

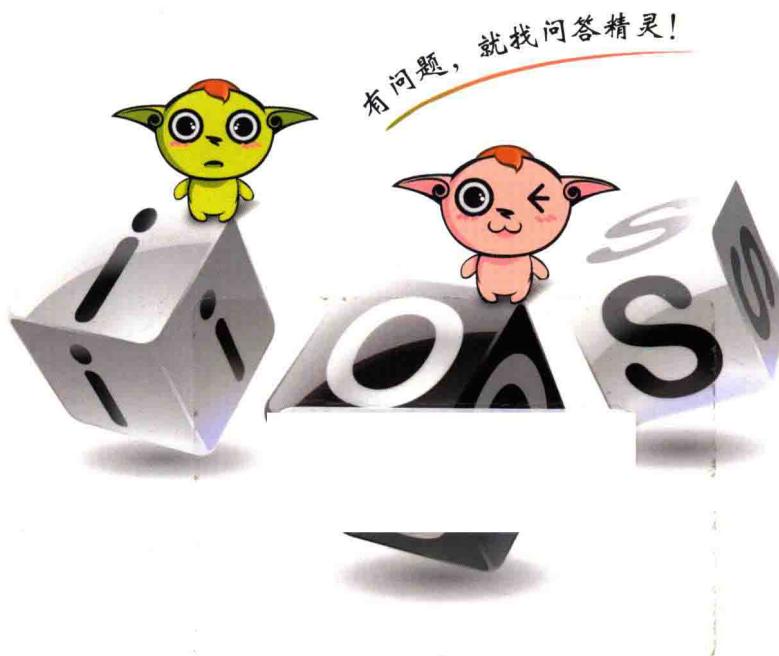
NITE 国家信息技术紧缺人才培养工程指定教材



工业和信息化人才培养规划教材

iOS开发项目化经典教程

传智播客高教产品研发部 编著



本书涵盖了iOS开发的中、高级技术及其当前流行的实用技术，提供了62个项目实战和20道iOS面试题。

提供免费教学资源，包括精美教学PPT、1068道测试题、长达32小时的教学视频等。

关注“wendajingling”微信公众号，享受最专业的免费技术答疑，同时享有一对一学习指导和职业规划服务。



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS



工业和信息化人才培养规划教材

出版日期：2013年1月

工业和信息化人才培养规划教材
出版日期：2013年1月

ISBN 978-7-115-38767-7

iOS开发项目化经典教程

传智播客高教产品研发部 编著

人民邮电出版社
北京

图书在版编目(CIP)数据

iOS开发项目化经典教程 / 传智播客高教产品研发部
编著. — 北京 : 人民邮电出版社, 2016. 2
工业和信息化人才培养规划教材
ISBN 978-7-115-41074-0

I. ①i... II. ①传... III. ①移动终端—应用程序—
程序设计—教材 IV. ①TN929.53

中国版本图书馆CIP数据核字(2016)第016271号

内 容 提 要

本书系统全面地讲解了 iOS 开发的中、高级知识，主要内容包括多线程编程、网络编程、iPad 开发、多媒体硬件、Address Book、使用 Mapkit 开发地图服务、推送机制、内购、广告、指纹识别、屏幕适配及国际化等。

本书采用项目驱动的方式来讲解理论。全书共有 60 余个经典的真实项目，这些项目可以帮助读者更好地理解各个知识点在实际开发中的应用，也可以供读者开发时作为参考。

本书附有配套视频、源代码、习题、教学课件等资源，而且为了帮助初学者更好地学习本教材中的内容，我们还提供了在线答疑，希望得到更多读者的关注。

本书既可作为高等院校本、专科计算机相关的程序设计课程教材，也可作为 iOS 技术提升的培训教材，适合有一定 iOS 开发基础的读者使用。

-
- ◆ 编 著 传智播客高教产品研发部
 - 责任编辑 范博涛
 - 责任印制 杨林杰
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 三河市海波印务有限公司印刷
 - ◆ 开本：787×1092 1/16
 - 印张：23.5 2016 年 2 月第 1 版
 - 字数：607 千字 2016 年 2 月河北第 1 次印刷
-

定价：49.80 元

读者服务热线：(010)81055256 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

目 录 CONTENTS

第 1 章 多线程编程 1

1.1 多线程概念	1	1.3.4 实战演练——使用 GCD 下载图片	30
1.1.1 多线程概述	1	1.3.5 单次或重复执行任务	32
1.1.2 线程的串行和并行	4	1.3.6 调度队列组	34
1.1.3 多线程技术种类	5	1.4 NSOperation 和 NSOperationQueue	37
1.2 使用 NSThread 实现多线程	5	1.4.1 NSOperation 简介	37
1.2.1 线程的创建和启动	5	1.4.2 NSOperationQueue 简介	39
1.2.2 线程的状态	9	1.4.3 使用 NSOperation 子类操作	42
1.2.3 线程间的安全隐患	11	1.4.4 实战演练——自定义 NSOperation 子类下载图片	44
1.2.4 线程间的通信	15	1.4.5 实战演练——对 NSOperation 操作设置依赖关系	47
1.3 使用 GCD 实现多线程	17	1.4.6 实战演练——模拟暂停和继续操作	48
1.3.1 GCD 简介	17	1.5 本章小结	50
1.3.2 创建队列	19		
1.3.3 提交任务	20		

