改 AppDelegate.m 文件的 demo 方法，把原有的代码都删掉，只写一句 [self showUnitAndItemCount]; 即可。

3. 运行应用程序。控制台里应该会输出如图 4-5 所示的日志。

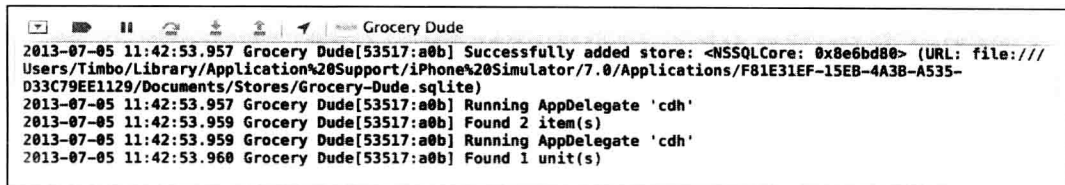


图 4-5 持久化存储区里面 item 对象与 unit 对象的个数

根据图中显示的结果可知，持久化存储区里面有两个 item 对象和一个 unit 对象。这两个 item 对象是刚才在实现程序清单 4-1 的时候插入的，它们分别是 **oranges** 和 **bananas**。仅有的那个 unit 对象就是 Kg，而这两个 item 对象都与之相关联着。现在来看看如果 Delete 规则是 **Deny**，那么在删掉名为 Kg 的 unit 对象时会发生什么。

请按下列步骤修改 Grocery Dude，以便将 Delete 规则设为 **Deny**：

1. 在 **Model 5** 的 **Unit** 实体中选定 **items** “关系”。
2. 通过 Data Model Inspector 界面（可按“**Option** + ⌘ + 3”组合键调出该界面），把 **items** “关系”的 **Delete Rule** 设为 **Deny**。

程序清单 4-3 中的这段代码用于删除名叫 **Kg** 的 **unit** 对象。

程序清单4-3 AppDelegate.m文件中的demo方法（用于删除unit对象）

```

NSLog(@"Before deletion of the unit entity:");
[self showUnitAndItemCount];

NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Unit"];
NSPredicate *filter =
[NSPredicate predicateWithFormat:@"name == %@", @"Kg"];
[request setPredicate:filter];
NSArray *kgUnit =
[[[self cdh] context] executeFetchRequest:request error:nil];
for (Unit *unit in kgUnit) {
    [_coreDataHelper.context deleteObject:unit];
    NSLog(@"A Kg unit object was deleted");
}

NSLog(@"After deletion of the unit entity:");
[self showUnitAndItemCount];
  
```

请按下列步骤修改 Grocery Dude，用代码来删掉名为 Kg 的 unit 对象：

1. 修改 AppDelegate.m 文件中的 demo 方法，用程序清单 4-3 里的代码替换原有代码。
2. 运行应用程序。你应该会在控制台中看到如图 4-6 所示的日志。

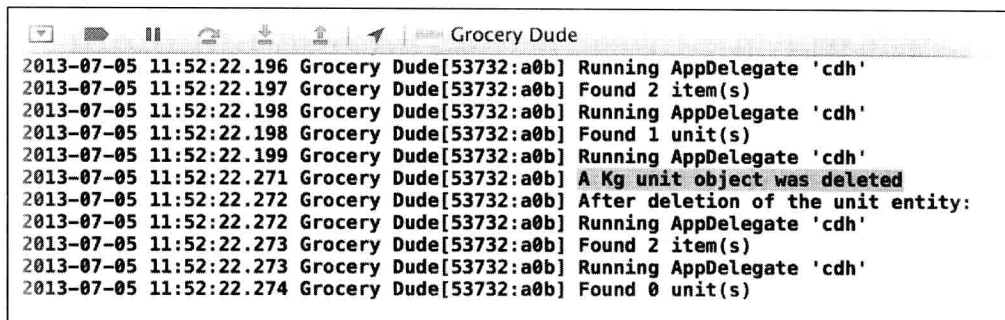


图 4-6 将 Delete 规则设为 Deny 之后，它生效了吗？

根据控制台中的日志来看，这条 Deny Delete 规则好像并没有生效。这是怎么回事呢？按照刚才讲的，由于 oranges 和 bananas 这两个对象还与 unit 对象有关联，所以 Deny 规则应该阻止我们删除这个名为 Kg 的 unit 对象，但是现在为什么程序里面已经没有 unit 对象了呢？这些问题都问得很有道理，然而关键之处在于，只有当真正保存上下文的时候，系统才会去实施 Delete 规则。

请按下列步骤修改 Grocery Dude，以便在删除 unit 对象之后保存上下文：

1. 修改 AppDelegate.m 文件中 demo 的方法，在其底部添加 `[[self cdh] saveContext];` 语句。
2. 重新运行应用程序，对上下文所做的 save（保存）操作应该会失败，如图 4-7 所示。

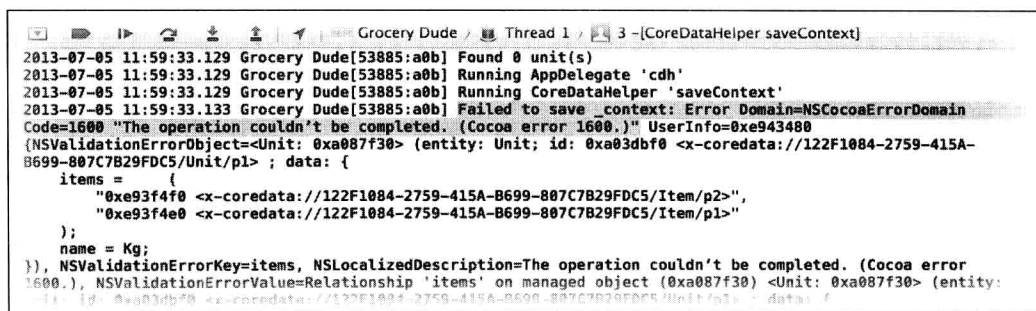


图 4-7 将 Delete 规则设为 Deny 之后，它会在保存上下文时生效

只有当尝试保存上下文的时候，系统才会去核查 Delete 规则是否生效。若发现违规，则会产生 NSCocoaErrorDomain 错误，它的错误码是 1600。如果想解决这个错误，那就必须在删掉 unit 对象之前确保该对象可以安全地移除。

假如没办法安全地删除对象，那么可以采取以下两种办法：

❑ 告知用户删除操作已遭系统拒绝，程序决定跳过该操作。

❑ 先清空 `unit.items`，然后再删掉 `unit` 对象。

在实际的应用程序中，当用户从表格视图界面里以滑动（swipe）的方式删除某个 `unit` 之后，可能就会引发这种错误。关系的 Delete 规则如果是 Deny，那么开发者就应该使用超类 `NSManagedObject` 里面名为 `validateForDelete` 的方法来判断是否能够安全地移除该对象。假如该方法返回 YES，那就表明可以把相关对象安全地删掉。程序清单 4-4 里的这段范例代码演示了此方法的用法。

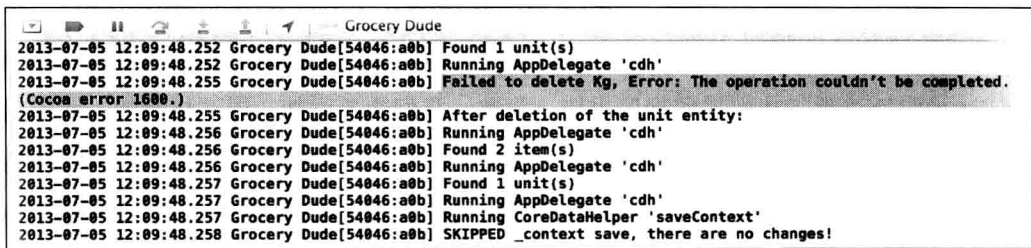
程序清单4-4 AppDelegate.m文件中的demo方法（用于验证是否能够执行删除操作）

```
NSError *error;
if ([unit validateForDelete:&error]) {
    NSLog(@"Deleting '%@'", unit.name);
    [_coreDataHelper.context deleteObject:unit];
} else {
    NSLog(@"Failed to delete %@, Error: %@",
        unit.name, error.localizedDescription);
}
```

请按下列步骤修改 Grocery Dude，以便在删除相关对象之前先验证删除操作是否符合 Delete 规则：

1. 修改 AppDelegate.m 文件中的 demo 方法，用程序清单 4-4 把 for 循环体里原有的代码替换掉。

2. 再度运行应用程序。这次应该会看到如图 4-8 所示的结果，也就是说，名为 Kg 的 `unit` 对象并未删掉。



```
2013-07-05 12:09:48.252 Grocery Dude[54046:a0b] Found 1 unit(s)
2013-07-05 12:09:48.252 Grocery Dude[54046:a0b] Running AppDelegate 'cdh'
2013-07-05 12:09:48.255 Grocery Dude[54046:a0b] Failed to delete Kg, Error: The operation couldn't be completed.
(Cocoa error 1600.)
2013-07-05 12:09:48.255 Grocery Dude[54046:a0b] After deletion of the unit entity:
2013-07-05 12:09:48.256 Grocery Dude[54046:a0b] Running AppDelegate 'cdh'
2013-07-05 12:09:48.256 Grocery Dude[54046:a0b] Found 2 item(s)
2013-07-05 12:09:48.256 Grocery Dude[54046:a0b] Running AppDelegate 'cdh'
2013-07-05 12:09:48.257 Grocery Dude[54046:a0b] Found 1 unit(s)
2013-07-05 12:09:48.257 Grocery Dude[54046:a0b] Running AppDelegate 'cdh'
2013-07-05 12:09:48.257 Grocery Dude[54046:a0b] Running CoreDataHelper 'saveContext'
2013-07-05 12:09:48.258 Grocery Dude[54046:a0b] SKIPPED _context save, there are no changes!
```

图 4-8 删除对象之前，先验证该操作是否符合 Delete 规则

下一步是把数据验证过程中所发生的错误展示给用户。尽管从用户体验的角度来讲这么做相当不好，但这也是迫不得已的办法。

4.3 数据验证错误

在把对象保存到持久化存储区之前，它们必须通过验证。假如某个对象未能通过验

证，那么系统就会抛出 domain 为 `NSCocoaErrorDomain` 的 `NSError`。验证错误 (validation error) 的类型一共有十几种，可以在 Xcode 里面跳转到 `NSManagedObject-ValidationErrors` 的定义以查看它们。跳转后的文件应该是 `CoreDataErrors.h`，其中定义了如表 4-1 所示的这些验证错误。

表4-1 有可能出现的验证错误类型

| Code | Constant |
|------|---|
| 1580 | <code>nsvalidationMultipleErrorsError</code> |
| 1570 | <code>NSValidationMissingMandatoryPropertyError</code> |
| 1580 | <code>NSValidationRelationshipLacksMinimumCountError</code> |
| 1590 | <code>NSValidationRelationshipExceedsMaximumCountError</code> |
| 1600 | <code>NSValidationRelationshipDeniedDeleteError</code> |
| 1610 | <code>NSValidationNumberTooLargeError</code> |
| 1620 | <code>NSValidationNumberTooSmallError</code> |
| 1630 | <code>NSValidationDateTooLateError</code> |
| 1640 | <code>NSValidationDateTooSoonError</code> |
| 1650 | <code>NSValidationInvalidDateError</code> |
| 1660 | <code>NSValidationStringTooLongError</code> |
| 1670 | <code>NSValidationStringTooShortError</code> |
| 1680 | <code>NSValidationStringPatternMatchingError</code> |

目前的应用程序无法保存相关对象，因为一旦在上下文上面执行保存，程序就会失败，并给出错误码 1600。我们刚才通过 `validateForDelete` 方法来防止程序执行不合乎 Delete 规则的保存操作，但是，在编写应用程序其他地方的代码时，开发者很容易就会忘了添加这种验证逻辑，从而导致程序出错，为此，我们需要添加一段“优雅的”错误提示代码，以作为最后的解决手段。这样的话，终端用户就可以把收到的错误码发送给开发者了。程序清单 4-5 中的这段代码用来引发验证错误。

程序清单4-5 AppDelegate.m文件中的demo方法（试图删除 unit 对象）

```
NSLog(@"Before deletion of the unit entity:");
[self showUnitAndItemCount];

NSFetchRequest *request =
[NSFetchRequest fetchRequestWithEntityName:@"Unit"];
NSPredicate *filter =
[NSPredicate predicateWithFormat:@"name == %@", @"Kg"];
[request setPredicate:filter];
NSArray *kgUnit =
[[[self cdh] context] executeFetchRequest:request error:nil];
```

```

for (Unit *unit in kgUnit) {
    [[[self cdh] context] deleteObject:unit];
    NSLog(@"A Kg unit object was deleted");
}

NSLog(@"After deletion of the unit entity:");
[self showUnitAndItemCount];
[[self cdh] saveContext];

```

请按下列步骤修改 Grocery Dude，使应用程序再度于保存上下文时出错：

1. 修改 AppDelegate.m 文件中的 demo 方法，用程序清单 4-5 替换原有代码。
2. 再次运行应用程序，这次应该会在保存的时候失败，并产生 1600 错误。

为了向用户显示验证错误，我们新编写了名为 showValidationError 的方法，用它拦截验证错误，并给出适当的警告信息。新方法必须根据 1600 这个错误码查出更为详细的信息，然后才能继续往下执行。根据表 4-1 可知，1600 这个错误码意味着发生了 NSValidationRelationshipDeniedDeleteError。程序清单 4-6 中的这个 showValidationError 方法可以捕获并向用户展示这种错误。

程序清单4-6 CoreDataHelper.m文件中的showValidationError方法

```

#pragma mark - VALIDATION ERROR HANDLING
- (void)showValidationError:(NSError *)anError {

    if (anError && [anError.domain isEqualToString:@"NSCocoaErrorDomain"]) {
        NSArray *errors = nil; // holds all errors
        NSString *txt = @""; // the error message text of the alert

        // Populate array with error(s)
        if (anError.code == NSValidationMultipleErrorsError) {
            errors = [anError.userInfo objectForKey:NSDetailedErrorsKey];
        } else {
            errors = [NSArray arrayWithObject:anError];
        }
        // Display the error(s)
        if (errors && errors.count > 0) {
            // Build error message text based on errors
            for (NSError * error in errors) {
                NSString *entity =
                    [[[error.userInfo objectForKey:@"NSValidationErrorObject"]entity]name];

                NSString *property =
                    [error.userInfo objectForKey:@"NSValidationErrorKey"];

                switch (error.code) {
                    case NSValidationRelationshipDeniedDeleteError:
                        txt = [txt stringByAppendingFormat:
                            @"%@ delete was denied because there are associated %@\n(Error Code
                            %li)\n\n"

```

```

        , entity, property, (long)error.code];
    break;
    default:
    txt = [txt stringByAppendingFormat:
        @"Unhandled error code %li in showValidationError method"
        , (long)error.code];
    break;
    }
}
// display error message txt message

```

showValidationError 方法首先剖析 NSError 的详细内容，然后把每个错误都分别放在 errors 数组里。假如这个数组不是空的，那么该方法就迭代它，并在这个过程中构建出表示错误信息的字符串。创建好字符串之后，把它传给 UIAlertView，以便展示给用户。

请按下列步骤修改 Grocery Dude，以便向用户弹出与验证错误有关的警示信息：

1. 把程序清单 4-6 中的代码添加到 CoreDataHelper.m 文件底部，并放在 @end 前面。

2. 修改 CoreDataHelper.m 文件中的 saveContext 方法，将 [self showValidationError:error]; 添加到 Failed to save 这一行的下边。

3. 运行应用程序。这次将会看到如图 4-9 所示的画面。

从现在开始，如果程序在保存上下文的时候因为数据验证问题而失败，那么用户至少会看到一个优雅的错误信息提示界面。我们很容易就能像程序清单 4-7 这样，把与其他类型的验证错误相关的描述信息添加到 showValidationError 方法里。

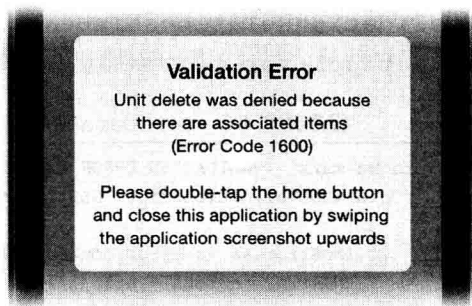


图 4-9 与直接令程序崩溃相比，通过警示视图来显示错误信息是一种对用户比较友好的方式

程序清单4-7 CoreDataHelper.m文件中的showValidationError方法（又添加了一些case分支）

```

case NSValidationRelationshipLacksMinimumCountError:
txt = [txt stringByAppendingFormat:
@"the '%@' relationship count is too small (Code %li).",
property, (long)error.code];
break;
case NSValidationRelationshipExceedsMaximumCountError:
txt = [txt stringByAppendingFormat:
@"the '%@' relationship count is too large (Code %li).",
property, (long)error.code];
break;
case NSValidationMissingMandatoryPropertyError:
txt = [txt stringByAppendingFormat:
@"the '%@' property is missing (Code %li).", property, (long)error.code];

```

```

break;
case NSValidationNumberTooSmallError:
txt = [txt stringByAppendingFormat:
@"the '%@' number is too small (Code %li).", property, (long)error.code];
break;
case NSValidationNumberTooLargeError:
txt = [txt stringByAppendingFormat:
@"the '%@' number is too large (Code %li).", property, (long)error.code];
break;
case NSValidationDateTooSoonError:
txt = [txt stringByAppendingFormat:
@"the '%@' date is too soon (Code %li).", property, (long)error.code];
break;
case NSValidationDateTooLateError:
txt = [txt stringByAppendingFormat:
@"the '%@' date is too late (Code %li).", property, (long)error.code];
break;
case NSValidationInvalidDateError:
txt = [txt stringByAppendingFormat:
@"the '%@' date is invalid (Code %li).", property, (long)error.code];
break;
case NSValidationStringTooLongError:
txt = [txt stringByAppendingFormat:
@"the '%@' text is too long (Code %li).", property, (long)error.code];
break;
case NSValidationStringTooShortError:
txt = [txt stringByAppendingFormat:
@"the '%@' text is too short (Code %li).", property, (long)error.code];
break;
case NSValidationStringPatternMatchingError:
txt = [txt stringByAppendingFormat:
@"the '%@' text doesn't match the specified pattern (Code %li).",
property, (long)error.code];
break;
case NSManagedObjectValidationError:
txt = [txt stringByAppendingFormat:
@"generated validation error (Code %li)", (long)error.code];
break;

```

请按下列步骤修改 Grocery Dude，以便增强错误信息提示界面对验证错误的描述能力：

1. 修改 CoreDataHelper.m 文件中的 showValidationErrors 方法，把程序清单 4-7 中的 case 分支添加到 default 这一行的上方。
2. 把 AppDelegate.m 文件 demo 方法中的所有代码都删掉。
3. 再度运行应用程序，这次应该不会再显示有关验证错误的信息提示界面了。

4.4 实体继承

与类一样，实体也可以继承自父实体。这个功能有助于简化数据模型。子实体会自动

继承父实体的各种属性。在底层的 SQLite 存储区里，“父-子”体系中的实体都放在同一张数据表里。

Grocery Dude 里面的货品有可能位于两个地方，一个是商店，一个是家中，而我们的范例程序就要围绕这一特性而展开。既然货品有可能出现在“商店”和“家”这两种位置，那么就可以通过“实体继承”来把这两种“位置”（location）所共有的属性提取到父实体中。比方说，这个父实体可以叫做 **Location**，它拥有名为 **summary** 的属性。而 **LocationAtHome** 实体或 **LocationAtShop** 实体则继承自 **Location**，从而自动具备 **summary** 属性。这种行为与类的继承是相似的。

如果不想令父实体实例化，那么可将其标注为抽象的。只有能够确定 Location 实体的实例在代码中毫无意义时，才应该启用这个选项。

请按下列步骤修改 Grocery Dude，以便配置实体及其继承关系：

1. 基于 **Model 5** 来添加新的模型版本，并将其命名为 **Model 6**。
2. 将 **Model 6** 设为 **Current Model**。
3. 确保 **Model 6.xcdatamodel** 处于选定状态。
4. 添加名为 **Location** 的新实体，并添加类型为 **String** 的属性，将其命名为 **summary**。
5. 选中 **Location** 实体，并在 **Utilities** 面板中打开 **Data Model Inspector** 界面，如图 4-10 所示。
6. 勾选 **Abstract Entity**，这将触发一条警告，提示开发者 Location 实体尚未有子实体。
7. 创建名为 **LocationAtHome** 的新实体，并添加类型为 **String** 的属性，将其命名为 **storedIn**。
8. 创建名为 **LocationAtShop** 的新实体，并添加类型为 **String** 的属性，将其命名为 **aisle**。

9. 点击 **LocationAtHome** 实体，在 Data Model Inspector 界面（可按“**Option+⌘+3**”组合键调出该界面）中把 Parent Entity（父实体）设为 **Location**。

10. 点击 **LocationAtShop** 实体，把 Parent Entity 设为 **Location**。

11. 如果编辑器当前不是 Graph 风格，那就将 **Editor Style** 切换到 **Graph**，然后按照图 4-10 所示来排布各个实体。要是无法在一屏中看到所有实体，则可能需要滚动屏幕。

把新的父实体与子实体准备好之后，我们应该将其链接到 Item 实体，这样的话，item 就可以同家中或商店中的某个位置相关联了。

请按下列步骤修改 Grocery Dude，以便配置 Item 实体与 LocationAtHome 实体之间的“关系”：

1. 按住 **Control** 键，用鼠标从 **LocationAtHome** 实体向 **Item** 实体拖曳一条线。
2. 将 **LocationAtHome** 实体的 **newRelationship** 改名为 **items**。
3. 将 **LocationAtHome** 实体的 **items** “关系”设为 **To-Many**。
4. 把 **Item** 实体的 **newRelationship** 改名为 **locationAtHome**。

请按下列步骤修改 Grocery Dude，以便配置 Item 实体与 LocationAtShop 实体之间的

“关系”：

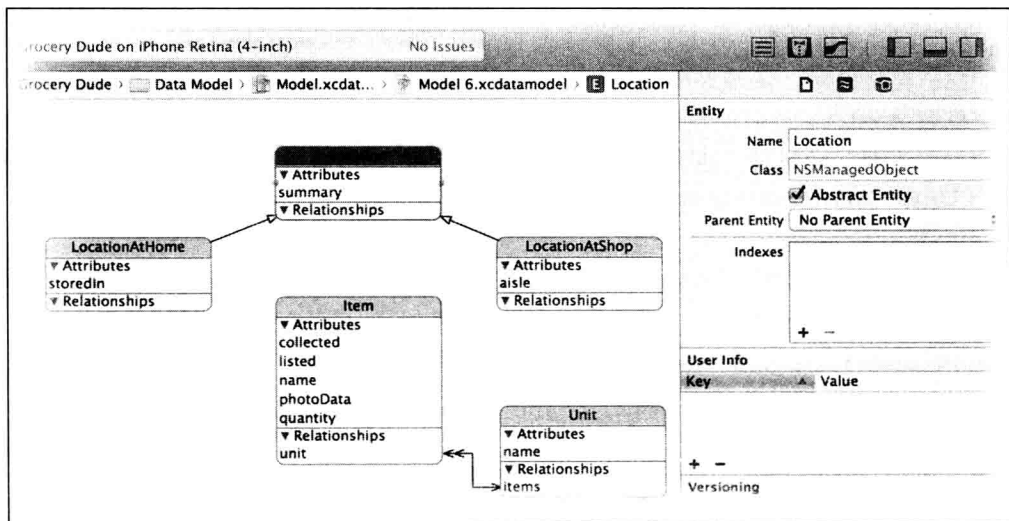


图 4-10 作为抽象父实体的 Location

1. 按住 **Control** 键，用鼠标从 **LocationAtShop** 实体向 **Item** 实体拖曳一条线。
2. 将 **LocationAtShop** 实体的 **newRelationship** 改名为 **items**。
3. 将 **LocationAtShop** 实体的 **items** 关系设为 **To-Many**。
4. 把 **Item** 实体的 **newRelationship** 改名为 **locationAtShop**。

现在的模型应该与图 4-11 相符。请注意，双箭头表示“一对多关系”。

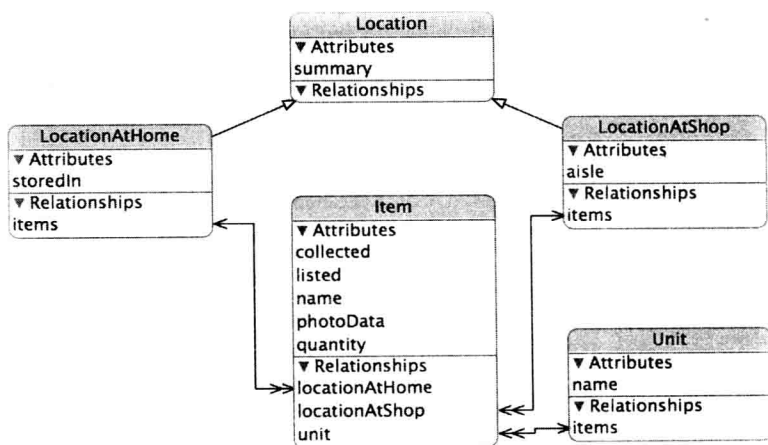


图 4-11 新模型为 Item 添加了对“位置”的支持

请按下列步骤修改 Grocery Dude，以便在代码中可以通过“点”(.)来访问新实体的属性：

1. 为 **Model 6** 的所有实体都创建相应的 `NSManagedObject` 子类，并把原有的文件替换掉。在保存类文件的那一步里，记得勾选 Targets 中的“Grocery Dude”。

生成子类文件时，系统在生成顺序上可能会遇到“先有鸡还是先有蛋”的问题。这个问题会导致表示“一对一关系”的特性类型变成 `NSManagedObject`，而正确的特性类型应该与“关系”的目标实例所具备的类型相同。图 4-12 演示了一份由 Xcode 所生成的 `Item.h` 文件，其中的特性类型是错误的。

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@class Unit;

@interface Item : NSManagedObject

@property (nonatomic, retain) NSNumber * collected;
@property (nonatomic, retain) NSNumber * listed;
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSData * photoData;
@property (nonatomic, retain) NSNumber * quantity;
@property (nonatomic, retain) Unit *unit;
@property (nonatomic, retain) NSManagedObject *locationAtHome;
@property (nonatomic, retain) NSManagedObject *locationAtShop;

@end
```

图 4-12 由 Xcode 所生成的 `NSManagedObject` 子类，其中的特性类型有误

至于哪些文件的特性类型会错误地变为基类类型 `NSManagedObject`，这要看 Xcode 在生成子类文件时所采用的顺序。若想解决这个问题，只需按照前面的方式把子类文件重新生成一遍即可。图 4-13 演示了由 Xcode 再度生成的 `Item.h` 文件，而这次的特性类型则是正确的。为了避免这个问题，开发者可能得养成两次生成 `NSManagedObject` 子类的习惯，这样就不用每次都去检查子类文件的特性类型是否正确了。

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@class LocationAtHome, LocationAtShop, Unit;

@interface Item : NSManagedObject

@property (nonatomic, retain) NSNumber * collected;
@property (nonatomic, retain) NSNumber * listed;
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSData * photoData;
@property (nonatomic, retain) NSNumber * quantity;
@property (nonatomic, retain) Unit *unit;
@property (nonatomic, retain) LocationAtHome *locationAtHome;
@property (nonatomic, retain) LocationAtShop *locationAtShop;

@end
```

图 4-13 由 Xcode 再度生成的 `NSManagedObject` 子类，这次的特性类型是正确的

请按下列步骤修改 Grocery Dude，以生成特性类型正确的子类文件：

1. 再度根据 **Model 6** 中的所有实体来创建各自的 `NSObject` 子类，并将原有的文件覆盖掉。

4.5 小结

本章讲解了如何创建并配置实体间的关系。在讨论使用关系所带来的好处时，我们也讲述了一些关键的设置，例如一对多关系、一对一关系以及 **Delete** 规则。在扩充模型的过程中，我们引入了以“位置”为中心的新实体，并创建了抽象的父实体，然后把这些实体纳入了一套实体继承体系之中。其后，我们将这套体系同 `Item` 实体关联起来，使货品可以描述出它在商店和在家中的位置。

验证数据时，如果发生了相关的错误，那么应用程序在关闭之前，会先向用户弹出错误信息提示界面。这种方式比直接令应用程序崩溃要好，因为用户可以把错误码告知开发者，以求更快地解决问题。为了防止在保存上下文的时候出错，我们介绍了 `NSObject` 子类所共有的 `validateForDelete` 方法，开发者在保存上下文之前可以先用此方法来检查待保存的数据是否合乎 **Delete** 规则。

写完基础部分的代码之后，接下来应该引入 **Table View**（表格视图），这样就能使应用程序初具规模了。我们将在下一章讨论这个问题。

4.6 习题

请根据所学内容尝试下列操作：

1. 插入新的 **Item**、**ShopLocation** 及 **HomeLocation** 对象。
2. 将 `item.shopLocation` 及 `item.homeLocation` 分别设为 **ShopLocation** 及 **HomeLocation** 对象。（提示：可参照程序清单 4-1）。
3. 把 **Unit** 实体的 **items** 关系的 **Delete** 规则从 **Deny** 改为 **Cascade**。现在删掉一个 `unit` 对象，然后观察在保存上下文的时候，相关的 `items` 对象会怎样？
4. 查看 `Grocery-Dude.sqlite` 文件中与 **Location** 相关的数据表，观察 **Location** 实体是怎样和数据库中的数据表对应起来的。同时也注意观察 **Location** 子实体中的 **storedIn** 属性与 **aisle** 属性是怎样存放在 **ZLOCATION** 表中的。

完成上述测试后，请把 **Unit** 实体的 **items** 关系的 **Delete** 规则改为 **Nullify**。

表格视图

要是我们知道自己在做什么的话，那就不叫“研究”了，对吧？

——阿尔伯特·爱因斯坦

第4章把关系和实体继承这两个特性加入到托管对象模型之中，使其变得更为灵活。但到目前为止，我们所演示的范例都局限于控制台的日志。本章将会讲解如何把 Core Data 所获取到的数据展示在表格视图里面，从而使应用程序具备更为良好的终端用户体验。首先我们简述表格视图的用法，然后直接开始构建由 Core Data 所驱动的表格视图控制器子类。这个能够复用的子类可以生成两种新的表格视图，一种用来制作购物之前准备待买货品时所使用的购物清单，而另一种则用来制作正在购物时所使用的购物清单。

5.1 表格视图基础

在 iOS 的各种界面元件中，最为常用的恐怕就是表格视图了。作为 UIScrollView 的子类，这个功能强大的 UITableView 是 UIKit 的一部分，它向开发者提供了一种极为灵活的定制方式，使他们可以把很多信息逐条展示在一张“单列表格”中。即便是 Core Data 新手，你也应该已经熟悉如何创建表格视图并用 NSArray 填充其内容了吧。假如不是很熟悉该控件，那么可以在本节学到一些表格视图的基础知识。

图 5-1 演示了表格视图的主要组件，其中每个部分的标题以及表格视图中每个单元格 cell（也叫行，row）的位置（location）应该都很明显。而部分索引（section index）则是用

小字以垂直方向显示在控件右侧的那段文本^①。用户可点击或拖曳部分索引，以便快速在表格视图的各个部分之间跳转。

要想给表格视图里填充数据，通常可以创建遵从 `UITableViewDataSource` 协议的 `UITableViewController` 子类，然后把这个表格视图控制器子类设置给故事板。`UITableViewDataSource` 协议规定了下面这两个必须实现的方法，它们用于向表格中填充数据：

- ❑ **`numberOfRowsInSection`** 方法用于指定表格视图的每个部分所拥有的行数。比方说，可以令这个方法返回 `[someArray count]`，这样的话，表格视图的行数就会与数据源数组里面的对象个数相符了。

- ❑ **`cellForRowAtIndexPath`** 用来指定每个单元格中所显示的内容。开发者通常会“深度定制”（*heavily customize*）这个方法。假如采用内置的单元格风格，那么就可以使用标准的 `UITableViewCell` 中的某些默认特性。比方说，在图 5-1 中，每一行里所显示的文本就是通过 `textLabel.text` 特性设置的。若想查阅完整的特性列表，请在 Xcode 中跳转至 `UITableViewCell` 的定义。

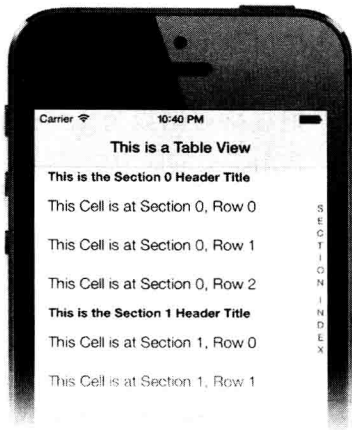


图 5-1 表格视图的基础组件

采用了 `UITableViewDataSource` 协议之后，还可以实现一些可选方法（*optional method*）。这些方法用于配置表格的编辑（*editing*）、渲染（*rendering*）、删除（*deleting*）、页眉（*header*）、页脚（*footer*）、索引（*index*）等诸多事宜。大部分方法都会在本章后面各节中讲到。

5.2 由 Core Data 所驱动的表格视图

刚才说过，若是不使用 `Core Data`，那么开发者通常会把 `NSArray` 用作数据源，并以之填充表格视图。但这种做法的关键问题在于：假如数组特别大，而占用的内存又特别多的话，那么程序性能将受到严重影响，这一点在原来的开发中可能已经体会到了。早前章节通过 `NSFetchRequest` 来获取 `Core Data` 数据，这种做法所返回的是个 `NSArray`。虽说我们也可以直接用这种数组来填充表格视图，但实际上还有更好的办法。这个办法就是依然使用 `NSFetchRequest` 来获取数据，不过这次额外配置一些选项，比方说，可以通过设定 `setFetchBatchSize` 选项来分批获取数据。这一选项虽小，但它对内存用量的影响却很大，而且还能因此改善整个程序的性能。我们所设的这个“批次获取量”（*batch size*）应该比屏幕上最多能显示出来的表格行数略微大一点。

① 也叫 `index list`。——译者注

想在 Core Data 与表格视图之间高效地管理获取到的数据，最好的方式就是使用 `NSFetchedResultsController`。假如直接使用由 `NSFetchRequest` 所返回的数组，而不使用 `NSFetchedResultsController`，那么当底层数据有变化时，数组里的对象可能就会失效，而应用程序也可能就会随之崩溃。

把表格视图设为 `NSFetchedResultsController` 的委托，可以使表格视图具备追踪数据变更的能力，也就是说，假如获取到的对象在底层的上下文中发生了变化，那么表格视图亦将随之自动更新。由 `NSFetchedResultsController` 所支援的表格视图其效率也可以通过设置缓存而提升，开发者只需给缓存取个独特的名称即可。用了缓存之后，就可以尽量减少那种无谓的重复获取操作了。除了可以提升性能及追踪数据变更之外，`NSFetchedResultsController` 还具备一些非常方便的特性，开发者可以用它们来轻松地配置由 Core Data 所驱动的表格视图。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter04.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product>Clean** 菜单项，这样可以清除掉同名项目所残留的缓存。另外，请把 iOS 设备或 iOS 仿真器中的 Grocery Dude 程序全部删掉。

5.3 创建 CoreDataTVC

`CoreDataTVC` 会是个可供复用的子类，Grocery Dude 里面所有由 Core Data 驱动的表格视图都需要靠它来支撑。这个类会写得相当通用，使得也能够用于你自己的应用程序。为了创建 `CoreDataTVC`，我们需要创建 `UITableViewController` 子类，并为其添加 `NSFetchedResultsController` 类型的实例变量。

请按下列步骤修改 Grocery Dude，以便创建 `CoreDataTVC`：

1. 选择名为 **Generic Core Data Classes** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，并点击 **Next** 按钮。
4. 将 **Subclass of** 设为 `UITableViewController`，并把 **Class** 名称设为 `CoreDataTVC`，然后点击 **Next** 按钮。
5. 确保 **Targets** 中的“Grocery Dude”处于勾选状态，然后点击 **Create** 按钮，这样就能在 Grocery Dude 项目的目录中创建类文件了。

完成上述操作之后的效果如图 5-2 所示。

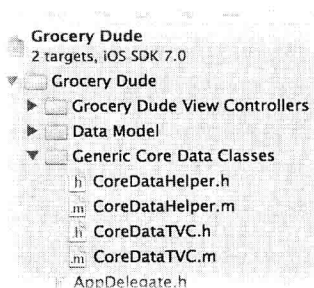


图 5-2 CoreDataTVC

选中 CoreDataTVC.m 文件，然后在类编辑器窗口的范围内点击鼠标。此时 Xcode 会警告说 CoreDataTVC.m 没有经过适当的配置。目前我们完全可以无视这一警告。



由于本书会反复用到这个类，所以为了缩减文本，笔者没有选择 CoreDataTableView-ViewController 这个名称，而是把它起名叫做 CoreDataTVC。这与 Objective-C 语言标准的命名约定相违背，若不是因为想要节省篇幅，笔者也不会提倡这种起名方式。在一般情况下，类的名称应该清晰地表达出它的用途，并且不能含有歧义。比方说，某些开发者可能会把 TVC 当成表格视图 单元格的缩写，而没有将其理解为表格视图控制器的简称。

CoreDataTVC 类很简单，它只是个采用了 NSFetchedResultsControllerDelegate 协议的 UITableViewController 子类而已，其中又有个类型为 NSFetchedResultsController 的 frc 特性。由程序清单 5-1 可知，该类还会包含名为 performFetch 的方法。此方法负责获取数据并刷新表格视图，另外，当获取操作失败时，还提供错误汇报功能。

程序清单5-1 CoreDataTVC.h文件

```
#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
@interface CoreDataTVC : UITableViewController
<NSFetchedResultsControllerDelegate>
@property (strong, nonatomic) NSFetchedResultsController *frc;
- (void)performFetch;
@end
```

请按下列步骤修改 Grocery Dude，以便配置 CoreDataTVC 的头文件：

1. 用程序清单 5-1 来替换 CoreDataTVC.h 文件原来所含的全部代码。忽略 Xcode 所发出的警告。

CoreDataTVC 类的实现文件主要分为三个主要部分。为了便于浏览及阅读代码，笔者用编译指示标记将其区隔开。

- ❑ FETCHING
- ❑ DATASOURCE: UITableView
- ❑ DELEGATE: NSFetchedResultsController

5.3.1 FETCHTING

如程序清单 5-2 所示，CoreDataTVC.m 会在 FETCHING 部分中实现 performFetch 方法。前面说过，该方法负责获取数据并刷新表格视图。假如在获取时出错，那么错误信息会记录到控制台。

程序清单5-2 CoreDataTVC.m文件的FETCHING部分

```

#import "CoreDataTVC.h"
@implementation CoreDataTVC
#define debug 1

#pragma mark - FETCHING
- (void)performFetch {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

if (self.frc) {
    [self.frc.managedObjectContext performBlockAndWait:^(

        NSError *error = nil;
        if (![self.frc performFetch:&error]) {

            NSLog(@"Failed to perform fetch: %@", error);
        }
        [self.tableView reloadData];
    )];
} else {
    NSLog(@"Failed to fetch, the fetched results controller is nil.");
}
}
@end

```

请按下列步骤修改 Grocery Dude，以便配置 CoreDataTVC 的实现文件：

1. 用程序清单 5-2 来替换 CoreDataTVC.m 文件里原有的代码。现在警告应该就会消失了。

5.3.2 DATASOURCE:UITableView

由于 CoreDataTVC 继承了 UITableViewController，所以它默认也就采用了 UITableViewDataSource 协议。而一旦采用该协议，则意味着 CoreDataTVC 或其子类必须实现刚才提到的 numberOfRowsInSection 方法及 cellForRowAtIndexPath 方法。假如没有这两个方法，那么表格视图里面就显示不出数据了。cellForRowAtIndexPath 稍后将交给 CoreDataTVC 的子类来实现，因为每个子类对该方法的实现方式都各不相同。

UITableViewDataSource 协议共有 9 个可选方法，其中 4 个比较通用，故而可以放在 CoreDataTVC 里面实现。对于这 4 个方法来说，NSFetchedResultsController 很容易就能提供协议所要求的返回值。

❑ **numberOfSectionsInTableView** 方法用于指定表格视图所具备的部分数量。假如不实现该方法，那就会采用默认值 1。创建 NSFetchedResultsController 实例的时候，可以配置 **sectionNameKeyPath**，它会把获取到的结果划分成部分。我们只需令 **numberOfSectionsInTableView** 方法返回 `[[self.frc sections]`

count], 即可把获取到的结果适当地划分成不同的部分。这么做的好处是: 假如 NSFetchedResultsController 需要有多部分, 那么表格视图控制器也会自动处理好这些部分。

❑ **sectionForSectionIndexTitle** 方法用于指明部分索引里面的某个部分的标题具体对应于表格视图本体中的哪个部分。由于 NSFetchedResultsController 专门有方法用于向 UITableViewDataSource 协议中的这个方法提供数据, 所以只需令其返回 [self.frc sectionForSectionIndexTitle:title atIndex:index] 即可。

❑ **titleForHeaderInSection** 方法用于指明每个部分的标题里应该写什么文本。一般来说, 只要令该方法返回 [[self.frc sections] objectAtIndex:section] name], 它就可以根据开发者所配置的 sectionNameKeyPath 来提供恰当的部分信息了。

❑ **sectionIndexTitlesForTableView** 方法用于指定表格视图的部分索引里面每个部分的标题的文字。由于 NSFetchedResultsController 专门有特性用于向 UITableViewDataSource 协议中的这个方法提供数据, 所以只需令其返回 [self.frc sectionIndexTitles] 即可。

正如程序清单 5-3 所示, 用 UITableViewDataSource 协议中的方法来为表格视图填充数据是件非常简单的事。稍后你可能会在 CoreDataTVC 的子类中覆写 titleForHeaderInSection 等方法, 不过就目前来说, 程序清单 5-3 可以视为一份良好的起始模板。

程序清单 5-3 CoreDataTVC.m 文件中的 DATASOURCE: UITableView 部分

```
#pragma mark - DATASOURCE: UITableView
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }
    return [[self.frc.sections objectAtIndex:section] numberOfObjects]
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }
    return [[self.frc.sections] count];
}
- (NSInteger)tableView:(UITableView *)tableView
sectionForSectionIndexTitle:(NSString *)title
atIndex:(NSInteger)index {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }
    return [self.frc sectionForSectionIndexTitle:title atIndex:index];
}
```

```

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return [[[self.frc sections] objectAtIndex:section] name];
}
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return [self.frc sectionIndexTitles];
}

```

请按下列步骤修改 Grocery Dude，以便更新 CoreDataTVC 的实现文件：

1. 将程序清单 5-3 中的代码添加到 CoreDataTVC.m 文件底部，并放在 @end 上方。

5.4 DELEGATE: NSFetchedResultsController

通过查看头文件，我们可以发现 CoreDataTVC 类采用了 NSFetchedResultsControllerDelegate 协议，这意味着开发者可以实现协议中的某些可选方法，以此来确保表格视图控制器能够正确地处理移动（move）、删除（delete）、更新（update）及插入（insertion）等操作。凡是要修改表格视图的时候，就需要调用它的 beginUpdates 方法，修改完之后，则需调用其 endUpdates 方法。由于范例程序使用 CoreDataTVC 来控制表格视图，所以我们需要像程序清单 5-4 这样，分别从 controllerWillChangeContent 及 controllerDidChangeContent 中调用上述两个方法。

程序清单5-4 需要添加到CoreDataTVC.m文件中的代码

```

#pragma mark - DELEGATE: NSFetchedResultsController
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller
{
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    [self.tableView beginUpdates];
}
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
{
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    [self.tableView endUpdates];
}

```

请按下列步骤修改 Grocery Dude，以便更新 CoreDataTVC 的实现文件：

1. 将程序清单 5-4 中的代码添加到 CoreDataTVC.m 文件底部, 并放在 @end 上方。

NSFetchedResultsControllerDelegate 协议中还有两个方法需要实现, 它们负责根据给定的 type 来分别处理移动、删除、更新及插入等操作。相关代码如程序清单 5-5 所示。请注意, 在 didChangeObject 方法中, 其他三种 type 都会有动画效果, 唯独 NSFetchedResultsControllerChangeUpdate 没有行动画 (row animation)。笔者之所以要这么做, 是为了提升用户同表格视图 单元格之间的交互速度。在 Grocery Dude 程序中, 如果用户把购物清单中的某项货品打上对勾 (tick off), 那么相应的对勾 (tick) 符号立刻就会显示出来。笔者所用的 withRowAnimation 选项是没有渐入延迟的。但你在开发自己的应用程序项目时, 可以把这个选项设成 UITableViewRowAnimationAutomatic。

程序清单5-5 CoreDataTVC.m文件的DELEGATE: NSFetchedResultsController

```
- (void)controller:(NSFetchedResultsController *)controller
    didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
        atIndex:(NSUInteger)sectionIndex
    forChangeType:(NSFetchedResultsControllerChangeType)type {

    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    switch(type) {
        case NSFetchedResultsControllerChangeInsert:
            [self.tableView insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                           withRowAnimation:UITableViewRowAnimationFade];
            break;
        case NSFetchedResultsControllerChangeDelete:
            [self.tableView deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                           withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}

- (void)controller:(NSFetchedResultsController *)controller
    didChangeObject:(id)anObject
        atIndexPath:(NSIndexPath *)indexPath
    forChangeType:(NSFetchedResultsControllerChangeType)type
    newIndexPath:(NSIndexPath *)newIndexPath {

    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    UITableView *tableView = self.tableView;
    switch(type) {
        case NSFetchedResultsControllerChangeInsert:
            [tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
                             withRowAnimation:UITableViewRowAnimationAutomatic];
```



```

break;
case NSFetchedResultsControllerDelete:
[tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationAutomatic];
break;
case NSFetchedResultsControllerUpdate:
if (!newIndexPath) {
    [tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
        withRowAnimation:UITableViewRowAnimationNone];
} else {
    [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
        withRowAnimation:UITableViewRowAnimationNone];
    [tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
        withRowAnimation:UITableViewRowAnimationNone];
}
break;
case NSFetchedResultsControllerMove:
[tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationAutomatic];
[tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
    withRowAnimation:UITableViewRowAnimationAutomatic];
break;
}
}
}

```

请按下列步骤修改 Grocery Dude，以更新 CoreDataTVC 的实现文件：

1. 把程序清单 5-5 中的代码添加到 CoreDataTVC.m 文件底部，并放在 @end 上方。

现在已经可以从 CoreDataTVC 中继承子类了。Grocery Dude 程序里将会有 5 个表格视图，这就意味着要有 5 个独立的 CoreDataTVC 子类，每个子类分别负责操控对应的自定义表格视图，而这 5 个子类的代码也彼此相似：

- ❑ **PrepareTVC** 可以列出能够添加到购物清单里的货品。
- ❑ **ShopTVC** 可以列出购物清单中的货品。
- ❑ **UnitsTVC** 可以列出货品所采用的计量单位（比方说 Kg、g 或 liter）。这个类会在下一章中实现。
- ❑ **LocationsAtHomeTVC** 会列出家中可以存放货品的各个位置。这个类将在下一章中实现。
- ❑ **LocationsAtShopTVC** 会列出商店里摆放相关货品的货架位置。这个类将在下一章中实现。

PrepareTVC 及 ShopTVC 这两个表格视图将会是 Grocery Dude 程序里最主要的视图，而我们现在则要给应用程序添加 Tab Bar 控制器，使用户能够在这两个视图之间切换。

请按下列步骤修改 Grocery Dude，以便添加 Tab Bar 控制器：

1. 选定 **Main.storyboard**。
2. 向故事板中拖放 **Tab Bar Controller**，将其置于原有的 **Navigation Controller** 左侧。

3. 删掉和 **Tab Bar Controller** 相连的那两个默认的视图控制器。
4. 按住 **Control** 键，用鼠标从 **Tab Bar Controller** 沿直线向 **Navigation Controller** 拖曳，然后在弹出的菜单中选择 **Relationship Segue > view controllers** 菜单项。
5. 在 **Attributes Inspector** 界面（可以按“**Option + ⌘ + 4**”组合键调出该界面）中，把 **Tab Bar Controller** 设为 Initial View Controller，如图 5-3 所示。

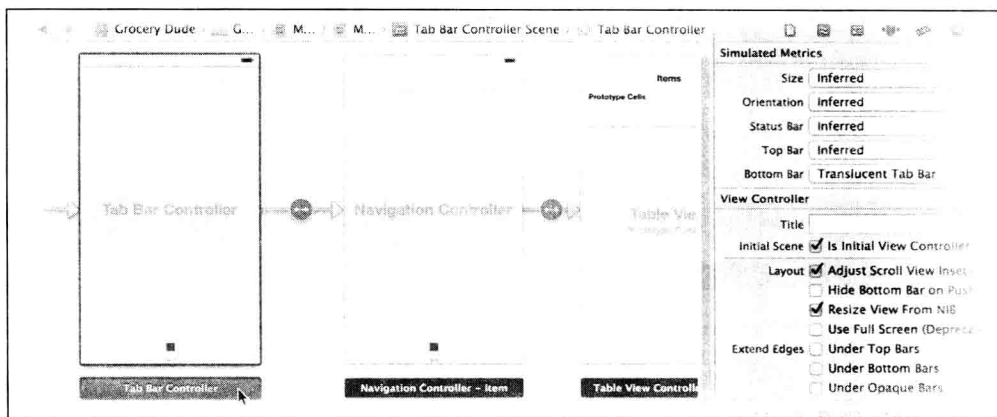


图 5-3 在表格视图之间切换时所需的 Tab Bar 控制器

假如只有一个选项卡（tab），那就体现不出 Tab Bar 控制器的优势了，所以我们需要再添加一个选项卡，以便和即将要编写的 ShopTVC 表格视图相对应。不过在添加选项卡之前，先得把 ShopTVC 表格视图本身创建出来。

请按下列步骤修改 Grocery Dude，以便新建表格视图并将其与 Tab Bar 相连：

1. 选定 **Main.storyboard**。
2. 向故事板中拖放一个新的 **Table View Controller**，将其放在 **Navigation Controller** 下边。
3. 选定这个新的 **Table View Controller**，然后点击 **Editor > Embed In > Navigation Controller**。
4. 在 **Attributes Inspector** 界面（可以按 **Option + ⌘ + 4** 组合键调出该界面）将新 **Table View Controller** 的 **Navigation Item Title** 设为 **Grocery Dude**。
5. 点击新 **Table View Controller** 的 **Prototype Table View Cell**，把 **Reuse Identifier** 设为 **Shop Cell**。ShopTVC 的 `cellForRowAtIndexPath` 方法将根据这个标识符向表格中的单元格（cell）填充数据。
6. 按住 **Control** 键，用鼠标从 **Tab Bar Controller** 沿直线向新的 **Navigation Controller** 拖曳，然后选择 **Relationship Segue > View controllers**。
7. 令 **Tab Bar Controller** 在垂直方向上居中，这样看起来更加美观，如图 5-4 所示。

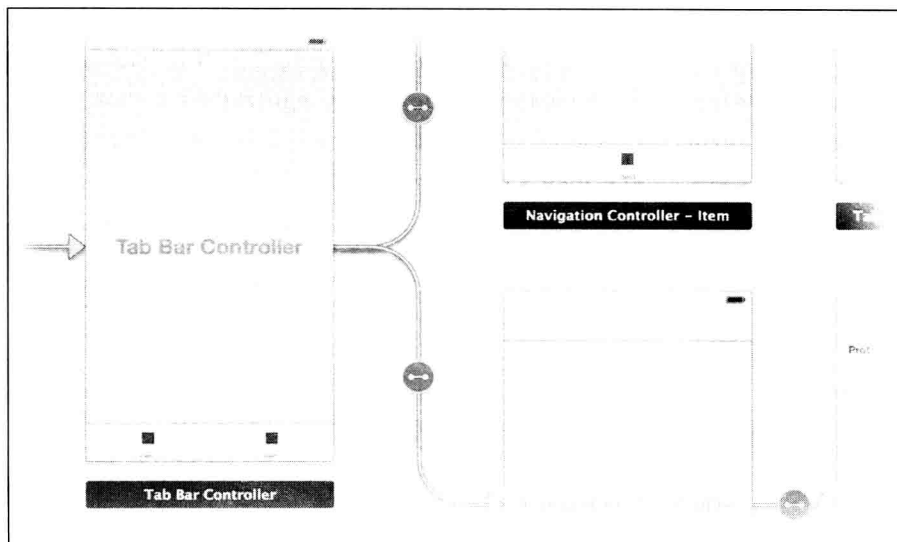


图 5-4 现在的 Tab Bar 控制器里面有两个 tab

Tab Bar 控制器现在有两个选项卡了，于是这个控制器也就可以在 PrepareTVC 与 ShopTVC 这两个表格视图之间切换了。在创建子类并将其设置给对应的表格视图之前，我们还需要做一些设定，使用户能够分辨这两个选项卡。

请按下列步骤修改 Grocery Dude，为 Tab Bar 添加图标：

1. 从 <http://www.timroadley.com/LearningCoreData/TabBarIcons.zip> 下载 Tab Bar 图标，并将其解压缩。
2. 把名叫 **Images.xcassets** 的 asset catalog 选中。
3. 像图 5-5 这样，把下载好的 Tab Bar 图标拖放到 **LaunchImage** 下方。

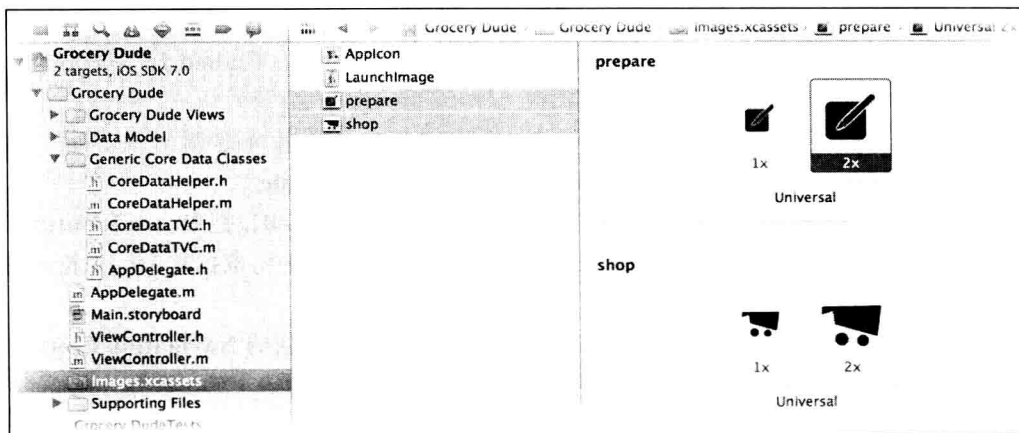


图 5-5 Tab Bar 控制器的图标

请按下列步骤修改 Grocery Dude，以配置两个选项卡：

1. 选中 **Main.storyboard**。
2. 在 **Items** 表格视图旁边的 **Navigation Controller** 里选中 **Tab Bar Item**。
3. 把 **Bar Item** 的 **Title** 设为 **Prepare**，把 **Bar Item** 的 **Image** 设为 **prepare**，如图 5-6 所示。

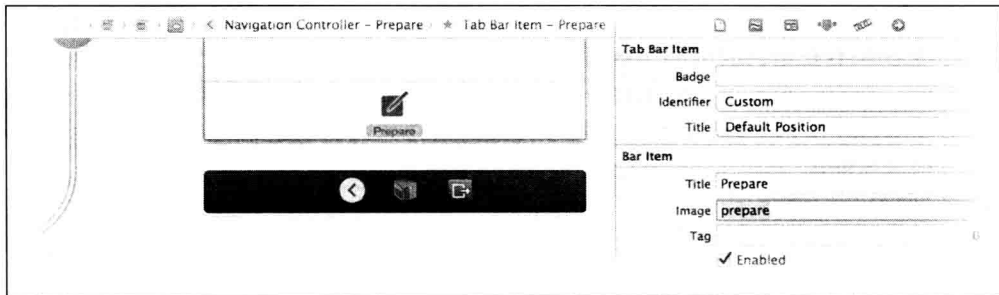


图 5-6 设置 Prepare tab

4. 在 **Grocery Dude** 表格视图旁边的 **Navigation Controller** 里选中 **Tab Bar Item**。
5. 按照第 3 步中的做法，把 **Bar Item** 的 **Title** 设为 **Shop**，把 **Bar Item** 的 **Image** 设为 **shop**。
6. 运行应用程序。现在应该可以在表格视图之间切换了，如图 5-7 所示。不过目前的表格中还没有数据。若发现 **Prepare** 和 **Shop** 这两个选项卡的顺序与图中不符，那么可在故事板的 **Tab Bar** 控制器里通过拖曳来摆正二者的顺序。

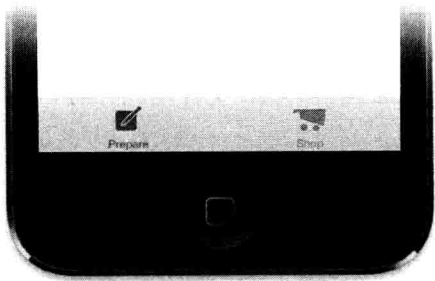


图 5-7 用户可通过 Grocery Dude 的 Tab Bar 控制器在不同的表格视图之间切换

5.5 AppDelegate 的 CoreDataHelper 实例

由于范例程序经常要用到 CoreDataHelper 类型的共享实例，所以我们需要在 AppDelegate.h 中把 AppDelegate.m 中的 cdh 代码公布出来，使其他代码可以访问它。程序清单 5-6 中的这行代码可将 cdh 公布出来。

程序清单5-6 AppDelegate.h文件中的cdh

```
-(CoreDataHelper*)cdh;
```

请按下列步骤修改 Grocery Dude，以公布 cdh 方法：

1. 把程序清单 5-6 中的代码添加到 AppDelegate.h 文件底部，并放在 @end 上方。

cdh 方法目前的实现方式是返回 CoreDataHelper 类型的共享实例。这种实现方式在多线程环境下并不安全，因为多个线程可能会各自实例化 CoreDataHelper，从而出现不止一个实例。程序清单 5-7 列出了修改后的版本。新版代码会把实例化 CoreDataHelper 的操作放在 dispatch_once 里面执行，这样就解决了线程安全问题。这么做可以确保在应用程序的生命期中，CoreDataHelper 只会实例化一次。

程序清单5-7 AppDelegate.m文件中的cdh方法

```
-(CoreDataHelper*)cdh {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    if (!_coreDataHelper) {
        static dispatch_once_t predicate;
        dispatch_once(&predicate, ^{
            _coreDataHelper = [CoreDataHelper new];
        });
        [_coreDataHelper setupCoreData];
    }
    return _coreDataHelper;
}
```

请按下列步骤修改 Grocery Dude，以实现线程安全的 cdh 方法：

1. 用程序清单 5-7 把 AppDelegate.m 文件里原有的 cdh 方法替换掉。

5.6 创建 PrepareTVC

为了给 **Prepare** 选项卡中的表格视图提供数据，我们需要从 CoreDataTVC 中继承新类，并将其命名为 PrepareTVC。这个子类会配置好获取请求，并把能够把放在购物清单中的货品显示出来。

请按下列步骤修改 Grocery Dude，以添加 PrepareTVC 类：

1. 在 **Grocery Dude** 组上面右击鼠标，创建名为 **Grocery Dude Table View Controllers** 的新组。
2. 确保 **Grocery Dude Table View Controllers group** 处于选中状态。
3. 点击 **File > New > File...** 菜单项。
4. 新建 **iOS > Cocoa Touch > Objective-C class**，并点击 **Next** 按钮。

5. 把 **Subclass of** 设为 CoreDataTVC，把 **Class** 名称设为 PrepareTVC。点击 **Next** 按钮。

6. 记得在 **Targets** 中勾选 **Grocery Dude**，然后点击 **Create** 按钮，在 **Grocery Dude** 项目的文件夹里创建类文件。

PrepareTVC 的头文件没有太多内容。程序清单 5-8 中唯一值得注意的地方是该类采用了 `UIActionSheetDelegate` 协议。此外，还有个用来表示动作表（`action sheet`）的实例变量。用户只需在 **Prepare** 选项卡中点击某个货品，即可将其放在 **Shop** 选项卡的购物清单里；同时我们还将在 **Prepare** 选项卡中实现“**Clear**”按钮，使用户能够通过它来彻底清空购物清单。但为了防止用户不小心触碰此按钮，我们需要在清空购物清单之前弹出动作表，令用户确认该操作。程序清单 5-8 中的 `clearConfirmActionSheet` 特性正是为了实现这一功能而设的。

程序清单5-8 PrepareTVC.h文件

```
#import <UIKit/UIKit.h>
#import "CoreDataTVC.h"
@interface PrepareTVC : CoreDataTVC <UIActionSheetDelegate>
@property (strong, nonatomic) UIActionSheet *clearConfirmActionSheet;
@end
```

请按下列步骤修改 **Grocery Dude**，以便配置 **PrepareTVC**：

1. 用程序清单 5-8 将 **PrepareTVC.h** 中的原有代码替换掉。

接下来我们要编写 **CoreDataTVC** 子类的实现文件了，该文件将分为 **DATA**、**VIEW** 及 **INTERACTION** 这三个部分。

5.6.1 DATA

在 **PrepareTVC** 实现文件的 **DATA** 部分中，只有一个名叫 `configureFetch` 的方法。该方法根据自定义的 `NSFetchRequest` 来创建 `NSFetchedResultsController`。此外，它也会把 **PrepareTVC** 设为 `NSFetchedResultsController` 的委托。`NSFetchedResultsControllerDelegate` 所要求的方法已经在超类 **CoreDataTVC** 里面实现过了，所以，**PrepareTVC** 这个子类无需再去实现一遍。相关代码如程序清单 5-9 所示。

程序清单5-9 PrepareTVC.m文件的DATA部分

```
#import "PrepareTVC.h"
#import "CoreDataHelper.h"
#import "Item.h"
#import "Unit.h"
#import "AppDelegate.h"

@implementation PrepareTVC
#define debug 1

#pragma mark - DATA
```

```

- (void)configureFetch {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];

NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:@"Item"];

request.sortDescriptors =
    [NSArray arrayWithObjects:
        [NSSortDescriptor sortDescriptorWithKey:@"locationAtHome.storedIn"
            ascending:YES],
        [NSSortDescriptor sortDescriptorWithKey:@"name"
            ascending:YES],
        nil];
[request setFetchBatchSize:50];
self.frc =
    [[NSFetchedResultsController alloc] initWithFetchRequest:request
        managedObjectContext:cdh.context
        sectionNameKeyPath:@"locationAtHome.storedIn"
        cacheName:nil];

self.frc.delegate = self;
}
@end

```

对 `configureFetch` 中的大部分代码应该已经比较熟悉了，因为其主要目标就是为了创建 `NSFetchRequest`，而它的创建方式我们已经在第2章中讨论过了。剩下的那些代码会把“获取”操作依照由 `setFetchBatchSize` 所指定的大小来分批处理，并用 `NSFetchedResultsController` 实例来配置 `self.frc`。想创建 `NSFetchedResultsController`，需要有下列四样东西：

- ❑ `NSFetchRequest` 实例。在本例中，我们用早前章节讲过的办法，在 `configureFetch` 方法开头将 `request` 创建出来。
- ❑ `NSManagedObjectContext` 实例。在本例中，我们通过 `AppDelegate` 里名为 `cdh` 的便捷方法来获取该实例。
- ❑ 表示 `sectionNameKeyPath` 的字符串。该字符串的值是实体中的某个属性 `key`，它用于将表格视图划分成不同的部分。在本例中，我们使用的字符串是 `locationAtHome.storedIn`，意思是要按照货品在用户家中的摆放位置来将表格视图划分为数个部分。一定要注意：该值必须和 `NSFetchRequest` 里首个 `NSSortDescriptor` 所使用的值相符。
- ❑ 表示缓存的字符串。虽说本例并未提供该字符串，但假如要提供的话，那就得保证该字符串在整个应用程序范围内是唯一的。

请按下列步骤修改 `Grocery Dude`，以便实现 `PrepareTVC` 的 `DATA` 部分：

1. 用程序清单 5-9 中的代码把 PrepareTVC.m 里原有的代码全都替换掉。

5.6.2 VIEW

PrepareTVC.m 里面的大部分操作都在 VIEW 部分中完成。这个部分里面基本上都是 UITableViewDataSource 协议中的方法，另外还有个 viewDidLoad 方法。相关代码如程序清单 5-10 所示。

程序清单5-10 PrepareTVC.m文件的VIEW部分

```
#pragma mark - VIEW
- (void)viewDidLoad {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }

    [super viewDidLoad];
    [self configureFetch];
    [self performFetch];
    self.clearConfirmActionSheet.delegate = self;

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(performFetch)
                                             name:@"SomethingChanged"
                                             object:nil];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }

    static NSString *cellIdentifier = @"Item Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:cellIdentifier
                                     forIndexPath:indexPath];

    cell.accessoryType = UITableViewCellAccessoryDetailButton;
    Item *item = [self.frc objectAtIndex:indexPath:indexPath];
    NSMutableString *title = [NSMutableString stringWithFormat:@"%i %i %i",
                                     item.quantity, item.unit.name, item.name];
    [title replaceOccurrencesOfString:@"(null)"
                               withString:@""
                               options:0
                               range:NSMakeRange(0, [title length])];

    cell.textLabel.text = title;

    // make selected items orange
    if ([item.listed boolValue]) {
        [cell.textLabel setFont:[UIFont
                                fontWithName:@"Helvetica Neue" size:18]];
        [cell.textLabel setTextColor:[UIColor orangeColor]];
    }
}
```



```

    else {
        [cell.textLabel setFont:[UIFont
            fontWithName:@"Helvetica Neue" size:16]];
        [cell.textLabel setTextColor:[UIColor grayColor]];
    }
    return cell;
}
- (NSArray*)sectionIndexTitlesForTableView:(UITableView *)tableView {
    if (debug==1) {
        NSLog(@"Running %@", NSStringFromClass(_cmd));
    }
    return nil; // we don't want a section index.
}

```

viewDidLoad 方法负责调用 **configureFetch** 及 **performFetch** 方法，以便驱动表格视图。另外，**PrepareTVC** 也把自己设为 **clearConfirmActionSheet** 的 **delegate**，这样的话，稍后用户想要清空整份购物清单时，就可以向其弹出确认框了。该方法最后一行代码用于监听 **SomethingChanged** 通知。这项操作使得应用程序其他地方的代码可在必要时触发 **re-fetch**（重新获取）事件，比方说，当 **Core Data Stack** 完全重置时，就可以这么做。

cellForRowAtIndexPath 方法负责提供需要显示在每个表格视图 单元格中的数据。我们用 **dequeueReusableCellWithIdentifier** 来优化表格视图 单元格的创建过程。其后，该方法会判断货品是否列在 **Shop** 选项卡中，并会根据判断结果采用不同的颜色来显示它。另外，如果用户没有输入某件货品的名称，那么 **cellForRowAtIndexPath** 就会碰到 (**null**)，于是，它会用一段代码把这种不美观的值替换掉。

sectionIndexTitlesForTableView 方法覆写了 **CoreDataTVC** 中的同名方法。本方法返回 **nil**，这种返回值意味着我们将会禁用部分索引。假如不覆写此方法，那就会使用超类中的实现代码，而这份实现代码将会启用部分索引。

请按下列步骤修改 **Grocery Dude**，以便实现 **VIEW** 部分：

1. 把程序清单 5-10 中的代码添加到 **PrepareTVC.m** 底部，并放在 **@end** 上方。
2. 选中 **Main.storyboard**。
3. 选定 **Items** 表格视图控制器。
4. 通过 **Identity Inspector** 界面（可按“**Option+⌘+3**”组合键调出该界面），将 **Items** 表格视图控制器的 **Custom Class** 设为 **PrepareTVC**，如图 5-8 所示。

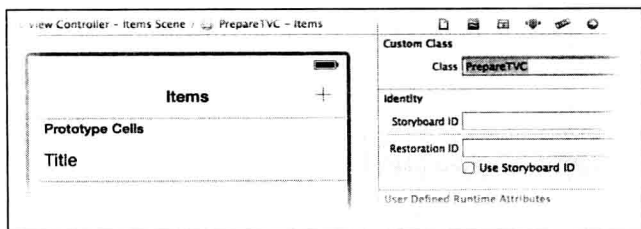


图 5-8 PrepareTVC 表格视图控制器

PrepareTVC 已经可以显示数据了，但在本章开头，笔者曾经叫大家把应用程序删了，所以现在持久化存储区里面还没有数据。

请按下列步骤修改 **Grocery Dude**，以插入一些供测试用的对象：

1. 把 `#import "LocationAtHome.h"` 添加到 `AppDelegate.m` 文件顶部。
2. 把 `#import "LocationAtShop.h"` 添加到 `AppDelegate.m` 文件顶部。
3. 修改 `AppDelegate.m` 文件的 `demo` 方法，用程序清单 5-11 中的代码替换原有代码。这段代码采用早前章节所讲的办法来插入测试数据。

程序清单5-11 AppDelegate.m文件中的demo方法

```

CoreDataHelper *cdh = [self cdh];
NSArray *homeLocations = [NSArray arrayWithObjects:
    @"Fruit Bowl",@"Pantry",@"Nursery",@"Bathroom",@"Fridge",nil];
NSArray *shopLocations = [NSArray arrayWithObjects:
    @"Produce",@"Aisle 1",@"Aisle 2",@"Aisle 3", @"Deli",nil];
NSArray *unitNames = [NSArray arrayWithObjects:
    @"g",@"pkt",@"box",@"ml",@"kg",nil];
NSArray *itemNames = [NSArray arrayWithObjects:
    @"Grapes",@"Biscuits",@"Nappies",@"Shampoo",@"Sausages",nil];

    int i = 0;
    for (NSString *itemName in itemNames) {
        LocationAtHome *locationAtHome =
            [NSEntityDescription
             insertNewObjectForEntityForName:@"LocationAtHome"
             inManagedObjectContext:cdh.context];
        LocationAtShop *locationAtShop =
            [NSEntityDescription
             insertNewObjectForEntityForName:@"LocationAtShop"
             inManagedObjectContext:cdh.context];
        Unit *unit =
            [NSEntityDescription insertNewObjectForEntityForName:@"Unit"
             inManagedObjectContext:cdh.context];

        Item *item =
            [NSEntityDescription insertNewObjectForEntityForName:@"Item"
             inManagedObjectContext:cdh.context];

        locationAtHome.storedIn = [homeLocations objectAtIndex:i];
        locationAtShop.aisle = [shopLocations objectAtIndex:i];
        unit.name = [unitNames objectAtIndex:i];
        item.name = [itemNames objectAtIndex:i];

        item.locationAtHome = locationAtHome;
        item.locationAtShop = locationAtShop;
        item.unit = unit;

        i++;
    }
    [cdh saveContext];

```

4. 运行一遍应用程序，以插入测试数据。正常的结果应该如图 5-9 所示。

5. 把 AppDelegate.m 文件 demo 方法体中的全部代码都删掉，以防再次插入同样的数据。

很棒！这个由 Core Data 所驱动的表格视图的基础部分已经做好了。接下来应该在 PrepareTVC 中添加一些应用程序用户所期望的功能了，比方说“删除货品”，或者“通过点击货品而将其添加到 Shop tab”等。

我们还需要实现 UITableViewDataSource 协议中的两个方法：

❑ **commitEditingStyle** 方法。当用户按住表格视图中的某个单元格并朝水平方向滑动时，该方法负责删除此件货品。其实，它不仅会删除相关的货品，而且还会把表格视图中的这一行也删掉。

❑ **didSelectRowAtIndexPath** 方法。该方法负责切换某件货品是否处在“listed”状态。处在“listed”状态中的货品也会出现在 Shop 选项卡中。另外，它还要保证新出现在购物清单中的货品既要处于“listed”状态，同时又不能标注为“collected”。假如某件货品标注为“collected”，那么它虽然仍显示在 Shop 选项卡里，但其右侧却会“打上对勾”。等用户点击 Shop 选项卡中的 Clear 按钮时，所有标柱为“collected”的货品都将从 Shop 选项卡里移除。

程序清单 5-12 列出了 VIEW 部分里的这两个新方法。



图 5-9 显示在 PrepareTVC 表格视图中的测试数据

程序清单5-12 PrepareTVC.m文件的VIEW部分（货品的选定与删除功能）

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        Item *deleteTarget = [self.frc objectAtIndexPath:indexPath];
        [self.frc.managedObjectContext deleteObject:deleteTarget];
        [self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    }
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    NSManagedObject *itemid =
        [[self.frc objectAtIndexPath:indexPath] objectAtIndex:0];
```

```

Item *item =
(Item*)[self.frc.managedObjectContext existingObjectWithID:itemid
                                             error:nil];

if ([item.listed boolValue]) {
    item.listed = [NSNumber numberWithInt:NO];
} else {
    item.listed = [NSNumber numberWithInt:YES];
    item.collected = [NSNumber numberWithInt:NO];
}
[self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                      withRowAnimation:UITableViewRowAnimationNone];
}

```

请按下列步骤修改 Grocery Dude，以便向原有的 VIEW 部分里添加新内容：

1. 把程序清单 5-12 中的代码添加到 PrepareTVC.m 文件的 VIEW 部分底部。
2. 选中 **Main.storyboard**。
3. 在 **Items** 表格视图控制器中选定 **Prototype Table View Cell**，然后打开 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面），如图 5-10 所示。

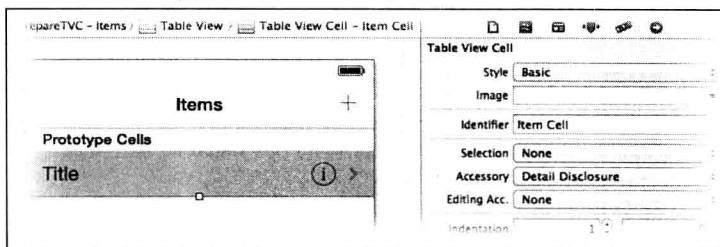


图 5-10 设定 Items 表格视图的 Prototype 单元格

4. 参照图 5-10，将表格视图 单元格的 **Selection** 设为 **None**。
5. 参照图 5-10，将表格视图 单元格的 **Accessory** 设为 **Detail Disclosure**。

再次运行应用程序，并观察程序有哪些新的行为。首先我们看到，当用户点击 Prepare 选项卡中的某项货品时，它会由橙色变为灰色。等本章稍后把 ShopTVC 表格视图配置好，你就会发现，Prepare 选项卡中显示为橙色的货品都会出现在 Shop 选项卡中——而这正是程序将货品添加到购物列表中的方式。

还有个新的行为，就是用户可以通过滑动操作来删除货品。你可以试着把“Biscuits”删掉。请注意：只有当程序保存上下文的时候，才会把删除操作的效果写入持久化存储区。

5.6.3 INTERACTION

PrepareTVC 的 INTERACTION 部分用于处理 Prepare 选项卡中新添加的 **Clear** 按钮。该按钮会把所有货品从 Shop 选项卡中移除。由于用户可能会不小心按到它，所以我们需要通过动作表来弹出确认对话框。

ShopTVC 表格视图负责显示购物时所用的货品清单，也就是说，它只会把 `item.listed` 等于 YES 的 `item` 显示出来。这就需要新建一份获取请求模板了，该模板不仅用来生成 ShopTVC 表格视图中的数据，而且当用户按下 Clear 按钮时，还会把货品从 “listed” 状态改为 “unlisted” 状态。

请按下列步骤修改 Grocery Dude，以便创建适用于购物清单的获取请求模板：

1. 选中 **Model 6.xcdatamodel**。
2. 将原有的获取请求模板名称从 **Test** 改为 **ShoppingList**，并按图 5-11 来配置。这种获取请求只会获取标注为 “listed” 的那些货品。对于 Boolean 类型的属性来说，1 一般表示 YES。

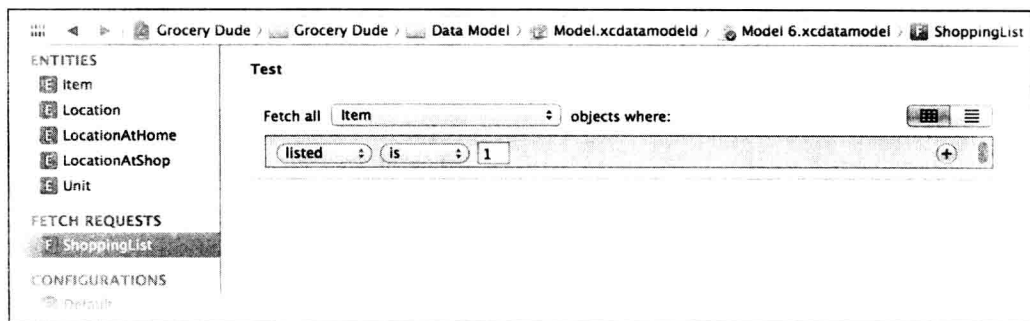


图 5-11 ShoppingList 只会获取标注为 listed 的那些货品

INTERACTION 部分里面将会有三个方法：

- ❑ **clear** 方法的返回值类型是 `IBAction`（interface builder action 的简称），该方法与 Items 表格视图中的 Clear 按钮相连。用户按下该按钮时，如果清单里至少有一件货品处于 “listed” 状态，那么就会弹出写有 “Clear entire shopping list?” 字样的动作表。
- ❑ **actionSheet** 是 `UIActionSheetDelegate` 协议中规定的方法，它用于处理用户对 Clear 操作的确认及取消。
- ❑ **clearList** 方法将会遍历清单中的所有货品，并将其标注为 “unlisted”。这在效果上也就相当于把货品从 Shop 选项卡中移走。

程序清单 5-13 列出了 INTERACTION 部分所需的三个新方法。

程序清单5-13 PrepareTVC.m文件的INTERACTION部分

```
#pragma mark - INTERACTION
- (IBAction)clear:(id)sender {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }
}
```

```

CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
NSFetchRequest *request =
    [cdh.model fetchRequestTemplateForName:@"ShoppingList"];
NSArray *shoppingList =
    [cdh.context executeFetchRequest:request error:nil];

if (shoppingList.count > 0) {

    self.clearConfirmActionSheet =
        [[UIActionSheet alloc] initWithTitle:@"Clear Entire Shopping List?"
                                             delegate:self
                                             cancelButtonTitle:@"Cancel"
                                             destructiveButtonTitle:@"Clear"
                                             otherButtonTitles:nil];
    [self.clearConfirmActionSheet
     showFromTabBar:self.navigationController.tabBarController.tabBar];
}
else {
    UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:@"Nothing to Clear"
                                         message:@"Add items to the Shop tab by
❖tapping them on the Prepare tab. Remove all items from the Shop
❖tab by clicking Clear on the Prepare tab"
                                         delegate:nil
                                         cancelButtonTitle:@"Ok"
                                         otherButtonTitles:nil];

    [alert show];
}
shoppingList = nil;
}

- (void)actionSheet:(UIActionSheet *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex {

    if (actionSheet == self.clearConfirmActionSheet) {
        if (buttonIndex == [actionSheet destructiveButtonIndex]) {
            [self performSelector:@selector(clearList)];
        }
        else if (buttonIndex == [actionSheet cancelButtonTitle]){
            [actionSheet dismissWithClickedButtonIndex:
             [actionSheet cancelButtonTitle] animated:YES];
        }
    }
}

- (void)clearList {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

CoreDataHelper *cdh =

```

```

    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    NSFetchRequest *request =
    [cdh.model fetchRequestTemplateName:@"ShoppingList"];
    NSArray *shoppingList =
    [cdh.context executeFetchRequest:request error:nil];

    for (Item *item in shoppingList) {
        item.listed = [NSNumber numberWithInt:NO];
    }
}

```

请按下列步骤修改 Grocery Dude，以便实现 INTERACTION 部分：

1. 把程序清单 5-13 中的代码添加到 PrepareTVC.m 文件底部，并放在 @end 上方，然后按 “⌘+S” 组合键保存这份类文件。
2. 选中 **Main.storyboard**。
3. 拖放一个 **Bar Button Item** 到 **Items** 表格视图的左上角。
4. 在 **Attributes Inspector** 界面（可按 “Option + ⌘ + 4” 组合键调出该界面）中，把 **Bar Item** 的 **Title** 设为 **Clear**，如图 5-12 所示。

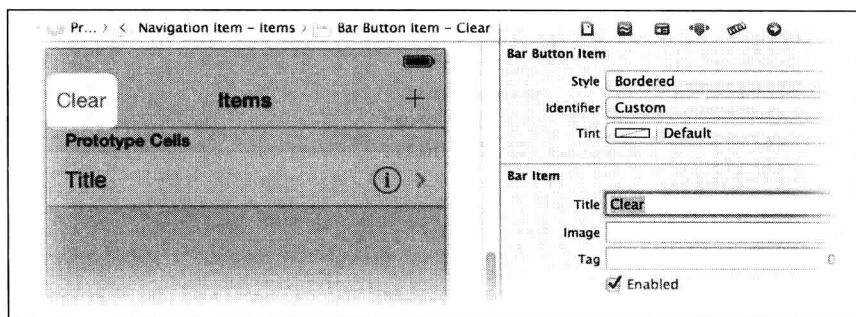


图 5-12 Clear 按钮将会把货品从购物清单中移除

5. 按住 **Control** 键，用鼠标从 Clear 按钮向 Items 表格视图底部的黄圆圈处拖一条线，从弹出的菜单里选择 **Sent Actions > clear:**。这将会把 Clear 按钮同 clear 方法连接起来。

再次运行应用程序，并观察这次有哪些新的行为。如果按下 Clear 按钮的时候，购物清单里没有货品处于受选状态，那么程序就会弹出通知，告诉用户应该先选中一些货品，然后再按 Clear 按钮。假如按下 Clear 按钮的时候，清单里确实有货品处在受选状态（也就是带橙色字样），那么用户就可以选择是确认该操作还是取消该操作。如果确认了 Clear 操作，那么所有橙色的货品都将会变为灰色。而到目前为止，Shop 选项卡里面还是空的，因为我们还未配置它。

5.7 创建 ShopTVC

ShopTVC 表格视图的用途是把购物清单中的货品按其在商店内的摆放位置罗列出来。ShopTVC 与 PrepareTVC 非常相似，某些方法的实现代码完全一致。

请按下列步骤修改 Grocery Dude，以创建 ShopTVC 类：

1. 确保 **Grocery Dude Table View Controllers** 组处于选中状态。
2. 点击 **File > New > File...** 菜单项。
3. 创建新的 **iOS > Cocoa Touch > Objective-C class**，并点击 **Next** 按钮。
4. 将 **Subclass of** 设为 CoreDataTVC，将 **Class** 名称设为 ShopTVC。点击 **Next** 按钮。
5. 记得在 Targets 中勾选 “Grocery Dude”，然后点击 **Create** 按钮，在 Grocery Dude 项目的目录下创建类文件。
6. 选中 **Main.storyboard**。
7. 采用与图 5-8 相同的方式来配置标题为 **Grocery Dude** 的表格视图控制器，将其 **Custom Class** 设为 ShopTVC。

ShopTVC 的头文件无须改动，但实现文件却需要更新。这次还是分成三个部分来写，它们分别是：DATA、VIEW 及 INTERACTION。

5.7.1 DATA

在 ShopTVC 实现文件的 DATA 部分里，只有一个名叫 configureFetch 的方法。这段代码与 PrepareTVC 类中的代码基本相似，唯一区别在于：这次要通过获取请求模板来限定获取到的数据，使其只包含标注为 “listed” 的货品。请注意，NSFetchedResultsController 使用的不是获取请求模板本身，而是它的一份拷贝。之所以要这样做，是因为我们要通过修改 NSFetchedRequest 来指定 NSSortDescriptor，但获取请求模板却不能直接编辑，所以只能先拷贝一份，然后在拷贝出来的这个 NSFetchedRequest 上面修改。程序清单 5-14 中的其他代码应该已为大家所熟悉了。

程序清单5-14 ShopTVC.m文件中的DATA部分

```
#import "ShopTVC.h"
#import "CoreDataHelper.h"
#import "Item.h"
#import "Unit.h"
#import "AppDelegate.h"

@implementation ShopTVC
#define debug 1

#pragma mark - DATA
- (void)configureFetch {
```



```

if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
NSFetchRequest *request =
    [[cdh.model fetchRequestTemplateName:@"ShoppingList"] copy];

request.sortDescriptors =
    [NSArray arrayWithObjects:
        [NSSortDescriptor sortDescriptorWithKey:@"locationAtShop.aisle"
            ascending:YES],
        [NSSortDescriptor sortDescriptorWithKey:@"name"
            ascending:YES],
        nil];
[request setFetchBatchSize:50];

self.frc =
    [[NSFetchedResultsController alloc] initWithFetchRequest:request
        managedObjectContext:cdh.context
        sectionNameKeyPath:@"locationAtShop.aisle"
        cacheName:nil];

self.frc.delegate = self;
}
@end

```

请按下列步骤修改 Grocery Dude，以实现 ShopTVC 的 DATA 部分：

1. 用程序清单 5-14 中的代码把 ShopTVC.m 里原有的代码全部替换掉。

5.7.2 VIEW

由于 VIEW 部分中的方法与 PrepareTVC 中的同名方法相似，所以这个部分看起来也应该比较熟悉才对。它与 PrepareTVC 的 VIEW 部分只有下面几个区别：

- ❑ 由于 ShopTVC 无须用户确认，所以 **viewDidLoad** 方法也就不再配置动作表委托了。
- ❑ 如果用户将某件货品标注为 “collected”，那么 **cellForRowAtIndexPath** 方法就会将其显示为绿色，并在旁边加上对勾符号（假如没有标注为 “collected”，则显示为橙色）。
- ❑ **sectionIndexTitlesForTableView** 方法没有变化。
- ❑ **didSelectRowAtIndexPath** 方法用于切换某项货品是否标注为 “collected”。用户点击表格中的某一行时，程序就会把该行内的货品打上对勾，以便稍后从购物清单里移走。
- ❑ ShopTVC 没有实现 **commitEditingStyle** 方法，这意味着用户不能在 Shop 选项卡中删除货品。假如我们允许用户在这个选项卡里删除货品，那就会与刚才说的

“打上对勾”功能相混淆，实际上我们是希望用户购入某项货品后，通过点击该货品给它“打上对勾”，这样就能在稍后将其从购物清单里面移走了。

程序清单 5-15 列出了 VIEW 部分中的代码。

程序清单5-15 ShopTVC.m文件的VIEW部分

```
#pragma mark - VIEW
- (void)viewDidLoad {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    [super viewDidLoad];
    [self configureFetch];
    [self performFetch];

    // Respond to changes in underlying store
    [[NSNotificationCenter defaultCenter] addObserver:self
                                                selector:@selector(performFetch)
                                                name:@"SomethingChanged"
                                                object:nil];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    static NSString *cellIdentifier = @"Shop Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:cellIdentifier
         forIndexPath:indexPath];

    Item *item = [self.frc objectAtIndex:indexPath.index];
    NSMutableString *title = [NSMutableString stringWithFormat:@"%d %s %s",
        item.quantity, item.unit.name, item.name];
    [title replaceOccurrencesOfString:@"(null)"
        withString:@" "
        options:0
        range:NSMakeRange(0, [title length])];
    cell.textLabel.text = title;

    // make collected items green
    if (item.collected.boolValue) {
        [cell.textLabel setFont:[UIFont
            fontWithName:@"Helvetica Neue" size:16]];
        [cell.textLabel setTextColor:
            [UIColor colorWithRed:0.368627450
                green:0.741176470
                blue:0.349019607 alpha:1.0]];
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
}
```