第 2 章 网络编程 51

2.1 网络基本概念	51	2.3.6 解析 JSON 文档	88
2.1.1 网络编程的原理	51	2.3.7 实战演练——使用 NSJSONSerialization 解析天气预报	89
2.1.2 URL 介绍	52	2.4 HTTP 请求	91
2.1.3 TCP/IP 和 TCP、UDP	53	2.4.1 HTTP 和 HTTPS	92
2.1.4 Socket 介绍	55	2.4.2 GET 和 POST 方法	92
2.1.5 实战演练——Socket 聊天	57	2.4.3 实战演练——模拟 POST 用户登录	94
2.2 原生网络框架 NSURLConnection	62	2.4.4 数据安全——MD5 算法	98
2.2.1 NSURLRequest 类	62	2.4.5 钥匙串访问	101
2.2.2 NSURLConnection 介绍	63	2.4.6 实战演练——模拟用户安全登录	101
2.2.3 Web 视图	65	2.5 文件的上传与下载	109
2.2.4 实战演练——Web 视图加载 百度页面	67	2.5.1 上传文件的原理	109
2.3 数据解析	70	2.5.2 实战演练——上传单个文件	112
2.3.1 配置 Apache 服务器	70	2.5.3 实战演练——上传多个文件	115
2.3.2 XML 文档结构	74	2.5.4 NSURLConnection 下载	118
2.3.3 解析 XML 文档	75	2.5.5 NSURLSession 介绍	121
2.3.4 实战演练——使用 NSXMLParser 解析 XML 文档	75	2.5.6 实战演练——使用 NSURLSession 实现下载功能	123
2.3.5 JSON 文档结构	87	2.6 第三方框架	127

2.6.1 SDWebImage 介绍	127	框架	131
2.6.2 AFNetworking 和 ASIHTTPRequest		2.7 本章小结	133

第 3 章 iPad 开发 135

3.1 iPhone 和 iPad 开发的异同	135	3.3 UISplitViewController	154
3.2 UIPopoverController	137	3.3.1 UISplitViewController 简介	154
3.2.1 UIPopoverController 简介	137	3.3.2 UISplitViewController 的使用	156
3.2.2 UIPopoverController 的使用	139	3.3.3 实战演练——菜谱	158
3.2.3 实战演练——弹出 Popover 视图	142	3.4 本章小结	172

第 4 章 多媒体和硬件 173

4.1 使用 AVAudioRecorder 录制音频	173	4.4 使用 MPMoviePlayerController	
4.2 音效、音频的播放	176	播放视频	199
4.2.1 使用系统声音服务播放音效	176	4.5 扫描二维码	203
4.2.2 使用 AVAudioPlayer 播放音乐	177	4.6 传感器、陀螺仪、加速计	206
4.2.3 使用 MPMediaPickerController		4.6.1 传感器介绍	206
选择系统音乐	180	4.6.2 距离传感器	206
4.2.4 播放在线音乐	182	4.6.3 陀螺仪介绍	207
4.2.5 实战演练——音乐播放器	185	4.6.4 加速计	210
4.3 相机和图库	193	4.6.5 实战演练——计步器	213
4.3.1 使用 UIImagePickerController		4.7 蓝牙	215
操作摄像头和照片库	193	4.8 本章小结	219
4.3.2 实战演练——拍照和相片库	196		

第 5 章 Address Book 220

5.1 iOS 7 及 iOS 8 的联系人管理框架	220	5.3 iOS 9 中管理联系人的新框架	236
5.1.1 使用 Address Book 框架管理联系人	220	5.3.1 使用 Contacts 框架管理联系人	236
5.1.2 使用 Address BookUI 框架管理联系人	225	5.3.2 使用 ContactsUI 框架管理联系人	240
5.2 实战演练——使用 UIApplication		5.4 本章小结	243
打电话和发短信	229		

第 6 章 使用 MapKit 开发地图服务 244

6.1 根据地址定位	244	6.2.2 指定地图显示中心和显示区域	253
6.1.1 根据地址定位	245	6.2.3 使用 iOS 7 新增的 MKMapView	255
6.1.2 正向地理编码和反向地理编码	249	6.3 在地图上添加锚点	257
6.2 MapKit 框架	251	6.3.1 添加简单的锚点	257
6.2.1 MKMapView 控件	251	6.3.2 添加自定义锚点	259

6.4 使用 iOS 7 新增的 MKTile Overlay 覆盖层	262	6.6 实战演练——行车导航仪	268
6.5 使用 iOS 7 新增的 MKDirections 获取导航路线	264	6.7 第三方使用——百度地图	272
		6.8 本章小结	278

第 7 章 推送机制 279

7.1 推送机制概述	279	7.4 iOS 远程推送通知	291
7.2 iOS 本地通知	281	7.5 极光推送	297
7.3 实战演练——闹钟	283	7.6 本章小结	302

第 8 章 内购、广告和指纹识别 303

8.1 内购	303	8.2 广告	321
8.1.1 在 App Store 上的准备工作	304	8.3 指纹识别	323
8.1.2 实现内购功能	318	8.4 本章小结	327

第 9 章 屏幕适配 328

9.1 屏幕适配历史背景介绍	328	9.4 Size Class	343
9.2 Autoresizing	330	9.4.1 在 Interface Builder 中使用 Size Class	343
9.2.1 在 Interface Builder 中使用 Autoresizing	330	9.4.2 实战演练——使用 Size Class 布局 QQ 登录界面	345
9.2.2 在代码中设置 AutoresizingMask 属性	333	9.5 第三方框架——Masonry 框架	347
9.3 Auto Layout	336	9.5.1 Masonry 框架介绍	347
9.3.1 在 Interface Builder 中管理 Auto Layout	336	9.5.2 Masonry 框架的使用	349
9.3.2 实战演练——使用 Auto Layout 布局界面	338	9.6 本章小结	352

第 10 章 国际化 353

10.1 概述	353	10.4 文本信息国际化	361
10.2 国际化应用程序显示名称	355	10.5 程序内部切换语言	363
10.3 国际化界面设计	359	10.6 本章小结	365

学习目标



- 理解多线程的概念
- 了解实现多线程的 4 种方式
- 掌握线程间的安全和通信
- 掌握 GCD 的基本操作
- 掌握 NSOperation 的基本操作

应用程序在运行时经常要同时处理多项任务，如一个音乐应用，在播放音乐的同时，用户还可以不停地下载歌曲、搜索歌曲等。也就是说，音乐应用同时进行着播放音乐、下载音乐和接受用户响应等多项任务。系统使用线程对任务进行处理，一条线程同一时间只能处理一个任务。多个任务同时执行就需要多条线程。iOS 平台对多线程提供了非常优秀的支持，本章将针对 iOS 系统中的多线程编程进行详细的讲解。

1.1 多线程概念

由于一条线程同一时间只能处理一个任务，所以一个线程里的任务必须顺序执行。如果遇到耗时操作（如网络通信、耗时运算、音乐播放等），等上一个操作完成再执行下一个任务，则在此时间内用户得不到任何响应，这是很糟糕的用户体验。因此在 iOS 编程中，通常将耗时操作单独放在一个线程里，而把与用户交互的操作放在主线程里，保证应用及时响应用户的操作，提高用户体验。接下来，将围绕多线程技术进行详细讲解。

1.1.1 多线程概述

多线程是指从软件或者硬件上实现多个线程并发执行的技术。多线程技术使得计算机能够在同一时间执行多个线程，从而提升其整体处理性能。

想要了解多线程，必须先理解进程和线程的概念。进程是指在系统中正在运行的应用程序。这个“运行中的程序”就是一个进程。每个进程拥有着自己的地址空间。

当一个程序进入内存运行时，就会变成一个进程，运行中的每个程序都对应着一个进程，且进程具有一定的独立功能。打开 Mac 的活动监测器，可以看到当前系统执行的进程，如图 1-1 所示。

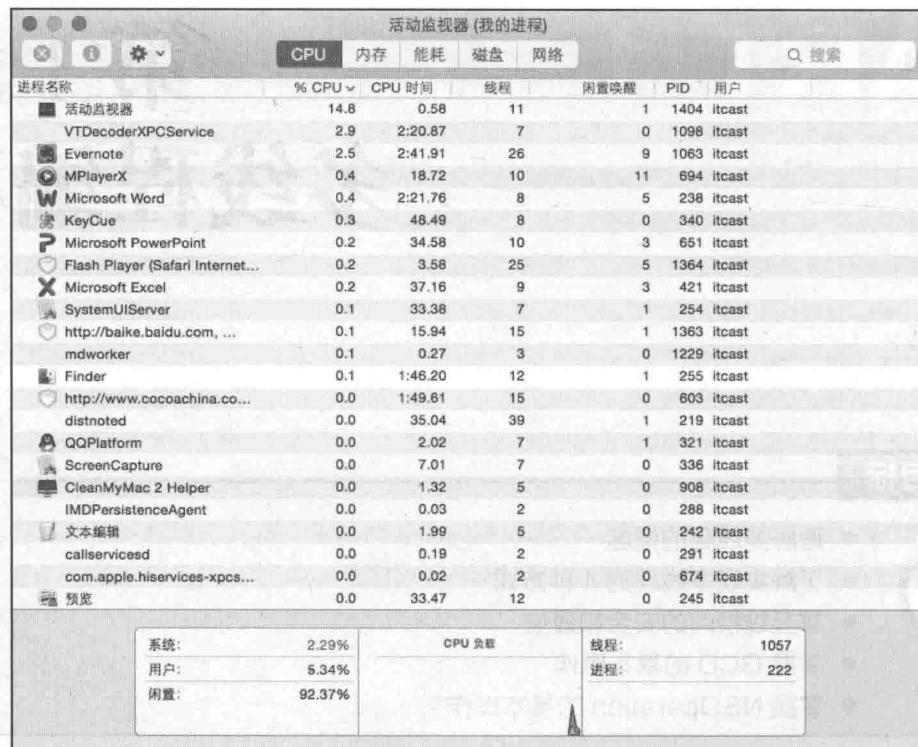


图 1-1 Mac 系统当前的进程

由图 1-1 可知, 该窗口第 1 列显示的是“进程名称”, 该列的每个应用程序就是一个单独的进程。第 4 列显示的是当前进程拥有的线程数量, 而且不止一条。

进程作为系统进行分配和调度的一个独立单位, 它主要包含以下 3 个主要特征。

(1) 独立性

进程是一个能够独立运行的基本单位, 它既拥有自己独立的资源, 又拥有着自己私有的地址空间。在没有经过进程本身允许的情况下, 一个用户的进程是不可以直接访问其他进程的地址空间的。