```

        else {
            [cell.textLabel setFont:[UIFont
                fontWithName:@"Helvetica Neue" size:18]];
            cell.textLabel.textColor = [UIColor orangeColor];
            cell.accessoryType = UITableViewCellAccessoryDetailButton;
        }
        return cell;
    }
- (NSArray*)sectionIndexTitlesForTableView:(UITableView *)tableView {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return nil; // prevent section index.
}
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    Item *item = [self.frc objectAtIndex:indexPath.index];
    if (item.collected.boolValue) {
        item.collected = [NSNumber numberWithInt:NO];
    }
    else {
        item.collected = [NSNumber numberWithInt:YES];
    }
    [self.tableView reloadRowsAtIndexPaths:
        [NSArray arrayWithObject:indexPath]
        withRowAnimation:UITableViewRowAnimationNone];
}

```

请按下列步骤修改 Grocery Dude，以添加 VIEW 部分：

1. 将程序清单 5-15 中的代码添加到 ShopTVC.m 文件底部，并放在 @end 上方。

5.7.3 INTERACTION

ShopTVC 的 INTERACTION 部分也用来处理 Clear 按钮，不过这个按钮和早前的不同。两者区别在于：Prepare 选项卡中的 Clear 按钮会清空 Shop 选项卡中的整份购物清单，而 Shop 选项卡中的 Clear 按钮则只会把已经拿到的（collected）货品从清单里删掉。用户在点击 Shop 选项卡中的 Clear 按钮时，我们也需要用名为 clear 的方法来处理此操作。由程序清单 5-16 可以看出，该方法要么会把标注为“collected”的那些货品删掉，要么会告知用户当前购物清单中没有可清理的货品。

程序清单5-16 ShopTVC.m文件中的INTERACTION部分

```

#pragma mark - INTERACTION
- (IBAction)clear:(id)sender {
    if (debug==1) {

```

```

NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if ([self.frc.fetchedObjects count] == 0) {

    UIAlertView *alert =
    [[UIAlertView alloc] initWithTitle:@"Nothing to Clear"
                                message:@"Add items using the Prepare tab"
                                delegate:nil
                                cancelButtonTitle:@"Ok" otherButtonTitles:nil];
    [alert show];
    return;
}

BOOL nothingCleared = YES;
for (Item *item in self.frc.fetchedObjects) {

    if (item.collected.boolValue)
    {
        item.listed = [NSNumber numberWithInt:NO];
        item.collected = [NSNumber numberWithInt:NO];
        nothingCleared = NO;
    }
}

if (nothingCleared) {
    UIAlertView *alert =
    [[UIAlertView alloc] initWithTitle:nil message:
     @"Select items to be removed from the list before pressing Clear"
     delegate:nil cancelButtonTitle:@"Ok" otherButtonTitles:nil];
    [alert show];
}
}
}

```

请按下列步骤修改 Grocery Dude，以实现 INTERACTION 部分：

1. 把程序清单 5-16 中的代码添加到 ShopTVC.m 文件底部，并放在 @end 上方，然后按 “⌘ + S” 组合键保存类文件。
2. 选中 **Main.storyboard**。
3. 拖放一个 **Bar Button Item** 到 **Grocery Dude** 表格视图的左上角，然后用早前设置 Items View 中的 Clear 按钮时所用的办法，将 **Bar Item** 的 **Title** 设为 **Clear**。
4. 按住 **Control** 键，由新的 Clear 按钮向 Grocery Dude 表格视图底部的黄圆圈拖一条线。然后在弹出式菜单中选择 **Sent Actions > clear:**。这样就把 Clear 按钮同 clear 方法链接起来了。
5. 在 **Grocery Dude** 表格视图控制器里选中 **Prototype Table View Cell**。
6. 将表格视图 单元格的 **Style** 设为 **Basic**。
7. 将表格视图 单元格的 **Selection** 设为 **None**。
8. 将表格视图 单元格的 **Accessory** 设为 **Detail Disclosure**。

运行应用程序，并在 Prepare 选项卡中添加一些货品，使之显示为橙色。切换到 Shop

选项卡，现在会看到，刚才那些货品已经出现在这个选项卡里了，而且已经按照它们所处的商店货架位置排好顺序了。点击 Shop 选项卡中的某件货品，可使它变为绿色，并且旁边还会出现一个“对勾”。按下 Shop 选项卡中的 Clear 按钮，即可把这些打上对勾的货品移除。返回 Prepare 选项卡，此时会看到已购入的那些货品（也就是刚才在 Shop 选项卡中打勾并清理掉的那些货品）现在已经不再显示为橙色了。

5.8 小结

核心功能实现出来之后，这个应用程序就已经初具规模了。Prepare 选项卡用于显示有可能会购买的货品，而 Shop 选项卡则用于列出正在购买的货品。在实现 PrepareTVC 类及 ShopTVC 类的过程中，我们运用了一些可重复使用的设计模式。可以通过修改 `configureFetch` 方法来把这些设计模式套用到自己的项目中。另外请注意，TVC 类里面的 `debug` 选项都是打开的，这将导致应用程序目前的运行速度比平常慢一些。

5.9 习题

请根据所学内容尝试下列操作。

1. 修改 PrepareTVC.m 文件中的 `configureFetch` 方法，把 `sectionNameKeyPath` 参数以及 `sortDescriptors` 的首选排序标准设为 **name**。如果代码实现得正确，那么货品将按照名称（name）来分组，并出现在表格视图的各个部分中。

2. 删掉 PrepareTVC.m 中的 `sectionIndexTitlesForTableView` 方法，并运行应用程序。这次你将在 Items 表格视图里看到部分索引。

3. 把程序清单 5-11 中的代码重新放入 AppDelegate 的 `demo` 方法中。运行三次应用程序，以便将每一条测试数据都插入三份。在 Items 表格视图里上下滚屏，并观察控制台的日志。在滚屏过程中，你应该会看到 `cellForRowAtIndexPath` 及 `titleForHeaderInSection` 反复出现在控制台的日志里。表格视图通过这种办法来高效地复用单元格，使用户以为所有数据都已载入内存，并可随时显示出来。

在阅读下一章之前，请把刚才做练习时对项目所做的修改都还原回去。



视图

人应该寻求事实，而不是寻求自己所期望的答案。

——阿尔伯特·爱因斯坦

第5章演示了如何创建由 Core Data 所驱动的表格视图。在此过程中，我们通过实现 CoreDataTVC 类而体会到了 NSFetchedResultsController 的好处。这个继承自 UITableViewController 的 CoreDataTVC 类是很有用的，我们又从中继承了两个子类，分别用于实现自定义的 Prepare 选项卡与 Shop 选项卡，以便在这两个选项卡中显示不同的信息。现在的 Grocery Dude 看起来更像个正规的 iOS 应用程序，而不再是那种为了演示 Core Data 理论所编写的习题。本章将要讲解怎样把表格视图中选定的托管对象传给视图。我们将配置自定义视图，以编辑选定的托管对象。在这一过程中，读者将学会如何配置 UITextField，使用户可通过该控件来修改托管对象的特性值。

6.1 概述

iOS 中最常用的标准界面元件是 UIView。作为 UIResponder 的子类，UIView 是 UIKit 里面功能比较强大的类，它提供了一种相当灵活的方式，可供应用程序向屏幕中显示内容。UIPickerView、UITextField、UIButton 及 UIScrollView 等界面元件都继承自 UIView。虽说 UITableView 也很适合用来显示或删除数据，但是，以 UIView 为基础来实现数据编辑功能会更好一些。把 UITextField 这样的 UIKit 组件添加到 View 中，用户就可以修改对象特性了。我们通常会专门使用某种自定义的 UIView 来编辑单个托管对象，Grocery Dude 程序也是如此。

如果想设计供用户编辑的 `UIView`，那么需要考虑用户在程序里进行编辑的同时，还可能在做什么事情。以 `Grocery Dude` 为例，用户在使用程序时，可能还推着购物车在超市里买东西，或是正在家中检查储藏的货品，看看有没有什么需要买的。这就意味着，用户只有一只手可以操作应用程序。所以我们应该尽量选用那种操作量较小的界面元件，使用户能非常方便地使用该程序。比方说，可以选用与选取器视图相结合的 `UITextField`，而不要单独使用 `UITextField`。另外，即便是在用户点击 `UITextField` 的时候显示出键盘，也会对提升用户体验大有帮助。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 `Grocery Dude` 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter05.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项，这样可以清除掉同名项目所残留的缓存。

6.2 范例程序所需的视图层级

在目前的视图层级中，处于根部的是 `Tab Bar` 控制器，它的下一级有两个表格视图。而再下一级，则有个空白的 `Item` 视图控制器，对于编辑 `item` 对象所要用到那些视图来说，这个 `Item` 视图控制器就是它们的中心。图 6-1 宏观地描述了范例程序所需的视图层级，到本章末尾，这些视图都会实现好。

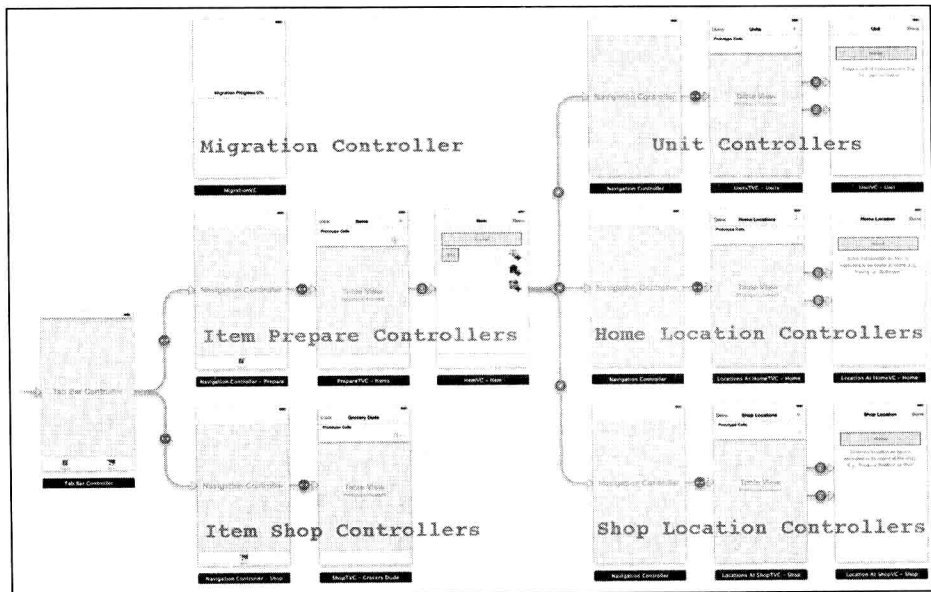


图 6-1 范例程序所需视图层级的概況

你应该已经熟悉 Prepare 选项卡和 Shop 选项卡了，二者分别用于显示 PrepareTVC 及 ShopTVC 这两个表格视图。当用户点击 PrepareTVC 表格视图右上角的“+”按钮后，程序会切换至 Item 视图控制器。切换的时候，需要创建新的托管对象，并将其 objectID 传给 Item 视图控制器。用户如果按下 PrepareTVC 表格视图中的 accessory detail 按钮，那么程序会把现有托管对象的 objectID 传给 Item 视图控制器。请注意，想在应用程序内传递托管对象，未必要通过 objectID 来做，但如果要在多个线程之间传递，那就需要用到它了。对于已经养成了使用 objectID 习惯的程序员来说，无需担心线程问题。

6.3 创建 ItemVC

根据 Item 实体而创建的托管对象中有很多特性可供用户编辑，例如“货品”（item）的“名称”（name）、“数量”（quantity）等。为了给程序添加这一功能，我们新建 UIViewController 的子类，并将其命名为 ItemVC。这个新类将同多个界面元件相连，使得用户可以通过它们来编辑 item。

请按下列步骤修改 Grocery Dude，以新建名为 ItemVC 的 UIViewController 子类：

1. 选中 **Grocery Dude View Controllers** 组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，并点击 **Next** 按钮。
4. 将 **Subclass of** 设为 UIViewController，将 **Class** 名称设为 ItemVC，并点击 **Next** 按钮。
5. 确保 Targets 中的“Grocery Dude”处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的目录中创建类文件。

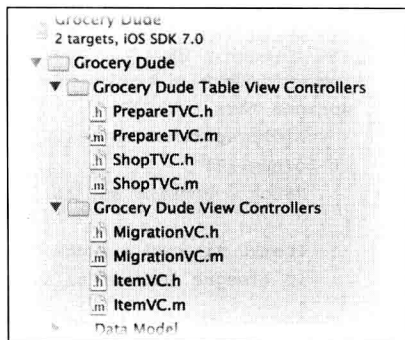


图 6-2 View 控制器与表格视图控制器的各个子类

执行完上述步骤之后，结果应该如图 6-2 所示。



与 CoreDataTVC 一样，笔者选了 ItemVC 这个类名来表示书中反复出现的 ItemViewController，以缩减文本长度。这与标准的 Objective-C 命名约定相违背，因而不值得提倡。笔者只是想节省篇幅，所以才选了 ItemVC 这个名字。一般情况下，类名应该清晰地表达出它的用途，并且不含歧义。

6.3.1 保存指向受选货品的引用

用户可以在 PrepareTVC 表格视图或 ShopTVC 表格视图中点击某项货品旁边的“accessory detail”按钮，以编辑该货品。程序将用受选货品的 objectID 来设置 ItemVC 的 selectedItemID 特性，以准备 ItemVC 视图。程序清单 6-1 列出了这项新特性，同

时还列出了新版的 ItemVC 头文件代码。

程序清单6-1 ItemVC.h

```
#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
@interface ItemVC : UIViewController <UITextFieldDelegate>
@property (strong, nonatomic) NSString *selectedItemID;
@end
```

请按下列步骤修改 Grocery Dude，以配置 ItemVC 的头文件：

1. 用程序清单 6-1 中的代码把 ItemVC.h 里原有的代码替换掉。

6.3.2 把受选货品传给 ItemVC

在 Grocery Dude 程序中，有两种方式用于把 item 传给 ItemVC。对于新创建的 item 来说，可使用 PrepareTVC 的 prepareForSegue 方法；而对于现有的 item 来说，则可使用 PrepareTVC 及 ShopTVC 的 accessoryButtonTappedForRowWithIndexPath 方法。程序清单 6-2 列出了这两个方法。

程序清单6-2 PrepareTVC.m和ShopTVC.m文件所用的prepareForSegue
及accessoryButtonTappedForRowWithIndexPath方法

```
#pragma mark - SEGUE
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }

    ItemVC *itemVC = segue.destinationViewController;
    if ([segue.identifier isEqualToString:@"Add Item Segue"])
    {
        CoreDataHelper *cdh =
            [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
        Item *newItem =
            [NSEntityDescription insertNewObjectForEntityForName:@"Item"
                inManagedObjectContext:cdh.context];

        NSError *error = nil;
        if (![cdh.context
            obtainPermanentIDsForObjects:[NSArray arrayWithObject:newItem]
            error:&error]) {
            NSLog(@"Couldn't obtain a permanent ID for object %@", error);
        }
        itemVC.selectedItemID = newItem.objectID;
    }
    else {
        NSLog(@"Unidentified Segue Attempted!");
    }
}
```

```

- (void)tableView:(UITableView *)tableView
accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

ItemVC *itemVC =
[self.storyboard instantiateViewControllerWithIdentifier:@"ItemVC"];
itemVC.selectedItemID =
[[self.frc objectAtIndex:indexPath.indexPath] objectID];
[self.navigationController pushViewController:itemVC animated:YES];
}

```

每逢有新 item 作为 Add Item Segue 的一部分插入程序时，我们就立刻调用相关方法，以获取该对象的永久 ID。假如不这么做的话，那么本书后面在引入多个上下文并与 iCloud 及 Web 服务相集成时，就可能会出问题，而此处的做法则能提前防止这些问题。

请按下列步骤修改 Grocery Dude，以配置 PrepareTVC 及 ShopTVC 的实现文件：

1. 把 import "ItemVC.h" 添加到 PrepareTVC.m 及 ShopTVC.m 文件顶部。我们必须引入 ItemVC.h 头文件，才能使这两个视图知道它们所要切换到的 ItemVC 到底是什么。
2. 将程序清单 6-2 中的代码添加到 PrepareTVC.m 文件底部，并放在 @end 语句上方。
3. 将程序清单 6-2 中的代码添加到 ShopTVC.m 文件底部，并放在 @end 语句上方。
4. 从 ShopTVC.m 文件里删掉 prepareForSegue 方法。由于用户并不需要在选项卡里新建 item，所以 ShopTVC 类无须包含该方法。

6.3.3 配置 ScrollView 及 UITextField

起初，输入焦点会停留在用于编辑 name 及 quantity 值的那两个 UITextField 对象上。到了下一章，我们就会制作自定义的 UITextField 对象，并把显示 UIPickerView 的功能加入其中。那些特殊的 UITextField 对象使用户能够从 picker 中选取货品的“计量单位名称”(unit.name)、“在家中的位置”(locationAtHome.storedIn)以及“在商店中的位置”(locationAtShop.aisle)。本章打算制作的 Item 视图其版式如图 6-3 所示。图中所有文本框都包含在滚动视图中，用户可在视图里上下滚动屏幕以查看它们。图中的三个按钮会将用户引入另外三个不同的界面，分别用于编辑货品的“计量单位”、“在家中的位置”以及“在商店中的位置”。



图 6-3 本章所要制作的 Item 视图控制器

若想在特性及相关视图之间建立链接，最简单的办法就是在故事板里配置界面元件。然后，就可以把视图拖放到对应类的头文件或实现文件上面了，假如特性是公开的，那就拖放到头文件上，若是私有的，则拖放到实现文件上。

请按下列步骤修改 Grocery Dude，以添加滚动视图：

1. 选中 **Main.storyboard**。
2. 选中 Item 视图控制器，并在 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面）中取消 **Extend Edges Under Top Bars** 选项的选取。
3. 拖放一个 **Scroll View** 到现有的 **Item** 视图里面，把它摆放在正中，以便占据整个 Item 视图。
4. 当 **Scroll View** 处于受选状态时，点击 **Editor > Resolve Auto Layout Issues > Add Missing Constraints** 菜单项。这样的话，在设备旋转时，滚动视图也会随之拉伸。
5. 如果 **Document Outline** 界面没有显示出来，那就点击 **Editor > Show Document Outline** 菜单项以显示该界面。
6. 当 **Scroll View** 处于受选状态时，点击 **Editor > Reveal in Document Outline** 菜单项，然后检查滚动视图所运用的 Constraints 是否与图 6-4 相符。

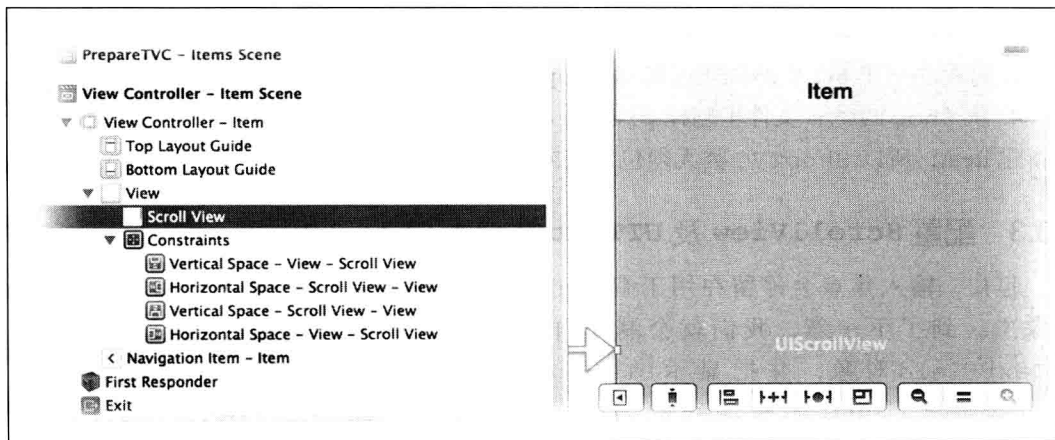


图 6-4 用 Auto Layout 制作出的新滚动视图

请按下列步骤修改 Grocery Dude，以添加 **Name** 及 **Quantity** 文本框：

1. 拖放一个 **Text Field** 到新 **Scroll View** 中的任意位置上，然后在 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面）中按照下述步骤配置它：
 - ❑ 把 **Font** 设为 **System Bold 17.0**。
 - ❑ 把 **Text Alignment** 设为 **Center**。
 - ❑ 把 **Placeholder Text** 设为 **Name**。
 - ❑ 把 **Border Style** 设为 **Line**（Attributes Inspector 界面以矩形来表示该选项）。
 - ❑ 把 **Capitalization** 设为 **Sentences**。

- ❑ 把视图区域中的 **Background** 设为 **Other > Crayons > Mercury** (Mercury 指的是第二个颜色最淡的灰色蜡笔)。
- 2. 用 **Size Inspector** 界面 (可按 “**Option + ⌘ + 5**” 组合键调出该界面) 把文本框的 **Height** 调整为 **48**。
- 3. 按住 **Option** 键, 将文本框向下拖动, 这样就能复制出新的文本框了。
- 4. 在 **Attributes Inspector** 界面 (可按 “**Option + ⌘ + 4**” 组合键调出该界面) 中按照下述步骤配置这个新复制出来的文本框:
 - ❑ 把 **Placeholder Text** 设为 **Qty**。
 - ❑ 把 **Keyboard** 设为 **Decimal Pad**。
 - ❑ 勾选 **Clear when editing begins** 选项。
- 5. 用 **Size Inspector** 界面 (可按 “**Option + ⌘ + 5**” 组合键调出该界面) 把 **Qty** 文本框的 **Width** 调整为 **60**。
- 6. 将 **Name** 文本框拓宽到“边界基准线” (edge guide), 并使之与该线对齐, 如图 6-5 所示。

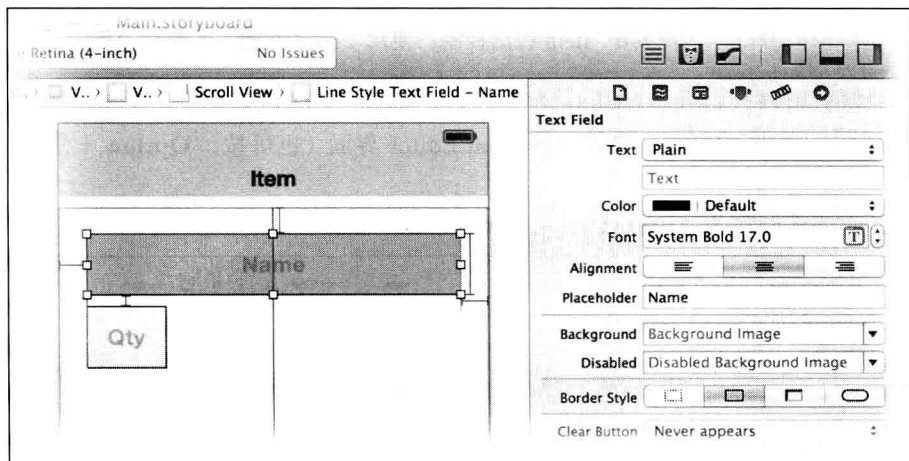


图 6-5 排布 Name 与 Qty 文本框的位置

7. 一次选中两个文本框 (按住键再点击第二个文本框), 然后点击 **Editor > Resolve Auto Layout Issues > Add Missing Constraints**, 这将确保文本框能在设备旋转时自动调整大小。设置好的结果如图 6-5 所示。

想要根据 Scroll 视图与文本框之间的链接来生成代码, 就必须配置 Item 视图控制器的 **Custom Class**。配置好之后, 就可以在 Assistant Editor 界面中把用户界面元件直接拖放到 ItemVC 的头文件上面了。

请按下列步骤修改 Grocery Dude, 以配置 Item 视图:

1. 选中 **Item** 视图控制器。

2. 在 **Identity Inspector** 界面（可按 **Option + ⌘ + 3** 组合键调出该界面）中把 **Item** 视图控制器的 **Custom Class** 与 **Storyboard ID** 均设为 **ItemVC**。当用户在 **Items** 表格视图里点击 **Detail Disclosure** 按钮时，程序要根据 **Storyboard ID** 来引用相关的视图。设置好的结果应该如图 6-6 所示。

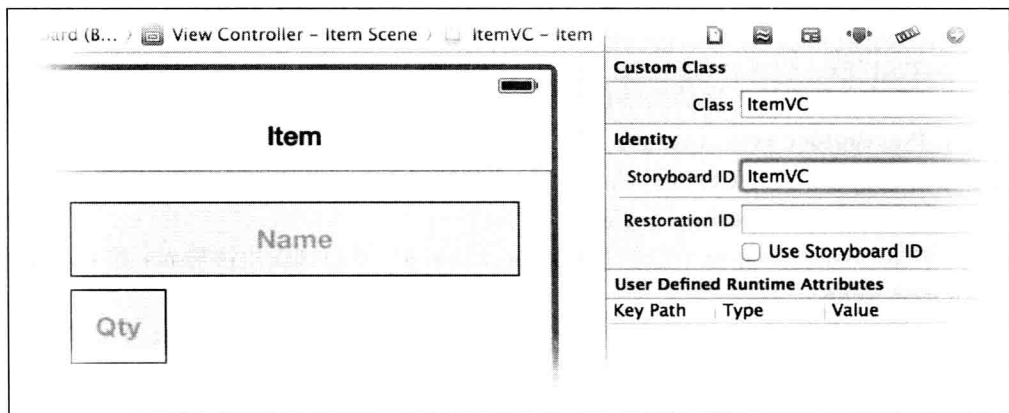


图 6-6 Item 视图控制器使用 ItemVC 类

请按下列步骤修改 **Grocery Dude**，以便根据 **Name** 及 **Qty** 文本框来创建相应的特性：

1. 选中 **Item** 视图控制器，调出 **Assistant Editor** 界面（也可按 “**Option + ⌘ + Return**” 组合键）。
2. 将界面顶部调整为 **Automatic > ItemVC.h**，如图 6-7 右上角所示。

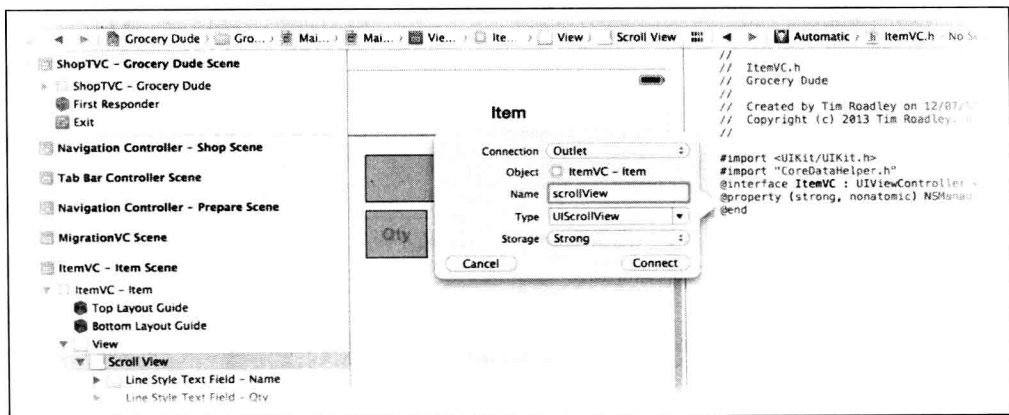


图 6-7 根据故事板中的元件来创建特性

3. 如果 **Document Outline** 界面没有显示出来，那就点击 **Editor > Show Document Outline** 以显示该界面。
4. 如图 6-7 左方所示，将 **ItemVC-Item Scene** 展开。

5. 按住 **Control** 键，从 **Scroll View** 向 `ItemVC.h` 的 `@end` 上方拖一条线，然后把特性的 **Name** 设为 `scrollView`，如图 6-7 中部所示。请确认 **Type** 是 `UIScrollView` 且 **Storage** 是 **Strong**。

6. 按住 **Control** 键，从 **Line Style Text Field-Name** 向 `ItemVC.h` 的 `@end` 上方拖一条线。把新特性的 **Name** 设为 `nameTextField`。请确认 **Type** 是 `UITextField` 且 **Storage** 是 **Strong**。

7. 按住 **Control** 键，从 **Line Style Text Field-Qty** 向 `ItemVC.h` 上方的 `@end` 拖一条线。把新特性的 **Name** 设为 `quantityTextField`。请确认 **Type** 是 `UITextField` 且 **Storage** 是 **Strong**。



提示 一般情况下，只有当本类以外的代码需要访问本类的某些特性时，我们才应该将其放到头文件里。但本书为了向读者强调这些特性，采用了与该建议相反的方式，也就是尽量把实现文件写得短小一些，并且尽量把特性放在头文件里。另外，在实现文件里还可以通过 `interface` 指令以私有的方式采用某协议。对于这种情况，Grocery Dude 项目为了使读者明白代码的意图，也将其放到了头文件里面。

执行完上述步骤之后，`ItemVC.h` 里面应该有 `scrollView`、`nameTextField` 及 `quantityTextField` 特性了。

6.3.4 实现 ItemVC

与 `PrepareTVC` 及 `ShopTVC` 相仿，`ItemVC` 的实现文件也分为若干个部分。一开始，它有下面这四个部分：

- ❑ **INTERACTION** 当用户点击界面背景时，该部分中有方法负责隐藏键盘。此外，当用户点击 **Done** 按钮时，该部分中也有方法负责把用户带回 `PrepareTVC` 表格视图界面。
- ❑ **DELEGATE:UITextField** 该部分中的方法可供实现了 `UITextFieldDelegate` 协议的类使用。这些方法可根据文本框的内容来设定货品名称，并确保程序不会接受长度为 0 的货品名称。
- ❑ **VIEW** 其中的方法负责用持久化存储区中的相关数据来刷新每个界面元件。
- ❑ **DATA** 其中的方法用于确保每件货品都设定了 `home location`（在家中的位置）和 `shop location`（在商店中的位置），即便 `location` 是 `unknown`（未知），也要设定。

6.3.5 INTERACTION

Grocery Dude 程序的每个视图控制器都要用到 **INTERACTION** 部分中的代码。`done` 方法与新的 **Done** 按钮相链接。用户点击按钮之后，程序会弹出 `ItemVC` 视图控制器，从而使用户回到早前的表格视图界面。`hideKeyboardWhenBackgroundIsTapped` 方法用

于配置 UITapGestureRecognizer, 以便在用户点击视图的背景时响应这一操作。用户点击视图界面的背景之后, hideKeyboard 方法将会调用 endEditing 以结束编辑状态, 从而将键盘隐藏起来。程序清单 6-3 列出了相关代码。

程序清单6-3 ItemVC.m文件的INTERACTION部分

```
#import "ItemVC.h"
@implementation ItemVC
#define debug 1

#pragma mark - INTERACTION
- (IBAction)done:(id)sender {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
    [self hideKeyboard];
    [self.navigationController popViewControllerAnimated:YES];
}
- (void)hideKeyboardWhenBackgroundIsTapped {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
    UITapGestureRecognizer *tgr =
        [[UITapGestureRecognizer alloc] initWithTarget:self
                                                    action:@selector(hideKeyboard)];

    [tgr set CancelsTouchesInView:NO];
    [self.view addGestureRecognizer:tgr];
}
- (void)hideKeyboard {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
    [self.view endEditing:YES];
}
@end
```

请按下列步骤修改 Grocery Dude, 以配置 INTERACTION 部分:

1. 把 **Standard Editor** 界面显示出来 (可按 “⌘ + Return” 组合键调出该界面)。
2. 用程序清单 6-3 中的代码把 ItemVC.m 文件里原有的代码全都替换掉, 然后保存类文件 (可按 “⌘ + S” 组合键)。
3. 选定 **Main.storyboard**。
4. 拖放一个 **Bar Button Item** 到 **Item** 视图右上角。
5. 在 **Attributes Inspector** 界面 (可按 “Option + ⌘ + 4” 组合键调出该界面) 中, 把 Bar Button Item 的 **Identifier** 设为 **Done**。设置好的结果如图 6-8 所示。

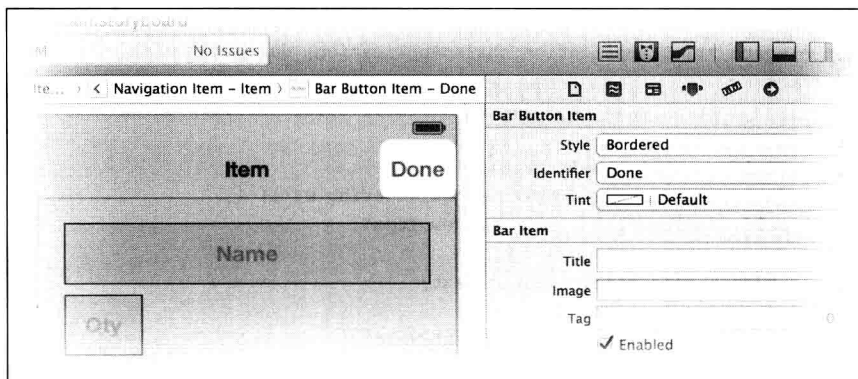


图 6-8 新的 Done 按钮

6. 按住 **Control** 键，由新的 **Done** 按钮向 **Item** 视图底部的黄圆圈拖一条线，然后选择 **Sent Actions > done**。

6.4 DELEGATE: UITextField

DELEGATE: UITextField 部分中的代码用于实现 `textFieldDidBeginEditing` 及 `textFieldDidEndEditing` 方法，二者都是 `UITextFieldDelegate` 协议中的可选方法。系统会在文本框获得输入焦点时调用前者，在失去焦点时调用后者。文本框获得焦点时，系统会将新货品的名称设置成长度为 0 的字符串，而当文本框失去焦点时，开发者则应根据文本框在失去焦点的那一刻所具备的内容来更新用户所选定的托管对象。程序清单 6-4 列出了相关代码。

程序清单6-4 ItemVC.m文件的DELEGATE:UITextField部分

```
#pragma mark - DELEGATE: UITextField
- (void)textFieldDidBeginEditing:(UITextField *)textField {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }
    if (textField == self.nameTextField) {

        if ([self.nameTextField.text isEqualToString:@"New Item"]) {
            self.nameTextField.text = @"";
        }
    }
}

- (void)textFieldDidEndEditing:(UITextField *)textField {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }
}
```



```

CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
Item *item =
    (Item*) [cdh.context existingObjectWithID:self.selectedItemID error:nil];

if (textField == self.nameTextField) {
    if ([self.nameTextField.text isEqualToString:@""]) {
        self.nameTextField.text = @"New Item";
    }
    item.name = self.nameTextField.text;
}
else if (textField == self.quantityTextField) {
    item.quantity =
        [NSNumber numberWithFloat:self.quantityTextField.text.floatValue];
}
}
}

```

请按下列步骤修改 Grocery Dude，以实现 DELEGATE: UITextField 部分：

1. 把 #import "AppDelegate.h" 及 #import "Item.h" 添加到 ItemVC.m 顶部。
2. 将程序清单 6-4 中的代码添加到 ItemVC.m 底部，并放在 @end 上方。

6.4.1 VIEW 部分

视图 (VIEW) 部分中有 4 个方法：

- ❑ **refreshInterface** 假如用户选中了某个托管对象，那么该方法就会用该对象里的值来刷新界面。
- ❑ **viewDidLoad** 该方法会将当前这个视图设置成文本框对象的委托。此外，它还会调用 INTERACTION 部分中的 hideKeyboardWhenBackgroundIsTapped 方法，该方法负责在用户点击界面背景时隐藏键盘。
- ❑ **viewWillAppear** 当系统即将把视图显示出来的时候，该方法会调用 refreshInterface 方法，以确保用户能够看到最新的数据。另外，当用户建新 item 时，该方法会立刻把键盘显示出来。
- ❑ **viewDidDisappear** 该方法用于在视图消失的时候保存上下文。虽说未必要在视图每次消失时都把上下文保存一遍，但定期将数据写入持久化存储区还是有好处的，这样可以防止由于设备电量耗尽而造成的数据丢失。等到把程序同 iCloud 及 Web 服务集成起来之后，保存上下文就显得更为重要了，因为那时开发者需要确保修改后的数据能够更新到应用程序以外的地方。

程序清单 6-5 列出了相关的代码。

程序清单6-5 ItemVC.m文件的VIEW部分

```

#pragma mark - VIEW
- (void)refreshInterface {
    if (debug==1) {

```

```

    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID
                error:nil];

    self.nameTextField.text = item.name;
    self.quantityTextField.text = item.quantity.stringValue;
}
}

- (void)viewDidLoad {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    [super viewDidLoad];
    [self hideKeyboardWhenBackgroundIsTapped];
    self.nameTextField.delegate = self;
    self.quantityTextField.delegate = self;
}

- (void)viewWillAppear:(BOOL)animated {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    [self refreshInterface];
    if ([self.nameTextField.text isEqual:@"New Item"]) {
        self.nameTextField.text = @" ";
        [self.nameTextField becomeFirstResponder];
    }
}

- (void)viewDidDisappear:(BOOL)animated {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    [cdh saveContext];
}
}

```

请按下列步骤修改 Grocery Dude，以实现 VIEW 部分：

1. 把程序清单 6-5 中的代码添加到 ItemVC.m 底部，并放在 @end 上方。
2. 将 Grocery Dude 程序从设备（或模拟器）中删除，以清空持久化存储区。
3. 点击 **Product > Clean** 菜单项。
4. 运行应用程序，并按下 Prepare tab 中的“+”按钮，以新建 item。
5. 将 item 的 **Name** 设为 **Coffee**，将 **Quantity** 设为 2，然后按 **Done** 按钮。

返回 Items 表格视图之后，你应该看不到刚才新建的 item，因为我们还没有把它在表

格视图中所处的部分配置好。假如能看见那个 item，那么在点击该 item 之后，它也会消失。出现这个问题的原因在于，该 item 没有设定 home location。此外，Shop 选项卡也会有这个问题，原因在于 item 没有设定 shop location。由于表格视图要根据 location 将货品显示到不同的部分中，所以这些 location 是相当重要的。为解决此问题，我们向 ItemVC 里添加 DATA 部分，并在其中编写两个新的方法。

6.4.2 DATA 部分

DATA 部分中会有两个方法，而这两个方法各自依赖于不同的获取请求模板。每个获取请求模板都会在相关的对象中，把 home location 或 shop location 能与 **..Unknown Location..** 精确匹配的那些对象找出来。Unknown Location 的开头加了两个点，这样可以确保经由 fetch（获取）的 NSSortDescriptor 排序之后，它总能显示在表格视图顶端。

请按下列步骤修改 Grocery Dude，以配置新的获取请求模板：

1. 选中 **Model 6.xcdatamodel**。
2. 添加名为 **UnknownLocationAtHome** 的获取请求。
3. 添加名为 **UnknownLocationAtShop** 的获取请求。
4. 配置名为 **UnknownLocationAtHome** 的获取请求模板，令其把 **storedIn** 为 **..Unknown Location..** 的全部 **LocationAtHome** 对象获取过来，如图 6-9 所示。你需要点击“+”按钮来为获取请求添加新的 criteria（标准）。

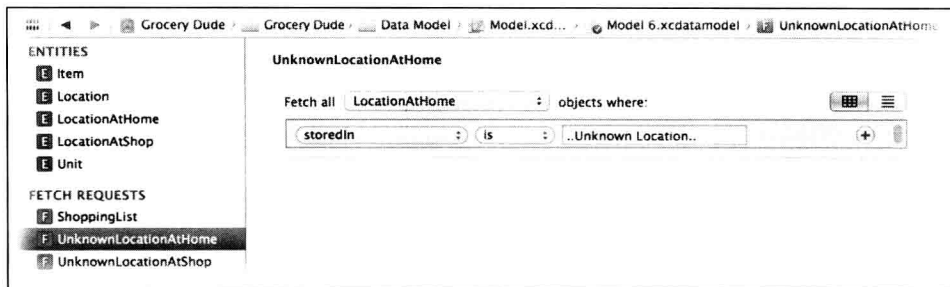


图 6-9 获取请求模板，用于获取 location 为 **..Unknown Location..** 的对象

5. 用类似的方式来配置名为 **UnknownLocationAtShop** 的获取请求模板，令其把 **aisle** 为 **..Unknown Location..** 的全部 **LocationAtShop** 对象获取过来。

下面这两个方法位于 DATA 部分中，它们将会用到刚才配置的获取请求模板：

- ❑ **ensureItemHomeLocationIsNotNull** 方法如果发现 `item.locationAtHome` 是 `nil`，那么就负责将 item 的 home location 设为 **..Unknown Location..**。
- ❑ **ensureItemShopLocationIsNotNull** 方法如果发现 `item.locationAtShop` 是 `nil`，那么就负责将 item 的 shop location 设为 **..Unknown Location..**。

程序清单 6-6 列出了相关代码。

程序清单6-6 ItemVC.m文件的DATA部分

```

#pragma mark - DATA
- (void)ensureItemHomeLocationIsNotNull {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID
                                error:nil];

    if (!item.locationAtHome) {
        NSFetchRequest *request =
            [[cdh model]
             fetchRequestTemplateForName:@"UnknownLocationAtHome"];
        NSArray *fetchObjects =
            [cdh.context executeFetchRequest:request error:nil];

        if ([fetchObjects count] > 0) {
            item.locationAtHome = [fetchObjects objectAtIndex:0];
        }
        else {
            LocationAtHome *locationAtHome =
                [NSEntityDescription
                 insertNewObjectForEntityForName:@"LocationAtHome"
                 inManagedObjectContext:cdh.context];
            NSError *error = nil;
            if (![cdh.context obtainPermanentIDsForObjects:
                [NSArray arrayWithObject:locationAtHome] error:&error]) {
                NSLog(@"Couldn't obtain a permanent ID for object %@",
                    error);
            }
            locationAtHome.storedIn = @"..Unknown Location..";
            item.locationAtHome = locationAtHome;
        }
    }
}

- (void)ensureItemShopLocationIsNotNull {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID
                                error:nil];

    if (!item.locationAtShop) {
        NSFetchRequest *request =
            [[cdh model]

```

```

        fetchRequestTemplateForName:@"UnknownLocationAtShop"];
    NSArray *fetchedObjects =
    [cdh.context executeFetchRequest:request error:nil];

    if ([fetchedObjects count] > 0) {
        item.locationAtShop = [fetchedObjects objectAtIndex:0];
    }
    else {
        LocationAtShop *locationAtShop =
        [NSEntityDescription
         insertNewObjectForEntityForName:@"LocationAtShop"
         inManagedObjectContext:cdh.context];
        NSError *error = nil;
        if (![cdh.context obtainPermanentIDsForObjects:
            [NSArray arrayWithObject:locationAtShop] error:&error]) {
            NSLog(@"Couldn't obtain a permanent ID for object %@",
                  error);
        }
        locationAtShop.aisle = @"..Unknown Location..";
        item.locationAtShop = locationAtShop;
    }
}
}
}

```

这两个方法首先要判断 `selectedItemID` 是不是 `nil`。如果不是 `nil`，那就声明 `cdh` 指针，并令其指向 `application delegate` 的 `CoreDataHelper` 实例。然后再通过该指针，用 `objectID` 来创建指向托管对象的 `item` 指针。假如 `item` 中已经有了相应的 `locationAtHome` 或 `locationAtShop` 对象，那就不执行任何操作。若是没有，则把 `item` 的 `locationAtHome` 或 `locationAtShop` 设为相应的 **..Unknown Location..** 对象。要是 **..Unknown Location..** 对象本身还没创建好，那就在第一次用到该对象时将其创建出来，并将其赋给 `item`。

请按下列步骤修改 `Grocery Dude`，以实现 `DATA` 部分：

1. 把 `#import "LocationAtHome.h"` 添加到 `ItemVC.m` 顶部。
2. 把 `#import "LocationAtShop.h"` 添加到 `ItemVC.m` 顶部。
3. 将程序清单 6-6 中的代码添加到 `ItemVC.m` 文件底部，并放在 `@end` 上方。
4. 修改 `ItemVC.m` 文件中的 `viewWillAppear` 方法，将下列代码放在 `[self refreshInterface]` 上方：

```

[self ensureItemHomeLocationIsNotNull];
[self ensureItemShopLocationIsNotNull];

```

5. 修改 `ItemVC.m` 文件中的 `viewDidDisappear` 方法，把下列代码放在声明 `cdh` 的那条语句上方：

```

[self ensureItemHomeLocationIsNotNull];
[self ensureItemShopLocationIsNotNull];

```

6. 再次运行应用程序，点击 **Coffee item** 右侧的 **accessory detail button**，然后点击 **Done** 按钮。结果应该如图 6-10 所示。如果没看到扩展指示器（disclosure indicator），并在程序还没设置 location 的时候就点击了 item，那么它可能会消失。假如发生了这种情况，只需再次运行应用程序即可。

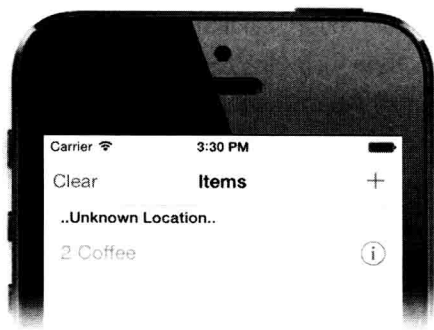


图 6-10 表格视图会把 location 为 ..Unknown Location.. 的 item 显示在同一个部分里

6.5 货品的计量单位、在家中的位置以及在商店中的位置

用户必须能够添加并编辑货品的“计量单位”（unit）、“在家中的位置”（home location）以及“在商店中的位置”（shop location）。实现此功能所采用的技术与早前实现“添加并编辑货品”这一功能时所采用的技术类似。也就是说，对于每一种对象类型，我们都要安排一个表格视图，并搭配一个自定义的 CoreDataTVC 子类，此外，还需再创建一个编辑此种对象所用的视图。

请按下列步骤修改 Grocery Dude，以添加通用的表格视图及视图：

1. 选定 **Main.storyboard**。
2. 向故事板中拖放一个新的 **Table View Controller**，并将其放在现有的 **Item** 视图右方。
3. 在新的 **Table View Controller** 里面选中 **Prototype Cell**，用 **Attributes Inspector** 界面（可按“**Option + Ⓔ + 4**”组合键调出该界面）将它的 **Reuse Identifier** 设为 **Cell**。
4. 选中新的 **Table View Controller**，点击 **Editor > Embed In > Navigation Controller** 菜单项。
5. 拖放一个 **Bar Button Item** 到新的 **Table View Controller** 左上角。
6. 将这个新 **Bar Button Item** 的 **Identifier** 设为 **Done**。
7. 拖放一个 **Bar Button Item** 到新的 **Table View Controller** 右上角。
8. 将这个新 **Bar Button Item** 的 **Identifier** 设为 **Add**。
9. 向故事板中拖放一个新的 **View Controller**，并把它放在新的 **Table View Controller** 右侧。
10. 按住 **Control** 键，由新 **Table View Controller** 的 **Prototype Cell** 向新的 **View Controller**

拖一条直线，并选择 **Selection Segue > push**。

11. 把新 segue 的 **Storyboard Segue Identifier** 设为 **Edit Object Segue**。
12. 按住 **Control** 键，由新 **Table View Controller** 的 **Add** 按钮（也就是“+”按钮）向新的 **View Controller** 拖一条线，并选择 **Action Segue > push**。
13. 把新 segue 的 **Storyboard Segue Identifier** 设为 **Add Object Segue**。
14. 拖放一个 **Bar Button Item** 到新的 **View Controller** 右上角。
15. 把新 **Bar Button Item** 的 **Identifier** 设为 **Done**。
16. 拖放一个新的 **Text Field** 到新 **View Controller** 的任意位置，然后在 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面）中按照下列步骤配置它：
 - ❑ 把 **Font** 设为 **System Bold 17.0**。
 - ❑ 把 **Text Alignment** 设为 **Center**。
 - ❑ 把 **Placeholder Text** 设为 **Name**。
 - ❑ 把 **Border Style** 设为 **Line**（Attributes Inspector 界面以矩形来表示该选项）。
 - ❑ 把 View 区域中的 **Background** 设为 **Other > Crayons > Mercury**（Mercury 指的是第二个颜色最淡的灰色蜡笔）。
17. 用 **Size Inspector** 界面（可按“**Option + ⌘ + 5**”组合键调出该界面）将文本框的 **Height** 设为 **48**。
18. 拓宽 **Name** 文本框，使之与“边线”（edge guide）对齐，如图 6-11 右侧所示。

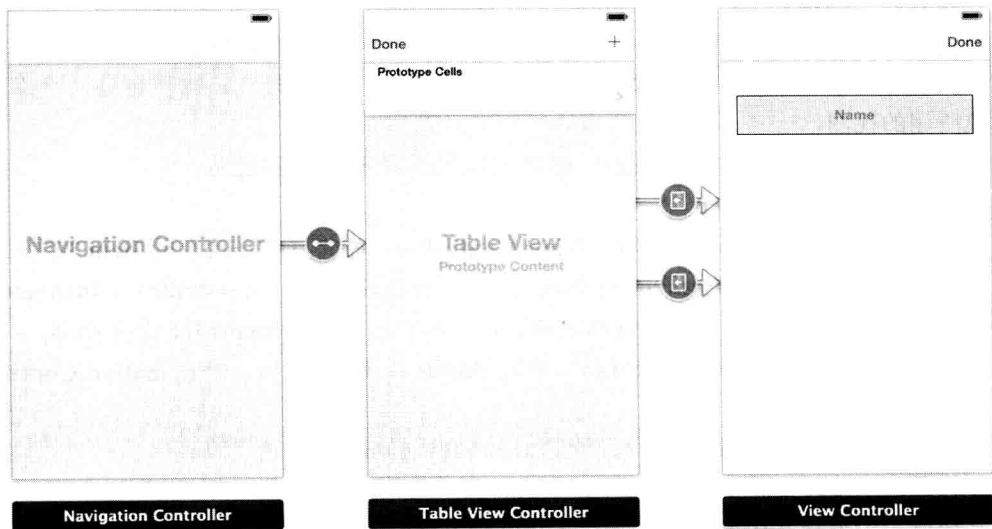


图 6-11 通用的控制器

准备好这些通用的控制器之后，我们就能够以此为基础来实现 unit、home location 及 shop location 的“添加 / 编辑”功能所需的那些控制器了。

请按下列步骤修改 Grocery Dude，以复制刚才制作的通用控制器：

1. 同时选中刚才制作的 **Navigation Controller**、**Table View Controller** 及 **View Controller**，然后点击 **Edit > Duplicate** 菜单项。
2. 把复制出来的这一组控制器拖曳到原有那一组的上方。
3. 按此方式再复制一组控制器，并依照图 6-12 把这三组控制器的位置摆放好。



图 6-12 三组通用的控制器

现在已经做好九个新的控制器了，如图 6-12 所示。其中，顶部三个用于添加或编辑货品的计量单位，中间三个用于添加或编辑货品在家中的位置，下面三个用于添加或编辑货品在商店中的位置。我们稍后会给 Item 视图里新建三个按钮，使用户能够通过它们来切换到这三组新的界面中。然而在这三个按钮尚未实现之时，Xcode 会警告开发者程序代码无法到达这些新的控制器，目前完全可以忽略这一警告。

6.5.1 添加与编辑计量单位

为了令程序代码能够“到达”(reach)这些新的控制器,我们必须给 Item 视图中添加新按钮。首先要添加的是 **Add Units** 按钮,该按钮会把程序切换到 Units 表格视图,而这个表格视图是嵌在 Navigation 控制器里面的。Units 表格视图与 Items 表格视图有些相似,它们都分别与另外一个视图相关,用户可在那个视图里编辑选定的对象(具体到本例,这个选定的对象指的就是某种计量单位(unit))。

请按下列步骤修改 Grocery Dude, 为按钮添加图标:

1. 从 http://www.timroadley.com/LearningCoreData/Icons_ItemVC.zip 下载按钮图标, 并将其解压缩。
2. 选定 asset catalog 里的 **Images.xcassets**。
3. 如图 6-13 所示, 把新的按钮图标拖放到 asset catalog 里面。

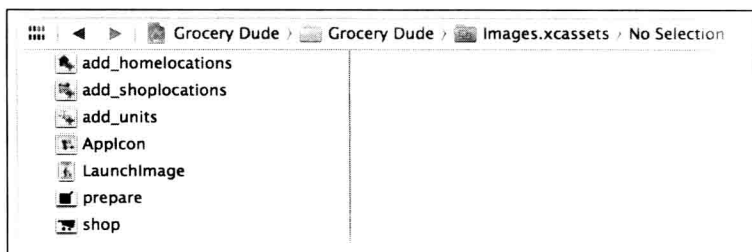


图 6-13 新按钮所需的图标

请按下列步骤修改 Grocery Dude, 以添加 **Add Unit** 按钮:

1. 选中 **Main.storyboard**。
2. 向 Item 视图中的 **Scroll View** 里拖放一个 **Button**, 摆在任意位置均可, 然后在 **Attributes Inspector** 界面(可按“**Option + ⌘ + 4**”组合键调出该界面)中按照下列步骤配置这个 Button:
 - ❑ 把 **Type** 设为 **Custom**。
 - ❑ 删掉“Button”字样的文本。
 - ❑ 将 **Image** 设为 **add_units**。
3. 在 **Size Inspector** 界面(可按“**Option + ⌘ + 5**”组合键调出该界面)中按照下列步骤配置 **Button**:
 - ❑ 将 **Width** 设为 **48**。
 - ❑ 将 **Height** 设为 **48**。
4. 按住 **Control** 键, 由新按钮向 **Navigation Controller** 拖一条线(早前我们曾经做了三个通用的表格视图控制器, 这个 Navigation 控制器位于顶部那个表格视图控制器的左侧), 然后选择 **Action Segue > modal**。
5. 在新的 modal segue 所指向的那个表格视图控制器中, 把 **Navigation Item Title** 设为

Units。

6. 在 Units 表格视图控制器中, 选定 **Prototype Cell**, 用 **Attributes Inspector** 界面 (可按 “**Option** + ⌘ + 4” 组合键调出该界面) 把 **Reuse Identifier** 设为 **Unit Cell**。

7. 在 Units 表格视图控制器所指向的那个 **View Controller** 中, 把 **Navigation Item Title** 设为 **Unit**。

8. 向 **Unit** 视图控制器中拖放一个 **Text View**, 令其居中, 并拓展其宽度, 使之恰好位于 **Name** 本框的正下方, 然后按如下步骤配置 Text 视图:

- ☐ 把 **Text** 的内容设为 **Enter a unit of measurement, E.g. ‘ml’, ‘pkt’ or ‘items’**。
- ☐ 把 **Color** 设为 **Light Grey Color**。
- ☐ 把 **Font** 设为 **System Bold 16.0**。
- ☐ 把 **Alignment** 设为 **Centered**。
- ☐ 取消 **Editable** 及 **Selectable** 选项的选择。

9. 选中 **Name Text Field** 与 **Text View**, 点击 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints** 菜单项。

假如现在运行应用程序并前往 **Item** 视图界面, 那就可以测试一下新的 **Add Unit** 按钮了。由于相关视图的代码还未实现, 所以为了使这些视图能正常运作, 我们现在应该创建 **CoreDataTVC** 的子类。要是没有这些代码的话, 那么对计量单位的名称所做的编辑操作就没有任何效果了, 而且 **Done** 按钮也会无效。也就是说, 用户目前会 “卡” (stuck) 在这个由程序所弹出来的 “模态” (modal) 界面里。

6.5.2 实现 UnitsTVC

Units 表格视图界面会把当前可供选用的计量单位 (unit) 列出来。程序加载该界面时, 应该把持久化存储区中的所有 unit 对象都放在表格里, 并按名称排序。用户横向滑动某个 unit 可将其删除。而按下 **Done** 按钮之后, Units 表格视图就应该消失。继承自 **CoreDataTVC** 的 **UnitsTVC** 类将负责实现这些功能, 其代码如程序清单 6-7 所示。读者对清单里的这些代码应该比较熟悉了, 因为早前我们曾经编写过与之类似的代码。

程序清单6-7 UnitsTVC.m文件

```
#import "UnitsTVC.h"
#import "CoreDataHelper.h"
#import "AppDelegate.h"
#import "Unit.h"
@implementation UnitsTVC
#define debug 1
#pragma mark - DATA
- (void)configureFetch {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
}
```

```

CoreDataHelper *cdh =
    [[AppDelegate *)[UIApplication sharedApplication] delegate] cdh];
NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:@"Unit"];
request.sortDescriptors = [NSArray arrayWithObjects:
    [NSSortDescriptor sortDescriptorWithKey:@"name" ascending:YES],nil];
[request setFetchBatchSize:50];
self.frc =
    [[NSFetchedResultsController alloc] initWithFetchRequest:request
                                           managedObjectContext:cdh.context
                                           sectionNameKeyPath:nil
                                           cacheName:nil];

self.frc.delegate = self;
}

#pragma mark - VIEW
- (void)viewDidLoad {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }

    [super viewDidLoad];
    [self configureFetch];
    [self performFetch];
    // Respond to changes in underlying store
    [[NSNotificationCenter defaultCenter] addObserver:self
                                           selector:@selector(performFetch)
                                           name:@"SomethingChanged"
                                           object:nil];
}

- (UITableViewCell*)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }

    static NSString *cellIdentifier = @"Unit Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:cellIdentifier
                                           forIndexPath:indexPath];

    Unit *unit = [self.frc objectAtIndexPath:indexPath];
    cell.textLabel.text = unit.name;
    return cell;
}

- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        Unit *deleteTarget = [self.frc objectAtIndexPath:indexPath];
        [self.frc.managedObjectContext deleteObject:deleteTarget];
    }
}

```

```

        [self.tableView reloadRowsAtIndexPaths:
            [NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    }
}

#pragma mark - INTERACTION
- (IBAction)done:(id)sender {
    if (debug==1) {
        NSLog(@"Running %@", NSStringFromClass(_cmd));
    }

    [self.parentViewController
        dismissViewControllerAnimated:YES completion:nil];
}
@end

```

请按下列步骤修改 Grocery Dude，以实现 UnitsTVC：

1. 选中名为 **Grocery Dude Table View Controllers** 的组。
 2. 点击 **File > New > File...** 菜单项。
 3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
 4. 把 **Subclass of** 设为 **CoreDataTVC**，把 **Class** 名称设为 **UnitsTVC**，然后点击 **Next** 按钮。
 5. 确保 **Targets** 中的 “Grocery Dude” 处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹中创建类。
 6. 用程序清单 6-7 把 UnitsTVC.m 文件里原有的代码全都替换掉，然后保存类文件（可按 “⌘+S” 组合键）。
 7. 选定 **Main.storyboard**。
 8. 在 **Identity Inspector** 界面（可按 “Option + ⌘ + 3” 组合键调出该界面）中，把 **Units** 表格视图控制器的 **Custom Class** 设为 UnitsTVC。
 9. 按住 **Control** 键，从 **Units** 表格视图控制器中的 **Done** 按钮向同一个视图底部的黄圆圈处拖一条直线，然后选择 **Sent Actions > done** 菜单项。
- Units 表格视图已经准备好了。接下来应该实现 Unit 视图控制器所需的代码了，这其中也包括切换到该视图所用的 segue。

6.5.3 实现 UnitVC

用户可以在 Unit 视图里通过文本框来编辑计量单位的名称。程序加载这个视图的时候，应该把用户在表格视图所选计量单位的名称放在文本框里。用户按下 **Done** 按钮之后，这个视图就应该消失。这些功能由继承自 **UIViewController** 的 **UnitVC** 类来实现。相关头文件如程序清单 6-8 所示。

程序清单6-8 UnitVC.h

```
#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
@interface UnitVC : UIViewController <UITextFieldDelegate>
@property (strong, nonatomic) NSString *selectedObjectID;
@property (strong, nonatomic) IBOutlet UITextField *nameTextField;
@end
```

请按下列步骤修改 Grocery Dude，以添加 UnitVC 类：

1. 选中 **Grocery Dude View Controllers** 组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 **UIViewController**，把 **Class** 名称设为 **UnitVC**，然后点击 **Next** 按钮。
5. 确保 **Targets** 中的 “Grocery Dude” 处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹里创建类。

6. 用程序清单 6-8 中的代码把 UnitVC.h 里原有的代码全都替换掉。

UnitVC.m 文件将实现下列三个部分：

- ❑ **VIEW** 部分中有三个方法。viewDidLoad 方法采用书中早前讲过的技术来实现标准的键盘隐藏功能。viewWillAppear 方法负责刷新界面，并且使视图中仅有的那个文本框最早响应，以便显示键盘。refreshInterface 方法会把用户所选 unit 对象的名称放到 nameTextField 文本框里。
- ❑ **TEXTFIELD** 部分用于实现 UITextFieldDelegate 协议中的可选方法，以便在用户结束编辑时，更新 unit.name 的值。
- ❑ **INTERACTION** 部分与 ItemVC 中的同名部分相似，它里面也有负责在用户点击界面背景时隐藏键盘的方法。此外，当用户点击新建的 Done 按钮时，还会有方法负责把用户带回 UnitsTVC 表格视图界面。

程序清单 6-9 列出了相关代码。

程序清单6-9 UnitVC.m文件

```
#import "UnitVC.h"
#import "Unit.h"
#import "AppDelegate.h"
@implementation UnitVC
#define debug 1
#pragma mark - VIEW
- (void)refreshInterface {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
if (self.selectedObjectID) {
```

```

        CoreDataHelper *cdh =
        [[AppDelegate *)[UIApplication sharedApplication] delegate] cdh];
        Unit *unit =
        (Unit*)[cdh.context existingObjectWithID:self.selectedObjectID
                                error:nil];
        self.nameTextField.text = unit.name;
    }
}

- (void)viewDidLoad {
if (debug==1) {
    NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
}

    [super viewDidLoad];
    [self hideKeyboardWhenBackgroundIsTapped];
    self.nameTextField.delegate = self;
}

- (void)viewWillAppear:(BOOL)animated {
if (debug==1) {
    NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
}

    [self refreshInterface];
    [self.nameTextField becomeFirstResponder];
}

#pragma mark - TEXTFIELD
- (void)textFieldDidEndEditing:(UITextField *)textField {
if (debug==1) {
    NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
}

    CoreDataHelper *cdh =
    [[AppDelegate *)[UIApplication sharedApplication] delegate] cdh];
    Unit *unit =
    (Unit*)[cdh.context existingObjectWithID:self.selectedObjectID
                                error:nil];
    if (textField == self.nameTextField) {
        unit.name = self.nameTextField.text;
        [[NSNotificationCenter defaultCenter]
            postNotificationName:@"SomethingChanged"
                                object:nil];
    }
}

#pragma mark - INTERACTION
- (IBAction)done:(id)sender {
if (debug==1) {
    NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
}

    [self hideKeyboard];
    [self.navigationController popViewControllerAnimated:YES];
}

- (void)hideKeyboardWhenBackgroundIsTapped {

```

```

if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

UITapGestureRecognizer *tgr =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                             action:@selector(hideKeyboard)];

[tgr setCancelsTouchesInView:NO];
[self.view addGestureRecognizer:tgr];
}
- (void)hideKeyboard {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

    [self.view endEditing:YES];
}
}
@end

```

请按下列步骤修改 Grocery Dude，以更新 UnitVC 的实现代码：

1. 用程序清单 6-9 中的代码把 UnitVC.m 里原有的代码全部替换掉，然后保存类文件（可按“⌘ + S”组合键）。
2. 选中 **Main.storyboard**。
3. 用 **Identity Inspector** 界面（可按“Option + ⌘ + 3”组合键调出该界面）把 Unit 视图控制器的 **Custom Class** 设为 UnitVC。
4. 按住 **Control** 键，从 Unit 视图控制器中的 **Done** 按钮向同一个视图底部的黄圆圈处拖一条线，然后选择 **Sent Actions > done**。
5. 把 **Assistant Editor** 界面显示出来（可按“Option + ⌘ + Return”组合键）。
6. 如图 6-14 顶部所示，将 **Assistant Editor** 界面上方调整为 **Automatic > UnitVC.h**。

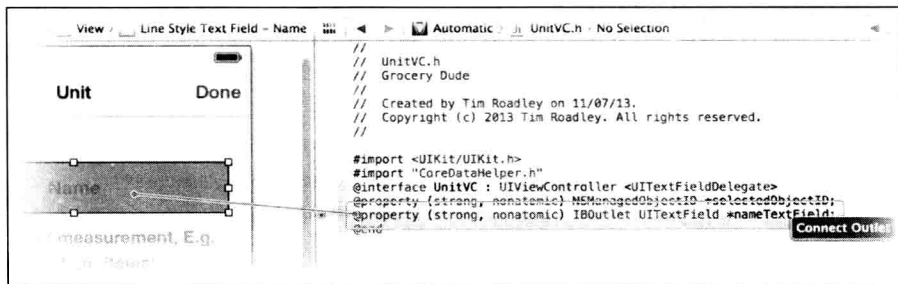


图 6-14 将 Unit 界面中的 Name 文本框与相关的代码连接起来

7. 按住 **Control** 键，从 Unit 界面里的 **Name** 文本框向 UnitVC.h 文件中的 nameTextField 特性拖一条线，如图 6-14 所示。

6.5.4 由 UnitsTVC 切换至 UnitVC

为了切换到 UnitVC，我们需要像程序清单 6-10 这样，实现从 UnitsTVC 所发出

的 segue。这个 segue 会把用户选中的计量单位所具备的 objectID 传给 UnitVC。而 UnitVC 则会根据这个 objectID 来判定用户在前一个界面里到底选了哪个计量单位，然后，它会用该 unit 对象的数据来刷新自己。

程序清单6-10 UnitsTVC.m文件中的prepareForSegue方法

```
#pragma mark - SEGUE
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

UnitVC *unitVC = segue.destinationViewController;
if ([segue.identifier isEqualToString:@"Add Object Segue"])
{
    CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Unit *newUnit =
    [NSEntityDescription insertNewObjectForEntityForName:@"Unit"
                                     inManagedObjectContext:cdh.context];

    NSError *error = nil;
    if (![cdh.context obtainPermanentIDsForObjects:
        [NSArray arrayWithObject:newUnit] error:&error]) {
        NSLog(@"Couldn't obtain a permanent ID for object %@", error);
    }
    unitVC.selectedObjectID = newUnit.objectID;
}
else if ([segue.identifier isEqualToString:@"Edit Object Segue"])
{
    NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
    unitVC.selectedObjectID =
    [[self.frc objectAtIndexPath:indexPath] objectID];
}
else {
    NSLog(@"Unidentified Segue Attempted!");
}
}
```

请按下列步骤修改 Grocery Dude，以实现从 UnitsTVC 到 UnitVC 的切换：

1. 把 **Standard Editor** 界面（可按“⌘ + Return”组合键调出该界面）显示出来。
2. 将 #import "UnitVC.h" 添加到 UnitsTVC.m 文件顶部。
3. 把程序清单 6-10 中的代码添加到 UnitsTVC.m 文件底部，并放在 @end 上方。

运行应用程序，试着用新编写的这几个 Unit 视图来添加一些计量单位。不过，我们目前还不能指定某件货品所使用的计量单位——那是下一章的目标。

6.5.5 添加与编辑货品在家中或商店中的位置

与 home location（货品在家中的位置）及 shop location（货品在商店中的位置）相关的

那两组表格视图及视图的代码和刚才的 UnitsTVC 及 UnitVC 很相似。唯一的区别在于，它们这次分别要支持的是 home location 对象或 shop location 对象，而不是 unit 对象。为了免去大量的代码复制与粘贴操作，笔者把预先制作好的 location 类文件放在网上，供读者下载。

请按下列步骤修改 Grocery Dude，将预先制作好的 location 类文件添加到项目中：

1. 从 <http://www.timroadley.com/LearningCoreData/LocationClasses.zip> 下载包含 location 类的压缩文件，并将其解压缩。

2. 把 LocationsAtHomeTVC 及 LocationsAtShopTVC 这两个类文件拖放到名为 **Grocery Dude Table View Controllers** 的组里。请确认 **Copy items into destination group's folder** 选项及 Targets 中的 “Grocery Dude” 处于勾选状态，然后点击 **Finish** 按钮。

3. 把 LocationAtHomeVC 及 LocationAtShopVC 这两个类文件拖放到名为 **Grocery Dude View Controllers** 的组里。请确认 **Copy items into destination group's folder** 选项及 Targets 中的 “Grocery Dude” 处于勾选状态，然后点击 **Finish** 按钮。

6.5.6 配置与 Home Location 相关的视图

为了使用这些新类，我们需要像早前配置与 Unit 相关的视图时那样，用相似的步骤将这些视图配置好。与 Home Location 有关的这几个视图需要具备相应的“按钮”（button）、“连接”（connection）及“文本框”（text field），下面这些步骤会把它们配置好。

请按下列步骤修改 Grocery Dude：

1. 选中 **Main.storyboard**。

2. 向现有 **Item** 视图中的 **Scroll View** 里拖放一个 **Button**，摆在任意位置均可，然后，在 **Attributes Inspector** 界面（可按 “**Option + ⌘ + 4**” 组合键调出该界面）按下列步骤配置它：

☐ 把 **Type** 设为 **Custom**。

☐ 删掉 “Button” 字样的文本。

☐ 把 **Image** 设为 **add_homelocations**。

3. 在 **Size Inspector** 界面（可按 “**Option + ⌘ + 5**” 组合键调出该界面）中，按照下列步骤配置按钮：

☐ 将 **Width** 设为 **48**。

☐ 将 **Height** 设为 **48**。

4. 按住 **Control** 键，由新的按钮向 Units Navigation 控制器下方的那个 Navigation 控制器拖一条线（早前我们曾经做了三组通用的控制器，而这条线将指向中间那一组控制器），然后选择 **Action Segue > modal**。

5. 在新的 modal segue 所指向的那个 **Table View Controller** 中，把 **Navigation Item Title** 设为 **Home Locations**。

6. 在 **Home Locations** 表格视图控制器中选择 **Prototype Cell**，在 **Attributes Inspector** 界面（可按 “**Option + ⌘ + 4**” 组合键调出该界面）中把 **Reuse Identifier** 设为 **Location-AtHome Cell**。

7. 在 Home Locations 表格视图控制器所指向的那个视图控制器中, 把 **Navigation Item Title** 设为 **Home Location**。

8. 把 **Unit** 视图里的 **Text View** 复制一份, 将其放在 **Home Location** 视图中的相同位置上, 然后把文本内容改为 **Enter the location an item is expected to be found at home. E.g. 'Pantry' or 'Bathroom'.**

9. 在 **Home Location** 视图中选择 **Name Text Field** 及 **Text View**, 然后点击选择 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints** 菜单项。

10. 在 **Identity Inspector** 界面(可按“**Option + ⌘ + 3**”组合键调出该界面)中, 把 **Home Location** 视图控制器的 **Custom Class** 设为 `LocationAtHomeVC`。

11. 用 **Identity Inspector** 界面(可按“**Option + ⌘ + 3**”组合键调出该界面)将 **Home Locations** 表格视图控制器的 **Custom Class** 设为 `LocationAtHomeTVC`。

12. 按住 **Control** 键, 从 **Home Locations** 表格视图控制器中的 **Done** 按钮向同一个视图底部的黄圆圈处拖一条线, 然后选择 **Sent Actions > done**。

13. 按住 **Control** 键, 从 **Home Location** 视图控制器中的 **Done** 按钮向同一个视图底部的黄圆圈处拖一条线, 然后选择 **Sent Actions > done**。

14. 把 **Assistant Editor** 界面显示出来(可按“**Option + ⌘ + Return**”组合键调出该界面)。

15. 参照早前图 6-14 所演示的办法, 把 Assistant Editor 界面顶部调整为 **Automatic > LocationAtHomeVC.h**。

16. 参照早前图 6-14 所演示的办法, 按住 **Control** 键, 从 **Home Location Name** 文本框向 `LocationAtHomeVC.h` 文件里现有的 `nameTextField` 特性拖一条线。

6.5.7 配置与 Shop Location 相关的视图

要想使用新编的 `shop location` 类, 我们还需执行与上一小节相似的操作才行。与 `Shop Location` 有关的这几个视图需要具备相应的“按钮”(button)、“连接”(connection)及“文本框”(text field), 下面这些步骤会把它们配置好。

请按下列步骤修改 `Grocery Dude`:

1. 选中 **Main.storyboard**, 将 **Standard Editor** 界面显示出来(可按“**⌘ + Return**”组合键调出该界面)。

2. 向现有 **Item** 视图中的 **Scroll View** 里拖放一个 **Button**, 摆在任意位置均可, 然后, 在 **Attributes Inspector** 界面(可按“**Option + ⌘ + 4**”组合键调出该界面)按下列步骤配置它:

- ☐ 把 **Type** 设为 **Custom**。
- ☐ 删掉“Button”字样的文本。
- ☐ 把 **Image** 设为 `add_shoplocations`。

3. 在 **Size Inspector** 界面(可按“**Option + ⌘ + 5**”组合键调出该界面)中, 按照下列步骤配置按钮:

❑ 将 **Width** 设为 **48**。

❑ 将 **Height** 设为 **48**。

4. 按住 **Control** 键，由新的按钮向 **Home Locations Navigation** 控制器下方的那个 **Navigation** 控制器拖一条线（早前我们曾经做了三组通用的控制器，而这条线将指向最下边那一组控制器），然后选择 **Action Segue > modal**。

5. 在新的 modal segue 所指向的那个 Table View Controller 中，把 **Navigation Item Title** 设为 **Shop Locations**。

6. 在 **Shop Locations** 表格视图控制器中选择 **Prototype Cell**，在 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面）中把 **Reuse Identifier** 设为 **LocationAtShop Cell**。

7. 在 **Shop Locations** 表格视图控制器所指向的那个视图控制器中，把 **Navigation Item Title** 设为 **Shop Location**。

8. 把 **Home Location** 视图里的 **Text View** 复制一份，将其放在 **Shop Location** 视图中的相同位置上，然后把文本内容改为 **Enter the location an item is expected to be found at the shop. E.g. ‘Produce Section’ or ‘Deli’**。

9. 在 **Shop Location** 视图中选择 **Name Text Field** 及 **Text View**，然后点击选择 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints** 菜单项。

10. 在 **Identity Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）中，把 **Shop Location** 视图控制器的 **Custom Class** 设为 **LocationAtShopVC**。

11. 用 **Identity Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）将 **Shop Locations** 表格视图控制器的 **Custom Class** 设为 **LocationAtShopTVC**。

12. 按住 **Control** 键，从 **Shop Locations** 表格视图控制器中的 **Done** 按钮向同一个视图底部的黄圆圈处拖一条线，然后选择 **Sent Actions > done**。

13. 按住 **Control** 键，从 **Shop Location** 视图控制器中的 **Done** 按钮向同一个视图底部的黄圆圈处拖一条线，然后选择 **Sent Actions > done**。

14. 把 **Assistant Editor** 界面显示出来（可按“**Option + ⌘ + Return**”组合键调出该界面）。

15. 参照早前图 6-14 所演示的办法，把 Assistant Editor 界面顶部调整为 **Automatic > LocationAtShopVC.h**。

16. 参照早前图 6-14 所演示的办法，按住 **Control** 键，从 **Shop Location Name** 文本框向 **LocationAtShopVC.h** 文件里现有的 **nameTextField** 特性拖一条线。

17. 把 **Standard Editor** 界面显示出来（可按“**⌘ + Return**”组合键调出该界面）。

unit（计量单位）、home location（货品在家中的位置）及 shop location（货品在商店中的位置）的编辑功能现在已经做好了。运行应用程序，看看自己的成果吧！

本章做好的“视图层级”（view hierarchy）应该如图 6-15 所示。请调整 Item 视图里面靠右的那三个按钮，令其顺序与本图相符。

2. 在 **Prepare** 选项卡中选择某些货品，令其变为橙色。然后切换到 **Shop** 选项卡，看看刚才所选的货品现在显示在表格视图的哪个部分中。你会发现：用户只需点击某项货品，程序就能按照它在商店中摆放的位置自动将其划分到对应的部分里，并显示在 **Shop** 选项卡中——而这正是 **Grocery Dude** 的关键功能。

3. 尝试一下 **Prepare** 选项卡与 **Shop** 选项卡的 **Clear** 功能。



选取器视图

困难之中蕴藏着机会。

——阿尔伯特·爱因斯坦

我们从第6章开始构建Item视图，并于那一章中演示了在应用程序里传递托管对象这一概念。上一章主要讲解如何通过文本框来编辑托管对象。本章将要讲述怎样创建特殊的文本框，此文本框会把由Core Data所驱动的UIPickerView当作其输入视图来用。这种选取器式文本框的目标是向用户提供一套简单而快捷的选取方式，令其可在一组预定义的数值中进行选择。

7.1 概述

用户很容易就能通过选取器视图把托管对象联系起来。item对象的关系特性（例如unit.name、locationAtHome.storedIn及locationAtShop.aisle等）非常适合通过选取器视图来设置。笔者这么说的原因在于，这些特性的取值可以为多个item所共用。比方说，可以像图7-1这样，令用户通过选取器视图来指定某件货品在商店中的摆放位置。为了将选取器视图显示出来，我们需要创建特殊的UITextField子类。这个子类会把选取器视图以输入视图的面貌展现出来，而键盘一般也是以输入视图这种形式来展现的。



图 7-1 选取器视图



提示

为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter06.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 Product > Clean 菜单项，这样可以清除掉同名项目所残留的缓存。

7.2 创建 CoreDataPickerTF

为了以输入视图方式来提供选取器视图，我们需要创建名为 CoreDataPickerTF 的类，它继承自 UITextField。这个新的类将会采用 UIKeyInput、UIPickerViewDelegate 及 UIPickerViewDataSource 协议。正如 CoreDataTVC 是 PrepareTVC 与 ShopTVC 的基础一样，CoreDataPickerTF 也是其他自定义子类的基础。而那些自定义的子类，则分别用于实现新的文本框，以供用户设定货品的计量单位 (unit)、在家中的位置 (home location) 以及在商店中的位置 (shop location)。

当用户在 Picker 中选择了某一项时，程序需要把用户所选的值反映到相关文本框中。例如，用户通过 Picker 选定了某货品在商店中摆放的位置，那么此时该位置的名称就应该显示在相关的文本框里才对。为实现这一功能，CoreDataPickerTF 类定义了 CoreDataPickerTFDelegate 协议。凡是使用了 CoreDataPickerTF 子类的文本框都需要采用此协议，以便从 Picker 中接收新的字符串值。程序清单 7-1 所列的 selectObjectID:changedForPickerTF 方法就是这份新协议中的一部分，此外，该程序清单还列出了一些新的特性。

程序清单7-1 CoreDataPickerTF.h文件

```

#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
@class CoreDataPickerTF;
@protocol CoreDataPickerTFDelegate <NSObject>
- (void)selectedObjectID:(NSManagedObjectID*)objectID
    changedForPickerTF:(CoreDataPickerTF*)pickerTF;
@optional
- (void)selectedObjectClearedForPickerTF:(CoreDataPickerTF*)pickerTF;
@end

@interface CoreDataPickerTF : UITextField
<UIKeyInput, UIPickerViewDelegate, UIPickerViewDataSource>
@property (nonatomic, weak) id <CoreDataPickerTFDelegate> pickerDelegate;
@property (nonatomic, strong) UIPickerView *picker;
@property (nonatomic, strong) NSArray *pickerData;
@property (nonatomic, strong) UIToolbar *toolbar;
@property (nonatomic) BOOL showToolbar;
@property (nonatomic, strong) NSManagedObjectID *selectedObjectID;
@end

```

如程序清单 7-1 所示，CoreDataPickerTF 里面共有六项特性：

- ❑ **pickerDelegate** 是个引用，它指向 CoreDataPickerTFDelegate 协议的委托。当用户选中 Picker 中的某一行时，系统将向其发送消息，把用户所选的内容告诉委托。
- ❑ **picker** 是指向选取器视图的引用。
- ❑ **pickerData** 是指向数组的引用，程序将通过 Core Data 来获取数据，并用获取到的数据填充此数组。
- ❑ **toolbar** 是指向工具栏的引用，该工具栏位于选取器视图顶部。工具栏中会有 Clear 按钮及 Done 按钮。Clear 按钮用于清除当前所选的内容，Done 按钮则可关闭选取器视图。
- ❑ **showToolbar** 是个标志，用于表示工具栏是否应该隐藏。该特性纯粹是为了便于隐藏工具栏而设的，开发者如果想在自己的应用程序中复用 CoreDataPickerTF，那么可以通过它来隐藏工具栏。Grocery Dude 程序总是会把工具栏显示出来。
- ❑ **selectedObjectID** 是个指向 objectID 的引用，而 objectID 则是用户所选托管对象的 ID，该托管对象与某件货品相关联。当用户在 Picker 中选取了不同的条目时，该特性也会随之变化，以反映用户新选中的条目。

请按下列步骤修改 Grocery Dude，以配置 CoreDataPickerTF：

1. 选中 **Generic Core Data Classes** 组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。

4. 将 **Subclass of** 设为 `UITextField`，将 **Class** 名称设为 `CoreDataPickerTF`，然后点击 **Next** 按钮。

5. 确保 **Targets** 中的 `Grocery Dude` 处于勾选状态，然后点击 **Create** 按钮，在 `Grocery Dude` 项目的文件夹中创建类。

6. 用程序清单 7-1 将 `CoreDataPickerTF.h` 文件里原有的代码全都替换掉。

`CoreDataPickerTF` 的实现文件有四个部分：

- ❑ **DELEGATE+DATASOURCE: UIPickerView** 部分用于实现协议中与填充 `UIPickerView` 的内容有关的那些方法。此外，当用户在选取器中选定某个对象时，它还会处理 `pickerDelegate` 的返回值。
- ❑ **INTERACTION** 部分里实现的方法负责处理输入附件视图上面的新按钮，那两个按钮分别用于清除当前所选内容以及隐藏选取器控件。
- ❑ **DATA** 部分里实现的方法负责用数据填充 `NSArray`，以驱动选取器。它也负责处理选取器在默认状态下所选定的行。
- ❑ **VIEW** 部分里实现的方法用于在输入视图中创建选取器视图，并在输入附件视图中创建工具栏。设备旋转时负责重绘的方法也放在这个部分。

7.3 DELEGATE+DATASOURCE:UIPickerView

这个部分有五个方法，有些用于实现 `UIPickerViewDataSource` 协议所规定的方法，另外一些用于实现 `UIPickerViewDelegate` 协议中的可选方法：

- ❑ **numberOfComponentsInPickerView** 方法用于指定选取器视图的列数。在选取器视图中，列称作组件（component）。`Grocery Dude` 程序只需一个组件，所以我们令该方法返回硬编码 1。在开发自己的项目时，如果想实现具有多个组件的 `Core Data` 选取器视图，可以在本方法中指定组件的数量。若有多个组件，则需为每个组件准备一个数组。
- ❑ **numberOfRowsInComponent** 方法用于指定选取器视图的总行数。该值由 `pickerData` 数组中的对象个数来确定。该方法最合适的返回值就是 `[pickerData-count]`，它表示 `pickerData` 数组中的对象个数。
- ❑ **widthForComponent** 方法用于指定每个组件的宽度，我们令其返回硬编码 280。
- ❑ **titleForRow** 方法用于指定每个组件的每一行所显示的内容。它与表格视图填充其每行内容所用的 `cellForRowAtIndexPath` 方法相似。
- ❑ **didSelectRow** 方法用于处理用户对某一行的选定操作。`CoreDataPickerTF` 默认会向委托发送消息，告知字符串值已经改变。`CoreDataPickerTF` 的子类应该覆写此方法。

程序清单 7-2 列出了相关代码。

程序清单7-2 CoreDataPickerTF.m文件中的DELEGATE+DATASOURCE: UIPickerView

```

#import "CoreDataPickerTF.h"
@implementation CoreDataPickerTF
#define debug 1

#pragma mark - DELEGATE & DATASOURCE: UIPickerView
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return 1;
}
- (NSInteger)pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return [self.pickerData count];
}
- (CGFloat)pickerView:(UIPickerView *)pickerView
rowHeightForComponent:(NSInteger)component {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return 44.0f;
}
- (CGFloat)pickerView:(UIPickerView *)pickerView
widthForComponent:(NSInteger)component {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return 280.0f;
}
- (NSString *)pickerView:(UIPickerView *)pickerView
titleForRow:(NSInteger)row
forComponent:(NSInteger)component {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    return [self.pickerData objectAtIndex:index:row];
}
- (void)pickerView:(UIPickerView *)pickerView
didSelectRow:(NSInteger)row
inComponent:(NSInteger)component {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    NSManagedObject *object = [self.pickerData objectAtIndex:index:row];
    [self.pickerDelegate selectedObjectID:object.objectID

```

```

        changedForPickerTF:self];
    }
@end

```

请按下列步骤修改 Grocery Dude，以实现 DELEGATE+DATASOURCE: UIPickerView 部分：

用程序清单 7-2 中的代码把 CoreDataPickerTF.m 文件里原有的代码全都替换掉。

7.3.1 INTERACTION

这个部分里面有两个简单的方法：

- ❑ **done** 方法会在用户点击选取器工具栏中的 Done 按钮时执行。该方法将把选取器隐藏起来。
- ❑ **clear** 方法会在用户点击选取器工具栏中的 Clear 按钮时执行。该方法会向选取器的委托发送消息，告知其应将当前所选内容清除。

程序清单 7-3 列出了相关的代码。

程序清单7-3 CoreDataPickerTF.m文件中的INTERACTION部分

```

#pragma mark - INTERACTION
- (void)done {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    [self resignFirstResponder];
}
- (void)clear {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
    [self.pickerDelegate selectedObjectClearedForPickerTF:self];
    [self resignFirstResponder];
}

```

请按下列步骤修改 Grocery Dude，以实现 INTERACTION 部分：

把程序清单 7-3 中的代码添加到 CoreDataPickerTF.m 文件底部，并放在 @end 上方。

7.3.2 DATA

这个部分里面有两个方法，它们一开始都没有实际内容：

- ❑ **fetch** 方法应该由 CoreDataPickerTF 的子类覆写，以使用相关对象来填充 self.pickerData，使选取器可以显示出内容。
- ❑ **selectDefaultRow** 方法应该由 CoreDataPickerTF 的子类覆写。覆写该方法时，应该指明 Picker 控件默认所选定的行。如果货品已经同某个对象建立了关联，

那么选取器会默认选定与该对象相对应的那一行。比方说，名为 Milk 的货品已经与名为 Fridge 的 home location 对象建立了关联，那么选取器控件就默认把 Fridge 这一行选中。

程序清单 7-4 列出了相关代码。这两个方法都必须由子类覆写，假如有人直接调用它们，那就令其抛出异常。

程序清单7-4 CoreDataPickerTF.m文件中的DATA部分

```
#pragma mark - DATA
- (void)fetch {
    [NSEException raise:NSInternalInconsistencyException format:
    @"You must override the '%@' method to provide data to the picker",
    NSStringFromSelector(_cmd)];
}
- (void)selectDefaultRow {
    [NSEException raise:NSInternalInconsistencyException format:
    @"You must override the '%@' method to set the default picker row",
    NSStringFromSelector(_cmd)];
}
```

请按下列步骤修改 Grocery Dude，以实现 DATA 部分：

把程序清单 7-4 中的代码添加到 CoreDataPickerTF.m 文件底部，并放在 @end 上方。

7.3.3 VIEW

这个部分里有五个方法：

- ❑ **createInputView** 方法返回包含 Picker 的 UIView。
- ❑ **createInputAccessoryView** 方法返回包含工具栏的 UIView，工具栏里会有 Clear 及 Done 按钮。
- ❑ **initWithFrame** 及 **initWithCoder** 方法分别用 createInputView 及 createInputAccessoryView 来设定继承自 UITextField 的 inputView 及 inputAccessoryView 特性。
- ❑ **deviceDidRotate** 方法用来确保选取器视图会在设备旋转时重绘。

程序清单 7-5 列出了相关代码。

程序清单7-5 CoreDataPickerTF.m文件中的VIEW部分

```
#pragma mark - VIEW
- (UIView *)createInputView {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }
    self.picker = [[UIPickerView alloc] initWithFrame:CGRectZero];
    self.picker.showsSelectionIndicator = YES;
    self.picker.autoresizingMask = UIViewAutoresizingFlexibleHeight;
```

```

        self.picker.dataSource = self;
        self.picker.delegate = self;
        [self fetch];
        return self.picker;
    }

    - (UIView *)createInputAccessoryView {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

    self.showToolbar = YES;
    if (!self.toolbar && self.showToolbar) {
        self.toolbar = [[UIToolbar alloc] init];
        self.toolbar.barStyle = UIBarStyleBlackTranslucent;
        self.toolbar.autoresizingMask = UIViewAutoresizingFlexibleHeight;
        [self.toolbar sizeToFit];
        CGRect frame = self.toolbar.frame;
        frame.size.height = 44.0f;
        self.toolbar.frame = frame;

        UIBarButtonItem *clearBtn = [[UIBarButtonItem alloc]
                                       initWithTitle:@"Clear"
                                       style:UIBarButtonItemStyleBordered
                                       target:self
                                       action:@selector(clear)];
        UIBarButtonItem *spacer = [[UIBarButtonItem alloc]
                                    initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
                                    target:nil
                                    action:nil];
        UIBarButtonItem *doneBtn = [[UIBarButtonItem alloc]
                                    initWithBarButtonSystemItem:UIBarButtonSystemItemDone
                                    target:self
                                    action:@selector(done)];

        NSArray *array =
            [NSArray arrayWithObjects:clearBtn, spacer, doneBtn, nil];
        [self.toolbar setItems:array];
    }
    return self.toolbar;
}

    - (id)initWithFrame:(CGRect)aRect {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

    if (self = [super initWithFrame:aRect]) {
        self.inputView = [self createInputView];
        self.inputAccessoryView = [self createInputAccessoryView];
    }
    return self;
}

    - (id)initWithCoder:(NSCoder*)aDecoder {

```

```

if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

if (self = [super initWithCoder:aDecoder]) {
    self.inputView = [self createInputView];
    self.inputAccessoryView = [self createInputAccessoryView];
}
return self;
}

- (void)deviceDidRotate:(NSNotification*)notification {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

[self.picker setNeedsLayout];
}
}

```

请按下列步骤修改 Grocery Dude，以实现 VIEW 部分：

1. 把程序清单 7-5 中的代码添加到 CoreDataPickerTF.m 文件底部，并放在 @end 上方。

CoreDataPickerTF 现在已经可以供子类来继承了，这些子类应该从 Core Data 中获取数据，并用这些数据填充选取器视图，然后将其提供给系统。我们需要在 Grocery Dude 中生成六份新文件，以便创建三个子类。生成文件之前，需要在 Xcode 中为项目添加新的组织结构。

请按下列步骤修改 Grocery Dude，以创建新的组：

1. 在现有的 **Grocery Dude** 组上右击鼠标，然后选择 **New Group**。
2. 把新的组命名为 **Grocery Dude Picker Text Fields**。

7.4 创建 UnitPickerTF

用户需要使用 Picker 控件从现有的各种计量单位中选择一种指定给某件货品。为了展示 Picker，我们需要有文本框以及自定义的 CoreDataPickerTF 子类。CoreDataPickerTF 子类实现了下面三个方法：

- ❑ **fetch** 方法负责构建 Core Data 获取需求，并用 Unit 对象来填充 self.pickerData 数组。
- ❑ **selectDefaultRow** 方法用于指明选取器默认所选的行。假如与货品相关联的对象在选取器里面，那么选取器就会默认选定表示该对象的那一行。比方说，名为 Bananas 的货品已经与名为 Kg 的 unit 对象关联起来了，那么选取器控件默认会选中 Kg。selectedObjectID 特性用于判定选取器默认应该选中哪个对象：selectDefaultRow 方法会遍历 self.pickerData 数组，逐个寻找能够与 selectedObject.name 相匹配的 unit 对象。请注意：目前我们允许数据库里有

多个同名的 unit 对象。本书在临近收尾的第 15 章中，会介绍重复数据去除功能，到那时，就不可能出现这种情况了。

- ❑ **titleForRow** 方法将配置 Picker 中的每一行，使其显示出该行所表示的计量单位名称。

程序清单 7-6 列出了相关代码。

程序清单 7-6 UnitPickerTF.m 文件

```
#import "UnitPickerTF.h"
#import "CoreDataHelper.h"
#import "AppDelegate.h"
#import "Unit.h"
@implementation UnitPickerTF
#define debug 1
- (void)fetch {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}

CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:@"Unit"];
NSSortDescriptor *sort =
    [NSSortDescriptor sortDescriptorWithKey:@"name" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sort]];
[request setFetchBatchSize:50];
NSError *error;
self.pickerData = [cdh.context executeFetchRequest:request
                                error:&error];

if (error) {
    NSLog(@"Error populating picker: %@", @"",
        , error, error.localizedDescription);
    [self selectDefaultRow];
}
- (void)selectDefaultRow {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedObjectID && [self.pickerData count] > 0) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Unit *selectedObject =
        (Unit*)[cdh.context existingObjectWithID:self.selectedObjectID
                                error:nil];
    [self.pickerData enumerateObjectsUsingBlock:^(
        Unit *unit, NSUInteger idx, BOOL *stop) {
        if ([unit.name compare:selectedObject.name] == NSOrderedSame) {
            [self.picker selectRow:idx inComponent:0 animated:NO];
            [self.pickerDelegate selectedObjectID:self.selectedObjectID
```

```

        changedForPickerTF:self];

        *stop = YES;
    }
    }];
}

- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    Unit *unit = [self.pickerData objectAtIndex:row];
    return unit.name;
}

@end

```

请按下列步骤修改 Grocery Dude，以便创建 UnitPickerTF 类：

1. 选定名为 **Grocery Dude Picker Text Fields** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 CoreDataPickerTF，把 **Class** 名称设为 UnitPickerTF，然后点击 **Next** 按钮。
5. 确保 Targets 中的 Grocery Dude 处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹下创建类。
6. 用程序清单 7-6 中的代码把 UnitPickerTF.m 文件里原有的代码全都替换掉。

7.4.1 创建 Unit 选取器

准备好 UnitPickerTF 类之后，我们需要在 ItemVC 上面创建新的文本框。

请按下列步骤修改 Grocery Dude，以创建选择计量单位所用的选取器式文本框：

1. 选定 **Main.storyboard**。
2. 向 **Item** 视图的 **Scroll View** 中拖放一个 **Text Field**，摆在任意位置均可，然后在 **Attributes Inspector** 界面（可按 “**Option + ⌘ + 4**” 组合键调出该界面）按如下步骤配置它：
 - ☐ 把 **Font** 设为 **System Bold 17.0**。
 - ☐ 把 **Text Alignment** 设为 **Center**。
 - ☐ 把 **Placeholder Text** 设为 **Unit**。
 - ☐ 把 **Border Style** 设为 **Line**（Attributes Inspector 界面用矩形来表示该选项）。
 - ☐ 把 **Background** 设为 **Other > Crayons > Mercury**（Mercury 指的是第二个颜色最淡的灰色蜡笔）。
3. 在 **Size Inspector** 界面（可按 “**Option + ⌘ + 5**” 组合键调出该界面）把 **Text Field** 的 **Height** 设为 **48**。

4. 按图 7-2 所示的边线位置来排布 **Unit** 文本框，然后在 **Identity Inspector** 界面（可按“**Option**+**⌘**+3”组合键调出该界面）把 **Custom Class** 设为 **UnitPickerTF**。

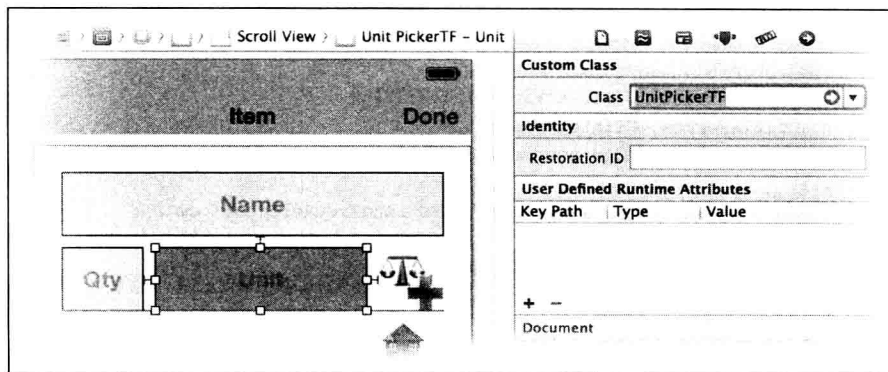


图 7-2 用户选择计量单位时所用的选取器式文本框

7.4.2 连接 Unit 选取器

为了能在代码里引用新建的选取器式文本框，我们需要把它同 **Outlet** 相连。这项操作可以用 **Assistant Editor** 来完成。

请按下列步骤修改 **Grocery Dude**，以便将 **Unit** 选取器文本框连接到 **ItemVC.h**：

1. 将 `#import "UnitPickerTF.h"` 添加到 **ItemVC.h** 顶部。
2. 选中 **Main.storyboard**。
3. 确保 **Item** 视图控制器处于受选状态，然后把 **Assistant Editor** 界面显示出来（可按“**Option**+**⌘**+**Return**”组合键调出该界面）。

4. 如果 **Assistant Editor** 顶部显示的不是 **Automatic > ItemVC.h**，那就将其调整为 **Automatic > ItemVC.h**。

5. 按住 **Control** 键，从 **Unit Text Field** 向 **ItemVC.h** 文件底部的 `@end` 上方拖一条线。将新特性的 **Name** 设为 `unitPickerTextField`，如图 7-3 所示。请确认 **Type** 是 **UnitPickerTF**，**Storage** 是 **Strong**。

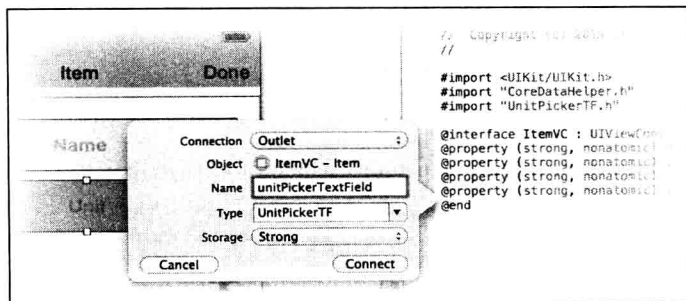


图 7-3 连接 Unit Picker 文本框

6. 把 Standard Editor 显示出来 (可按“⌘+Return”组合键调出该界面)。

7.4.3 为 Unit 选取器配置 ItemVC

刚才我们在 Assistant Editor 界面中通过新的 unitPickerTextField 特性创建了指向 Unit 选取器文本框的引用。要想使这个 Unit 选取器文本框能够把用户所选定的值回传给其中的普通文本框,我们就必须令 ItemVC 采用 CoreDataPickerTFDelegate 协议。

请按下列步骤修改 Grocery Dude:

1. 修改 ItemVC.h 文件,令其采用 CoreDataPickerTFDelegate 协议。为了便于大家参考,程序清单 7-7 列出了修改后的 ItemVC.h 文件。

程序清单 7-7 ItemVC.h

```
#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
#import "UnitPickerTF.h"
@interface ItemVC : UIViewController
<UITextFieldDelegate, CoreDataPickerTFDelegate>
@property (strong, nonatomic) NSString *selectedItemID;
@property (strong, nonatomic) IBOutlet UIScrollView *scrollView;
@property (strong, nonatomic) IBOutlet UITextField *nameTextField;
@property (strong, nonatomic) IBOutlet UITextField *quantityTextField;
@property (strong, nonatomic) IBOutlet UnitPickerTF *unitPickerTextField;
@end
```

由于 ItemVC 类同时采用了 CoreDataPickerTFDelegate 及 UITextFieldDelegate 协议,所以它需要同时充当选取器式文本框的 delegate 与 pickerDelegate。充当委托可保证用户在 Unit 选取器中所选定的值能够反映在 Unit 文本框里,而充当 pickerDelegate 则可保证当 Unit 选取器把 Unit 文本框挡住的时候,Unit 文本框能够显示在用户可以看见的范围内。为实现此功能,本章稍后会通过 UITextFieldDelegate 方法把选定的文本框设为 activeField。程序清单 7-8 列出了设置这两个委托所用的代码。

程序清单 7-8 ItemVC.m 的 viewDidLoad 方法

```
self.unitPickerTextField.delegate = self;
self.unitPickerTextField.pickerDelegate = self;
```

请按下列步骤修改 Grocery Dude,以配置 Unit 选取器文本框的两个委托:

1. 修改 ItemVC.m 文件的 viewDidLoad 方法,把程序清单 7-8 中的代码添加到该方法底部。

做好上述修改之后,你可能会注意到,Xcode 将显示警告信息,说 selectedItemID:changedForPickerTF 方法还没有实现。CoreDataPickerTFDelegate 协议要求开发

者必须编写该方法。每实现一种选取器式文本框，就要修改一次 `changedForPickerTF` 方法。于是，我们在 `ItemVC.m` 里面新建名为 `PICKERS` 的部分，以便在其中实现这些代码。

`PICKERS` 部分里面会有两个方法：

❑ **`selectedObjectID:changedForPickerTF`** 方法。选取器每次向其 `delegate` 发送该消息时，系统就会调用这个方法。此方法会把用户所选的“计量单位”（`unit`）设置到“货品”（`item`）上面，并且会更新 `unitPickerTextField.text` 的值，使之与计量单位的名称相符。稍后的 `homeLocationPickerTextField` 及 `shopLocationPickerTextField` 也将使用这个办法。

❑ **`selectedObjectClearedForPickerTF`** 方法。选取器每次向其 `delegate` 发送该消息时，系统就会调用这个方法。此方法会把货品的计量单位清除掉，同时也会清除 `unitPickerTextField.text` 中的文本。稍后的 `homeLocationPickerTextField` 及 `shopLocationPickerTextField` 也将使用这个办法。

程序清单 7-9 列出了相关的代码。

程序清单 7-9 `ItemVC.m` 文件中的 `PICKERS` 部分

```
#pragma mark - PICKERS
- (void)selectedObjectID:(NSManagedObjectID *)objectID
    changedForPickerTF:(CoreDataPickerTF *)pickerTF {
    if (debug==1) {
        NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
    }

    if (self.selectedItemID) {
        CoreDataHelper *cdh =
            [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];

        Item *item =
            (Item*)[cdh.context existingObjectWithID:self.selectedItemID
                                error:nil];

        NSError *error;
        if (pickerTF == self.unitPickerTextField) {
            Unit *unit = (Unit*)[cdh.context existingObjectWithID:objectID
                                error:&error];

            item.unit = unit;
            self.unitPickerTextField.text = item.unit.name;
        }
        [self refreshInterface];
        if (error) {
            NSLog(@"Error selecting object on picker: %@", @"%",
                error, error.localizedDescription);
        }
    }
}

- (void)selectedObjectClearedForPickerTF:(CoreDataPickerTF *)pickerTF {
```

```

if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID
                error:nil];

    if (pickerTF == self.unitPickerTextField) {
        item.unit = nil;
        self.unitPickerTextField.text = @"";
    }
    [self refreshInterface];
}
}

```

请按下列步骤修改 Grocery Dude，以实现 PICKERS 部分：

1. 把 #import "Unit.h" 添加到 ItemVC.m 顶部。
2. 把程序清单 7-9 中的代码添加到 ItemVC.m 底部，并放在 @end 上方。

为了确保 Unit Picker 每次都能够显示出最新的数据，我们需要将 UnitPickerTF.m 中的 fetch 方法公开。这样的话，ItemVC.m 的 textFieldDidBeginEditing 方法就可以调用它了。此外，调用完 fetch 方法之后，还需要重新加载 _unitPickerTextField 中的 picker。程序清单 7-10 列出了相关代码。

程序清单7-10 ItemVC.m文件中的textFieldDidBeginEditing方法

```

if (textField == _unitPickerTextField && _unitPickerTextField.picker) {
    [_unitPickerTextField fetch];
    [_unitPickerTextField.picker reloadData];
}

```

请按下列步骤修改 Grocery Dude，以确保 Unit Picker 总能够显示出最新的数据：

1. 把下列代码添加到 UnitPickerTF.h 的 @end 之前：

```
- (void)fetch;
```

2. 修改 ItemVC.m 的 textFieldDidBeginEditing 方法，把程序清单 7-10 中的代码添加到该方法底部。

Unit 选取器文本框所需实现的最后一项功能是确保当应用程序把视图显示出来的时候，选取器文本框的文本能够与当前计量单位的名称相符。由于目前的 viewWillAppear 方法已经调用 refreshInterface，所以在此处设置 Unit 选取器文本框的文本值是比较合适的。此外，我们还要通过 objectID 把选取器当前所选定的 unit 设置成已经与货品关联起来的那种计量单位。程序清单 7-11 列出了相关代码。

程序清单7-11 ItemVC.m文件中的refreshInterface方法

```

- (void)refreshInterface {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID
            error:nil];

    self.nameTextField.text = item.name;
    self.quantityTextField.text = item.quantity.stringValue;
    self.unitPickerTextField.text = item.unit.name;
    self.unitPickerTextField.selectedObjectID = item.unit.objectID;
}
}

```

请按下列步骤修改 Grocery Dude，以确保 unitPickerTextField 能够显示出适当的计量单位名称：

1. 用程序清单 7-11 中的方法把 ItemVC.m 文件里原有的 refreshInterface 方法替换掉。
2. 运行应用程序。如果持久化存储区中没有货品或计量单位，那就通过前一章创建的视图来添加一些进去。当持久化存储区里有了货品和计量单位之后，试着用 Picker 来设定某件货品的计量单位。正常的运行效果应该如图 7-4 所示。

Unit 选取器现在已经完全能够运作了，所以接下来应该用相似的流程来实现 Home Location 选取器和 Shop Location 选取器。

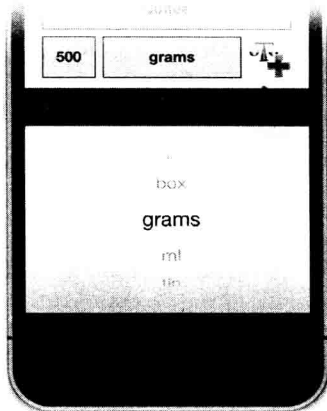


图 7-4 已经实现好的 Unit Picker 文本框

7.5 创建 LocationAtHomePickerTF

我们需要创建一个选取器，使得用户可以通过它来指定货品在家中的存放地点。创建这个 Home Location 选取器所采用的办法与早前创建 Unit 选取器时相似，这次也是在 CoreDataPickerTF 的子类中实现 fetch、selectDefaultRow 与 titleForRow 方法。

程序清单 7-12 列出了相关代码。

程序清单7-12 LocationAtHomePickerTF.m文件

```

#import "LocationAtHomePickerTF.h"
#import "CoreDataHelper.h"
#import "AppDelegate.h"

```

```

#import "LocationAtHome.h"
@implementation LocationAtHomePickerTF
#define debug 1
- (void)fetch {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:@"LocationAtHome"];
    NSSortDescriptor *sort =
    [NSSortDescriptor sortDescriptorWithKey:@"storedIn" ascending:YES];
    [request setSortDescriptors:[NSArray arrayWithObject:sort]];
    [request setFetchBatchSize:50];
    NSError *error;
    self.pickerData = [cdh.context executeFetchRequest:request
                                error:&error];

    if (error) {
        NSLog(@"Error populating picker: %@", error,
            error.localizedDescription);
    }
    [self selectDefaultRow];
}

- (void)selectDefaultRow {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    if (self.selectedObjectID && [self.pickerData count] > 0) {
        CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
        LocationAtHome *selectedObject = (LocationAtHome*)[cdh.context
            existingObjectWithID:self.selectedObjectID
            error:nil];

        [self.pickerData enumerateObjectsUsingBlock:^(
            LocationAtHome *locationAtHome, NSUInteger idx, BOOL *stop) {
            if ([locationAtHome.storedIn compare:selectedObject.storedIn
                == NSOrderedSame) {
                [self.picker selectRow:idx inComponent:0 animated:NO];
                [self.pickerDelegate selectedObjectID:self.selectedObjectID
                    changedForPickerTF:self];

                *stop = YES;
            }
        }];
    }

    - (NSString *)pickerView:(UIPickerView *)pickerView
        titleForRow:(NSInteger)row
        forComponent:(NSInteger)component {
if (debug==1) {

```

```

        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }

    LocationAtHome *locationAtHome = [self.pickerData objectAtIndex:row];
    return locationAtHome.storedIn;
}

@end

```

请按下列步骤修改 Grocery Dude，以创建 LocationAtHomePickerTF 类：

1. 选定名为 **Grocery Dude Picker Text Fields** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 CoreDataPickerTF，把 **Class** 名称设为 LocationAtHomePickerTF，然后点击 **Next** 按钮。
5. 确保 Targets 中的 “Grocery Dude” 处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹中创建类。
6. 用程序清单 7-12 中的代码把 LocationAtHomePickerTF.m 文件里原有的代码全都替换掉。

由于代码结构非常接近，所以现在也为 Shop Location Picker 创建与此相似的代码。

7.6 创建 LocationAtShopPickerTF

我们需要创建一个选取器，以供用户设定货品在商店中摆放的地点。这次和上一节类似，也是在 CoreDataPickerTF 的子类中实现 `fetch`、`selectDefaultRow` 与 `titleForRow` 方法。

程序清单 7-13 列出了相关的代码。

程序清单7-13 LocationAtShopPickerTF.m文件

```

#import "LocationAtShopPickerTF.h"
#import "CoreDataHelper.h"
#import "AppDelegate.h"
#import "LocationAtShop.h"
@implementation LocationAtShopPickerTF
#define debug 1
- (void)fetch {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }

    CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:@"LocationAtShop"];
    NSSortDescriptor *sort =

```

```

[NSSortDescriptor sortDescriptorWithKey:@"aisle" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sort]];
[request setFetchBatchSize:50];
NSError *error;
self.pickerData = [cdh.context executeFetchRequest:request
                        error:&error];

if (error) {
    NSLog(@"Error populating picker: %@", %@,
          error, error.localizedDescription);
}
[self selectDefaultRow];
}
- (void)selectDefaultRow {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedObjectID && [self.pickerData count] > 0) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[UIApplication sharedApplication] delegate] cdh;
    LocationAtShop *selectedObject = (LocationAtShop*)[cdh.context
        existingObjectWithID:self.selectedObjectID
        error:nil];

    [self.pickerData enumerateObjectsUsingBlock:^(
        LocationAtShop *locationAtShop, NSUInteger idx, BOOL *stop) {
        if ([locationAtShop.aisle compare:selectedObject.aisle]
            == NSOrderedSame) {
            [self.picker selectRow:idx inComponent:0 animated:NO];
            [self.pickerDelegate selectedObjectID:self.selectedObjectID
                changedForPickerTF:self];

            *stop = YES;
        }
    }];
}

- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}

    LocationAtShop *locationAtShop = [self.pickerData objectAtIndex:row];
    return locationAtShop.aisle;
}
@end

```

请按下列步骤修改 Grocery Dude，以创建 LocationAtShopPickerTF 类：

1. 选定名为 **Grocery Dude Picker Text Fields** 的组。
2. 点击 **File > New > File...** 菜单项。

3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。

4. 把 **Subclass of** 设为 **CoreDataPickerTF**，把 **Class** 名称设为 **LocationAtShopPickerTF**，然后点击 **Next** 按钮。

5. 确保 **Targets** 中的“**Grocery Dude**”处于勾选状态，然后点击 **Create** 按钮，在 **Grocery Dude** 项目的目录下创建类。

6. 用程序清单 7-13 中的代码把 **LocationAtShopPickerTF.m** 文件里原有的代码全都替换掉。

7.6.1 创建 Location Picker

准备好 **LocationAtHomePickerTF** 及 **LocationAtShopPickerTF** 之后，我们需要在 **ItemVC** 上新建两个文本框。

请按下列步骤修改 **Grocery Dude**，以创建 **Home Location Picker** 文本框及 **Shop Location Picker** 文本框：

1. 选定 **Main.storyboard**。

2. 向 **Item** 视图中的 **Scroll View** 里拖放两个文本框，摆在任意位置均可，然后在 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面）中，按照下列步骤配置它们：

- ☐ 把这两个新文本框的 **Font** 都设为 **System Bold 17.0**。
- ☐ 把二者的 **Text Alignment** 都设为 **Center**。
- ☐ 把二者的 **Border Style** 都设为 **Line**（**Attributes Inspector** 界面以矩形来表示该选项）。
- ☐ 把二者的 **Background** 都设为 **Other > Crayons > Mercury**（**Mercury** 指的是第二个颜色最淡的灰色蜡笔）。
- ☐ 把其中一个文本框的 **Placeholder Text** 设为 **Location at Home**，另一个的设为 **Location at Shop**。

3. 在 **Identity Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）中，将 **Location at Home** 文本框的 **Custom Class** 设为 **LocationAtHomePickerTF**，将 **Location at Shop** 文本框的 **Custom Class** 设为 **LocationAtShopPickerTF**。

4. 用 **Size Inspector** 界面（可按“**Option + ⌘ + 5**”组合键调出该界面）把两个文本框的 **Height** 都设为 **48**。

5. 请按照图 7-5 所示的边线位置来排布这两个文本框。然后点击 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints in ItemVC**。

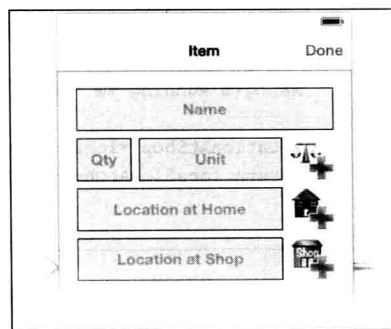


图 7-5 Home Location Picker 文本框与 Shop Location Picker 文本框

7.6.2 连接 Location 选取器

为了能在代码中引用新的选取器文本框，我们需要将其连接到对应的 Outlet 上面。该操作可以通过 Assistant Editor 来完成。

请按下列步骤修改 Grocery Dude，将 Home Location 选取器文本框与 Shop Location 选取器文本框连接到 ItemVC.h：

1. 将 #import "LocationAtHomePickerTF.h" 语句添加到 ItemVC.h 文件顶部。
2. 将 #import "LocationAtShopPickerTF.h" 语句添加到 ItemVC.h 文件顶部。
3. 选定 **Main.storyboard**。
4. 确保 **Item** 视图控制器处于选中状态。
5. 把 **Assistant Editor** 界面显示出来（可按“**Option + ⌘ + Return**”组合键调出该界面）。
6. 如果 Assistant Editor 顶部显示的不是 **Automatic > ItemVC.h**，那就将其调整为 **Automatic > ItemVC.h**。

7. 按住 **Control** 键，从 **Location at Home** 文本框向 ItemVC.h 文件的 @end 上方拖一条线。把新特性的 **Name** 设为 homeLocationPickerTextField。请确认 **Type** 是 LocationAtHomePickerTF，**Storage** 是 **Strong**。

8. 按住 **Control** 键，从 **Location at Shop** 文本框向 ItemVC.h 文件的 @end 上方拖一条线。把新特性的 **Name** 设为 shopLocationPickerTextField。请确认 **Type** 是 LocationAtShopPickerTF，**Storage** 是 **Strong**。

9. 把 **Standard Editor** 界面显示出来（可按“**⌘ + Return**”组合键调出该界面）。

查看 ItemVC.h 文件我们会发现，与选取器文本框相连接的那些特性左侧会出现中间含有圆点的圆圈。正常的结果应该如图 7-6 所示。

```
#import "LocationAtHomePickerTF.h"
#import "CoreDataHelper.h"
#import "UnitPickerTF.h"
#import "LocationAtHomePickerTF.h"
#import "LocationAtShopPickerTF.h"

@interface ItemVC : UIViewController <UITextFieldDelegate, CoreDataPickerTFDelegate>
@property (strong, nonatomic) NSManagedObjectID *selectedItemID;
* @property (strong, nonatomic) IBOutlet UIScrollView *scrollView;
* @property (strong, nonatomic) IBOutlet UITextField *nameTextField;
* @property (strong, nonatomic) IBOutlet UITextField *quantityTextField;
* @property (strong, nonatomic) IBOutlet UnitPickerTF *unitPickerTextField;
* @property (strong, nonatomic) IBOutlet LocationAtHomePickerTF *homeLocationPickerTextField;
* @property (strong, nonatomic) IBOutlet LocationAtShopPickerTF *shopLocationPickerTextField;
@end
```

图 7-6 与选取器文本框相连的特性

7.6.3 为 Location 选取器配置 ItemVC

为了使 Home Location 选取器与 Shop Location 选取器像 Unit 选取器那样能够显示出最新的数据，我们需要把对应的 fetch 方法公布出来，而且还要编写调用 fetch 及

reloadAllComponents 方法的语句。程序清单 7-14 以粗体代码标出了相关的改动。

程序清单7-14 ItemVC.m文件的textFieldDidBeginEditing方法

```

- (void)textFieldDidBeginEditing:(UITextField *)textField {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    if (textField == self.nameTextField) {
        if ([self.nameTextField.text isEqualToString:@"New Item"]) {
            self.nameTextField.text = @"";
        }
    }
    if (textField == _unitPickerTextField && _unitPickerTextField.picker) {
        [_unitPickerTextField fetch];
        [_unitPickerTextField.picker reloadAllComponents];
    } else if (textField == _homeLocationPickerTextField &&
        _homeLocationPickerTextField.picker) {
        [_homeLocationPickerTextField fetch];
        [_homeLocationPickerTextField.picker reloadAllComponents];
    } else if (textField == _shopLocationPickerTextField &&
        _shopLocationPickerTextField.picker) {
        [_shopLocationPickerTextField fetch];
        [_shopLocationPickerTextField.picker reloadAllComponents];
    }
}

```

请按下列步骤修改 Grocery Dude，以确保 Home Location 选取器与 Shop Location 选取器总能显示出最新的数据：

1. 把下列代码分别添加到 LocationAtHomePickerTF.h 及 LocationAtShopPickerTF.h 文件底部的 @end 上方：

```
- (void)fetch;
```

2. 用程序清单 7-14 中的代码把 ItemVC.m 文件里原有的 textFieldDidBeginEditing 方法替换掉。

与 unitPickerTextField 一样，我们也要用相同的方式把 ItemVC 设置成 homeLocationPickerTextField 与 shopLocationPickerTextField 的 delegate。程序清单 7-15 列出了配置 delegate 所用的代码。

程序清单7-15 ItemVC.m文件的viewDidLoad方法

```

self.homeLocationPickerTextField.delegate = self;
self.homeLocationPickerTextField.pickerDelegate = self;
self.shopLocationPickerTextField.delegate = self;
self.shopLocationPickerTextField.pickerDelegate = self;

```

请按下列步骤修改 Grocery Dude，以配置 homeLocationPickerTextField 与 shop-

LocationPickerTextField 的委托:

修改 ItemVC.m 文件的 viewDidLoad 方法, 把程序清单 7-15 中的代码添加到该方法底部。

此外, 为了配合 Home Location 选取器文本框与 Shop Location 选取器文本框, 我们还需修改 refreshInterface 方法。程序清单 7-16 列出了相关代码。

程序清单7-16 ItemVC.m文件的refreshInterface方法

```

- (void)refreshInterface {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    if (self.selectedItemID) {
        CoreDataHelper *cdh =
            [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
        Item *item =
            (Item*)[cdh.context existingObjectWithID:self.selectedItemID
                error:nil];

        self.nameTextField.text = item.name;
        self.quantityTextField.text = item.quantity.stringValue;
        self.unitPickerTextField.text = item.unit.name;
        self.unitPickerTextField.selectedObjectID = item.unit.objectID;
        self.homeLocationPickerTextField.text =
            item.locationAtHome.storedIn;
        self.homeLocationPickerTextField.selectedObjectID =
            item.locationAtHome.objectID;
        self.shopLocationPickerTextField.text =
            item.locationAtShop.aisle;
        self.shopLocationPickerTextField.selectedObjectID =
            item.locationAtShop.objectID;
    }
}

```

请按下列步骤修改 Grocery Dude, 以确保 Home Location 文本框及 Shop Location 文本框能够显示出适当的信息:

用程序清单 7-16 中的方法把 ItemVC.m 文件里原有的 **refreshInterface** 方法替换掉。

最后, 为了把 Home Location 选取器及 Shop Location 选取器实现好, ItemVC.m 文件的 PICKERS 部分中还有一些方法也需要修改。ItemVC.m 文件的 selectedObjectID:changedForPickerTF 方法里有个 if/else 语句, 当系统调用委托方法时, 我们用它来判断并处理不同的选取器。程序清单 7-17 以粗体标出了新的代码。

程序清单7-17 ItemVC.m文件的selectedObjectID:changedForPickerTF方法

```

- (void)selectedObjectID:(NSManagedObjectID *)objectID
    changedForPickerTF:(CoresDataPickerTF *)pickerTF {
    if (debug==1) {

```

```

        NSLog(@"Running %@", NSStringFromClass(_cmd));
    }

    if (self.selectedItemID) {
        CoreDataHelper *cdh =
            [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];

        Item *item =
            (Item*) [cdh.context existingObjectWithID:self.selectedItemID
                                error:nil];

        NSError *error;
        if (pickerTF == self.unitPickerTextField) {
            Unit *unit = (Unit*) [cdh.context existingObjectWithID:objectID
                                error:&error];

            item.unit = unit;
            self.unitPickerTextField.text = item.unit.name;
        }
        else if (pickerTF == self.homeLocationPickerTextField) {
            LocationAtHome *locationAtHome =
                (LocationAtHome*) [cdh.context existingObjectWithID:objectID
                                error:&error];

            item.locationAtHome = locationAtHome;
            self.homeLocationPickerTextField.text =
                item.locationAtHome.storedIn;
        }
        else if (pickerTF == self.shopLocationPickerTextField) {
            LocationAtShop *locationAtShop =
                (LocationAtShop*) [cdh.context existingObjectWithID:objectID
                                error:&error];

            item.locationAtShop = locationAtShop;
            self.shopLocationPickerTextField.text =
                item.locationAtShop.aisle;
        }
        [self refreshInterface];
        if (error) {
            NSLog(@"Error selecting object on picker: %@", @"",
                error, error.localizedDescription);
        }
    }
}
}

```

请按下列步骤修改 Grocery Dude，以更新 PICKERS 部分中的第一个方法：

用程序清单 7-17 中的代码来替换 ItemVC.m 文件里原有的 **selectedObjectID:changedForPickerTF** 方法。

我们也用同样的办法来修改 ItemVC.m 文件中的 **selectedObjectClearedForPickerTF** 方法。该方法会把 item 的 LocationAtHome 或 LocationAtShop 设成 nil，而这又将迫使应用程序把它们设置为 “..Unknown..” 对象。程序清单 7-18 用粗体标出了新加进来的代码。

程序清单7-18 ItemVC.m文件的selectedObjectClearedForPickerTF方法

```

- (void)selectedObjectClearedForPickerTF:(CoreDataPickerTF *)pickerTF {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*) [cdh.context existingObjectWithID:self.selectedItemID
                                     error:nil];

    if (pickerTF == self.unitPickerTextField) {
        item.unit = nil;
        self.unitPickerTextField.text = @"";
    }
    else if (pickerTF == self.homeLocationPickerTextField) {
        item.LocationAtHome = nil;
        self.homeLocationPickerTextField.text = @"";
    }
    else if (pickerTF == self.shopLocationPickerTextField) {
        item.LocationAtShop = nil;
        self.shopLocationPickerTextField.text = @"";
    }
    [self refreshInterface];
}
}

```

请按下列步骤修改 Grocery Dude，以更新 PICKERS 部分中的第二个方法：

用程序清单 7-18 中的代码把 ItemVC.m 文件里原有的 selectedObjectCleared-ForPickerTF 方法替换掉。

所有选取器视图都已完全配置好。运行应用程序，创建一些 Unit、Home Location 与 Shop Location。

7.7 使选取器不遮住文本框

程序在显示选取器的时候，屏幕上用于显示相关文本框的空间就变小了，这意味着选取器有可能会把相关的文本框（指“活动的”文本框）挡在其后面。为了使用户能在 Item 视图中看到当前的活动文本框，我们需要根据程序是否显示键盘来调整 Scroll 视图的尺寸。调整好 Scroll 视图的尺寸之后，就可以用 UIScrollView 类的 scrollRectToVisible 方法把活动文本框显示在用户可以看到的范围内了。下面这两个方法用于实现此功能：

❑ **keyboardDidShow** 方法用来查找键盘输入视图顶端的坐标（也就是选取器的坐

标), 并据此来调整 scrollView 的大小。具体的偏移量会因为当前的屏幕方向而有所不同。scrollView 的框架需要与屏幕上所剩的可见区域大小相符, 保证了这一点之后, 就可以把活动文本框移动到用户可以看见的范围内了。

❑ **keyboardWillHide** 方法所做的事情与 keyboardWillShow 相似, 只不过它会把 scrollView 扩大, 而不是缩小。

程序清单 7-19 列出了相关的代码。

程序清单7-19 ItemVC.m文件的INTERACTION部分

```
- (void)keyboardDidShow:(NSNotification *)n {

    // Find top of keyboard input view (i.e. picker)
    CGRect keyboardRect =
    [[[n userInfo] objectForKey:UIKeyboardFrameEndUserInfoKey] CGRectValue];
    keyboardRect = [self.view convertRect:keyboardRect fromView:nil];
    CGFloat keyboardTop = keyboardRect.origin.y;

    // Resize scroll view
    CGRect newScrollViewFrame =
    CGRectMake(0, 0, self.view.bounds.size.width, keyboardTop);
    newScrollViewFrame.size.height = keyboardTop - self.view.bounds.origin.y;
    [self.scrollView setFrame:newScrollViewFrame];

    // Scroll to the active Text-Field
    [self.scrollView scrollRectToVisible:self.activeField.frame animated:YES];
}
- (void)keyboardWillHide:(NSNotification *)n {
if (debug==1) {
    NSLog(@"Running %@ '%@'", self.class, NSStringFromSelector(_cmd));
}
    CGRect defaultFrame =
    CGRectMake(self.scrollView.frame.origin.x,
               self.scrollView.frame.origin.y,
               self.view.frame.size.width,
               self.view.frame.size.height);

    // Reset Scrollview to the same size as the containing view
    [self.scrollView setFrame:defaultFrame];

    // Scroll to the top again
    [self.scrollView scrollRectToVisible:self.nameTextField.frame
      animated:YES];
}
```

请按下列步骤修改 Grocery Dude, 以添加 INTERACTION 部分:

1. 把下列特性添加到 ItemVC.h 文件的 @end 上方。该特性用来保存指向活动文本框的引用:

```
@property (strong, nonatomic) IBOutlet UITextField *activeField;
```

2. 把程序清单 7-19 中的代码添加到 ItemVC.m 文件 INTERACTION 部分的底部。

3. 修改 ItemVC.m 文件的 textFieldDidBeginEditing 方法, 把 _activeField=textField; 语句添加到该方法底部。

4. 修改 ItemVC.m 文件的 textFieldDidEndEditing 方法, 把 _activeField= nil; 语句添加到该方法底部。

接下来需要确保程序在显示或隐藏键盘时, 能够调用程序清单 7-19 中的那两个方法, 这可以通过监听 UIKeyboardDidShowNotification 与 UIKeyboardWillHideNotification 事件来实现。相关代码列在程序清单 7-20 里。

程序清单7-20 ItemVC.m文件的viewWillAppear方法

```
// Register for keyboard notifications while the view is visible.
[[NSNotificationCenter defaultCenter] addObserver:self
                                     selector:@selector(keyboardDidShow:)
                                     name:UIKeyboardDidShowNotification
                                     object:self.view.window];
[[NSNotificationCenter defaultCenter] addObserver:self
                                     selector:@selector(keyboardWillHide:)
                                     name:UIKeyboardWillHideNotification
                                     object:self.view.window];
```

请按下列步骤修改 Grocery Dude, 以监听并响应键盘事件:

修改 ItemVC.m 文件的 viewWillAppear 方法, 把程序清单 7-20 中的代码添加到该方法顶部。

当 Item 视图未出现在屏幕上面时, 就没有必要继续监听键盘事件了。在 viewDidDisappear 方法中移除这些监听器是比较合适的。程序清单 7-21 列出了相关代码。

程序清单7-21 ItemVC.m文件的viewDidDisappear方法

```
// Unregister for keyboard notifications while the view is not visible.
[[NSNotificationCenter defaultCenter] removeObserver:self
                                     name:UIKeyboardDidShowNotification
                                     object:nil];
[[NSNotificationCenter defaultCenter] removeObserver:self
                                     name:UIKeyboardWillHideNotification
                                     object:nil];
```

请按下列步骤修改 Grocery Dude, 以停止监听键盘事件:

修改 ItemVC.m 文件的 viewDidDisappear 方法, 把程序清单 7-21 中的代码添加到该方法底部。

再次运行应用程序, 确保持久化存储区里面已经添加有一些 Shop Location 了。然后在 Item 视图界面中点击 Shop Location 选取器文本框, 修改某件货品在商店中的位置 (shop

location)。你应该会注意到：Shop Location 选取器文本框会自动移入用户的视野中。

7.8 小结

本章先讲解了怎样把通过 Core Data 所获取到的结果绑定到选取器，然后描述了如何从自定义的文本框中触发 `inputView`，以便将这种选取器显示出来。在此过程中，我们完全实现了 Grocery Dude 程序所需的 Picker 式文本框，如此一来，用户就可以非常迅速地配置货品的信息了。当持久化存储区中的 Unit、Home Location 及 Shop Location 增多了之后，这种优势就会体现得更加明显。

7.9 习题

请基于所学知识完成下列试验。

1. 临时修改 `CoreDataPickerTF.m` 文件的 `createInputAccessoryView` 方法，把 `self.showToolbar` 设为 NO，然后运行应用程序，看看每个选取器视图的工具栏是不是都消失了。请注意，假如只想在某些特定的选取器中隐藏工具栏，那么可于相应的子类中覆写该方法。
2. 临时修改某个 `CoreDataPickerTF` 子类，再添加一个数组，以创建具有多个组件的选取器。请注意，必须覆写 `numberOfComponentsInPickerView` 方法才能完成此操作。



预先加载数据

能够查到的东西不用记。

——阿尔伯特·爱因斯坦

第 7 章着重讲解了如何用 Core Data 对象来配置用户界面上的各个元件，之后，Grocery Dude 的主要功能就已经实现好了。本章回过头来谈谈数据模型，讲解并演示如何预先加载默认数据。想要为应用程序提供默认数据有好几种办法。某些情况下，可以像早前章节那样，直接在代码里导入数据，但更先进一些的做法则是根据 XML 文件来生成持久化存储区。这种存储区可以随应用程序一同发布，使程序在首次设定 Core Data 之前，先将其用作初始的持久化存储区。

8.1 默认的数据

发布 Core Data 应用程序的时候，开发者可能想放一些默认的数据进去。在某些情况下，这些默认数据只是为了演示程序的用法，而在另外一些情况下，这些默认数据却是必不可少的，否则应用程序就没有实际意义了。假如 Grocery Dude 程序不包含默认数据，那么用户可能无法快速掌握它的用法，尤其是在 Item 视图中面对着空的选取器视图时会更加不知所措。要是能够包含一些默认数据的话，那么用起来就会简单一些。程序做得越易用，用户就越有可能继续用它，而用得越久，就越有可能把程序推荐给其他用户，这样也就提升了它的销售潜力。

在为应用程序导入默认数据之前，应该先进行两次确认：

- ❑ 应用程序确实需要导入数据。
- ❑ 用户确实想导入默认的数据（该项可选）。

默认数据自何处导入会因具体情况而有很大差别。但不论数据源是什么，我们都可以将原始数据放入电子表格，并用下面将要讲到的办法把它做成 XML 文件，然后据此生成持久化存储区，再随应用程序一同发布。开发者一定要注意，供 Core Data 所使用的持久化存储区必须由 Core Data 来生成。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter07.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项，这样可以清除掉同名项目所残留的缓存。

8.2 判断应用程序是否需要导入数据

开发者可在持久化存储区的元数据里放入一个适当的值，以表示应用程序不需要导入数据。每次运行程序时都检测该值，以确认这次是否要执行数据导入操作。这个办法就像安全开关一样，可防止应用程序导入重复的默认数据。程序初次启动时，元数据里并没有该值，因此它可以导入默认数据。

你也许还想再加一层防护，也就是令用户来确认是否需要导入默认数据，以防止重复导入。Grocery Dude 程序也将采用这种办法，把是否导入数据交由用户来决定。加上这层检测之后，还会带来一个好处：万一程序由于 bug 等原因不小心触发了数据导入操作，那么用户还有机会取消该操作。我们用 UIAlertView 来询问用户是否想继续执行数据导入操作，而 UIAlertViewDelegate 则用于接收并处理用户所做的决定。

请按下列步骤修改 Grocery Dude，以添加 importalertView：

1. 参照下列代码修改 CoreDataHelper.h 的接口声明，令其采用 UIAlertViewDelegate 协议：

```
@interface CoreDataHelper : NSObject <UIAlertViewDelegate>
```

2. 把下列特性添加到 CoreDataHelper.h 文件里现有的各条特性声明语句下方：

```
@property (nonatomic, retain) UIAlertView *importalertView;
```

当 Grocery Dude 程序启动并设定好持久化存储区之后，就该判断是否需要导入默认数据了。程序清单 8-1 列出了 DATA_IMPORT 部分中的第一个方法，该部分是新加进来的。

程序清单8-1 CoreDataHelper.m文件中的isDefaultDataAlreadyImportedForStoreWithURL方法

```
#pragma mark - DATA_IMPORT
- (BOOL)isDefaultDataAlreadyImportedForStoreWithURL:(NSURL*)url
```

```

                                ofType:(NSString*)type {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
NSError *error;
NSDictionary *dictionary =
    [NSPersistentStoreCoordinator metadataForPersistentStoreOfType:type
                                     URL:url
                                     error:&error];

if (error) {
    NSLog(@"Error reading persistent store metadata: %@",
          error.localizedDescription);
}
else {
    NSNumber *defaultDataAlreadyImported =
        [dictionary valueForKey:@"DefaultDataImported"];
    if (![defaultDataAlreadyImported boolValue]) {
        NSLog(@"Default Data has NOT already been imported");
        return NO;
    }
}
if (debug==1) {NSLog(@"Default Data HAS already been imported");}
return YES;
}

```

程序会向 `isDefaultDataAlreadyImportedForStoreWithURL` 方法查询某个特定的存储区中是否已经导入了默认的数据，而该方法则负责返回 YES 或 NO。它的具体做法是：以 `"DefaultDataImported"` 为键，在元数据中寻找是否有值与之对应。如果没有值与它相对应，或者虽然有值，但值却是 NO，那么就表明程序应该导入默认的数据。



提示 `"DefaultDataImported"` 这个键名是随意（或随机）取的。键名本身并不重要，重要的是它必须和接下来要讲的 `setDefaultDataAsImportedForStoreWithURL` 方法所使用的键名相同，而那个方法负责把存储区标注为“已导入”（imported）。

请按下列步骤修改 Grocery Dude，以添加 DATA_IMPORT 部分：

1. 把程序清单 8-1 中的代码添加到 `CoreDataHelper.m` 文件底部，并放在 `@end` 上方。

接下来要实现的方法是 `checkIfDefaultDataNeedsImporting`。该方法会调用 `isDefaultDataAlreadyImportedForStoreWithURL` 方法来检测应用程序是否需要导入数据。如果需要，那么还要向用户显示 `importAlert`，令其确认这次导入操作；如果不需要，那就不执行任何操作。程序清单 8-2 列出了相关代码。

程序清单 8-2 CoreDataHelper.m 文件中的 `checkIfDefaultDataNeedsImporting` 方法

```

- (void)checkIfDefaultDataNeedsImporting {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
}

```

```

        if (![self isDefaultDataAlreadyImportedForStoreWithURL:[self storeURL]
                                                    ofType:NSSQLiteStoreType]) {
            self.importAlertView =
                [[UIAlertView alloc] initWithTitle:@"Import Default Data?"
                                                message:
@"If you've never used Grocery Dude before then some default data might
help you understand how to use it. Tap 'Import' to import default data.
Tap 'Cancel' to skip the import, especially if you've done this before on other
devices."
                                                delegate:self
                                                cancelButtonTitle:@"Cancel"
                                                otherButtonTitles:@"Import", nil];

            [self.importAlertView show];
        }
    }
}

```

请按下列步骤修改 Grocery Dude，编写代码来判断是否需要导入默认数据：

1. 把程序清单 8-2 中的代码添加到 CoreDataHelper.m 文件的 DATA_IMPORT 部分底部，并放在 @end 上方。

2. 修改 CoreDataHelper.m 文件的 setupCoreData 方法，把 [self checkIfDefaultDataNeedsImporting]; 语句添加到方法底部。

做好上述修改之后，运行应用程序。结果应该如图 8-1 所示。由于我们还没有实现导入数据所需的代码，所以目前的程序在每次启动时都要显示这个询问对话框。

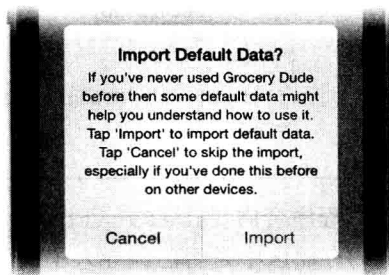


图 8-1 询问用户是否想要加载默认的数据

8.3 从 XML 中导入数据

学会如何从 XML 中导入数据之后，就可以用这项技术来生成包含默认数据的持久化存储区了。有了这个默认数据持久化存储区，我们就可以把它与应用程序一并发布，而不再需要 XML 文件了。这样做的好处是：由于应用程序能够直接使用默认数据，所以不需要执行导入 XML 文件的过程。

很多优秀的 XML 解析器都可用来创建包含默认数据的存储区。虽说某些解析器的性能比较高，但 iOS SDK 中包含的 NSXMLParser 足以实现我们的需求。由于创建默认数据存储区并不是一个需要用户耐心等待的漫长过程，所以无须担心性能问题。NSXMLParser 是个由事件所驱动的流解析器，也就是说，调用了 NSXMLParser 实例的 parse 方法之后，解析器就会把它在 XML 文件里发现的东西逐个通知给 delegate。



想从文件中导入数据，XML并不是唯一的选项。例如你也可以用NSJSON-Serialization从JSON文件里导入数据，或是用property list（特性列表）充当数据源。

请按下列步骤修改 Grocery Dude，以采用 NSXMLParserDelegate 协议：

1. 参照下列代码来修改 CoreDataHelper.h 文件的接口声明，使 CoreDataHelper 采用 NSXMLParserDelegate 协议：

```
@interface CoreDataHelper : NSObject <UIAlertViewDelegate,
NSXMLParserDelegate>
```

下一步是实现 importFromXML。该方法负责把 CoreDataHelper 实例配置成 NSXMLParser 的委托，然后调用 parse，以触发 XML 文件的解析过程。解析完成之后，向表格视图发送通知，确保它能够用最新的数据来刷新自己。假如 context 是导入的上下文的父 context，那就不用发送这条通知了。第 11 章将会讨论父上下文层级（parent context hierarchy）。

程序清单 8-3 列出了该方法的实现代码。

程序清单8-3 CoreDataHelper.m文件中的importFromXML方法

```
- (void)importFromXML:(NSURL*)url {
if (debug==1) {
    NSLog(@"Running %@", @"%", self.class, NSStringFromSelector(_cmd));
}

self.parser = [[NSXMLParser alloc] initWithContentsOfURL:url];
self.parser.delegate = self;

NSLog(@"***** START PARSE OF %@", url.path);
[self.parser parse];
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"SomethingChanged" object:nil];
NSLog(@"***** END PARSE OF %@", url.path);
}
```

请按下列步骤修改 Grocery Dude，编写代码来触发 XML 解析器：

1. 把下列特性添加到 CoreDataHelper.h 文件现有的各条特性声明语句下面：

```
@property (nonatomic, strong) NSXMLParser *parser;
```

2. 把程序清单 8-3 中的代码添加到 CoreDataHelper.m 文件 DATA_IMPORT 部分的底部，并放在 @end 之上。

在导入数据之前，先得有个包含数据的 XML 文件才行。Grocery Dude 所需的 XML 格式如程序清单 8-4 所示。

程序清单8-4 包含默认数据的XML格式范例文件

```

<items>
<item name="" unit="" locationathome="" locationatshop="" ></item>
</items>

```

用 Numbers 或 Excel 这样的电子表格编辑器来创建 XML 格式的文件是非常简单的。我们可以先把现有的数据粘贴到电子表格中，然后像图 8-2 这样，把待创建的 XML 字符串里相关的部分插到对应的列中。使用 Numbers 或 Excel 的好处是可以批量填写重复出现的 XML 标记（XML tag）。当然你也可以采用自己喜欢的编辑器来制作 XML 文件，只要保证格式正确就行。

| | A | B | C | D | E | F | G | H | I |
|----|--------------|--------------------------------------|----------|------|--------------------|---------|--------------------|---------|----------|
| 3 | <item name=" | After Shave | " unit=" | " | locationathome=" | Ensuite | " locationatshop=" | Aisle 6 | ></item> |
| 4 | <item name=" | Air Freshener | " unit=" | can | " locationathome=" | Ensuite | " locationatshop=" | Aisle 7 | ></item> |
| 5 | <item name=" | Aluminum Foil | " unit=" | roll | " locationathome=" | Pantry | " locationatshop=" | Aisle 2 | ></item> |
| 6 | <item name=" | Apple Sauce | " unit=" | jar | " locationathome=" | Fridge | " locationatshop=" | Aisle 2 | ></item> |
| 7 | <item name=" | Apples | " unit=" | " | " locationathome=" | Fridge | " locationatshop=" | Produce | ></item> |
| 8 | <item name=" | Arborio Rice | " unit=" | pk | " locationathome=" | Pantry | " locationatshop=" | Produce | ></item> |
| 9 | <item name=" | Avocado | " unit=" | " | " locationathome=" | Fridge | " locationatshop=" | Produce | ></item> |
| 10 | <item name=" | Baby Clothes Washing Liquid - Purity | " unit=" | " | " locationathome=" | Laundry | " locationatshop=" | Aisle 7 | ></item> |
| 11 | <item name=" | Baby Corn | " unit=" | " | " locationathome=" | Fridge | " locationatshop=" | Produce | ></item> |

图 8-2 用 Excel 或 Numbers 来创建 XML 文件

图 8-2 用两种颜色表示各列，灰色代表数据，白色代表 XML 元素。等程序导入这些数据的时候，表格里的每一行就会变成一个基于 Item 实体的托管对象了。准备好电子表格之后，只需将其保存成后缀名是 XML 的纯文本文件即可。电子表格转换成文本时，可能会多出来一些制表位或空格，把它们删去之后，这份 XML 文件就可以使用了。由于笔者预先已经准备了一份包含大约四百件货品的 XML 文件，所以我们现在只需专心构建 Grocery Dude 即可。

请按下列步骤修改 Grocery Dude，将包含默认数据的 XML 文件添加到项目中：

1. 从 <http://www.timroadley.com/LearningCoreData/DefaultData.xml.zip> 下载 zip 文件，并将笔者预先制作的 **DefaultData.xml** 解压缩。
2. 把 **DefaultData.xml** 拖放到名为 **Data Model** 的组里。请确认 **Copy items into destination group's folder** 选项及 Targets 中的“Grocery Dude”处于勾选状态，然后点击 **Finish** 按钮。在 Xcode 中点击 XML 文件，将会看到如图 8-3 所示的画面。

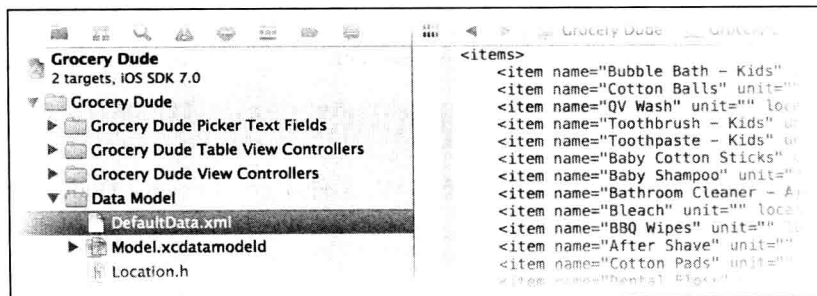


图 8-3 包含默认数据的 XML 文件，该文件已经可以导入了

8.4 创建导入默认数据所需的上下文

导入数据时所使用的托管对象上下文不应该运行在主线程上面。笔者推荐单独为数据导入操作创建一个上下文，这样它就不会阻塞其他上下文所在的队列了，尤其是不会阻塞正在使用其他上下文的那个主队列。导入默认数据所用的这个上下文[Ⓐ]，可以和其他上下文共用同一个持久化存储协调器。为了减少导入上下文的资源需求，笔者建议把撤销管理器（undo manager）设为 nil，以将其禁用。虽说 iOS 本身就会把上下文的撤销管理器默认设为 nil，但为了防止将来 iOS 改变这一默认设定，我们最好还是明确将其设为 nil 比较好。

请按下列步骤修改 Grocery Dude，以添加 importContext 特性：

把下列特性添加到 CoreDataHelper.h 文件中，将其放在现有的 context 特性声明语句下方：

```
@property (nonatomic, readonly) NSManagedObjectContext
*importContext;
```

import 上下文的实现方式与前台上下文（foreground 上下文）相同，但是它的 ConcurrencyType 却与之不同。配置 NSManagedObjectContext 时，可在下列三种 ConcurrencyType 里选择其一：

1. NSMainQueueConcurrencyType 如果上下文要在主线程上执行操作，那么就应该使用这种 ConcurrencyType。假如把繁重的任务放在主队列上完成，那么就可能导致用户界面反应不畅，甚至失去响应。开发者至少应该留一个上下文在前台运行，以便更新用户界面上的元件。

2. NSPrivateQueueConcurrencyType 假如上下文不在主线程上执行操作，那么就应该用这种 ConcurrencyType。如果要执行保存数据或导入数据等可能比较繁重的工作，那么选择这种 ConcurrencyType 会比较好。

3. NSConfinementConcurrencyType 这是默认的遗留选项，开发者一般不应该使用此选项，除非需要兼容 iOS 版本低于 5.0 的设备。

如果某个上下文的 ConcurrencyType 是 NSPrivateQueueConcurrencyType，那么开发者只应该向其发送 performBlock 或 performBlockAndWait 消息。假如不关心块何时返回，那么可以使用 performBlock；假如必须等块返回之后才能往下执行，那么可以使用 performBlockAndWait。若上下文运行在主线程上，而开发者调用的又是 performBlockAndWait，那么就会令主线程阻塞。程序清单 8-5 里的代码会把 importContext 放在“专用队列”（private queue，又叫后台队列（background queue））上面运行。

程序清单8-5 CoreDataHelper.m文件的init方法

```
_importContext = [[NSManagedObjectContext alloc]
initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_importContext performBlockAndWait:^(
    [_importContext setPersistentStoreCoordinator:_coordinator];
    [_importContext setUndoManager:nil]; // the default on iOS
)];
```

Ⓐ 作者将这种上下文称为导入上下文（import context），下同。——译者注

请按下列步骤修改 Grocery Dude，以实现导入上下文：

1. 修改 CoreDataHelper.m 文件的 init 方法，把程序清单 8-5 中的代码添加到 return self; 这一行之上。

8.5 防止重复导入默认数据

为了防止应用程序多次导入默认数据，我们应该在元数据中设置值为 YES 的 DefaultDataImported 键，以应用于持久化存储区。为此，需要先把持久化存储区里包含现有元数据的那个 NSDictionary 拷贝一份，然后将 DefaultDataImported 键添加到拷贝出来的这份元数据字典中，再把其重新放回持久化存储区。此过程由 setDefaultDataAsImportedForStore 方法完成，其代码如程序清单 8-6 所示。

程序清单8-6 CoreDataHelper.m文件中的setDefaultDataAsImportedForStore方法

```
- (void)setDefaultDataAsImportedForStore:(NSPersistentStore*)aStore {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }

    // get metadata dictionary
    NSMutableDictionary *dictionary =
        [NSMutableDictionary dictionaryWithDictionary:[aStore metadata] copy];
    if (debug==1) {
        NSLog(@"__Store Metadata BEFORE changes__ \n %@", dictionary);
    }

    // edit metadata dictionary
    [dictionary setObject:@YES forKey:@"DefaultDataImported"];

    // set metadata dictionary
    [self.coordinator setMetadata:dictionary forPersistentStore:aStore];

    if (debug==1) {NSLog(@"__Store Metadata AFTER changes__ \n %@", dictionary);}
}
```

请按下列步骤修改 Grocery Dude，用代码将默认数据标注成“已导入”(imported) 状态：

1. 把程序清单 8-6 中的代码添加到 CoreDataHelper.m 文件 DATA_IMPORT 部分的底部，并放在 @end 之上。

8.6 触发导入默认数据的操作

importAlertView 显示出来的时候，用户可以点击 Cancel 按钮来跳过导入操作，也可以点击 Import 按钮来加载默认的数据。我们在 CoreDataHelper.m 文件里新建 DELEGATE: UIAlertView 部分，然后在其中实现名为 alertView:clickedButtonAtIndex 的方法，

以处理用户所做的选择。程序清单 8-7 列出了相关代码。

程序清单8-7 CoreDataHelper.m文件中的alertView:clickedButtonAtIndex方法

```
#pragma mark - DELEGATE: UIAlertView
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    if (alertView == self.importAlertView) {
        if (buttonIndex == 1) { // The 'Import' button on the importAlertView

            NSLog(@"Default Data Import Approved by User");
            [_importContext performBlock:^(
                // XML Import
                [self importFromXML:[NSBundle mainBundle]
                    URLForResource:@"DefaultData" withExtension:@"xml"]];
            });
        } else {
            NSLog(@"Default Data Import Cancelled by User");
        }
        // Set the data as imported regardless of the user's decision
        [self setDefaultDataAsImportedForStore:_store];
    }
}
```

当用户点击索引为 1 的按钮时，就会调用早前实现好的 importFromXML 方法。索引为 1 的按钮指的就是 Import 按钮。请注意，数据导入操作是用 performBlock 方法执行的，而不是用 performBlockAndWait 方法。也就是说，代码块执行完毕之后会自动返回应用程序，由于 performBlock 不阻塞当前线程，所以不会影响用户体验。

无论用户在 importAlertView 界面选了哪个选项，都要调用 setDefaultDataAsImportedForStore 方法，这样一来，应用程序以后就不会在每次启动时都去打扰用户了。假如用户选错了，那么目前还没有办法能够撤销这一决定。为简洁起见，我们先不讨论这个问题，而是把精力放在本章的主要任务上面。假如要在自己的应用程序中处理这种状况，那么可参考第 14 章，看看如何使用户能够通过 Settings 应用程序来打开或关闭 iCloud 功能。我们可以用同样的方式给用户提供一个 **import default data** 选项，使用户通过该选项来撤销由 setDefaultDataAsImportedForStore 方法所设定的标志。

请按下列步骤修改 Grocery Dude，实现相关的代码，以便触发导入默认数据的操作：

把程序清单 8-7 中的代码添加到 CoreDataHelper.m 文件底部的 @end 上方。

运行应用程序，并点击 Import 按钮，开始导入数据。正常的运行结果应该如图 8-4 所示。控制台日志里的信息顺序可能会与图中不同，原因在于 parse 方法并不是运行在主线程上的。请注意，setDefaultDataAsImportedForStoreWithURL 方法执行完毕之后，存储区的元数据里面就会包含 DefaultDataImported=1；这一行内容。

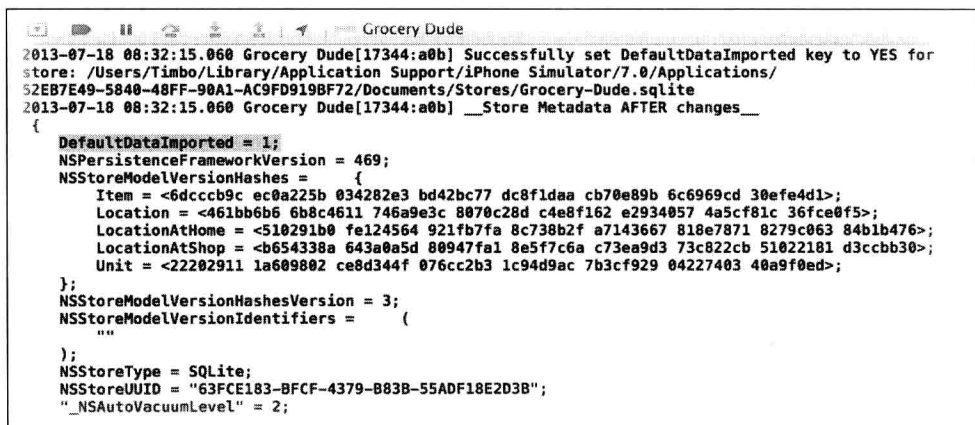


图 8-4 metadata 中的 DefaultDataImported 选项

虽说已经触发了 parse 方法，但并没有数据导入进来。这是因为 NSXMLParser 的 delegate 方法还没有实现。

8.7 创建 CoreDataImporter

为了根据 XML 文件来创建托管对象，我们需要指定数据间的映射关系。要想实现这一映射过程，开发者就必须熟知应用程序的源数据模型与目标数据模型。因为 Grocery Dude 程序的数据模型比较简单，所以这个过程很容易实现。Grocery Dude 程序所用的原理也同样适用于数据模型更为复杂的程序。为了使 CoreDataHelper 尽量保持清晰，我们新建名为 CoreDataImporter 的类。该类包含一些通用的方法，用来导入“互不相同的托管对象”（unique managed object）。由于这些方法较为通用，所以也适用于其他数据模型。

向目标上下文中插入新的对象之前，首先要保证待插入的对象并未出现在上下文中。由于数据是从 XML 导入的，所以若想保证唯一性，我们只能把源数据里某个属性的值同目标上下文里相关实体的属性值相比较，以此判断是否重复。具体到 Grocery Dude 程序，我们很容易就能确定这些属性，因为 Item 实体的名称、Unit 实体的名称、LocationAtHome 实体的 storedIn 以及 LocationAtShop 实体的 aisle 都恰好符合这一要求。而在其他应用程序中，电子邮件地址或电话号码或许更加合适。某些情况下，开发者也许要在源数据和目标数据中添加“唯一性 ID”（uniqueness ID），以判断数据是否重复。

CoreDataImporter 实例需要依赖 NSDictionary 对象，这个 NSDictionary 里面的每个键都是某一实体的名称，而对应的值则是我们所选定的属性，该属性用来判断是否有重复数据。于是，我们需要新建名为 entitiesWithUniqueAttributes 的字典，以保存这些判定重复数据所用的属性。为了能够根据给定的实体查出与之相对应的属性名

称, 我们还需实现 `uniqueAttributeForEntity` 方法。程序清单 8-8 列出了 `CoreData-Importer` 头文件里的这些“方法头”(method header)。

程序清单8-8 CoreDataImporter.h文件

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
@interface CoreDataImporter : NSObject
@property (nonatomic, retain) NSDictionary *entitiesWithUniqueAttributes;

+ (void)saveContext:(NSManagedObjectContext*)context;
- (CoreDataImporter*)initWithUniqueAttributes:(NSDictionary*)uniqueAttributes;
- (NSString*)uniqueAttributeForEntity:(NSString*)entity;

- (NSManagedObject*)insertUniqueObjectInTargetEntity:(NSString*)entity
    uniqueAttributeValue:(NSString*)uniqueAttributeValue
    attributeValues:(NSDictionary*)attributeValues
    inContext:(NSManagedObjectContext*)context;

- (NSManagedObject*)insertBasicObjectInTargetEntity:(NSString*)entity
    targetEntityAttribute:(NSString*)targetEntityAttribute
    sourceXMLAttribute:(NSString*)sourceXMLAttribute
    attributeDict:(NSDictionary*)attributeDict
    context:(NSManagedObjectContext*)context;

@end
```

请按下列步骤修改 Grocery Dude, 以创建 `CoreDataImporter` 类:

1. 选中名为 **Generic Core Data Classes** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**, 然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 `NSObject`, 把 **Class** 名称设为 `CoreDataImporter`。点击 **Next** 按钮。
5. 确保 **Targets** 中的 **Grocery Dude** 处于勾选状态, 然后点击 **Create** 按钮, 在 **Grocery Dude** 项目的文件夹下创建类。

6. 用程序清单 8-8 将 `CoreDataImporter.h` 文件中的所有代码全都替换掉。

创建 `CoreDataImporter` 实例的时候, 应该用 `initWithUniqueAttributes` 来初始化它。这就使得 `CoreDataImporter` 能够拥有一个 `NSDictionary`, 而这个 `NSDictionary` 中保存了每个目标实体与其 `unique` 属性名称之间的映射关系。正如前面所说, 我们还需要编写 `uniqueAttributeForEntity` 方法, 以便根据给定的实体查出与之对应的 `unique` 属性。程序清单 8-9 列出了相关代码。

程序清单8-9 CoreDataImporter.m文件的saveContext、initWithUniqueAttributes及
uniqueAttributeForEntity方法

```

#import "CoreDataImporter.h"
@implementation CoreDataImporter
#define debug 1
+ (void)saveContext:(NSManagedObjectContext*)context {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}
[context performBlockAndWait:^(
    if ([context hasChanges]) {
        NSError *error = nil;
        if ([context save:&error]) {NSLog(
@"CoreDataImporter SAVED changes from context to persistent store";
        } else {NSLog(
@"CoreDataImporter FAILED to save changes from context to persistent store: %@",
error);
        }
    } else {NSLog(
@"CoreDataImporter SKIPPED saving context as there are no changes";
    }
)];
}
- (CoreDataImporter*)initWithUniqueAttributes:(NSDictionary*)uniqueAttributes {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}
if (self = [super init]) {

    self.entitiesWithUniqueAttributes = uniqueAttributes;

    if (self.entitiesWithUniqueAttributes) {
        return self;
    } else {NSLog(
@"FAILED to initialize CoreDataImporter: entitiesWithUniqueAttributes is nil");
        return nil;
    }
}
return nil;
}
- (NSString*)uniqueAttributeForEntity:(NSString*)entity {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}
return [self.entitiesWithUniqueAttributes valueForKey:entity];
}
@end

```

请注意，CoreDataImporter 里面也包含了一个 saveContext 方法，它和 CoreData-

Helper 里的同名方法相仿。这样做使得 CoreDataImporter 更容易移植到其他应用程序里。

请按下列步骤修改 Grocery Dude，以实现 CoreDataImporter 类中的前几个方法：

用程序清单 8-9 把 CoreDataImporter.m 文件里原有的代码全都替换掉。修改之后，Xcode 仍然会有警告信息，提示 CoreDataImporter 类实现得不够完整。

插入托管对象之前，首先要保证待插入的这个对象没有出现在目标上下文之中。为了完成这项检测，我们需要在目标上下文上面执行 NSFetchRequest，这个 NSFetchRequest 里面所含的谓词 (NSPredicate) 是针对 unique 属性的名称及其取值来编写的。程序清单 8-10 列出了相关代码。

程序清单8-10 CoreDataImporter.m文件中的existingObjectInContext方法

```

- (NSManagedObject*)existingObjectInContext:(NSManagedObjectContext*)context
    forEntity:(NSString*)entity
    withUniqueAttributeValue:(NSString*)uniqueAttributeValue {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    NSString *uniqueAttribute = [self uniqueAttributeForEntity:entity];
    NSPredicate *predicate =
        [NSPredicate predicateWithFormat:@"%K==%@",
         uniqueAttribute, uniqueAttributeValue];

    NSFetchRequest *fetchRequest =
        [NSFetchRequest fetchRequestWithEntityName:entity];
    [fetchRequest setPredicate:predicate];
    [fetchRequest setFetchLimit:1];
    NSError *error;
    NSArray *fetchRequestResults =
        [context executeFetchRequest:fetchRequest error:&error];
    if (error) {NSLog(@"Error: %@", error.localizedDescription);}
    if (fetchRequestResults.count == 0) {return nil;}
    return fetchRequestResults.lastObject;
}

```

请按下列步骤修改 Grocery Dude，以实现 existingObjectInContext 方法：

1. 把程序清单 8-10 中的代码添加到 CoreDataImporter.m 文件末尾的 @end 上方。修改之后，Xcode 中仍然会有警告信息，提示 CoreDataImporter 类实现得不完整。

如果 existingObjectInContext 方法返回 nil，那就表示待插入的对象不在目标上下文里面。这意味着我们需要在目标上下文里面插入新的对象，该对象具备相应的 unique 属性值。这个用来插入对象的新方法叫做 insertUniqueObjectInTargetEntity，其代码列在程序清单 8-11 中。

程序清单8-11 CoreDataImporter.m文件中的insertUniqueObjectInTargetEntity方法

```

- (NSManagedObject*)insertUniqueObjectInTargetEntity:(NSString*)entity
    uniqueAttributeValue:(NSString*)uniqueAttributeValue
    attributeValues:(NSDictionary*)attributeValues
    inContext:(NSManagedObjectContext*)context {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }
    NSString *uniqueAttribute = [self uniqueAttributeForEntity:entity];
    if (uniqueAttributeValue.length > 0) {
        NSManagedObject *existingObject =
            [self existingObjectInContext:context
                forEntity:entity
                withUniqueAttributeValue:uniqueAttributeValue];
        if (existingObject) {
            NSLog(@"%@ object with %@ value '%%' already exists",
                entity, uniqueAttribute, uniqueAttributeValue);
            return existingObject;
        } else {
            NSManagedObject *newObject =
                [NSEntityDescription insertNewObjectForEntityForName:entity
                    inManagedObjectContext:context];
            [newObject setValuesForKeysWithDictionary:attributeValues];
            NSLog(@"Created %@ object with %@", @"%@",
                entity, uniqueAttribute, uniqueAttributeValue);
            return newObject;
        }
    } else {
        NSLog(@"Skipped %@ object creation: unique attribute value is 0 length",
            entity);
    }
    return nil;
}

```

insertUniqueObjectInTargetEntity方法会返回NSManagedObject对象，该对象的各项属性是根据attributeValues参数来填充的，而这个参数是个NSDictionary，其中包含了待填充的各项属性及其取值。假如没有指定unique属性的值，那么该方法就返回nil。请按下列步骤修改Grocery Dude，以实现insertUniqueObjectInTargetEntity方法：

把程序清单8-11中的代码添加到CoreDataImporter.m文件底部的@end上方。修改之后，Xcode依然会发出警告，提示CoreDataImporter类实现得不够完整。

在CoreDataImporter类中，最后一个要实现的方法是insertBasicObjectInTargetEntity，它会利用insertUniqueObjectInTargetEntity来完成自己的任务。该方法的目标是插入一个基本的NSManagedObject托管对象，该对象中只会有一项属性，那就是unique属性。笔者之所以要编写这个方法，只是为了把最终导入默认数据所

用的那部分代码写得更加易读一些。该方法的调用者需要提供目标实体、目标实体的属性、源 XML 文件中与该属性等价的那个属性、由 XMLParser 的委托方法所传来的那个完整的 attributeDict 以及上下文。假如除了 unique 属性之外还有其他属性需要添加,那么开发者可以在该方法所返回的托管对象上面添加那些属性。程序清单 8-12 列出了相关代码。

程序清单8-12 CoreDataImporter.m文件的insertBasicObjectInTargetEntity方法

```
- (NSManagedObject*)insertBasicObjectInTargetEntity:(NSString*)entity
    targetEntityAttribute:(NSString*)targetEntityAttribute
    sourceXMLAttribute:(NSString*)sourceXMLAttribute
    attributeDict:(NSDictionary*)attributeDict
    context:(NSManagedObjectContext*)context {

    NSArray *attributes = [NSArray arrayWithObject:targetEntityAttribute];
    NSArray *values =
        [NSArray arrayWithObject:[attributeDict valueForKey:sourceXMLAttribute]];

    NSDictionary *attributeValues =
        [NSDictionary dictionaryWithObjects:values forKeys:attributes];
    return [self insertUniqueObjectInTargetEntity:entity
        uniqueAttributeValue:[attributeDict valueForKey:sourceXMLAttribute]
        attributeValues:attributeValues
        inContext:context];
}
```

请按下列步骤修改 Grocery Dude, 以实现 insertBasicObjectInTargetEntity 方法:

1. 把程序清单 8-12 中的代码添加到 CoreDataImporter.m 文件底部的 @end 上方。修改之后, 原来提示“CoreDataImporter 类尚未完全实现”的那条警告就应该消失了。



你可能注意到了, CoreDataImporter 类没有使用 NSManagedObject 的子类, 而是通过“key-value coding”(键值编码)来访问实体的属性。笔者刻意要这样做, 是因为它虽然不如 NSManagedObject 子类那样可以方便地使用“点”(.)来访问特性, 但却能适应各种数据模型。采用这种做法的时候要注意, 在指定谓词时, %K 用来表示“key path”(键路径)。程序清单 8-10 就用到了 %K。

8.8 选定各实体的 Unique 属性

使用 CoreDataImporter 导入数据之前, 必须先为每个实体选定一项 unique 属性。于是, 我们新建名为 selectedUniqueAttributes 的方法, 用来配置实体与 unique 属性之间的映射关系。该方法会返回 NSDictionary, 而选定的各项 unique 属性都放在这个 NSDictionary 中, 这个方法的代码如程序清单 8-13 所示。注意, 这段

代码应该放在 CoreDataHelper 类而不是 CoreDataImporter 类之中。假如想把 CoreDataImporter 与 CoreDataHelper 重新部署到自己的应用程序里面,并用它们导入数据,那么需要先按照自己的托管对象模型来确定各项 unique 属性。

程序清单8-13 CoreDataHelper.m文件中的selectedUniqueAttributes方法

```
#pragma mark - UNIQUE ATTRIBUTE SELECTION (This code is Grocery Dude data
specific and is used when instantiating CoreDataImporter)
- (NSDictionary*)selectedUniqueAttributes {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

NSMutableArray *entities = [NSMutableArray new];
NSMutableArray *attributes = [NSMutableArray new];

// Select an attribute in each entity for uniqueness
[entities addObject:@"Item"];[attributes addObject:@"name"];
[entities addObject:@"Unit"];[attributes addObject:@"name"];
[entities addObject:@"LocationAtHome"];[attributes addObject:@"storedIn"];
[entities addObject:@"LocationAtShop"];[attributes addObject:@"aisle"];

NSDictionary *dictionary = [NSDictionary dictionaryWithObjects:attributes
                                                                forKeys:entities];

return dictionary;
}
```

请按下列步骤修改 Grocery Dude, 以实现 UNIQUE ATTRIBUTE SELECTION 部分:

1. 把程序清单 8-13 中的代码添加到 CoreDataHelper.m 文件底部的 @end 之上。

8.9 把 XML 中的数据映射到实体的属性

NSXMLParser 对象的 parse 方法现在已经可以利用由 CoreDataImporter 所提供的数据库导入引擎 (data import engine) 了。剩下的事情就是把 NSXMLParserDelegate 协议所定义的某些 delegate 方法实现出来。我们需要新建 DELEGATE: NSXMLParser 部分,并在其中实现下面这两个方法:

- ❑ **parseErrorOccurred** 方法 该方法用于记录 XML 解析过程中所发生的错误,这些错误一般都是 NSXMLParserErrorDomain 错误。假如收到了错误信息,那可能说明 XML 文件的格式不对,或其中有无效字符。
- ❑ **didStartElement** 方法 解析器每次在 XML 文件里找到新的元素时,就会调用该方法。对于 Grocery Dude 程序所用的这份包含默认数据的 XML 文件来说,这种元素指的就是 <item>。解析器会把它在元素里找到的每一项属性及其取值都放在 NSDictionary 里面,然后把 NSDictionary 传给相应的委托方法。这个

NSDictionary 字典很适合用来创建托管对象。假如要把这种数据导入技术用于其他应用程序，那就应该在 didStartElement 方法中针对那个程序的模型来修改相关的数据导入语句。

程序清单 8-14 列出了这两个方法的代码。

程序清单8-14 CoreDataHelper.m文件的DELEGATE: NSXMLParser部分

```
#pragma mark - DELEGATE: NSXMLParser (This code is Grocery Dude data
specific)
- (void)parser:(NSXMLParser *)parser
    parseErrorOccurred:(NSError *)parseError {
if (debug==1) {
    NSLog(@"Parser Error: %@", parseError.localizedDescription);
}
}

- (void)parser:(NSXMLParser *)parser
    didStartElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName
    attributes:(NSDictionary *)attributeDict {

[self.importContext performBlockAndWait:^(

    // STEP 1: Process only the 'item' element in the XML file
    if ([elementName isEqualToString:@"item"]) {

        // STEP 2: Prepare the Core Data Importer
        CoreDataImporter *importer =
        [[CoreDataImporter alloc] initWithUniqueAttributes:
        [self selectedUniqueAttributes]];

        // STEP 3a: Insert a unique 'Item' object
        NSManagedObject *item =
        [importer insertBasicObjectInTargetEntity:@"Item"
        targetEntityAttribute:@"name"
        sourceXMLAttribute:@"name"
        attributeDict:attributeDict
        context:_importContext];

        // STEP 3b: Insert a unique 'Unit' object
        NSManagedObject *unit =
        [importer insertBasicObjectInTargetEntity:@"Unit"
        targetEntityAttribute:@"name"
        sourceXMLAttribute:@"unit"
        attributeDict:attributeDict
        context:_importContext];

        // STEP 3c: Insert a unique 'LocationAtHome' object
        NSManagedObject *locationAtHome =
```

$$\left\{ \begin{array}{l} \\ \end{array} \right\};$$

1. 把 `#import "CoreDataImporter.h"` 添加到 `CoreDataHelper.m` 顶部。
2. 把程序清单 8-14 中的代码添加到 `CoreDataHelper.m` 文件底部的 `@end` 上方。

2. 把程序清单 8-14 中的代码添加到 CoreDataHelper.m 文件底部的 @end 上方。

3. 从 iOS 仿真器中删掉 Grocery Dude 程序，使持久化存储区中不会出现值为 1 的 DefaultDataImported 键。

4. 点击 **Product > Clean** 菜单项，然后在 iOS 仿真器中运行 Grocery Dude。点击 **Import** 按钮，开始导入数据。在导入默认数据的过程中仍然可以操作应用程序。正常的运行结果应该如图 8-5 所示。



图 8-5 预先加载进来的默认数据

8.10 从持久化存储区中导入数据

假如要发布带有默认数据的应用程序，而默认数据又放在持久化存储区里，那么开发者可以考虑下列两种办法。具体选哪一种应根据用户设备中是否已有持久化存储区来确定。

- ❑ **方法 1：把包含默认数据的持久化存储区用作初始的持久化存储区。**我们只需在程序初次启动之前，先把包含默认数据的存储区复制到设备中即可。这是目前最简单而且最高效的办法。如果原来发布应用程序的时候没有包含默认数据，那么就不能采用这个办法了。因为在那种情况下，持久化存储区里可能已经有用户自己的数据了，我们不想把那些数据覆盖掉。
- ❑ **方法 2：对包含默认数据的持久化存储区执行深拷贝，将其中不重复的那部分数据复制到现有的持久化存储区里。**这需要复制每个实体的属性值及关系。这种拷贝之所以叫深 (deep) 拷贝，是因为在遍历数据的过程中，必须根据需要把各种关系及相关的对象创建出来。这种通过“查找”及“创建”操作来拷贝数据的办法需要很大的运算量，所以最好放在后台执行。这种做法是个比较复杂的话题，我们放在第 9 章中讨论。开发者应该尽量避免采用此办法。另一个方案是采用 `NSPersistentStore` 类中的 `migratePersistentStore` 实例方法来做，该方案的速度比较快，但可能会产生重复的对象。

把包含默认数据的存储区用作初始存储区

早前我们从 XML 文件中导入数据的时候，Core Data 已经创建了一份包含全部默认数据的持久化存储区。由于该存储区由 Core Data 创建，所以其格式是正确的。为了把这个默认存储区设为初始存储区，我们需要把相关的数据库文件添加到应用程序包 (application bundle) 里面。为待发布的应用程序准备这份默认数据库时，需要考虑数据库日志记录模式 (database journaling mode)。从 iOS 7 开始，SQLite 数据库会默认采用一种新的日志记录模式，叫做预写式日志记录 (Write-Ahead Logging, 简称 WAL)。这种新的默认模式会提升性能，而且也能更好地支持并发。对于每个数据库来说，该模式将会默认产生下面三个文件。

- ❑ **sqlite** 文件与往常一样，是个数据库文件。

❑ **sqlite-wal** 文件是预写日志 (**Write-Ahead Log**) 文件, 其中包含尚未提交的数据库事务 (**uncommitted database transaction**)。若是删掉此文件, 则可能会丢失数据。如果本来就没有这个文件, 那说明数据库里没有尚未处理的事务等着提交。

❑ **sqlite-shm** 文件是共享内存 (**Shared Memory**) 文件, 其中包含一份 WAL 文件的索引。系统可以自动重新生成该文件, 开发者无须担心它。

图 8-6 演示了 Grocery Dude 项目 SQLite 数据库的 WAL 与 SHM 文件。

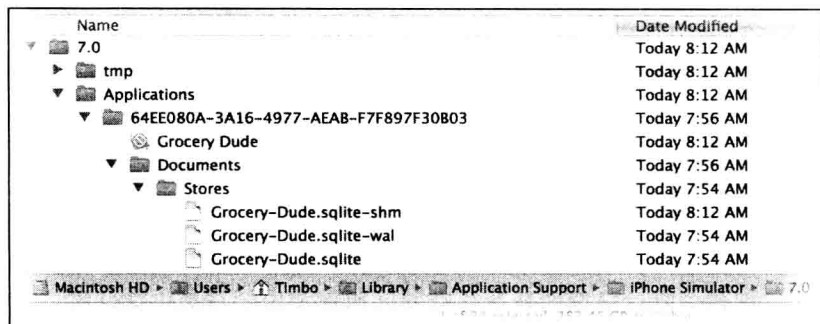


图 8-6 “WAL 日志记录模式”所产生的 SQLite 数据库文件

我们在第 2 章刻意把 `journal_mode` 设为 `DELETE`, 以确保不会出现 `-wal` 及 `-shm` 文件。这样做使得开发者可以把新生成的持久化存储区文件直接拷贝出来, 而不用担心那两个附加文件。

请按下列步骤提取 Grocery Dude 的持久化存储区文件:

1. 右击 **Finder**, 选择 **Go to Folder...**。
2. 在 **Go to the Folder** 对话框中输入 `/Users/Tim/Library/Application Support/iPhone Simulator/`。(注意: 其中的“Tim”要替换成你的用户名。)

3. 按照图 8-7 所示的文件夹路径寻找 `Grocery-Dude.sqlite` 文件。应用程序的 GUID 可能和图中不同。假如模拟器里安装了很多应用程序, 而又不容易确定哪一个是 Grocery Dude 程序的 GUID, 那么请在控制台的日志中搜寻 `Grocery-Dude.sqlite`, 查看该文件所在的路径。