(2) 动态性

进程的实质是程序在系统中的一次执行过程, 程序只是一个静态的指令集合, 而进程是一个正在系统中活动的指令集合。在进程中加入了时间的概念, 它就具有自己的生命周期和各自不同的状态, 进程是动态消亡的。

(3) 并发性

多个进程可以在单个处理器中同时执行, 而不会相互影响, 并发就是同时进行的意思。

需要注意的是, CPU 在某一个时间点只能执行一个程序, 即只能运行一个单独的进程, CPU 会不断地在这些进程之间轮换执行。假如 Mac 同时运行着 QQ、music、PPT、film4 个程序, 它们在 CPU 中所对应的进程如图 1-2 所示。

图 1-2 展示了 CPU 在多个进程间切换执行的效果, 由于 CPU 的执行速度相对人的感觉而言太快, 造成了多个程序同时运行的假象。如果启动了足够多个程序, CPU 就会在这么多个程序间切换, 这时用户会明显感觉到程序的运行速度下降。

多线程扩展了多进程的概念, 使得同一个进程可以同时并发处理多个任务。一个程序的运行至少要有一个线程, 一个进程要想执行任务, 必须依靠至少一个线程, 这个线程就被称

作主线程。线程是进程的基本执行单元，对于大多数应用程序而言，通常只有一个主线程。

当进程被初始化之后，主线程就被创建了，主线程是其他线程最终的父线程，所有界面的显示操作必须在主线程进行。开发者可以创建多个子线程，每条线程之间是相互独立的。假如 QQ 进程中有 3 个线程，分别用来处理数据发送、数据显示、数据接收，则它们在进程中的关系如图 1-3 所示。

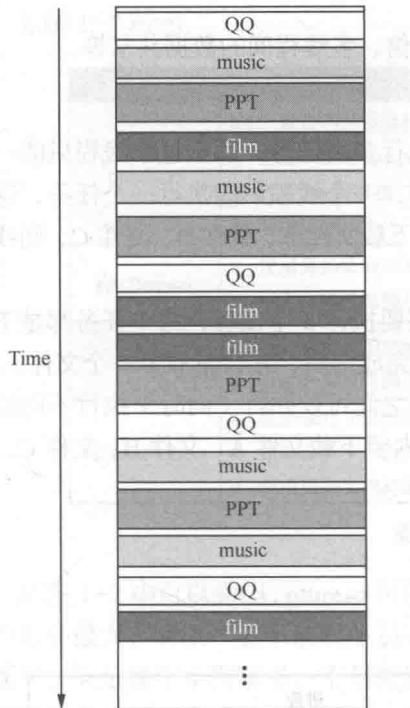


图 1-2 CPU 执行的进程

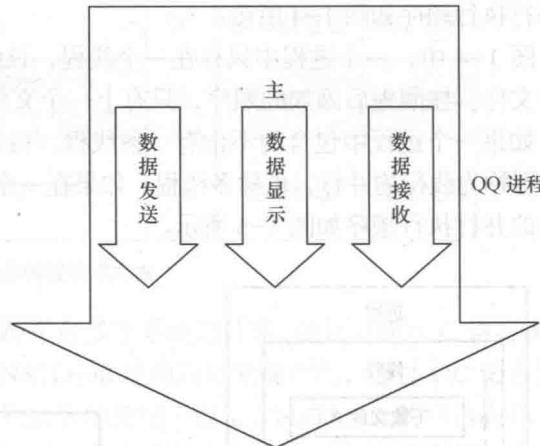


图 1-3 进程与线程

由图 1-3 可知，一个进程必定有一个主线程，每个线程之间是独立运行的，因此，线程的执行是抢占式的，只有当前的线程被挂起，另一个线程才可以运行。

一个进程中包含若干个线程，这些线程可以利用进程所拥有的资源。通常都是把进程作为分配资源的基本单位，把线程作为独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，因此对它进行调度所付出的开销就会小得多，能更高效地提高系统内多个程序间并发执行的程度。

当操作系统创建一个进程的时候，必须为进程分配独立的内存空间，并且分配大量的相关资源。但是创建一个线程则要简单得多，因此使用多线程实现并发要比使用多进程性能高很多。总体而言，使用多线程编程有以下优势。

- (1) 进程间不能共享内存，但是线程之间共享内存是非常容易的。
- (2) 当硬件处理器的数量有所增加时，程序运行的速度更快，无需做其他任何调整。
- (3) 充分发挥多核处理器的优势，将不同的任务分配给不同的处理器，真正进入“并行运算”的状态。
- (4) 将耗时、轮询或者并发需求高的任务分配到其他线程执行，而主线程则负责统一更新界面，这样使得应用程序更加流畅，用户的体验更好。

凡事有利必有弊，多线程既有着一定的优势，同样也有着一定的劣势。如何扬长避短，

这是开发者需要注意的问题。多线程的劣势包括以下 3 方面。

(1) 开启线程需要占用一定的内存空间(默认情况下,主线程最大占用 1M 的栈区空间,子线程最大占用 512K 的栈区空间),如果要开启大量的线程,势必会占用大量的内存空间,从而降低程序的性能。