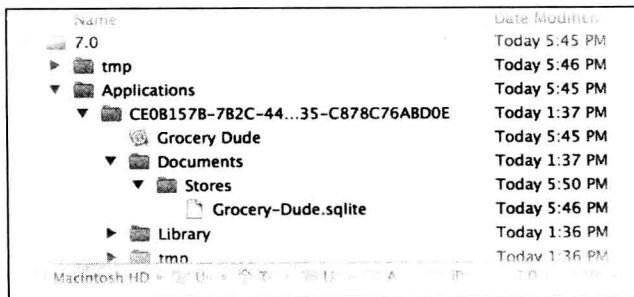


图 8-7 “DELETE 日志记录模式”所生成的 SQLite 数据库文件

4. 确保 Grocery Dude 程序没有处在运行状态。
 5. 把 Grocery-Dude.sqlite 复制到桌面，将其重命名为 DefaultData.sqlite。
- 图 8-7 演示了持久化存储区在 iOS 仿真器的应用程序沙箱之中的位置。



提示 如果找不到 Grocery-Dude.sqlite 文件，可以从 <http://www.timroadley.com/LearningCoreData/DefaultData.sqlite.zip> 下载一份拷贝。

有了 DefaultData.sqlite 这个存储区文件之后，可以将其放在应用程序包中，然后重新启用 WAL 模式。

请按下列步骤修改 Grocery Dude：

1. 在 CoreDataHelper.m 文件的 loadStore 方法中，把 NSSQLitePragmasOption 选项注释掉，以重新启用 WAL 日志记录模式。
2. 把 DefaultData.sqlite 文件从桌面拖放到 Xcode 的 **Data Model** 组里。请确认 **Copy items into destination group's folder** 选项及 Targets 中的 “Grocery Dude” 处于勾选状态，然后点击 **Finish** 按钮。

把 DefaultData.sqlite 文件放到应用程序包里面之后，就可以实现名为 setDefaultDataStoreAsInitialStore 的新方法了。该方法首先会由 setupCoreData 来调用，如果包含默认数据的存储区文件没有放在合适的位置上，那么该方法会令此文件就位，以供稍后使用。想把这个默认的存储区文件移动到合适的位置上有两种办法。一种办法是使用 NSPersistentStoreCoordinator 的 migratePersistentStore 方法。值得注意的是，此方法可以“透明地”处理存储区文件的类型转换问题。另一种办法则是采用 NSFileManager 的 copyItemAtURL 方法，我们的 Grocery Dude 项目就打算使用这种比较基础的做法。相关代码如程序清单 8-15 所示。

程序清单8-15 CoreDataHelper.m文件的setDefaultDataStoreAsInitialStore方法

```
- (void)setDefaultDataStoreAsInitialStore {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }
    -NSFileManager *fileManager = [NSFileManager defaultManager];
    if (![fileManager fileExistsAtPath:self.storeURL.path]) {
        NSURL *defaultDataURL =
            [NSURL URLWithString:[NSBundle mainBundle]
                pathForResource:@"DefaultData" ofType:@"sqlite"];
        NSError *error;
        if (![fileManager copyItemAtURL:defaultDataURL
            toURL:self.storeURL
            error:&error]) {
            NSLog(@"DefaultData.sqlite copy FAIL: %@",
                error.localizedDescription);
        }
        else {

```

```

NSLog(@"A copy of DefaultData.sqlite was set as the initial store for %@",
self.storeURL.path);
    }
}
}
}

```

setDefaultDataStoreAsInitialStore 方法首先判断目标位置是否已经有持久化存储区文件了。如果没有，那就调用 copyItemAtURL 方法，把包含默认数据的持久化存储区文件从主程序包中拷贝过去。

请按下列步骤修改 Grocery Dude，以实现 setDefaultDataStoreAsInitialStore 方法：

1. 把程序清单 8-15 中的代码添加到 CoreDataHelper.m 文件 DATA_IMPORT 部分的底部。
2. 修改 CoreDataHelper.m 文件的 setupCoreData 方法，把 [self setDefaultDataStoreAsInitialStore]; 添加到 [self loadStore]; 这一行代码的上方。
3. 从设备或模拟器中删掉 Grocery Dude 程序。
4. 点击 **Product > Clean**，清除残留的缓存。
5. 运行应用程序，它应该会把预先生成的默认数据加载进来。运行结果如图 8-8 所示。

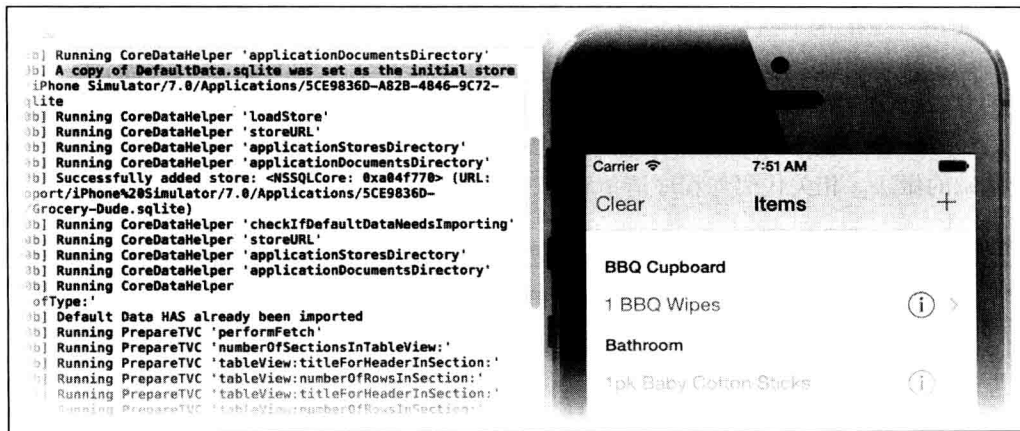


图 8-8 预先加载进来的默认数据

8.11 小结

本章演示了如何配置多个托管对象上下文，使数据导入操作能够于后台完成。在实现 CoreDataImporter 的基本功能时，笔者给出了一些技巧，告诉大家怎样把原始数据方便地制作成 XML 文件。这个新建的 CoreDataImporter 类适用于各种模型，它能够根

据一份 XML 文件在目标上下文中创建出互不重复的托管对象。开发者要注意的是，待引入的 XML 文件应该尽量短小一些，因为在导入数据时，整份文件都必须放在内存中。如果只是想用 iOS 仿真器运行应用程序，并根据该文件生成包含默认数据的存储区文件，那就不必太担心这个问题了。在发布应用程序的时候，笔者推荐最好能带上一份持久化存储区文件，并在其中预先放入一些数据，这样的话，程序就可以把它当作初始的存储区来用了。假如用户的设备中已经有了另一份持久化存储区，那可能就需要对包含预设数据的这个持久化存储区执行深拷贝或数据迁移操作。这个话题我们放在下一章讲。

8.12 习题

请根据所学内容完成下列试验。

1. 修改 DefaultData.XML 文件，为其中某些货品添加名为 **quantity** 的属性，并将其值设为随机数。

2. 修改 CoreDataHelper.m 文件的 setupCoreData 方法，把 [self setDefault-DataStoreAsInitialStore]; 注释掉，以便重新从 XML 文件中导入数据。修改 CoreDataHelper.m 文件的 parser:didStartElement 方法，将程序清单 8-16 中的代码添加到 **STEP 4** 底部。删除应用程序，然后重新运行，以触发数据导入操作。添加到 DefaultData.XML 文件中的 quantity 值应该会出现导入的对象中。

3. 把 WAL 日志记录模式关掉，并重新运行应用程序。然后把程序停止，用 **SQLite Database Browser** 打开持久化存储区，并修改某件货品的名称。再度运行应用程序，这次应该能在 iOS 仿真器中看到修改后的效果。请注意，不要令 SQLite Database Browser 与 iOS 仿真器同时打开应用程序的 SQLite 文件。

程序清单8-16 CoreDataHelper.m文件中的didStartElement方法 (STEP 4)

```
NSNumberFormatter *f = [NSNumberFormatter new];
if ([attributeDict valueForKey:@"quantity"]) {
    [item setValue:[f numberFromString:[attributeDict valueForKey:@"quantity"]]
        forKey:@"quantity"];
}
```

在学习下一章之前，请把试验过程中所做的修改全部复原。

深拷贝

若要绝对不犯错，那只有一个办法，就是别去想新的主意。

——阿尔伯特·爱因斯坦

第8章演示了如何从XML文件中导入默认数据。只有当XML源文件非常小且完全可以装入内存时，我们才能使用这种导入办法。另外一个办法是把预先填充好的持久化存储区当成应用程序的初始持久化存储区。假如用户的设备中已经有了一些数据，而开发者又想向其中再添加一些的话，那这个办法也用不成了。这种情况下，我们可能要对托管对象执行深拷贝，将其从源持久化存储区复制到现有的持久化存储区里。该方案的速度虽然不如NSPersistentStore类的migratePersistentStore实例方法快，但是却能提供去除重复数据的功能，而且粒度也更为精细。

9.1 概述

在深拷贝的过程中，托管对象及其关系会从一个持久化存储区复制到另一个里面。拷贝对象时，还需沿着该源对象的各项关系，找到与之有关的其他源对象，而那些源对象也得按照需要复制到目标存储区里。在目标存储区中，拷贝过来的这些对象之间的关系与其在源存储区中的关系是相同的。这个拷贝过程要运用到每一个对象上面，最终使得所有对象及其关系都复制到目标持久化存储区中。毫无疑问，这是个运算量非常大的任务，所以只应该放在后台执行。

应用程序的数据模型各有不同，有些情况下，先把所有对象拷贝过去，稍后再重建关系这样做可能会更有效率一些。实际上，这就是首选的方案。但不巧的是，它不适用于

Grocery Dude 程序。原因在于，Prepare 选项卡和 Shop 选项卡上面列出的货品都要依赖于相关的对象。假如还没建立好它们与 HomeLocation 或 ShopLocation 之间的关系就直接将其导入，那么在导入数据的过程中，表格视图就没办法把这些货品显示到对应的部分中。从用户的角度来看，这么做是不对的，他们会以为应用程序出了 bug。

只有当源存储区和目标存储区都采用相同的托管对象模型时，才能够执行深拷贝。即便符合这一要求，我们也需要为源存储区和目标存储区分别准备 coordinator 及上下文。源存储区和目标存储区所用的上下文还应该同主队列中的上下文区分开。图 9-1 从宏观角度概述了执行深拷贝所需的 Core Data 组件。

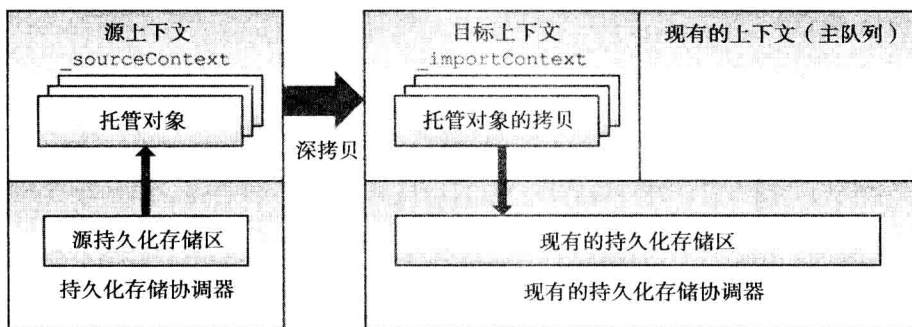


图 9-1 深拷贝

在演示深拷贝时，我们将把目前的 `_importContext` 复用为目标上下文。把对象从一个上下文复制到另一个上下文之中并不像执行复制和粘贴操作那样简单。所谓复制对象，实际上是在目标上下文里创建新的对象，然后把源对象的全部属性值都复制到这个新的对象中。复制好对象之后，就该处理关系了，而复制关系时，则不能采用与对象相同的办法来做。假如我们真的像复制属性值那样把关系复制过去，那么拷贝过去的对象（即副本对象）与源存储区里的对象之间就形成了非法的跨存储区关系（illegal cross-store relationship）。为了在深拷贝的过程中正确地复制关系，我们必须于目标上下文里辨识出相互关联的副本对象，并据此在这些副本对象之间建立关系。图 9-2 演示了“一对一关系”的拷贝过程。

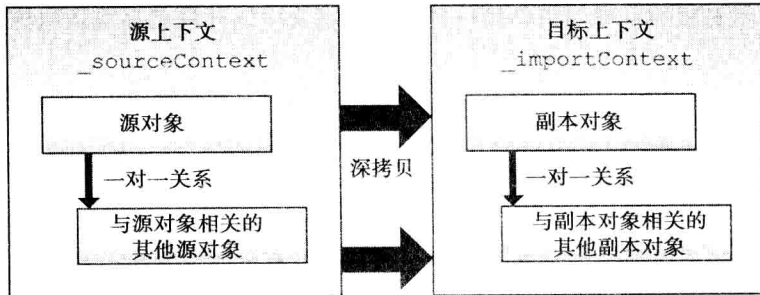


图 9-2 拷贝“一对一关系”

拷贝关系之前，必须先检查该关系所涉及的对象在不在目标上下文里面。假如目标上下文里面还没有相关对象，那就参照源对象来创建等效对象。

一对多关系的拷贝则比较困难，深拷贝之所以是个运算量极大的任务，这也是其中一大原因。由于深拷贝必须逐个遍历每个对象及其全部关系，所以整个过程可能非常耗时。另外还要考虑的是，待拷贝的关系是有序的（ordered）还是无序的（unordered）。假如是有序关系，那我们在底层就要用 `NSMutableOrderedSet` 来存放相关对象；若是无序关系，则要用 `NSMutableSet` 来存放相关对象。编写深拷贝的代码时，必须反映出这一区别。图 9-3 演示了这种更为复杂的关系拷贝操作。

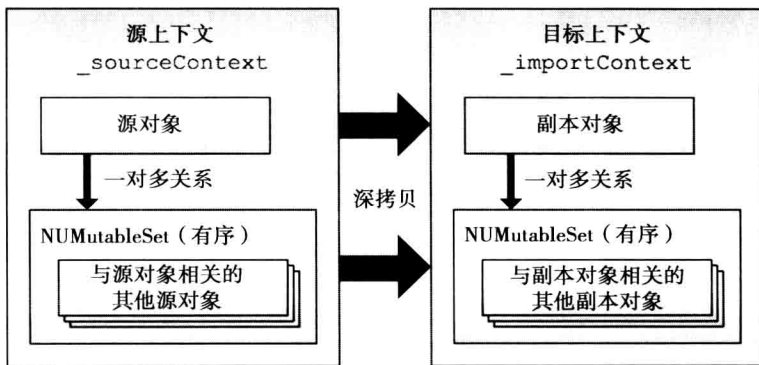


图 9-3 拷贝“一对多关系”

对象的深拷贝要用到前面章节所讲的若干方法。我们要预先为每个实体指定一个 `unique` 属性，而深拷贝的过程将会依赖这个属性的名称。这样做能够在拷贝对象之前先进行“唯一性检测”（uniqueness check），防止产生重复数据。深拷贝的执行过程可以宏观地描述成下面这几个步骤：

- ❑ 创建 `CoreDataImporter` 实例，创建时要提供 `NSDictionary`，其中存有从实体名称到 `unique` 属性的映射，每个实体都要有这样一条映射。
- ❑ 给 `CoreDataImporter` 实例传入 `NSArray`，这个数组中包含了待复制的各实体名称，而 `CoreDataImporter` 实例则会依次遍历这些名称，并根据需要，把每个实体所对应的全部对象都拷贝到目标上下文中。开发者只需指定待拷贝的实体名称，与指明的实体相关联的所有对象都将拷贝过去。
- ❑ 假如目标上下文中没有与源上下文相等价的对象，那就新建托管对象，并将其插入目标上下文。新对象的各属性值将根据源对象来设置。
- ❑ 如果源对象有关系，那就沿着该关系找到相关的对象。然后根据需要，把那些相关的对象拷贝到目标上下文里。
- ❑ 如果某条关系所涉及的全部对象都已拷贝到目标上下文之中，那就在副本对象和与之相关的其他副本对象之间重建关系，如图 9-2 及图 9-3 所示。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChar08.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

笔者要把深拷贝的过程当作另一种导入默认数据的办法演示给大家，而不想把它当成一种向现有的持久化存储区中添加数据的办法。目前的 Grocery Dude 程序会在启动时判断是否已有持久化存储区。如果没有，那它就把包含默认数据的持久化存储区配置成初始的存储区。为了演示深拷贝，我们现在要禁止这一行为。

请按下列步骤修改 Grocery Dude 项目，以防止应用程序把包含默认数据的存储区当成初始的存储区来用：

1. 修改 CoreDataHelper.m 文件中的 setupCoreData 方法，把 [self setDefaultDataStoreAsInitialStore]; 这条语句注释掉。
2. 从你所使用的设备或 iOS 仿真器中删除 Grocery Dude 程序。
3. 点击 Xcode 的 **Product > Clean** 菜单项。

9.2 配置拷贝源数据所用的 Core Data 栈

Core Data 栈 (Core Data stack) 这个术语是对持久化存储区、持久化存储协调器、托管对象模型以及托管对象上下文的合称。为了对源存储区实施深拷贝，我们需要再使用一套与目标存储区不同的 Core Data 栈。这样做的效果就相当于把深拷贝操作所涉及的源上下文和目标上下文区隔开。两个栈之间唯一的共性在于，它们都使用同样的托管对象模型。程序清单 9-1 列出了 CoreDataHelper 头文件中的新代码，这些代码是为了使用新的 Core Data 栈而编写的。

程序清单9-1 CoreDataHelper.h文件

```
@property (nonatomic, readonly) NSManagedObjectContext *sourceContext;
@property (nonatomic, readonly) NSPersistentStoreCoordinator
*sourceCoordinator;
@property (nonatomic, readonly) NSPersistentStore *sourceStore;
```

请按下列步骤修改 Grocery Dude 项目，以新建拷贝源数据所用的 Core Data 栈：

1. 把程序清单 9-1 中的代码添加到 CoreDataHelper.h 文件里，将其放在 store 的声明语句之后。

接下来应该用这些新特性初始化新的 Core Data 栈，以供拷贝源数据所用。初始化 _sourceContext 所用的方式与初始化 _importContext 的相同，而配置 _source-

Coordinator 所用的模型也与现有的 `_coordinator` 相同。程序清单 9-2 列出了相关的代码。

程序清单9-2 CoreDataHelper.m文件中的init方法

```

_sourceCoordinator =
[[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:_model];
_sourceContext =
[[NSManagedObjectContext alloc]
initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_sourceContext performBlockAndWait:^(
    [_sourceContext setPersistentStoreCoordinator:_sourceCoordinator];
    [_sourceContext setUndoManager:nil]; // the default on iOS
)];

```

请按下列步骤修改 Grocery Dude 项目，为其添加对 Source 栈的支持：

1. 修改 CoreDataHelper.m 文件的 `init` 方法，把程序清单 9-2 中的代码添加到该方法底部的 `return self;` 语句上方。

配置存放源数据的存储区

配置源存储区所用的办法与配置现有的存储区的相同。也就是说，需要指定源存储区的文件名，并且要新编一个方法，用以返回源存储区的 `NSURL`。这两者都将供新建的 `loadSourceStore` 方法所用，该方法最终负责在执行深拷贝之前把源存储区加载好。程序清单 9-3 里的这行新代码需要添加到 `FILES` 部分之中，它指定了源存储区的文件名。演示深拷贝时，我们将把目前的 `DefaultData.sqlite` 文件复用为源存储区文件。

程序清单9-3 CoreDataHelper.m文件的FILES部分

```

NSString *sourceStoreFilename = @"DefaultData.sqlite";

```

请按下列步骤修改 Grocery Dude 项目，以便配置表示源存储区的文件名：

1. 修改 CoreDataHelper.m 文件顶部的 `FILES` 部分，把程序清单 9-3 里的代码添加到该部分的底端。

现有的 `storeURL` 方法会返回现有存储区的 `URL`，而与之相仿，我们将要新建的 `sourceStoreURL` 方法则会返回源存储区的 `URL`。程序清单 9-4 列出了相关代码。

程序清单9-4 CoreDataHelper.m文件的PATHS部分

```

- (NSURL *)sourceStoreURL {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}
}

```

```

return [NSURL fileURLWithPath:[NSBundle mainBundle]
        pathForResource:[sourceStoreFilename stringByDeletingPathExtension]
        ofType:[sourceStoreFilename pathExtension]];
}

```

请按下列步骤修改 Grocery Dude，以编写 sourceStoreURL 方法：

1. 修改 CoreDataHelper.m 文件的 PATHS 部分，把程序清单 9-4 中的代码添加到该部分的底端。

接下来需要实现 loadSourceStore 方法。该方法负责把源存储区添加到处理源数据所用的持久化存储协调器（source coordinator）之中。由于源存储区位于“应用程序包”里，所以必须用“只读”方式加载它。请注意，这同时也意味着如果将来要升级模型，那么开发者必须提前把 DefaultData.sqlite 升级到新版本，并把升级后的版本随应用程序一同发布。程序清单 9-5 列出了相关代码，这些代码与 loadStore 方法里现有的代码相仿。

程序清单9-5 CoreDataHelper.m文件中的loadSourceStore方法

```

- (void)loadSourceStore {
if (debug==1) {
    NSLog(@"Running %@", @"'", self.class, NSStringFromSelector(_cmd));
}

if (_sourceStore) {return;} // Don't load source store if it's already loaded

NSMutableDictionary *options =
@{
    NSReadOnlyPersistentStoreOption:@YES
};
NSError *error = nil;
_sourceStore =
[_sourceCoordinator addPersistentStoreWithType:NSSQLiteStoreType
                                     configuration:nil
                                     URL:[self sourceStoreURL]
                                     options:options
                                     error:&error];

if (!_sourceStore) {
    NSLog(@"Failed to add source store. Error: %@",
          error);abort();
} else {
    NSLog(@"Successfully added source store: %@", _sourceStore);
}
}
}

```

请按下列步骤修改 Grocery Dude，以补充完整 Core Data 栈的初始化代码：

1. 修改 CoreDataHelper.m 文件的 SETUP 部分，把程序清单 9-5 中的 loadSourceStore 方法放在现有的 loadStore 方法下面。

假如现在运行应用程序，那么它仍然会从 XML 文件中导入默认数据。我们需要把“从

XML 中导入数据”这一行为改成“从持久化存储区中导入数据”。在执行这次修改之前，必须先更新 CoreDataImporter 类的代码，以便支持对持久化存储区的深拷贝操作。

9.3 增强 CoreDataImporter 类

为了支持深拷贝，我们必须增强 CoreDataImporter 类的功能，使其可以完成“拷贝托管对象”这种比较复杂的流程。有八个新的方法需要实现，它们的代码从几行到二十几行不等。由于要拷贝关系，而且还要沿着关系去寻找相关对象，所以这次修改会比较复杂。我们必须支持三种类型的关系，也就是一对一关系、一对多关系以及有序的一对多关系。鉴于整个流程比较复杂，所以笔者将其分解成数个部分，使大家理解起来更为容易。正是由于整个流程拆解成了好几个部分，所以才需要编写多达八个方法来实现深拷贝。

9.3.1 objectInfo 方法

在待实现的八个方法中，首先要编写的这个 objectInfo 方法是最简单的。该方法有助于减少重复代码，假如不编写该方法，那么在实现深拷贝功能所需的其他方法里基本上就会出现重复的代码。开发者把托管对象传给该方法之后，就会得到一个包含对象信息的 NSString，其中含有对象的实体名称、unique 属性以及 unique 属性值。程序清单 9-6 列出了相关代码。

程序清单9-6 CoreDataImporter.m文件中的objectInfo方法

```
#pragma mark - DEEP COPY
- (NSString*)objectInfo:(NSManagedObject*)object {

    if (!object) {return nil;}
    NSString *entity = object.entity.name;
    NSString *uniqueAttribute = [self uniqueAttributeForEntity:entity];
    NSString *uniqueAttributeValue = [object valueForKey:uniqueAttribute];

    return [NSString stringWithFormat:@"%s@ '%s'",
        entity, uniqueAttributeValue];
}
```

请按下列步骤修改 Grocery Dude，以实现 objectInfo 方法：

1. 把程序清单 9-6 中的代码添加到 CoreDataImporter.m 文件底端的 @end 语句上方。

9.3.2 arrayForEntity 方法

接下来要实现的是 arrayForEntity 方法。该方法会根据给定的实体、上下文及谓词返回含有托管对象的数组。它也是个辅助方法，可以减少实现深拷贝所需的代码量。程

序清单 9-7 列出了相关代码，读者应该比较熟悉这些代码了，因为它们与早前章节中的代码相似。

程序清单9-7 CoreDataImporter.m文件中的arrayForEntity方法

```

- (NSArray*)arrayForEntity:(NSString*)entity
    inContext:(NSManagedObjectContext*)context
    withPredicate:(NSPredicate*)predicate {
if (debug==1) {
    NSLog(@"Running %@", '%@'", self.class, NSStringFromSelector(_cmd));
}
    NSFetchRequest *request =
        [NSFetchRequest fetchRequestWithEntityName:entity];
    [request setFetchBatchSize:50];
    [request setPredicate:predicate];
    NSError *error;
    NSArray *array = [context executeFetchRequest:request error:&error];
    if (error) {
        NSLog(@"ERROR fetching objects: %@", error.localizedDescription);
    }
    return array;
}

```

请按下列步骤修改 Grocery Dude，以实现 arrayForEntity 方法：

1. 把程序清单 9-7 中的代码添加到 CoreDataImporter.m 文件底部的 @end 语句上方。

9.3.3 copyUniqueObject 方法

接下来实现 copyUniqueObject 方法。该方法负责把对象拷贝到给定的上下文之中，并且确保每个对象只拷贝一次。假如 object 或 targetContext 参数是 nil，那么该方法就返回 nil。从技术角度看，copyUniqueObject 方法其实并没有真的拷贝托管对象，它只是在目标上下文中新建了一个对象，并把源对象的各项属性值拷贝到这个新对象而已。正如前面章节所说，我们要用 insertUniqueObjectInTargetEntity 方法来确保每个对象只插入一次。假如待插入的对象已经在 targetContext 里面了，那么此方法就直接将这个对象返回。请注意，对象间的关系并不在 copyUniqueObject 方法里拷贝，因为我们要采用另一种方式来拷贝它们。程序清单 9-8 列出了 copyUniqueObject 方法的代码。

程序清单9-8 CoreDataImporter.m文件中的copyUniqueObject方法

```

- (NSManagedObject*)copyUniqueObject:(NSManagedObject*)object
    toContext:(NSManagedObjectContext*)targetContext {
if (debug==1) {
    NSLog(@"Running %@", '%@'", self.class, NSStringFromSelector(_cmd));
}
    // SKIP copying object with missing info

```

```

if (!object || !targetContext) {
    NSLog(@"FAILED to copy %@ to context %@",
        [self objectInfo:object], targetContext);
    return nil;
}

// PREPARE variables
NSString *entity = object.entity.name;
NSString *uniqueAttribute = [self uniqueAttributeForEntity:entity];
NSString *uniqueAttributeValue = [object valueForKey:uniqueAttribute];

if (uniqueAttributeValue.length > 0) {

    // PREPARE attributes to copy
    NSMutableDictionary *attributeValuesToCopy =
        [NSMutableDictionary new];
    for (NSString *attribute in object.entity.attributesByName) {
        [attributeValuesToCopy setValue:[object valueForKey:attribute] copy]
            forKey:attribute];
    }

    // COPY object
    NSManagedObject *copiedObject =
        [self insertUniqueObjectInTargetEntity:entity
            uniqueAttributeValue:uniqueAttributeValue
            attributeValues:attributeValuesToCopy
            inContext:targetContext];

    return copiedObject;
}
return nil;
}

```

请按下列步骤修改 Grocery Dude，以实现 copyUniqueObject 方法：

1. 把程序清单 9-8 中的代码添加到 CoreDataImporter.m 文件底部的 @end 语句上方。

9.3.4 建立一对一关系

接下来要实现 establishToOneRelationship 方法。该方法会根据关系的名称来创建由一个对象指向另一个对象的一对一关系。方法中的大部分代码都用于验证待创建的关系是否合法。假如碰到下列三种情况之一，那么该方法就不创建关系：

- ❑ 给定的源对象、目标对象或关系名称为 nil。
- ❑ 待创建的关系已经存在。
- ❑ 该关系所要关联的那个对象其实体类型与关系所要求的类型不符。

创建一对一关系只需一行代码。这行代码用于设定对象上表示关系的那个键值对的值。其中，键是关系的名称，而值则是关系所关联的对象。

establishToOneRelationship 方法的最后一部分要执行相当重要的清理工作，也就是要从每个上下文里面把指向特定对象的引用移除。保存好上下文之后，我们在每个对象的 managedObjectContext 上面调用 refreshObject 方法，把它们转成“fault”。这样做可以把对象从内存中移走，于是也就能打破由强引用所构成的循环；假如不这么做，那么系统就会保留这些无用的对象，从而造成资源浪费。若是缺了这一步，那么所有的源数据都要载入内存，于是，“从持久化存储区中导入数据”的优势就体现不出来了，它也就和“从 XML 中导入数据”没什么区别了。虽说频繁调用 save 的开销是比较大的，但却能使内存占用量变得小一些。此外，由于整个过程在后台执行，所以不会影响用户界面。图 9-2 中曾经粗略地描述了这一概念，现在我们用程序清单 9-9 中的代码把它实现出来。

程序清单9-9 CoreDataImporter.m文件中的establishToOneRelationship方法

```

- (void)establishToOneRelationship:(NSString*)relationshipName
    fromObject:(NSManagedObject*)object
    toObject:(NSManagedObject*)relatedObject {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }
    // SKIP establishing a relationship with missing info
    if (!relationshipName || !object || !relatedObject) {
        NSLog(@"SKIPPED establishing To-One relationship '%@" between %@ and %@",
            relationshipName,
            [self objectInfo:object],
            [self objectInfo:relatedObject]);
        NSLog(@"Due to missing Info!");
        return;
    }

    // SKIP establishing an existing relationship
    NSManagedObject *existingRelatedObject =
    [object valueForKey:relationshipName];
    if (existingRelatedObject) {
        return;
    }

    // SKIP establishing a relationship to the wrong entity
    NSDictionary *relationships = [object.entity relationshipsByName];
    NSRelationshipDescription *relationship =
    [relationships objectForKey:relationshipName];
    if (![relatedObject.entity isEqual:relationship.destinationEntity]) {
        NSLog(@"%@ is the of wrong entity type to relate to %@",
            [self objectInfo:object], [self objectInfo:relatedObject]);
        return;
    }

    // ESTABLISH the relationship

```

```

[object setValue:relatedObject forKey:relationshipName];
NSLog(@"ESTABLISHED %@ relationship from %@ to %@",
relationshipName,
[self objectInfo:object],
[self objectInfo:relatedObject]);

// REMOVE the relationship from memory after it is committed to disk
[CoreDataImporter saveContext:relatedObject.managedObjectContext];
[CoreDataImporter saveContext:object.managedObjectContext];
[object.managedObjectContext refreshObject:object mergeChanges:NO];
[relatedObject.managedObjectContext refreshObject:relatedObject
mergeChanges:NO];
}

```

请按下列步骤修改 Grocery Dude，以实现 `establishToOneRelationship` 方法：

1. 把程序清单 9-9 中的代码添加到 `CoreDataImporter.m` 文件底部的 `@end` 语句上方。

9.3.5 建立一对多关系

接下来要实现 `establishToManyRelationship` 方法，该方法负责创建对象的一对多关系。传给该方法的对象应该位于深拷贝操作的目标上下文之中。传给该方法的 `NSMutableSet` 里面应该包含源上下文中的对象。在建立新关系的过程中，此方法会根据需要，在目标上下文里面创建缺失的对象。

建立一对多关系的办法是：把 `sourceSet` 中的对象添加到另一个对象的 `NSMutableSet` 里面，而那个 `NSMutableSet` 正是用来表示这一关系的。我们通过对象的键值对来访问 `NSMutableSet`。关系名称是键，`NSMutableSet` 是值。由于 `NSMutableSet` 只能包含互不相同的对象，所以不必担心无意间为某对象创建了重复的关系。图 9-3 中曾经粗略地描述了这一概念，我们现在用程序清单 9-10 中的代码将其实现出来。

程序清单9-10 CoreDataImporter.m文件中的establishToManyRelationship方法

```

- (void)establishToManyRelationship:(NSString*)relationshipName
fromObject:(NSManagedObject*)object
withSourceSet:(NSMutableSet*)sourceSet {

    if (!object || !sourceSet || !relationshipName) {
        NSLog(@"SKIPPED establishing a To-Many relationship from %@",
        [self objectInfo:object]);
        NSLog(@"Due to missing Info!");
        return;
    }

    NSMutableSet *copiedSet =
    [object mutableSetValueForKey:relationshipName];

```

```

for (NSManagedObject *relatedObject in sourceSet) {

    NSManagedObject *copiedRelatedObject =
        [self copyUniqueObject:relatedObject
            toContext:object.managedObjectContext];

    if (copiedRelatedObject) {
        [copiedSet addObject:copiedRelatedObject];
        NSLog(@"A copy of %@ is now related via To-Many '%@' relationship to %@",
            [self objectInfo:object],
            relationshipName,
            [self objectInfo:copiedRelatedObject]);
    }
}

// REMOVE the relationship from memory after it is committed to disk
[CoreDataImporter saveContext:object.managedObjectContext];
[object.managedObjectContext refreshObject:object mergeChanges:NO];
}

```

请按下列步骤修改 Grocery Dude，以实现 establishToManyRelationship 方法：

1. 把程序清单 9-10 中的代码添加到 CoreDataImporter.m 文件底部的 @end 语句上方。

9.3.6 建立有序的一对多关系

接下来要实现的方法是 establishOrderedToManyRelationship，它负责为对象创建有序的一对多关系。传给该方法的对象应该位于深拷贝操作的目标上下文中。传给该方法的 NSMutableOrderedSet 应该包含源上下文里的对象。在建立新关系的过程中，该方法会根据需要，在目标上下文中创建缺失的对象。

有序的一对多关系的建立方式为：把 sourceSet 里的对象添加到另一个对象的 NSMutableOrderedSet 里面，而那个 NSMutableOrderedSet 正是用来表示这一关系的。我们通过对象的键值对来访问 NSMutableOrderedSet。键是“关系”名称，值是 NSMutableOrderedSet。由于 NSMutableOrderedSet 只能包含互不相同的对象，所以用不着担心无意间为某对象创建了重复的关系。目标上下文中的那个 NSMutableOrderedSet 其对象的顺序要和源上下文中对应的 NSMutableOrderedSet 相符。我们根据在 sourceSet 中所找到的对象来创建等价对象，并依次将其添加到目标对象的 NSMutableOrderedSet 里，以保证它们的顺序与 sourceSet 相符。早前的图 9-3 中曾经粗略描述了这一概念，现在我们用程序清单 9-11 中的代码将其实现出来。

程序清单9-11 CoreDataImporter.m的establishOrderedToManyRelationship方法

```

- (void)establishOrderedToManyRelationship:(NSString*)relationshipName
    fromObject:(NSManagedObject*)object
    withSourceSet:(NSMutableOrderedSet*)sourceSet {

```

```

if (!object || !sourceSet || !relationshipName) {
    NSLog(@"SKIPPED establishment of an Ordered To-Many relationship from %@",
        [self objectInfo:object]);
    NSLog(@"Due to missing Info!");
    return;
}

NSMutableOrderedSet *copiedSet =
    [object mutableOrderedSetValueForKey:relationshipName];

for (NSManagedObject *relatedObject in sourceSet) {
    NSManagedObject *copiedRelatedObject =
        [self copyUniqueObject:relatedObject
            toContext:object.managedObjectContext];

    if (copiedRelatedObject) {
        [copiedSet addObject:copiedRelatedObject];
        NSLog(@"A copy of %@ is related via Ordered To-Many '%@' relationship to %@",
            [self objectInfo:object],
            relationshipName,
            [self objectInfo:copiedRelatedObject]);
    }
}

// REMOVE the relationship from memory after it is committed to disk
[CoreDataImporter saveContext:object.managedObjectContext];
[object.managedObjectContext refreshObject:object mergeChanges:NO];
}

```

请按下列步骤修改 Grocery Dude，以实现 `establishOrderedToManyRelationship` 方法：

1. 把程序清单 9-11 中的代码添加到 `CoreDataImporter.m` 文件底部的 `@end` 语句上方。

虽说 Grocery Dude 程序并未使用有序关系，但笔者还是把这个方法放在这里，以便大家能够在自己的项目中使用 `CoreDataImporter` 类。

9.3.7 拷贝关系

接下来要实现的方法是 `copyRelationshipsFromObject`，该方法负责把源上下文里某个对象的全部关系都拷贝到目标上下文里的等价对象中。刚才所讲的那些方法就是为了编写这个方法而实现的。

在确认了开发者所传入的 `sourceObject` 及 `targetContext` 参数都不是 `nil` 之后，该方法首先要判断目标上下文之中有没有与 `sourceObject` 相等价的对象。这个等价的对象称为 `copiedObject`，我们用早前实现好的 `copyUniqueObject` 方法来创建它。假如在尝试了 `copyUniqueObject` 方法之后 `copiedObject` 依然为 `nil`，那么此方法就提

前返回。

在拷贝关系的时候,该方法首先用 [sourceObject.entity relationshipsByName] 查出源对象所具备的各种关系,然后在获取到的 NSDictionary 里面遍历,找出对源对象有效的关系。假如源对象确实具备某条关系,那我们就在 copiedObject 上面重新创建与之等价的关系。在拷贝某条关系之前,还得先确定其类型。假如是一对多关系或有序的一对多关系,那么就把适当的 sourceSet 传给 establishToManyRelationship 或 establishOrderedToManyRelationship 方法,以拷贝此关系。假如是一对一关系,那就先把相关联的对象拷贝到目标上下文里面,然后再调用相应的方法来建立关系。程序清单 9-12 列出了该方法的代码。

程序清单9-12 CoreDataImporter.m文件中的copyRelationshipsFromObject方法

```
- (void)copyRelationshipsFromObject:(NSManagedObject*)sourceObject
    toContext:(NSManagedObjectContext*)targetContext {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }
    // SKIP establishing relationships with missing info
    if (!sourceObject || !targetContext) {
        NSLog(@"FAILED to copy relationships from '%" to context '%"
            [self objectInfo:sourceObject], targetContext);
        return;
    }

    // SKIP establishing relationships from nil objects
    NSManagedObject *copiedObject =
    [self copyUniqueObject:sourceObject toContext:targetContext];
    if (!copiedObject) {
        return;
    }

    // COPY relationships
    NSDictionary *relationships = [sourceObject.entity relationshipsByName];
    for (NSString *relationshipName in relationships) {

        NSRelationshipDescription *relationship =
        [relationships objectForKey:relationshipName];
        if ([sourceObject valueForKey:relationshipName]) {

            if (relationship.isToMany && relationship.isOrdered) {

                // COPY To-Many Ordered
                NSMutableOrderedSet *sourceSet =
                [sourceObject mutableOrderedSetValueForKey:relationshipName];
                [self establishOrderedToManyRelationship:relationshipName
                    fromObject:copiedObject
                    withSourceSet:sourceSet];
            }
        }
    }
}
```

```

    } else if (relationship.isToMany && !relationship.isOrdered) {

        // COPY To-Many
        NSMutableSet *sourceSet =
        [sourceObject mutableSetValueForKey:relationshipName];
        [self establishToManyRelationship:relationshipName
         fromObject:copiedObject
         withSourceSet:sourceSet];

    } else {

        // COPY To-One
        NSManagedObject *relatedSourceObject =
        [sourceObject valueForKey:relationshipName];
        NSManagedObject *relatedCopiedObject =
        [self copyUniqueObject:relatedSourceObject
         toContext:targetContext];
        [self establishToOneRelationship:relationshipName
         fromObject:copiedObject
         toObject:relatedCopiedObject];

    }
}
}
}

```

请按下列步骤修改 Grocery Dude，以实现 `copyRelationshipsFromObject` 方法：

1. 把程序清单 9-12 中的代码添加到 `CoreDataImporter.m` 文件底部的 `@end` 语句上方。

9.3.8 对实体执行深拷贝

最后要实现的是 `deepCopyEntities` 方法，该方法会根据给定的实体，把全部对象从一个上下文拷贝到另一个上下文之中。实现该方法的具体办法有很多种，而用户对这些办法体验也各有不同。在网上搜索 **core data programming guide: efficiently importing data**，应该会找到一篇由 Apple 所写的编程指南，其中讨论了导入数据所用的技术。那篇文章里面说，应该尽量先把所有的对象都一次性拷贝过去，然后再修正它们之间的“关系”。有些应用程序能够使用这个办法，另外一些程序也许不行。导入数据会比较耗时，假如导入时没有建立关系，哪怕只有几秒钟的时间，用户也可能会认为程序出了 bug。为了应对这一问题，我们有下面几种解决办法（用户可根据应用程序的性质来决定具体方案）：

- ❑ **禁止用户使用应用程序的部分功能或全部功能。**在导入数据时，可以通过 `MBProgressHUD`^① 等方式来显示导入进度。如果导入时间过长，用户可能会着急。所以，对于某些应用程序来说，在导入数据时也许并不能把全部功能都禁用，而是只能禁用其中一部分功能，等数据导入好之后，再将其重新启用。

① 详情参见：<https://github.com/jdg/MBProgressHUD>。——译者注

❑ **先导入所有对象，然后再建立关系。**如果采用这个办法，那么用户可能会看到尚未完全导入的数据，这些数据之间的“关系”可能刚刚建立，也可能还没有建立起来。该办法是否可行，要根据数据模型来定。

❑ **把对象和关系一并导入。**虽说这个办法的效率不如另外两种高，但是整个深拷贝的过程可以放在后台运行，这样的话，对用户所造成的影响就非常小了，甚至根本不会影响用户。这种办法所消耗的设备电量会多一些，但却能使用户在导入数据的过程中继续操作应用程序。

CoreDataImporter 所采用的办法是把对象和关系一并导入。每个对象的深拷贝操作都包裹在 autorelease 池（autorelease pool）中执行，这样的话，系统在数据导入过程中可以定期释放内存。程序清单 9-13 列出了相关代码。

程序清单9-13 CoreDataImporter.m文件中的deepCopyEntities方法

```

- (void)deepCopyEntities:(NSArray*)entities
    fromContext:(NSManagedObjectContext*)sourceContext
    toContext:(NSManagedObjectContext*)targetContext {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }
    for (NSString *entity in entities) {

        NSLog(@"COPYING %@ objects to target context...", entity);
        NSArray *sourceObjects =
            [self arrayForEntity:entity
                inContext:sourceContext
                withPredicate:nil];

        for (NSManagedObject *sourceObject in sourceObjects) {

            if (sourceObject) {
                @autoreleasepool {
                    [self copyUniqueObject:sourceObject
                        toContext:targetContext];
                    [self copyRelationshipsFromObject:sourceObject
                        toContext:targetContext];
                }
            }
        }
    }
}

```

请按下列步骤修改 Grocery Dude，以实现 deepCopyEntities 方法：

1. 把程序清单 9-13 中的代码添加到 CoreDataImporter.m 文件底部的 @end 语句上方。

深拷贝功能所需的代码现在已经写好了，剩下要做的就是 CoreDataHelper.m 里面触发它了。为此，我们需要把 deepCopyEntities 的方法头（method header）放到

CoreDataImporter.h 里面。程序清单 9-14 列出了相关代码。

程序清单9-14 CoreDataHelper.h文件中的deepCopyEntities方法

```
- (void)deepCopyEntities:(NSArray*)entities
    fromContext:(NSManagedObjectContext*)sourceContext
    toContext:(NSManagedObjectContext*)targetContext;
```

请按下列步骤修改 Grocery Dude，把 deepCopyEntities 添加到 CoreDataImporter 的头文件里：

1. 将程序清单 9-14 中的代码添加到 CoreDataImporter.h 文件底部的 @end 语句上方。

9.4 触发深拷贝

本章开头说过，我们打算用现有的这个存储区来演示深拷贝，此存储区里含有默认数据。早前的 setupCoreData 方法曾经通过 setDefaultDataStoreAsInitialStore 把默认数据存储区所在的 DefaultData.sqlite 文件设为程序的初始存储区。我们在本章开头把这行方法调用语句注释掉了，于是，初次安装 Grocery Dude 程序时，它会触发“从 XML 里导入默认数据”的操作。这是因为 setupCoreData 方法还要调用 checkIfDefaultDataNeedsImporting 方法，而后者会触发警示（alert）视图界面，使用户选择是否需要导入默认数据，如果用户选了导入，那就会触发 importFromXML。现在我们要触发的是对持久化存储区的深拷贝操作，所以需要在 CoreDataHelper.m 文件里新建名叫 deepCopyFromPersistentStore 的方法。创建好 CoreDataImporter 实例之后，就可以用 CoreDataImporter 类里新编写的 deepCopyEntities 方法来触发深拷贝了。拷贝过程结束后，应用程序会再刷新一次界面，以响应 SomethingChanged 通知。程序清单 9-15 列出了相关代码。

程序清单9-15 CoreDataHelper.m文件中的deepCopyFromPersistentStore方法

```
- (void)deepCopyFromPersistentStore:(NSURL*)url {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class,
            NSStringFromSelector(_cmd),url.path);
    }

    [_sourceContext performBlock:^(

        NSLog(@"*** STARTED DEEP COPY FROM DEFAULT DATA PERSISTENT STORE ***");

        NSArray *entitiesToCopy = [NSArray arrayWithObjects:
            @"LocationAtHome",@"LocationAtShop",@"Unit",@"Item", nil];

        CoreDataImporter *importer = [[CoreDataImporter alloc]
```

```

initWithUniqueAttributes:[self selectedUniqueAttributes]];

[importer deepCopyEntities:entitiesToCopy
    fromContext:_sourceContext
    toContext:_importContext];

[_context performBlock:^(
    // Tell the interface to refresh once import completes
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"SomethingChanged" object:nil];
)];

NSLog(@"*** FINISHED DEEP COPY FROM DEFAULT DATA PERSISTENT STORE ***");
}
}

```

请按下列步骤修改 Grocery Dude，以实现 deepCopyFromPersistentStore 方法：

1. 把程序清单 9-15 中的代码添加到 CoreDataHelper.m 文件的 DATA_IMPORT 部分底部。

警示视图在响应用户操作时，应该调用的是 deepCopyFromPersistentStore 方法，而不是 importFromXML 方法，所以我们最后还要再修改一次代码。也就是说，需要按照程序清单 9-16 所示，来更新涉及 UIAlertView 的委托方法。

程序清单9-16 CoreDataHelper.m文件中的alertView:clickedButtonAtIndex方法

```

- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    if (alertView == self.importAlertView) {
        if (buttonIndex == 1) { // The 'Import' button on the importAlertView

            NSLog(@"Default Data Import Approved by User");
            /*
            // XML Import
            [_importContext performBlock:^(
                [self importFromXML:[NSBundle mainBundle]
                    URLForResource:@"DefaultData"
                    withExtension:@"xml"]];
            */

            // Deep Copy Import From Persistent Store
            [self loadSourceStore];
            [self deepCopyFromPersistentStore:[self sourceStoreURL]];

        } else {
            NSLog(@"Default Data Import Cancelled by User");
        }
    }
}

```

```

    }
    // Set the data as imported regardless of the user's decision
    [self setDefaultDataAsImportedForStore:_store];
}
}

```

请按下列步骤修改 Grocery Dude，以更新涉及 UIAlertView 的 delegate 方法：

1. 用程序清单 9-16 中的方法，把 CoreDataHelper.m 文件里原有的 alertView:clickedButtonAtIndex 方法替换掉。

2. 从你的设备或 iOS 仿真器中把 Grocery Dude 程序删掉，然后点击 Xcode 的 **Product>Clean** 菜单项。

3. 运行应用程序，点击 **Import** 按钮，导入默认数据。

你会注意到：在导入数据的过程中，用户仍然可以操作应用程序。图 9-4 演示了数据导入操作完成之后的效果。

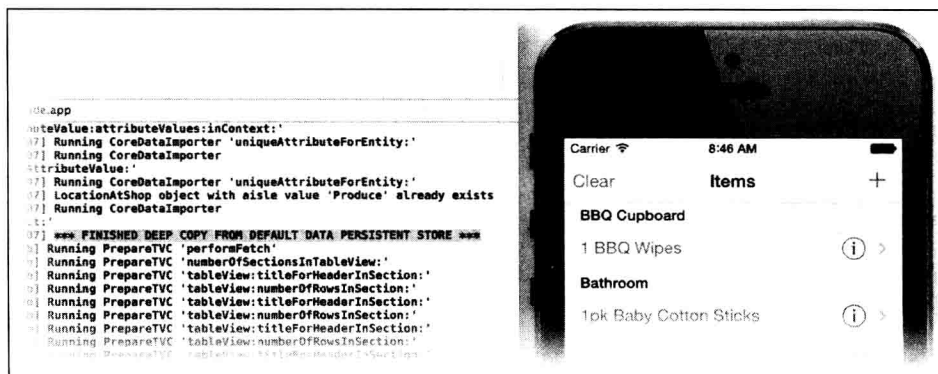


图 9-4 通过深拷贝方式从持久化存储区中导入数据

数据导入要花一定的时间，尤其是在老一些的手机上。如果能在导入过程中定期更新用户界面，把目前已导入的新数据显示出来，那效果就会更好一些。为了准备此功能，我们在 CoreDataHelper.m 文件里添加一个新的方法，用来发送 SomethingChanged 通知。程序清单 9-17 列出了相关代码。

程序清单9-17 CoreDataHelper.m文件中的somethingChanged方法

```

#pragma mark - UNDERLYING DATA CHANGE NOTIFICATION
- (void)somethingChanged {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }
    // Send a notification that tells observing interfaces to refresh their data
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"SomethingChanged" object:nil];
}

```

请按下列步骤修改 Grocery Dude，以实现 somethingChanged 方法：

1. 把程序清单 9-17 中的代码添加到 CoreDataHelper.m 文件底部的 @end 语句上方。

在导入数据的过程中，我们用 NSTimer 来定期触发对 somethingChanged 方法的调用。程序清单 9-18 是修改后的 deepCopyFromPersistentStore 方法，新加进来的代码以粗体表示。

程序清单9-18 CoreDataHelper.m文件中的deepCopyFromPersistentStore方法

```

- (void)deepCopyFromPersistentStore:(NSURL*)url {
if (debug==1) {
    NSLog(@"Running %@", self.class,
        NSStringFromSelector(_cmd),url.path);
}
// Periodically refresh the interface during the import
_importTimer =
[NSTimer scheduledTimerWithTimeInterval:2.0
    target:self
    selector:@selector(somethingChanged)
    userInfo:nil
    repeats:YES];

[_sourceContext performBlock:^(

    NSLog(@"*** STARTED DEEP COPY FROM DEFAULT DATA PERSISTENT STORE ***");

    NSArray *entitiesToCopy = [NSArray arrayWithObjects:
        @"LocationAtHome",@"LocationAtShop",@"Unit",@"Item", nil];

    CoreDataImporter *importer = [[CoreDataImporter alloc]
        initWithUniqueAttributes:[self selectedUniqueAttributes]];

    [importer deepCopyEntities:entitiesToCopy
        fromContext:_sourceContext
        toContext:_importContext];

    [_context performBlock:^(
        // Stop periodically refreshing the interface
        [_importTimer invalidate];

        // Tell the interface to refresh once import completes
        [self somethingChanged];
    )];

    NSLog(@"*** FINISHED DEEP COPY FROM DEFAULT DATA PERSISTENT STORE ***");
}];
}

```

请按下列步骤修改 Grocery Dude，以便在导入数据的过程中定期更新用户界面：

1. 修改 CoreDataHelper.h 文件，把下面这项特性添加到现有的特性下方：

```
@property (nonatomic, strong) NSTimer *importTimer;
```

2. 用程序清单 9-18 中的代码把 CoreDataHelper.m 文件里原有的 deepCopy-FromPersistentStore 方法替换掉。

3. 从你的设备或 iOS 仿真器中删掉 Grocery Dude 程序。

4. 点击 **Product > Clean** 菜单项。

5. 运行 Grocery Dude，点击 Import 按钮，导入默认数据。

每导入一部分数据，程序界面就会在几秒钟之后把这部分数据显示出来，在整个过程中，用户体验不会受到实际影响。

9.5 小结

大家已经学习了深拷贝这一复杂的话题。假如想在现有的持久化存储区里插入数据，而又不想令这些数据相互重复，那么就可以考虑用深拷贝来做。由于 CoreDataImporter 类与具体的模型无关，所以你可以将其运用到自己的应用程序里面。但是要注意，为了防止重复导入数据，应该在现有的持久化存储区的 metadata 里面设定一个键值对。由于上一章已经讲过该技巧，所以此处就不再重述了。

在决定如何将数据从一个存储区迁移至另一个存储区时，一定要先考虑一下 NSPersistentStoreCoordinator 的 migratePersistentStore 方法。该方法可以把持久化存储区中的全部内容都迁移到另一个持久化存储区里。但缺点则是没有提供去除重复数据的选项，而且也不能手工选定想要迁移的实体。虽说如此，但它的速度确实比深拷贝快得多。

9.6 习题

请根据所学内容完成下列试验：

1. 修改下面几个实现文件，将原来的 #define debug 1 改为 #define debug 0，然后重新触发数据导入过程。你会注意到：这次控制台里不会出现大量的记录信息，而且导入数据的速度也比原来快了。

- ☐ CoreDataImporter.m
- ☐ CoreDataTVC.m
- ☐ LocationsAtHomeTVC.m
- ☐ LocationsAtShopTVC.m
- ☐ PrepareTVC.m
- ☐ ShopTVC.m
- ☐ UnitsTVC.m

2. 从设备中删除 Grocery Dude 程序，重新安装并触发数据导入操作，然后赶快按下 Home 按钮（如果在模拟器里运行，可以按“Shift + ⌘ + H”组合键）。这时你会注意到：现有的数据导入过程暂停了，等返回应用程序之后，它会重新开始导入。假如没有返回 Grocery Dude 程序，而是将它终止了，那么程序就不会再次触发数据导入操作了，致使导入的数据可能不够完整。

3. 修改 CoreDataHelper.m 文件的 setupCoreData 方法，把程序清单 9-19 中的代码添加到该方法底部，以尝试 NSPersistentStoreCoordinator 的 migratePersistentStore 方法。再次运行应用程序，你会发现程序里插入了重复的数据。

在学习下一章之前，请先修改 CoreDataHelper.m 文件的 setupCoreData 方法，把 [self setDefaultDataStoreAsInitialStore]; 这一行代码注释掉。此外，也要把习题 3 所添加的代码注释掉。

程序清单9-19 CoreDataHelper.m文件中的setupCoreData方法

```
// The code below demonstrates how to migrate one persistent store to another.
[self loadSourceStore];
[_sourceContext performBlock:^(

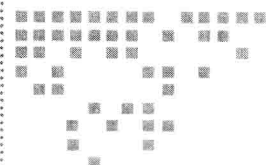
    NSLog(@"*** Attempting to migrate '%@' into '%@' .. Please Wait ***",
        sourceStoreFilename, storeFilename);

    NSError *error = nil;
    if (![_sourceCoordinator migratePersistentStore:_sourceStore
                                                toURL:[self storeURL]
                                                options:nil
                                                ofType:NSSQLiteStoreType
                                                error:&error]) {

        NSLog(@"FAILED to migrate: %@", error);
    } else {
        NSLog(@"The source store '%@' has been migrated to the target store '%@'",
            sourceStoreFilename, storeFilename);

        [_context performBlock:^(

            // Tell the interface to refresh once import completes
            [self somethingChanged];
        )];
    }
}];
```



所谓神经错乱，就是反复做同一件事，却盼着能有不同的结果。

——阿尔伯特·爱因斯坦

第 9 章演示了向持久化存储区中填充数据时所用的技巧。持久化存储区会变得越来越 大，而开发者一定要确保应用程序的反应能力不受影响。我们早前为了提升 Grocery Dude 程序的 性能，已经对 `NSFetchedResultsController` 做了配置，令 `NSFetchRequest` 分批次来获取数据。然而，还有一处不太明显的地方也值得优化，这就是托管对象模型的设计，它对提升性能有重要作用。本章将要讲解如何发现并解决性能问题。

10.1 发现性能问题

当应用程序的开发周期临近尾声时，一定要设法解决各种性能问题。假如没有提前进行适当的性能测试，那么等到应用程序真正表现出性能问题时，可能就来不及了。最糟糕的情况是：用户使用了一段时间之后，发现数据越来越多，而程序也运行得越来越慢。为了预防这一情况，笔者推荐大家尽量在速度非常慢的设备上测试程序，而且测试时所使用的数据集也应该比一般用户所能用到的更大。

由于应用程序的性质不同，所以其表现出来的性能问题也各有区别。但是，有一些通用的指标可以用来判断 iOS 应用程序的性能到底好不好：

- ❑ 应用程序的加载速度应该比较快。
- ❑ 表格视图的滚动效果应该比较平滑。

□ 视图之间的切换速度应该比较快。

□ 无论何时，用户界面都应该保持响应。

测试所用的数据集越大，性能问题也就应该表现得越为明显。而性能问题表现得越为明显，开发者就越容易查到问题的根源。除了数据集要大一些以外，还可以考虑采用照片等比较大的对象来测试，这种对象经常会使应用程序出现性能问题。现在，我们把拍照功能加入 Grocery Dude 之中，使程序里有可能出现较大的对象。本章稍后将会告诉大家一些优化模型的技巧，它们适用于照片这种大对象。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter09.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。开始学习下面的内容之前，请先把设备里原有的 Grocery Dude 程序全都删掉，以确保程序下次运行时能够载入默认的数据。

10.2 实现拍照功能

用户在准备购物清单时，如果能把自己钟爱的那款咖啡照下来就好了，因为它的品牌名称可能不太容易记住。等到了商店之后，万一忘记咖啡是什么牌子的，便可以打开程序看看刚拍的照片。我们打算把拍照功能添加到现有的 Item 视图里面。如果某件货品有照片，那么该照片会显示在 Item 视图之中，此外，还会显示在表格视图的 Prepare 选项卡和 Shop 选项卡里面。想要实现拍照功能，我们得在 Item 视图上面添加一个按钮，并且制作一个用来显示照片的 Image 视图。

请按下列步骤修改 Grocery Dude 项目，以实现拍照按钮和 Image 视图界面：

1. 从 http://www.timroadley.com/LearningCoreData/Icons_add_photo.zip 下载 zip 文件，并将其中的 add_photo 图标解压缩。
2. 选定名为 **Images.xcassets** 的 asset catalog，然后把刚下载的 **add_photo** 图标拖到里面。
3. 选中 **Main.storyboard**。
4. 向 Item 视图中现有的 **Scroll View** 里面拖放一个 **Image View**。
5. 确保 **Attributes Inspector** 界面处于可见状态（可以按“**Option + ⌘ + 4**”组合键调出该界面）。
6. 把 **Image View** 的 **Background** 颜色设为 **Other > Crayons > Mercury**（Mercury 指的是第二个颜色最淡的灰色蜡笔）。
7. 向 Item 视图中现有的 Scroll 视图里拖放一个 **Button**，并在 **Attributes Inspector** 界面（可以按“**Option + ⌘ + 4**”组合键调出该界面）中按照下列步骤配置它：

- 把 **Type** 设为 **Custom**。
- 把写有“Button”字样的标题文本删掉。
- 把 **Image** 设为 **add_photo**。

8. 在 **Size Inspector** 界面（可以按“**Option** + **⌘** + **5**”组合键调出该界面）中，把新按钮的 **Height** 与 **Width** 都设为 **48**。

9. 调整 **Image** 视图的大小，使其宽度与高度相同，然后按图 10-1 所示，把它和新的按钮对齐。

10. 选定 **Item** 视图里面的 **Scroll** 视图，然后点击 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints in ItemVC** 菜单项。

我们必须把这些新对象同对应的 outlet 相连，使开发者可以在代码中引用它们。这一操作要在 **Assistant Editor** 里完成。

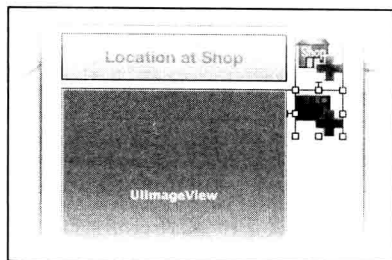


图 10-1 add_photo 按钮与 Image 视图

请按下列步骤修改 Grocery Dude 项目，把拍照按钮及 Image 视图连接到 ItemVC.h：

1. 确保 **Main.storyboard** 处于选中状态。
2. 选定 **Item View Controller**。
3. 把 **Assistant Editor** 界面显示出来（可以按“**Option** + **⌘** + **Return**”组合键）。
4. 假如 Assistant Editor 上方显示的不是 **Automatic > ItemVC.h**，那就将其调整为 **Automatic > ItemVC.h**，如图 10-2 顶部所示。

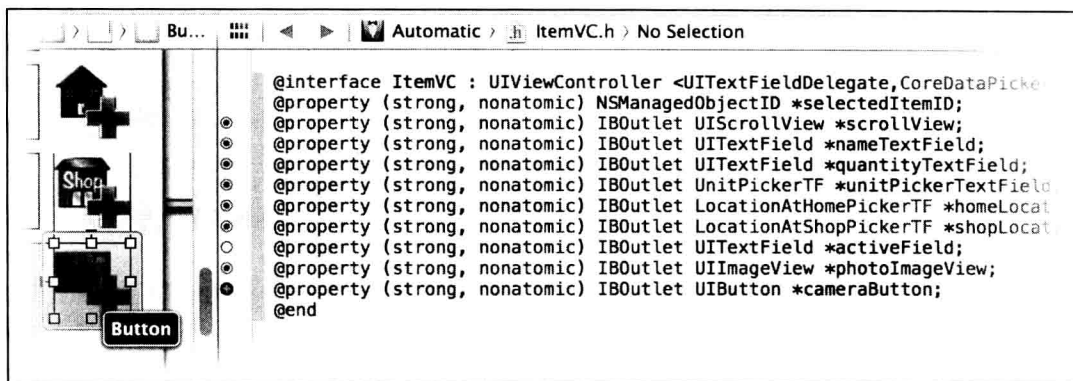


图 10-2 与拍照按钮相连的 cameraButton 特性

5. 按住 **Control** 键，由新建的 **Image View** 向 ItemVC.h 底部的 @end 上方拖一条直线。把新特性的名称设为 photoImageView。请确认 **Type** 是 UIImageView，**Storage** 是 **Strong**。

6. 按住 **Control** 键，从拍照按钮向 ItemVC.h 文件末尾的 @end 上方拖一条直线。把新特性的名称设为 cameraButton。请确认 **Type** 是 UIButton，**Storage** 是 **Strong**。

连接好之后，把鼠标移到 cameraButton 特性左侧的圆圈，此时拍照按钮会凸显出来，如图 10-2 所示。

7. 把 **Standard Editor** 界面显示出来（可以按“⌘ + Return”组合键）。

有了拍照用的控件之后，接下来需要配置适当的协议及委托。ItemVC 类将会采用 UIImagePickerControllerDelegate 协议，以便接收消息，消息中含有通过摄像头所捕获到的照片数据。另外，ItemVC 类还会采用 UINavigationControllerDelegate 协议，以展示拍照界面。最后，需要一个特性来存放 UIImagePickerController 实例，我们把这个特性叫做 camera。

请按下列步骤修改 Grocery Dude 项目，配置 ItemVC 的头文件，以实现拍照功能：

1. 参照下列代码，把新采用的两个协议添加到 ItemVC.h 文件中（新添加的代码以粗体表示）：

```
<UITextFieldDelegate, CoreDataPickerTFDelegate,  
UIImagePickerControllerDelegate, UINavigationControllerDelegate>
```

2. 把下面这个特性放在 ItemVC.h 底端的 @end 语句上方：

```
@property (strong, nonatomic) UIImagePickerController *camera;
```

拍照功能实质上就是 UIImagePickerController，其中有四个方法需要实现：

- ❑ **checkCamera** 方法用于确认摄像头是否可以使用，并据此来调整“拍照按钮”（camera button）的状态。每次显示 Item 视图的时候，都会调用该方法。
- ❑ **showCamera** 方法和 cameraButton 相连。用户按下该按钮时，拍照界面会显示出来。
- ❑ **imagePickerController:didFinishPickingMediaWithInfo** 方法是 Image Picker 的委托方法。用户拍摄照片之后，系统会调用这个委托方法。我们在该方法中把照片数据存放到目前选定的“货品”（item）里面，存放照片数据值的那个属性名字叫做 photoData。照片的存储格式是 JPG 图像，其质量为 50%，其大小为 640×640。
- ❑ **imagePickerControllerDidCancel** 方法也是 Image 选取器的委托方法。如果用户想取消 Camera 视图，那么系统就会调用这个方法。我们在该方法里用代码将 Camera 视图隐藏起来。

程序清单 10-1 列出了相关代码。

程序清单10-1 ItemVC.m文件中的CAMERA 部分

```
#pragma mark - CAMERA  
- (void)checkCamera {  
    if (debug==1) {  
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));  
    }  
}
```

```

        self.cameraButton.enabled =
        UIImagePickerController
            isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera];
    }
    - (IBAction)showCamera:(id)sender {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }
    if ([UIImagePickerController
        isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
    {
        NSLog(@"Camera is available");
        _camera = [[UIImagePickerController alloc] init];
        _camera.sourceType = UIImagePickerControllerSourceTypeCamera;
        _camera.mediaTypes =
            UIImagePickerController
                availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];
        _camera.allowsEditing = YES;
        _camera.delegate = self;
        [self.navigationController presentViewController:_camera
            animated:YES
            completion:nil];
    }
    else
    {
        NSLog(@"Camera not available");
    }
    }

    - (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];

    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID error:nil];

    UIImage *photo =
        (UIImage *)[info objectForKey:UIImagePickerControllerEditedImage];

    NSLog(@"Captured %f x %f photo",photo.size.height, photo.size.width);

    item.photoData = UIImageJPEGRepresentation(photo, 0.5);

    self.photoImageView.image = photo;

    [picker dismissViewControllerAnimated:YES completion:nil];
    }

```

```

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
    [picker dismissViewControllerAnimated:YES completion:nil];
}

```

请按下列步骤修改 Grocery Dude 项目，以实现拍照功能：

1. 把程序清单 10-1 中的代码添加到 ItemVC.m 文件底部的 @end 语句上方。
2. 修改 ItemVC.m 文件的 refreshInterface 方法，把下列代码添加到方法尾部的 if 语句里面，将其放在语句体的最后：

```

self.photoImageView.image = [UIImage imageWithData:item.photoData];
[self checkCamera];

```

3. 选定 **Main.storyboard**。

4. 按住 **Control** 键，从拍照按钮向 Item 视图底部的黄圆圈处拖一条直线，然后选择 **Sent Events > showCamera:** 菜单项。假如没有看到 **Sent Events > showCamera:** 选项，请保存 ItemVC.m 文件后再试一次。

5. 在真机上面测试刚实现好的拍照功能。你应该可以为货品拍摄照片了。请注意，拍照功能在 iOS 仿真器中无法使用。

拍照功能实现好之后，我们来修改表格视图的 Prepare 选项卡和 Shop 选项卡，使其可以将照片显示出来。

请按下列步骤修改 Grocery Dude 项目，把表格视图 单元格准备好，以便在其中显示照片：

1. 确保 **Main.storyboard** 处于选中状态。
2. 在标题为 **Grocery Dude** 的表格视图中选定 **Prototype Cell**。
3. 通过 **Attributes Inspector** 界面（可以按 “**Option** + ⌘ + 4” 组合键调出该界面）把 **Prototype Cell** 的 **Style** 设为 **Right Detail**。
4. 将 Prototype 单元格中的 “Detail” 字样删去。
5. 在标题为 Item 的表格视图中选定其 **Prototype Cell**。
6. 把 Prototype 单元格的 **Style** 设为 **Right Detail**。
7. 将 Prototype Cell 中的 “Detail” 字样删去。
8. 分别修改 PrepareTVC.m 及 ShopTVC.m 文件中的 cellForRowAtIndexPath 方法，把下列代码添加到该方法底部的 return cell; 语句上方：

```

cell.imageView.image = [UIImage imageWithData:item.photoData];

```

再次运行应用程序。你所拍摄的照片应该已经显示在表格视图里面了。正常的运行结果如图 10-3 所示。



图 10-3 字号较小且支持照片显示功能的“Prepare”表格视图

10.3 生成测试数据

用 for 循环很容易就能创建出庞大的测试数据集，只要稍有耐心即可。假如需要的数据特别多，那就不停地创建对象，直到满足需求为止。本书早前曾经讲过如何生成测试数据，此处仍将使用原来的办法，只不过这次还会包含一张比较大的图像。笔者故意将这张图像拷贝许多次，并分别设置给测试所用的每一件货品。这样做的效果就好似每件测试货品真的都有一张互不相同的照片。测试用的图像是 2000×2000 像素的，其尺寸相当大，足以模拟由摄像头所拍出来的照片。程序清单 10-2 列出了生成测试数据所用的代码。请注意，由于该方法只是为了示范而写的，所以它并没有依赖 NSManagedObject 子类的文件，而是使用了针对 Grocery Dude 模型的“键值编码”(key-value coding) 技术。

程序清单10-2 CoreDataHelper.m文件的TEST DATA IMPORT部分

```
#pragma mark - TEST DATA IMPORT (This code is Grocery Dude data specific)
- (void)importGroceryDudeTestData {
    if (debug==1) {
        NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
    }

    NSNumber *imported =
        [[NSUserDefaults standardUserDefaults] objectForKey:@"TestDataImport"];

    if (!imported.boolValue) {
        NSLog(@"Importing test data...");
        [_importContext performBlock:^(

            NSManagedObject *locationAtHome =
                [NSEntityDescription insertNewObjectForEntityForName:@"LocationAtHome"
```

```

                                inManagedObjectContext:_importContext];
NSManagedObject *locationAtShop =
[NSEntityDescription insertNewObjectForEntityForName:@"LocationAtShop"
                                inManagedObjectContext:_importContext];
[locationAtHome setValue:@"Test Home Location" forKey:@"storedIn"];
[locationAtShop setValue:@"Test Shop Location" forKey:@"aisle"];

for (int a = 1; a < 101; a++) {

    @autoreleasepool {

        NSManagedObject *item =
        [NSEntityDescription insertNewObjectForEntityForName:@"Item"
                                inManagedObjectContext:_importContext];
        [item setValue:[NSString stringWithFormat:
                                @"Test Item %i",a] forKey:@"name"];
        [item setValue:locationAtHome forKey:@"locationAtHome"];
        [item setValue:locationAtShop forKey:@"locationAtShop"];
        [item setValue:[UIImagePNGRepresentation(
            [UIImage imageNamed:@"GroceryHead.png"])
            forKey:@"photoData"];
        NSLog(@"Inserting %@", [item valueForKey:@"name"]);
        [CoreDataImporter saveContext:_importContext];
        [_importContext refreshObject:item mergeChanges:NO];
    }
}
// force table view refresh
[self somethingChanged];

// ensure import was a one off
[[NSUserDefaults standardUserDefaults]
    setObject:[NSNumber numberWithBool:YES]
    forKey:@"TestDataImport"];
[[NSUserDefaults standardUserDefaults] synchronize];
}];
}
else {
    NSLog(@"Skipped test data import");
}
}

```

请按下列步骤修改 Grocery Dude，以实现导入测试数据的功能：

1. 修改 CoreDataHelper.m 文件的 DATA_IMPORT 部分，把程序清单 10-2 中的代码添加到现有代码的后方。

2. 从 <http://www.timroadley.com/LearningCoreData/GroceryHead.zip> 下载 zip 文件，将其中的大图像解压缩，然后添加到 **Images.xcassets** 之中。

我们将从 setupCoreData 方法中调用 importGroceryDudeTestData 方法，并把原有的数据导入方法注释掉。程序清单 10-3 列出了修改后的代码。

程序清单10-3 CoreDataHelper.m文件中的setupCoreData方法

```

- (void)setupCoreData {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

    //[self setDefaultDataStoreAsInitialStore];
    [self loadStore];
    [self importGroceryDudeTestData];
    //[self checkIfDefaultDataNeedsImporting];
}

```

如果在导入数据的过程中用户修改了货品，那么同一个对象就有可能在两个上下文中同时被修改。稍后在保存其中一个上下文时，会发成合并冲突。若想处理这种合并冲突，我们必须预先配置合并策略。合并策略用来判定发生冲突时以哪一方的数据为准。有下列五种策略可供选择：

- ❑ **NSErrorMergePolicy** 它是默认的策略。这种合并策略将使得上下文无法完成保存操作。
- ❑ **NSMergeByPropertyObjectTrumpMergePolicy** 该策略解决合并冲突的办法是以上下文中的对象特性值来覆写（overwrite）持久化存储区中相应对象的特性值。
- ❑ **NSMergeByPropertyStoreTrumpMergePolicy** 该策略解决合并冲突的办法是以持久化存储区中的对象特性值来覆写上下文中相应对象的特性值。
- ❑ **NSOverwriteMergePolicy** 该策略解决合并冲突的办法是用上下文中的整个对象来覆写持久化存储区中的相应对象。
- ❑ **NSRollbackMergePolicy** 该策略解决合并冲突的办法是用持久化存储区中的整个对象来覆写上下文中的相应对象。

当数据产生冲突时，合并策略决定了系统是以上下文的内容为准还是以持久化存储区的内容为准。合并可以在对象级别上执行，也可以在特性级别上执行。

请按下列步骤修改 Grocery Dude 项目，以启用合并策略并触发“导入测试数据”的操作：

1. 参照程序清单 10-3 来修改 CoreDataHelper.m 文件的 setupCoreData 方法。
2. 用程序清单 10-4 中的 init 方法来替换 CoreDataHelper.m 文件里原有的 init 方法。新添加的这三行代码以粗体显示，它们分别用来为每个上下文设置合并策略。

程序清单10-4 CoreDataHelper.m文件中的init方法

```

- (id)init {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

    self = [super init];
    if (!self) {return nil;}
}

```

```

_model = [NSManagedObjectModel mergedModelFromBundles:nil];
_coordinator = [[NSPersistentStoreCoordinator alloc]
                initWithManagedObjectModel:_model];
_context = [[NSManagedObjectContext alloc]
            initWithConcurrencyType:NSMainQueueConcurrencyType];
[_context setPersistentStoreCoordinator:_coordinator];
[_context setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];

_importContext = [[NSManagedObjectContext alloc]
                 initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_importContext performBlockAndWait:^(
    [_importContext setPersistentStoreCoordinator:_coordinator];
    [_importContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_importContext setUndoManager:nil]; // the default on iOS
)];

_sourceCoordinator = [[NSPersistentStoreCoordinator alloc]
                     initWithManagedObjectModel:_model];
_sourceContext = [[NSManagedObjectContext alloc]
                 initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_sourceContext performBlockAndWait:^(
    [_sourceContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_sourceContext setPersistentStoreCoordinator:_sourceCoordinator];
    [_sourceContext setUndoManager:nil]; // the default on iOS
)];
return self;
}

```

从 iOS 仿真器中把 Grocery Dude 程序删掉，然后在 Xcode 里重新运行并安装该程序。导入测试数据之后的效果应该如图 10-4 所示。

导入测试数据之后，试着滚动一下这些货品。除非你有一台速度非常快的 Mac，否则滚动效果会显得不顺畅。假如在 iOS 设备上运行 Grocery Dude，那么会出现内存警告，而且程序很可能会崩溃。

10.4 用 SQLDebug 测量性能

假如底层的持久化存储区是 SQLite 格式，那么可以用 SQLDebug 技术来判断 Core Data 应用程序的性能问题。通过 SQLDebug，我们可以看出由程序自动生成的这些 SQL 查询语句需要多久才能执行完毕。想要查看查询语句的执行时间，把 Debug Level（调试级别）设为 1 就足够了。第 2 章曾经讲过，编辑应用程序的 scheme 即可启用 SQLDebug。

请按下列步骤修改 Grocery Dude 项目，以启用 1 级 SQLDebug：

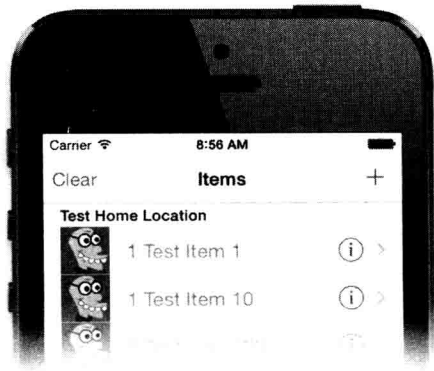


图 10-4 导入测试数据之后的程序界面

1. 点击 **Product > Scheme > Edit Scheme...**。
2. 点击 **Run Grocery Dude**，并切换到 **Arguments** 分页。
3. 点击 **Arguments Passed On Launch** 区域中的“+”按钮，以新增参数。
4. 输入新参数 **-com.apple.CoreData.SQLDebug 1**，然后点击 **OK** 按钮。
5. 在 iOS 仿真器中再次运行 Grocery Dude 程序，并查看控制台日志。正确的结果应该如图 10-5 所示。

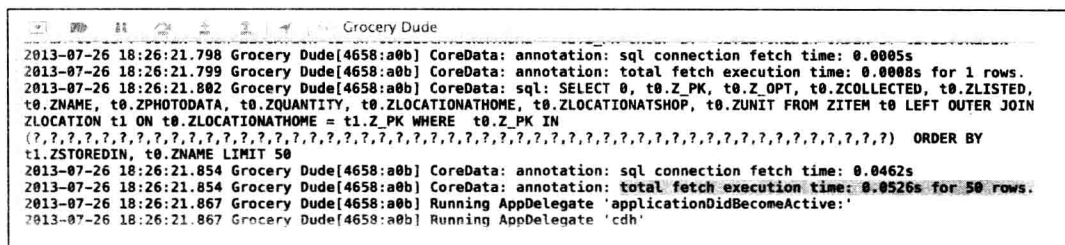


图 10-5 测量 SQL 查询语句的执行时间

为了获取需要显示在 Prepare 表格视图中的货品，程序自动生成了一些 SQL 查询语句，如图 10-5 所示。尽管持久化存储区里有 100 件测试用的货品，但程序却只获取了 50 行记录，这是因为我们曾在 PrepareTVC.m 文件的 configureFetch 方法里以 50 为参数调用了 setFetchBatchSize。即便持久化存储区里有上千件货品，但只要写了这行代码，程序就不会把这些货品一次性地全都加载到内存之中。有一条好的经验法则可用来选定较为合适的“批次获取量”（batch size），就是考虑同时需要显示多少个对象。在使用默认行高的情况下，4 英吋的 Retina 显示屏可以在表格视图里同时显示大约 10 行对象。这就意味着把批次获取量设为 50 显得太高了。在同屏只能显示 10 行的情况下，没有必要再多加载 40 行数据，那样做会浪费资源。批次获取量只需比同屏能够显示的行数稍稍高一点即可，对于标准的表格视图来说，把这个值设为 15 就够了。对选取器视图来说，把批次获取量设置得与它同时能显示的行数差不多就行。

请按下列步骤修改 Grocery Dude 项目，以修改获取请求的批次获取量：

1. 在 Xcode 项目中搜寻 setFetchBatchSize:50。
2. 把项目中每个类里出现的 setFetchBatchSize:50 都换成 setFetchBatchSize:15。Xcode 如果提示启用快照，不用理会就好。
3. 在 iOS 仿真器中再度运行 Grocery Dude 程序，并查看控制台日志。控制台输出的信息应该如图 10-6 所示。

因为获取的行数比原来少，所以 SQL 查询语句的执行时间就比原来短。对于程序的优化工作来说，选取合适的批次获取量，是个良好的开端，但滚动效果依然不够流畅，所以，我们还需继续探查。

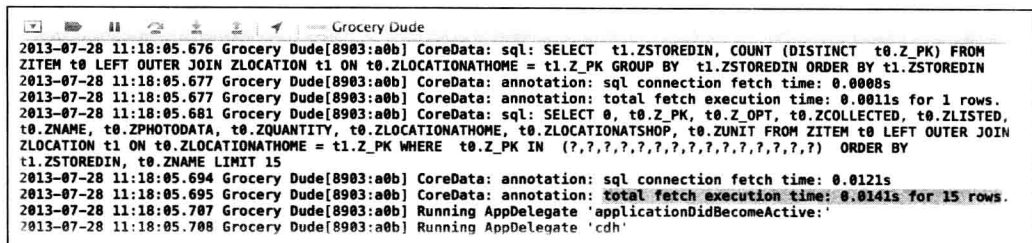


图 10-6 测试数据

10.5 用 Instruments 测量性能

想要测量应用程序的性能，我们可以通过 Instruments 对其进行“性能分析”（Profile）。但遗憾的是，这种分析 Core Data 程序性能的方式不能用在 iOS 真机上面。也就是说，我们必须在 iOS 仿真器中分析 Core Data 程序的性能。这虽然不够理想，但仍然能使我们深入了解 Core Data 程序的一些关键指标，比如 Fetch 操作、“缓存未命中”（Cache Miss）以及 Save 操作的情况。

想要通过 Instruments 对应用程序做性能分析，只需换一种方式来运行程序就可以了。这次我们不直接点击 Run 按钮，而是按住 Run 按钮不放，然后选择 Profile。请读者现在就尝试一下，并注意观察当 Instruments 加载进来的时候按钮会有什么变化。正确的效果应该如图 10-7 所示。

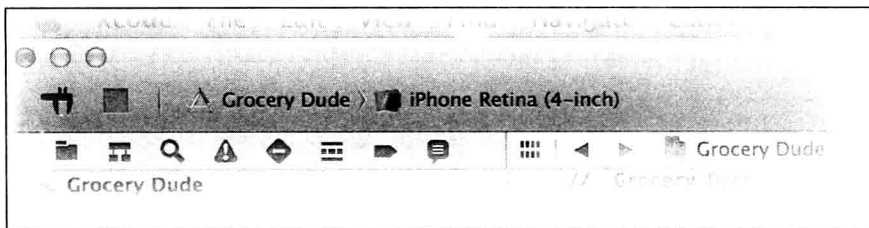


图 10-7 分析应用程序的性能

Instruments 加载进来之后，Xcode 就会像图 10-8 这样提示开发者选取 Trace Template。我们要用的 Core Data 模板位于 File System 分组之中。

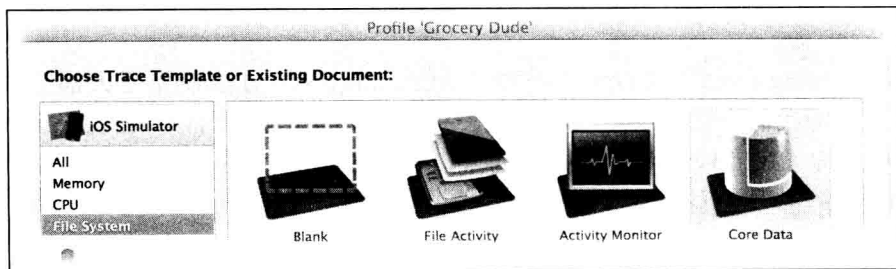


图 10-8 用 Core Data 模板来对程序进行性能分析

请选定 Core Data 模板，然后点击 Profile。这样 Xcode 就会开始记录 Grocery Dude 的性能分析数据了。笔者推荐将 Time Profiler 及 Allocations 同 Core Data 模板结合起来使用，因此我们先停止记录，把这两者添加进来。

请按下列步骤配置 Instruments，以便添加 Time Profiler 及 Allocations：

1. 假如没有看到 Library 界面，那就按“⌘+L”组合键将其显示出来。

2. 把 Time Profiler 及 Allocations 从 Library 界面拖放到 Instruments 主窗口。

3. (该步骤可选) 点击 **File > Save As Template...**，并填入适当的信息，以保存当前模板。这样一来，每次执行性能分析的时候就不用重新配置了。

4. 按下 **Record** 按钮，再度开始性能分析，并测试 Prepare 表格视图的滚动速度。正确的运行效果应该如图 10-9 所示。

应用程序启动之后，Time Profiler 会记录下一大堆活动。Allocations 中的指标应该随着表格视图的滚动而增长，同时，Core Data Fetches 界面中的数值也将与之一并提高。点击 Time Profiler 或 Allocations 之后，使用 Call Tree 选项，就可以发现最占资源的那些方法了。

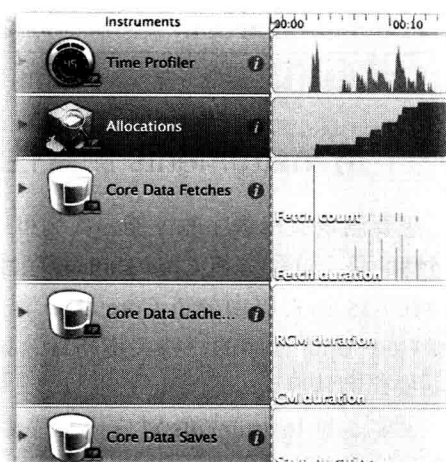


图 10-9 对 Grocery Dude 程序执行性能分析

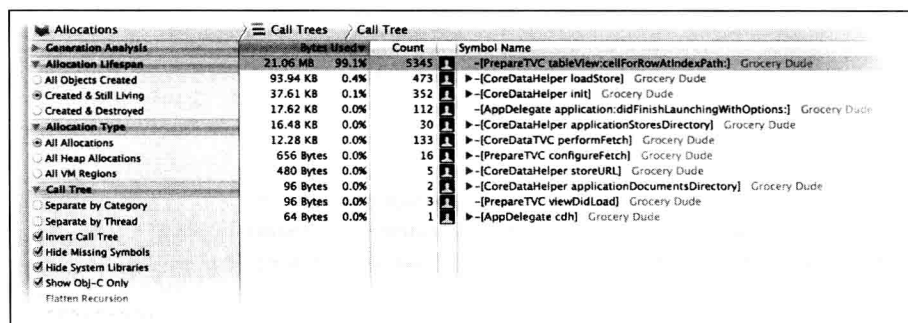


图 10-10 Grocery Dude 程序中各个方法的资源占用量

很显然，PrepareTVC 的 cellForRowAtIndexPathIndexPath 方法使用了大部分内存。这并不奇怪，因为我们令表格视图的单元格去显示一张测试用的图像，而这张图的尺寸是 2000×2000。为了找出 Fetch 操作所花的时长，我们现在点击 Core Data Fetches，然后查看 Event List，如图 10-11 所示。

| Event List / Core Data Fetches | | | | |
|--------------------------------|--|--------------|-------------|----------------|
| # | Caller | Fetch entity | Fetch count | Fetch duration |
| 18 | -(NSManagedObjectContext executeFetchRequest:error:) | | 15 | 24502 |
| 17 | -(NSManagedObjectContext executeFetchRequest:error:) | | 15 | 21672 |
| 16 | -(NSManagedObjectContext executeFetchRequest:error:) | | 15 | 19439 |
| 15 | -(NSManagedObjectContext executeFetchRequest:error:) | | 15 | 17812 |
| 14 | -(NSManagedObjectContext executeFetchRequest:error:) | | 15 | 14616 |
| 13 | -(NSManagedObjectContext executeFetchRequest:error:) | | 15 | 14051 |
| 12 | -(NSManagedObjectContext executeFetchRequest:error:) | | 10 | 13412 |
| 11 | -(NSManagedObjectContext executeFetchRequest:error:) | | 1 | 3738 |
| 10 | -(NSManagedObjectContext executeFetchRequest:error:) | | 100 | 3051 |