(2) 开启的线程越多,CPU 在调度线程上的开销就会越大,一般最好不要同时开启超过 5 个线程。

(3) 程序的设计会变得更加复杂,如线程之间的通信、多线程间的数据共享等。

1.1.2 线程的串行和并行

如果在一个进程中只有一个线程,而这个进程要执行多个任务,那么这个线程只能一个一个地按顺序执行这些任务,也就是说,在同一个时间内,一个线程只能执行一个任务,这样的线程执行方式称为线程的串行。如果在一个进程内要下载文件 A、文件 B、文件 C,则线程的串行执行顺序如图 1-4 所示。

图 1-4 中,一个进程中只存在一个线程,该线程需要执行 3 个任务,每个任务都是下载一个文件。按照先后添加的顺序,只有上一个文件下载完成之后,才会下载下一个文件。

如果一个进程中包含的不止有一条线程,每条线程之间可以并行(同时)执行不同的任务,则称为线程的并行,也称多线程。如果在一个进程内要下载文件 A、文件 B、文件 C,则线程的并行执行顺序如图 1-5 所示。

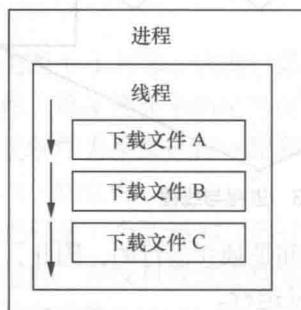


图 1-4 线程的串行

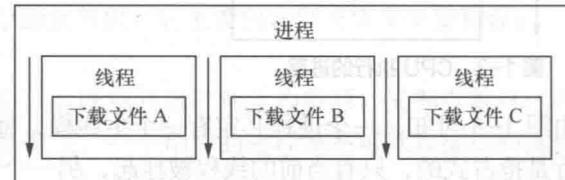


图 1-5 线程的并行

图 1-5 中,一个进程拥有 3 个线程,每个线程执行 1 个任务,3 个下载任务没有先后顺序,可以同时执行。

同一时间,CPU 只能处理一个线程,也就是只有一个线程在工作。由于 CPU 快速地在多个线程之间调度,人眼无法感觉到,就造成了多线程并发执行的假象,多线程原理图如图 1-6 所示。



图 1-6 多线程的原理

由图 1-6 可知,一个进程拥有 3 个线程,分别为线程 A、线程 B、线程 C。某一时刻,

CPU 执行线程 A 为一个小箭头的时间，之后又切换到线程 B、线程 C，依次类推，这样就形成了多条线程并发执行的效果。

1.1.3 多线程技术种类

iOS 提供了 4 种实现多线程的技术，这些技术各有侧重，既有着一定的优势，也存在着各自的不足，开发者可根据自己的实际情况选择。接下来，通过一张图来描述这 4 种技术方案，如图 1-7 所示。

技术方案	简介	语言	线程生命周期	使用频率
pthread	<ul style="list-style-type: none"> ■ 一套通用的多线程 API ■ 适用于 Unix\Linux\Windows 等系统 ■ 跨平台\可移植 ■ 使用难度大 	C	程序员管理	几乎不用
NSThread	<ul style="list-style-type: none"> ■ 使用更加面向对象 ■ 可直接操作线程对象 	OC	程序员管理	偶尔使用
GCD	<ul style="list-style-type: none"> ■ 旨在替代 NSThread 等线程技术 ■ 充分利用设备的多核 	C	自动管理	经常使用
NSOperation	<ul style="list-style-type: none"> ■ 基于 GCD (底层是 GCD) ■ 比 GCD 多了一些更简单实用的功能 ■ 使用更加面向对象 	OC	自动管理	经常使用

图 1-7 多线程技术方案

从图 1-7 中可以看出，pthread 可以实现跨平台多个系统的开发，但是它通过 C 语言操作，使用难度很大，所以一般不被程序员采用。NSThread 是面向对象操作的，通过 OC 语言来执行操作，但是操作步骤繁多，不易控制，也只是偶尔使用。但是，NSThread 的内容有助于初学者理解多线程的本质和实现原理，后面会针对它进行详细的介绍。GCD 充分利用了设备的多核优势，用于替代 NSThread 技术。NSOperation 基于 GCD，更加面向对象。GCD 和 NSOperation 都是自动管理线程的，在实际开发中更受开发者推崇。后面的小节也会对 GCD 和 NSOperation 进行详细的介绍。

1.2 使用 NSThread 实现多线程

前面已经简单介绍过，NSThread 类是实现多线程的一种方案，也是实现多线程的最简单方式。本节将针对 NSThread 相关的内容进行详细的介绍。

1.2.1 线程的创建和启动

在 iOS 开发中，通过创建一个 NSThread 类的实例作为一个线程，一个线程就是一个 NSThread 对象。要想使用 NSThread 类创建线程，有 3 种方法，具体如下所示：

```
// 1. 创建新的线程
-(instancetype)initWithTarget:(id)target selector:(SEL)selector
object:(id)argument
// 2. 创建线程后自动启动线程
+ (void)detachNewThreadSelector:(SEL)selector toTarget:(id)target
```

```

withObject:(id)argument;
// 3. 隐式创建线程
- (void)performSelectorInBackground:(SEL)aSelector withObject:(id)arg;