图 10-11 测量 Fetch 操作的耗时

执行 Fetch 操作的“耗时”(duration)以毫秒显示,另外,由于你所用的 Mac 型号可能与笔者的不同,所以测得的 Fetch 操作耗时也可能会和截图中的数据有所区别。假如想知道这些 Fetch 调用是由哪个类的哪个方法所发起的,那么可以点击写有“Event List”字样的按钮,并将其改为“Call Tree”。当然了,我们可能还想在 Call Tree 选项中勾选“Show Obj-C Only”及“Hide System Libraries”,以便只把想要看到的那部分数据显示出来。同样的技巧也适用于 Core Data Saves 及 Core Data Cache Misses 界面。如果程序所需的对象不在内存中,那么它就要从持久化存储区里获取数据,这种现象叫做缓存未命中。

10.6 改善程序性能

将每次执行 Fetch 操作时获取的对象数量减少有助于提升性能。除了 `setFetchBatchSize` 之外,还有一些 `NSFetchRequest` 选项也能用来尽量减低获取到的数据集大小。并不是每个选项都适用于各种场合:

- ❑ **`setFetchLimit`** 及 **`setFetchOffset`** 可用来将 Fetch 操作的结果中所包含的行数减少到某个值,并且使 Fetch 操作能够从某个定点开始执行。这对于使用 Page 视图控制器来显示信息的应用程序也许比较合适。每个 Page 的 Fetch Limit (获取操作所能获取的最大记录数)可能相同,但是其 Fetch Offset (执行获取操作时的偏移量)会有所不同。
- ❑ **`setPredicate`** 可以根据开发者所提供的谓词来减少“获取集”的大小。假如开发者提供的是“复合谓词”(compound predicate,也就是根据两个或两个以上的标准来筛选数据),那么应该把“能从结果中排除最多内容”(cut out the most results)的那个谓词放在前面。另外,把过滤数值的谓词放在过滤文本值的谓词之前也能提升性能。比方说, `someNumber>0 && someText LIKE 'whatever'` 就要比 `someText LIKE 'whatever' && someNumber>0` 更高效。
- ❑ **`setPropertiesToGroupBy`** 与 **`setResultType`** 结合起来使用可以模拟出纯 SQL 的 GROUP BY 操作符。假如与 `setHavingPredicate` 结合起来,那么还可

以继续限定获取到的结果。比方说，可以用这些选项来查询放在家中某个位置的货品数量。

- ❑ **setIncludesPropertyValues:NO** 可使获取请求在执行的时候把特性值都排除掉，如果开发者只想获取每个对象的 `objectID`，那么可以使用该选项。
- ❑ **setRelationshipKeyPathsForPrefetching** 选项用来表示某些关系应该“预先获取”（pre-fetch）。pre-fetch 操作会把与某关系相关联的对象提前加载到上下文之中，这样就可以免去“把 fault 还原为托管对象”（fire the fault）这一操作，从而能够提升效率。假如开发者知道自己即将用到这些关联对象，那么就可以使用该选项。

以上所说的这些获取请求选项都不适用于 Grocery Dude 程序，而该程序目前的性能依然比较糟糕。将获取请求完全优化好之后，还应该继续探索其他的性能优化手段。致使该程序性能低下的主要原因显然是表格视图里显示的那张大图，因为通过性能分析可以看出，`cellForRowAtIndexPath` 是“最忙的方法”。如果用预先制作的缩略图来代替它，那么效果也许会更好。缩略图的生成放在下一章里面讲，而我们现在可以先做一些准备工作。

请按下列步骤修改 Grocery Dude 项目，以实现缩略图：

1. 选定 **Model.xcdatamodeld**。
2. 点击 **Editor > Add Model Version...** 菜单项。
3. 点击 **Finish** 按钮，将默认的 **Model 7** 用作模型的名称。
4. 确保 **Model 7.xcdatamodel** 处于选中状态。
5. 在 Item 实体中新增名为 **thumbnail** 的 **Binary Data** 属性。
6. 选定 **Model.xcdatamodeld**，然后在 **File Inspector** 界面（可按“**Option + ⌘ + 1**”组合键调出该界面）中把 **Current Model Version** 设为 **Model 7**。
7. 点击 **Editor > Create NSManagedObject Subclass...**，按照提示重新生成针对 Item 实体的 `NSManagedObject` 子类文件，并覆盖原有的文件。请先确认 **Targets** 中的“**Grocery Dude**”处于勾选状态，然后再点击 **Create** 按钮。
8. 参照下列代码修改 `PrepareTVC.m` 及 `ShopTVC.m` 文件中的 `cellForRowAtIndexPath` 方法，把表格视图里的每一行所要显示的图像改为 `thumbnail`，而不是像原来那样把完整的 `photoData` 显示出来：

```
cell.imageView.image = [UIImage imageWithData:item.thumbnail];
```

对 Grocery Dude 程序做 Profile（性能分析），同时在 Prepare 表格视图界面中再次滚动查看各货品。此时程序显示不出缩略图，因为名叫 `thumbnail` 的那个属性里面还没有数据。尽管没有缩略图，但内存占用量依然非常大。目前的性能分析情况应该如图 10-12 所示。

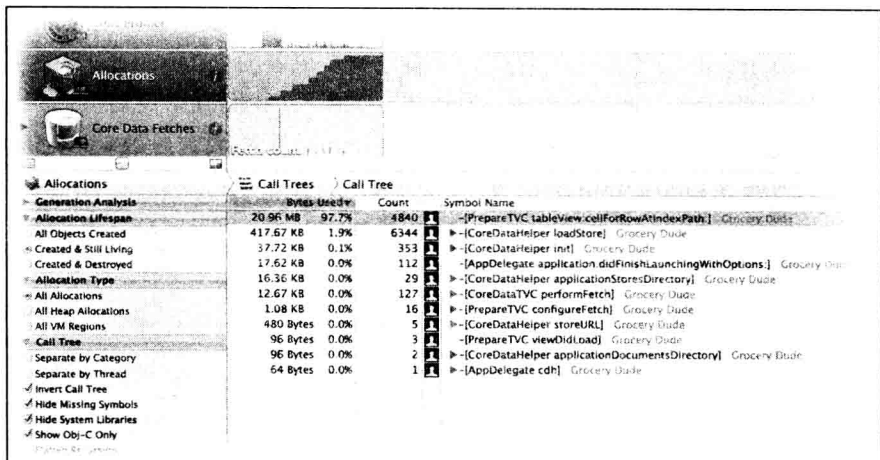


图 10-12 Grocery Dude 程序的内存占用量依然很大

性能问题的关键在于，从持久化存储区中获取对象的时候，对象的所有属性都要载入内存，其中也包括那张大图。为了回避这一模型上的限制，我们应该把包含庞大数据值的属性搬移到另外的实体中。这样一来，就可以在本实体与那个实体之间创建关系，使得当前实体仍然能够按照需要来访问另外一个实体中的数据。采用了这种 faulting 技术之后，只有当程序确实需要那个大对象时，系统才会把它载入内存。等用完了那个对象之后，记得用 `mergeChanges:NO` 选项来调用 `refreshObject` 方法，将其重新转回 `fault`。假如开发者修改了那个对象，那么应先执行保存操作，然后再将其转回 `fault`。

除了使用 faulting 技术之外，还应该允许系统能够把大一些的对象放在持久化存储区之外。对于作为 SQLite 持久化存储区底层数据的二进制数据 属性来说，我们可以对其使用 `Allows External Storage` 选项。该选项允许系统把大于 1MB 左右的对象自动保存到 SQLite 数据库之外。用这种办法来保存大对象的效率更高。图 10-13 演示了“应用程序沙箱”里面的大对象，这些对象没有保存在 sqlite 文件里面。假如开启了该选项的应用程序曾在 iOS 设备上运行过，那么可点击 Xcode 的 **Window > Organizer** 菜单项，并在与电脑相连的设备名称下选择 **Applications**，然后就能看到“沙箱”里的内容了。

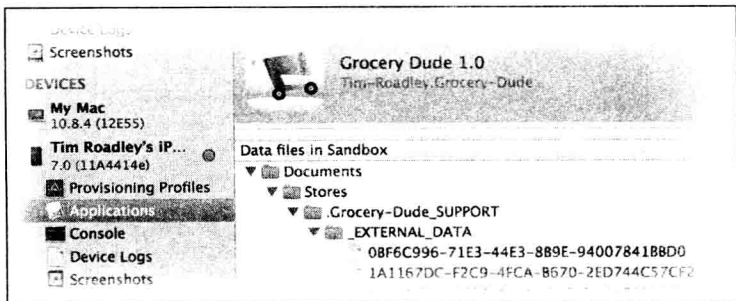


图 10-13 开启了 Allows External Storage 选项之后的效果

为了将刚才改善的模型设计原则用于 Grocery Dude 程序，我们需要把名为 **photoData** 的属性迁移到 **Item_Photo** 这个新的实体中，并将此属性更名为 **data**。**Item** 实体将有一条新的“一对一关系”，它的名字叫做 **photo**，通过该“关系”，我们可以找到与之相连的 **Item_Photo** 实体。而反向的那个“一对一关系”则叫做 **item**。原来需要通过 **photoData** 来访问的数据现在则改为通过 **photo.data** 来访问。

请按下列步骤修改 Grocery Dude 项目，以迁移 photoData：

1. 确保 **Model.xcdatamodeld** 处于选中状态。
2. 点击 **Editor > Add Model Version...** 菜单项。
3. 点击 **Finish** 按钮，把默认的 **Model 8** 用作实体名称。
4. 确保 **Model 8.xcdatamodel** 处于选中状态。
5. 从 **Item** 实体中删除名为 **photoData** 的属性。

6. 新建名叫 **Item_Photo** 的实体。

7. 在 **Item_Photo** 实体中新建 **Binary Data** 类型的属性，并将其命名为 **data**。

8. 在 **Data Model Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）里配置这个名叫 **data** 的属性，启用 **Allows External Storage** 选项。

9. 创建由 **Item** 实体指向 **Item_Photo** 实体的“一对一关系”（**To-One relationship**），并将其命名为 **photo**。把反向的关系命名为 **item**。可以把编辑器的风格切换为 **Graph**，因为在这种界面里创建“关系”是最为方便的。只需按住 **Control** 键，并从一个实体向另一个实体拖曳一条直线，即可创建“关系”。

10. 在 **Data Model Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）中，把 **photo** 关系的 **Delete Rule** 设为 **Cascade**。这样一来，在删除了某件货品之后，其照片数据也会一并删掉。

11. 修改 **CoreDataHelper.m** 文件中的 **selectedUniqueAttributes** 方法，以适应新创建的 **Item_Photo** 实体。请把下面这行代码放在创建 **NSDictionary** 的那行代码上方：

```
[entities addObject:@"Item_Photo"];[attributes addObject:@"data"];
```

完成上述步骤之后，数据模型如图 10-14 所示。

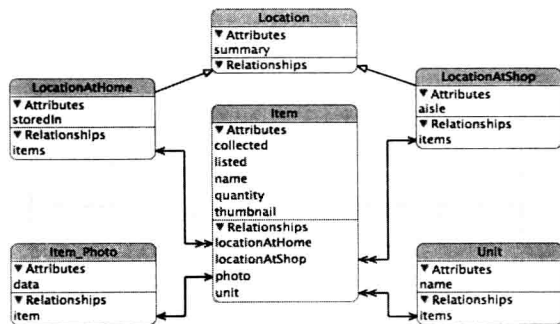


图 10-14 Grocery Dude 程序的“第 8 版数据模型”（Model 8）

这种类型的模型变更不能完全交由 Core Data 来推断，否则就会丢失 **photoData** 里现有的值。在开发真实的应用程序时，我们不想使用户所拍的照片丢失，所以，需要编写“模型映射文件”（model-mapping file）。Core Data 在自动迁移存储区的时候，会推断新旧模型之间的映射方式，而在推断之前，它会先检测模型映射文件。模型映射文件需要完成下面三件事：

- ❑ 把 **Item** 实体里名为 **photoData** 的旧属性映射到 **Item_Photo** 实体里名叫 **data** 的新属性。
- ❑ 映射新的 **photo** 关系，令其从 **Item** 实体指向 **Item_Photo** 实体。
- ❑ 映射新的 **item** 关系，令其从 **Item_Photo** 实体指向 **Item** 实体。

photoData 属性需要映射到 **data** 属性，这一点也许比较明显，但大家可能不太容易看出来的是，新的 **photo** 与 **item** 关系其实也需要映射。假如不映射关系，那么系统就没办法把 **photoData** 里面现存的照片同新的 **data** 特性正确地联系起来。我们用 Value Expression 来创建这种映射，这与模型映射文件推断其他关系的映射方式时所用的办法类似。

请按下列步骤修改 Grocery Dude 项目，手动调整从 Model 7 到 Model 8 的映射：

1. 选中名为 **Data Model** 的组，然后点击 **File > New > File...** 菜单项。
2. 点击 **iOS > Core Data > Mapping Model > Next** 菜单项。
3. 把 **Model 7.xcdatamodel** 选为源模型，然后点击 **Next** 按钮。
4. 把 **Model 8.xcdatamodel** 选为目标模型，然后点击 **Next** 按钮。
5. 将映射模型保存为 **Model7toModel8**，确保 Targets 中的“Grocery Dude”处于选中状态，然后点击 **Create** 按钮。
6. 在 **Model7toModel8.xcmappingmodel** 中选定名为 **Item_Photo** 的 Entity Mapping。
7. 在 **Mapping Model Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）的 Entity Mapping 区域中，把 **Item_Photo** 的 **Source** 设为 **Item**。这样的话，Entity Mapping 的名称就自动改为 **ItemToItem_Photo** 了。
8. 对于名叫 **data** 的 **Destination Attribute** 来说，我们把其 **Value Expression** 设为 `$source.photoData`，以确保旧的 **photoData** 属性能够迁移到新的 **data** 属性。
9. 与之类似，对于名叫 **item** 的 **Destination Relationship** 来说，我们将其 **Value Expression** 设为：

```
FUNCTION($manager, "destinationInstancesForEntityMappingNamed:sourceInstances:"
, "ItemToItem", $source)
```

10. 在 Entity Mappings 中选定 **ItemToItem**。

11. 对于名叫 **photo** 的 **Destination Relationship** 来说，我们将其 **Value Expression** 设为：

```
FUNCTION($manager, "destinationInstancesForEntityMappingNamed:sourceInstances:"
, "ItemToItem_Photo", $source)
```


12. 选定 **Model.xcdatamodeld**，用 **File Inspector** 界面（可按“**Option** + **⌘** + **1**”组合键调出该界面）把 **Current Model Version** 设为 **Model 8**。

13. 点击 **Editor > Create NSManagedObject Subclass...** 菜单项，并根据提示，为 Model 8 中的所有实体重新生成对应的 **NSManagedObject** 子类文件，把原有的文件覆盖掉。确保 **Targets** 中的“**Grocery Dude**”处于勾选状态，然后点击 **Create** 按钮。

14. 重复第 13 步，以确保 Xcode 能够在 **NSManagedObject** 子类的文件中生成正确的关系。

升级完模型之后，一定要记得目前用来填充默认数据的 **sourceStore** 已经同新的数据模型不兼容了。假如还使用原来的 **sourceStore**，那程序就会崩溃。笔者把预先升级过的 **sourceStore** 放在了项目的 **bundle** 中，这样大家用起来会方便一些。但在开发真实的应用程序时，你需要先导入采用 Model 6 模型的默认数据，然后将其升级到 Model 7，再升级到 Model 8。请试着再运行一遍应用程序，这次 Xcode 会提示程序引用了名为 **photoData** 的属性，而这个属性目前还找不到。

请按下列步骤修改 **Grocery Dude** 项目，使程序能够使用新建的属性，并且能够使用升级过的默认存储区：

1. 把 `#import "Item_Photo.h"` 语句添加到 **ItemVC.m** 文件顶端。

2. 修改 **ItemVC.m** 文件中的 **refreshInterface** 方法，用下面这行代码来设置 **Image** 视图所要显示的货品图像：

```
self.photoImageView.image = [UIImage imageWithData:item.photo.data];
```

3. 修改 **ItemVC.m** 文件的 **didFinishPickingMediaWithInfo** 方法，以便将拍好的照片保存到新建的 **data** 属性里。刚才 Xcode 曾说该方法中的一行代码有问题，现在请用下列代码把有问题的那行代码替换掉：

```
if (!item.photo) { // Create photo object it doesn't exist
    Item_Photo *newPhoto =
        [NSEntityDescription insertNewObjectForEntityForName:@"Item_Photo"
                                                inManagedObjectContext:cdh.context];
    [cdh.context obtainPermanentIDsForObjects:
        [NSArray arrayWithObject:newPhoto] error:nil];
    item.photo = newPhoto;
}
item.photo.data = UIImageJPEGRepresentation(photo, 0.5);
```

4. 从下列 URL 下载 ZIP 文件，并把表示持久化存储区的 **sqlite** 文件解压缩，然后将其拖放到 **Grocery Dude** 项目中：http://www.timroadley.com/LearningCoreData/Model_8_DefaultData.zip。请确认 **Copy items into destination group's folder** 选项及 **Targets** 中的“**Grocery Dude**”处于勾选状态。拖放进来的这个持久化存储区就是 Model 8 版本的 **DefaultData.sqlite** 文件，请把原来的那个文件替换掉。

再次对 Grocery Dude 程序做 Profile，并在 Prepare 表格视图中滚动查看各项货品。将早前图 10-12 中的数据同图 10-15 中的数据相对比。

| Call Trees / Call Tree | | | |
|------------------------|-------|-------------|---|
| Bytes Used | Count | Symbol Name | |
| 263.07 KB | 58.8% | 4471 | [-[PrepareTVC tableView:cellForRowAtIndexPath:] Grocery Dude |
| 98.17 KB | 21.9% | 533 | [-[CoreDataHelper loadStore] Grocery Dude |
| 38.42 KB | 8.5% | 368 | [-[CoreDataHelper init] Grocery Dude |
| 17.62 KB | 3.9% | 112 | [-[AppDelegate application:didFinishLaunchingWithOptions:] Grocery Dude |
| 16.48 KB | 3.6% | 30 | [-[CoreDataHelper applicationStoresDirectory] Grocery Dude |
| 12.23 KB | 2.7% | 120 | [-[CoreDataTVC performFetch] Grocery Dude |
| 656 Bytes | 0.1% | 16 | [-[PrepareTVC configureFetch] Grocery Dude |
| 480 Bytes | 0.1% | 5 | [-[CoreDataHelper storeURL] Grocery Dude |
| 96 Bytes | 0.0% | 3 | [-[PrepareTVC viewDidLoad] Grocery Dude |
| 96 Bytes | 0.0% | 2 | [-[CoreDataHelper applicationDocumentsDirectory] Grocery Dude |
| 64 Bytes | 0.0% | 1 | [-[AppDelegate cdh] Grocery Dude |

图 10-15 程序的内存占用量已尽量降至最低

我们可以注意到：内存占用量已经大幅降低了，原先的 `cellForRowAtIndexPath` 方法使用了 20 多 MB 的内存，而现在已经降低到 263KB 左右。把大对象移动到 `Item_Photo` 实体之后，照片数据就成了 `fault`。这意味着在获取 `Item` 对象时，无需把这些照片数据一并载入内存之中。

假如升级模型之前还没有在设备上运行过应用程序，那么等升级完模型后再去设备上运行时，程序就会崩溃，其原因在于 `CoreDataHelper.m` 文件中的 `import-GroceryDudeTestData` 方法是为旧版模型而编写的。程序清单 10-5 列出了修改后的方法代码，该方法现在支持 Model 8。

程序清单10-5 CoreDataHelper.m文件中的importGroceryDudeTestData方法

```
#pragma mark - TEST DATA IMPORT (This code is Grocery Dude data specific)
- (void)importGroceryDudeTestData {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    NSNumber *imported =
        [[NSUserDefaults standardUserDefaults] objectForKey:@"TestDataImport"];

    if (!imported.boolValue) {
        NSLog(@"Importing test data...");
        [_importContext performBlock:^(

            NSManagedObject *locationAtHome =
                [NSEntityDescription insertNewObjectForEntityForName:@"LocationAtHome"
                    inManagedObjectContext:_importContext];

            NSManagedObject *locationAtShop =
                [NSEntityDescription insertNewObjectForEntityForName:@"LocationAtShop"
                    inManagedObjectContext:_importContext];

            [locationAtHome setValue:@"Test Home Location" forKey:@"storedIn"];
            [locationAtShop setValue:@"Test Shop Location" forKey:@"aisle"];
```

```

for (int a = 1; a < 101; a++) {

    @autoreleasepool {

        // Insert Item
        NSManagedObject *item =
        [NSEntityDescription insertNewObjectForEntityForName:@"Item"
         inManagedObjectContext:_importContext];
        [item setValue:[NSString stringWithFormat:@"Test Item %i",a]
         forKey:@"name"];
        [item setValue:locationAtHome forKey:@"locationAtHome"];
        [item setValue:locationAtShop forKey:@"locationAtShop"];

        // Insert Photo
        NSManagedObject *photo =
        [NSEntityDescription insertNewObjectForEntityForName:@"Item_Photo"
         inManagedObjectContext:_importContext];
        [photo setValue:UIImagePNGRepresentation(
         [UIImage imageNamed:@"GroceryHead.png"])
         forKey:@"data"];

        // Relate Item and Photo
        [item setValue:photo forKey:@"photo"];

        NSLog(@"Inserting %@", [item valueForKey:@"name"]);
        [CoreDataImporter saveContext:_importContext];
        [_importContext refreshObject:item mergeChanges:NO];
        [_importContext refreshObject:photo mergeChanges:NO];
    }
}

// force table view refresh
[self somethingChanged];

// ensure import was a one off
[[NSUserDefaults standardUserDefaults]
 setObject:[NSNumber numberWithInt:YES]
 forKey:@"TestDataImport"];
[[NSUserDefaults standardUserDefaults] synchronize];
}];
}
else {
    NSLog(@"Skipped test data import");
}
}
}

```

请按下列步骤修改 Grocery Dude 项目，令导入测试数据所用的那个方法支持 Model 8:

1. 用程序清单 10-5 中的方法把 CoreDataHelper.m 文件里原有的 importGroceryDudeTestData 方法替换掉。

10.7 清理

用完托管对象之后，一定要将其从上下文中移除，以便释放内存。可以通过把它们转换成 fault 来实现这一点。当 Item 视图消失的时候，正应该把用户原来所选定的货品和照片转为 fault。

请按下列步骤修改 Grocery Dude，以便将用户在 Item 视图消失之前所选定的货品转为 fault：

1. 修改 ItemVC.m 文件中的 viewDidDisappear 方法，把程序清单 10-6 中的代码添加到该方法底部。

程序清单10-6 ItemVC.m文件中的viewDidDisappear方法

```
// Turn item & item photo into a fault
NSError *error;
Item *item =
(Item*)[cdh.context existingObjectWithID:self.selectedItemID error:&error];
if (error) {
    NSLog(@"ERROR!!! --> %@", error.localizedDescription);
} else {
    [cdh.context refreshObject:item.photo mergeChanges:NO];
    [cdh.context refreshObject:item mergeChanges:NO];
}
```

10.8 小结

本章讲解了如何用 Instruments 来测量 Core Data 应用程序的性能，还讲解了 NSFetchedRequest 的诸多选项，以及各个选项的优点。同时，大家也看到了模型设计对应用程序性能所起的关键作用，尤其是当相关对象比较大的时候，模型设计就显得更为重要了。请务必记得：尽量在速度最慢的设备上测试程序，而且测试所用的数据集也要尽量大一些，最好与用户可能使用到的最大数据量相仿。假如程序能在非常慢的设备上流畅地处理大量数据，那么，它在新设备上的性能肯定会好很多。

10.9 习题

请根据所学内容完成下列试验：

1. 修改 PrepareTVC.m 文件中的 configureFetch 方法，把配置“批次获取量”的代码注释掉。看看这次程序执行 SQL 查询语句要花多长时间，并用 Instruments 来探查程序的性能。

2. 修改 PrepareTVC.m 文件中的 configureFetch 方法，用程序清单 10-7 中的代

码来设置 request 的 Fetch Limit 和 Fetch Offset。观察程序执行 SQL 查询语句所花的时间，并用 Instruments 来探查程序的性能。请注意观察 Prepare 表格视图里所显示的货品与修改代码之前有何区别。

程序清单10-7 PrepareTVC.m文件中的configureFetch方法

```
[request setFetchLimit:20];  
[request setFetchOffset:50];
```



后台处理

凡事都应尽量简化，但又不能简化得太过分。

——阿尔伯特·爱因斯坦

笔者在第 10 章中给出了一些建议，告诉大家应该如何配置托管对象模型才能达到理想的性能。另外，该章也演示了怎样用 Instruments 来测量程序性能。我们已经把全尺寸（full-size）的照片从 Prepare 表格视图及 Shop 表格视图的单元格里面移走了，然而缩略图还没设置到位。根据照片来创建缩略图是个运算量比较大的过程，所以不能放在前台执行。其中一个原因是创建缩略图所需的计算量比较大，而另外一个原因则在于，如果有很多数据变更需要提交，那么即便是像保存上下文这种简单的操作都可能会影响到用户界面。本章将以缩略图的生成为例，演示如何以专用队列上下文来执行计算量比较大的任务。此外，为了实现后台保存功能，我们还要在持久化存储区与现有的主队列上下文之间创建一个父上下文。

11.1 后台保存

内存占用量过大的问题已经在第 10 章中解决了，而 Prepare 表格视图与 Shop 表格视图里面要显示的缩略图目前仍然是 nil。根据图像动态地创建其缩略图是个计算量很大的过程，所以应该在后台完成，而且不要影响到用户界面。为了能在后台创建缩略图，我们首先要实现后台保存功能。要想实现后台保存，最简单的办法是使用两个具有上下级关系的上下文。我们把后台上下文称为 _parentContext，并将它的 ConcurrencyType 设为 NSPrivateQueueConcurrencyType，令其使用专用队列。而

支援用户界面的前台上下文则称为 `_context`。我们已经把它的 `ConcurrencyType` 设为 `NSMainQueueConcurrencyType` 了，所以它会使用主队列。由于这两个上下文都在内存里，所以它们之间的相互通信是非常迅速的。

子上下文没有持久化存储区。对于子上下文来说，它的父上下文就扮演其持久化存储区的角色。如果在子上下文上面执行保存操作，那么变更后的数据就将保存至父上下文之中。若想将变更后的数据写入持久化存储区，那么就必须要在父上下文上面也执行保存操作。由于父上下文运行在专用队列上，所以保存操作不会影响用户界面。图 11-1 概述了这一流程。

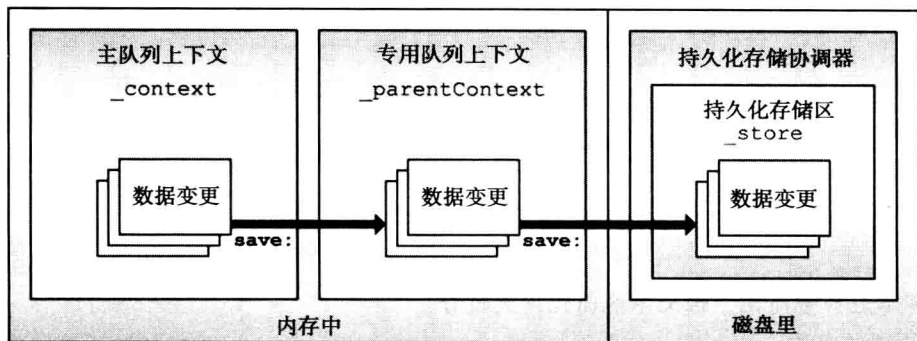


图 11-1 用父上下文与子上下文实现后台保存



提示 为了继续构建范例程序，你需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter10.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

配置好上下文层级 (context hierarchy) 之后，至少需要把一个上下文放在专用队列上运行，而配置专用队列上下文这一操作则应该通过块 (block) 来执行。为了使上下文运行在专用队列上，我们用 `performBlockAndWait` 来配置它，这样做的好处是在需要用到该上下文的时候，它肯定已经配置好了。而 `_context` 本来就运行在主队列上，所以对它的配置工作可以放在“块”的外面执行。正如第 10 章所说，我们应该为所有的上下文都指定一种非默认的 (non-default) 合并策略，以便解决可能发生的冲突。程序清单 11-1 列出了修改之后的代码，其中，配置上下文层级所用的代码以粗体表示。

程序清单 11-1 CoreDataHelper.m 文件中的 `init` 方法

```
- (id)init {
    if (debug==1) {
```

```

    NSLog(@"Running %@ '%@'", self.class, NSStringFromSelector(_cmd));
}

self = [super init];
if (!self) {return nil;}

_model = [NSManagedObjectModel mergedModelFromBundles:nil];
_coordinator =
[[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:_model];

_parentContext = [[NSManagedObjectContext alloc]
    initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_parentContext performBlockAndWait:^(
    [_parentContext setPersistentStoreCoordinator:_coordinator];
    [_parentContext
        setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
)];

_context = [[NSManagedObjectContext alloc]
    initWithConcurrencyType:NSMainQueueConcurrencyType];
[_context setParentContext:_parentContext];
[_context setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];

_importContext = [[NSManagedObjectContext alloc]
    initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_importContext performBlockAndWait:^(
    [_importContext setPersistentStoreCoordinator:_coordinator];
    [_importContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_importContext setUndoManager:nil]; // the default on iOS
)];

_sourceCoordinator = [[NSPersistentStoreCoordinator alloc]
    initWithManagedObjectModel:_model];
_sourceContext = [[NSManagedObjectContext alloc]
    initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_sourceContext performBlockAndWait:^(
    [_sourceContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_sourceContext setPersistentStoreCoordinator:_sourceCoordinator];
    [_sourceContext setUndoManager:nil]; // the default on iOS
)];
return self;
}

```

请按下列步骤修改 Grocery Dude 项目，以实现由父上下文及子上下文所构成的上下文层级：

1. 修改 CoreDataHelper.h 文件，把下列代码添加到其他特性声明语句的上方：

```
@property (nonatomic, readonly) NSManagedObjectContext *parentContext;
```

2. 用程序清单 11-1 中的代码把 CoreDataHelper.m 文件里原有的 init 方法替换掉。

接下来要实现名为 `backgroundSaveContext` 的新方法。该方法将把子上下文保存到父上下文，然后把父上下文保存到持久化存储区。“保存到父上下文”这一操作会执行得很快，因为它是在内存中执行的，而“把父上下文保存到持久化存储区”这一操作，则可能要花些时间，不过这没什么大问题，因为该操作是在专用队列上面以异步方式完成的。程序清单 11-2 列出了相关的代码，这段代码与现有的 `saveContext` 方法相似。

程序清单 11-2 CoreDataHelper.m 文件中的 `backgroundSaveContext` 方法

```

- (void)backgroundSaveContext {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

// First, save the child context in the foreground (fast, all in memory)
[self saveContext];

// Then, save the parent context.
[_parentContext performBlock:^(
    if ([_parentContext hasChanges]) {
        NSError *error = nil;
        if ([_parentContext save:&error]) {
            NSLog(@"_parentContext SAVED changes to persistent store");
        }
        else {
            NSLog(@"_parentContext FAILED to save: %@", error);
            [self showValidationError:error];
        }
    }
    else {
        NSLog(@"_parentContext SKIPPED saving as there are no changes");
    }
}];
}

```

请按下列步骤修改 `Grocery Dude` 项目，以实现后台保存功能：

1. 修改 `CoreDataHelper.m` 文件的 `SAVING` 部分，把程序清单 11-2 中的代码添加到该部分底部。
2. 把下列代码放在 `CoreDataHelper.h` 文件的 `@end` 语句上方，使其他类也可以使用后台保存功能：

```
- (void)backgroundSaveContext;
```

3. 修改 `ItemVC.m` 文件中的 `viewDidDisappear` 方法，把 `[cdh saveContext];` 替换为 `[cdh backgroundSaveContext];`。

4. 分别修改 `AppDelegate.m` 文件中的 `applicationDidEnterBackground` 方法及 `applicationWillTerminate` 方法，把 `[[self cdh] saveContext];` 替换成

```
[[self cdh] backgroundSaveContext];。
```

5. 分别修改 PrepareTVC.m 文件中的 clear 方法及 clearList 方法, 把 [cdh backgroundSaveContext]; 语句添加到方法底部。

6. 修改 ShopTVC.m 文件中的 clear 方法, 并分别修改 ShopTVC.m 文件及 PrepareTVC.m 文件中的 didSelectRowAtIndexPath 方法, 将下列代码添加到这三个方法底部:

```
CoreDataHelper *cdh = [(AppDelegate *)[[UIApplication
sharedApplication] delegate] cdh];
[cdh backgroundSaveContext];
```

7. 分别修改 LocationsAtHomeTVC.m、LocationsAtShopTVC.m、PrepareTVC.m 及 UnitsTVC.m 中的 commitEditingStyle 方法, 把上一步里的代码添加到这四个方法底部。

后台保存功能现在已经实现好了。假如某件货品与某个对象相关联, 而该对象中的某个属性值改变了, 那么开发者必须手动刷新 Prepare 表格视图与 Shop 表格视图。这是因为 NSFetchedResultsController 并不会追踪与获取到的对象相关的其他对象中所发生的变化。具体到 Grocery Dude 程序来说, 这就意味着 PrepareTVC 及 ShopTVC 的 NSFetchedResultsController 只会追踪 Item 对象本身所发生的变化。为了解决这个问题, 我们需要在 LocationAtHomeVC.m、LocationAtShopVC.m 及 UnitVC.m 文件中添加 viewDidDisappear 方法。当相关的视图消失时, 程序清单 11-3 里的这个新方法会投递 SomethingChanged 通知, 而收到该通知的界面则会刷新自己。

程序清单11-3 LocationAtHomeVC.m、LocationAtShopVC.m及UnitVC.m文件中的

viewDidDisappear方法

```
- (void)viewDidDisappear:(BOOL)animated {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
[cdh backgroundSaveContext];
[[NSNotificationCenter defaultCenter] postNotificationName:@"SomethingChanged"
                                         object:nil
                                         userInfo:nil];
}
```

请按下列步骤修改 Grocery Dude 项目, 以便在相关对象发生变化时投递通知:

1. 分别修改 LocationAtHomeVC.m、LocationAtShopVC.m 及 UnitVC.m 文件中的 VIEW 部分, 把程序清单 11-3 中的代码添加到部分底部。

现在可以测试应用程序的后台保存功能了:

1. 删掉 iOS 仿真器里面的 Grocery Dude 程序，然后在 Xcode 中点击 **Product > Clean** 菜单项。

2. 在 iOS 仿真器中运行 Grocery Dude 程序，等待程序把测试用的数据全部导入进来。你可以通过查看控制台日志来追踪导入进度，在导入操作完成之前，表格视图里一直是空白。

3. 在 Prepare 选项卡中点击“+”按钮，新建货品。随意起个名字，然后点击 **Done** 按钮。当 ItemVC 界面消失的时候，会触发保存操作。

只要触发了保存操作，运行在主队列上的 `_context` 与运行在专用队列上的 `_parentContext` 就会向控制台日志里写入一条含有“SAVED”字样的信息，如图 11-2 所示。

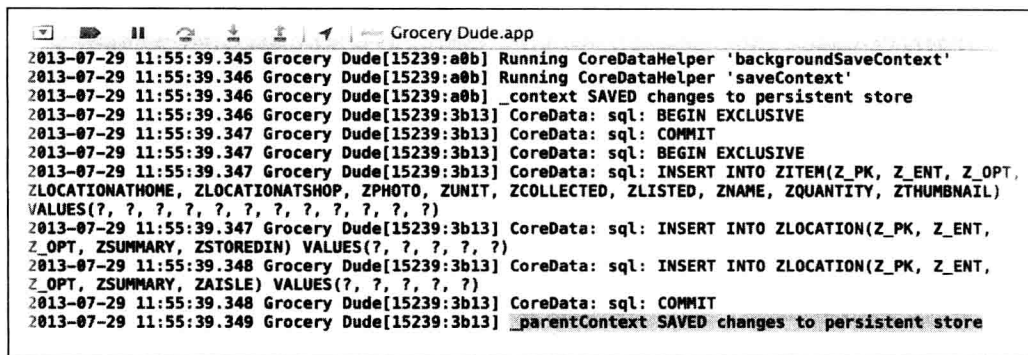


图 11-2 后台保存

实现好后台保存功能之后，你可能会问，它到底给程序带来了什么好处呢？刚开始使用 Grocery Dude 时，哪怕在前台保存数据其速度也照样很快，于是，有人会觉得后台保存所带来的好处是比较少的。当待保存的数据量变大时，后台保存功能的优点才会变得更加明显。比方说，在导入数据的过程中，就可以体现出后台保存的优势了。

11.2 后台处理

建立了 `_parentContext` 之后，数据导入的方式也受到了影响。大家会发现，在导入数据的过程中，已经导入的测试数据并没有出现在表格视图里面。其原因在于，Items 表格视图的 `NSFetchedResultsController` 并不知道它已经可以使用持久化存储区里的新数据了。我们可以设定 `observer`，然后向其发送通知，令其触发合并操作，但其实还有另外一个办法，就是把 `_context` 配置成 `_importContext` 的父上下文。图 11-3 演示了这一概念。

请按下列步骤修改 Grocery Dude 项目，以配置 `_importContext` 的父上下文：

1. 从 `CoreDataHelper.m` 文件的 `init` 方法中移除下列代码：

```
[_importContext setPersistentStoreCoordinator:_coordinator];
```

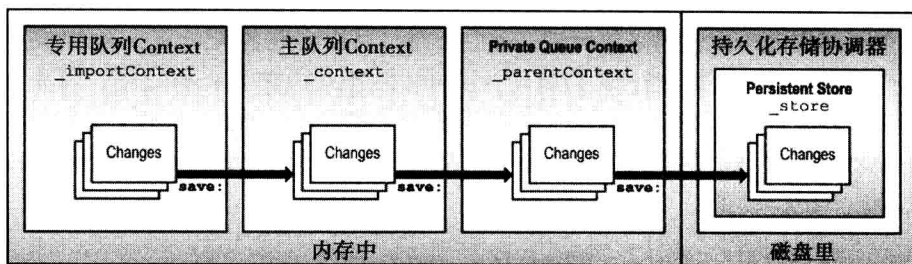


图 11-3 把 `_context` 用作 `_importContext` 的父上下文

2. 把下列代码添加到刚才删除代码的地方：

```
[_importContext setParentContext:_context];
```

虽说本章用不到，但我们现在也应该把 `_context` 设为 `_sourceContext` 的父上下文。请按下列步骤修改 Grocery Dude 项目，以配置 `_sourceContext` 的父上下文：

1. 从 `CoreDataHelper.m` 文件的 `init` 方法中删除下面这行代码：

```
[_sourceContext setPersistentStoreCoordinator:_sourceCoordinator];
```

2. 把下列代码添加到刚才删除代码的地方：

```
[_sourceContext setParentContext:_context];
```

3. 把 `CoreDataHelper` 文件的 `init` 方法中的下列代码注释掉：

```
_sourceCoordinator =  
[[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:_model];
```

由于 `_importContext` 现在已经是 `_context` 的子上下文了，所以只要对 `_importContext` 执行保存操作，那么导入的数据就立刻会出现在监视 `_context` 的那些表格视图之中。这就是使用由“`NSFetchedResultsController` 支持的表格视图”所带来的好处之一。

11.3 建立 `Faulter` 类

生成缩略图的过程可能会影响到大量的对象。凡是需要处理大量对象的场合，就一定要注意避免内存用量暴增，同时要避免用户界面出现“卡顿”现象。第 10 章曾经说过，用完了对象之后，就应该对包含该对象的上下文执行保存操作，然后通过 `refreshObject:mergeChanges:NO` 将对象转为 `fault`。在由父上下文和子上下文所构成的上下文层级中，开发者需要对该层级体系里的每个上下文都做这种处理。也就是说，每导入一个对象，就需要在三个上下文上面都执行一遍保存及转为 `fault` 的操作。

保存上下文并把特定的对象转为 `fault` 是个可以重复的过程。无论针对哪个上下文，我们都可以重复使用同一段代码来执行该任务。于是，可以实现名叫 `Faulter` 的新类，该类能够根据给定的 `objectID` 把相关对象转为 `fault`。在把对象转为 `fault` 之前，首先会保存给定的上下文。假如这个上下文还有父上下文，那么 `Faulter` 会自动在上下文层级中的父上下文上面重复执行这一过程，以便把其中的对象转为 `fault`。程序清单 11-4 列出了 `Faulter` 类的头文件。

程序清单11-4 `Faulter.h`

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface Faulter : NSObject

+ (void)faultObjectWithID:(NSManagedObjectID*)objectID
    inContext:(NSManagedObjectContext*)context;

@end
```

请按下列步骤修改 `Grocery Dude` 项目，以添加 `Faulter` 类：

1. 选定名叫 **Generic Core Data Classes** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 `NSObject`，把 **Class** 名称设为 `Faulter`，然后点击 **Next** 按钮。
5. 确保 **Targets** 中的“`Grocery Dude`”处于勾选状态，然后点击 **Create** 按钮，在 `Grocery Dude` 项目的文件夹里创建类。
6. 用程序清单 11-4 中的代码把 `Faulter.h` 文件里原有的代码全都替换掉。

`Faulter` 类所实现的这个 `faultObjectWithID:inContext` 方法首先会在给定的上下文上面调用 `save` 方法，把尚未保存的变更保存起来。然后，如果相关对象还未转为 `fault`，那就通过 `NSManagedObjectContext` 的 `refreshObject` 方法将其转为 `fault`。把给定的上下文中的对象转为 `fault` 之后，假如还有父上下文，那就在父上下文上面重复执行这一过程。相关代码如程序清单 11-5 所示。

程序清单11-5 `Faulter.m`

```
#import "Faulter.h"
@implementation Faulter

+ (void)faultObjectWithID:(NSManagedObjectID*)objectID
    inContext:(NSManagedObjectContext*)context {
    if (!objectID || !context) {
        return;
    }
}
```

```

[context performBlockAndWait:^(
    NSManagedObject *object = [context objectWithID:objectID];

    if (object.hasChanges) {
        NSError *error = nil;
        if (![context save:&error]) {
            NSLog(@"ERROR saving: %@", error);
        }
    }

    if (!object.isFault) {

        NSLog(@"Faulting object %@ in context %@", object.objectID, context);
        [context refreshObject:object mergeChanges:NO];
    } else {
        NSLog(@"Skipped faulting an object that is already a fault");
    }

    // Repeat the process if the context has a parent
    if (context.parentContext) {
        [self faultObjectWithID:objectID inContext:context.parentContext];
    }
}];
}
@end

```

请按下列步骤修改 Grocery Dude 项目，以实现 Faulter 类：

1. 用程序清单 11-5 中的代码把 Faulter.m 文件里原有的代码全都替换掉。

首先需要用到 Faulter 类的方法，它是 CoreDataHelper.m 文件里现有的 importGroceryDudeTestData 方法。目前，在导入测试数据的过程中，我们只对 _importContext 中的对象执行了保存和转为 fault 的操作。在构建好新的上下文层级之后，我们还需要在导入数据的过程中，对 _context 及 _parentContext 里的新对象也执行保存及转为 fault 的操作才行。假如不执行这些操作，那么插入的数据就没办法保存到持久化存储区里面。程序清单 11-6 列出了修改后的测试数据导入方法。

程序清单 11-6 CoreDataHelper.m 文件中的 importGroceryDudeTestData 方法

```

- (void)importGroceryDudeTestData {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    NSNumber *imported =
        [[NSUserDefaults standardUserDefaults] objectForKey:@"TestDataImport"];

    if (!imported.boolValue) {
        NSLog(@"Importing test data...");
        [_importContext performBlock:^(

```

```

    NSManagedObject *locationAtHome =
    [NSEntityDescription insertNewObjectForEntityForName:@"LocationAtHome"
     inManagedObjectContext:_importContext];
    NSManagedObject *locationAtShop =
    [NSEntityDescription insertNewObjectForEntityForName:@"LocationAtShop"
     inManagedObjectContext:_importContext];
    [locationAtHome setValue:@"Test Home Location" forKey:@"storedIn"];
    [locationAtShop setValue:@"Test Shop Location" forKey:@"aisle"];

    for (int a = 1; a < 101; a++) {

        @autoreleasepool {

            // Insert Item
            NSManagedObject *item =
            [NSEntityDescription insertNewObjectForEntityForName:@"Item"
             inManagedObjectContext:_importContext];
            [item setValue:[NSString stringWithFormat:@"Test Item %i",a]
             forKey:@"name"];
            [item setValue:locationAtHome
             forKey:@"locationAtHome"];
            [item setValue:locationAtShop
             forKey:@"locationAtShop"];

            // Insert Photo
            NSManagedObject *photo =
            [NSEntityDescription insertNewObjectForEntityForName:@"Item_Photo"
             inManagedObjectContext:_importContext];
            [photo setValue:[UIImagePNGRepresentation(
             [UIImage imageNamed:@"GroceryHead.png"])]
             forKey:@"data"];

            // Relate Item and Photo
            [item setValue:photo forKey:@"photo"];
            NSLog(@"Inserting %@", [item valueForKey:@"name"]);
            [Faulter faultObjectWithID:photo.objectID
             inContext:_importContext];
            [Faulter faultObjectWithID:item.objectID
             inContext:_importContext];
        }
    }
    [_importContext reset];

    // ensure import was a one off
    [[NSUserDefaults standardUserDefaults]
     setObject:[NSNumber numberWithBool:YES]
     forKey:@"TestDataImport"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}];
}

```

```

    else {
        NSLog(@"Skipped test data import");
    }
}

```

请按下列步骤修改 Grocery Dude 项目，以改进“测试数据导入方法”：

1. 把 `#import "Faulter.h"` 语句添加到 `CoreDataHelper.m` 文件顶部。
 2. 用程序清单 11-6 中的方法把 `CoreDataHelper.m` 文件里原有的 `importGroceryDudeTestData` 方法替换掉。
 3. 删掉 iOS 仿真器中的 Grocery Dude 程序，然后在 Xcode 里面点击 **Product > Clean** 菜单项。
 4. 在 iOS 仿真器中运行 Grocery Dude 程序，并等待导入测试数据的过程执行完毕。现在你应该会看到导入的数据立刻就能显示出来，不过，目前还是没有缩略图。
- 我们要新建一个能够创建缩略图的类，这个类会用到 `Faulter` 类。

11.4 建立 Thumbnailer 类

这个新类的名字叫做 `Thumbnailer`，它会在后台生成照片的缩略图。该类假定程序里已经配置好了如图 11-3 所示的上下文层级。为了把该类写得通用一些，使开发者能够在其他应用程序里复用它，我们给 `createMissingThumbnailsForEntityName` 方法编写了下面这些参数变量：

- ❑ **entityName** 参数是个字符串，用来表示实体的名称，该实体中有个属性用来存放缩略图。
- ❑ **thumbnailAttributeName** 参数是个字符串，用来表示二进制数据类型属性的名称，该属性就是应该在其中创建缩略图的那个属性。我们假定由 `entityName` 参数所指定的实体里面确实有该属性。
- ❑ **photoRelationshipName** 是个用来表示关系名称的字符串，该关系的起点是由 `entityName` 参数所确定的那个实体，而终点则是包含照片数据的另外一个实体。
- ❑ **photoAttributeName** 参数是个字符串，它用来表示二进制数据类型属性的名称，而该属性里含有照片数据。我们假定由 `photoRelationshipName` 所确定的那条关系确实指向这个属性。
- ❑ **sortDescriptors** 是个可选的参数变量，它用来排列数组中的对象，这些对象的缩略图尚待生成。`Thumbnailer` 类会按照各对象在数组中的顺序、分别为其生成缩略图。如果生成缩略图的顺序与表格视图显示这些缩略图的顺序相符，那么看起来效果会好些。
- ❑ **importContext** 应该是个运行在专用队列上的上下文。缩略图将会创建在这个上下文里面。

程序清单 11-7 列出了 Thumbnailer 类的头文件。

程序清单 11-7 Thumbnailer.h

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface Thumbnailer : NSObject

+ (void)createMissingThumbnailsForEntityName:(NSString*)entityName
    withThumbnailAttributeName:(NSString*)thumbnailAttributeName
    withPhotoRelationshipName:(NSString*)photoRelationshipName
    withPhotoAttributeName:(NSString*)photoAttributeName
    withSortDescriptors:(NSArray*)sortDescriptors
    withImportContext:(NSManagedObjectContext*)importContext;

@end
```

请按下列步骤修改 Grocery Dude 项目，以添加 Thumbnailer 类：

1. 选定名为 **Generic Core Data Classes** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 NSObject，把 **Class** 名称设为 Thumbnailer，然后点击 **Next** 按钮。
5. 确保 Targets 中的“Grocery Dude”处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹里创建类。

6. 用程序清单 11-7 中的代码把 Thumbnailer.h 文件里原有的代码全都替换掉。

Thumbnailer 类把实现缩略图生成功能的代码包裹在块 (block) 里面。调用者所传入的 importContext 参数应该是个运行在专用后台队列上面的上下文。调用 createMissingThumbnailsForEntityName 方法时，Thumbnailer 类首先要根据给定的参数变量来构建 NSFetchedRequest。执行完这个 NSFetchedRequest 之后，就得到了一个包含指针的数组，这些指针所指向的对象都有照片，但都还没有生成缩略图。现在针对每个对象来生成其缩略图。创建好缩略图以后，就把用不到的对象转为 fault，以节省内存。程序清单 11-8 列出了相关代码。

程序清单 11-8 Thumbnailer.m

```
#import "Thumbnailer.h"
#import "Faulter.h"

@implementation Thumbnailer
#define debug 1

+ (void)createMissingThumbnailsForEntityName:(NSString*)entityName
    withThumbnailAttributeName:(NSString*)thumbnailAttributeName
    withPhotoRelationshipName:(NSString*)photoRelationshipName
```

```

        withPhotoAttributeName:(NSString*)photoAttributeName
        withSortDescriptors:(NSArray*)sortDescriptors
        withImportContext:(NSManagedObjectContext*)importContext {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

[importContext performBlock:^(

    NSFetchRequest *request =
    [NSFetchRequest fetchRequestWithEntityName:entityName];
    request.predicate =
    [NSPredicate predicateWithFormat:@"%K==nil && %K.%K!=nil",
        thumbnailAttributeName, photoRelationshipName, photoAttributeName];
    request.sortDescriptors = sortDescriptors;
    request.fetchBatchSize = 15;
    NSError *error;
    NSArray *missingThumbnails =
    [importContext executeFetchRequest:request error:&error];
    if (error) {NSLog(@"Error: %@", error.localizedDescription);}

    for (NSManagedObject *object in missingThumbnails) {

        NSManagedObject *photoObject =
        [object valueForKey:photoRelationshipName];

        if (![object valueForKey:thumbnailAttributeName] &&
            [photoObject valueForKey:photoAttributeName]) {

            // Create Thumbnail
            UIImage *photo =
            [UIImage imageWithData:[photoObject valueForKey:photoAttributeName]];
            CGSize size = CGSizeMake(66, 66);

            UIGraphicsBeginImageContextWithOptions(size, NO, 0.0);
            [photo drawInRect:CGRectMake(0, 0, size.width, size.height)];
            UIImage *thumbnail = UIGraphicsGetImageFromCurrentImageContext();
            UIGraphicsEndImageContext();
            [object setValue:UIImagePNGRepresentation(thumbnail)
                forKey:thumbnailAttributeName];

            // Fault photo object out of memory
            [Faulter faultObjectWithID:photoObject.objectID
                inContext:importContext];
            [Faulter faultObjectWithID:object.objectID
                inContext:importContext];

            // Remove unused variables
            photo = nil;
            thumbnail = nil;
        }
    }
}

```

```

    }];
}
@end

```

请按下列步骤修改 Grocery Dude 项目，以实现 Thumbnailer 类：

1. 用程序清单 11-8 中的代码把 Thumbnailer.m 文件里原有的代码全都替换掉。

包含货品的表格视图界面显示出来的时候，应该用 Thumbnailer 类来生成缺失的缩略图。也就是说，需要向 PrepareTVC 类及 ShopTVC 类里添加 viewDidLoad 方法。程序清单 11-9 列出了该方法的代码，这段代码调用了 Thumbnailer 类的类方法（class method）。

程序清单 11-9 PrepareTVC.m 文件与 ShopTVC.m 文件中的 viewDidLoad 方法

```

- (void)viewDidLoad:(BOOL)animated {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

[super viewDidLoad:animated];

// Create missing Thumbnails
CoreDataHelper *cdh =
[[AppDelegate *)[UIApplication sharedApplication] delegate] cdh];
NSArray *sortDescriptors =
    [NSArray arrayWithObjects:
        [NSSortDescriptor sortDescriptorWithKey:@"locationAtHome.storedIn"
            ascending:YES],
        [NSSortDescriptor sortDescriptorWithKey:@"name"
            ascending:YES],
        nil];

[Thumbnailer createMissingThumbnailsForEntityName:@"Item"
    withThumbnailAttributeName:@"thumbnail"
    withPhotoRelationshipName:@"photo"
    withPhotoAttributeName:@"data"
    withSortDescriptors:sortDescriptors
    withImportContext:cdh.importContext];
}

```

请按下列步骤修改 Grocery Dude 项目，使程序能够把缺失的缩略图创建出来：

1. 将 #import "Thumbnailer.h" 语句分别添加到 PrepareTVC.m 文件及 ShopTVC.m 文件顶部。

2. 把程序清单 11-9 中的 viewDidLoad 方法分别添加到 PrepareTVC.m 文件及 ShopTVC.m 文件 VIEW 部分的顶部。

再次运行应用程序，这次你会看到，程序将缩略图自动生成好之后，缩略图就会显示出来了（参见图 11-4）。无论程序处在何种状态，滚动效果都会很流畅。

假如用户在程序已经生成好缩略图之后又修改了照片，那么程序并不会根据新照片来更新缩略图。为了解决这一问题，我们需要在用户设定新照片时，把相关的缩略图设为 nil。

请按下列步骤修改 Grocery Dude 项目，使程序能根据用户所设置的新照片来生成相应的缩略图：

1. 修改 ItemVC.m 文件中的 didFinishPickingMediaWithInfo 方法，在设定 item.photo.data 的那行代码上方，添加 item.thumbnail=nil; 语句。

假如在速度较慢的设备上运行应用程序，并切换至 Item 视图界面，那么你会注意到程序在加载该界面时会稍显迟钝。这是因为它需要把存储区里的大图加载到内存中。我们可以通过 performBlock 把这种计算量比较大的任务分流出去。程序清单 11-10 中的粗体代码可用于在 Item 视图界面中“延迟加载”(lazily load) 照片。



图 11-4 程序在后台所生成的照片缩略图

程序清单11-10 ItemVC.m文件中的refreshInterface方法

```
- (void)refreshInterface {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.selectedItemID) {
    CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    Item *item =
        (Item*)[cdh.context existingObjectWithID:self.selectedItemID error:nil];
    self.nameTextField.text = item.name;
    self.quantityTextField.text = item.quantity.stringValue;
    self.unitPickerTextField.text = item.unit.name;
    self.unitPickerTextField.selectedObjectID = item.unit.objectID;
    self.homeLocationPickerTextField.text = item.locationAtHome.storedIn;
    self.homeLocationPickerTextField.selectedObjectID =
        item.locationAtHome.objectID;
    self.shopLocationPickerTextField.text = item.locationAtShop.aisle;
    self.shopLocationPickerTextField.selectedObjectID =
        item.locationAtShop.objectID;

    [cdh.context performBlock:^(
        self.photoImageView.image =
            [UIImage imageWithData:item.photo.data];
    )];

    [self checkCamera];
}
}
```

请按下列步骤修改 Grocery Dude 项目，以便在 Item 视图界面里延迟加载照片：

1. 用程序清单 11-10 中的方法把 ItemVC.m 文件里原有的 refreshInterface 方法替换掉。

再度运行应用程序，并切换至 Item 视图界面。Item 视图界面载入后，照片就会紧跟着加载进来。但在 iOS 仿真器上面可能注意不到这段延迟，因为它可以使用的资源比 iOS 真机要多。

11.5 小结

本章讲解了如何在后台保存并导入数据。这些技巧能保证在执行运算量较大的操作时不会影响用户界面。另外，我们也创建了一个有用的类，它就是 `Faulter`。该类可以在上下文层级里执行“保存”及“把对象转为 fault”的操作，使开发者能够少写一些代码。如果你自己的项目里也有上下文层级的话，那就可以套用 `Faulter` 类了。与之相似，我们还编写了名叫 `Thumbnailer` 的新类，开发者也可以把它套用在自己的项目里，以便根据原图像来生成缩略图。

11.6 习题

请根据所学内容完成下列试验：

1. 如果有 iOS 设备，那就在该设备上面测试一下，看看导入数据及创建缩略图的时候用户界面会不会受影响。

2. 修改 `CoreDataHelper.m` 文件中的 `parser:didStartElement` 方法，为上下文层级添加 `Faulter` 支持。这样就可以在有多个上下文的情况下测试“从 XML 中导入数据”这一功能了。在 `CoreDataHelper.m` 文件的 `parser:didStartElement` 方法里，把 STEP 7 的原有代码替换为程序清单 11-11 中的代码。

程序清单11-11 CoreDataHelper.m文件中的parser:didStartElement方法

```
[Faulter faultObjectWithID:item.objectID inContext:_importContext];
[Faulter faultObjectWithID:unit.objectID inContext:_importContext];
[Faulter faultObjectWithID:locationAtHome.objectID inContext:_importContext];
[Faulter faultObjectWithID:locationAtShop.objectID inContext:_importContext];
```

3. 按照下列步骤修改代码，以便在有 `Faulter` 支持的情况下，测试“从 XML 中导入数据”这一功能：

- ❑ 修改 `CoreDataHelper.m` 文件中的 `setupCoreData` 方法，把 `[self import-GroceryDudeTestData];` 语句注释掉。
- ❑ 修改 `CoreDataHelper.m` 文件中的 `setupCoreData` 方法，取消对 `[self check-IfDefaultDataNeedsImporting];` 语句的注释。

- 用程序清单 11-12 中的代码把 CoreDataHelper.m 文件里原有的 alertView:clickedButtonAtIndex 方法替换掉。

程序清单 11-12 CoreDataHelper.m 文件中的 alertView:clickedButtonAtIndex 方法

```

- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    if (alertView == self.importAlertView) {
        if (buttonIndex == 1) { // The 'Import' button on the importAlertView

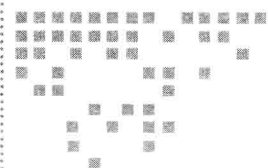
            NSLog(@"Default Data Import Approved by User");
            // XML Import
            [_importContext performBlock:^(
                [self importFromXML:[NSBundle mainBundle]
                URLForResource:@"DefaultData"
                withExtension:@"xml"]];
            });
            // Deep Copy Import From Persistent Store
            //[self loadSourceStore];
            //[self deepCopyFromPersistentStore:[self sourceStoreURL]];

        } else {
            NSLog(@"Default Data Import Cancelled by User");
        }
        // Set the data as imported regardless of the user's decision
        [self setDefaultDataAsImportedForStore:_store];
    }
}

```

- 删掉应用程序并再次运行，以便在有 Faulter 支持的情况下，测试从 XML 中导入数据这一功能。每个对象都会经由上下文层级转为 fault，并保存到持久化存储区里，而开发者则可以在控制台日志中看到该过程。通过控制台中的消息，我们可以发现 importContext、context 及 parentContext 都有不同的内存地址（比方说，会看到 “<NSManagedObjectContext:0x8e12345>” 等字样）。

在学习下一章之前，请把 CoreDataHelper.m 文件 setupCoreData 方法里的其他所有代码都注释掉，只保留 [self setDefaultDataStoreAsInitialStore]; 和 [self loadStore];。



Chapter 12

第 12 章

搜 索

聪不聪明，要看懂不懂得改变。

——阿尔伯特·爱因斯坦

上一章演示了如何在后台执行保存、导入以及创建缩略图等任务。本章将要在 Prepare 选项卡的 PrepareTVC 表格视图中实现搜索功能。在搜索结果中，各件货品所处的部分以及货品的排列顺序都与 PrepareTVC 表格视图相符，这样看上去就好像是就地筛选过一样。我们添加该功能的时候，要使开发者只需修改少许代码，即可令任意的表格视图轻松具备搜索能力。为了达到这种灵活程度，我们把搜索功能尽量放在底层的 CoreDataTVC 类里面实现。本章还要讨论谓词的构成方式对程序性能所带来的影响，而且也会针对如何在大数据集中优化搜索功能给出一些建议。

为了给表格视图添加搜索功能，我们需要使用 UISearchDisplayController。该类含有用来显示搜索结果的表格视图。CoreDataTVC 类曾经实现了 UITableViewDataSource 协议中的一些方法，而这些方法现在需要修改，以应对新加进来的表格视图。用户点击“搜索框”（search bar），UISearchDisplayController 的表格视图就会显示出来。当用户输入待查询的文本时，程序实际上正在根据这些文本构造谓词，以便对执行获取请求所获的结果进行筛选。我们将给这个获取请求新配置一个专门用于搜索的 NSFetchedResultsController。为了应对这个新的 NSFetchedResultsController，我们还需修改 CoreDataTVC 类所实现的 NSFetchedResultsController Delegate 方法。



为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter11.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

12.1 修改 CoreDataTVC 类

等到修改完 CoreDataTVC 类并使其具备搜索能力之后，开发者就可以在 Grocery Dude 项目中所有继承自该类的那些类上面轻松地实现搜索功能了。而为了使 CoreDataTVC 具备搜索能力，我们需要修改其头文件，令该类采用 UISearchBarDelegate 及 UISearchDisplayDelegate 协议。程序清单 12-1 列出了采用这两个搜索协议之后的 CoreDataTVC 头文件。

程序清单12-1 CoreDataTVC.h

```
#import <UIKit/UIKit.h>
#import "CoreDataHelper.h"
@interface CoreDataTVC : UITableViewController
<NSFetchedResultsControllerDelegate, UISearchBarDelegate, UISearchDisplayDelegate>
@property (strong, nonatomic) NSFetchedResultsController *frc;
- (void)performFetch;
@end
```

请按下列步骤修改 Grocery Dude 程序，令其采用相关的搜索协议：

1. 用程序清单 12-1 中的代码把 CoreDataTVC.h 文件里原有的代码全都替换掉，以便使 CoreDataTVC.h 采用 UISearchBarDelegate 及 UISearchDisplayDelegate 协议。

接下来将把两个新特性添加到 CoreDataTVC 里面。第一个特性叫做 searchFRC，它是个 NSFetchedResultsController，用来管理搜索结果。用户输入待搜索的文本时，程序会根据文本构建新的谓词，并用该谓词重新制作 searchFRC。第二个特性叫做 searchDC，它是个 UISearchDisplayController，用来显示搜索结果。程序清单 12-2 列出了这两个新的特性。

程序清单12-2 CoreDataTVC.h文件中的searchFRC方法与searchDC方法

```
@property (strong, nonatomic) NSFetchedResultsController *searchFRC;
@property (strong, nonatomic) UISearchDisplayController *searchDC;
```

请按下列步骤修改 Grocery Dude 项目，以添加实现搜索功能所用的两个新特性：

1. 将程序清单 12-2 中的两个新特性添加到 CoreDataTVC.h 文件里现有的 frc 特性下方。

为了能够同时支持 `self.frc` 及 `self.searchFRC`，我们现在需要修改 `CoreDataTVC` 类的代码。也就是说，要修改涉及 `UITableViewDataSource` 协议及 `NSFetchedResultsControllerDelegate` 协议的相关方法。不过，在修改这些方法之前，我们先添加两个新方法，一个方法可以根据表格视图找到与之对应的 `NSFetchedResultsController`，而另一个则相反，可以根据 `NSFetchedResultsController` 找到与之对应的表格视图。程序清单 12-3 列出了这两个方法的代码，我们用三元操作符来实现这两个判断。

程序清单 12-3 CoreDataTVC.m 文件中的 GENERAL 部分

```
#pragma mark - GENERAL
- (NSFetchedResultsController*)frcFromTV:(UITableView*)tableView {
    /*
        If the given tableView is self.tableView return self.frc,
        otherwise self.searchFRC
    */
    return (tableView == self.tableView) ? self.frc : self.searchFRC;
}
- (UITableView*)TVFromFRC:(NSFetchedResultsController*)frc {
    /*
        If the given fetched results controller is self.frc return self.tableView,
        otherwise self.searchDC.searchResultsTableView
    */
    return (frc == self.frc) ? self.tableView : self.searchDC.searchResultsTableView;
}
```

请按下列步骤修改 `Grocery Dude` 项目，以便在 GENERAL 部分中实现两个新的方法：

1. 把程序清单 12-3 中的代码添加到 `CoreDataTVC.m` 文件底部的 `@end` 语句上方。
2. 把下列代码添加到 `CoreDataTVC.h` 文件底部的 `@end` 语句上方。

```
- (NSFetchedResultsController*)frcFromTV:(UITableView*)tableView;
- (UITableView*)TVFromFRC:(NSFetchedResultsController*)frc;
```

程序清单 12-4 列出了修改之后的“`UITableView` 数据源方法”[⊖]，这些方法利用了我们刚才在 GENERAL 部分里新编写的那些辅助方法。

程序清单 12-4 CoreDataTVC.m 文件中的 DATASOURCE: UITableView 部分

```
#pragma mark - DATASOURCE: UITableView
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
}
```

⊖ `UITableView` 数据源方法，实际上是指 `UITableViewDataSource` 协议里所规定的一些方法，下同。——译者注

```

        return [[[self frcFromTV:tableView] sections]
                objectAtIndex:section] numberOfObjects];
    }
    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
        return [[[self frcFromTV:tableView] sections] count];
    }
    - (NSInteger)tableView:(UITableView *)tableView
    sectionForSectionIndexTitle:(NSString *)title
        atIndex:(NSInteger)index {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
        return [[self frcFromTV:tableView]
                sectionForSectionIndexTitle:title atIndex:index];
    }
    - (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
        return [[[self frcFromTV:tableView] sections] objectAtIndex:section] name];
    }
    - (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }
        return [[self frcFromTV:tableView] sectionIndexTitles];
    }
}

```

请按下列步骤修改 Grocery Dude 项目，以添加对 searchFRC 的支持：

1. 用程序清单 12-4 中的代码把 CoreDataTVC.m 文件里原有的整个 DATASOURCE: UITableView 部分全都替换掉（提示：把光标放在 CoreDataTVC.m 文件里，然后点击 **Editor > Code Folding > Fold Methods & Functions** 菜单项，这样就能更清晰地看到类文件里的各个部分了）。

搜索结果将会显示在表格视图里面，而这个表格视图是 searchDC 的一个特性。通过 searchDC.searchResultsTableView 即可访问该特性。为了用搜索到的结果来填充表格视图，我们需要修改 CoreDataTVC.m 文件里的某些方法，这些方法是为了实现 NSFetchedResultsControllerDelegate 协议而写的。这次我们还会用到 GENERAL 部分里面的辅助方法。程序清单 12-5 列出了相关代码。

程序清单12-5 CoreDataTVC.m文件中的DELEGATE: NSFetchedResultsController部分

```

#pragma mark - DELEGATE: NSFetchedResultsController
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    [[self TVFromFRC:controller] beginUpdates];
}
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    [[self TVFromFRC:controller] endUpdates];
}
- (void)controller:(NSFetchedResultsController *)controller
    didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
        atIndex:(NSUInteger)sectionIndex
    forChangeType:(NSFetchedResultsChangeType)type {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    switch(type) {
        case NSFetchedResultsChangeInsert:
            [[self TVFromFRC:controller]
                insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeDelete:
            [[self TVFromFRC:controller]
                deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}
- (void)controller:(NSFetchedResultsController *)controller
    didChangeObject:(id)anObject
        atIndexPath:(NSIndexPath *)indexPath
    forChangeType:(NSFetchedResultsChangeType)type
    newIndexPath:(NSIndexPath *)newIndexPath {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    switch(type) {
        case NSFetchedResultsChangeInsert:
            [[self TVFromFRC:controller]
                insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]

```

```

        withRowAnimation:UITableViewRowAnimationAutomatic];
    break;

case NSFetchedResultsControllerChangeDelete:
    [[self TVFromFRC:controller]
     deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
     withRowAnimation:UITableViewRowAnimationAutomatic];
    break;

case NSFetchedResultsControllerChangeUpdate:
    if (!newIndexPath) {
        [[self TVFromFRC:controller]
         reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationNone];
    }
    else {
        [[self TVFromFRC:controller]
         deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationNone];
        [[self TVFromFRC:controller]
         insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
         withRowAnimation:UITableViewRowAnimationNone];
    }
    break;

case NSFetchedResultsControllerChangeMove:
    [[self TVFromFRC:controller]
     deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
     withRowAnimation:UITableViewRowAnimationAutomatic];
    [[self TVFromFRC:controller]
     insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
     withRowAnimation:UITableViewRowAnimationAutomatic];
    break;
}
}
}

```

请按下列步骤修改 Grocery Dude 项目，以便支持 searchDC 中的表格视图界面：

1. 用程序清单 12-5 中的代码把 CoreDataTVC.m 文件里原有的 DELEGATE: NSFetchedResultsController 部分替换掉。

接下来，需要在 CoreDataTVC.m 文件中添加一个针对 UISearchDisplayController 的委托方法。当搜索结束时，该方法会把 NSFetchedResultsController 类型的 searchFRC 特性及其 delegate 设为 nil。程序清单 12-6 列出了相关代码。

程序清单12-6 CoreDataTVC.m文件中的DELEGATE: UISearchDisplayController部分

```

#pragma mark - DELEGATE: UISearchDisplayController
- (void)searchDisplayControllerDidEndSearch:(UISearchDisplayController *)controller {

```

```

if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

self.searchFRC.delegate = nil;
self.searchFRC = nil;
}

```

请按下列步骤修改 Grocery Dude 项目，以实现 `searchDisplayControllerDidEndSearch` 方法：

1. 把程序清单 12-6 中的代码添加到 `CoreDataTVC.m` 文件底部的 `@end` 语句上方。

接下来要修改 `CoreDataTVC` 类，使其能够在用户修改了待搜索的文本后重新加载 `searchFRC`。为此，我们要实现一个新的方法，它会在给定的上下文里按谓词来获取某给定实体的对象。排序和划分部分所用的信息也会一并传给该方法，使它可以据此把获取到的对象排好顺序。程序清单 12-7 列出了这个新方法的方法头。

程序清单12-7 `CoreDataTVC.h`文件中的`reloadSearchFRCForPredicate`方法头

```

- (void)reloadSearchFRCForPredicate:(NSPredicate*)predicate
    withEntity:(NSString*)entity
    inContext:(NSManagedObjectContext*)context
    withSortDescriptors:(NSArray*)sortDescriptors
    withSectionNameKeyPath:(NSString*)sectionNameKeyPath;

```

请按下列步骤修改 Grocery Dude 项目，以添加 `reloadSearchFRCForPredicate` 方法头：

1. 把程序清单 12-7 中的代码添加到 `CoreDataTVC.h` 文件底部的 `@end` 语句上方。

对于 `reloadSearchFRCForPredicate` 方法的实现代码大家应该相当熟悉了吧。该方法所做的事情就是用给定的变量来创建获取请求，并执行这个 `Fetch`。程序清单 12-8 列出了相关代码。

程序清单12-8 `CoreDataTVC.m`文件中的`reloadSearchFRCForPredicate`方法

```

#pragma mark - SEARCH
- (void)reloadSearchFRCForPredicate:(NSPredicate*)predicate
    withEntity:(NSString*)entity
    inContext:(NSManagedObjectContext*)context
    withSortDescriptors:(NSArray*)sortDescriptors
    withSectionNameKeyPath:(NSString*)sectionNameKeyPath {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }

    NSFetchedRequest *request = [[NSFetchedRequest alloc] initWithEntityName:entity];
    request.sortDescriptors = sortDescriptors;
    request.predicate = predicate;
    request.fetchBatchSize = 15;
}

```

```

self.searchFRC =
    [[NSFetchedResultsController alloc] initWithFetchRequest:request
                                         managedObjectContext:context
                                         sectionNameKeyPath:sectionNameKeyPath
                                         cacheName:nil];

self.searchFRC.delegate = self;

[self.searchFRC.managedObjectContext performBlockAndWait:^(
    NSError *error;
    if (![self.searchFRC performFetch:&error]) {
        NSLog(@"SEARCH FETCH ERROR: %@", error);
    }
)];
}
}

```

请按下列步骤修改 Grocery Dude 项目，以实现 reloadSearchFRCForPredicate 方法：

1. 把程序清单 12-8 中的代码添加到 CoreDataTVC.m 文件底部的 @end 语句上方。

最后还要给 CoreDataTVC 里面再添加一个新的方法，即 configureSearch 方法，CoreDataTVC 的子类可以用该方法实现搜索功能。这个新的方法将通过程序代码来向相关的表格视图顶部添加 UISearchBar。此外，该方法还会给搜索框配置适当的委托和数据源。程序清单 12-9 列出了相关代码。

程序清单 12-9 CoreDataTVC.m 文件中的 configureSearch 方法

```

- (void)configureSearch {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

UISearchBar *searchBar =
    [[UISearchBar alloc] initWithFrame:
        CGRectMake(0, 0, self.tableView.frame.size.width, 44.0)];
searchBar.autocorrectionType = UITextAutocorrectionTypeNo;
self.tableView.tableHeaderView = searchBar;

self.searchDC =
    [[UISearchDisplayController alloc] initWithSearchBar:searchBar
                                         contentsController:self];

self.searchDC.delegate = self;
self.searchDC.searchResultsDataSource = self;
self.searchDC.searchResultsDelegate = self;
}

```

请根据下列步骤修改 Grocery Dude 项目，以实现 configureSearch 方法：

1. 把程序清单 12-9 中的代码添加到 CoreDataTVC.m 文件 SEARCH 部分底部的 @end 语句上方。
2. 把下列代码添加到 CoreDataTVC.h 文件底部的 @end 语句上方：

```
-(void)configureSearch;
```

CoreDataTVC 子类实现搜索功能所需的基础代码现在都已经写好了。

12.2 修改 PrepareTVC 类

对于 PrepareTVC 来说，实现搜索功能所需的大部分代码都已由其父类 CoreDataTVC 写好了。我们现在只需配置表格视图界面所要显示的数据即可。PrepareTVC 类还要实现一个“UISearchDisplayController 的 delegate 方法”。每当用户在搜索框中输入文本时，系统都会调用该方法。假如文本长度大于 0，那么该方法就根据给定的 searchString 参数来创建谓词。我们把这个谓词传给 reloadDataSearchFRCForPredicate 方法，以便重新加载 searchFRC。sectionNameKeyPath 参数和 sortDescriptors 参数的取值应该与向 PrepareTVC 里填充数据时所用的值相符。程序清单 12-10 列出了相应的代码。

程序清单12-10 PrepareTVC.m文件中的SEARCH部分

```
#pragma mark - SEARCH
- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
shouldReloadTableForSearchString:(NSString *)searchString {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }
    if (searchString.length > 0) {
        NSLog(@"--> Searching for '%@'", searchString);
        NSPredicate *predicate =
            [NSPredicate predicateWithFormat:@"name CONTAINS[cd] %@", searchString];

        NSArray *sortDescriptors =
            [NSArray arrayWithObjects:
             [NSSortDescriptor sortDescriptorWithKey:@"locationAtHome.storedIn"
              ascending:YES],
             [NSSortDescriptor sortDescriptorWithKey:@"name"
              ascending:YES], nil];

        CoreDataHelper *cdh =
            [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
        [self reloadDataSearchFRCForPredicate:predicate
         withEntity:@"Item"
         inContext:cdh.context
         withSortDescriptors:sortDescriptors
         withSectionNameKeyPath:@"locationAtHome.storedIn"];
    } else {
        return NO;
    }
}
```

```

    }
    return YES;
}

```

请按照下列步骤修改 Grocery Dude 项目，以实现 PrepareTVC 的搜索功能：

1. 把程序清单 12-10 中的代码添加到 PrepareTVC.m 文件底部的 @end 语句上方。
2. 修改 PrepareTVC.m 文件中的 viewDidLoad 方法，把 [self configureSearch]; 语句添加到该方法底部。

在表格视图里面实现搜索功能的核心内容就是 shouldReloadTableForSearchString 方法所配置的那些谓词。假如你要用这些代码为自己的应用程序添加搜索功能，那么最需要修改的地方就是配置谓词所用的那些语句了。实现搜索功能时，要留意下面几个关键问题：

- ❑ 如果使用复合谓词（也就是同时使用两个或两个以上的谓词），那么应该把“能从结果中排除最多内容”（filter out the most results）的那个谓词放在前面。指定谓词时所用的顺序会极大地影响到 SQL 查询语句的执行时间。
- ❑ 如果使用复合谓词，那么应该把“按照日期或数字来筛选数据”的谓词放在“按照文本来筛选数据”的那些谓词前面。这一条的优先程度低于前一条。
- ❑ 假如要处理包含上千行数据的大数据集（large data set），那么可以给待搜索的实体添加一个预先处理好的属性，这个属性里面是个“规格化的字符串”（normalized string）。该字符串里应该都是小写字母，并且不包含诸如“á”这样的字符。有了该字符串之后，开发者就无需专门去配置不区分大小写及不区分附加符号的谓词了，而这将会提升搜索的速度。我们可以在谓词中使用 [cd] 来表示该谓词对大小写和附加符号不敏感。

假如现在运行应用程序，并试图在 Prepare 选项卡中搜寻货品，那么程序就会崩溃，因为 cellForRowAtIndexPath 方法还没有为支持 searchFRC 做好准备。程序清单 12-11 列出了修改后的代码，其中，有变动的那些代码以粗体表示。

程序清单 12-11 PrepareTVC.m 文件中的 cellForRowAtIndexPath 方法

```

- (UITableViewCell*)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
      NSLog(@"Running %@ '%@'", self.class, NSStringFromSelector(_cmd));
    }

    static NSString *cellIdentifier = @"Item Cell";

    UITableViewCell *cell =
      [tableView dequeueReusableCellWithIdentifier:cellIdentifier];
    if (cell == nil) {
      cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1
                                         reuseIdentifier:cellIdentifier];
    }

```



```

cell.accessoryType = UITableViewCellAccessoryDetailButton;
Item *item = [[self frcFromTV:tableView] objectAtIndex:indexPath.index];

NSMutableString *title = [NSMutableString stringWithFormat:@"%i %i %i",
                        item.quantity, item.unit.name, item.name];
[title replaceOccurrencesOfString:@"(null)"
                        withString:@""
                        options:0
                        range:NSMakeRange(0, [title length])];
cell.textLabel.text = title;

// make selected items orange
if ([item.listed boolValue]) {
    [cell.textLabel setFont:[UIFont fontWithName:@"Helvetica Neue" size:18]];
    [cell.textLabel setTextColor:[UIColor orangeColor]];
}
else {
    [cell.textLabel setFont:[UIFont fontWithName:@"Helvetica Neue" size:16]];
    [cell.textLabel setTextColor:[UIColor grayColor]];
}
cell.imageView.image = [UIImage imageWithData:item.thumbnail];
return cell;
}

```

假如把 indexPath 传给显示搜索结果的那个表格视图界面，那么将会使程序崩溃，所以，我们这次令 cellForRowAtIndexPath 方法调用 dequeueReusableCellWithIdentifier，而不是像原来那样调用 dequeueReusableCellWithIdentifier:forIndexPath。此外，该方法这次会根据相应的表格视图来决定加载到单元格中的 item 对象。

请按下列步骤修改 Grocery Dude 项目，令 cellForRowAtIndexPath 方法支持 searchFRC：

1. 用程序清单 12-11 中的代码把 PrepareTVC.m 文件里原有的 cellForRowAtIndexPath 方法替换掉。

删除并重新运行应用程序，然后搜索“baby”。正常的运行结果应该如图 12-1 所示。

搜索功能已经见效了，不过为了使 PrepareTVC 类完全支持 searchFRC，我们还得修改 commitEditingStyle、didSelectRowAtIndexPath 及 accessoryButtonTappedForRowWithIndexPath 方法才行。修改后的 commitEditingStyle 方法如程序清单 12-12 所示。

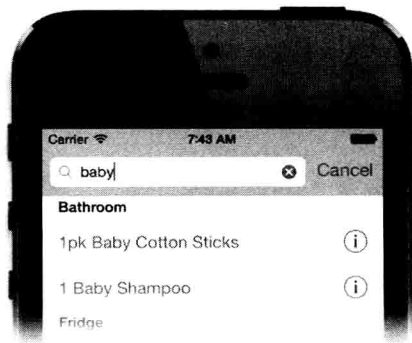


图 12-1 搜索

程序清单12-12 PrepareTVC.m文件中的commitEditingStyle方法

```

- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }

    if (editingStyle == UITableViewCellEditingStyleDelete) {

        NSFetchedResultsController *frc = [self frcFromTV:tableView];
        Item *deleteTarget = [frc objectAtIndex:indexPath:indexPath];
        [frc.managedObjectContext deleteObject:deleteTarget];
        [tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
    }
    CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    [cdh backgroundSaveContext];
}

```

请按下列步骤修改 Grocery Dude 项目，继续为 PrepareTVC 类添加对 searchFRC 的支持：

1. 用程序清单 12-12 中的代码把 PrepareTVC.m 文件里原有的 commitEditingStyle 方法替换掉。

程序清单 12-13 列出了修改后的 didSelectRowAtIndexPath 方法。

程序清单12-13 PrepareTVC.m文件中的didSelectRowAtIndexPath方法

```

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }

    NSFetchedResultsController *frc = [self frcFromTV:tableView];
    NSManagedObject *itemid = [[frc objectAtIndex:indexPath:indexPath] objectAtIndex:indexPath];
    Item *item =
    (Item*)[frc.managedObjectContext existingObjectWithID:itemid error:nil];
    if ([item.listed boolValue]) {
        item.listed = [NSNumber numberWithInt:NO];
    }
    else {
        item.listed = [NSNumber numberWithInt:YES];
        item.collected = [NSNumber numberWithInt:NO];
    }
    CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
    [cdh backgroundSaveContext];
}

```

请按下列步骤修改 Grocery Dude 项目，继续为 PrepareTVC 类添加对 searchFRC 的支持：

1. 用程序清单 12-13 中的代码把 PrepareTVC.m 文件里原有的 didSelectRowAtIndexPath 方法替换掉。

程序清单 12-14 列出了修改后的 accessoryButtonTappedForRowWithIndexPath 方法。

程序清单12-14 PrepareTVC.m文件的accessoryButtonTappedForRowWithIndexPath方法

```
- (void)tableView:(UITableView *)tableView
accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    ItemVC *itemVC =
    [self.storyboard instantiateViewControllerWithIdentifier:@"ItemVC"];
    itemVC.selectedItemID =
    [[[self frcFromTV:tableView] objectAtIndex:indexPath.index] objectAtIndex:indexPath.item];
    [self.navigationController pushViewController:itemVC animated:YES];
}
```

请按下列步骤修改 Grocery Dude，继续为 PrepareTVC 类添加对 searchFRC 的支持：

1. 用程序清单 12-14 中的代码把 PrepareTVC.m 文件里原有的 accessoryButtonTappedForRowWithIndexPath 方法替换掉。

12.3 小结

本章讲解了为应用程序添加搜索功能所用的办法，使得开发者可以轻松地令任何一个表格视图界面都具备搜索能力。在实现搜索功能时，我们采用三元操作符在多个表格视图与 NSFetchedResultsController 之间进行判断，这种技巧可以防止代码量膨胀。如果要在自己的应用程序里实现搜索，那么需要仔细思考待搜索的数据集到底有多大。假如待搜索的是个“大数据集”，那么就应该考虑在实体中添加一种特殊的属性，这种属性的值是预先处理过的“规格化字符串”。此做法可以确保用户能享有平滑而流畅的搜索体验。要是对搜索有比较高端的需求，那么请研究一下 Apple 的 Search Kit 框架。

12.4 习题

请根据所学内容完成下列试验：

1. 按照下面三段代码分别修改 PrepareTVC 类 shouldReloadTableForSearchString 方法中的搜索谓词。然后搜索字母“j”，并对比 SQL 查询语句在不同谓词下的执行时间与

搜索到的结果：

```
[NSPredicate predicateWithFormat:
@"name beginswith[cd] %@ OR name endswith[cd]%"
, searchString, searchString];

[NSPredicate predicateWithFormat:@"name contains %@", searchString];

[NSPredicate predicateWithFormat:@"name like[cd] %@", [[[NSString
stringWithFormat:@"%*" stringByAppendingString:searchString]
stringByAppendingString:@"%*"]];
```

2. 用与 PrepareTVC 相同的办法来修改 ShopTVC 类，为其实现搜索功能。给 ShopTVC.m 文件里添加 shouldReloadTableForSearchString 方法的时候，别忘了根据 locationAtShop.aisle 来修改 sectionNameKeyPath 参数及 sortDescriptors。为了防止程序崩溃，还需要像对 PrepareTVC.m 那样，修改 ShopTVC.m 文件中的 cellForRowAtIndexPath 方法。

3. 修改 ShopTVC.m 文件中的搜索谓词，只在搜索结果中显示处于“listed”状态的货品。

做完练习之后，请禁用 SQL Debug 模式。除了 CoreDataHelper.m 及 AppDelegate.m 之外，把其他所有类的调试标志都禁止掉。想要禁用调试标志，可以在代码中搜索 #define debug 1，并将其替换为 #define debug 0。

本章范例项目含有在 ShopTVC 上面启用搜索功能所需的代码，笔者将这部分代码注释掉，以供大家参考。

与 Dropbox 相结合的备份与恢复

信息不等于知识。

——阿尔伯特·爱因斯坦

第 12 章已经把搜索功能添加到 Grocery Dude 程序了。现在我们要继续扩充该程序的功能集 (feature set)，为其添加数据备份与数据恢复功能。为了实现此功能，我们需要把程序同 Dropbox 相集成。Dropbox 是个免费的 Web 服务，它提供有限的基于云端的存储。开发者可以利用 Dropbox Sync API for iOS (www.dropbox.com/developers/sync) 在设备端创建一个 Dropbox 文件系统。凡是拷贝到这个文件系统里的内容都会自动同步到用户的 Dropbox 账号。假如 Grocery Dude 的用户还没有 Dropbox 账号，那就需要先创建一个。本章要演示创建 ZIP 文件与从 ZIP 文件中恢复备份数据的全过程，同时还要讲解如何把这些文件传给 Dropbox，以及如何从 Dropbox 中取回这些文件。恢复数据的过程中，我们需要在恢复了备份文件之后，按照相关的步骤来重新加载持久化存储区。

读者不要误以为本章是在讲数据库同步，本章要演示的是如何备份并恢复 Core Data 持久化存储区。假如你想在某个用户的多台 iOS 设备之间同步持久化存储区，那么应该看看接下来的两章，它们都和 iCloud 有关。如果要在多个用户的设备间同步持久化存储区，那么就应该阅读第 16 章。

为了备份 Core Data，我们需要保存 `applicationStoresDirectory` 中的所有内容。启动 Grocery Dude 程序时，可以在控制台日志中看到 `Grocery-Dude.sqlite` 文件正是放在这个文件夹里面的。备份持久化存储区文件时，一定要把与之相伴的其他关键文件也备份起来，这其中还包括一些隐藏文件。这些文件都要和 SQLite 文件一并保存起来。

由于我们启用了 WAL 日志记录模式，并且在配置实体的属性时启用了 **Allows External Storage**，所以会出现这些文件。另外，将来程序里可能还会出现其他存储区，所以，最好是把整个 Stores 目录备份起来，这样比较通用，能够适应以后的变化。

备份数据的时候，我们会创建一个 ZIP 文件，并把整个 Stores 目录都拷贝到该文件中。创建 ZIP 文件是一种存放备份数据的好办法，因为它占用的空间比原始文件少，而且更利于传输。创建好 ZIP 文件之后，我们把它移动到设备本地的 Dropbox 缓存中，这样就能自动同步到 Dropbox Web 服务了。图 13-1 概述了这一过程。