```

在上述代码中，这 3 种方法都是将 target 对象或者其所在对象的 selector 方法转化为线程的执行者。其中，selector 方法最多可以接收一个参数，而 object 后面对应的就是它接收的参数。

这 3 种方法中，第 1 种方法是对象方法，返回一个 NSThread 对象，并可以对该对象进行详细的设置，必须通过调用 start 方法来启动线程；第 2 种方法是类方法，创建对象成功之后就会直接启动线程，前两个方法没有本质的区别；第 3 种创建方式属于隐式创建，主要在后台创建线程。

除了以上 3 种方法，NSThread 类还提供了两个方法用于获取当前线程和主线程，具体的定义格式如下：

```

// 获取当前线程
+ (NSThread *)currentThread;
// 获得主线程
+ (NSThread *)mainThread;

```

为了大家能够更好地理解，接下来通过一个示例讲解如何运用以上 3 种方法创建并启动线程，具体步骤如下所示。

(1) 新建一个 Single View Application 应用，名称为 01–NSThreadDemo。

(2) 进入 Main.StoryBoard，从对象库添加一个 Button 和一个 Text View。其中，Button 用于响应用户单击事件，而 Text View 用于测试线程的阻塞，设计好的界面如图 1–8 所示。

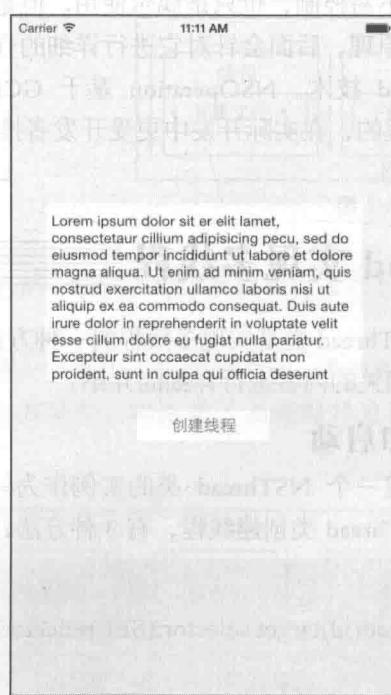


图 1–8 设计完成的界面

(3) 将 StoryBoard 上面的 Button 通过拖曳的方式，在 ViewController.m 中进行声明以响应 btnClick: 消息。通过 3 种创建线程的方法创建线程，这 3 种方法分别被封装在 threadCreate1、threadCreate2、threadCreate3 三个方法中，之后依次在 btnClick: 中被调用，代码如例 1-1 所示。

【例 1-1】ViewController.m

```
1 #import "ViewController.h"
2 @interface ViewController ()
3 // 按钮被单击
4 - (IBAction)btnClick:(id)sender;
5 @end
6 @implementation ViewController
7 - (IBAction)btnClick:(UIButton *)sender {
8     // 获取当前线程
9     NSThread *current = [NSThread currentThread];
10    NSLog(@"%@", current);
11    // 获取主线程
12    NSThread *main = [NSThread mainThread];
13    NSLog(@"%@", main);
14    [self threadCreate1];
15 }
16 - (void)run:(NSString *)param
17 {
18     // 获取当前线程
19     NSThread *current = [NSThread currentThread];
20     for (int i = 0; i<10; i++) {
21         NSLog(@"%@", current, param);
22     }
23 }
24 // 第 1 种创建方式
25 - (void)threadCreate1 {
26     NSThread *threadA = [[NSThread alloc] initWithTarget:self
27             selector:@selector(run:) object:@"哈哈"];
28     threadA.name = @"线程 A";
29     // 开启线程 A
30     [threadA start];
31     NSThread *threadB = [[NSThread alloc] initWithTarget:self
32             selector:@selector(run:) object:@"哈哈"];
33     threadB.name = @"线程 B";
34     // 开启线程 B
35     [threadB start];
36 }
37 // 第 2 种创建方式
```

```
38 - (void)threadCreate2{
39     [NSThread detachNewThreadSelector:@selector(run:)
40      toTarget:self withObject:@"我是参数"];
41 }
42 //隐式创建线程且启动，在后台线程中执行，也就是在子线程中执行
43 - (void)threadCreate3{
44     [self performSelectorInBackground:@selector(run:) withObject:@"参数 3"];
45 }
46 // 测试阻塞线程
47 - (void)test{
48     NSLog(@"%@",[NSThread currentThread]);
49     for (int i = 0; i<10000; i++) {
50         NSLog(@"-----%d", i);
51     }
52 }
53 @end
```

在例 1-1 中，第 14 行代码调用了 `threadCreate1` 方法，选择第 1 种方式创建并启动线程。第 25 ~ 36 行代码创建了 `threadA` 和 `threadB` 两条线程，并调用 `start` 方法开启了线程。线程一旦启动，就会在线程 `thread` 中执行 `self` 的 `run` 方法，并且将文字“哈哈”作为参数传递给 `run` 方法。程序的运行结果如图 1-9 所示。

图 1-9 第 1 种方式的运行结果

从图 1-9 中可以看出，主线程的 number 值为 1，且 btnClick 操作的当前线程的 number 值也为 1，说明按钮单击事件被系统自动放在主线程中。然后可以看到线程 A 和线程 B 并发执行的效果，它们的 number 值分别为 2 和 3，属于不同子线程。

(4) 修改第 14 行代码为 “[self threadCreate2];”，选择第 2 种方式。第 38~41 行代码创建了一个线程，线程一旦启动，就会在线程 thread 中执行 self 的 run:方法，并且将文字“我是参数”作为参数传递给 run:方法，程序的运行结果如图 1-10 所示。

从图 1-10 中可以看出，创建了一个 number 值为 2 的子线程，并且 run:方法获取到了“我是参数”这个参数。

```

2015-09-11 14:07:09.386 01-NSThreadDemo[1343:1087078] btnClick--<NSThread: 0x7fae7a713450>{number = 1, name = main}--current
2015-09-11 14:07:09.387 01-NSThreadDemo[1343:1087078] btnClick--<NSThread: 0x7fae7a713450>{number = 1, name = main}--main
2015-09-11 14:07:09.387 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.388 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.392 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数
2015-09-11 14:07:09.393 01-NSThreadDemo[1343:1087545] <NSThread: 0x7fae7a60e490>{number = 2, name = (null)}----run---我是参数

```

图 1-10 第 2 种方式的运行结果

(5) 修改第 14 行代码为 “[self threadCreate3];”，隐式创建一个线程。第 43~45 行代码创建了一个线程，线程一旦启动，就会在线程 thread 中执行 self 的 run:方法，并且将“参数 3”作为参数传递给 run:方法，程序的运行结果如图 1-11 所示。

```

2015-09-11 14:10:27.351 01-NSThreadDemo[1364:1111983] btnClick--<NSThread: 0x7fb1dac14220>{number = 1, name = main}--current
2015-09-11 14:10:27.352 01-NSThreadDemo[1364:1111983] btnClick--<NSThread: 0x7fb1dac14220>{number = 1, name = main}--main
2015-09-11 14:10:27.352 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.353 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.354 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.354 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3
2015-09-11 14:10:27.365 01-NSThreadDemo[1364:1112293] <NSThread: 0x7fb1dae1e6b0>{number = 2, name = (null)}----run---参数3

```

图 1-11 第 3 种方式的运行结果

从图 1-11 中可以看出，创建了一个 number 值为 2 的子线程，并且 run:方法获取到了“参数 3”这个参数。

(6) 修改第 14 行代码为 “[self test];”，用于测试线程的阻塞情况。重新运行程序，单击按钮后，发现 Debug 输出栏一直在打印输出，说明线程仍被占用。这时，拖曳屏幕中的文本视图，发现该文本视图没有任何响应。待输出栏停止输出的时候，将输出栏的滚动条滑至顶部，程序的运行结果如图 1-12 所示。

```

2015-09-11 14:12:37.442 01-NSThreadDemo[1385:1127674] btnClick--<NSThread: 0x7fc70be133d0>{number = 1, name = main}--current
2015-09-11 14:12:37.442 01-NSThreadDemo[1385:1127674] btnClick--<NSThread: 0x7fc70be133d0>{number = 1, name = main}--main
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] <NSThread: 0x7fc70be133d0>{number = 1, name = main}
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----0
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----1
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----2
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----3
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----4
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----5
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----6
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----7
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----8
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----9
2015-09-11 14:12:37.443 01-NSThreadDemo[1385:1127674] -----10

```

图 1-12 test 阻塞运行结果

从图 1-12 可以看出，test 方法执行时所处的线程为主线程，如果把大量耗时的操作放在主线程当中，就会阻塞主线程，影响主线程的其他操作正常响应。

1.2.2 线程的状态

当线程被创建并启动之后，它既不是一启动就进入了执行状态，也不是一直处于执行状态，即便线程开始运行以后，它也不可能一直占用着 CPU 独自运行。由于 CPU 需要在多

个线程之间进行切换，造成了线程的状态也会在多次运行、就绪之间进行切换，如图 1-13 所示。

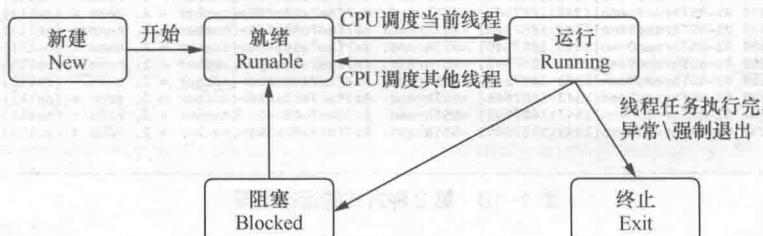


图 1-13 线程状态的切换

由图 1-13 可知，线程主要有 5 个状态，并按照相应的逻辑顺利地在这几个状态中切换。这些状态的具体介绍如下。

1. 新建 (New)

当程序新建了一个线程之后，该线程就处于新建状态。这时，它和其他对象一样，仅仅是由系统分配了内存，并初始化了其内部成员变量的值，此时的线程没有任何动态特征。

2. 就绪 (Runnable)

当线程对象调用了 `start` 方法后，该线程就处于就绪状态，系统会为其创建方法调用的栈和程序计数器，处于这种状态中的线程并没有开始运行，只是代表该线程可以运行了，但到底何时开始运行，由系统来进行控制。

3. 运行 (Running)

当 CPU 调度当前线程的时候，将其他线程挂起，当前线程变成运行状态；当 CPU 调度其他线程的时候，当前线程处于就绪状态。要测试某个线程是否正在运行，可以调用 `isExecuting` 方法，若返回 YES，则表示该线程处于运行状态。