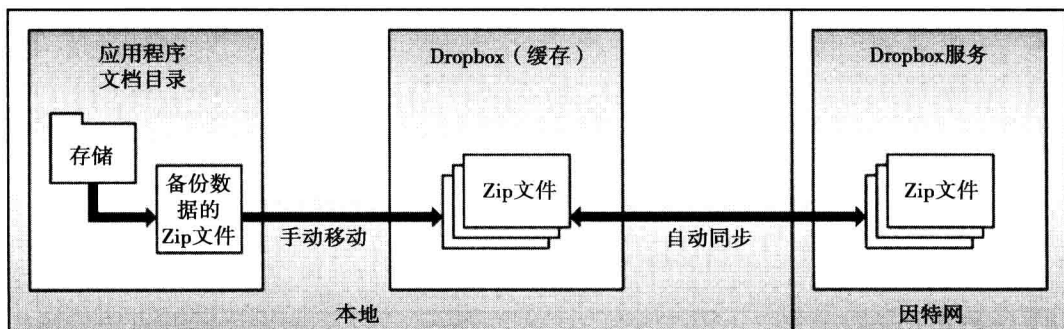


图 13-1 通过 Dropbox 来备份数据

13.1 与 Dropbox 相集成

把应用程序同 Dropbox 集成起来之后，就可以访问与用户的 Dropbox 文件系统相连的本地缓存了。与 Dropbox 相集成的应用程序可以通过 **Sync API** 在 Dropbox App 文件夹下访问它们各自的子文件夹。例如，对于 Grocery Dude 程序来说，Apps/Grocery Dude/ 就是它能够看到的根文件夹。Dropbox 框架会把这个缓存文件夹里的所有内容都自动同步到 Dropbox Web Service。假如你还没有 Dropbox 账号，那就需要在 www.dropbox.com 上创建一个。

要想把应用程序与 Dropbox 集成起来，我们需要通过 Dropbox 门户网站来配置一个新的应用程序。程序的名称必须是独一无二的，由于“Grocery Dude”这个名字已经有人使用了，所以读者需要使用类似“Grocery Dude by 你的名字”这样的名称来确保你的应用程序不和别人的重名。

请按下列步骤新建 Dropbox API app：

1. 用读者自己的 Dropbox 账号访问 <https://www.dropbox.com/developers/apps>。
2. 创建新的 **Dropbox API app**，选择 **Files and datastores** 选项，把该 app 的访问范围局限在由它自己所创建的文件之内（这种权限也称为“App Folder”）。
3. 将 **App name** 设为“Grocery Dude by 你的名字”，或者选一个不和别的 app 相重复

的名字。

Dropbox 网站的布局将来可能会有所变化，读者可能需要参考该网站上的教程，以了解如何新建 Sync API app。一般来说，正常的结果应该如图 13-2 所示。

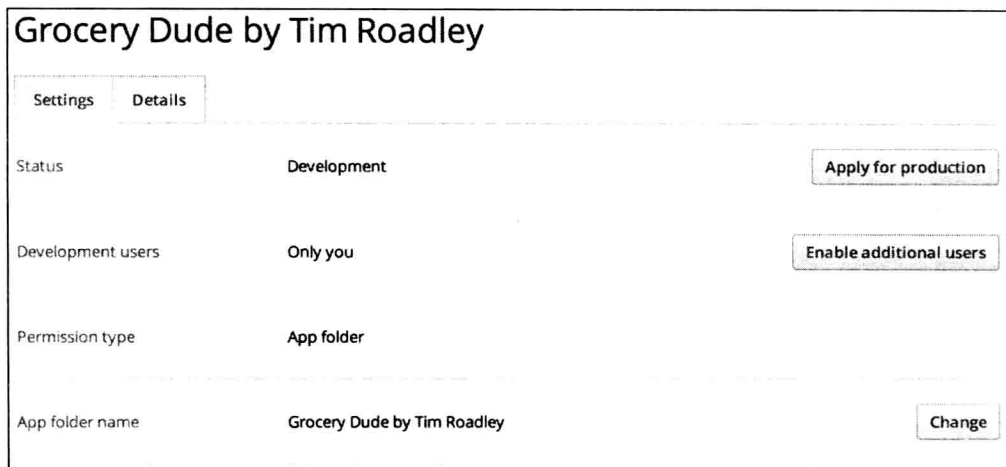


图 13-2 通过 Dropbox 来备份数据

Dropbox API app 的页面中还会显示出 **app key** 与 **app secret**。开发者需要把这两个东西记下来，因为稍后需要用它们来替换相关的代码。为了便于演示，笔者在范例代码中使用 APP_KEY 及 APP_SECRET 来表示它们，而读者则需要分别将其改为自己的 app key 与 app secret。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter12.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

13.1.1 支持框架

为了给 Grocery Dude 添加 Dropbox 支持，我们首先需要下载 Dropbox Sync API iOS SDK。在笔者编写本书时，其最新版本为 1.1.3，但该版本并不支持新的 iOS 64 位架构。等 Dropbox 发布了新的支持框架后，笔者会更新网上的范例项目，以供大家下载。除了 SDK 之外，Grocery Dude 项目还需要同几个支持框架（supporting framework）相链接。

请按下列步骤修改 Grocery Dude 项目，添加所需的框架，以支持 Dropbox：

1. 从 Dropbox 网站 (<http://www.dropbox.com/developers/sync>) 下载最新的 **Sync API iOS SDK** 并将其解压缩。笔者编写本书时, v1.1.3 版的 SDK 可以从 **Sync API > Install SDK > iOS** 下载。

2. 把下载并解压好的 **Dropbox.framework** 目录拖放到 Grocery Dude 项目的 **Frameworks** 文件夹里, 请确认 **Copy items into destination group's folder** 选项及 Targets 中的 “Grocery Dude” 处于勾选状态, 然后点击 **Finish** 按钮。

3. 在 TARGETS 中选定 **Grocery Dude**, 切换到 **General** 分页, 然后向下滚动到 **Linked Frameworks and Libraries** 部分, 如图 13-3 所示。

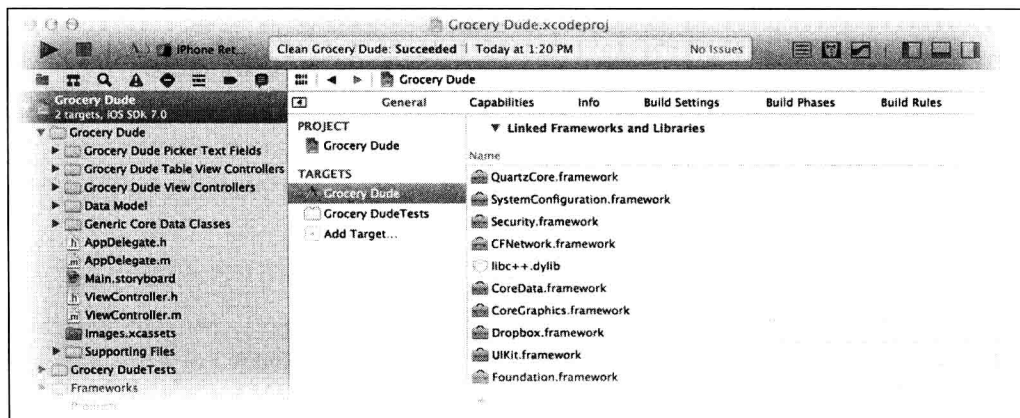


图 13-3 与 Dropbox 集成所需的其他框架

4. 把 **libc++.dylib**、**CFNetwork**、**Security**、**SystemConfiguration** 及 **QuartzCore** 框架添加进来, 如图 13-3 所示。

13.1.2 同 Dropbox 相链接

为了把 Grocery Dude 程序和 Dropbox 集成起来, 我们需要在 AppDelegate 里面添加新的 `application:openURL:` 方法, 并更新原有的 `didFinishLaunchingWithOptions` 方法。Dropbox 网站上有一篇针对 iOS 的 Dropbox Sync API 教程, 其中提供了一段样板代码。程序清单 13-1 列出了相关代码。

程序清单13-1 AppDelegate.m文件

```
#pragma mark - DROPBOX
- (BOOL)application:(UIApplication *)app openURL:(NSURL *)url
    sourceApplication:(NSString *)source annotation:(id)annotation {

    DBAccount *account = [[DBAccountManager sharedManager] handleOpenURL:url];
    if (account) {
```



```

        DBFileSystem *filesystem = [[DBFileSystem alloc] initWithAccount:account];
        [DBFileSystem setSharedFileSystem:filesystem];
        NSLog(@"Linked to Dropbox!");
        return YES;
    }
    return NO;
}

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    if (debug==1) {
        NSLog(@"Running %@", NSStringFromClass(_cmd));
    }

    DBAccountManager* accountMgr =
    [[DBAccountManager alloc] initWithAppKey:@"APP_KEY" secret:@"APP_SECRET"];
    [DBAccountManager setSharedManager:accountMgr];
    DBAccount *account = accountMgr.linkedAccount;
    if (account) {
        DBFileSystem *filesystem = [[DBFileSystem alloc] initWithAccount:account];
        [DBFileSystem setSharedFileSystem:filesystem];
    }
    return YES;
}

```

请按下列步骤修改 Grocery Dude 项目，令其与 Dropbox 集成起来：

1. 把 #import <Dropbox/Dropbox.h> 语句添加到 AppDelegate.m 文件开头。
2. 用程序清单 13-1 中的代码（也包括新的 application:openURL 方法）把 AppDelegate.m 文件里原有的 didFinishLaunchingWithOptions 方法替换掉。
3. 用早前提到过的 app key 与 app secret 来替换 didFinishLaunchingWithOptions 方法中的 APP_KEY 与 APP_SECRET。

接下来需要修改 Grocery Dude 项目的 “information property list”（信息特性列表），以便注册 URL scheme，Dropbox 的验证工作需要用到它。URL scheme 的前缀是 db-，后面是开发者自己的 APP_KEY，如程序清单 13-2 所示。

程序清单13-2 Grocery Dude-Info.plist

```

<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>db-APP_KEY</string>
        </array>
    </dict>
</array>

```

请按下列步骤修改 Grocery Dude 项目，以注册 Dropbox 验证所需的 URL scheme：

1. 在 **Supporting Files/Grocery Dude-Info.plist** 上面点击鼠标右键，然后选择 **Open As > Source Code**。

2. 把程序清单 13-2 中的代码粘贴到 Grocery Dude-Info.plist 文件里首个 `<dict>` 标签的下一行。

3. 把 Grocery Dude-Info.plist 文件中的 APP_KEY 换成早前提到过的 app key，记得保留 db- 前缀。

13.1.3 创建 DropboxHelper 类

为了使开发者能够轻松地给自己的应用程序添加 Dropbox 支持，我们需要创建名叫 DropboxHelper 的类，并把集成 Dropbox 所要用到的绝大部分代码都放在该类之中。这个类里面有一些便捷方法，它们能够完成下列事项：

- ❑ 链接到 Dropbox 账号；与 Dropbox 账号解除链接
- ❑ 在 Dropbox 缓存与本地文件系统之间管理文件
- ❑ 把数据备份到 ZIP 文件；从 ZIP 文件里恢复数据

一开始，DropboxHelper 头文件会如程序清单 13-3 所示。这里面有两个方法，一个用来链接账号，另一个用来解除链接。链接 Dropbox 账号所用的那个方法还需要有个视图控制器，以便使系统能够从这个视图控制器转向用户认证界面。

程序清单13-3 DropboxHelper.h

```
#import <Foundation/Foundation.h>
#import <Dropbox/Dropbox.h>
#import "CoreDataHelper.h"

@interface DropboxHelper : NSObject

#pragma mark - DROPBOX ACCOUNT
+ (void)linkToDropboxWithUI:(UIViewController*)controller;
+ (void)unlinkFromDropbox;

@end
```

请按下列步骤修改 Grocery Dude，以添加 DropboxHelper 类：

1. 选定名为 **Grocery Dude** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 NSObject，把 **Class** 名称设为 DropboxHelper，然后点击 **Next** 按钮。
5. 确保 Targets 中的 “Grocery Dude” 处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹中创建类。
6. 用程序清单 13-3 中的代码把 DropboxHelper.h 文件里原有的代码全都替换掉。

把 Grocery Dude 项目与 Dropbox 框架相链接之后，你应该会注意到现在多出来了一些可供使用的 Dropbox 类。这些类的描述信息都能在参考文档里面找到，而参考文档就包含在刚才下载的 SDK 之中。通过 Finder 找到 com.dropbox.Dropbox.docset 文件，用鼠标右击它，选择 **Show Package Contents**（显示套件内容）。然后，循着 Contents/Resources/Documents/Index.html 路径打开文档。早前我们曾在 AppDelegate.m 文件中新编写了一段与 Dropbox 有关的代码，读者可于参考文档里发现其中用到的一些类。

DropboxHelper 首先要用到的两个 Dropbox 类是 DBAccount 与 DBAccountManager。DBAccountManager 可以引用已链接的账号。程序清单 13-4 列出了链接账号及解除链接所用的代码。

程序清单13-4 DropboxHelper.m文件

```
#import "DropboxHelper.h"
@implementation DropboxHelper

#pragma mark - DROPBOX ACCOUNT
+ (void)linkToDropboxWithUI:(UIViewController*)controller {
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (!account.isLinked) {
        NSLog(@"Linking to Dropbox...");
        [[DBAccountManager sharedManager] linkFromController:controller];
    } else {
        NSLog(@"Already linked to Dropbox as %@", account.info.displayName);
    }
}

+ (void)unlinkFromDropbox {
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (account.isLinked) {
        [account unlink];
        NSLog(@"Unlinked from Dropbox");
    }
}

@end
```

请按下列步骤修改 Grocery Dude 项目，以便和 Dropbox 相链接：

1. 用程序清单 13-4 中的代码把 DropboxHelper.m 文件里原有的代码全都替换掉。

13.1.4 创建 DropboxTVC 类

为了创建备份文件并执行数据恢复，我们需要再编写一个新的用户界面。此外，用户应该能把程序同 Dropbox 账号链接起来，而且还应该能解除链接，以便切换账号。为了实现这个新界面，我们需要添加名为 Backups 的新选项卡。这个选项卡里面将包含一个表格视图，用来显示备份文件列表。而这个表格视图是由一个新的类来驱动的，该类名叫 DropboxTVC，它是 UITableViewController 的子类。我们首先用这个类来演示如何

与 Dropbox 账号相链接，然后再扩展其功能，使之具有备份及恢复数据的能力。

最初的 DropboxTVC 的头文件如程序清单 13-5 所示。文件开头有两个特性，程序在加载界面时会设置它们。contents 特性用于保存由 DBFileInfo 对象所构成的数组，该数组用来表示 Dropbox 缓存中的内容。表格视图里将要显示的东西就是由这个数组所决定的。另一个特性叫做加载（loading），表示程序是否正在加载表格视图里面的内容，该特性用来防止重复载入 contents。

程序清单13-5 DropboxTVC.h文件

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"
#import "CoreDataHelper.h"
#import "DropboxHelper.h"

@interface DropboxTVC : UITableViewController
@property (strong, nonatomic) NSMutableArray *contents;
@property (assign, nonatomic) BOOL loading;
@end
```

请按下列步骤修改 Grocery Dude 项目，以添加 DropboxTVC 类：

1. 选中名叫 **Grocery Dude Table View Controllers** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 UITableViewController，把 **Class** 名称设为 DropboxTVC，然后点击 **Next** 按钮。
5. 确保 Targets 中的“Grocery Dude”处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的文件夹中创建类。
6. 用程序清单 13-5 中的代码把 DropboxTVC.h 文件里原有的代码全都替换掉。

为了演示与 Dropbox 账号链接的过程，我们一开始只在 DropboxTVC 的实现文件里面编写一个方法。这个 viewDidLoad 方法先设置前面提到的那两个特性，然后与 Dropbox 相链接。程序清单 13-6 列出了相关代码。

程序清单13-6 DropboxTVC.m文件

```
#import "DropboxTVC.h"
@implementation DropboxTVC

#pragma mark - VIEW
- (void)viewDidLoad {
    _contents = [NSMutableArray new];
    _loading = NO;
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (!account.isLinked) {
        [DropboxHelper linkToDropboxWithUI:self];
    }
}
```

```

    }
    [super viewDidLoad];
}
@end

```

请按下列步骤修改 Grocery Dude 项目，以实现 DropboxTVC：

1. 用程序清单 13-6 中的代码把 DropboxTVC.m 文件里原有的代码全都替换掉。

接下来要添加一些界面元件，它们会用到 DropboxTVC 类。这需要新配置一个 Backups 选项卡，并且需要配置与表格视图控制器相关的 DropboxTVC。Backups 选项卡需要有图标，而 Dropbox 表格视图里面列出的那些备份文件也要使用图标。

请按下列步骤修改 Grocery Dude 项目，以配置与 Dropbox 功能相关的选项卡及表格视图：

1. 选定 **Images.xcassets**。

2. 从 http://www.timroadley.com/LearningCoreData/Icons_DataBackup.zip 下载 zip 文件，并将其中的图像解压缩，然后把它们拖放到名为 **Images.xcassets** 的 asset catalog 里面。

3. 选中 **Main.storyboard**。

4. 向故事板里拖放一个新的 **Table View Controller**，然后把它和下边原有的 **Navigation Controller-Shop** 对齐。

5. 确保这个新的 **Table View Controller** 处于选中状态，然后点击 **Editor > Embed In > Navigation Controller**。

6. 按住 **Control** 键不放，从 **Tab Bar Controller** 的中心向新的 **Navigation Controller** 拖一条直线，然后从弹出式菜单里选择 **Relationship Segue > view controllers**。

7. 通过 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面），把新 Navigation 控制器的 **Bar Item Title** 设为 **Backups**。

8. 把新 Navigation 控制器的 **Bar Item Image** 设为 **data**。设置好的效果应该如图 13-4 所示。

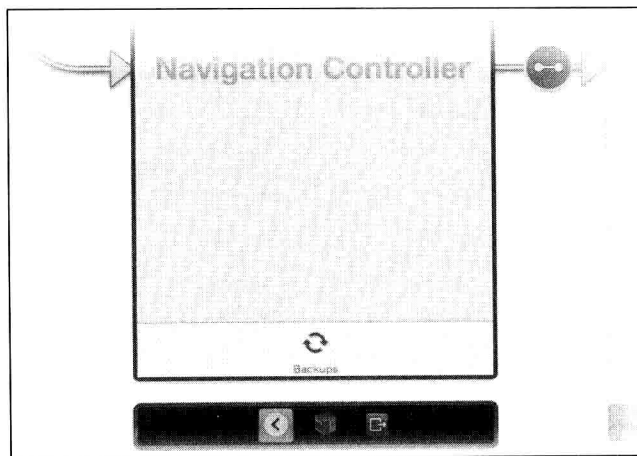


图 13-4 新创建的 Backups tab

9. 把新表格视图控制器的 **Navigation Item Title** 设为 **Backups**。

10. 通过 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面），把 **Backups** 表格视图控制器中 **Prototype Cell** 的 **Identifier** 设为 **Backup Cell**。

11. 通过 **Identity Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面），将 **Backups** 表格视图控制器的 **Custom Class** 设为 **DropboxTVC**。操作时请小心，不要把 **UITableViewCell** 的自定义类给改了。

12. 从设备或 iOS 仿真器里面删掉应用程序，这样一来，下次运行程序时，持久化存储区里就会出现默认的数据了。然后，点击 Xcode 的 **Product > Clean** 菜单项。

13. 运行应用程序，选中 **Backups** 选项卡。你应该会看到如图 13-5 所示的 Dropbox 认证界面。

输入 Dropbox 账号及密码之后，控制台日志里应该会出现写有“**Linked to Dropbox!**”字样的信息。假如控制台日志里提示了有关 URL scheme 的错误，那么请核查 app key 及 app secret 设置得是否正确。



图 13-5 与 Dropbox 链接

13.2 在 CoreDataHelper 类中准备相关代码

稍后我们将进一步开发 **DropboxHelper** 类，以便实现备份与恢复功能，而在这之前，先要增强 **Core-DataHelper** 类，使其可以重新加载 store。恢复数据时，需要把整个存储区的路径都替换掉。替换的时候，应该把所有上下文都清空，并把旧的存储区文件及其底层路径移除。然后像往常一样，通过 **setupCoreData** 重新设置 Core Data 栈。我们需要在 **CoreDataHelper** 类中添加新的方法，以实现此操作。程序清单 13-7 列出了相关代码。

程序清单13-7 CoreDataHelper.m文件中的CORE DATA RESET部分

```
#pragma mark - CORE DATA RESET
- (void)resetContext:(NSManagedObjectContext*)moc {
    [moc performBlockAndWait:^(
        [moc reset];
    )];
}

- (BOOL)reloadStore {
    BOOL success = NO;
    NSError *error = nil;
    if (![coordinator removePersistentStore:_store error:&error]) {
        NSLog(@"Unable to remove persistent store : %@", error);
    }
    [self resetContext:_sourceContext];
    [self resetContext:_importContext];
}
```

```

    [self resetContext:_context];
    [self resetContext:_parentContext];
    _store = nil;
    [self setupCoreData];
    [self somethingChanged];
    if (_store) {success = YES;}
    return success;
}

```

请按下列步骤修改 Grocery Dude 项目，以便在 CoreDataHelper 类中准备好相关的代码：

1. 把程序清单 13-7 中的代码添加到 CoreDataHelper.m 文件底部的 @end 语句上方。
2. 把 -(BOOL)reloadStore; 语句添加到 CoreDataHelper.h 文件底部的 @end 语句上方。

13.3 构建 DropboxHelper 类

创建备份文件以及执行数据恢复所需的大部分操作都与文件管理有关。尤其需要用到的是创建 ZIP 文件及在本地文件系统与 Dropbox 之间移动文件这两项能力。DropboxHelper 类应该尽可能多地提供一些便捷方法，使开发者可以把数据备份及恢复功能轻松地移植到其他应用程序之中。为了实现备份及恢复功能，我们需要在 DropboxHelper 里面编写三个新的部分。

13.3.1 本地文件管理

LOCAL FILE MANAGEMENT 部分包含下列三个方法，以供程序在备份及恢复数据的过程中使用：

- ❑ **renameLastPathComponentOfURL** 这个方法会根据给定的 URL 来修改相关文件的名称。在实际编写该方法时，我们通过移动文件来实现重命名的效果。
- ❑ **deleteFileAtURL** 假如给定的 URL 处确实有某个文件，那么该方法就会把此文件删掉。
- ❑ **createParentFolderForFile** 在解压缩 ZIP 文件之前，程序可通过该方法来确保解压缩操作的目标文件夹已经准备好了。

程序清单 13-8 列出了这个新部分中的代码。

程序清单 13-8 DropboxHelper.m 文件中的 LOCAL FILE MANAGEMENT 部分

```

#pragma mark - LOCAL FILE MANAGEMENT
+ (NSURL*)renameLastPathComponentOfURL: (NSURL*)url toName: (NSString*)name {

    NSURL *urlPath = [url URLByDeletingLastPathComponent];

```

```

NSURL *newURL = [urlPath URLByAppendingPathComponent:name];
NSError *error;
[[NSFileManager defaultManager] moveItemAtPath:url.path
                                toPath:newURL.path error:&error];

if (error) {
    NSLog(@"ERROR renaming (i.e. moving) %@ to %@",
          url.lastPathComponent, newURL.lastPathComponent);
} else {
    NSLog(@"Renamed %@ to %@", url.lastPathComponent, newURL.lastPathComponent);
}
return newURL;
}

+ (BOOL)deleteFileAtURL:(NSURL*)url {

    if ([[NSFileManager defaultManager] fileExistsAtPath:url.path]) {
        NSError *error;
        [[NSFileManager defaultManager] removeItemAtPath:url.path error:&error];
        if (error) {NSLog(@"Error deleting %@", url.lastPathComponent);}
        else {NSLog(@"Deleted %@", url.lastPathComponent);return YES;}
    }
    return NO;
}

+ (void)createParentFolderForFile:(NSURL*)url {

    NSURL *parent = [url URLByDeletingLastPathComponent];
    if (![NSFileManager defaultManager] fileExistsAtPath:parent.path) {
        NSError *error;
        [[NSFileManager defaultManager] createDirectoryAtURL:parent
                                withIntermediateDirectories:YES
                                attributes:nil
                                error:&error];
        if (error) {NSLog(@"Error creating directory: %@", error);}
    }
}

```

请按下列步骤修改 Grocery Dude，以实现这个新的部分：

1. 把程序清单 13-8 中的代码复制到 DropboxHelper.m 文件底部的 @end 语句上方。

13.3.2 Dropbox 文件管理

DROPBOX FILE MANAGEMENT 部分由下列五个方法构成，它们用来管理 Dropbox 缓存内的文件：

- ❑ **fileExistsAtDropboxPath** 方法用来判断 Dropbox 缓存中是否有某个文件放在调用者所指定的路径上。
- ❑ **listFilesAtDropboxPath** 方法会根据给定的 Dropbox 路径，在控制台日志里列出其中的内容。该方法运行在后台线程之中，它唯一的输出行为就是向控制台中写入记录信息，所以，无须担心线程问题。

- ❑ `deleteFileAtDropboxPath` 方法会根据给定的 Dropbox 路径来删除文件。
 - ❑ `copyFileAtDropboxPath:toURL` 方法会把文件从 Dropbox 缓存拷贝到本地文件系统。
 - ❑ `copyFileAtURL:toDropboxPath` 用于将文件从本地文件系统拷贝到 Dropbox 缓存。
- 程序清单 13-9 列出了这个新部分中的代码。

程序清单13-9 DropboxHelper.m文件中的DROPBOX FILE MANAGEMENT部分

```
#pragma mark - DROPBOX FILE MANAGEMENT
+ (BOOL)fileExistsAtDropboxPath:(DBPath*)dropboxPath {

    DBFile *existingFile =
    [[DBFilesystem sharedFilesystem] openFile:dropboxPath error:nil];
    if (existingFile) {
        [existingFile close];
        return YES;
    }
    return NO;
}

+ (void)listFilesAtDropboxPath:(DBPath*)dropboxPath {

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
        NSError *error = nil;
        NSArray *contents =
        [[DBFilesystem sharedFilesystem] listFolder:dropboxPath error:&error];
        if (contents) {
            NSLog(@"***** Dropbox Directory Contents (path /%@",
                                                           dropboxPath.stringValue);

            for (DBFileInfo *info in contents) {
                float fileSize = info.size;
                NSLog(@"  %@", (%.2fMB)", info.path, fileSize/1024/1024);
            }
            NSLog(@"*****");
            if (error) {
                NSLog(@"ERROR listing Dropbox contents for %@ : %@",
                                                                dropboxPath.stringValue, error);
            }
        } else {
            NSLog(@"Dropbox path '%@' is empty", dropboxPath.stringValue);
        }
    });
}

+ (void)deleteFileAtDropboxPath:(DBPath*)dropboxPath {
    [[DBFilesystem sharedFilesystem] deletePath:dropboxPath error:nil];
}

+ (void)copyFileAtDropboxPath:(DBPath*)dropboxPath toURL:(NSURL*)url {

    DBError *openError = nil;
```

```

DBFile *file = [[DBFilesystem sharedFilesystem] openFile:dropboxPath
                                                    error:&openError];

if (openError) {
    NSLog(@"Error opening file '%@': %@", dropboxPath.stringValue, openError);
}

DBError *readError = nil;
NSData *fileData = [file readData:&readError];
if (readError) {
    NSLog(@"Error reading file '%@': %@", dropboxPath.stringValue, readError);
}

[self deleteFileAtURL:url];
[[NSFileManager defaultManager] createFileAtPath:url.path
                                             contents:fileData
                                             attributes:nil];
}

+ (void)copyFileAtURL:(NSURL*)url toDropboxPath:(DBPath*)dropboxPath {

    NSLog(@"Copying %@ to Dropbox Path %@", url, dropboxPath);

    // Create File
    DBError *errorCreating;
    DBFile *file =
    [[DBFilesystem sharedFilesystem] createFile:dropboxPath error:&errorCreating];
    if (!file || errorCreating) {
        NSLog(@"Error creating file in Dropbox: %@", errorCreating);
    }

    // Write File
    DBError *errorWriting;
    if ([file writeContentsOfFile:url.path shouldSteal:NO error:&errorWriting]) {
        NSLog(@"Successfully copied %@ to Dropbox:%@",
              url.lastPathComponent, dropboxPath.stringValue);
    } else {
        NSLog(@"Error writing file to Dropbox: %@", errorWriting);
    }
}

```

请按下列步骤修改 Grocery Dude 项目，以实现这个新的部分：

1. 把程序清单 13-9 中的代码复制到 DropboxHelper.m 文件底部的 @end 语句上方。

13.3.3 备份与恢复

BACKUP/RESTORE 部分包含下列三个新方法，它们会在执行备份或恢复操作时互相配合。这三个方法要用到刚才实现好的那些方法。

- ❑ **zipFolderAtURL** 方法用来创建 ZIP 文件，这个文件里面包含了给定的 URL 中的内容。假如原来已经有同名的 ZIP 文件，那么新文件就会把旧文件覆盖掉。ZIP 文件的压缩是由 Objective-Zip 提供的，它是个针对 MiniZip 的 Objective-C 包装器

(wrapper)。MiniZip 可以创建及解压缩 .zip 文档。

- ❑ **unzipFileAtURL** 方法会根据给定的 URL 来解压缩 ZIP 文件中的内容。解压缩操作也使用 Objective-Zip 来做。
- ❑ **restoreFromDropboxStoresZip** 方法会根据给定的 ZIP 文件内容来恢复 Stores 文件夹。假如恢复操作失败了，那么该方法就需要采取一定的措施来执行回滚 (rollback)，以便使 Core Data 栈能够正常运作。此方法要知道应用程序的 stores 目录在何处。所以，我们得公开 CoreDataHelper 类的 applicationStoresDirectory 方法。

程序清单 13-10 列出了这个新部分中的代码。

程序清单13-10 DropboxHelper.m文件中的BACKUP/RESTORE部分

```
#pragma mark - BACKUP / RESTORE
+ (NSURL*)zipFolderAtURL:(NSURL*)url withZipfileName:(NSString*)zipFileName {

    NSURL *zipFileURL =
        [[url URLByDeletingLastPathComponent] URLByAppendingPathComponent:zipFileName];

    // Remove existing zip
    [self deleteFileAtURL:zipFileURL];

    // Create new zip
    ZipFile *zipFile =
        [[ZipFile alloc] initWithFileName:zipFileURL.path mode:ZipFileModeCreate];

    // Enumerate directory structure
    NSFileManager *fileManager = [[NSFileManager alloc] init];
    NSDirectoryEnumerator *directoryEnumerator =
        [fileManager enumeratorAtPath:url.path];

    // Write zip files for each file in the directory structure
    NSString *fileName;
    while (fileName = [directoryEnumerator nextObject]) {
        BOOL directory;
        NSString *filePath = [url.path stringByAppendingPathComponent:fileName];
        [fileManager fileExistsAtPath:filePath isDirectory:&directory];
        if (!directory) {

            // get file attributes
            NSError *error = nil;
            NSDictionary *attributes =
                [[NSFileManager defaultManager] attributesOfItemAtPath:filePath
                                                         error:&error];

            if (error) {
                NSLog(@"Failed to create zip, could not get file attributes. Error: %@",
                                                              error);

                return nil;
            } else {
```

```

        NSDate *fileDate = [attributes objectForKey:NSFileCreationDate];
        ZipWriteStream *stream =
        [zipFile writeFileInZipWithName:fileName
                    fileDate:fileDate
                    compressionLevel:ZipCompressionLevelBest];
        NSData *data = [NSData dataWithContentsOfFile:filePath];
        [stream writeData:data];
        [stream finishedWriting];
    }
}
[zipFile close];

return zipFileURL;
}

+ (void)unzipFileAtURL:(NSURL*)zipFileURL toURL:(NSURL*)unzipURL {
    @autoreleasepool {
        ZipFile *unzipFile = [[ZipFile alloc] initWithFileName:zipFileURL.path
                                                            mode:ZipFileModeUnzip];
        [unzipFile goToFirstFileInZip];
        for (int i = 0; i < [unzipFile numFilesInZip]; i++) {
            FileInZipInfo *info = [unzipFile getCurrentFileInZipInfo];
            [self createParentFolderForFile:
                [unzipURL URLByAppendingPathComponent:info.name]];
            NSLog(@"Unzipping '%@'...", info.name);
            ZipReadStream *read = [unzipFile readCurrentFileInZip];
            NSMutableData *data = [[NSMutableData alloc] initWithLength:info.length];
            [read readDataWithBuffer:data];
            [data writeToFile:[NSString stringWithFormat:@"%@/%@",
                unzipURL.path, info.name] atomically:YES];
            [read finishedReading];
            [unzipFile goToNextFileInZip];
        }
        [unzipFile close];
    }
}

+ (void)restoreFromDropboxStoresZip:(NSString*) fileName
    withCoreDataHelper:(CoreDataHelper*)cdh {

    [cdh.context performBlock:^(
        DBPath *zipFileInDropbox = [[DBPath alloc] initWithString:fileName];
        NSURL *zipFileInSandbox =
        [[[cdh applicationStoresDirectory] URLByDeletingLastPathComponent]
            URLByAppendingPathComponent:fileName];

        NSURL *unzipFolder =
        [[[cdh applicationStoresDirectory] URLByDeletingLastPathComponent]
            URLByAppendingPathComponent:@"Stores_New"];

        NSURL *oldBackupURL =

```


引入每个 Objective-Zip 类。

2. 把 `#import "Objective-Zip.h"` 语句添加到 `DropboxHelper.m` 文件顶部。

3. 把下列代码添加到 `CoreDataHelper.h` 文件底部的 `@end` 语句上方：

```
- (NSURL *)applicationStoresDirectory;
```

4. 把程序清单 13-10 中的代码复制到 `DropboxHelper.m` 文件底部的 `@end` 语句上方。

为了使外界能够访问 `DropboxHelper` 类的方法，我们需要将其添加到头文件里。程序清单 13-11 列出了相关代码。

程序清单13-11 DropboxHelper.h文件

```
#pragma mark - LOCAL FILE MANAGEMENT
+ (NSURL*)renameLastPathComponentOfURL:(NSURL*)url toName:(NSString*)name;
+ (BOOL)deleteFileAtURL:(NSURL*)url;
+ (void)createParentFolderForFile:(NSURL*)url;

#pragma mark - DROPBOX FILE MANAGEMENT
+ (BOOL)fileExistsAtDropboxPath:(DBPath*)dropboxPath;
+ (void)listFilesAtDropboxPath:(DBPath*)dropboxPath;
+ (void)deleteFileAtDropboxPath:(DBPath*)dropboxPath;
+ (void)copyFileAtDropboxPath:(DBPath*)dropboxPath toURL:(NSURL*)url;
+ (void)copyFileAtURL:(NSURL*)url toDropboxPath:(DBPath*)dropboxPath;

#pragma mark - BACKUP / RESTORE
+ (NSURL*)zipFolderAtURL:(NSURL*)url withZipfileName:(NSString*)zipFileName;
+ (void)unzipFileAtURL:(NSURL*)zipFileURL toURL:(NSURL*)unzipURL;
+ (void)restoreFromDropboxStoresZip:(NSString*)fileName
    withCoreDataHelper:(CoreDataHelper*)cdh;
```

请按下列步骤修改 Grocery Dude 项目，使外界能够看到这些新方法：

1. 把程序清单 13-11 中的代码添加到 `DropboxHelper.h` 文件底部的 `@end` 语句上方。

13.4 构建 DropboxTVC 类

写好 `DropboxHelper` 类之后，现在该用它来实现备份文件的创建、显示及恢复了。我们将会扩充 `DropboxTVC` 类，令其运用 `DropboxHelper` 中的功能。`DropboxTVC` 的头文件里面要添加三个新的特性和两个协议。

- ❑ 为了实现与 Dropbox 账号的链接以及解除链接的功能，我们需要使用名叫 `options` 的动作表。`DropboxTVC` 也将成为 `UIActionSheetDelegate`，以便接收用户做出的选择。
- ❑ 为了确认用户对数据的恢复操作，我们需要添加一个名叫 `confirmRestore` 的警示视图属性。`DropboxTVC` 也将成为 `UIAlertViewDelegate`，以接收用户在警示

视图界面做出的选择。

- ❑ 在恢复数据时，为了保存用户所选定的 ZIP 文件名，我们需要使用名叫 `selectedZipFileName` 的字符串特性。

程序清单 13-12 列出了这三个新的特性。

程序清单13-12 DropboxTVC.h文件

```
@property (strong, nonatomic) UIAlertController *options;
@property (strong, nonatomic) UIAlertView *confirmRestore;
@property (strong, nonatomic) NSString *selectedZipFileName;
```

请按下列步骤修改 Grocery Dude 项目，以添加三项新特性，并采用相关协议：

1. 把程序清单 13-12 中的特性添加到 `DropboxTVC.h` 文件底部的 `@end` 语句上方。
2. 把 `DropboxTVC.h` 文件的 `@interface` 声明替换成下列代码，令其采用 `UIAlertViewDelegate` 协议及 `UIAlertSheetDelegate` 协议：

```
@interface DropboxTVC : UITableViewController
<UIAlertViewDelegate, UIAlertController>
```

Backups 表格视图是由 `DropboxTVC` 驱动的，而这个表格视图里的内容则要根据“本地 Dropbox 缓存”（local Dropbox cache）中的内容来定。`contents` 数组用于填充表格视图，当 Dropbox 缓存有变化时，我们也需要更新这个数组。名叫 `reload` 的新方法会列出 Dropbox 缓存中的内容，并据此填充 `contents` 数组。还有个新的功能叫做 `sort`，它会根据修改日期（modified date）来排列数组里的元素。程序在添加及移除文件时，“Navigation Item Title”里面会显示同步状态。这个状态信息是由名叫 `refreshStatus` 的新方法来更新的，而状态值则取自共享的 `DBFilesystem` 里面的 `status` 特性，请注意，由于我们打算支持“文件夹导航”（folder navigation）功能，所以会把与文件夹有关的内容从 `contents` 数组中移除。程序清单 13-13 列出了相关代码。

程序清单13-13 DropboxTVC.m文件中的DATA部分

```
#pragma mark - DATA
NSInteger sort(DBFileInfo *a, DBFileInfo *b, void *ctx) {
    return [[b modifiedTime] compare:[a modifiedTime]];
}
- (void)reload {
    [self refreshStatus];
    if (!_loading) return; _loading = YES;
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^() {
        NSArray *actualContents =
            [[DBFilesystem sharedFilesystem] listFolder:[DBPath root] error:nil];
        NSMutableArray *updatedContents =
            [NSMutableArray arrayWithArray:actualContents];

        // Don't list folders
        NSMutableArray *folders = [NSMutableArray new];
```

```

    for (DBFileInfo *info in updatedContents) {
        if (info.isFolder) {[folders addObject:info];}
    }
    [updatedContents removeObjectsWithIdentifiers:folders];

    // Don't list files that don't end with 'Stores.zip'
    NSMutableArray *notValid = [NSMutableArray new];
    for (DBFileInfo *info in updatedContents) {
        if (![[[info path] stringValue] hasSuffix:@"Stores.zip"]) {
            NSLog(@"Not listing invalid file: %@", [[info path] stringValue]);
            [notValid addObject:info];
        }
    }
    [updatedContents removeObjectsWithIdentifiers:notValid];

    [updatedContents sortUsingFunction:sort context:NULL];
    dispatch_async(dispatch_get_main_queue(), ^() {
        self.contents = updatedContents;
        _loading = NO;
        [self.tableView reloadData];
        [self refreshStatus];
    });
});
}

- (void)refreshStatus {
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (!account.isLinked) {
        self.navigationItem.title = @"Unlinked";
    } else if ([[[DBFilesystem sharedFilesystem] status] > DBSyncStatusActive]) {
        self.navigationItem.title = @"Syncing";
    } else {
        self.navigationItem.title = @"Backups";
    }
}
}

```

请按下列步骤修改 Grocery Dude 项目，把重新载入 contents 数组所需的代码实现出来：

1. 将程序清单 13-13 中的代码添加到 DropboxTVC.m 文件里现有的 VIEW 部分上方。

接下来在适当的时机调用 reload 方法。每次显示表格视图的时候，我们都应该重新把它加载一遍，使其内容能与底层的 Dropbox 缓存相符。另外，还需要设定一个 observer，如果程序在显示表格视图的时候 Dropbox 缓存里的内容有了变化，那么这个 observer 就可以触发 reload 方法，以执行重新加载操作。为了使用户能够看到同步状态（sync status），我们还需要再添加一个 observer 来监测此状态，并在状态有变化时刷新 DropboxTVC。程序清单 13-14 列出了相关代码。

程序清单 13-14 DropboxTVC.m 文件中的 VIEW 部分

```

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
}

```



```

__block DropboxTVC *DropboxTVC = self;
[[DBFileSystem sharedFileSystem] addObserver:self block:^( ){[self reload];}];
[[DBFileSystem sharedFileSystem] addObserver:self
    forPathAndChildren:[DBPath root] block:^( ){
        [DropboxTVC reload];
    }];
    [DropboxTVC reload];
}
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [[DBFileSystem sharedFileSystem] removeObserver:self];
}

```

请按下列步骤修改 Grocery Dude 项目，编写代码，以便及时触发 reload：

1. 修改 DropboxTVC.m 文件中现有的 VIEW 部分，把程序清单 13-14 中的代码添加到该部分底端。

13.4.1 创建备份文件

写好 DropboxHelper 之后，想创建备份文件就很容易了。在做备份之前，应该把所有上下文都保存起来，以确保最新的数据能够写入持久化存储区。保存上下文时所依照的顺序应该与其在层级中所处的位置相符。也就是说，应该先保存子上下文，然后再保存父上下文，这样能够保证在保存父上下文的时候，会把由子上下文所传过来的数据变更一起保存进去。在开发自己的项目时，你可能会针对保存操作做一些错误处理，而不会像下面将要列出的这段代码一样，给 save 方法传入 nil 参数。

创建备份文件的步骤是：先给 ZIP 备份文件起个合适的文件名，然后创建 ZIP 文件，接着把它移动到 Dropbox 缓存中。此外，还需要用少量的逻辑代码来应对已经有重名文件的情况。用户必须先把程序同 Dropbox 账号相链接，然后才能正常执行这个备份流程。程序会根据备份操作的结果来显示对应的警示视图界面。程序清单 13-15 列出了相关代码。

程序清单13-15 DropboxTVC.m文件中的BACKUP部分

```

#pragma mark - BACKUP
- (IBAction)backup:(id)sender {
    [DropboxHelper linkToDropboxWithUI:self];
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (account.isLinked) {
        CoreDataHelper *cdh =
            [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
        [cdh.context performBlock:^(
            // Save all contexts
            [cdh.sourceContext performBlockAndWait:^( ){[cdh.sourceContext save:nil];}];
            [cdh.importContext performBlockAndWait:^( ){[cdh.importContext save:nil];}];
            [cdh.context performBlockAndWait:^( ){[cdh.context save:nil];}];
            [cdh.parentContext performBlockAndWait:^( ){[cdh.parentContext save:nil];}];

```

```

NSLog(@"Creating a dated backup of the Stores directory...");
NSDateFormatter *formatter = [NSDateFormatter new];
[formatter setDateFormat:@"%Y-MM-dd hh.mm a"];
NSString *date = [formatter stringFromDate:[NSDate date]];
NSString *zipFileName =
[NSString stringWithFormat:@"%s Stores.zip", date];
NSURL *zipFile =
[DropboxHelper zipFolderAtURL:[cdh applicationStoresDirectory]
withZipfileName:zipFileName];

NSLog(@"Copying the backup zip to Dropbox...");
DBPath *zipFileInDropbox =
[[DBPath root] childPath:zipFile.lastPathComponent];
if ([DropboxHelper fileExistsAtDropboxPath:zipFileInDropbox]) {
    NSLog(@"Removing existing backup with same name...");
    [DropboxHelper deleteFileAtDropboxPath:zipFileInDropbox];
}
[DropboxHelper copyFileAtURL:zipFile toDropboxPath:zipFileInDropbox];
NSLog(@"Deleting the local backup zip...");
[DropboxHelper deleteFileAtURL:zipFile];
[DropboxHelper listFilesAtDropboxPath:[DBPath root]];
[self alertSuccess:YES];
}
} else {
    [self alertSuccess:NO];
}
}
- (void)alertSuccess:(BOOL)success {
    NSString *title;
    NSString *message;
    if (success) {
        title = [NSString stringWithFormat:@"Success"];
        message = [NSString stringWithFormat:@"A backup has been created. It will
appear in the Apps/Grocery Dude directory of your Dropbox. Consider removing
old backups when you no longer require them"];
    } else {
        title = [NSString stringWithFormat:@"Fail"];
        message = @"You must be logged in to Dropbox to create backups";
    }
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:title
message:message
delegate:nil
cancelButtonTitle:nil
otherButtonTitles:@"Ok", nil];

    [alert show];
}

```

请按下列步骤修改 Grocery Dude 项目，以编写创建备份文件所需的实现代码，并把这部分代码同新的按钮链接起来：

1. 将程序清单 13-15 中的代码添加到 DropboxTVC.m 文件底部的 @end 语句上方。
2. 选定 **Main.storyboard**。
3. 向 **Backups Table View Controller** 的左上角拖放一个 **Bar Button Item**。
4. 用 **Attributes Inspector** 界面（可以按“Option + ⌘ + 4”组合键调出该界面）把新 **Bar Button Item** 的 **Bar Item Title** 设为 **Create**。

5. 把新 **Bar Button Item** 的 **Bar Item Style** 设为 **Done**。

6. 按住 **Control** 键，从 **Create** 按钮向 **Backups** 表格视图控制器底部的黄圆圈处拖一条直线，然后选择 **Sent Actions > backup:**。

运行应用程序，然后点击 **Backups** tab 里的 **Create** 按钮，以创建备份文件。控制台的记录信息里可能会显示你已经把程序链接到 **Dropbox** 账号了，而且会报告说文件系统里还没有名叫 *thedata_stores.zip* 的文件^①。程序在验证 **Dropbox** 账号及创建备份文件的过程中会执行一些检测，而刚才说的那些信息都是检测时所打印出来的正常消息。执行完备份操作之后，程序界面应该如图 13-6 所示：表格视图目前还是空的，因为我们还没配置它要显示的内容。

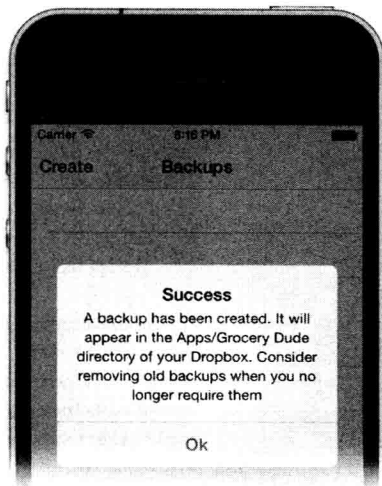


图 13-6 顺利执行完备份操作之后的程序界面

13.4.2 显示备份文件

为了把备份好的文件显示出来，需要修改 **DropboxTVC** 里面与 **UITableView-DataSource** 协议有关的一些方法。这次的表格视图只有一个部分，其行数根据 **contents** 数组里的对象个数来定。我们要配置这个表格视图的单元格，以用户容易看懂的方式来显示备份文件的名称，也就是说，不显示文件名中的“**stores**”字样及“.zip”扩展名。不以“**Stores.zip**”结尾的文件将视为无效文件，我们不会把这种文件显示出来。对于剩下的文件来说，我们要在相关位置显示出文件大小以及上传和下载的进度。为了方便用户操作，我们还添加了 **swipe-to-delete** 功能^②。程序清单 13-16 列出了相关代码。

程序清单13-16 DropboxTVC.m文件中的DATASOURCE:UITableView部分

```
#pragma mark - DATASOURCE: UITableView
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
```

① “*thedata*”指的是当前日期。——译者注

② 意思就是说，用户可以通过 **swipe**（滑动、扫屏）操作来删除相关的内容。——译者注

```

numberOfRowsInSection: (NSInteger)section {
    return [_contents count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath: (NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Backup Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier
            forIndexPath:indexPath];

    DBFileInfo *info = [_contents objectAtIndex:indexPath.row];
    NSString *string = info.path.name;
    cell.textLabel.text =
        [string stringByReplacingOccurrencesOfString:@" Stores.zip" withString:@""];
    float fileSize = info.size;

    NSMutableString *subtitle = [NSMutableString new];

    // Show transfer progress
    DBError *openError = nil;
    DBFile *file = [[DBFilesystem sharedFilesystem] openFile:info.path
        error:&openError];

    if (!file) {
        NSLog(@"Error opening file '%@': %@", info.path.stringValue, openError);
    }

    int progress = [[file status] progress] * 100;
    if (progress != 100) {
        if ([[file status] state] == DBFileStateDownloading) {
            [subtitle appendString:
                [NSString stringWithFormat:@"Downloaded %i%% of ", progress]];
        } else if ([[file status] state] == DBFileStateUploading) {
            [subtitle appendString:
                [NSString stringWithFormat:@"Uploaded %i%% of ", progress]];
        }
    }

    // Show File Size
    [subtitle appendString:
        [NSString stringWithFormat:@"%i.2f Megabytes", fileSize/1024/1024]];
    cell.detailTextLabel.text = subtitle;

    return cell;
}

- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (tableView == self.tableView &&
        editingStyle == UITableViewCellEditingStyleDelete) {
        DBFileInfo *deleteTarget = [_contents objectAtIndex:indexPath.row];
        [DropboxHelper deleteFileAtDropboxPath:deleteTarget.path];
        [_contents removeObjectAtIndex:indexPath.row];
    }
}

```

```

        [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
          withRowAnimation:UITableViewRowAnimationFade];
    }
}

```

请按下列步骤修改 Grocery Dude 项目，使用户可以看到备份好的文件：

1. 把程序清单 13-16 中的代码添加到 DropboxTVC.m 文件底部的 @end 语句上方。
2. 选定 **Main.storyboard**。
3. 用 **Attributes Inspector** 界面（可按“**Option + ⌘ + 4**”组合键调出该界面）把 **Backups Table View** 中 **Prototype Cell** 的 **Style** 设为 **Subtitle**。
4. 把 Backups 表格视图中 Prototype 单元格的 **Image** 设为 **backup**。

再次运行应用程序，这次你会看到备份好的文件已经显示在 Backups 表格视图之中了。请注意，由于该表格视图中的数据是以异步方式填充的，所以用户可能要稍微等待一段时间才会看到最新的数据，尤其是首次验证完 Dropbox 账号的时候。图 13-7 演示了正常的运行效果。

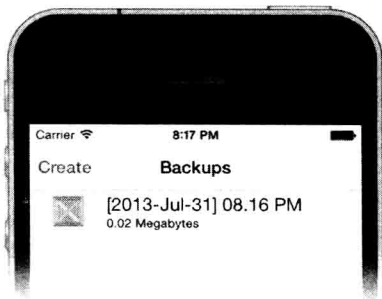


图 13-7 显示在 Backups 表格视图里面的备份文件

13.4.3 恢复数据

为了实现数据恢复功能，我们需要在 DropboxTVC 里面添加四个新方法。由于恢复数据是一种破坏性的操作，所以用户必须点击数个按钮才能启动这个数据恢复流程。首先，用户要选择从哪个文件里恢复数据。其次，用户要点击 **options** 按钮，然后再点击 **restore** 按钮。最后，用户还要确认 **restore** 操作。之所以要多次确认，就是为了防止用户不小心执行了数据恢复。DropboxTVC 类里的四个新方法分别是：

- ❑ **restore** 方法 该方法会设置用户所选定的 ZIP 文件名，并且会显示警示视图界面，令用户确认数据恢复操作。假如文件还在下载、还没有缓存到本地，或是还有比它新的版本，那么程序就会跳过恢复操作，并通知用户。
- ❑ **options** 方法 该方法与一个新的按钮相连，点击了那个按钮之后，程序会显示出一份动作表，用户可以选择“与 Dropbox 账号链接”、“解除链接”或“执行数据恢复”。
- ❑ **actionSheet:clickedButtonAtIndex** 方法 如果用户点击了动作表中的某个按钮，那么该方法就会处理相关事宜，以便执行与 Dropbox 账号链接、解除链接或执行数据恢复等操作。
- ❑ **alertView:clickedButtonAtIndex** 方法 用户对数据恢复操作进行确认之后，该方法就会启动数据恢复流程。

程序清单 13-17 列出了相关代码。

程序清单13-17 ,DropboxTVC.m文件中的RESTORE部分

```
#pragma mark - RESTORE
- (void)restore {
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (!account.isLinked) {[DropboxHelper linkToDropboxWithUI:self];}
    if (account) {
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        if (indexPath) {
            DBFileInfo *info = [_contents objectAtIndex:indexPath.row];

            // Don't restore partially downloaded files
            if (![[[DBFilesystem sharedFilesystem] openFile:info.path
                                                         error:nil] status] cached] ||
                [[DBFilesystem sharedFilesystem] openFile:info.path
                                                         error:nil] newerStatus]
            ) {

                UIAlertView *failAlert = [[UIAlertView alloc]
                                           initWithTitle:@"Failed to Restore"
                                           message:@"The file is not ready"
                                           delegate:nil
                                           cancelButtonTitle:nil
                                           otherButtonTitles:@"Close", nil];

                [failAlert show];
                return;
            }
            _selectedZipFileName = info.path.name;
            NSLog(@"Selected '%@' for restore", _selectedZipFileName);
            NSString *restorePoint =
                [_selectedZipFileName stringByReplacingOccurrencesOfString:@" Stores.zip"
                                                                    withString:@""];

            NSString *message = [NSString stringWithFormat:@"Are you sure want to
➤ restore from %@ backup? Existing data will be lost. The application may pause
➤ for the duration of the restore.", restorePoint];

            _confirmRestore = [[UIAlertView alloc] initWithTitle:nil
                                                            message:message
                                                            delegate:self
                                                            cancelButtonTitle:@"Cancel"
                                                            otherButtonTitles:@"Restore", nil];

            [_confirmRestore show];
        } else {
            UIAlertView *alert =
                [[UIAlertView alloc] initWithTitle:nil
```

```

        message:@"Please select a backup to restore"
        delegate:self
        cancelButtonTitle:@"Ok"
        otherButtonTitles:nil];

[alert show];
    }
}

- (IBAction)options:(id)sender {
    NSString *title, *toggleLink, *restore;
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (account.isLinked) {
        restore = [NSString stringWithFormat:@"Restore Selected Backup"];
        toggleLink = [NSString stringWithFormat:@"Unlink from Dropbox"];
        if (account.info.displayName) {
            title = [NSString stringWithFormat:@"Dropbox: %@",
                                                account.info.displayName];
        } else {
            title = [NSString stringWithFormat:@"Dropbox: Linked"];
        }
    } else {
        toggleLink = [NSString stringWithFormat:@"Link to Dropbox"];
        title = [NSString stringWithFormat:@"Dropbox: Not Linked"];
    }
    _options = [[UIActionSheet alloc] initWithTitle:title
                                                delegate:self
                                                cancelButtonTitle:@"Cancel"
                                                destructiveButtonTitle:nil
                                                otherButtonTitles:toggleLink, restore, nil];
    [_options showFromTabBar:self.navigationController.tabBarController.tabBar];
}

- (void)actionSheet:(UIActionSheet *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex {
    DBAccount *account = [[DBAccountManager sharedManager] linkedAccount];
    if (actionSheet == _options) {
        switch (buttonIndex) {
            case 0:
                if (account.isLinked) {
                    [DropboxHelper unlinkFromDropbox];
                    [self reload];
                } else {
                    [DropboxHelper linkToDropboxWithUI:self];
                    [self reload];
                }
                break;
            case 1:
                [self restore];
                break;
            default:
                break;
        }
    }
}

```

```

    }
}
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (alertView == _confirmRestore && buttonIndex == 1) {
        CoreDataHelper *cdh =
        [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
        [DropboxHelper restoreFromDropboxStoresZip:_selectedZipFileName
        withCoreDataHelper:cdh];
    }
}
}

```

请按下列步骤修改 Grocery Dude 项目，以实现数据恢复功能：

1. 把程序清单 13-17 中的代码添加到 DropboxTVC.m 文件底部的 @end 语句上方。
2. 选定 Main.storyboard。
3. 向 Backups Table View Controller 右上角拖放一个 Bar Button Item。
4. 把新 Bar Button Item 的 Bar Item Title 设为 Options。

5. 按住 Control 键不放，从 Options 按钮向 Backups Table View Controller 底部的黄圆圈处拖一条直线，然后选择 Sent Actions > options:。假如弹出式菜单中没有“options:”这一项，那么就保存 DropboxTVC.m 文件之后重试。

再次运行 Grocery Dude 程序，这次应该可以从备份文件中恢复数据了。记住首先要选中某个备份文件，然后依次点击 Options > Restore Selected Backup > Restore。正常的运行效果如图 13-8 所示。

请注意：备份数据与恢复数据的过程都是在主线程上面执行的，在执行这两个操作的过程中，应用程序会冻结（freeze）。除非存储区特别大，否则整个过程只需几秒钟就能完成，而且程序事先也会告知用户这一点。在实际的开发中，我们可能会像第 3 章那样，暂时阻止用户操作应用程序。假如采用了那种方式，就需要在后台执行备份或恢复操作，并在界面中显示进度条。为了简洁起见，本章把那些步骤省略了。

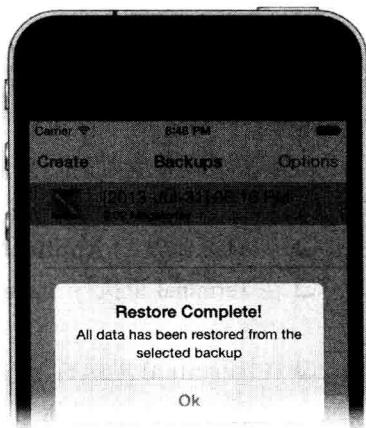


图 13-8 从备份文件中恢复数据

13.5 小结

大家已经学到了如何将应用程序同 Dropbox 账号相集成，以实现数据备份及数据恢复功能。我们创建了包含数据的备份文件，并把整个 Stores 目录都放到了这个 ZIP 文件中。有了备份及恢复功能之后，用户就会觉得数据更安全了。把重要数据存储在他们自己的设备上会使他们觉得更舒服。如果创建了备份文件，那么即便设备失窃，用户也无须担心数

据会丢失。虽说范例程序的备份与恢复功能似乎实现得比较简单，但要想在不同的设备之间传输同一个应用程序的数据，这确实是个好办法。

13.6 习题

请根据所学内容完成下列试验：

1. 试着通过 Data tab 的 Options 菜单来解除程序与 Dropbox 账号之间的链接，然后再重新建立链接。由于这么做会清空本地 Dropbox 缓存，所以程序需要花些时间来重新同步数据，具体耗时多少要根据备份文件的数量来定。

2. 修改 CoreDataHelper.m 文件中的 setupCoreData 方法，令该方法导入测试数据，而不是像本章范例程序那样，把默认存储区设为初始存储区。修改完代码之后，把程序从设备里删掉，这样一来，下次启动时就能导入测试数据了。等程序执行完数据导入并自动把缩略图生成好之后，我们将它与 Dropbox 账号相链接，并创建备份文件。这次的 ZIP 文件大约有 19MB，需要花些时间才能上传到 Dropbox。



提示 在压缩图像时，控制台日志里出现“bit length overflow”（位长溢出）的消息是正常的。

3. 登录 Dropbox 网站，从 Apps/Grocery Dude 目录把这个大约 19MB 的备份的 ZIP 文件下载到电脑中。解压该文件并查看其内容。你会发现 Grocery-Dude.sqlite 文件，还会看到与之相关的 WAL 与 SHM 文件。但是，存储在数据库之外的图像却是隐藏的。

4. 请按下列步骤临时启用 Finder 中的隐藏文件查看功能，这样我们就能在解压备份文件之后看到其中的隐藏内容了：

- ❑ 从 Mac 系统的 **Applications > Utilities** 中打开 **Terminal**。
- ❑ 在 **Terminal** 里执行 `defaults write com.apple.Finder AppleShowAllFiles YES` 命令。
- ❑ 在 **Terminal** 里执行 `killall Finder` 命令。
- ❑ 查看第 3 步中解压好的 ZIP 文件的内容。正常结果应该会如图 13-9 所示。
_EXTERNAL_DATA 目录是由 Core Data 所管理的文件夹，其中能够看到保存于 SQLite 存储区之外的图像文件。如果给文件添上 .png 后缀，那就可以看到其中的图像了！
- ❑ 在 **Terminal** 里执行 `defaults write com.apple.Finder AppleShowAllFiles NO` 命令。
- ❑ 在 **Terminal** 里执行 `killall Finder` 命令。

阅读下一章之前，请将第 2 步所做的修改复原。

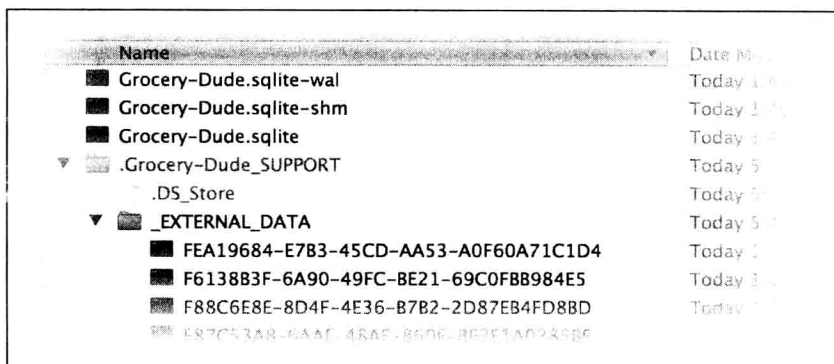


图 13-9 把包含备份数据的 ZIP 文件解压缩之后，可以看到带有 _EXTERNAL_DATA 结构的内容

iCloud

只要不放弃尝试，就永远不会失败。

——阿尔伯特·爱因斯坦

在第 13 章中，我们演示了如何以手动方式利用 Dropbox 把数据文件存储到云端。本章将要讲解怎样利用 iCloud 在用户设备之间实现自动数据同步。把 Core Data 同 iCloud 集成起来之后，用户对应用程序数据所做的修改就会自动反映到其所使用的其他设备上面。笔者编写本书时，Dropbox 及 iCloud 都不支持在多个账号之间同步数据。假如应用程序需要在多个账号之间同步相同的数据，那么可以把它和定制好的 Web 服务集成起来，这样做要比使用 Dropbox 或 iCloud 更为合适。与之类似，假如应用程序要用到有序关系，或是需要在将来的发行版中使用映射模型来迁移数据，那么请注意，iCloud 目前还不支持这两项功能。

本章讲解 Core Data 与 iCloud 相集成的一些基本知识。首先，我们要演示如何启用 iCloud 功能，然后演示如何将 Core Data 与 iCloud 集成起来，对于 Xcode 5 及 iOS 7 开发环境来说，这是个相当简单的过程。集成好之后，我们会介绍 iCloud 里面一些新的调试功能，同时还会给出 iCloud 开发过程中的一些技巧。虽说用户可能已经在使用验证过的 iCloud 账号了，但是本章最后还是要讲一下怎样禁用 iCloud。

14.1 概述

iCloud 可以在用户的多台设备之间同步文档及数据。把 Core Data 同 iCloud 集成起来之后，其数据就遍布各处（ubiquitous）了。“ubiquitous”这个词的意思就是随处可用，而这也正是 iCloud 的基本目标。为了能够把一台设备之中的变更反映到另一台设备，用户需要

在每台设备上面都登入相同的 iCloud 账号。与 iCloud 相结合的应用程序初次运行在某台设备上面时，iCloud 中会形成一份针对该程序的“ubiquitous Core Data baseline”。其他设备将以这个 baseline 为起点，从 iCloud 中把应用程序数据复制并构建到设备里面。拷贝到设备里面的这一份应用程序数据就叫做 iCloud Store，它会自动根据“ubiquity 容器”（ubiquity container）中的变更日志来更新自己，而这个容器由 Apple 的 iCloud 服务器所维护。

把 Core Data 同 iCloud 相集成时，要涉及三个关键组件：

- ❑ Application Stores Directory 它也叫做应用程序沙箱（application sandbox），这是个本地目录，其中含有原始的 Grocery-Dude.sqlite 存储区文件。系统将会把名叫 iCloud.sqlite 的存储区文件添加到该文件夹下的子文件夹里，而每个 iCloud 用户都会有一个对应的子文件夹。Core Data 对每位用户的子文件夹实施透明管理（managed transparently）。大家在本章末尾的习题中会看到文件夹的结构。
- ❑ Ubiquity Container 对于验证过的 iCloud 用户来说，其 iCloud 文档与数据将会出现在这里。该文件夹中的所有内容都自动与 iCloud 服务器相同步。但假如 ubiquity 容器中的某个目录名以 .nosync 结尾，那么它的内容就不会同步。你可能见过有些开发者在实现 iCloud 时，把 iCloud Store 文件放在了 Ubiquity Container 的 .nosync 文件夹下。在 iOS 7 系统中，已经不建议这样做了，因为假如采用这个办法，那么 Core Data 就不能透明地管理 **Fallback Store** 了。Fallback Store 用于在 iCloud 账号间执行无缝迁移，而且还可以减少用户初次使用 iCloud Store 时的等待时间。当用户登出 iCloud 或禁用 iCloud 的文档与数据同步功能时，也需要用到 Fallback Store。
- ❑ iCloud iCloud 是这项服务的名称，它允许用户在自己的多台设备之间同步数据。为了调试程序，你可以在 <https://developer.icloud.com/> 中查看 iCloud 容器里的内容。另外，也可以通过查询元数据或使用由 Xcode 5 所引入的 iCloud Debug Navigator 来检视 iCloud 的内容。本章稍后就会介绍 iCloud Debug Navigator 的用法。在 iCloud 的根目录下，每一个使用 iCloud 的应用程序都会有其子目录，而同一位开发者所开发的应用程序则可以共享某个 ubiquity 容器里的内容。假如要分开维护某程序的免费版与收费版，那么就可借助此特性来访问同一份数据。

图 14-1 概述了把 Core Data 同 iCloud 相集成的过程中所涉及的关键组件，同时也描述了存储区文件（store）及变更日志（change log）的位置。



提示 为了继续构建范例程序，需要把上一章中的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter13.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

为了确保 iCloud 能够正常运作，你必须以有效的开发者身份来配置“Code Signing Identity”（代码签名标识）。请于 Xcode 中选定与 Grocery Dude 应用程序相对应的 target，然后切换到 **General** 分页，并在 **Identity** 区域里配置该选项。

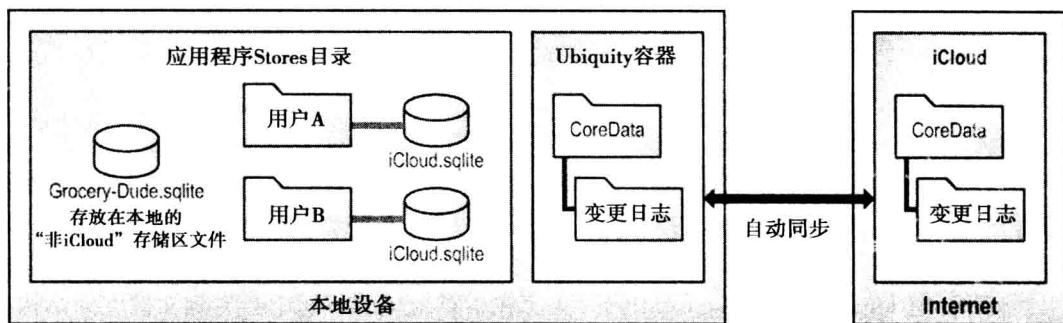


图 14-1 iCloud 概述

14.2 启用 iCloud

为 iOS 7 之前的版本做开发时，必须先有 App ID 及 provisioning profile（配置文件），然后才能启用 iCloud。虽说现在仍然需要这些东西，但是打开 iCloud 开关之后，系统会自动处理此问题。在与应用程序相对应的 target 中，我们可以通过 **Capabilities** 分页来启用 iCloud。

请按下列步骤修改 Grocery Dude，以启用 iCloud 功能：

1. 在名为“Grocery Dude”的 TARGET 中，选择 **Capabilities** 分页，如图 14-2 所示。

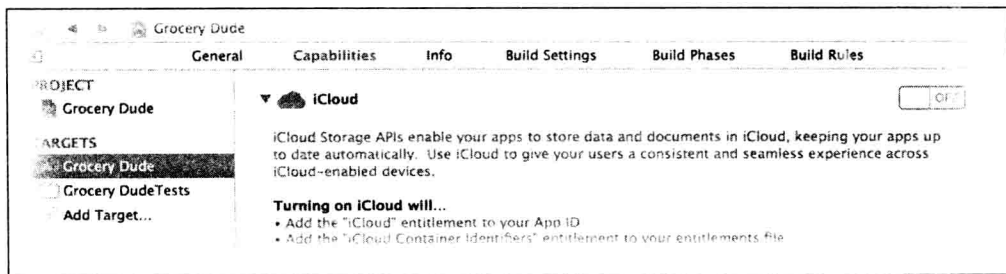


图 14-2 iCloud 功能处于“OFF”（关闭）状态

2. 在连接到互联网的情况下把 iCloud 打开，如图 14-3 所示。

在启用 iCloud 的过程中，需要选择适当的“Development Team”（开发团队）。启用 iCloud 之后，应该会看到自动生成的 Ubiquity Container 名称。只要用户还没有在该位置存储数据，就可以随意改换这个容器的名字。假如需要在程序的免费版与付费版之间保持数据

一致，那么请确认这两种应用程序所使用的 Ubiquity Container 名称相同。

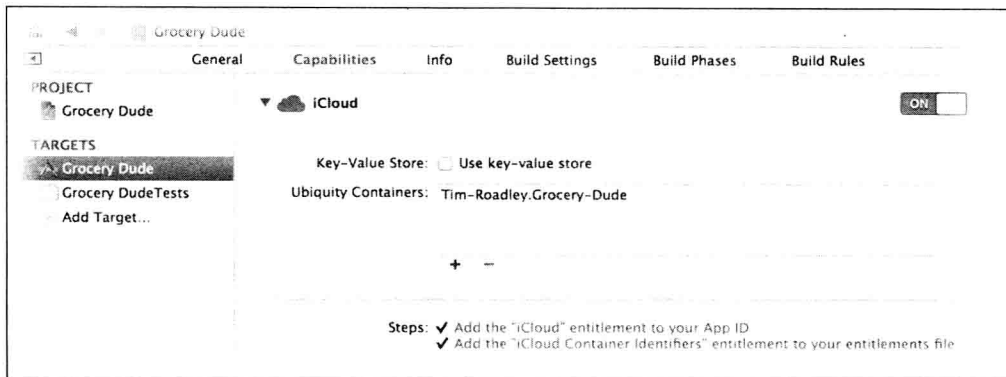


图 14-3 iCloud 功能处于“ON”(开启)状态

14.3 为 CoreDataHelper 类添加 iCloud 功能

为了使大家能够把范例代码更加轻松地套用在自己的项目中，我们现在要修改 CoreDataHelper，给其添加 iCloud 支持。首先需要有个名叫 iCloudAccountIsSignedIn 的新方法，用它来检测 iCloud 账号的状态。只有当 iCloud 账号已获认证时，该方法才会返回 YES。程序清单 14-1 列出了相关代码。

程序清单 14-1 CoreDataHelper.m 文件中的 iCloudAccountIsSignedIn 方法

```
#pragma mark - ICLOUD
- (BOOL)iCloudAccountIsSignedIn {
    if (debug==1) {
        NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
    }

    id token = [[NSFileManager defaultManager] ubiquityIdentityToken];
    if (token) {
        NSLog(@"*** iCloud is SIGNED IN with token '%@'", token);
        return YES;
    }

    NSLog(@"*** iCloud is NOT SIGNED IN ***");
    NSLog(@"--> Is iCloud Documents and Data enabled for a valid iCloud account on
    your Mac & iOS Device or iOS Simulator?");
    NSLog(@"--> Have you enabled the iCloud Capability in the Application Target?");
    NSLog(@"--> Is there a CODE_SIGN_ENTITLEMENTS Xcode warning that needs fixing?
    You may need to specifically choose a developer instead of using Automatic
    selection");
    NSLog(@"--> Are you using a Pre-iOS7 Simulator?");
    return NO;
}
```

请按下列步骤修改 Grocery Dude 项目，以实现对 iCloud 账号状态的检测：

1. 把下列代码添加到 CoreDataHelper.h 文件底部的 @end 语句上方：

```
- (BOOL)iCloudAccountIsSignedIn;
```

2. 把程序清单 14-1 中的代码添加到 CoreDataHelper.m 文件底部的 @end 语句上方。

3. 修改 AppDelegate.m 文件中的 didFinishLaunchingWithOptions 方法，把下列代码添加到该方法底部的 return YES; 语句上方：

```
[[self cdh] iCloudAccountIsSignedIn];
```

4. 确保测试所用的 iOS 设备或 iOS 仿真器已经登录到 iCloud。必须要开启的一项 iCloud 设置就是 **Documents & Data**（文档与数据）。可以通过 **Settings > iCloud** 来核查该选项是否已经开启。

5. 在测试用的设备上运行 Grocery Dude 程序，然后查看控制台日志顶部的内容。正常的运行效果应该如图 14-4 所示，但其中的 token 会与图中有所不同。



图 14-4 在已登录到 iCloud 的设备上运行 Grocery Dude 程序时控制台所输出的信息

14.3.1 iCloud Store

为了保存指向 iCloud Store 的引用，我们需要在 CoreDataHelper 的头文件里添加名叫 iCloudStore 的新特性。程序清单 14-2 列出了相关代码。

程序清单14-2 CoreDataHelper.h文件

```
@property (nonatomic, readonly) NSPersistentStore *iCloudStore;
```

请按下列步骤修改 Grocery Dude 项目，以添加 iCloudStore 特性：

1. 把程序清单 14-2 中的特性添加到 CoreDataHelper.h 文件里现有的 sourceStore 特性声明下方。

对于受过认证的 iCloud 用户来说，系统会根据 ubiquity 容器里面的变更日志来自动生成 iCloud Store 的内容。我们在范例项目中，把 iCloud Store 的文件名设为 iCloud.sqlite，而在自己的项目中，则可以随意指定自己认为合适的名字。

请按下列步骤修改 Grocery Dude 项目，以配置 iCloud Store 的文件名：

1. 修改 CoreDataHelper.m 文件的 FILES 部分，把下列代码添加到该部分底部：

```
NSString *iCloudStoreFilename = @"iCloud.sqlite";
```

系统将会把 iCloud Store 放在应用程序沙箱里的 Store 目录之中。为了方便起见，我们给 CoreDataHelper.m 的 PATHS 部分里面添加一个新方法，用于返回 iCloud Store 的 URL。程序清单 14-3 列出了相关代码。

程序清单 14-3 CoreDataHelper.m 文件中的 iCloudStoreURL 方法

```
- (NSURL *)iCloudStoreURL {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

return [[self applicationStoresDirectory]
        URLByAppendingPathComponent:iCloudStoreFilename];
}
```

请按下列步骤修改 Grocery Dude 项目，以实现 iCloudStoreURL 方法：

1. 修改 CoreDataHelper.m 文件的 PATHS 部分，把程序清单 14-3 中的代码添加到该部分底部。

接下来要实现加载 iCloud Store 所用的方法。程序清单 14-4 列出了相关代码。

程序清单 14-4 CoreDataHelper.m 文件中的 load iCloudStore 方法

```
- (BOOL)load iCloudStore {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

if (!_iCloudStore) {return YES;} // Don't load iCloud store if it's already loaded

NSDictionary *options =
@{
    NSMigratePersistentStoresAutomaticallyOption:@YES
    ,NSInferMappingModelAutomaticallyOption:@YES
    ,NSPersistentStoreUbiquitousContentNameKey:@"Grocery-Dude"
    //,NSPersistentStoreUbiquitousContentURLKey:@"ChangeLogs" // Optional since iOS7
};

NSError *error;
_iCloudStore = [_coordinator addPersistentStoreWithType:NSSQLiteStoreType
                                                configuration:nil
                                                URL:[self iCloudStoreURL]
                                                options:options
                                                error:&error];

if (!_iCloudStore) {
    NSLog(@"*** The iCloud Store has been successfully configured at '%@' ***",
          _iCloudStore.URL.path);

    return YES;
}

NSLog(@"*** FAILED to configure the iCloud Store : %@", error);
return NO;
}
```

iOS 7 极大地简化了 iCloud Store 的加载，使整个过程可以放在主线程上面执行。与 loadStore 方法所采用的方式类似，我们仍然需要向持久化存储协调器里添加持久化存储区。为了能够使用 iCloud，我们这次在存放选项的 NSDictionary 里面多添加一个“键”（key），它的名字叫做 NSPersistentStoreUbiquitousContentNameKey。在把 Core Data 与 iCloud 相集成的过程中，这个 key 是必须要配置的。它的值可以是能够表示 ubiquitous store 的任意字符串。把 Core Data 同 iCloud 集成好之后，请查看 <https://developer.icloud.com/> 中的内容，你会发现这个 key 将用作“变更日志”里的目录名。

针对 iOS 7 之前的系统做开发时，必须指定“变更日志”的位置，而从 iOS 7 开始，它变成可选的 key 了。这个 key 的名字叫做 NSPersistentStoreUbiquitousContentURLKey，只有当开发者需要手动控制 ubiquitous 变更日志的位置时，才需要指定它。比方说，iCloud 应用程序原来曾在 iOS 6 上面创建了一些数据，那么现在就需要用这个 key 来指定数据的位置。

把所有与持久化存储区相关的选项都设置好之后，我们将其传给 addPersistentStore 方法。该方法立刻就会返回一个可供应用程序使用的 Store。返回的这个 Store 实际上是个 Fallback Store，在真正的 iCloud Store 准备好之前，我们可以把它当成 iCloud Store 来操作。

请按照下列步骤修改 Grocery Dude 项目，以实现 loadiCloudStore 方法：

1. 修改 CoreDataHelper.m 文件中的 ICLOUD 部分，把程序清单 14-4 中的方法添加到该部分底部。

14.3.2 与 iCloud 有关的通知

当系统把 iCloud Store 准备好以后，会发送 NSPersistentStoreCoordinatorStoresWillChangeNotification 通知，用来表示存储区即将发生变化。当存储区已经改变时，系统会发送另外一个名叫 NSPersistentStoreCoordinatorStoresDidChangeNotification 的通知。此外，当存储区引入了 ubiquitous content 时，系统会发送 NSPersistentStoreDidImportUbiquitousContentChangesNotification 通知。为了使程序能够做出适当的回应，我们必须监听这三种通知。程序清单 14-5 列出了相关的代码，这段代码写在名叫 listenForStoreChanges 的方法里面，用来配置应用程序，使其能够监听这些关键的通知。

程序清单 14-5 CoreDataHelper.m 文件中的 listenForStoreChanges 方法

```
- (void)listenForStoreChanges {
    if (debug==1) {
        NSLog(@"Running %@", NSStringFromClass(_cmd));
    }

    NSNotificationCenter *dc = [NSNotificationCenter defaultCenter];
    [dc addObserver:self
```

```

        selector:@selector(storesWillChange:)
        name:NSPersistentStoreCoordinatorStoresWillChangeNotification
        object:_coordinator];

[dc addObserver:self
 selector:@selector(storesDidChange:)
 name:NSPersistentStoreCoordinatorStoresDidChangeNotification
 object:_coordinator];

[dc addObserver:self
 selector:@selector(persistentStoreDidImportUbiquitousContentChanges:)
 name:NSPersistentStoreDidImportUbiquitousContentChangesNotification
 object:_coordinator];
}

```

请按下列步骤修改 Grocery Dude，以监听与存储区变更有关的通知：

1. 修改 CoreDataHelper.m 文件中的 ICLOUD 部分，把程序清单 14-5 中的代码添加到该部分底部。Xcode 会警告说这些通知所要触发的“选择子”（selector）还未声明。我们马上就要解决这个问题。

2. 修改 CoreDataHelper.m 文件中的 init 方法，把 [self listenForStoreChanges]; 语句添加到 return self; 上方。

程序清单 14-5 里面提到了三种与存储区变更有关的通知，接下来我们需要实现三个方法，使程序在接到某种通知时，可以调用与该通知相关的方法。程序清单 14-6 列出了它们的代码。

程序清单 14-6 CoreDataHelper.m 文件中的 ICLOUD 部分

```

- (void)storesWillChange:(NSNotification *)n {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

[_importContext performBlockAndWait:^(
    [_importContext save:nil];
    [self resetContext:_importContext];
)];

[_context performBlockAndWait:^(
    [_context save:nil];
    [self resetContext:_context];
)];

[_parentContext performBlockAndWait:^(
    [_parentContext save:nil];
    [self resetContext:_parentContext];
)];

// Refresh UI
[[NSNotificationCenter defaultCenter] postNotificationName:@"SomethingChanged"

```

```

        object:nil
        userInfo:nil];
    }
- (void)storesDidChange:(NSNotification *)n {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
    // Refresh UI
    [[NSNotificationCenter defaultCenter] postNotificationName:@"SomethingChanged"
        object:nil
        userInfo:nil];
}
- (void)persistentStoreDidImportUbiquitousContentChanges:(NSNotification*)n {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
    [_context performBlock:^(
        [_context mergeChangesFromContextDidSaveNotification:n];
        [[NSNotificationCenter defaultCenter] postNotificationName:@"SomethingChanged"
            object:nil];
    )];
}
}

```

在前面的程序清单 14-5 中，有三种与存储区变更有关的通知，而程序清单 14-6 中的这三个方法则分别用来应对这三种通知：

- ❑ 在底层的持久化存储区即将变更之前，程序会调用 `storesWillChange` 方法。当目前的 iCloud 账号有变化的时候，就会出现这种情况。此时底层的存储区必须切换成另外一个存储区，而那个存储区里面将包含新用户的数据。由于老用户将来可能还会再次使用其账号，所以我们应该利用这个机会来保存其数据，并重置每一个上下文。因为 Grocery Dude 程序中的上下文是按层级排布的，所以我们必须依照由子上下文到父上下文的顺序来对每一个上下文执行同步的保存及重置操作。`storesWillChange` 方法必须以同步的方式来操作相关的上下文，因为等该方法返回之后，原有的存储区就与程序分离了。
- ❑ 当底层存储区已经改变之后，系统会调用 `storesDidChange` 方法，我们可以在这个时候刷新用户界面。由于 Grocery Dude 程序里的相关视图已经监听了 `SomethingChanged` 通知，所以我们只需发送该通知，即可令界面刷新。
- ❑ `persistentStoreDidImportUbiquitousContentChanges` 方法用来更新托管对象上下文，使其与 iCloud 中所发生的变化相符。只要底层 iCloud Store 的内容有变化，那么系统就会发出相关通知，以触发该方法，而该方法则会更新驱动用户界面（drive the user interface）的那个 `_context`。

请按下列步骤修改 Grocery Dude 项目，以实现响应 iCloud 通知所用的方法：

1. 修改 `CoreDataHelper.m` 文件中的 `ICLOUD` 部分，把程序清单 14-6 中的代码添

加到该部分底部。

现在需要修改 CoreDataHelper 类的 setupCoreData 方法,使得该方法能够把适当的存储区加载进来。假如无法载入 iCloud Store,那就改为加载本地的存储区。程序清单 14-7 列出了相关代码。

程序清单14-7 CoreDataHelper.m文件中的loadiCloudStore方法

```
if (![self loadiCloudStore]) {
    [self setDefaultDataStoreAsInitialStore];
    [self loadStore];
}
```

请按下列步骤修改 Grocery Dude 项目,确保程序能够把适当的存储区加载进来:

1. 修改 CoreDataHelper.m 文件的 setupCoreData 方法,用程序清单 14-7 中的代码把方法里原有的代码替换掉。
2. 运行应用程序,并查看控制台日志。正常的运行效果应该如图 14-5 所示。



图 14-5 Core Data 应用程序已经与 iCloud 集成好了

从 iOS 7 开始,Core Data 会维护一份 **Fallback Store**。有了 Fallback Store 之后,即便没有网络,用户也能立刻开始使用同 iCloud 相集成的应用程序。现在控制台日志里应该会出现下面两种信息:

- ❑ **Using Local Storage 1** 这种信息表示程序正在使用 Fallback Store,也就是说,必须等首次同步操作完成之后,其他设备才能看到用户的数据。
- ❑ **Using Local Storage 0** 这种信息表示程序已经完成了向 iCloud Store 的迁移,并且不会再使用 Fallback Store 了。如果在控制台里发现了这个消息,那就表明 iCloud 同步功能已经配置完毕,并且能够正常运行了。此时,你会看到数据变更已经反映到了使用相同程序并且开启 iCloud 功能的其他设备上面。

你可能不会马上在控制台里看到写有 Using Local Storage 0 字样的消息,但只要看到 Using Local Storage 1,就可以操作持久化存储区了。等后台进程完全把 iCloud Store 准备好之后,控制台日志里就会出现 Using Local Storage 0 字样的消息。

14.4 Debug Navigator

从 Xcode 5 起，开发者会更容易看到程序里面与 iCloud 相关的活动。运行 iCloud 程序时，我们可以通过 **Debug Navigator** 界面来查看容量、上传情况、下载情况以及每个文档的状态。开发者可由此追踪“变更记录”的处理进度，并且能够很好地了解底层所发生的事情。图 14-6 演示了正在调试 iCloud 程序的 Debug Navigator 界面。

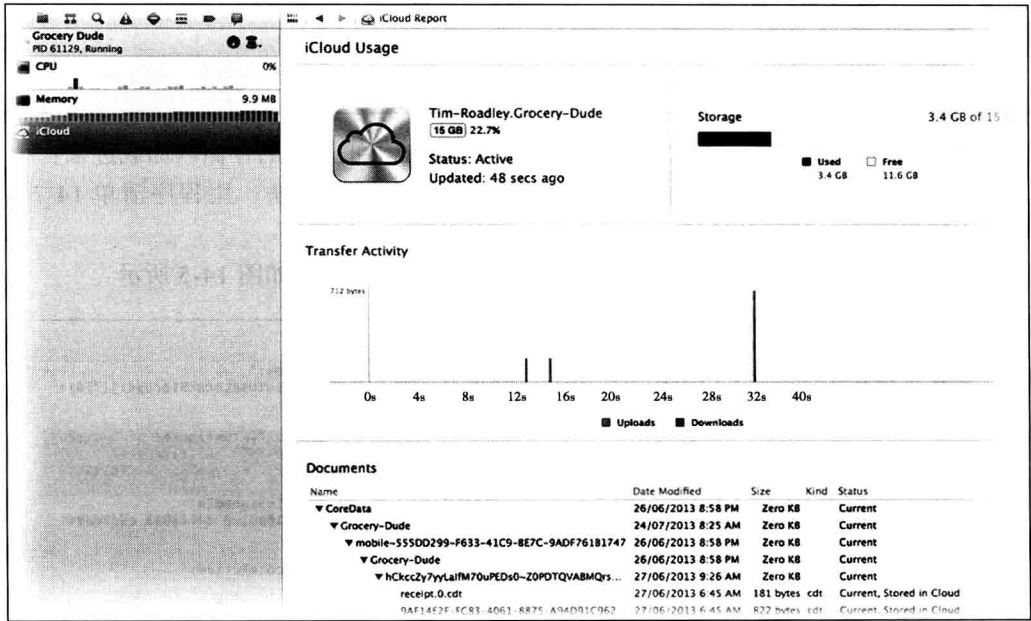


图 14-6 用 Debug Navigator 来调试 iCloud 程序

假如 Debug Navigator 提示说 iCloud 还未配置，那么就请在 Mac 里通过 **System Preferences > iCloud**（系统偏好设置 > iCloud）来激活 **Documents & Data**（文稿与数据）选项。

14.5 禁用 iCloud

尽管 Grocery Dude 程序现在已经支持 iCloud 了，但并不意味着用户就一定想使用该功能。用户或许想要禁用 Grocery Dude 的 iCloud 功能，同时又继续在其他应用程序里使用 iCloud。为了使范例程序更加灵活，我们准备向其中添加包含 `iCloudEnabled` 键的 settings bundle（设置捆绑包，设置束），这样一来，用户就可以自行决定是否在 Grocery Dude 程序中启用 iCloud 了。

请按下列步骤修改 Grocery Dude 项目，以实现 `iCloudEnabled` 设置：

1. 选定名叫 **Grocery Dude** 的组，然后点击 **File > New > File...** 菜单项。

2. 点击 **iOS > Resource > Settings Bundle** 菜单项，然后点击 **Next** 按钮。
3. 确保 Targets 中的 “Grocery Dude” 处于勾选状态，并选定适当的目录，然后点击 **Create** 按钮。
4. 展开 **Settings.bundle**，选中 **Root.plist**。
5. 参照图 14-7 来配置 Root.plist。

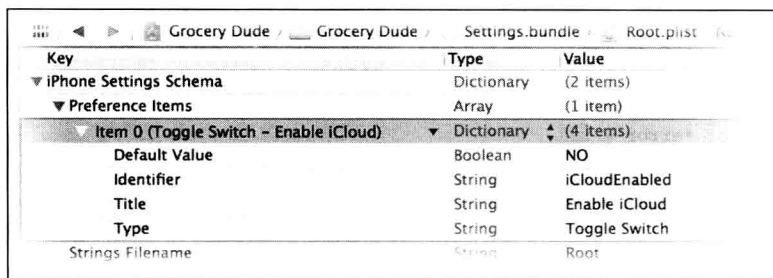


图 14-7 Settings bundle

为了获取 `iCloudEnabled` 键的当前值，我们需要向 `CoreDataHelper.m` 文件里添加名为 `iCloudEnabledByUser` 的新方法。如程序清单 14-8 所示，该方法会获取 `iCloudEnabled` 键的当前值，并返回与之对应的 `BOOL` 值。

程序清单14-8 CoreDataHelper.m文件中的iCloudEnabledByUser方法