4. 终止 (Exit)

当线程遇到以下 3 种情况时，线程会由运行状态切换到终止状态，具体如下。

(1) 线程执行方法执行完成，线程正常结束。

(2) 线程执行的过程中出现了异常，线程崩溃结束。

(3) 直接调用 `NSThread` 类的 `exit` 方法来终止当前正在执行的线程。

若要测试某个线程是否结束，可以调用 `isFinished` 方法判断，若返回 YES，则表示该线程已经终止。

5. 阻塞 (Blocked)

如果当前正在执行的线程需要暂停一段时间，并进入阻塞状态，可以通过 `NSThread` 类提供的两个类方法来完成，具体定义格式如下：

```

// 让当前正在执行的线程暂停到 date 参数代表的时间，并且进入阻塞状态
+(void)sleepUntilDate:(NSDate *)date;
// 让正在执行的线程暂停 ti 秒，并且进入阻塞状态。
+(void)sleepForTimeInterval:(NSTimeInterval)ti;
  
```

需要注意的是，当线程进入阻塞状态之后，在其睡眠的时间内，该线程不会获得执行的机会，即便系统中没有其他可执行的线程，处于阻塞状态的线程也不会执行。

1.2.3 线程间的安全隐患

进程中的一块资源可能会被多个线程共享，也就是多个线程可能会访问同一块资源，这里的资源包括对象、变量、文件等。当多个线程同时访问同一块资源时，会造成资源抢夺，很容易引发数据错乱和数据安全问题。

这里有一个很经典的卖火车票的例子，假设有 1000 张火车票，同时开启两个窗口执行卖票的动作，每出售一张票后就返回当前的剩余票数，由于两个线程共同抢夺 1000 张的票数资源，容易造成剩余票数混乱，具体如图 1-14 所示。

在图 1-14 所示案例中，两个线程同时读取当前票数是 1000，然后线程 1 的卖票窗口 1 售出 1 张票，使票数减 1 变成 999，同时线程 2 也售出 1 张票，使票数减 1 变成 999。结果是售出了 2 张票，但是剩余票数是 999，这就造成了数据的错误。为了解决这个问题，实现数据的安全访问，可以

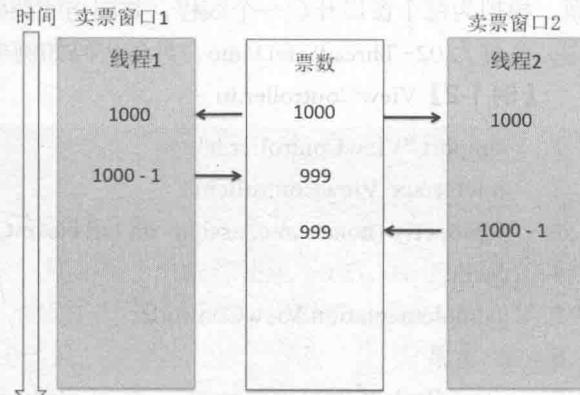


图 1-14 卖火车票案例

使用线程间加锁。针对加锁前后线程 A 和线程 B 的变化，分别可用图 1-15 和图 1-16 表示。

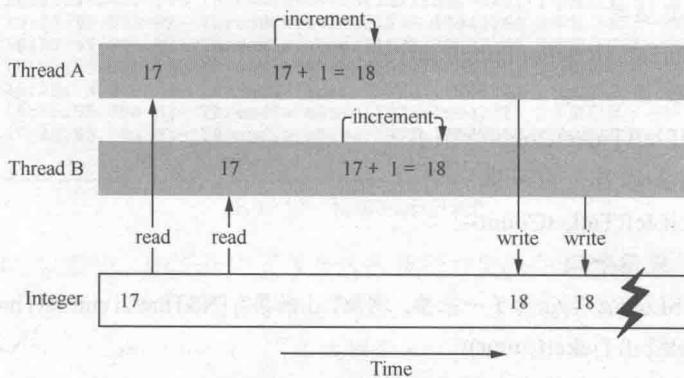


图 1-15 加锁前的示意图

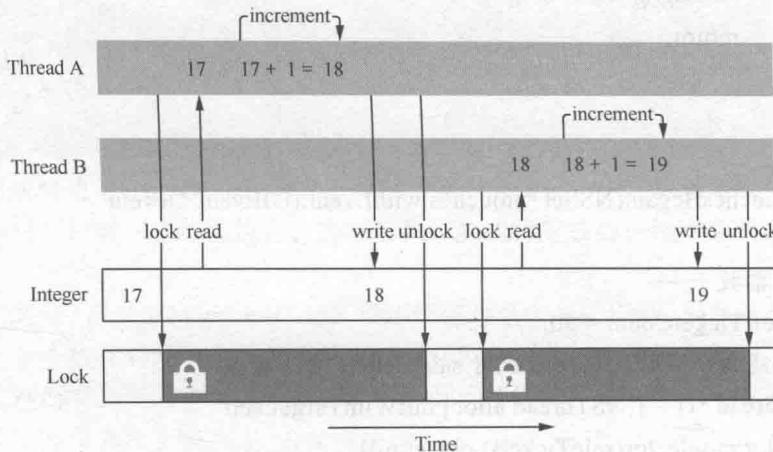


图 1-16 加锁后的示意图