```
- (BOOL)iCloudEnabledByUser {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

[[NSUserDefaults standardUserDefaults] synchronize]; // Ensure current value
if ([[NSUserDefaults standardUserDefaults]
    objectForKey:@"iCloudEnabled"] boolValue) {
    NSLog(@"*** iCloud is ENABLED in Settings ***");
    return YES;
}
NSLog(@"*** iCloud is DISABLED in Settings ***");
return NO;
}
```

请按下列步骤修改 Grocery Dude 项目，向其中添加新代码，以检测用户的 iCloud 设置：

1. 修改 `CoreDataHelper.m` 文件中的 `ICLOUD` 部分，把程序清单 14-8 里的代码添加到该部分底部。

在开发自己的应用程序时，你可能想给用户展示一个警示视图界面，询问其是否想使用 iCloud。不过笔者想把本章尽可能写得简洁一些，所以 Grocery Dude 程序的用户只能通过 Settings App 来启用 iCloud 功能。由于该项设置只能在程序切入后台时调整，所以每次程序进入前台时，我们都得判断用户所设置的 iCloud 选项。假如某位受认证的 iCloud 用

户想要禁用 Grocery Dude 的 iCloud 支持，那么就需要重置 Core Data 栈，并且改用“非 iCloud 存储区”（non-iCloud store）来取代 iCloud store。程序清单 14-9 列出了与重置 Core Data 栈有关的代码。

程序清单14-9 CoreDataHelper.m文件中的CORE DATA RESET部分

```

- (void)removeAllStoresFromCoordinator:(NSPersistentStoreCoordinator*)psc {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    for (NSPersistentStore *s in psc.persistentStores) {
        NSError *error = nil;
        if (![psc removePersistentStore:s error:&error]) {
            NSLog(@"Error removing persistent store: %@", error);
        }
    }
}
- (void)resetCoreData {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    [_importContext performBlockAndWait:^(
        [_importContext save:nil];
        [self resetContext:_importContext];
    )];
    [_context performBlockAndWait:^(
        [_context save:nil];
        [self resetContext:_context];
    )];
    [_parentContext performBlockAndWait:^(
        [_parentContext save:nil];
        [self resetContext:_parentContext];
    )];
    [self removeAllStoresFromCoordinator:_coordinator];
    _store = nil;
    _iCloudStore = nil;
}

```

removeAllStoresFromCoordinator 方法会在给定的 NSPersistentStoreCoordinator 里面寻找 NSPersistentStore，并将它们全都移除。resetCoreData 方法先对每个上下文执行保存及重置操作，然后用 removeAllStoresFromCoordinator 方法把 _coordinator 里面的所有存储区全都移除。最后一步是把每个与存储区有关的变量都设为 nil，这样的话，程序就可以重新配置 Core Data 栈了。

请按下列步骤修改 Grocery Dude 项目，以实现 CORE DATA RESET 部分里尚未编写的那部分代码：

1. 修改 CoreDataHelper.m 文件中的 CORE DATA RESET 部分，把程序清单 14-9

中的代码添加到该部分底部。

当应用程序处于活动状态时，我们需要判断它是否已经加载了适当的存储区。程序清单 14-10 列出了名为 `ensureAppropriateStoreIsLoaded` 的新方法，该方法用来完成此任务。

程序清单 14-10 CoreDataHelper.m 文件中的 `ensureAppropriateStoreIsLoaded` 方法

```

- (void)ensureAppropriateStoreIsLoaded {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    if (!_store && !_iCloudStore) {
        return; // If neither store is loaded, skip (usually first launch)
    }
    if (![self iCloudEnabledByUser] && _store) {
        NSLog(@"The Non-iCloud Store is loaded as it should be");
        return;
    }
    if ([self iCloudEnabledByUser] && _iCloudStore) {
        NSLog(@"The iCloud Store is loaded as it should be");
        return;
    }
    NSLog(@"** The user preference on using iCloud with this application appears to
have changed. Core Data will now be reset. **");
    [self resetCoreData];
    [self setupCoreData];

    [[NSNotificationCenter defaultCenter] postNotificationName:@"SomethingChanged"
                                                object:nil];

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
@"Your preference on using iCloud with this application appears to have changed"
                                                message:
@"Content has been updated accordingly"
                                                delegate:nil
                                                cancelButtonTitle:nil
                                                otherButtonTitles:@"Ok", nil];

    [alert show];
}

```

`ensureAppropriateStoreIsLoaded` 方法写得相当直白。如果 `_store` 和 `_iCloudStore` 都没有载入，那么该方法就提前返回。首次启动应用程序时会出现这种情况。如果 iCloud 处于禁用状态而程序已经把非 iCloud 的 `_store` 加载进来了，或是 iCloud 处于启用状态而程序已经把 `_iCloudStore` 加载进来了，那么该方法也会提前返回。假如不是以上三种情况，那就表明 iCloud 选项与程序所加载的存储区不相符，此时我们会对 Core Data 执行“重置”(reset)及“设置”(setup)操作，然后通知相关的视图并告知用户。

请按下列步骤修改 Grocery Dude 项目，确保应用程序在每次进入活动状态时所加载的存储区都是正确的：

1. 修改 CoreDataHelper.m 文件的 ICLOUD 部分，把程序清单 14-10 中的代码添加到该部分底部。

2. 将下列代码放在 CoreDataHelper.h 文件底部的 @end 语句上方：

```
- (void)ensureAppropriateStoreIsLoaded;
```

3. 修改 AppDelegate.m 文件中的 applicationWillEnterForeground 方法，把下列代码添加到该方法底部：

```
[[self cdh] ensureAppropriateStoreIsLoaded];
```

应用程序在进入前台的时候，已经能够判断出它所加载的存储区到底对不对了，尽管如此，我们仍然需要修改 setupCoreData 方法，使程序能够把正确的存储区设置好。究竟应该设置哪个存储区要根据用户的 iCloudEnabled 选项来定。程序清单 14-11 中的代码就是修改后的 setupCoreData 方法。

程序清单14-11 CoreDataHelper.m文件中的setupCoreData方法

```
- (void)setupCoreData {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}
    if (!_store && !_iCloudStore) {
        if ([self iCloudEnabledByUser]) {
            NSLog(@"** Attempting to load the iCloud Store **");
            if ([self loadiCloudStore]) {
                return;
            }
        }
        NSLog(@"** Attempting to load the Local, Non-iCloud Store **");
        [self setDefaultDataStoreAsInitialStore];
        [self loadStore];
    } else {
        NSLog(@"SKIPPED setupCoreData, there's an existing Store:\n ** _store(%@)\n **
        _iCloudStore(%@)", _store, _iCloudStore);
    }
}
}
```

修改后的 setupCoreData 方法首先判断是否已经加载了 _store 或 _iCloudStore。如果两者之一已经载入，那就提前返回；否则就根据用户的 iCloud 选项来加载适当的存储区。假如载入 iCloud Store 的操作失败了，那就改为加载非 iCloud 的存储区。

请按下列步骤修改 Grocery Dude 项目，以实现修改后的 setupCoreData 方法：

1. 用程序清单 14-11 中的方法把 CoreDataHelper.m 文件里原有的 setupCoreData 方法替换掉。

2. 在测试用的设备上运行 Grocery Dude 程序。设备本地的非 iCloud 存储区里应该会填充好默认的数据。
3. 按下 **Home** 键，打开 **Settings App**。
4. 向下滚动，选择“Grocery Dude”这一项，然后按图 14-8 来启用 iCloud。

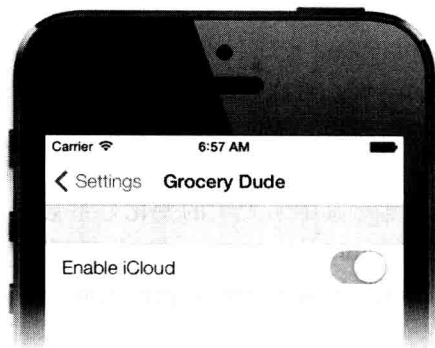


图 14-8 启用 Grocery Dude 程序的 iCloud 功能

5. 返回 Grocery Dude，程序应该会提示数据集已经改变了，如图 14-9 所示。

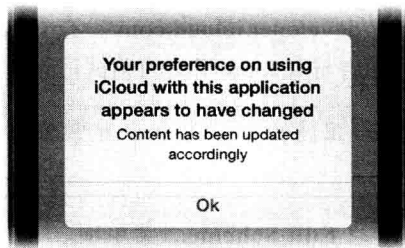


图 14-9 用户修改了 iCloud 选项并返回程序后的效果

有两种禁用 iCloud 的办法，但它们的效果不同，开发者一定要意识到这种区别：

- ❑ 假如通过 **Settings > Grocery Dude > Enable iCloud** 来关闭 iCloud，那么程序会使用本地的非 iCloud 存储区。
- ❑ 假如通过 **iCloud > Documents & Data > Grocery Dude** 来关闭 iCloud，那么程序将会“透明地”(transparently)使用 Fallback Store。

如果程序未在 App Store 上架，或是不需要考虑用户原有的持久化存储区里面的数据，那么就不用添加 Settings.bundle 了，或者可以修改 CoreDataHelper.m 文件里的 setupCoreData 方法，把针对 [self iCloudEnabledByUser] 的检测去掉。去掉了这项检测之后，程序就无法使用非 iCloud 存储区了，而是总会使用 iCloud Store 或 Fallback Store 之一。无论用户开不开启 iCloud 的“Documents & Data”选项，应用程序都能访问其

数据。假如采用了这种方式，那么就不应该使用诸如“Dropbox 备份”等功能，开发者需要把这些功能禁用，我们将在下一章讨论此话题。

14.6 小结

Core Data 程序现在已经和 iCloud 集成起来了，但我们还需要再做一些调整，才能使其行为看上去更加正规。读者会发现，首次同步所耗费的时间各有不同，具体时长取决于待回放的变更日志的状态以及网络连接的速度。由于 iCloud 是在后台同步的，所以一台设备上所发生的变化可能需要花些时间才能反映到其他设备中。大家一定要记住：如果还没有对上下文执行保存操作，那么其中所发生的变化是不会出现在其他设备上面的。下一章要演示“去除重复数据”（de-duplication）以及“数据散播”（seeding）等技巧，这些技巧可以确保程序有良好的用户体验。笔者建议读者先看完下一章，然后再对 iCloud 程序做更为深入的严格测试。

14.7 习题

请根据所学内容完成下列试验：

1. 用开发者账号登录 <https://developer.icloud.com/>，查看 iCloud 里的内容。
2. 请参照下列步骤，试着开启与 iCloud 相关的调试记录：



警告 一旦启用该功能，控制台里输出的信息就会非常繁杂，所以请在试验完毕后将其禁用。

- a. 点击 **Product > Scheme > Edit Scheme...** 菜单项。
 - b. 点击 **Run Grocery Dude**，并切换到 **Arguments** 分页。
 - c. 点击 **Arguments Passed On Launch** 区域中的“+”按钮，以新增参数。
 - d. 输入新参数 **-com.apple.coredata.ubiquity.logLevel 3**，然后点击 OK 按钮。
 - e. 再次运行应用程序，然后观察控制台输出的记录信息。
3. 为了应对多个 iCloud 账号，与 iCloud 相集成的应用程序会在应用程序沙箱内维护一套文件结构。请按下列步骤查看 Grocery Dude 程序沙箱里的内容：
- a. 将 iOS 设备与电脑连接，启用 iCloud 并登入 iCloud 账号，然后运行 Grocery Dude 程序。
 - b. 点击 Xcode 的 **Window > Organizer** 菜单项。
 - c. 选中设备名称下方的 **Applications** 子目录。
 - d. 选定 **Grocery Dude** 程序，然后点击 **Download**，把后缀名为 **.xcappdata** 的文件保存到 **Documents** 目录中。

e. 在 Finder 中找到刚才下载好的文件，用鼠标右键点击它，然后选择 **Show Package Contents**。

f. 参照图 14-10 来展开目录结构。

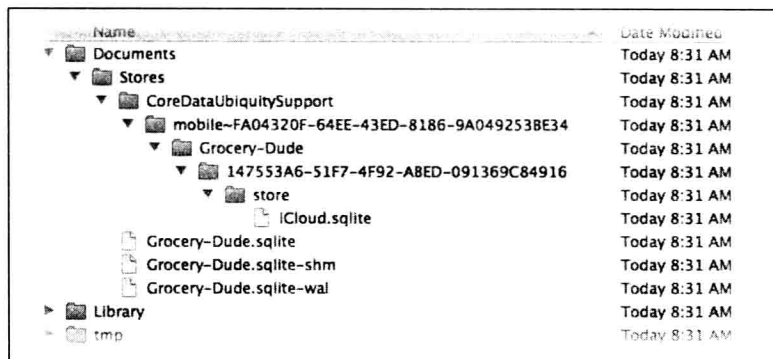


图 14-10 Core Data 应用程序沙箱的目录结构

从图 14-10 中可以看出，iCloud.sqlite 文件的路径与我们在 iCloudStoreURL 方法中指定的路径有所不同。由于系统要平滑地支持多个 iCloud 账号，所以它在原路径里又添加了一些路径。具体的路径会依照是否使用本地存储而有所区别。假如你还有其他的 iCloud 账号，那么请试着以那些账号来使用 Grocery Dude 程序，然后看看系统还会生成哪些存储区和文件夹。

iCloud 高级使用技巧

只有敢发奇想者才能做出奇事。

——阿尔伯特·爱因斯坦

上一章演示了把 Core Data 程序同 iCloud 相集成的基本步骤。为了确保用户能够尽可能顺畅地使用 iCloud，我们还需要解决若干问题。比方说，如果有多个设备同时启用了 iCloud，那么就可能出现某些重复的数据。我们将在完善 Grocery Dude 范例程序的过程中，讨论去除重复数据所用的技术。另外，笔者也会演示如何把本地非 iCloud 持久化存储区中的数据散播到 iCloud 里面。这个办法能够确保用户在初次使用 iCloud 时不会丢失数据，而且即便其所拥有的多台设备在初次使用 iCloud 之前都存放有数据，这些数据也照样不会丢失。

15.1 去除重复数据

在多台设备上使用 iCloud 时，用户可能会创建重复的货品，而在向 iCloud 散播数据时，也有可能出现重复的货品。用户在某设备处于离线状态时所创建的货品必须要等网络连接恢复了之后才会出现在其他设备中，而这尤其容易产生重复数据。此时，假如另一台设备上面也创建了同名的货品，那么就会出现重复。为解决此问题，我们需要新建名叫 `Deduplicator` 的类，用它来有选择地把最旧的重复货品删掉。在编写自己的应用程序时，可能想要再增加一些逻辑代码，用以判断怎样删除重复数据才算合适。比方说，你可能想通过比对 `NSUUID` 这样的属性值来决定删除哪份重复数据，甚至在判定待删除的重复

数据之前，还会考虑到数据间的关系。另外，如果对应用程序合适的话，你还可以自编一段逻辑代码，用来合并相关的值。



提示 为了继续构建范例程序，需要把上一章的代码添加到 Grocery Dude 项目中。或者可以去 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter14.zip> 下载 ZIP 文件，并将其解压缩，这个文件包含了项目到目前为止的全部内容。在使用来自 ZIP 文件的 Xcode 项目时，应该先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

同时，也要配置好 **Code Signing Identity**。请打开 **Grocery Dude target** 的 **Build Settings** 分页，并在 **Code Signing section** 中配置该选项。

如果使用笔者提供的范例代码，那么请先在 Xcode 中关闭 iCloud 功能，然后重新开启，这样就可以使用自己的开发团队了。

请按下列步骤修改 Grocery Dude 项目，以添加 Deduplicator 类：

1. 选中名叫 **Generic Core Data Classes** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 新建 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 **NSObject**，把 **Class** 名称设为 **Deduplicator**，然后点击 **Next** 按钮。
5. 确保 **Targets** 中的“Grocery Dude”处于勾选状态，然后点击 **Create** 按钮，在 Grocery Dude 项目的目录中创建类。

在去除重复对象之前，必须先判断出那些对象是重复的。为了判断某对象是否与其他对象重复，需要选定一个属性，用它来确定对象的唯一性。换句话说，不同对象的这个属性值应该互不相同。具体到 Grocery Dude 项目来说，我们会给每个实体都选定一个 **unique** 属性，用以消除该实体的重复对象。表 15-1 列出了每个实体的 **unique** 属性。

表 15-1 各实体 Unique 属性的名称

| 实 体 | Unique 属性的名称 |
|----------------|--------------|
| Item | name |
| Unit | name |
| LocationAtHome | storedIn |
| LocationAtShop | aisle |

选定了 **unique** 属性之后，我们会根据它把具备每一种值的对象都列出来，并清点其个数。在正常情况下，具备某种 **unique** 属性值的对象只应该有一个才对。假如多个对象都具备了同一种 **unique** 属性值，那就说明数据有重复。程序清单 15-1 里的这段代码可以根据给定的实体来查出有多少个对象的 **unique** 属性值彼此相同，然后把这些重复出现的 **unique** 属

性值放在数组里，返回给调用者。

程序清单15-1 Deduplicator.m文件中的duplicatesForEntityWithName方法

```
+ (NSArray*)duplicatesForEntityWithName:(NSString*)entityName
    withUniqueAttributeName:(NSString*)uniqueAttributeName
    withContext:(NSManagedObjectContext*)context {

    // GET UNIQUE ATTRIBUTE
    NSDictionary *allEntities =
    [[context.persistentStoreCoordinator managedObjectModel] entitiesByName];
    NSAttributedString *uniqueAttribute =
    [[[allEntities objectForKey:entityName] propertiesByName]
     objectForKey:uniqueAttributeName];

    // CREATE COUNT EXPRESSION
    NSEntityDescription *countExpression = [NSEntityDescription new];
    [countExpression setName:@"count"];
    [countExpression setExpression:
    [NSEntityDescription expressionWithFormat:@"count:(%K)", uniqueAttributeName]];
    [countExpression setExpressionResultType:NSInteger64AttributeType];

    // CREATE AN ARRAY OF _UNIQUE_ ATTRIBUTE VALUES
    NSFetchRequest *fetchRequest =
    [[NSFetchRequest alloc] initWithEntityName:entityName];
    [fetchRequest setIncludesPendingChanges:NO];
    [fetchRequest setFetchBatchSize:100];
    [fetchRequest setPropertiesToFetch:
    [NSArray arrayWithObjects:uniqueAttribute, countExpression, nil]];
    [fetchRequest setPropertiesToGroupBy:[NSArray arrayWithObject:uniqueAttribute]];
    [fetchRequest setResultType:NSDictionaryResultType];

    NSError *error;

    NSArray *instances = [context executeFetchRequest:fetchRequest error:&error];
    if (error) {NSLog(@"Fetch Error: %@", error);}

    // RETURN AN ARRAY OF _DUPLICATE_ ATTRIBUTE VALUES
    NSArray *duplicates = [instances filteredArrayUsingPredicate:
    [NSPredicate predicateWithFormat:@"count > 1"]];

    return duplicates;
}
```

由 duplicatesForEntityWithName 方法返回给调用者的那个数组的元素都是重复出现的 unique 属性值。调用该方法时，需要传入实体名称、unique 属性的名称以及该方法所要针对的上下文。为了构造这个数组，首先要创建 NSFetchRequest，令其返回每一种 unique 属性值以及具备这种值的对象个数。只要发现某种 unique 属性值的出现次数多于 1，那就说明出现了重复。我们会通过谓词来过滤刚才查询到的 instances 数组，然后令

该方法把过滤后的数组返回给调用者。

请按下列步骤修改 Grocery Dude 项目，以实现 `duplicatesForEntityWithName` 方法：

1. 把 `#import <CoreData/CoreData.h>` 语句添加到 `Deduplicator.h` 顶部。
2. 把程序清单 15-1 中的代码添加到 `Deduplicator.m` 文件底部的 `@end` 语句上方。

下一步是给 `Deduplicator` 类里面添加名为 `deduplicateEntityWithName` 的新方法，这个新方法用来去除重复的对象。我们将把“去除重复对象”这一过程安排在 `import` 上下文里执行，而这个上下文是运行在后台的。这样做能够保证在处理大量的重复对象时，用户界面不会受影响。而且还有个好处，就是只要对 `import` 上下文执行了保存操作，那么它里面所发生的变化就会立刻反映到主上下文中。之所以有这种效果，是因为我们在第 11 章中为 Grocery Dude 项目构建了上下文层级。

`deduplicateEntityWithName` 方法首先要做的，是采用刚才写好的 `duplicatesForEntityWithName` 方法来获取数组，数组中的元素都是曾经重复出现的 `unique` 属性值。然后，我们用这些值来构建 `NSFetchRequest`，并为其配置 `IN` 谓词。`IN` 谓词可用来限定 `NSFetchRequest`，只有当对象的 `unique` 属性值与列表里的某个值相符时，系统才会把这个对象放入执行结果之中，而那个列表里面包含的都是重复出现的 `unique` 属性值。

执行完 `fetch` 操作之后，会得到一个数组，里面装的都是重复的对象，而程序则要根据这个数组来移除那些重复对象。移除完之后，需要执行清理工作，也就是说，需要保存上下文层级，并且使用已经编写好的 `Faulter` 类，把每个对象都转为 `fault`。程序清单 15-2 列出了相关代码。

程序清单15-2 Deduplicator.m文件中的deduplicateEntityWithName方法

```
+ (void)deduplicateEntityWithName:(NSString*)entityName
    withUniqueAttributeName:(NSString*)uniqueAttributeName
    withImportContext:(NSManagedObjectContext*)importContext {

    [importContext performBlock:^(

        NSArray *duplicates =
        [Deduplicator duplicatesForEntityWithName:entityName
                                withUniqueAttributeName:uniqueAttributeName
                                withContext:importContext];

        // FETCH DUPLICATE OBJECTS
        NSFetchRequest *fetchRequest =
        [[NSFetchRequest alloc] initWithEntityName:entityName];
        NSArray *sortDescriptors =
        [NSArray arrayWithObjects:
            [NSSortDescriptor sortDescriptorWithKey:uniqueAttributeName
                                                ascending:YES], nil];
        [fetchRequest setSortDescriptors:sortDescriptors];
        [fetchRequest setPredicate:[NSPredicate predicateWithFormat:@"%K IN (%@.%K)",
```



```

        uniqueAttributeName, duplicates, uniqueAttributeName]];
[fetchRequest setFetchBatchSize:100];
[fetchRequest setIncludesPendingChanges:NO];
NSError *error;
NSArray *duplicateObjects =
[importContext executeFetchRequest:fetchRequest error:&error];
if (error) {NSLog(@"Fetch Error: %@", error);}

// DELETE DUPLICATES
NSManagedObject *lastObject;
for (NSManagedObject *object in duplicateObjects) {
    if (lastObject) {
        if ([[object valueForKey:uniqueAttributeName]
            isEqual:[lastObject valueForKey:uniqueAttributeName]]) {

            // Add deletion logic here

            // Save & fault objects
            [Faulter faultObjectWithID:object.objectID
                inContext:importContext];
            [Faulter faultObjectWithID:lastObject.objectID
                inContext:importContext];

        }
    }
    lastObject = object;
}
}
};
}

```

请按下列步骤修改 Grocery Dude 项目，以实现 deDuplicateEntityWithName 方法：

1. 把 #import "Faulter.h" 语句添加到 Deduplicator.m 文件顶部。
2. 把程序清单 15-2 中的代码添加到 Deduplicator.m 文件底部的 @end 语句上方。
3. 将程序清单 15-3 里的代码添加到 Deduplicator.h 文件的 @end 语句上方。有了这段代码之后，其他类就可以调用 deDuplicateEntityWithName 方法了。

程序清单15-3 Deduplicator.h文件中的deDuplicateEntityWithName方法


```

+ (void)deDuplicateEntityWithName:(NSString*)entityName
    withUniqueAttributeName:(NSString*)uniqueAttributeName
    withImportContext:(NSManagedObjectContext*)importContext;

```

拟定删除重复对象所依据的标准时，一定要保证在所有包含重复对象的设备中，程序所删掉的都是同一组对象。为实现这一目标，我们要修改 Deduplicator 类，令其比对名为 **modified** 的日期类型的属性。这个属性用来判定哪个对象最旧，而程序则应该把这个最旧的对象删掉。想要防止重复对象，我们还应设计一套备用方案，也就是说，如果两个对象的修改日期相同，那么程序应该把“非 nil”的属性值最少的那个对象删掉。假如依照此条还是无法判断应该删掉哪个对象，那就跳过“去除重复对象”这一操作。

我们需要添加名为 **Model 9** 的新版模型，并把名叫 **modified** 的属性放入其中，这样做可以确保那些已经安装好的应用程序不会因为模型不兼容而崩溃。

 **提示** 假如用户安装了多个版本的 iCloud 应用程序，而这些程序所使用的数据模型又不相同，那么各设备之间就无法执行数据同步。用户必须把每台设备中的程序都升级到最新版，才能够继续同步。开发者在测试自己的程序时，也要注意这一限制。

请按下列步骤修改 Grocery Dude 项目，以实现名叫 **modified** 的属性：

1. 选定 **Model.xcdatamodeld**。
2. 点击 **Editor > Add Model Version...** 菜单项。
3. 点击 **Finish** 按钮，将 **Model 9** 用作版本名称。
4. 选定 **Model 9.xcdatamodel**。
5. 向 **Item** 实体中添加名为 **modified** 的属性，然后将其类型设为 **Date**。
6. 选中刚才添加的 **modified** 属性，在 **Data Model Inspector** 界面（可按“**Option + ⌘ + 3**”组合键调出该界面）中，取消对 **Optional** 的勾选。
7. 把 **modified** 属性的 **Default Value** 设为 1970-01-01 12:00:00+0000。
8. 把 **Item** 实体里名叫 **modified** 的属性拷贝到 **Unit** 实体及 **Location** 实体中。由于 **LocationAtHome** 实体及 **LocationAtShop** 实体均继承自 **Location** 实体，所以无须在这两个实体里创建 **modified** 属性。在某些版本的 Xcode 中，开发者可能需要把 **Editor Style** 切换成 **Table**，然后才能够复制及粘贴属性。
9. 选中 **Model 9** 中的全部实体，然后重新生成 **NSManagedObject** 的子类，并把原有的文件替换掉（可以通过 **Editor > Create NSManagedObject Subclass...** 菜单来完成此操作）。在替换原有的文件之前，别忘记勾选 **Targets** 中的“**Grocery Dude**”。
10. 选中 **Model.xcdatamodeld**，然后用 **File Inspector** 界面（可按“**Option + ⌘ + 1**”组合键调出该界面）把当前模型设为 **Model 9**，如图 15-1 所示。

把新版模型准备好之后，需要调整各种编辑界面，使它们能够访问到相关对象的修改日期。由于每个编辑界面都有指向待编辑对象的指针，所以可以分别修改其 **refreshInterface** 方法，把新代码添加进去。每一种编辑界面所要添加的代码都略有区别，如表 15-2 所示。

表 15-2 为访问相关对象的 **modified** 属性而编写的代码

| 类 文 件 | 代 码 |
|--------------------|---|
| ItemVC.m | <code>item.modified=[NSDate date];</code> |
| UnitVC.m | <code>unit.modified=[NSDate date];</code> |
| LocationAtHomeVC.m | <code>locationAtHome.modified=[NSDate date];</code> |
| LocationAtShopVC.m | <code>locationAtshop.modified=[NSDate date];</code> |

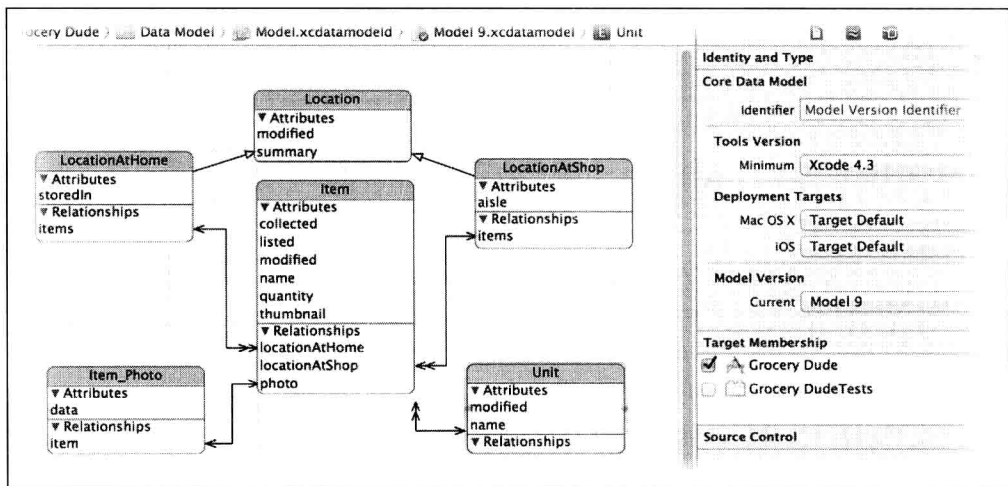


图 15-1 将 Model 9 设为当前模型

请按下列步骤修改 Grocery Dude 项目，使程序能在用户编辑相关对象的时候，把其 modified 属性设置成当前的日期与时间：

1. 修改 ItemVC.m 文件中的 refreshInterface 方法，把 item.modified=[NSDate date]; 语句添加到创建 item 对象的那一行语句下方。
2. 修改 UnitVC.m 文件中的 refreshInterface 方法，把 unit.modified=[NSDate date]; 语句添加到创建 unit 对象的那一行语句下方。
3. 修改 LocationAtHomeVC.m 文件中的 refreshInterface 方法，把 locationAtHome.modified = [NSDate date]; 语句添加到创建 locationAtHome 对象的那一行语句下方。
4. 修改 LocationAtShopVC.m 文件中的 refreshInterface 方法，把 locationAtShop.modified = [NSDate date]; 语句添加到创建 locationAtShop 对象的那一行语句下方。

为了把去除重复对象的逻辑添加到 Deduplicator 类的 deDuplicateEntityWithName 方法里面，我们需要实现保存整个上下文层级的功能。而为了便于实现此功能，需要给 Deduplicator 类里面添加名叫 saveContextHierarchy 的新方法。该方法会沿着上下文的父上下文来遍历上下文层级，并调用沿途每个上下文的 processPendingChanges 及 save: 方法。程序清单 15-4 列出了相关代码。

程序清单15-4 Deduplicator.m文件中的saveContextHierarchy方法

```
#pragma mark - SAVING
+ (void)saveContextHierarchy: (NSManagedObjectContext*)moc {
    [moc performBlockAndWait:^(
        if ([moc hasChanges]) {
```

```

        [moc processPendingChanges];
        NSError *error;
        if (![moc save:&error]) {
            NSLog(@"ERROR Saving: %@",error);
        }
    }
    // Save the parent context, if any.
    if ([moc parentContext]) {
        [self saveContextHierarchy:moc.parentContext];
    }
}
}];
}
}

```

请按照下列步骤修改 Grocery Dude 项目，以实现对上下文层级的保存：

1. 把程序清单 15-4 中的代码添加到 Deduplicator.m 文件底部的 @end 语句上方。

刚才我们已经编写了适当的代码，使得程序会给每个对象都设定修改日期，而现在则要根据这个表示修改日期的属性来实现去除重复对象的代码。程序清单 15-5 列出了相关代码。

程序清单15-5 Deduplicator.m文件中的deDuplicateEntityWithName方法（续）

```

NSLog(@"*** Deleting Duplicate %@ with %@ '%@' ***",

        entityName, uniqueAttributeName, [object valueForKey:uniqueAttributeName]);

// DELETE OLDEST DUPLICATE...
NSDate *date1 = [object valueForKey:@"modified"];
NSDate *date2 = [lastObject valueForKey:@"modified"];

if ([date1 compare:date2] == NSOrderedAscending) {
    [importContext deleteObject:object];
} else if ([date1 compare:date2] == NSOrderedDescending) {
    [importContext deleteObject:lastObject];
}

// ..or.. DELETE DUPLICATE WITH LESS ATTRIBUTE VALUES (if dates match)
else if ([object committedValuesForKeys:nil] count] >
        [[lastObject committedValuesForKeys:nil] count]) {
    [importContext deleteObject:lastObject];
} else {
    NSLog(@"Skipped De-duplication, dates and value counts match");
}

[self saveContextHierarchy:importContext];

```

要想实现去除重复对象的功能，最简单的办法就是把最旧的对象删掉，但这是一种比较武断的逻辑。在某些情况下，这种比较僵化的删除规则可能会令用户感到困扰。比方说，某用户原来创建了名为 apples 的货品，并为其添加了一张好看的照片，但后来该用户忘记了这件事，并且又新建了一个名叫 apples 的货品，于是，前面那件货品里的照片就丢失了。在开发自己的项目时，你可能会考虑再增加一些逻辑代码：假如新对象某个属性的

值是 nil，那么就把旧对象里相关的属性值“合并”过去。简言之，添加这种逻辑代码对 Grocery Dude 项目来说没有太大意义。但由于我们已经把基础架构写好了，所以开发者可以任意添加逻辑代码来实现他们自己所选的策略，以便去除重复的对象。

请按下列步骤修改 Grocery Dude 项目，以实现删除重复对象的逻辑代码：

1. 修改 Deduplicator.m 文件中的 deDuplicateEntityWithName 方法，用程序清单 15-5 里的代码取代 //Add deletion logic here 这行注释。

实现去除重复对象功能的最后一步是在适当的时候来调用 deDuplicateEntityWithName 方法。程序清单 15-6 列出了相关代码。

程序清单15-6 PrepareTVC.m文件中的viewDidAppear方法

```
[cdh.context performBlock:^(
    [Deduplicator deDuplicateEntityWithName:@"Item"
        withUniqueAttributeName:@"name"
        withImportContext:cdh.context];

    [Deduplicator deDuplicateEntityWithName:@"Unit"
        withUniqueAttributeName:@"name"
        withImportContext:cdh.context];

    [Deduplicator deDuplicateEntityWithName:@"LocationAtHome"
        withUniqueAttributeName:@"storedIn"
        withImportContext:cdh.context];

    [Deduplicator deDuplicateEntityWithName:@"LocationAtShop"
        withUniqueAttributeName:@"aisle"
        withImportContext:cdh.context];
});
```

请按下列步骤修改 Grocery Dude 项目，以启用去除重复对象功能：

1. 把 #import "Deduplicator.h" 语句添加到 PrepareTVC.m 文件顶端。

2. 修改 PrepareTVC.m 文件的 viewDidAppear 方法，把程序清单 15-6 中的代码添加到该方法底部。

运行应用程序，新建货品，并把它名字取得和已有的另一件货品相同。此时程序会自动将其中一件货品删掉。正常的运行效果应该如图 15-2 所示。

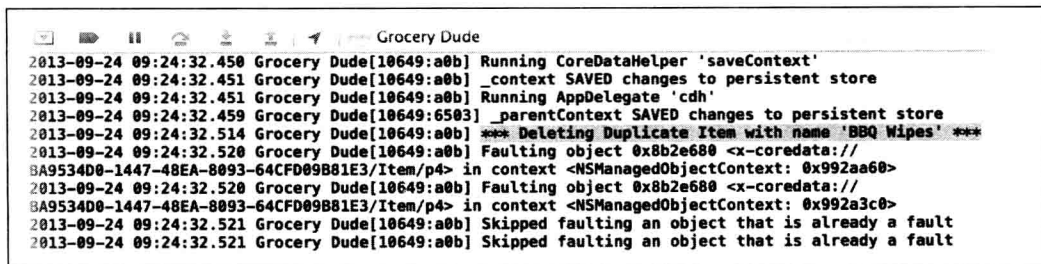


图 15-2 去除重复对象

15.2 散播数据

如果用户在开启 iCloud 功能之前已经开始使用应用程序了,那么可能要把原来的数据迁移到 iCloud。我们将采用第9章实现的深拷贝技术,把 Grocery Dude 程序中的非 iCloud 数据迁移到 iCloud。虽说这种做法比 migratePersistentStore 更耗费 CPU 资源,但是它可以实现以实体为粒度的迁移,并且可以在迁移过程中更新对象里的属性值。由于可以在迁移数据时更新属性值,所以我们能够把 modified 属性设为当天的日期。这样一来,当多台设备都要迁移含有相同日期的默认数据时,去除重复对象的过程会执行得更高效。程序清单 15-7 列出了 CoreData-Importer 类中的一段新代码,这段代码可以在拷贝对象时,把 modified 属性设为当天的日期。

程序清单15-7 CoreDataImporter.m文件中的copyUniqueObject:toContext方法

```
// Update modified date as appropriate
if ([[copiedObject entity] attributesByName] objectForKey:@"modified"]) {
    [copiedObject setValue:[NSDate date] forKey:@"modified"];
}
```

请按下列步骤修改 Grocery Dude 项目,使程序可以在执行深拷贝的时候更新对象的修改日期:

1. 修改 CoreDataImporter.m 文件的 copyUniqueObject:toContext 方法,把程序清单 15-7 中的代码放在 return copiedObject; 语句上方。

数据散播的过程所使用的那个上下文不能是 parentContext 的子上下文,所以不能使用现有的 sourceContext 来做,而是要向 CoreDataHelper 中添加 seedContext 及 seedCoordinator。此外,还要添加新的 seedStore,程序会以只读的方式把非 iCloud 存储区加载进来,并赋给该变量,以便稍后把其中的数据散播(seed)到 iCloud 里面。程序清单 15-8 列出了所要添加的新特性,其中有个新的 UIAlertView,程序通过它来向用户确认是否需要执行散播操作,另外,还有个 BOOL 特性,用来防止程序重复触发数据迁移。

程序清单15-8 CoreDataHelper.h文件

```
@property (nonatomic, readonly) NSManagedObjectContext      *seedContext;
@property (nonatomic, readonly) NSPersistentStoreCoordinator *seedCoordinator;
@property (nonatomic, readonly) NSPersistentStore             *seedStore;
@property (nonatomic, retain) UIAlertView                     *seedalertView;
@property (nonatomic) BOOL                                     seedInProgress;
```

请按下列步骤修改 Grocery Dude 项目,以实现散播操作所需的新特性:

1. 把程序清单 15-8 中的特性添加到 CoreDataHelper.h 文件里。

_seedContext 的配置方式与其他上下文相同,但它并没有父上下文,而是拥有自己

的 `NSPersistentStoreCoordinator`。程序清单 15-9 列出了相关代码。

程序清单15-9 CoreDataHelper.m文件中的init方法

```

_seedCoordinator =
[[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:_model];
_seedContext = [[NSManagedObjectContext alloc]
initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_seedContext performBlockAndWait:^(
    [_seedContext setPersistentStoreCoordinator:_seedCoordinator];
    [_seedContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_seedContext setUndoManager:nil]; // the default on iOS
)];
_seedInProgress = NO;

```

请按下列步骤修改 Grocery Dude，以配置与散播操作相关的特性：

1. 修改 `CoreDataHelper.m` 文件的 `init` 方法，把程序清单 15-9 中的代码添加到 `[self listenForStoreChanges];` 语句上方。

数据散播过程会依赖两个新的辅助方法，我们要把这二者添加到 `CoreDataHelper.m` 文件现有的 `CORE DATA RESET` 部分里面去。第一个方法叫做 `unloadStore`，用来移除给定的存储区，以确保其不再出现。它首先要将给定的存储区从持久化存储协调器里面移除，然后再把该存储区设为 `nil`。

第二个方法叫做 `removeFileAtURL`，数据散播操作执行完毕之后，该方法会把旧的非 iCloud 存储区删掉。为了简单起见，在删除非 iCloud 数据之前，我们就不检查散播操作是否执行成功了。开发自己的应用程序时，你可以在删除旧存储区之前先提示用户，也可以编写一段代码，在删除旧存储区之前先检测散播操作的执行状况。

程序清单 15-10 列出了这两个新方法的代码。

程序清单15-10 CoreDataHelper.m文件中的CORE DATA RESET部分

```

- (BOOL)unloadStore:(NSPersistentStore*)ps {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
if (ps) {
    NSPersistentStoreCoordinator *psc = ps.persistentStoreCoordinator;
    NSError *error = nil;
    if (![psc removePersistentStore:ps error:&error]) {
        NSLog(@"ERROR removing store from the coordinator: %@", error);
        return NO; // Fail
    } else {
        ps = nil;
        return YES; // Reset complete
    }
}
}

```

```

    }
    return YES; // No need to reset, store is nil
}
- (void)removeFileAtURL:(NSURL*)url {
    NSError *error = nil;
    if (![NSFileManager defaultManager] removeItemAtURL:url error:&error) {
        NSLog(@"Failed to delete '%@' from '%@'",
            [url lastPathComponent], [url URLByDeletingLastPathComponent]);
    } else {
        NSLog(@"Deleted '%@' from '%@'",
            [url lastPathComponent], [url URLByDeletingLastPathComponent]);
    }
}
}

```

请按下列步骤修改 Grocery Dude 项目，以便添加两个新方法：

1. 把程序清单 15-10 中的代码添加到 CoreDataHelper.m 文件现有的 CORE DATA RESET 部分底部。

接下来需要添加名为 loadNoniCloudStoreAsSeedStore 的新方法。顾名思义，这个方法负责把非 iCloud 存储区加载为源存储区，然后将其中的数据散播到 iCloud。此方法与 CoreDataHelper.m 文件里现有的 loadStore 方法相似。程序清单 15-11 列出了相关代码。

程序清单15-11 CoreDataHelper.m文件中的loadNoniCloudStoreAsSeedStore方法

```

#pragma mark - ICLOUD SEEDING
- (BOOL)loadNoniCloudStoreAsSeedStore {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    if (_seedInProgress) {
        NSLog(@"Seed already in progress ...");
        return NO;
    }

    if (![self unloadStore:_seedStore]) {
        NSLog(@"Failed to ensure _seedStore was removed prior to migration.");
        return NO;
    }

    if (![self unloadStore:_store]) {
        NSLog(@"Failed to ensure _store was removed prior to migration.");
        return NO;
    }

    NSDictionary *options =
    @{

```



```

        NSReadOnlyPersistentStoreOption:@YES
    };
    NSError *error = nil;
    _seedStore = [_seedCoordinator addPersistentStoreWithType:NSSQLiteStoreType
                                                configuration:nil
                                                URL:[self storeURL]
                                                options:options error:&error];

    if (!_seedStore) {
        NSLog(@"Failed to load Non-iCloud Store as Seed Store. Error: %@", error);
        return NO;
    }
    NSLog(@"Successfully loaded Non-iCloud Store as Seed Store: %@", _seedStore);
    return YES;
}

```

请按下列步骤修改 Grocery Dude 项目，实现新方法，以便加载 _seedStore：

1. 把程序清单 15-11 中的代码添加到 CoreDataHelper.m 文件底部的 @end 语句上方。

接下来需要实现的这个方法负责把散播数据所用的非 iCloud 存储区迁移到 iCloud 之中。mergeNoniCloudDataWithiCloud 方法首先安排一个计时器，令其定期刷新表格视图界面。然后，运行在后台的 seedContext 会把非 iCloud 存储区加载为 seedStore。如果该操作能够顺利执行，那么就把待拷贝的实体名称指定好，并将其传给 CoreDataImporter 实例。然后触发深拷贝操作，等执行完深拷贝之后，将旧的存储区移除。程序清单 15-12 列出了相关代码。

程序清单15-12 CoreDataHelper.m文件中的mergeNoniCloudDataWithiCloud方法

```

- (void)mergeNoniCloudDataWithiCloud {
if (debug==1) {
    NSLog(@"Running %@", '%%', self.class, NSStringFromSelector(_cmd));
}

    _importTimer = [NSTimer scheduledTimerWithTimeInterval:5.0
                                                target:self
                                                selector:@selector(somethingChanged)
                                                userInfo:nil
                                                repeats:YES];

    [_seedContext performBlock:^(

        if ([self loadNoniCloudStoreAsSeedStore]) {

            NSLog(@"*** STARTED DEEP COPY FROM NON-ICLOUD STORE TO ICLOUD STORE ***");
            NSArray *entitiesToCopy = [NSArray arrayWithObjects:
                @"LocationAtHome", @"LocationAtShop", @"Unit", @"Item", nil];
            CoreDataImporter *importer = [[CoreDataImporter alloc]
                initWithUniqueAttributes:[self selectedUniqueAttributes]];
            [importer deepCopyEntities:entitiesToCopy fromContext:_seedContext
                toContext:_importContext];
        }
    )];
}

```

```

[_context performBlock:^(
    // Tell the interface to refresh once import completes
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"SomethingChanged" object:nil];
)];
NSLog(@"*** FINISHED DEEP COPY FROM NON-ICLOUD STORE TO ICLOUD STORE ***");
NSLog(@"*** REMOVING OLD NON-ICLOUD STORE ***");
if ([self unloadStore:_seedStore]) {

    [_context performBlock:^(
        // Tell the interface to refresh once import completes
        [[NSNotificationCenter defaultCenter]
            postNotificationName:@"SomethingChanged"
            object:nil];

        // Remove migrated store
        NSString *wal = [storeFilename stringByAppendingString:@"-wal"];
        NSString *shm = [storeFilename stringByAppendingString:@"-shm"];
        [self removeFileAtURL:[self storeURL]];
        [self removeFileAtURL:[self applicationStoresDirectory
            URLByAppendingPathComponent:wal]];
        [self removeFileAtURL:[self applicationStoresDirectory
            URLByAppendingPathComponent:shm]];

    ]];
}
}
[_context performBlock:^(
    // Stop periodically refreshing the interface
    [_importTimer invalidate];
)];
}];
}
}

```

由于开启了 WAL 日志记录模式，所以在迁移完存储区之后，不仅要删掉该文件本身，而且还得把附带的 wal 与 shm 一并删掉。

请按下列步骤修改 Grocery Dude 项目，以实现向 iCloud 中散播数据所需的代码：

1. 把程序清单 15-12 中的代码添加到 CoreDataHelper.m 文件 ICLOUD SEEDING 部分底部。

程序把 iCloud 存储区加载进来之后，会判断本地是否有数据需要迁移到 iCloud。如果有，那么就询问用户是否愿意将其与 iCloud 合并。我们编写名为 confirmMergeWithiCloud 的新方法来完成此操作，该方法也会把 seedAlertView 显示出来。程序清单 15-13 列出了相关代码。

程序清单15-13 CoreDataHelper.m文件中的confirmMergeWithiCloud方法

```

- (void)confirmMergeWithiCloud {
    if (debug==1) {

```

```

        NSLog(@"Running %@", NSStringFromClass(self.class), NSStringFromSelector(_cmd));
    }
    if (![NSFileManager defaultManager] fileExistsAtPath:[self storeURL path]) {
        NSLog(@"Skipped unnecessary migration of Non-iCloud store to iCloud (there's no
        ➤store file).");
        return;
    }
    _seedAlertView = [[UIAlertView alloc] initWithTitle:@"Merge with iCloud?"
        message:@"This will move your
        ➤existing data into iCloud. If you don't merge now, you can merge later by
        ➤toggling iCloud for this application in Settings."
        delegate:self
        cancelButtonTitle:@"Don't Merge"
        otherButtonTitles:@"Merge", nil];

    [_seedAlertView show];
}

```

请按下列步骤修改 Grocery Dude 项目，实现确认数据合并到 iCloud 的代码：

1. 把程序清单 15-13 中的代码添加到 CoreDataHelper.m 文件的 ICLOUD SEEDING 部分底部。

2. 修改 CoreDataHelper.m 文件中的 alertView:clickedButtonAtIndex 方法，把下列代码添加到该方法底部：

```

if (alertView == self.seedAlertView) {
    if (buttonIndex == alertView.firstOtherButtonIndex) {
        [self mergeNoniCloudDataWithiCloud];
    }
}

```

3. 修改 CoreDataHelper.m 文件中的 loadiCloudStore 方法，把下列语句添加到 return YES; 语句上方：

```
[self confirmMergeWithiCloud];
```

测试数据散播功能所需的全部代码都编好了。现在请按下列步骤测试：

1. 点击 **Product > Clean** 菜单项。
2. 在设备或 iOS 仿真器中运行 **Grocery Dude** 程序。
3. 按 **Home** 键（如果在 iOS 仿真器中运行，那么可以按 “**Shift + ⌘ + H**” 组合键），然后进入 **Settings App**（设置）。
4. 向下滚动，点击 **Grocery Dude**，确保 **Enable iCloud** 选项处于 “off”（关闭）状态。
5. 运行 **Grocery Dude** 程序，创建名叫 **LOCAL ITEM** 的货品。
6. 返回到 **Settings > Grocery Dude**，把 **Enable iCloud** 选项切换到 “on”（开启）。
7. 请确认手机或模拟器上面已经登录了 iCloud 账号。
8. 返回到 Grocery Dude 程序，点击 **Ok** 按钮。然后按照图 15-3 所示，点击 **Merge** 按钮。

数据迁移完之后，设备本地的数据就会出现在 iCloud 里面了，而非 iCloud 存储区将会移除。如果还有其他设备，那么可以在这些设备上面也创建一些可供辨识的本地数据，然后将其迁移到 iCloud。首次导入数据时，程序要执行大量散播操作，稍后还要去除其中的重复对象。若是还有一台设备也在向 iCloud 中填充数据，那么在去除重复对象这一操作完成之前，控制台日志里通常会出现许多写有“Skipped faulting an object that is already a fault”（由于对象已经是 fault 了，所以跳过对它的 fault 操作）字样的消息。只要设备连上网，并且其 ubiquity 容器与 iCloud 一致，那么在启用 iCloud 几分钟之后，就应该能在控制台的日志里面搜索到 Using local storage:0 消息了。

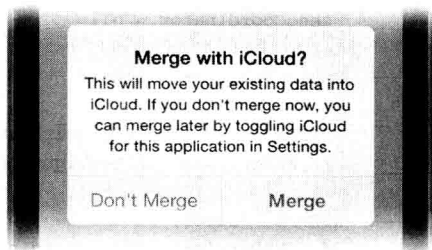


图 15-3 把非 iCloud 数据合并到 iCloud

15.3 打造干净的开发环境

在测试程序的过程中，开发者有时可能想把 iCloud 清空。之所以要清空 iCloud，通常是想模拟用户初次安装应用程序时的情况，以观察程序在这种情况下的行为。若想清空 iCloud，其实并不需要每次都去删除应用程序的 iCloud 目录中的内容，而是可以通过 NSPersistentStoreCoordinator 的类方法 removeUbiquitousContentAndPersistentStoreAtURL 来实现。该方法会同步地把 iCloud 中与特定持久化存储区有关的内容全都删掉，这需要一些时间。当使用同一程序的其他设备上线时，删除操作会传播到那些设备上面。与添加持久化存储区时的做法类似，开发者在调用该方法时，也需要提供 NSDictionary，其中的选项可以帮助系统找到与特定持久化存储区相关联的文件。调用该方法时，一定要确保程序里面没有尚在运行的 NSPersistentStoreCoordinator。程序清单 15-14 列出了相关代码，这些代码写在新的 ICLOUD RESET 部分里面。

程序清单15-14 CoreDataHelper.m文件中的ICLOUD RESET部分

```
#pragma mark - ICLOUD RESET
- (void)destroyAlliCloudDataForThisApplication {

    if (![NSFileManager defaultManager] fileExistsAtPath:[[_iCloudStore URL] path]) {
        NSLog(@"Skipped destroying iCloud content, _iCloudStore.URL is %@",
            [[_iCloudStore URL] path]);
        return;
    }

    NSLog(@"\n\n\n\n\n **** Destroying ALL iCloud content for this application, this
    ➡could take a while... **** \n\n\n\n\n");
```

```

[self removeAllStoresFromCoordinator:_coordinator];
[self removeAllStoresFromCoordinator:_seedCoordinator];
_coordinator = nil;
_seedCoordinator = nil;

NSDictionary *options =
@{
    NSPersistentStoreUbiquitousContentNameKey:@"Grocery-Dude"
    //,NSPersistentStoreUbiquitousContentURLKey:@"ChangeLogs" // Optional since iOS7
};
NSError *error;
if ([NSPersistentStoreCoordinator
    removeUbiquitousContentAndPersistentStoreAtURL:[_iCloudStore URL]
        options:options
        error:&error]) {
    NSLog(@"\n\n\n\n\n");
    NSLog(@"**      This application's iCloud content has been destroyed      **");
    NSLog(@"** On ALL devices, please delete any reference to this application in **");
    NSLog(@"**  Settings > iCloud > Storage & Backup > Manage Storage > Show All  **");
    NSLog(@"*\n\n\n\n\n");
    abort();
}
/*
    The application is force closed to ensure iCloud data is wiped cleanly.
    This method shouldn't be called in a production application.
*/
} else {
    NSLog(@"*\n\n FAILED to destroy iCloud content at URL: %@ Error:%@",
        [_iCloudStore URL],error);
}
}
}

```

destroyAlliCloudDataForThisApplication 方法首先判断给定的路径上面是否有存储区。如果有，那就在每个 coordinator 上面调用 removeAllStoresFromCoordinator 方法，然后创建包含 iCloud 选项的 NSDictionary。调用 removeUbiquitousContentAndPersistentStoreAtURL 方法的时候，需要把这个 options 字典与 iCloud 的 URL 一并传入。该操作执行成功之后，程序会提示开发者把每台设备上的应用程序数据都清除掉。这个时候，如果有必要，可以考虑把整个应用程序都删掉，以便更加准确地模拟出用户首次使用程序时的样子。

请按下列步骤修改 Grocery Dude 项目，以实现重置 iCloud 的功能：

1. 把程序清单 15-14 中的代码添加到 CoreDataHelper.m 文件底端的 @end 语句上方。
2. 修改 CoreDataHelper.m 文件中的 loadiCloudStore 方法，把下列代码添加到 return YES; 这一行之上。程序加载 iCloud 存储区的时候，会触发下面这行代码，它会把每台设备上与该程序有关的数据完全清除掉。

```
[self destroyAlliCloudDataForThisApplication];
```

3. 运行应用程序, 等待其加载并删掉 iCloud 存储区。执行完 `abort()`; 之后, 程序就应该自行终止了。正常的运行效果应该如图 15-4 所示。

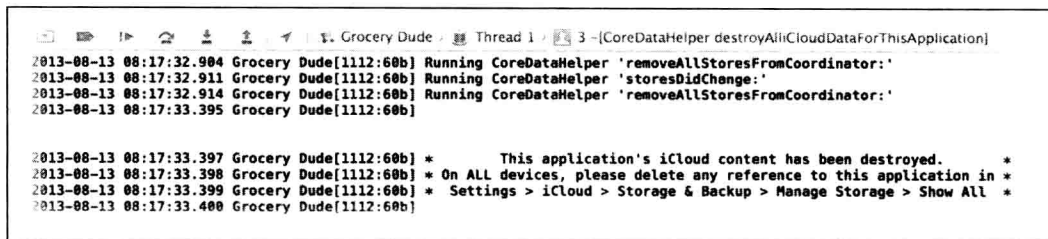


图 15-4 重置 iCloud

4. 在收到了 iCloud 内容已被摧毁的确认通知之后, 请停止应用程序, 并把第 2 步里添加到 `CoreDataHelper.m` 文件的 `loadiCloudStore` 方法中的那行代码注释掉。

5. 通过 **Settings > iCloud > Storage & Backup > Manage Storage > Show All** 来确认所有与该 iCloud 账号相同步的设备上面都没有 Grocery Dude 的数据了。`removeUbiquitousContentAndPersistentStoreAtURL` 方法应该会自行处理好这件事, 但有时候, 删除操作可能尚未传播到某些设备, 所以需要确认一下。

6. 点击 Xcode 的 **Product > Clean** 菜单项。

7. 登录 <https://developer.icloud.com/>, 确认 Grocery Dude 目录下已经没有文件了。



提示 重置 iCloud 数据之后, 如果有设备无法同步, 那么请通过 **Settings > General > Reset All Settings** (设置 > 通用 > Reset) 来重置所有设置。该方法不会删掉应用程序的数据, 而且能够解决与“Error attempting to read ubiquity root url”(试读取 ubiquity root url 时出错) 这条错误消息有关的问题。

15.4 Core Data 程序的配置

虽然 Grocery Dude 项目用不到, 但大家最好还是了解一下 Core Data 应用程序的配置 (configuration)。通过配置, 开发者可以把不同的实体放在不同的存储区里面。假如想把适合存放在 iCloud 中的数据同适合存放在本地设备中的数据分开, 那么就可以使用这个功能。比方说, 静态数据或者与设备相关而且变化迅速的数据 (例如地理位置) 最好能够存放在非 iCloud 存储区里。凡是没有必要复制到其他设备的数据就应该放在本机。我们可以像图 15-5 这样, 通过 Model Editor 来创建配置。

配置的创建方式与实体 (Entity) 或获取请求模板的创建方式相似。开发者无法删除默认的配置, 如果想把实体划分到 iCloud 配置和非 iCloud 配置里面, 那么就需要新建两个配置才行。创建好二者之后, 就可以把每个实体都拖放到适当的配置里面了。

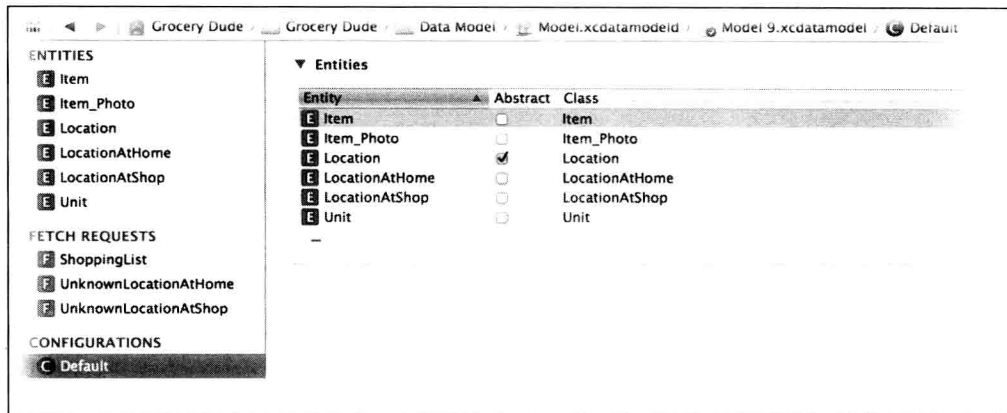


图 15-5 Core Data 程序的配置

要想使用某个配置，可以在向 `NSPersistentStoreCoordinator` 中添加 `NSPersistentStore` 的时候，把配置的名称传给 `addPersistentStoreWithType` 方法。比方说，`loadStore` 方法目前给 `addPersistentStoreWithType` 传递的 `configuration` 的参数值是 `nil`，意思就是使用默认的配置。也可以传入 `nil` 之外的值，以便使用与该名称相对应的配置。

使用配置的时候，一定要注意，不同的数据会划分到不同的存储区里。由于不同的存储区之间是不能建立关系的，所以开发者必须设法解决这一限制。

15.5 收尾工作

由于我们已将 Core Data 程序同 iCloud 相集成，所以在启用 iCloud 之后，Dropbox 的备份与恢复流程就没有明显效果了。这是因为与 Dropbox 相集成的是非 iCloud 存储区，而此时我们使用的却是 iCloud 存储区。实际上，存放在每台设备中的数据只不过是 iCloud 缓存 (iCloud cache) 而已，它并不是 iCloud 服务背后所存放的真实数据。这就意味着即便用户丢失了设备，也不会造成数据损失，因为系统可以从 iCloud 中把存储区重新构建出来。此外，试图从 iCloud 存储区里恢复数据也会导致无法预料的后果，所以，在启用了 iCloud 功能之后，应该把与 Dropbox 相集成的备份与恢复功能关掉。程序清单 15-15 列出了展示警示视图界面所用的代码，这个界面会告诉用户：当 iCloud 功能启用的时候，Dropbox 功能会被禁用。

程序清单 15-15 DropboxTVC.m 文件中的 backup/restore 部分

```
CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];
if ([cdh iCloudEnabledByUser]) {
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Not Supported"
```

```

        message:@"This functionality is disabled because iCloud is enabled"
        delegate:nil
        cancelButtonTitle:nil
        otherButtonTitles:@"OK", nil];
[alert show];
return;
}

```

请按下列步骤修改 Grocery Dude 项目，以便在启用 iCloud 时禁用“备份与恢复”功能：

1. 把下列代码添加到 CoreDataHelper.h 文件底部的 @end 语句上方：

```
- (BOOL)iCloudEnabledByUser;
```

2. 修改 DropboxTVC.m 文件中的 backup 方法，把程序清单 15-15 中的代码添加到该方法顶部。

3. 修改 DropboxTVC.m 文件中的 restore 方法，把程序清单 15-15 中的代码添加到该方法顶部。

4. 在数台设备上面运行 Grocery Dude，并启用“Settings”（设置）里的 iCloud 功能。假如原来曾经重置过模拟器，那么别忘了重新登入 iCloud 账号。你会发现，变更过的数据在保存到一台设备之后，也会出现在其他启用 iCloud 并装有 Grocery Dude 程序的设备上面。

15.6 小结

在实现了本章所讲的这些附加功能之后，我们的 Core Data 程序及其 iCloud 功能就显得更加人性化了。无论是去除重复对象还是数据散播，这些技术都进一步提升了程序的用户体验，使之变得更为流畅。最后要注意的是，即便设备没有登入 iCloud，我们也依然可以加载 iCloud 存储区。这正是 Fallback Store 的妙处，这种存储区运作于程序底层，它并不会自动和非 iCloud 存储区相适配，所以，我们给 Grocery Dude 程序加上了 iCloud Enabled（启用 iCloud）这一选项，使用户可以把 iCloud 存储区和 Fallback Store 一起屏蔽掉。如果想令 CoreDataHelper 使用你自己已经拥有的存储区，那么只需修改 FILES 部分及 PATHS 部分中的相关方法，使其返回指向该存储区的 URL 即可。

15.7 习题

请根据所学内容完成下列试验，用书中讲到的辅助类给全新的应用程序添加 Core Data 与 iCloud 功能。

1. 在 Xcode 里新建针对 iPhone 的 Single View Application，并把项目命名为 EasyiCloud。



提示 如果要把这套流程运用在其他类型的项目上，那么请不要勾选“Use Core Data”这一项。

2. 从 <http://timroadley.com/LearningCoreData/Generic%20Core%20Data%20Classes.zip> 下载 .zip 文件，并将其中的 **Generic Core Data Classes** 文件夹解压缩。

3. 把 **Generic Core Data Classes** 文件夹拖放到 **EasyiCloud** 项目的 **EasyiCloud** 组里面。请确认 **Copy items into destination group's folder** 选项及 Targets 中的 “EasyiCloud” 处于勾选状态，然后点击 **Finish** 按钮。

4. 按下列步骤添加 **Data Model**：

a. 点击 **File > New > File...** 菜单项，创建 **iOS > Core Data > Data Model**。

b. 在 Targets 中勾选 **EasyiCloud**，点击 **Create** 按钮，用 **Model** 做文件名。

5. 按下列步骤配置 **Model.xcdatamodeld**：

a. 添加名为 **Test** 的实体，并添加三个属性，它们分别叫做 **modified**、**device** 及 **someValue**。

b. 把 **modified** 属性的类型设为 **Date**。

c. 把 **device** 及 **someValue** 这两个属性的类型设为 **String**。

6. 创建与 **Test** 实体相对应的 **NSManagedObject** 子类。点击 **Create** 按钮之前请先选中 EasyiCloud 这个 target。

7. 通过 **iOS > Cocoa Touch > Objective-C class** 创建 **CoreDataTVC** 的子类，并将其命名为 **TestTVC**，然后用程序清单 15-16 中的代码把 **TestTVC.m** 文件里原有的代码替换掉。

程序清单15-16 TestTVC.m文件

```
#import "TestTVC.h"
#import "CoreDataHelper.h"
#import "AppDelegate.h"
#import "Deduplicator.h"
#import "Test.h"

@implementation TestTVC

#pragma mark - DATA
- (void)configureFetch {
    CoreDataHelper *cdh = [CoreDataHelper sharedHelper];
    NSFetchRequest *request =
        [NSFetchRequest fetchRequestWithEntityName:@"Test"];
    request.sortDescriptors = [NSArray arrayWithObjects:
        [NSSortDescriptor sortDescriptorWithKey:@"modified" ascending:NO], nil];
    [request setFetchBatchSize:15];
    self.frc =
        [[NSFetchedResultsController alloc] initWithFetchRequest:request
                                                managedObjectContext:cdh.context
                                                sectionNameKeyPath:nil
                                                cacheName:nil];
    self.frc.delegate = self;
}
```

```

#pragma mark - VIEW
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];

    CoreDataHelper *cdh = [CoreDataHelper sharedHelper];
    [Deduplicator deDuplicateEntityWithName:@"Test"
        withUniqueAttributeName:@"someValue"
        withImportContext:cdh.importContext];
}
- (void)viewDidLoad {
    [super viewDidLoad];
    [self configureFetch];
    [self performFetch];
    // Respond to changes in underlying store
    [[NSNotificationCenter defaultCenter] addObserver:self
                                                selector:@selector(performFetch)
                                                name:@"SomethingChanged"
                                                object:nil];
}
- (UITableViewCell*)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *cellIdentifier = @"Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:cellIdentifier
            forIndexPath:indexPath];
    Test *test = [self.frc objectAtIndexPath:indexPath];

    cell.textLabel.text = [NSString stringWithFormat:@"From: %@", test.device];
    cell.detailTextLabel.text = test.someValue;
    return cell;
}

- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        NSManagedObject *deleteTarget = [self.frc objectAtIndexPath:indexPath];
        [self.frc.managedObjectContext deleteObject:deleteTarget];
        [self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    }
    CoreDataHelper *cdh = [CoreDataHelper sharedHelper];
    [cdh backgroundSaveContext];
}

#pragma mark - INTERACTION
- (IBAction)add:(id)sender {

    CoreDataHelper *cdh = [CoreDataHelper sharedHelper];

```

```

Test *object =
[NSEntityDescription insertNewObjectForEntityForName:@"Test"
                        inManagedObjectContext:cdh.context];

NSError *error = nil;
if (![cdh.context obtainPermanentIDsForObjects:[NSArray arrayWithObject:object]
                                     error:&error]) {
    NSLog(@"Couldn't obtain a permanent ID for object %@", error);
}
UIDevice *thisDevice = [UIDevice new];
object.device = thisDevice.name;
object.modified = [NSDate date];
object.someValue = [NSString stringWithFormat:@"Test: %@",
                                             [[NSUUID UUID] UUIDString]];

[cdh backgroundSaveContext];
}
@end

```

8. 按下列步骤操作，把默认的视图换成表格视图：

- a. 选定 **Main.storyboard**。
- b. 删掉现有的 **View Controller**。
- c. 向故事板中拖放一个 **Table View Controller**，然后点击 **Editor > Embed In > Navigation Controller**。

d. 向 **Table View Controller** 的右上角拖放一个 **Bar Button Item**。

e. 通过 **Attributes Inspector** 界面（可以按“**Option + ⌘ + 4**”组合键调出该界面）把新 **Bar Button Item** 的 **Identifier** 设为 **Add**。

f. 选中 **Prototype Cell**，将它的 **Style** 设为 **Subtitle**，将它的 **Identifier** 设为 **Cell**。

g. 选中 **Table View Controller**，用 **Identity Inspector** 界面（可以按“**Option + ⌘ + 3**”组合键调出该界面）把它的 **Custom Class** 设为 **TestTVC**。

h. 按住 **Control** 键，从 **Add** 按钮向 **Table View Controller** 底部的黄圆圈处拖一条直线。然后选择 **Sent Actions > add**。

9. 用第 14 章讲过的办法为 EasyiCloud 项目开启 iCloud 功能。

10. 按下列步骤配置 **Settings Bundle**：

a. 点击 **File > New > File...**，创建 **Resource > Settings Bundle**，把 **Targets** 里的 EasyiCloud 选中，然后点击 **Create** 按钮。

b. 选定 **/Settings.bundle/Root.plist**，展开 **Preference Items**，把不是 **Toggle Switch** 的那三项删掉。

c. 将 **Item 0** 的 **Default Value** 设为 **NO**。

d. 把 **Item 0** 的 **Identifier** 设为 **iCloudEnabled**。

e. 把 **Item 0** 的 **Title** 设为 **Enable iCloud**。

f. 将 **#import "CoreDataHelper.h"** 语句添加到 **AppDelegate.m** 文件顶部。

g. 将 **[[CoreDataHelper sharedHelper] ensureAppropriateStoreIsLoaded];** 语

句添加到 AppDelegate.m 文件中的 applicationWillEnterForeground 方法里。sharedHelper 是个新的类方法，它使 CoreDataHelper 变得更为灵活。其详细用法请参见 CoreDataHelper.h。

11. 修改 AppDelegate.m 文件中的 didFinishLaunchingWithOptions 方法，把 `[[CoreDataHelper sharedHelper] iCloudAccountIsSignedIn];` 语句添加到 `return YES;` 语句上方。

12. 把 `[[CoreDataHelper sharedHelper] backgroundSaveContext];` 语句添加到 AppDelegate.m 文件中的 applicationDidEnterBackground 方法及 applicationWillTerminate 方法里。

13. 修改 CoreDataHelper.m 文件中的 selectedUniqueAttributes 方法，使其与新的数据模型相适应。用程序清单 15-17 中的方法把原来的 selectedUniqueAttributes 方法替换掉。

程序清单15-17 CoreDataHelper.m文件中的selectedUniqueAttributes方法

```
- (NSDictionary*)selectedUniqueAttributes {
    if (debug==1) {
        NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
    }

    NSMutableArray *entities = [NSMutableArray new];
    NSMutableArray *attributes = [NSMutableArray new];

    // Select an attribute in each entity for uniqueness
    [entities addObject:@"Test"];[attributes addObject:@"someValue"];
    //[entities addObject:@"Item"];[attributes addObject:@"name"];
    //[entities addObject:@"Unit"];[attributes addObject:@"name"];
    //[entities addObject:@"LocationAtHome"];[attributes addObject:@"storedIn"];
    //[entities addObject:@"LocationAtShop"];[attributes addObject:@"aisle"];
    //[entities addObject:@"Item_Photo"];[attributes addObject:@"data"];

    NSDictionary *dictionary = [NSDictionary dictionaryWithObjects:attributes
                                                                    forKeys:entities];

    return dictionary;
}
```

14. 修改 CoreDataHelper.m 文件的 mergeNoniCloudDataWithiCloud 方法里的 entitiesToCopy 变量，使其与新的数据模型相适应。用下面这行代码来替换原有的那行代码：

```
NSArray *entitiesToCopy = [NSArray arrayWithObjects:@"Test", nil];
```

15. 在两台或更多台设备上运行程序，然后通过 **Settings > EasyiCloud** 启用 iCloud 功能。如果把 CoreDataHelper.m 文件的 loadiCloudStore 方法里的 confirmMergeWithiCloud 注释掉，那么程序就不会把数据合并到 iCloud 了。

16. 点击“+”按钮，在每台设备上上面创建一些测试用的对象。等到控制台日志里出现“Using local storage:0”消息之后，就应该能在其他设备上看到这些数据了。正常的运行效果如图 15-6 所示。如果模拟器的同步速度看上去比较慢，那么可以点击它的 Debug > Trigger iCloud Sync 菜单项。

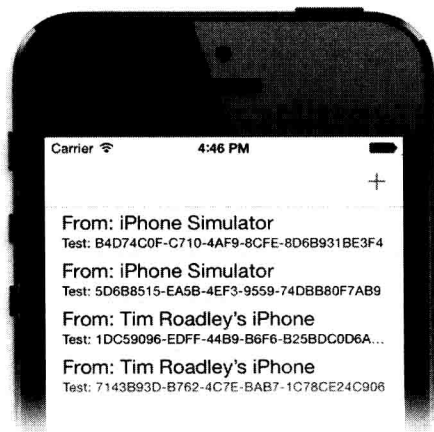


图 15-6 EasyiCloud 程序的运行截图

为了使大家用起来方便一些，笔者已经把 EasyiCloud 项目放在网上了，读者可以从 <http://timroadley.com/LearningCoreData/EasyiCloud.zip> 下载它。



与 Web 服务相集成

每个人都是天才。但假如要用爬树能力来判定一条鱼的才智，那它一辈子都会以为自己很傻。

——阿尔伯特·爱因斯坦

通过第 14 章和第 15 章，我们已经把 Grocery Dude 程序的 Core Data 栈同 iCloud 集成起来了。但是 iCloud 有个很大的局限，就是其数据只能针对一个 iCloud 账号。由于 iCloud 账号与用户设备中的许多东西密切相关，所以共用 iCloud 账号的想法是不现实的，笔者也不推荐那样做。这一局限意味着我们不能以 iCloud 账号来实现多人协作功能。比方说，夫妻两人要共同准备一份购物清单。目前来说，这个功能还不能用 iCloud 完成。要想解决这个问题，关键是得考虑换一套后端系统才行。具体办法有很多种，开发者应该衡量每种办法的优缺点。要管理后端系统，就要涉及服务器、数据库、安全等事项，而笔者觉得我们不应该自己从头处理这些事情，以免“重新发明轮子”。笔者推荐的办法是采用一些行之有效的框架来做，比方说，StackMob 这样的服务提供者就研发了这种框架。本章要按步骤解释并示范如何把 Core Data 程序同 StackMob 集成起来。之所以采用 StackMob，是因为它和 Core Data 配合得比较顺畅。

16.1 StackMob 简介

StackMob 是一家提供“企业级 BaaS”(enterprise-class Backend-as-a-Service[⊖])的公司，

[⊖] BaaS 中文叫做“后端即服务”、“后端作为服务”或“作为服务的后端”。——译者注

只要相关设备能够连接到 REST API，并且能够以 OAuth2 方式完成验证，那么这些设备之间就可以分享数据。由于 StackMob iOS SDK 与 Core Data 集成得相当紧密，所以开发者只需把持久化存储区替换成 StackMob 存储区，差不多就可以开始使用了。

提示 REST 是 Representational State Transfer^①的缩写，它是一种为系统间交流（intersystem communication）而设的软件架构方式，StackMob 就采用这种架构方式。对于应用程序组件来说，REST 就是它们的“互联网”。假如系统某个区域想要访问另外一个区域的资源，那么可以像用户浏览网站那样，使用 URL 来完成。这种客户端/服务器模型效果很好，它可以令分布式系统保持模块化。也就是说，每个组件都自成一体，开发者可以单独升级某组件，而无须担心它会破坏上游（upstream）或下游（downstream）软件的稳定性。

StackMob 将会使用现有的 Core Data 托管对象模型。不过开发者需要配置该模型，使其可以同 StackMob 一并运作。更新完本地模型之后，需要配置等效的 StackMob schema。后端系统需要依靠这种 schema 来获知某模型能够产生何种对象。StackMob schema 就相当于 Core Data 实体。要想创建 schema，只需为每个实体都创建一个托管对象就行了，系统会据此自动生成等效的 schema。

有了 StackMob 客户端之后，就无需再使用持久化存储协调器了，这个客户端自己会提供托管对象上下文以及缓存选项。StackMob 在后端会把除了二进制数据之外的东西都保存起来，而二进制数据则要存放到 Amazon 的 Simple Storage Service^②（简称 S3）里面。使用 S3 服务是需要付费的，为了使大家能够免费使用我们的范例项目，笔者决定不在本章实现照片及缩略图功能。图 16-1 概述了 Core Data 同 StackMob 的集成。

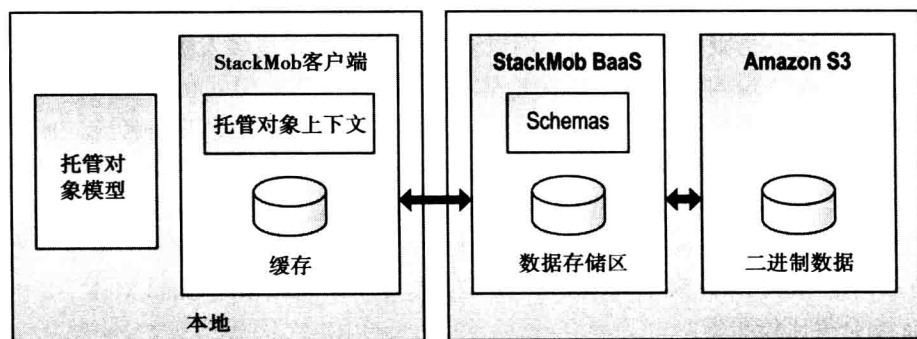


图 16-1 StackMob 概览

开发者只需把 StackMob iOS SDK 及其支持框架添加到 Xcode 项目里，即可把 Core

① 中文叫做“含状态传输”或“表述性状态传输”。——译者注

② 中文叫做“亚马逊简易存储服务”。——译者注

Data 程序同 StackMob 集成起来。所有东西都集成好之后，就可以创建 `SMClient` 实例了，我们可以为该客户端实例提供托管对象模型，以便获取持久化存储区以及操作数据所用的上下文。同时也要设置“缓存策略”，以确保所有的获取操作都会缓存在本地存储区里面，这样的话，无论是否联网，我们都能访问到其中的数据。



提示 为了演示如何将 StackMob 同 Grocery Dude 相集成，笔者创建了名叫 Grocery Cloud 的新项目。附录 B 给出了这个新项目的创建过程。如果你要自己创建这个基础项目，那就请参照附录 B 中的步骤来做。另外，也可以从 <http://www.timroadley.com/LearningCoreData/GroceryCloud-AfterAppendixB.zip> 下载 .zip 文件，并将其中的项目解压缩，然后用 Xcode 打开。使用来自 ZIP 文件的 Xcode 项目时，最好先点击 **Product > Clean** 菜单项。这样可以清除掉同名项目所残留的缓存。

16.2 StackMob SDK

为了给项目添加 StackMob 支持，首先需要下载 StackMob iOS SDK。笔者编写本书时，其最新版是 2.1.1。此外。我们还需要把 Grocery Cloud 项目与一些支持框架链接起来，这样才能使 StackMob 正常运作。

请按下列步骤修改 Grocery Cloud 项目，以添加必要的 SDK 及框架：

1. 从下面的 URL 中下载最新的 iOS SDK ^①，并将其解压缩：<https://developer.stackmob.com/sdks>。然后参照其中的安装步骤来安装它。这些步骤与下面的第 2 步至第 5 步相似。

2. 把下载好的 StackMob 目录拖放到 Grocery Cloud 这个 Xcode 项目的根目录里，然后记得把 **Copy items into destination group's folder** 打上勾。此外也要选中 **Create groups for any added folders** 以及 **Targets** 里面的 **Grocery Cloud**，然后点击 **Finish** 按钮。

3. 选中 Grocery Cloud target，切换到 **General** 分页，向下滚动到 **Linked Frameworks and Libraries** 区域。参照图 16-2 添加下列框架：

- ☐ MobileCoreServices
- ☐ SystemConfiguration
- ☐ Security
- ☐ CoreLocation

4. 切换到 Grocery Cloud target 的 **Build Settings** 分页，在 **Other Linker Flags** 区域里添加 `-ObjC`，如图 16-3 所示。（小技巧：可以点击 **All** 按钮，把所有设置都显示出来，然后搜索 **Other Linker Flags**。）

^① StackMob 项目已经停止运作。读者可从 <http://timroadley.com/LearningCoreData/GroceryCloudAfterChapter16.zip> 下载 zip 文件，该文件里含有与 StackMob iOS SDK 相关的内容。——译者注

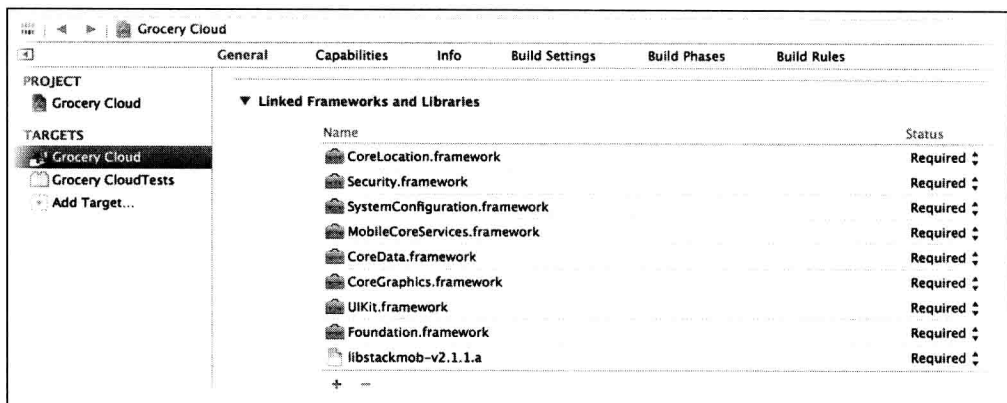


图 16-2 添加 StackMob 所需的其他框架

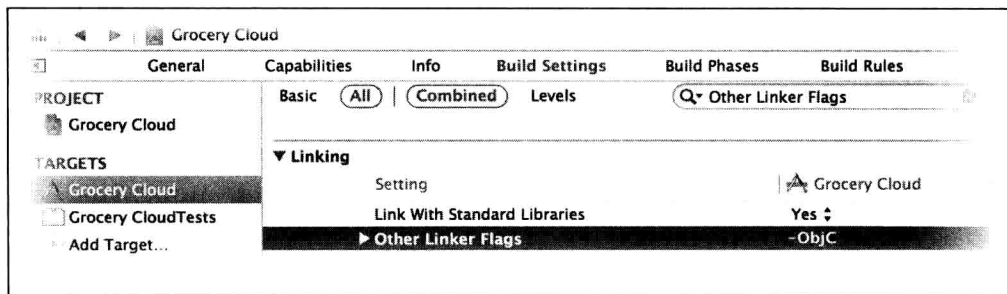


图 16-3 为使用 StackMob iOS SDK 而开启 ObjC “链接标志” (linker flag)

5. 把下列代码添加到 `/Supporting Files/Grocery Cloud-Prefix.pch` 文件底部的 `#endif` 语句上方：

```
#import <SystemConfiguration/SystemConfiguration.h>
#import <MobileCoreServices/MobileCoreServices.h>
```

前面说过，与 StackMob Web 服务的通信需要经由 StackMob 客户端来完成。在使用 StackMob 客户端之前，我们得先在 StackMob 服务器里定义一个应用程序才行。

16.3 创建 StackMob 应用程序

要想在 StackMob 上面创建应用程序，必须得有 StackMob 账号或 GitHub 账号。请登录 www.stackmob.com 网站创建账号。创建好账号之后，网站会转向一个页面，在那里可以设定新的应用程序。

请创建名为 `grocery_cloud` 的应用程序。此时网站会显示一份教程，你可以略过它。请切换到 **StackMob Dashboard**，然后点击 **Schema Configuration**。开发者可以在这里查

看 Grocery Cloud 项目现有的后台 schema。目前只有一个名叫 `user` 的 schema，如图 16-4 所示。

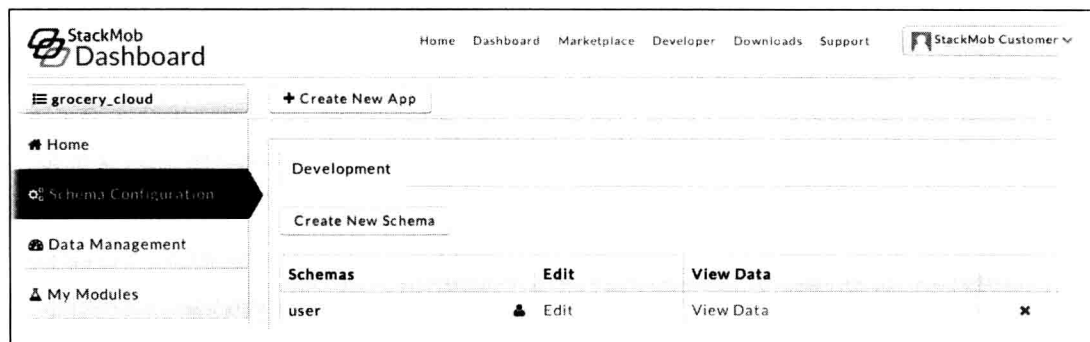


图 16-4 StackMob 应用程序的 schema

16.4 准备托管对象模型

从图 16-1 中可以看出，使用 StackMob 的时候，我们依然采用托管对象模型来定义应用程序的数据结构。但是要想和 StackMob 相集成，就必须先修改现有的模型。

从第 2 版开始，StackMob iOS SDK 就支持 **Offline Sync**（离线同步）功能了。该功能使得应用程序在无法联网时依然可以运作。离线状态下的“写入”操作会缓存起来，待网络连接恢复之后，再同步到服务器。

为了支持 Offline Sync 功能，我们需要给每个实体添加两个日期类型的属性，它们是 `createddate` 和 `lastmoddate`。如果设备在离线时修改了某份数据，而有人又通过其他方式修改了同一份数据，那么就会产生冲突，我们将用这两个属性来解决冲突。为了简单起见，我们把这两个属性直接添加到每个实体之中，这样就无须使用映射模型了。对于尚未公开发行的应用程序来说，最好能够新建名为 StackMob 的实体，并把它用作现存所有实体的父实体。这样做可以简化模型的设计，StackMob 自编的文档详细解释了其中的原因。

除了这两个新的日期类型的属性之外，还要给每个实体添加字符串类型的属性，用来表示对象的“StackMob primary key ID”（StackMob 主键 ID）。该属性的名称是包含它的那个实体名称（全小写）加上 `_id` 后缀。比方说，Item 实体里的这个新属性就叫做 `item_id`。

请按下列步骤修改 Grocery Cloud 项目，以配置与 StackMob 相集成时所需的模型：

1. 选定 **Model.xcdatamodeld**。
2. 点击 **Editor > Add Model Version...** 菜单项。
3. 点击 **Finish** 按钮，把 **Model 10** 用作版本名称。
4. 确保 **Model 10.xcdatamodel** 处于受选状态。

5. 向 **Item** 实体中添加名为 `createddate` 的属性，并将其类型设为 **Date**。
6. 向 **Item** 实体中添加名为 `lastmoddate` 的属性，并将其类型设为 **Date**。
7. 把 **Item** 实体的 `createddate` 与 `lastmoddate` 这两个属性拷贝到 **Item_Photo**、**Location** 及 **Unit** 实体中。在某些版本的 Xcode 中，可能需要把 Model Editor 切换到 **Table Editor Style**，才能拷贝及粘贴属性。
8. 给每个实体添加全小写的 **String** 型属性，并按照 `entityname_id` 方式为其取名。比方说，在 **Item** 实体中，这个 **String** 型的属性就叫做 `item_id`。请参照图 16-5 来确定每个实体里面该属性的准确名称。

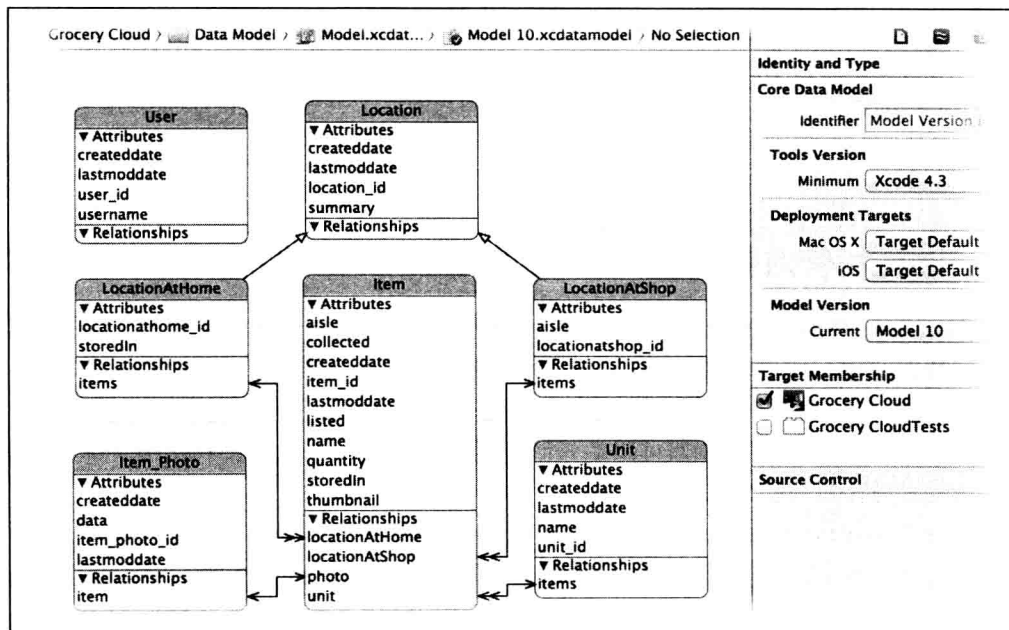


图 16-5 与 StackMob 相兼容的模型

为了完成认证，还需添加名为 **User** 的实体。

请按下列步骤修改 Grocery Cloud 项目，添加 User 实体，并开始使用 Model 10:

1. 添加名为 **User** 的实体。
2. 向 **User** 实体中添加名叫 `user_id` 的属性，并将其类型设为 **String**。
3. 向 **User** 实体中添加名叫 `username` 的属性，并将其类型设为 **String**。
4. 向 **User** 实体中添加名叫 `createddate` 的属性，并将其类型设为 **Date**。
5. 向 **User** 实体中添加名叫 `lastmoddate` 的属性，并将其类型设为 **Date**。
6. 选中 **Model 10** 里的所有实体，为其生成 `NSManagedObject` 子类，并把原有的文件替换掉（可以通过 **Editor > Create NSManagedObject Subclass...** 执行该操作）。记得勾选 **Targets** 里面的“**Grocery Cloud**”，然后点击 **Create** 按钮。

7. 再把第 6 步执行一遍，确保 Xcode 能够正确生成子类文件里的“关系”。

8. 选定 **Model.xcdatamodeld**，然后用 **File Inspector** 界面（可按“**Option + ⌘ + 1**”组合键调出该界面）把当前模型设为 **Model 10**。设置好的效果如图 16-5 所示。



提示 有时候，创建 **NSManagedObject** 子类文件会导致同一份文件在 Xcode 里链接了两次。如果出现这种情况，那么就会在运行期引发“**Mach-O Linker Error**”错误。这是由于项目把每个子类文件都引用了两遍，开发者需要将其中之一删掉，以免重复。

16.5 配置 StackMob 客户端

开发者需要通过 StackMob 客户端（也就是 **SMClient**）来和 StackMob Web 服务相交流。在创建 **SMClient** 实例时，要提供 StackMob 客户端所需的公钥。在图 16-4 所示的 Dashboard（管理面板）里面，可以找到与 StackMob 应用程序相对应的 key，这些 key 可供开发及发行之用。请在 StackMob 的 Dashboard 中点击 **Home**，以查看与你自己的 **grocery_cloud** 程序相对应的公钥。

为了给项目添加 StackMob 支持，需要在 **CoreDataHelper.h** 文件里新增一项特性，用来表示客户端实例，另外还要新增一项特性，用来表示 StackMob 特有的持久化存储区。这种存储区和其他持久化存储区相似。要想对其执行保存操作，开发者只需通过 StackMob 客户端获取当前线程的上下文即可。程序清单 16-1 列出了这两项新特性，它们用于实现与 StackMob 的集成。

程序清单 16-1 CoreDataHelper.h 文件

```
@property (retain, nonatomic) SMClient *stackMobClient;
@property (retain, nonatomic) SMCoreDataStore *stackMobStore;
```

请按下列步骤修改 Grocery Cloud 项目，以便与 StackMob 相集成：

1. 把 `#import "StackMob.h"` 语句添加到 **CoreDataHelper.h** 文件顶部。
2. 把程序清单 16-1 中的特性添加到 **CoreDataHelper.h** 文件现有的各项特性下方。

接下来需要修改 **CoreDataHelper.m** 文件，令其使用 StackMob 客户端。该客户端将会包含 StackMob 的托管对象上下文以及持久化存储协调器。程序清单 16-2 列出了与配置 StackMob 客户端有关的代码。

程序清单 16-2 CoreDataHelper.m 文件中的 init 方法

```
- (id)init {
if (debug==1) {
    NSLog(@"Running %@", @"", self.class, NSStringFromSelector(_cmd));
}
self = [super init];
if (!self) {return nil;}
}
```

```

_model = [NSManagedObjectModel mergedModelFromBundles:nil];
// Use StackMob Cache Store in case network is unavailable
SM_CACHE_ENABLED = YES;

// Verbose logging
SM_CORE_DATA_DEBUG = NO;

_stackMobClient = // APIVersion 0 = Dev, 1 = Prod
[[SMClient alloc] initWithAPIVersion:@"0" publicKey:@"YOUR_APP_KEY"];
_stackMobStore =
[_stackMobClient coreDataStoreWithManagedObjectModel:_model];

_weak SMCoreDataStore *cds = _stackMobStore;
[_stackMobClient.session.networkMonitor
    setNetworkStatusChangeBlockWithCachePolicyReturn:^(SMCachePolicy(
        SMNetworkStatus status
    ){
        if (status == SMNetworkStatusReachable) {
            [cds syncWithServer];
            return SMCachePolicyTryNetworkElseCache;
        } else {
            return SMCachePolicyTryCacheOnly;
        }
    }
)];

_context = [_stackMobStore contextForCurrentThread];

_importContext =
[[NSManagedObjectContext alloc]
    initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_importContext performBlockAndWait:^(
    [_importContext setParentContext:_context];
    [_importContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_importContext setContextShouldObtainPermanentIDsBeforeSaving:YES];
    [_importContext setUndoManager:nil]; // the default on iOS
)];

_sourceContext =
[[NSManagedObjectContext alloc]
    initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[_sourceContext performBlockAndWait:^(
    [_sourceContext setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
    [_sourceContext setParentContext:_context];
    [_sourceContext setContextShouldObtainPermanentIDsBeforeSaving:YES];
    [_sourceContext setUndoManager:nil]; // the default on iOS
)];
return self;
}

```

CoreDataHelper.m 文件的 init 方法的前一半代码几乎都改了，因为这次要由

StackMob 客户端来处理持久化存储区的协调事宜。配置客户端时采用的 API 版本是 0，而不是 1，0 表示 Development（开发），1 表示 Production（生产、发行）。程序清单 16-2 里并没有包含公钥，读者需要用自己在 StackMob Dashboard 中查到的应用程序的公钥来替换其中的 YOUR_APP_KEY。

init 方法会根据是否能够联网来设置缓存策略，设置好之后，剩下的代码就和原来差不多了。为了遵从 StackMob 的开发建议，我们这次要在 _importContext 和 _sourceContext 上面调用新的设置方法，即 setContextShouldObtainPermanentIDsBeforeSaving，此方法可以确保与新插入的对象相对应的“permanent ID”（永久 ID）会创建在相关的上下文里面。开发者如果想使用 StackMob 的上下文，那么应该在 StackMob 存储区上面调用特制的 contextForCurrentThread 方法来获取。

请按下列步骤修改 Grocery Cloud 项目，以配置 StackMob 客户端：

1. 用程序清单 16-2 中的代码把 CoreDataHelper.m 文件里原有的 init 方法替换掉。
2. 从 StackMob Dashboard 查到自己应用程序的公钥，并用它来替换代码中的 YOUR_APP_KEY。
3. 把 CoreDataHelper.m 文件的 setupCoreData 方法里的所有代码全都注释掉。虽说这段代码已经用不到了，但为了便于稍后试验，我们还是把它留在文件里面。

16.6 SAVING

使用 StackMob 时，有两种保存方式，一种是同步保存，另一种是异步保存。为了使应用程序能够及时响应用户的操作，笔者推荐尽量采用异步方式来保存。我们要修改现有的 saveContext 及 backgroundSaveContext 方法，以便对 StackMob 上下文执行保存操作。程序清单 16-3 列出了更新之后的方法代码。

程序清单16-3 CoreDataHelper.m文件中的SAVING部分

```
- (void)saveContext {
if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}

NSManagedObjectContext *stackMobContext =
    [_stackMobStore contextForCurrentThread];
if (!stackMobContext) {
    NSLog(@"StackMob context is nil, so FAILED to save");
    return;
}

NSError *error;
[stackMobContext saveAndWait:&error];
if (!error) {
    NSLog(@"SAVED changes to StackMob store (in the foreground)");
}
```

```

    } else {
        NSLog(@"FAILED to save changes to StackMob store (in the foreground): %@",
              , error);
    }
}
- (void)backgroundSaveContext {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

    NSManagedObjectContext *stackMobContext =
    [_stackMobStore contextForCurrentThread];
    if (!stackMobContext) {
        NSLog(@"StackMob context is nil, so FAILED to save");
        return;
    }

    [stackMobContext saveOnSuccess:^(
        NSLog(@"SAVED changes to StackMob store (in the background)");
    ) onFailure:^(NSError *error) {
        NSLog(@"FAILED to save changes to StackMob store (in the background): %@",
              , error);
    }];
}
}

```

请按下列步骤修改 Grocery Cloud 项目，以实现对 StackMob 上下文的保存：

1. 用程序清单 16-3 里的对应方法把 CoreDataHelper.m 文件中的 saveContext 及 backgroundSaveContext 方法分别替换掉。
2. 修改 AppDelegate.m 文件中的 applicationDidEnterBackground 与 applicationWillTerminate 方法，把 [[self cdh] backgroundSaveContext]; 改为 [[self cdh] saveContext];。
3. 修改 ItemVC.m 文件中的 textFieldDidEndEditing 方法，把 [cdh saveContext]; 语句添加到该方法底部。

创建新对象的时候，需要传入 primary key（主键），然后，程序会把新对象立刻保存起来。在 Grocery Cloud 项目中，每个方法插入新对象时所用的代码都略有不同，程序清单 16-4 将其全部列了出来。这些代码应该分别放在相应的 insertNewObjectForEntityForName 调用语句之后。

程序清单16-4 Grocery Cloud项目中与插入对象有关的代码

```

// In PrepareTVC.m prepareForSegue
[newItem setValue:[newItem assignObjectId]
              forKey:[newItem primaryKeyField]];
[cdh saveContext];
// In LocationsAtHomeTVC.m prepareForSegue
[newLocationAtHome setValue:[newLocationAtHome assignObjectId]

```

```

        forKey:[newLocationAtHome primaryKeyField]];
[cdh saveContext];

// In LocationsAtShopTVC.m prepareForSegue
[newLocationAtShop setValue:[newLocationAtShop assignObjectId]
    forKey:[ newLocationAtShop primaryKeyField]];
[cdh saveContext];

// In UnitsTVC.m prepareForSegue
[newUnit setValue:[newUnit assignObjectId]
    forKey:[newUnit primaryKeyField]];
[cdh saveContext];

// In ItemVC.m ensureItemHomeLocationIsNotNull
[locationAtHome setValue:[locationAtHome assignObjectId]
    forKey:[ locationAtHome primaryKeyField]];
[cdh saveContext];

// In ItemVC.m ensureItemShopLocationIsNotNull
[locationAtShop setValue:[locationAtShop assignObjectId]
    forKey:[locationAtShop primaryKeyField]];
[cdh saveContext];

// In ItemVC.m UIImagePickerController:didFinishPickingMediaWithInfo
[newPhoto setValue:[newPhoto assignObjectId]
    forKey:[newPhoto primaryKeyField]];
[cdh saveContext];

```

请按下列步骤修改 Grocery Cloud 项目，以确保每个新对象都能有 primary key:

1. 修改程序清单 16-4 中所提到的那些方法，在每个方法中找到调用 insertNewObjectForEntityForName 的语句，然后把相应代码添加到该语句下方。

请注意，某些与本章内容无关的类并未改动（例如，CoreDataImporter 类就没有修改）。

16.7 响应底层数据的变更

笔者编写本书时，StackMob 里面还没有可供监测的相关通知（notification），假如有这种通知的话，那我们就可以在底层数据发生改变时触发表格视图的刷新操作了。比方说，在与 iCloud 相集成时，可以通过监测 NSPersistentStoreDidImportUbiquitousContentChangesNotification 来刷新表格视图。为了解决这个问题，我们需要添加 viewDidAppear 方法，以便在每个表格视图上面触发刷新操作。程序清单 16-5 列出了相关代码。

程序清单16-5 PrepareTVC.m、ShopTVC.m、UnitsTVC.m、LocationsAtHomeTVC.m及

LocationsAtShopTVC.m文件中的viewWillAppear方法

```

- (void)viewDidAppear:(BOOL)animated {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    [super viewDidAppear:animated];
    [self configureFetch];
    [self performFetch];
}

```

请按下列步骤修改 Grocery Cloud 项目，确保每个表格视图都能在出现时刷新自己，以显示最新数据：

1. 修改每个表格视图控制器类文件（也就是 PrepareTVC.m、ShopTVC.m、UnitsTVC.m、LocationsAtHomeTVC.m 及 LocationsAtShopTVC.m 文件）里的 viewDidLoad 方法，把 [self performFetch]; 这一行代码注释掉。

2. 修改每个表格视图控制器类文件（也就是 PrepareTVC.m、ShopTVC.m、UnitsTVC.m、LocationsAtHomeTVC.m 及 LocationsAtShopTVC.m 文件）里的 VIEW 部分，把程序清单 16-5 中的方法分别添加到该部分顶部。

本章稍后还要向 CoreDataTVC 里面添加 UIRefreshControl，使用户可以手动触发刷新操作。

16.8 自动生成 Schema

后端 StackMob 服务器要求每个待同步的实体都必须配置 schema。实体的命名约定与 schema 的命名约定稍有不同。StackMob 的网站上面可以查到《StackMob iOS SDK Reference》，其中的 StackMob Core Data Coding Practices 一节解释了两者的区别。但是目前来说，我们不一定非要掌握这两者的区别不可，因为开发阶段的 schema 是自动生成的。schema 的生成与创建新对象同样简单，只要创建一个新对象，相应的 schema（及关系）就能生成好。程序清单 16-6 列出了一个新的方法，这段一次性（one-off）代码用来为每个实体生成等效的 StackMob schema。

程序清单16-6 AppDelegate.m文件中的generateStackMobSchema方法

```

- (void)generateStackMobSchema {

if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

    CoreDataHelper *cdh = [self cdh];

```

```

NSManagedObjectContext *stackMobContext =
[cdh.stackMobStore contextForCurrentThread];

// Create new objects for each entity
LocationAtHome *locationAtHome =
[NSEntityDescription insertNewObjectForEntityForName:@"LocationAtHome"
inManagedObjectContext:stackMobContext];
LocationAtShop *locationAtShop =
[NSEntityDescription insertNewObjectForEntityForName:@"LocationAtShop"
inManagedObjectContext:stackMobContext];
Unit *unit =
[NSEntityDescription insertNewObjectForEntityForName:@"Unit"
inManagedObjectContext:stackMobContext];
Item *item =
[NSEntityDescription insertNewObjectForEntityForName:@"Item"
inManagedObjectContext:stackMobContext];
Item_Photo *item_photo =
[NSEntityDescription insertNewObjectForEntityForName:@"Item_Photo"
inManagedObjectContext:stackMobContext];

// Set Primary Key Fields
[locationAtHome setValue:[locationAtHome assignObjectId]
forKey:[locationAtHome primaryKeyField]];
[locationAtShop setValue:[locationAtShop assignObjectId]
forKey:[locationAtShop primaryKeyField]];
[unit setValue:[unit assignObjectId]
forKey:[unit primaryKeyField]];
[item setValue:[item assignObjectId]
forKey:[item primaryKeyField]];
[item_photo setValue:[item_photo assignObjectId]
forKey:[item_photo primaryKeyField]];

// Give each attribute a value so the schema is generated automatically
locationAtHome.storedIn = @"Fridge";
locationAtShop.aisle = @"Cold Section";
unit.name = @"L";
item.name = @"Milk";
item.collected = [NSNumber numberWithInt:NO];
item.listed = [NSNumber numberWithInt:YES];
item.quantity = [NSNumber numberWithInt:1];

// sectionNameKeyPath WORKAROUND (See Appendix B)
item.storedIn = @"Fridge";
item.aisle = @"Cold Section";

// Always save objects before relationships are created to avoid corruption
[cdh saveContext];

// Create relationships and then save again
item.unit = unit;
item.locationAtHome = locationAtHome;
item.locationAtShop = locationAtShop;

```

```

        item.photo = item_photo;
        [cdh saveContext];
    }

```

generateStackMobSchema 方法里的代码看起来并不陌生，因为这些代码都曾在本书前面出现过。我们针对每个实体创建一个对象，然后填充每个对象的属性，使得这些属性能够包含在自动生成的 StackMob schema 里面。

请按下列步骤修改 Grocery Cloud 项目，以便根据每个实体来生成 StackMob schema：

1. 把 #import "Item_Photo.h" 语句添加到 AppDelegate.m 文件顶部。
2. 把程序清单 16-6 中的代码添加到 AppDelegate.m 文件底部的 @end 语句上方。
3. 修改 AppDelegate.m 文件的 didFinishLaunchingWithOptions 方法，把 [self generateStackMobSchema]; 语句添加到方法底部的 return YES; 语句上方。
4. 在设备或 iOS 仿真器上面运行 Grocery Cloud 程序，以便自动生成 StackMob schema。首次运行时必须联网，因为只有这样，StackMob 才能自动生成 schema。无须理会“Object will be placed in unnamed section”这样的消息，附录 B 里说明了这个问题。
5. 在 StackMob Dashboard 中点击 **Schema Configuration**，查看自动生成的 schema。正确效果应该如图 16-6 所示。

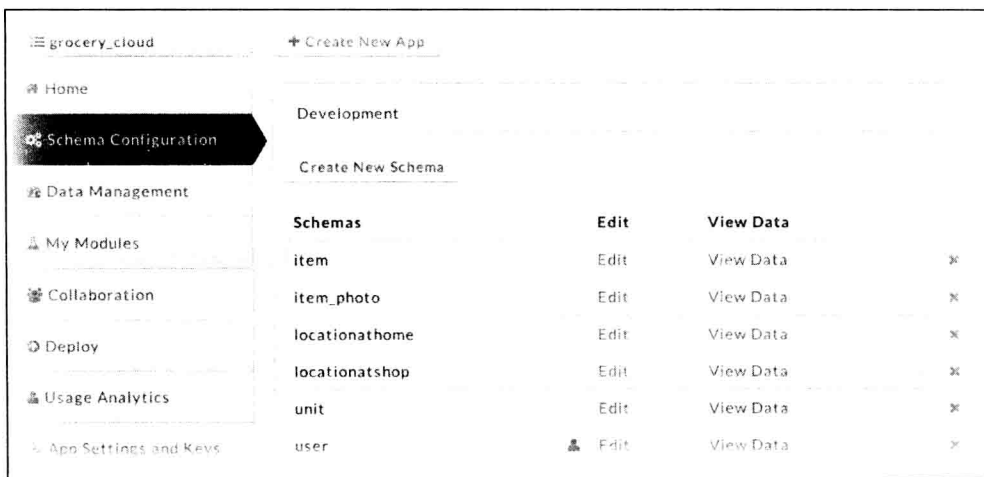


图 16-6 自动生成的 schema

6. 把 AppDelegate.m 文件的 didFinishLaunchingWithOptions 方法里的 [self generateStackMobSchema]; 语句注释掉。

16.9 Schema 的权限

默认情况下，StackMob 的 schema 权限是放开的。也就是说，任何人都能够创建

(Create)、读取 (Read)、更改 (Update) 及删除 (Delete) 对象 (这四种操作合称 CRUD^①)。对于 Grocery Cloud 项目的开发 (development) 阶段来说, 这么做是有意义的, 但到了发行 (production) 阶段, 就不合适了。之所以要创建 Grocery Cloud 项目, 就是为了使多位用户能够共同管理一份购物清单。为了实现这一目标, 我们需要以“用户账号”^②来限制每个人所能执行的操作。假如两个人需要共享一份购物清单, 那么他们就得用同一个 StackMob 用户账号来认证。对于终端用户来说, 他们实际面对的是“共享清单”(shared list), 而这种共享清单正是以 StackMob 用户账号的形式来实现的。所以说, 这种共享清单只不过是个加了密码的 StackMob 用户账号而已。

请按下列步骤修改 **grocery_cloud** 应用程序的 schema, 以限制访问权限:

1. 进入 StackMob Dashboard 中的 **Schema Configuration** 界面。
2. 编辑名为 **item** 的 schema。
3. 向下滚动到 **Schema Permissions** 区域, 然后按照图 16-7 来配置“Permission Level” (权限级别)。

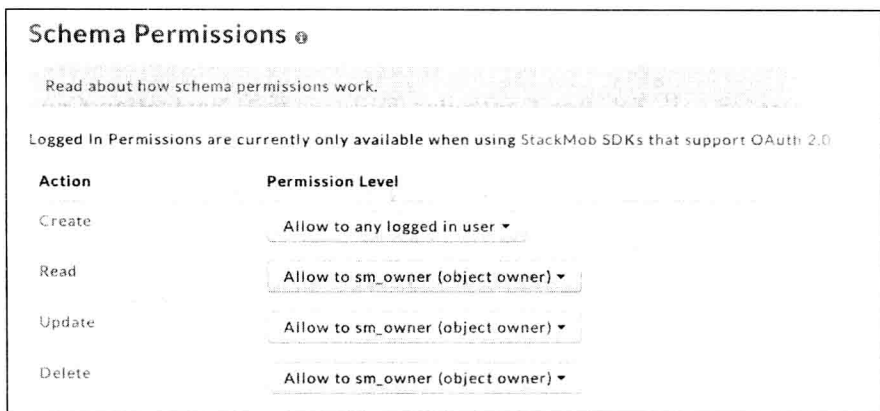


图 16-7 配置 schema 的权限级别

4. 点击 **Save Schema** 按钮。
5. 点击 **Schema Configuration**, 返回 schema 列表。
6. 针对 **item_photo**、**locationathome**、**locationatshop** 及 **unit** 这四个 schema, 重复执行第 2 至第 5 步。

配置好权限级别之后, 再想操作数据就得使用 StackMob 的用户账号了。为此, 我们添加一个新的界面, 使终端用户可以在该界面中创建共享清单 (它实际上是以 StackMob 用户账号来实现的)。在此之前所创建的对象仍然具备开放权限。但是在此之后执行的创建对象

① 中文可以叫做“增、删、改、查”。——译者注

② 此处的“用户”是针对 StackMob 来说的, 与“使用 Grocery Cloud 程序的人” (也就是作者在后面提到的终端用户) 不是同一个概念。为了避免混淆, 译文用 StackMob 用户和终端用户来区分这两个概念。——译者注

或编辑对象操作则会被 StackMob 所拒绝，而对相关的上下文所执行的保存操作也将失败。

16.10 认证

我们在上一节里配置了“权限结构”（permissions structure），使得终端用户只能对其所创建的内容执行读取、修改或删除操作。也就是说，每位终端用户可以维护各自的购物清单，而其他终端用户则无法看到该清单中的内容。由于 StackMob 用户与共享清单是一回事，所以如果想和他人共享某份清单，那么只需告知与该共享清单相对应的 StackMob 用户名与密码即可。为了支持此功能，我们要实现一种新的界面，使终端用户可以创建或登录共享清单。

请按下列步骤修改 Grocery Cloud 项目，制作 **More** 选项卡，以实现清单共享：

1. 选中 **Main.storyboard**。
2. 向故事板中拖放一个 **View Controller**，把它放在 **Navigation Controller-Shop** 下方。
3. 选中这个新的 **View Controller**，然后点击 **Editor > Embed In > Navigation Controller** 菜单项。
4. 按住 **Control** 键，从 **Tab Bar Controller** 的中心向新的 **Navigation Controller** 拖一条直线，然后选择 **Relationship Segue > view controllers**。
5. 用 **Attributes Inspector** 界面（可以按“**Option + ⌘ + 4**”组合键调出该界面）把新 **Tab Bar Item** 的 **Identifier** 设为 **More**。
6. 把新 **View Controller** 的 **Navigation Item Title** 设为 **Shared Lists**，配置好的效果应该如图 16-8 所示。

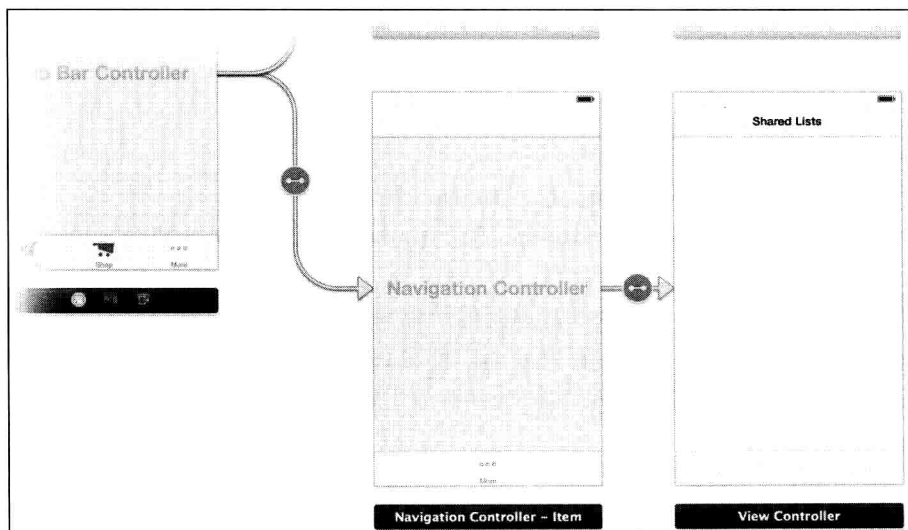



图 16-8 用于共享清单的 LoginVC 界面

从现在开始，我们就把这个新的视图叫做 LoginVC 视图，稍后将要专门为这个视图新建一个自定义的类，而这个类的名字就是 LoginVC。首先要给视图中添加一些新的界面元件，使终端用户可以认证并创建与共享清单有关的 StackMob 账号。

请按下列步骤修改 Grocery Cloud 项目，以修改 LoginVC 视图：

1. 向 LoginVC 视图中拖放两个 **Text Field**、两个 **Button** 和一个 **Label**，位置可随意摆放。
2. 用 **Attributes Inspector** 界面（可以按“**Option** + ⌘ + 4”组合键调出该界面）来配置这两个 Text Field：

- ❑ 把 **Alignment** 设为 **Center**。
- ❑ 把 **Border Style** 设为 **Line**（Attributes Inspector 界面以矩形表示该选项）。
- 3. 把其中一个文本框里的文本改为 **Shared List Name**，另一个改为 **Password**。
- 4. 勾选 **Password** 文本框的 **Secure** 选项，把它设为 **Secure** .



技巧 可以在 keyboard 选项下方找到 **Secure** 选项。

5. 用 **Size Inspector** 界面（可以按“**Option** + ⌘ + 5”组合键调出该界面）把两个文本框的 **Height** 都设为 **44**。
6. 把两个按钮的 **Width** 都设为 **120**。
7. 用 **Attributes Inspector** 界面（可以按“**Option** + ⌘ + 4”组合键调出该界面）把其中一个按钮的文本设为 **Create**，另一个设为 **Enter**。
8. 把标签的 **Alignment** 设为 **Center**。
9. 把标签的文本设为 **Create or Enter a Shared List**。
10. 按照图 16-9 来摆放各控件的位置。在调整控件的过程中，请把文本框与标签控件拓宽到“边线”（edge guide）。

11. 选中 Shared Lists 视图，然后点击 **Editor > Resolve Auto Layout Issues > Reset to Suggested Constraints in View Controller** 菜单项。如果这样做无法解决“Auto Layout Issue”（自动布局问题），那么你可能要参照本章末尾的成品项目来手动配置 constraint。

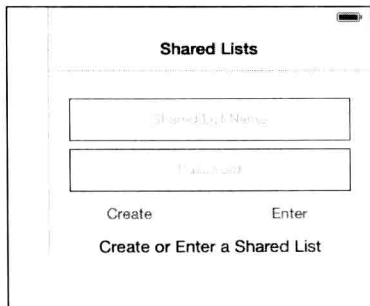


图 16-9 修改 LoginVC 界面，以实现共享清单功能

16.10.1 给 User 类增加安全措施

终端用户验证其账号的时候，其密码通常会以明文传输。为了防止明文传输，我们要对 NSManagedObject 的子类 User 做一些定制。定制之后，只要开发者采用程序清单 16-7 里的 initWithNewUserInContext 方法来创建 User 对象，那么敏感数据就会以加密的形式传送。

② 意思是说，其中的密码不会显示为普通的字符，而是会以圆点等特殊符号来表示。——译者注

程序清单16-7 User.h文件

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
#import "StackMob.h"
@interface User : SMUserManagedObject
@property (nonatomic, retain) NSString * username;
@property (nonatomic, retain) NSDate * createddate;
@property (nonatomic, retain) NSDate * lastmoddate;
- (id)initNewUserInContext:(NSManagedObjectContext *)context;
@end
```

程序清单 16-7 中的代码来自 StackMob 的《Creating a User Object》教程，你可以从该网站中搜到这篇文章。User 类的实现代码如程序清单 16-8 所示。

程序清单16-8 User.m文件

```
#import "User.h"
@implementation User
@dynamic createddate;
@dynamic lastmoddate;
@dynamic username;

- (id)initNewUserInContext:(NSManagedObjectContext *)context {
    self = [super initWithEntityName:@"User"
        insertIntoManagedObjectContext:context];
    return self;
}
@end
```

请按下列步骤修改 Grocery Cloud 项目，以确保能够安全地创建 User 对象：

1. 用程序清单 16-7 把 User.h 文件里原有的代码替换掉。
2. 用程序清单 16-8 把 User.m 文件里原有的代码替换掉。



提示 如果重新生成 NSManagedObject 的子类，那么我们为 User 类编写的那些自定义代码就会丢失。所以，重新生成子类文件之后，请记得按照程序清单 16-7 和程序清单 16-8 来修改这个类。

16.10.2 创建 LoginVC 类

接下来要添加驱动 Shared List 视图界面所需的代码。前面说过，驱动 Shared List 视图界面的那个自定义类的名字叫做 LoginVC，该类负责处理共享清单（也就是 StackMob 用户账号）的创建及验证。程序清单 16-9 列出了头文件的代码。

程序清单16-9 LoginVC.h文件

```
#import <UIKit/UIKit.h>
@interface LoginVC : UIViewController <UITextFieldDelegate>
```

```

@property (strong, nonatomic) IBOutlet UITextField *usernameTextField;
@property (strong, nonatomic) IBOutlet UITextField *passwordTextField;
@property (strong, nonatomic) IBOutlet UILabel *statusLabel;
@property (strong, nonatomic) UIActivityIndicatorView *activityIndicatorView;
@property (strong, nonatomic) UIView *activityIndicatorBackground;

- (IBAction)create:(id)sender;
- (IBAction)authenticate:(id)sender;
@end

```

LoginVC 视图里面有两个文本框，一个用于输入用户名（也就是共享清单的名称），另一个用于输入密码。LoginVC 类也遵从了 UITextFieldDelegate 协议，这样一来，当用户输入完某个文本框的内容之后，程序就可以把键盘隐藏起来了。此外，还有用于显示状态的标签以及用于显示程序状况的 Activity 视图。create 方法及 authenticate 方法分别用来创建新账号及登录现有账号。

请按下列步骤修改 Grocery Cloud 项目，以便创建 LoginVC 类：

1. 选中名为 **Grocery Cloud View Controllers** 的组。
2. 点击 **File > New > File...** 菜单项。
3. 点击 **iOS > Cocoa Touch > Objective-C class**，然后点击 **Next** 按钮。
4. 把 **Subclass of** 设为 UIViewController。
5. 把类名设为 LoginVC。
6. 点击 **Next** 按钮，勾选 Targets 里的“Grocery Cloud”，然后点击 **Create** 按钮。
7. 用程序清单 16-9 中的代码把 LoginVC.h 文件里原有的内容替换掉。如果点击 LoginVC.m 文件，那么 Xcode 可能会警告说 create 及 authenticate 方法还没有实现。
8. 选中 **Main.storyboard**。
9. 选定 **Shared Lists View Controller**，用 **Identity Inspector** 界面（可以按“**Option + ⌘ + 3**”组合键调出该界面）把它的 Custom Class 设为 LoginVC，如图 16-10 所示。

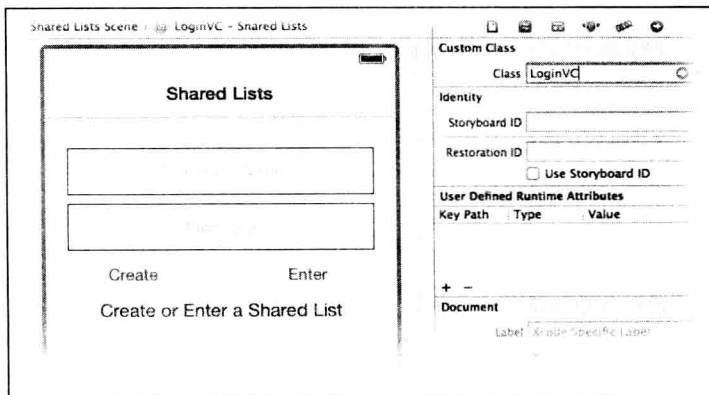


图 16-10 把 Custom Class 设为 LoginVC

接下来，我们要把文本框、按钮以及标签分别链接到 LoginVC 头文件中的对应特性上面，这样可以使 LoginVC 的实现代码能够引用它们。

请按下列步骤修改 Grocery Cloud 项目，把用户界面里的新元件与对应的代码相链接：

1. 双击 LoginVC.h，在新窗口里打开该文件。
2. 选定 **Main.storyboard**，然后把 LoginVC.h 放在靠近 **Shared Lists View Controller** 的地方。
3. 按住 **Control** 键，从 Shared List Name 文本框向 LoginVC.h 文件里的 username-TextField 特性拖一条线。
4. 按住 **Control** 键，从 Password 文本框向 LoginVC.h 文件里的 passwordTextField 特性拖一条线。
5. 按住 **Control** 键，从 **Label** 向 LoginVC.h 文件里的 statusLabel 特性拖一条线。
6. 按住 **Control** 键，从 **Create** 按钮向 LoginVC.h 文件里的 create 方法拖一条线。
7. 按住 **Control** 键，从 **Enter** 按钮向 LoginVC.h 文件里的 authenticate 方法拖一条线。

把界面上的相关元件与对应的代码链接好之后，就可以实现 LoginVC 了。程序清单 16-10 列出了 LoginVC.m 文件开头部分的代码。

程序清单 16-10 LoginVC.m 文件

```
#import "LoginVC.h"
#import "CoreDataHelper.h"
#import "AppDelegate.h"
#import "User.h"

@implementation LoginVC
#define debug 1

#pragma mark - VIEW
- (void)updateStatus {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    CoreDataHelper *cdh =
    [(AppDelegate *)[[UIApplication sharedApplication] delegate] cdh];

    if([cdh.stackMobClient isLoggedIn]) {

        [cdh.stackMobClient getLoggedInUserOnSuccess:^(NSDictionary *result) {
            self.statusLabel.text =
            [NSString stringWithFormat:@"You're using '%@'",
                                     [result objectForKey:@"username"]];
        } onFailure:^(NSError *error) {
            self.statusLabel.text = @"Create or Enter a Shared List";
        }];
    }
}
```

```

    } else {
        self.statusLabel.text = @"Create or Enter a Shared List";
    }
}
- (void)viewDidLoad {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

    [super viewDidLoad];
    [_usernameTextField setDelegate:self];
    [_passwordTextField setDelegate:self];
    [self hideKeyboardWhenBackgroundIsTapped];
    [self updateStatus];
}

#pragma mark - WAITING
- (void)showWait:(BOOL)visible {
if (debug==1) {
    NSLog(@"Running %@", NSStringFromClass(_cmd));
}

    if (!_activityIndicatorBackground) {
        _activityIndicatorBackground =
            [[UIView alloc] initWithFrame:CGRectMake(0, 0, 100, 100)];
    }
    [_activityIndicatorBackground
        setCenter:CGPointMake(self.view.frame.size.width/2,
                               self.view.frame.size.height/2)];
    [_activityIndicatorBackground setBackgroundColor:[UIColor blackColor]];
    [_activityIndicatorBackground setAlpha:0.5];
    _activityIndicatorView = [[UIActivityIndicatorView alloc]
        initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhite];
    _activityIndicatorView.center =
        CGPointMake(_activityIndicatorBackground.frame.size.width/2,
                     _activityIndicatorBackground.frame.size.height/2);

    if (visible) {
        [self.view addSubview:_activityIndicatorBackground];
        [_activityIndicatorBackground addSubview:_activityIndicatorView];
        [_activityIndicatorView startAnimating];
    }
    else {
        [_activityIndicatorView stopAnimating];
        [_activityIndicatorView removeFromSuperview];
        [_activityIndicatorBackground removeFromSuperview];
    }
}

#pragma mark - ALERTING

```

```

- (void)showAlertWithTitle:(NSString*)title message:(NSString*)message {

if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:title
                                                         message:message
                                                         delegate:nil
                                                         cancelButtonTitle:nil
                                                         otherButtonTitles:@"Ok", nil];

    [alert show];
    [self showWait:NO];
}

#pragma mark - VALIDATION
- (BOOL)textFieldIsBlank {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    if ([_usernameTextField.text isEqualToString:@""] ||
        [_passwordTextField.text isEqualToString:@""]) {

        [self showAlertWithTitle:@"Please Enter a Shared List Name and Password"
                                message:@"If you don't have a Shared List you can create
one by filling in a Shared List Name and a Password, then clicking Create"];
        return YES;
    }
    return NO;
}

#pragma mark - DELEGATE: UITextField
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    [textField resignFirstResponder];
    return YES;
}

#pragma mark - INTERACTION
- (void)hideKeyboardWhenBackgroundIsTapped {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    UITapGestureRecognizer *tgr =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                                action:@selector(hideKeyboard)];

    [tgr setCancelsTouchesInView:NO];
    [self.view addGestureRecognizer:tgr];
}

- (void)hideKeyboard {

```

```

if (debug==1) {
    NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
}
[self.view endEditing:YES];
}
@end

```

这些新代码一开始分为六个部分：

- ❑ **VIEW** 部分里面的方法会根据共享清单的状态来更新状态标签，并配置初始的视图界面。
- ❑ **WAITING** 部分里面有个方法，当程序必须执行“网络调用”（network call）时，它会负责显示或隐藏活动指示器（Activity Indicator）。
- ❑ **ALERTING** 部分里面的方法包装了一个标准的 `UIAlertView`。由于将来需要多次创建警示视图，所以我们通过这种做法来减少开发者所要编写的代码量。
- ❑ 终端用户在创建或认证共享清单的时候，如果没有填写某个文本框，那么程序就用 **VALIDATION** 部分中的方法来提示用户。
- ❑ **DELEGATE: UITextField** 部分里的方法用于在文本框失去焦点时隐藏键盘。
- ❑ **INTERACTION** 部分中包含两个方法，当用户触摸程序背景区域时，它们负责把键盘隐藏起来。

请按下列步骤修改 Grocery Cloud 项目，开始实现 `LoginVC` 类：

1. 用程序清单 16-10 把 `LoginVC.m` 文件里原有的代码全部替换掉。

`LoginVC.m` 文件里还需要两个方法，而这两个方法曾在 `LoginVC.h` 中提到过。如果不编写这两个方法，那么 Xcode 就会警告开发者 `LoginVC` 类实现得不够完整。首先来实现 `create` 方法，程序清单 16-11 列出了它的实现代码，我们在 `LoginVC.m` 中新建了名叫 `ACCOUNT` 的部分，`create` 方法和另外一个待实现的方法都放在这个部分里。

程序清单16-11 `LoginVC.m`文件中的`create`方法

```

#pragma mark - ACCOUNT
- (IBAction)create:(id)sender {
    if (debug==1) {
        NSLog(@"Running %@", "%@", self.class, NSStringFromSelector(_cmd));
    }

    CoreDataHelper *cdh =
        [[AppDelegate *)[UIApplication sharedApplication] delegate] cdh];
    NSManagedObjectContext *stackMobContext =
        [cdh.stackMobStore contextForCurrentThread];

    [self showWait:YES];
    if ([self textFieldIsBlank]) {
        return;
    }

    // ENSURE NETWORK IS REACHABLE
    if (![cdh.stackMobClient.networkMonitor.currentNetworkStatus ==
        SMNetworkStatusReachable]) {

```

```

        [self showAlertWithTitle:@"Failed to Create Shared List"
                    message:@"The Internet connection appears to be offline."];
        [self updateStatus];
        return;
    }

    // ENSURE USER DOESN'T EXIST
    NSFetchRequest *fetchRequest =
    [[NSFetchRequest alloc] initWithEntityName:@"User"];
    [fetchRequest setPredicate:[NSPredicate predicateWithFormat:@"username==%@",
                                                                    self.usernameTextField.text]];

    [stackMobContext executeFetchRequest:fetchRequest
                    onSuccess:^(NSArray *results) {

        if ([results count] == 1) {
            // USER ALREADY EXISTS
            [self showAlertWithTitle:@"Please choose another Shared List Name"
                        message:[NSString stringWithFormat:@"Someone has
➡already created a list with the name '%@'", _usernameTextField.text]];
        } else {

            // CREATE USER
            self.statusLabel.text =
            [NSString stringWithFormat:@"Creating Shared List '%@'...",
                                        _usernameTextField.text];

            User *newUser =
            [[User alloc] initWithContext:stackMobContext];
            [newUser setUsername:_usernameTextField.text];
            [newUser setPassword:_passwordTextField.text];

            [stackMobContext saveOnSuccess:^(

                // USER CREATED SUCCESSFULLY
                [self updateStatus];
                [self showWait:NO];
                [self authenticate:self];

            ) onFailure:^(NSError *error) {

                // USER CREATION FAILED
                [stackMobContext deleteObject:newUser];
                [newUser removePassword];
                [self updateStatus];
                [self showWait:NO];
                [self showAlertWithTitle:@"Failed to Create Shared List"
                            message:[NSString stringWithFormat:@"%@", error]];
            }]];
        }
    } onFailure:^(NSError *error) {

```

```

// UNSURE IF USER EXISTS
[self showAlertWithTitle:@"Failed to Check if Shared List Exists"
                    message:[NSString stringWithFormat:@"%@",error]];
    }];
}

```

程序清单 16-11 中的注释非常有助于读者理解这段代码里每个步骤的意图。首先，我们在开始创建账号时把活动指示器显示出来。然后判断用户是否给两个文本框里都填写了内容，并判断网络是否可用，如果这两个条件未能满足，那么该方法就提前返回。接下来，检测 StackMob 应用程序里面是否有这个用户名，如果还没有，那就创建新用户。要是检测用户名或创建用户的操作失败了，那么该方法就通知程序的终端用户，然后返回。如果创建用户这一操作能够顺利执行，那么程序就通知终端用户，然后触发 `authenticate` 方法。

请按下列步骤修改 Grocery Cloud 项目，以实现 `create` 方法：

1. 把程序清单 16-11 里的代码添加到 `LoginVC.m` 文件中，并放在 VIEW 部分的上方。

`LoginVC.m` 文件所需的最后一个方法是 `authenticate`。程序清单 16-12 列出了这个新方法的代码。

程序清单16-12 `LoginVC.m`文件中的`authenticate`方法

```

- (IBAction)authenticate:(id)sender {
if (debug==1) {
    NSLog(@"Running %@", @"%@", self.class, NSStringFromSelector(_cmd));
}

    if ([self textFieldIsBlank]) {
        return;
    }

    CoreDataHelper *cdh =
    -[([AppDelegate *)[UIApplication sharedApplication] delegate] cdh];

    self.statusLabel.text =
    [NSString stringWithFormat:@"Connecting to Shared List '%@'...",
                                _usernameTextField.text];

    [self showWait:YES];

    // ensure new objects are saved prior to an account switch
    [[cdh.stackMobStore contextForCurrentThread] saveOnSuccess:^(
        [cdh.stackMobClient loginWithUsername:_usernameTextField.text
                                password:_passwordTextField.text
                                onSuccess:^(NSDictionary *results) {

                [self showAlertWithTitle:@"Success!"
                                message:[NSString stringWithFormat:
@"You're now using Shared List '%@'", [results valueForKey:@"username"]]];
                [self updateStatus];
                [self showWait:NO];
            }

```

```

[[NSNotificationCenter defaultCenter]
    postNotificationName:@"SomethingChanged"
    object:nil
    userInfo:nil];
} onFailure:^(NSError *error) {

    if (error.code == 401) {
        [self showAlertWithTitle:@"Failed to Enter Shared List"
            message:@"Access Denied"];
    } else {
        [self showAlertWithTitle:@"Failed to Enter Shared List"
            message:[NSString stringWithFormat:@"%@",
                error.localizedDescription]];
    }
    [self updateStatus];
    [self showWait:NO];
}];
} onFailure:^(NSError *error) {
    NSLog(@"Failed to save context prior to account switch");
}];
}

```

在开始执行任务之前，authenticate 方法首先把活动指示器显示出来，并更新 statusLabel 中的文本。如果两个文本框里有一个是空着的，那么该方法也会提前退出。终端用户在试图登录某份共享清单之前，程序里面可能还有一份旧的清单尚未保存到持久化存储区中，为了应对这种情况，我们在登录之前先对上下文执行保存操作。假如该操作能够顺利完成，那么就调用 StackMob 客户端的 loginWithUsername 方法。最常见的错误应答码是 401，意思是“访问遭到拒绝”（access has been denied）。我们会专门用一段代码来处理这种登录错误。要是认证操作能够顺利执行，那就更新 statusLabel 中的文本，并把活动指示器隐藏起来，然后发送 SomethingChanged 通知，令相关的表格视图刷新其内容。

请按下列步骤修改 Grocery Cloud 项目，以实现 create 方法：

1. 修改 LoginVC.m 文件中的 ACCOUNT 部分，把程序清单 16-12 中的代码添加到该部分底部，使它刚好出现在 VIEW 部分上方。

2. 从设备中删掉 Grocery Cloud 程序，点击 Xcode 的 **Product > Clean** 菜单项，然后重新运行此项目，以便在设备中重新安装这个程序。

3. 点击 **More** tab，输入清单名称和密码，然后点击 **Create** 按钮，创建一份共享清单。正常的运行效果如图 16-11 所示。

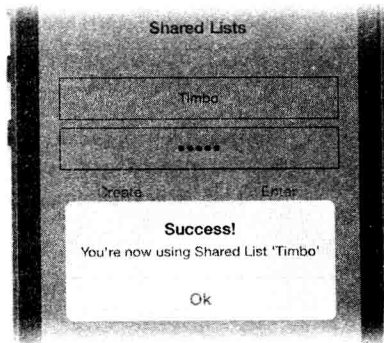


图 16-11 创建好了一份共享清单
(实际上创建的是 StackMob 账号)

4. 返回到 **Prepare** 选项卡，并创建新的货品。创建完货品之后，请再新建 Home-Location 和 ShopLocation。然后，通过选取器视图把刚才新建的 Location 设定到这件货品上面。如果新货品没有显示出来，那就试着在各选项卡之间切换，以触发刷新操作。

16.11 使程序保持响应

即便开启了缓存，程序在某些时候还是得执行网络调用。LoginVC 类在执行网络调用时会通过 showWait 方法来显示 Activity Indicator。每个表格视图在执行 fetch（获取）操作的时候，也应该采用类似的办法来提示用户。为了实现这种效果，我们要修改 CoreDataTVC.m 文件里的 performFetch 方法，使其能够复用 LoginVC 类中 WAITING 部分及 ALERTING 部分里的代码。同时，还要实现 UIRefreshControl，使用户可以在由 CoreDataTVC 所驱动的任意表格视图中，通过“向下滑动”（swipe down）操作来手动触发 performFetch 方法。

程序清单 16-13 列出了修改后的 performFetch 方法。

程序清单16-13 CoreDataTVC.m文件中的performFetch方法

```
- (void)performFetch {
if (debug==1) {
    NSLog(@"Running %@", self.class, NSStringFromSelector(_cmd));
}

if (self.refreshControl.refreshing) {
    [self.refreshControl endRefreshing];
} else {
    [self showWait:YES];
}

if (self.frc) {
    [self.frc.managedObjectContext performBlock:^(

        NSError *error = nil;
        if (![self.frc performFetch:&error]) {

            NSLog(@"%@ %@", self.class, NSStringFromSelector(_cmd),
                                                         error);
            if ([error.domain isEqualToString:@"HTTP"] && error.code == 401) {

                [self showAlertWithTitle:@"Access Denied"
                message:@"Please Create or Enter a Shared List on the More tab"];
            } else {
                if (error) {
                    [self showAlertWithTitle:@"Fetch Failed"
                    message:[NSString stringWithFormat:@"%@,
                                                         error.localizedDescription]];
                }
            }
        }
    ]
}
```



```

        [self.tableView reloadData];
        [self showWait:NO];
    }];
} else {
    NSLog(@"Failed to perform fetch: The fetched results controller is nil.");
}
}

```

请按下列步骤修改 Grocery Cloud 项目，确保相关的表格视图能够显示活动指示器：

1. 把 LoginVC.h 文件里的 activityIndicatorView 特性和 activityIndicatorBackground 特性拷贝到 CoreDataTVC.h 之中，将这两个特性放在现有的各项特性下方。
2. 把 LoginVC.m 文件里的 WAITING 部分和 ALERTING 部分拷贝到 CoreDataTVC.m 文件之中，将它们放在现有的 FETCHING 部分上方。这两个部分里面包含 showWait 及 showAlertWithTitle 方法。
3. 用程序清单 16-13 中的方法把 CoreDataTVC.m 文件里的 performFetch 方法替换掉。
4. 修改 CoreDataTVC.m 文件，将程序清单 16-14 中的代码添加到 FETCHING section 上面的 #pragma mark-FETCHING 这一行代码的上方。
5. 再次运行应用程序，看看程序在执行 fetch 操作的时候表格视图界面里会不会显示出 Activity Indicator。

程序清单16-14 CoreDataTVC.m文件中的viewDidLoad方法

```

#pragma mark - VIEW
- (void)viewDidLoad {
    [super viewDidLoad];
    UIRefreshControl *refreshControl = [UIRefreshControl new];
    [refreshControl addTarget:self
                        action:@selector(performFetch)
                        forControlEvents:UIControlEventValueChanged];
    [self setRefreshControl:refreshControl];
}

```

16.12 小结

值得高兴的是，我们已经配置好了后端服务，并把它同现有的 Core Data 应用程序集成起来了。程序里面还有一些地方也需要改进，不过笔者为了能把本章写得尽量简洁一些，就没有涉及那些内容。比方说，由于要执行网络调用，所以从 PrepareTVC 切换到 ItemVC 的时候，程序会有卡顿现象。另外，还应该重构代码，使程序在插入货品或是从网络中获取货品时，能够显示出活动指示器。虽说范例程序仍有尚待改善之处，但读者现在应该已经可以把 StackMob 框架轻松地运用到自己的应用程序上面了。

如果需要添加对照片等二进制数据的支持，那就得使用 Amazon S3 的 bucket(存储段)。若想创建 S3 bucket，首先要在 <http://aws.amazon.com/> 注册 Amazon Web Services (AWS, 亚马逊网络服务系统) 账号。创建 Amazon AWS 账号时，需提供有效的信用卡号码。<http://aws.amazon.com/pricing/s3/> 网页给出了定价，bucket 的价格会根据其所在地区而不同。欲知详情，可在 StackMob 网站搜寻《Upload to S3》教程。

16.13 习题

请根据所学内容完成下列试验：

1. 访问 StackMob Dashboard 中的 **Schema Configuration** 页面，点击 item schema 这一行里的 **View Data** 按钮，以查看其内容。
2. 创建一份新的 **Shared List**，并在其中添加一些新的货品。然后再度访问 **Schema Configuration** 页面，查看名为 item 的这个 schema 里所包含的数据。请注意，schema 的 `sm_owner` 字段用来表示 **Shared List** 的拥有者（也就是与该清单相对应的用户名），此用户可以对每一行数据执行读取、更新及删除操作。
3. 修改 `CoreDataHelper.m` 文件中的 `init` 方法，把 `SM_CORE_DATA_DEBUG` 设为 YES，以启用 StackMob 的调试功能。再次运行应用程序，并查看控制台日志。现在应该能够看到许多详尽的调试信息了，通过这些信息，开发者可以更清楚地了解到底层所发生的事情。

为了大家用起来方便，笔者把最终的 Grocery Cloud 项目放在 <http://timroadley.com/LearningCoreData/GroceryCloud-AfterChapter16.zip>，供各位读者下载。

为第 1 章的 Grocery Dude 程序所做的准备工作

本附录按步骤演示如何从头开始构建 Grocery Dude 这个范例程序。Grocery Dude 会贯穿本书学习过程，书中的每一章都要介绍与 Core Data 有关的话题，而 Grocery Dude 也会不断扩充，以供读者实践自己所学的内容。为了构建这个程序，所用到的开发环境和手机操作系统至少应该是 Xcode 5 及 iOS 7。

A.1 新建 Xcode 项目

创建基本 Xcode 项目的过程与 Core Data 无关，所以笔者把相关的操作步骤放在本附录中。你也可以不创建这个项目，而是直接从第 1 章开始阅读。笔者把本附录所创建好的这个 Xcode 项目放在了网上，并在第 1 章里给出了下载链接。

请按如下流程创建名为 Grocery Dude 的 Xcode 项目：

1. 打开 Xcode，点击 **Create a new Xcode project**。
2. 选择 **iOS > Application > Single View Application** 模板，并点击 **Next** 按钮。
3. 按图 A-1 设置项目选项，把 Tim Roadley 替换成你自己所使用的开发者名称，然后点击 **Next** 按钮。
4. 为新项目指定适当的目录，但不要勾选 **Create git repository** 选项，然后点击 **Create** 按钮。

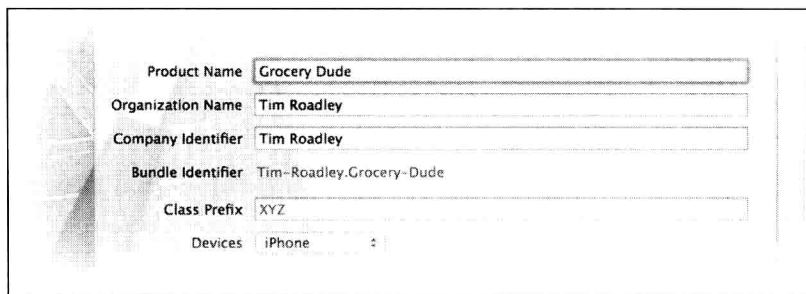


图 A-1 项目选项设置对话框

A.2 设计故事板

一开始的故事板很简单，它是由视图控制器及表格视图控制器所组成的，二者都嵌入在 Navigation 控制器里面。

请按下列步骤修改 Grocery Dude，以创建初始故事板：

1. 在名为 **Grocery Dude** 的 Xcode 组中，选择 **Main.storyboard**。
2. 从 **Utilities** 面板中把 **Table View Controller** 拖放到故事板中，将其放在现有的 **View Controller** 左侧。可以通过 “**Option + ⌘ + 4**” 组合键来显示或隐藏 **Utilities** 面板。
3. 将新放入的 **Table View Controller** 选中。
4. 请按图 A-2 右方所示，勾选 **Is Initial View Controller** 选项。如果看不到这些设置选项，那么请把 **Utilities** 面板切换至 **Attributes Inspector** 分页。可以通过 **View > Utilities > Show Attributes Inspector** 菜单项或 “**Option + ⌘ + 4**” 组合键来切换。

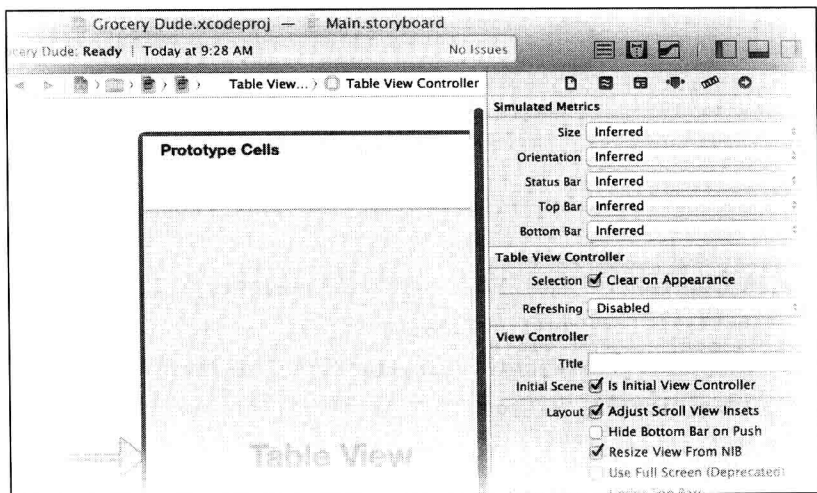


图 A-2 设置初始的视图控制器

5. 确保 **Table View Controller** 处于选中状态，然后点击 **Editor > Embed In > Navigation Controller**。

6. 把 **Bar Button Item** 拖放到 **Table View Controller** 右上角。

7. 选中这个新的 **Bar Button Item**。

8. 依照图 A-3 所示，在 **Attributes Inspector** 分页的 **Bar Button Item** section 中，把 **Identifier** 改为 **Add**。

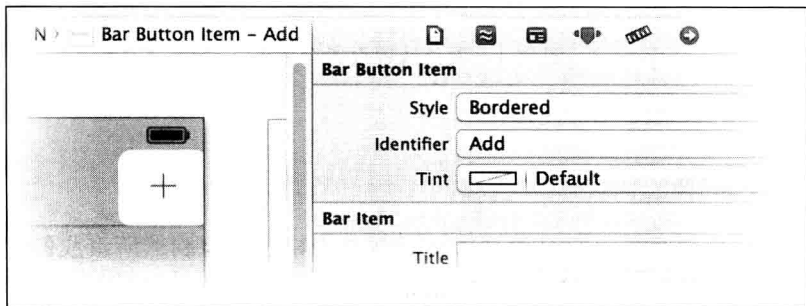


图 A-3 设置 Bar Button Item 的 Identifier

9. 按住 **Control** 键，从 **Add** 按钮向 **View Controller** 的中心拖一条线，然后选择 **Action Segue > push**。

10. 选中这个新的 action segue，然后按图 A-4 所示，在 **Attributes Inspector** 分页中把它的 **Identifier** 填写为 **Add Item Segue**。

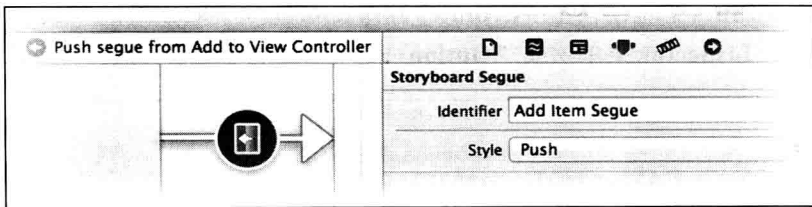


图 A-4 添加 Item Segue

11. 双击 **Table View Controller** 的标题区域，把 **Table View Controller** 的 **Title** 设为 **Items**。

12. 双击 **View Controller** 的标题区域，把 **View Controller** 的 **Title** 设为 **Item**。

13. 选中 **Table View** 中的 **Prototype Cell**。

14. 按照图 A-5 所示，在 **Attributes Inspector** 分页中把 **Table View Cell** 的 **Style** 设为 **Basic**，并将它的 **Identifier** 填写为 **Item Cell**。

15. 按图 A-6 来摆放故事板。

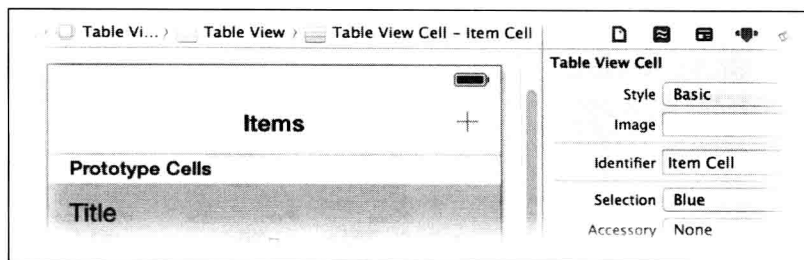


图 A-5 Item 单元格

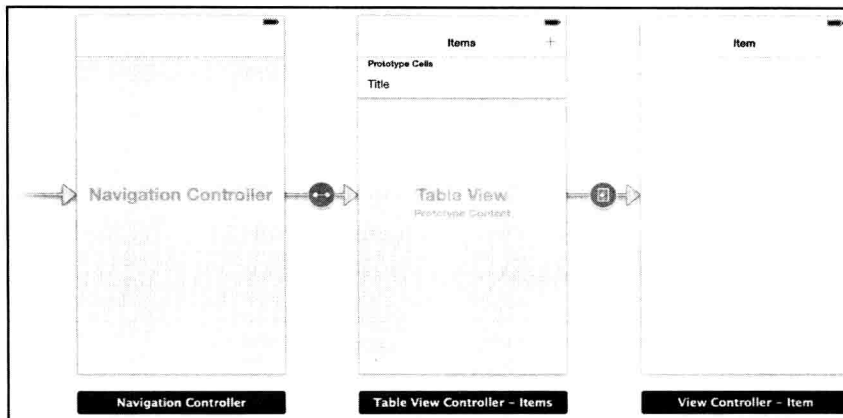


图 A-6 初始的故事板

A.3 App Icon 与 Launch Image

和其他 iOS app 一样，Grocery Dude 也要有“图标”(icon)与“图像”(image)，这样看上去会别致一些。

请按下列步骤修改 Grocery Dude，为其配置好适当的应用程序图标：

1. 从 http://www.timroadley.com/LearningCoreData/Icons_Images.zip 下载压缩包并将其解压缩。
 2. 选中 **Images.xcassets**，这是个 **Asset Catalog**，其中包含应用程序所用的图像。
 3. 选择 **AppIcon**。
 4. 将 **AppIcon_29pt.png** 拖放到写有 **iPhone 29pt** 字样的虚线框中。
 5. 将 **AppIcon_40pt.png** 拖放到写有 **iPhone Spotlight 40pt** 字样的虚线框中。
 6. 将 **AppIcon_60pt.png** 拖放到写有 **iPhone App 60pt** 字样的虚线框中。
- 设置好的 **AppIcon** 应该和图 A-7 一样。

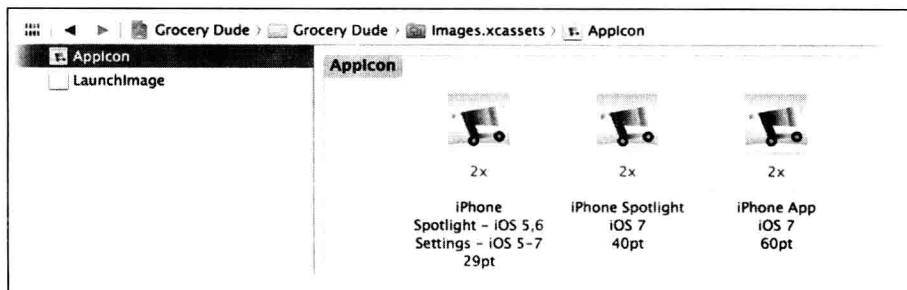


图 A-7 App Icon

请按下列步骤修改 Grocery Dude，为其配置好适当的 launch image（启动画面）：

1. 确保 **Images.xcassets** 仍然处于选中状态。
 2. 选择 **LaunchImage**。
 3. 将 **LaunchImage_2x.png** 拖放到写有 **iPhone Portrait-2x** 字样的虚线框中。
 4. 将 **LaunchImage_R4.png** 拖放到写有 **iPhone Portrait-R4** 字样的虚线框中。
- 设置好的 LaunchImage 如图 A-8 所示。



图 A-8 Launch Image

第 1 章要用到的范例项目现在已经准备好了，你可以在 iOS 仿真器或 iOS 设备上面运行 Grocery Dude，看看自己目前取得的成果！请注意，由于 deployment target（部署目标）是 iOS 7，所以早前版本的 iOS 是不能运行 Grocery Dude 程序的。



为第 16 章的 Grocery Cloud 程序所做的 的准备工作

在第 16 章中，我们要把 StackMob 集成到名叫 Grocery Cloud 的新项目里。本附录演示如何根据第 12 章末尾的 Grocery Dude 来准备这个初始的 Grocery Cloud 项目。由于这部分内容与 Core Data 无关，所以笔者将其放在附录中。读者不一定要按本附录操作一遍，也可以从第 16 章所给的网址中把 Grocery Cloud 项目下载下来。必须使用 Xcode 5 或后续版本才能执行接下来的这些操作。

B.1 修改 Grocery Dude 项目的名称

首先我们要把 Grocery Dude 项目重命名为 Grocery Cloud，使这两个项目不会相互混淆。请按下列步骤把 Grocery Dude 改名为 Grocery Cloud：

1. 从 <http://www.timroadley.com/LearningCoreData/GroceryDude-AfterChapter12.zip> 把第 12 章末尾的 **Grocery Dude** 项目下载下来，并将 .zip 文件解压缩。
2. 将含有项目内容的文件夹从 Grocery Dude 改名为 **Grocery Cloud**。改完名称之后，注意别把文件夹里面的那个 Grocery Dude 目录名也改了。
3. 双击 Grocery Cloud 文件夹里的 **Grocery Dude.xcodeproj** 文件，用 Xcode 5 或后续版本打开此项目。
4. 点击 **Product > Clean** 菜单项，清除其他项目所残留的缓存。
5. 确保 **Project Navigator** 界面处于可见状态（可以按“⌘ + 1”组合键把该界面显示出

来), 然后缓慢地双击项目名称, 将其重命名为 **Grocery Cloud**, 如图 B-1 左上角所示。刚修改完项目名称的时候, 会弹出如图 B-1 右侧所示的窗口。

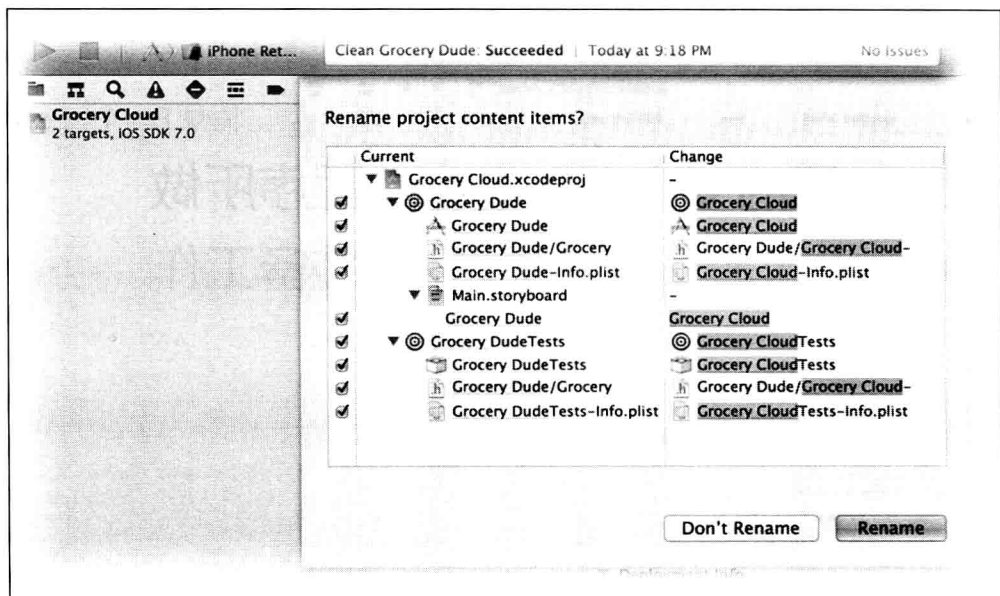


图 B-1 把项目名称从 Grocery Dude 改为 Grocery Cloud

6. 点击 **Rename** 按钮。如果 Xcode 弹出警告框, 提示说 **Main.storyboard** 已经改变了, 那就点击 **Revert** 按钮。

7. 把 **Grocery Cloud > Targets > Grocery Cloud > Info > Bundle display name** 从 `$(PRODUCT_NAME)` 改为 **GroceryCloud**。

B.2 重新指定文件路径

等执行完重命名的步骤之后, 项目里面就会有 Info.plist 及 Precompiled Prefix Header 文件了, 而我们现在需要把两者的路径修改好。

请按下列步骤修改 Grocery Cloud 项目, 以便重新指定这两个文件的路径:

1. 选中 Grocery Cloud 这个 target, 切换到 **Build Settings** 分页, 并点击 **Basic** 按钮, 如图 B-2 所示。

2. 将 **Grocery Cloud-Info.plist** 文件与 **Grocery Cloud-Prefix.pch** 文件所在的目录从 **Grocery Dude** 改为 **Grocery Cloud**, 如图 B-2 所示。

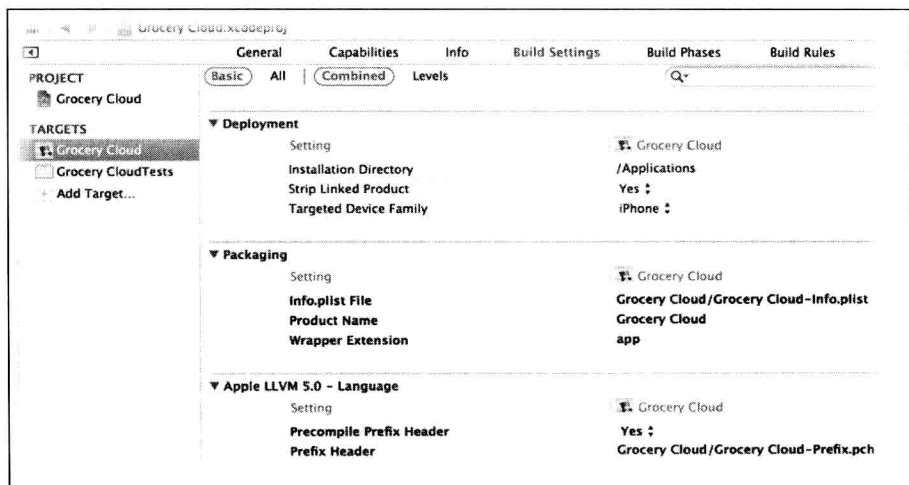


图 B-2 修改相关文件所在的路径

B.3 修改组与 tests 的名称

Xcode 使用组来组织项目的类文件。我们现在要修改这些组的名称，使其与新的项目名称相符。

请按下列步骤修改 Grocery Cloud 项目，为 Xcode 中的组重新命名：

1. 修改项目里每个组的名字，令其以 **Grocery Cloud** 开头。然后把 **Grocery_DudeTests.m** 改名为 **Grocery_CloudTests.m**，如图 B-3 所示。

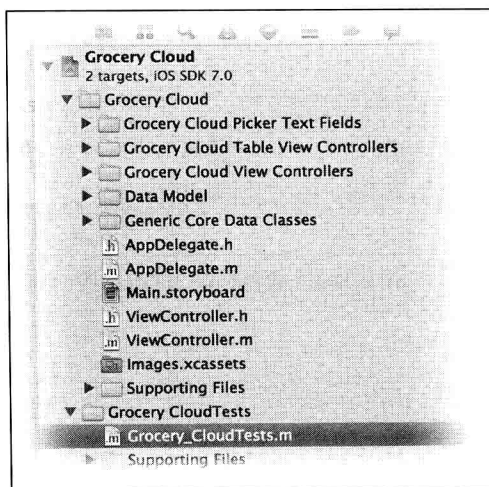


图 B-3 把组名称及 tests 名称中的 Grocery Dude 改为 Grocery Cloud

2. 右击图 B-3 顶部的 **Grocery Cloud** 组，然后选择 **Show in Finder**。

3. 在 Finder 中，把 **Grocery Dude** 目录的名称改为 **Grocery Cloud**。修改之后，Xcode 里的某些文件会显示成红色，因为其上级路径目前是无效的。

4. 在 Finder 中，把 **Grocery DudeTests** 目录的名称改为 **Grocery CloudTests**。由于上级路径无效，所以 Xcode 里面还会有一些文件也显示为红色。

5. 返回 Xcode，选中图 B-3 顶端的 **Grocery Cloud** 组。打开 **File Inspector** 界面（可按“**Option + ⌘ + 1**”组合键调出该界面），参照图 B-4，点击 **Identity and Type** 区域里的那个图标。

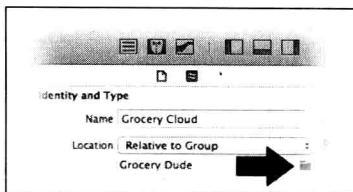


图 B-4 重新指定 Grocery Cloud 组的位置

6. 选择与 **Grocery Cloud.xcodeproj** 处在同一个级别的 **Grocery Cloud** 目录，用这个目录来表示 Xcode 中的 **Grocery Cloud** 组。该组里面原来显示为红色的那些文件现在应该变回黑色了。

7. 重复第 5 步和第 6 步。这次是把 **Grocery CloudTests** 组的位置设为重命名之后的 **Grocery CloudTests** 目录。

B.4 重命名 Scheme

本项目的 scheme 目前叫做 **Grocery Dude**。为了在项目中保持一致，我们将其改名为 **Grocery Cloud**。

请按下列步骤修改 Grocery Cloud 项目：

1. 点击 **Product > Scheme > Manage Schemes...** 菜单项。

2. 缓慢地双击 scheme 的名称，将其重命名为 **Grocery Cloud**，如图 B-5 所示，然后点击 **OK** 按钮。

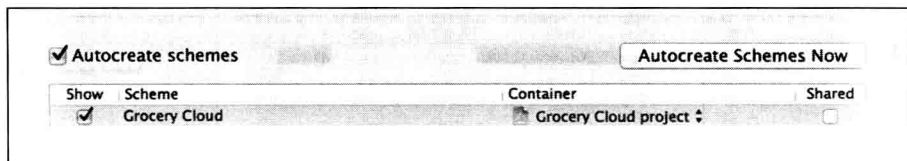


图 B-5 把 scheme 的名称改为 Grocery Cloud

B.5 修改图片资源

为了与 Grocery Dude 项目相区别，Grocery Cloud 将使用另一套图标及启动画面（splash）。

请按下列步骤修改 Grocery Cloud 项目，用新的图片来取代原有的图片：

1. 从 http://www.timroadley.com/LearningCoreData/Icons_GroceryCloud.zip 下载 .zip 文件，该文件里包含新的图片资源。
2. 选定名为 **Images.xcassets** 的 asset catalog。
3. 选定 **AppIcon**。
4. 把 **AppIcon_29pt.png** 拖放到写有 **29pt** 字样的方框中，以取代原有的图标。
5. 把 **AppIcon_40pt.png** 拖放到写有 **40pt** 字样的方框中，以取代原有的图标。
6. 把 **AppIcon_60pt.png** 拖放到写有 **60pt** 字样的方框中，以取代原有的图标。
7. 选定 **LaunchImage**。
8. 把 **LaunchImage_2x.png** 拖放到写有 **2x** 字样的方框中，以取代原有的图像。
9. 把 **LaunchImage_R4.png** 拖放到写有 **R4** 字样的方框中，以取代原有的图像。

B.6 禁用拍照及显示货品图像功能

为了把第 16 章的项目做得精简一些，也为了使 Grocery Cloud 能够作为免费程序在 App Store 上架，我们需要移除 Grocery Cloud 程序对图像的支持。为此，我们要把 `PrepareTVC.m` 及 `ShopTVC.m` 文件中的 `viewDidAppear` 方法注释掉，使其不再触发“缩略图生成”操作。笔者并不会把这些代码从文件里移除，这样一来，读者稍后就能够给程序自行添加图像支持了。

请按下列步骤修改 Grocery Cloud 项目，以便禁用图像支持功能：

1. 把 `PrepareTVC.m` 文件及 `ShopTVC.m` 文件里的 `viewDidAppear` 方法注释掉。
2. 修改 `ItemVC.m` 文件中的 `viewDidLoad` 方法，把下列代码添加到该方法底部：

```
self.cameraButton.hidden = YES;
self.photoImageView.hidden = YES;
```

B.7 解决 `sectionNameKeyPath` 问题

StackMob 提供了一套框架，我们在第 16 章将会用这套框架把 Core Data 程序同后端的 web service 集成起来。StackMob 框架有个限制，它不能把“关系”设置成 key path，而 `UITableView` 却得根据 key path 来决定应该把信息显示在哪个部分里面。不巧的是，我们的 Grocery Cloud 恰恰需要用到这个功能。所以，为了解决此问题，我们要向 Item 实体里添加两个字符串类型的属性。程序将会根据与之等价的关系来填充属性的内容，使其能够表示货品的位置，而表格视图也将据此把货品划分到相应的部分里面。请注意，在程序尚未运行到解决 `sectionNameKeyPath` 问题所用的那段代码时，控制台日志中可能会出现写有“Object will be placed in unnamed section”（对象将会放入未命名的部分）字样的错误消息。

请按下列步骤修改 Grocery Cloud 项目，以解决 `sectionNameKeyPath` 问题：

1. 选定 **Model.xcdatamodeld**。
2. 点击 **Editor > Add Model Version...**。
3. 点击 **Finish** 按钮，把 **Model 9** 用作新 model 的名称。
4. 确保 **Model 9.xcdatamodel** 处于受选状态。
5. 在 **Item** 实体中创建 **String** 型属性，将其命名为 **storedIn**。
6. 在 **Item** 实体中创建 **String** 型属性，将其命名为 **aisle**。
7. 选中 **Model 9** 里的所有实体，然后点击 **Editor > Create NSManagedObject Subclass...** 菜单项，依照提示为所有实体重新生成 `NSManagedObject` 子类文件，并覆盖原有文件。记得先勾选 **Targets** 中的 “Grocery Cloud”，然后再点击 **Create** 按钮。
8. 确保 **Model.xcdatamodeld** 处于受选状态。
9. 用 **File Inspector** 界面（可以按 “**Option** + ⌘ + 1” 组合键调出该界面）把 **Current Model Version** 设为 **Model 9**。

程序清单 B-1 中列出的这段代码可以把与某件货品有关的地点名称拷贝到 **Item** 实体对应的属性里面。

程序清单B-1 PrepareTVC.m文件中的cellForRowAtIndexPath方法

```
// StackMob Relationship sectionNameKeyPath Workaround
[self.frc.managedObjectContext performBlock:^(
    if (!item.storedIn ||
        ![item.storedIn isEqualToString:item.locationAtHome.storedIn]) {
        item.storedIn = item.locationAtHome.storedIn;
        NSLog(@"sectionNameKeyPath WORKAROUND (See Appendix B):");
        NSLog(@"item.storedIn is now = '%@'", item.storedIn);
    }
    if (!item.aisle ||
        ![item.aisle isEqualToString:item.locationAtShop.aisle]) {
        item.aisle = item.locationAtShop.aisle;
        NSLog(@"sectionNameKeyPath WORKAROUND (See Appendix B):");
        NSLog(@"item.aisle is now = '%@'", item.aisle);
    }
});
```

请按下列步骤修改 Grocery Cloud 项目，以解决 `sectionNameKeyPath` 问题：

1. 把 `#import "LocationAtHome.h"` 语句添加到 `PrepareTVC.m` 文件顶部。
2. 把 `#import "LocationAtShop.h"` 语句添加到 `PrepareTVC.m` 文件顶部。
3. 把程序清单 B-1 中的代码拷贝到 `PrepareTVC.m` 文件 `cellForRowAtIndexPath` 方法底部的 `return cell;` 语句上方。
4. 重复执行第 1、2、3 步，这次针对 `ShopTVC.m` 文件，而不是 `PrepareTVC.m` 文件。由于现在已经无须依赖 `locationAtHome.storedIn` 及 `locationAtShop.aisle` 了，所以应该修改 `configureFetch` 方法里面与 `sectionNameKeyPath` 有关的代码。

请按下列步骤修改 Grocery Cloud 项目，使 `sectionNameKeyPath` 不再依赖于“关系”：

1. 修改 `PrepareTVC.m` 文件中的 `configureFetch` 方法，把该方法里的两个 `locationAtHome.storedIn` 都替换成 `storedIn`。
2. 修改 `ShopTVC.m` 文件中的 `configureFetch` 方法，把该方法里的两个 `locationAtShop.aisle` 都替换成 `aisle`。

B.8 小结

供第 16 章所用的起始项目现在已经创建好了。请注意，我们不再需要使用设备本地的持久化存储区了。假如设备上已经装有 Grocery Cloud 程序（可能是从 App Store 安装的），那么笔者推荐在阅读第 16 章之前先将其移除。如果现在运行应用程序，那么控制台日志里会出现许多与解决 `sectionNameKeyPath` 问题有关的消息。

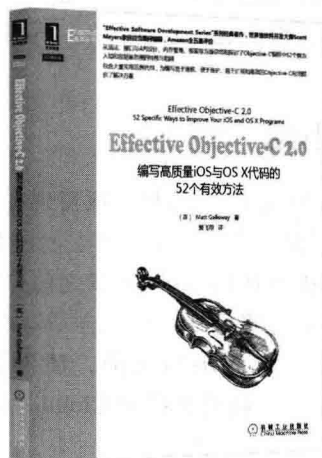
开始学习第 16 章之前，请再做最后一次修改：在 Xcode 里面搜索项目中所出现的 `Dude` 一词，并将其替换为 `Cloud`。

推荐阅读



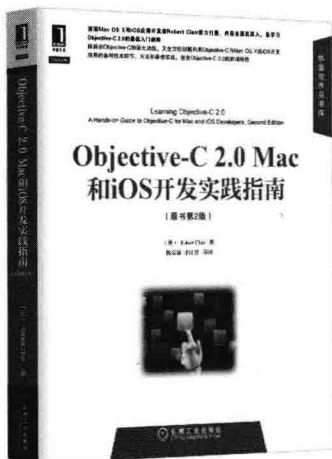
iOS应用软件开发之道

作者: William Van Hecke ISBN: 978-7-111-47833-1 定价: 69.00元



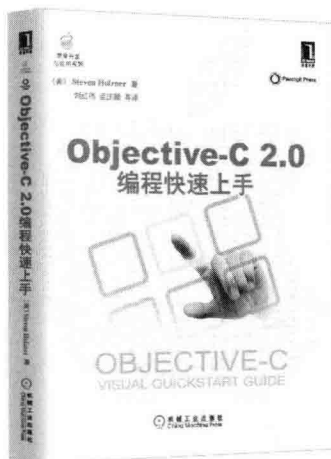
Effective Objective-C 2.0: 编写高质量iOS与OS X代码的52个有效方法

作者: Matt Galloway ISBN: 978-7-111-45129-7 定价: 69.00元



Objective-C 2.0 Mac和iOS开发实践指南 (原书第2版)

作者: Robert Clair ISBN: 978-7-111-48456-1 定价: 79.00元



Objective-C 2.0编程快速上手

作者: Steven Holzner ISBN: 978-7-111-30874-4 定价: 35.00元

这是一本全面涵盖苹果开发平台新特性的Core Data教程，其中讲到了Apple对iCloud所做的重大改进。本书按步骤讲解程序开发的全过程，引领读者使用Storyboard、ARC（自动引用计数）和Xcode 创建出一款由数据所驱动的iOS 应用程序。

书中介绍了一些新的编程范式及开发技巧，帮助读者克服Core Data开发中的各种困难。开发过程分为很多小的步骤，在学习这些步骤的过程中，你将不断丰富自己的编程技能，而且还能掌握一些高阶技巧，例如复杂模型的迁移、深拷贝、后台处理，以及同Dropbox、StackMob、iCloud的集成等。

本书每一章都提供了网址，读者可以从中下载与该章进度相对应的范例项目，以便准确地了解项目在每个阶段的状况，另外，你也可以把这种按步骤实现出来的代码运用到自己的项目中。每章后面都有一些习题，无论你是自学者，还是参加iOS开发课程的学生，都可以通过这些题目来进一步探索Core Data领域。

本书提供了许多技巧、工具、代码和编程范式，如果你是一位有经验的iOS开发者，那么可以利用这些内容，轻而易举地为任意一款应用程序迅速添加强大的数据管理功能。

通过阅读本书，你将学到：

- 理解Core Data的概念
- 为既有项目添加Core Data支持
- 设计数据模型、升级数据模型、迁移数据模型（包括自动迁移和能够显示出迁移进度的手动迁移）
- 用数据来填充表格视图、选取器视图等视图界面
- 预先把一些默认数据放在XML格式的持久化存储区里，然后在程序运行时将其加载进来
- 通过深拷贝的方式，把一个持久化存储区中的数据复制到另一个里面
- 以大尺寸照片为例，演示如何通过Instruments等工具来优化程序性能
- 以缩略图的生成为例，演示如何实现后台处理
- 实现高效的搜索功能
- 与Dropbox相结合，实现流畅的数据备份及数据恢复功能
- 与iCloud稳固地集成起来，以便完全支持多账号登录、数据散播和去除重复数据等功能
- 以StackMob为例，演示如何把Core Data程序同Web服务相集成



PEARSON

www.pearson.com

投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/程序设计/移动开发

ISBN 978-7-111-48226-0



9 787111 482260 >

定价: 79.00元