TURING 图灵程序设计丛书

Apress®

苹果源代码不会告诉你的

Objective-C 高级编程

iOS与OS X多线程和内存管理

【日】Kazuki Sakamoto　Tomohiko Furumoto 著

黎华 译

人民邮电出版社

POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Objective-C 高级编程

iOS与OS X多线程和内存管理

【日】Kazuki Sakamoto　Tomohiko Furumoto 著
黎华 译

人民邮电出版社
北京

图书在版编目 (CIP) 数据

Objective-C 高级编程 : iOS 与 OS X 多线程和内存管理 / (日) 坂本一树 , (日) 古本智彦著 ; 黎华译 . -- 北京 : 人民邮电出版社 , 2013.6
（图灵程序设计丛书）
书名原文 : Pro multithreading and memory management for iOS and OS X
ISBN 978-7-115-31809-1

Ⅰ. ① O… Ⅱ. ①坂… ②古… ③黎… Ⅲ. ① C 语言—程序设计 Ⅳ. ① TP312

中国版本图书馆 CIP 数据核字（2013）第 095069 号

内 容 提 要

本书在苹果公司公开的源代码基础上，深入剖析了对应用于内存管理的 ARC 以及应用于多线程开发的 Blocks 和 GCD。这些新技术看似简单，实则非常容易成为技术开发的陷阱，开发者仅靠阅读苹果公司的文档是不够的。

本书适合有一定基础的 iOS 开发者阅读。

◆ 著 ［日］Kazuki Sakamoto, Tomohiko Furumoto
译 黎 华
责任编辑 乐 馨
执行编辑 金松月
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市潮河印业有限公司印刷

◆ 开本：800×1000 1/16
印张：12
字数：284 千字 2013 年 6 月第 1 版
印数：1-3 000 册 2013 年 6 月河北第 1 次印刷
著作权合同登记号 图字：01-2012-4467 号

定价：49.00 元

读者服务热线：(010)51095186 转 604 印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

序

本书旨在解说 iOS 与 OS X 中的 ARC、Blocks 和 Grand Central Dispatch（GCD）。

与一般讲解书略有不同，这是一本“非常深奥的书”，内容涉及

- iOS 5、OS X Lion 引入的新的内存管理技术 ARC；
- iOS 4、OS X Snow Leopard 引入的多线程应用技术 Blocks 和 GCD。

这些新技术在面向 iOS 5、OS X Lion 应用开发时不可或缺。它们看似简单，但若无深入了解，就会变成技术开发的陷阱。本书在苹果公司公开的源代码基础上加以解说，深入剖析，这些内容是仅靠阅读苹果公司的参考文档而难以企及的。

读者对象

- 熟悉 C/C++ 但不熟悉 Objective-C 的读者。
- 想知道 Objective-C 源代码究竟是如何运行的读者。
- iOS 或者 Mac 应用开发者（初学者以上水平、想进一步深入学习）。

致谢

在此向爽快接受紧急销售任务的达人出版会高桥先生、Impress Japan 公司的畑中先生、赋予我编写机会的畑先生、百忙中进行排版制作的井田先生，以及提出中肯建议的 Samuli 先生和野崎先生一并致谢。

感谢主动承担翻译工作并提供全方位帮助的古本先生。

“我一直想拥有和控制我们所做的主要技术。”

——史蒂夫·乔布斯

没有您的热情和技术，不可能有这本书。感谢您，愿您安息。

特别提示

为了确保译文的准确性，本书直接翻译自日文版『エキスパート Objective-C プログラミング: iOS/OS X のメモリ管理とマルチスレッド』（インプレスジャパン），并采用的日文版的编排方式。特此说明 。

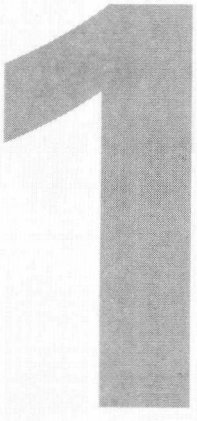
图灵公司感谢何文祥、朱星垠的审读。

目录 Contents

第1章

自动引用计数

本章主要介绍从 OS X Lion 和 iOS 5 引入的内存管理新功能——自动引用计数。让我们在复习 Objective-C 的内存管理的同时，来详细了解这项新功能会带来怎样的变化。



1.1　什么是自动引用计数

顾名思义，自动引用计数（ARC，Automatic Reference Counting）是指内存管理中对引用采取自动计数的技术。以下摘自苹果的官方说明。

> 在 Objective-C 中采用 Automatic Reference Counting（ARC）机制，让编译器来进行内存管理。在新一代 Apple LLVM 编译器中设置 ARC 为有效状态，就无需再次键入 retain 或者 release 代码，这在降低程序崩溃、内存泄漏等风险的同时，很大程度上减少了开发程序的工作量。编译器完全清楚目标对象，并能立刻释放那些不再被使用的对象。如此一来，应用程序将具有可预测性，且能流畅运行，速度也将大幅提升。[①]

这些优点无疑极具吸引力，但关于 ARC 技术，最重要的还是下面这一点：

“在 LLVM 编译器中设置 ARC 为有效状态，就无需再次键入 retain 或者是 release 代码。”

换言之，若满足以下条件，就无需手工输入 retain 和 release 代码了。

- 使用 Xcode 4.2 或以上版本。
- 使用 LLVM 编译器 3.0 或以上版本。
- 编译器选项中设置 ARC 为有效。

在以上条件下编译源代码时，编译器将自动进行内存管理，这正是每个程序员都梦寐以求的。在正式讲解精彩的 ARC 技术之前，我们先来了解一下，在此之前，程序员在代码中是如何手工进行内存管理的。

1.2　内存管理 / 引用计数

1.2.1　概要

Objective-C 中的内存管理，也就是引用计数。可以用开关房间的灯为例来说明引用计数的机制，如图 1-1 所示。

① 引自 http://developer.apple.com/jp/technologies/ios5。

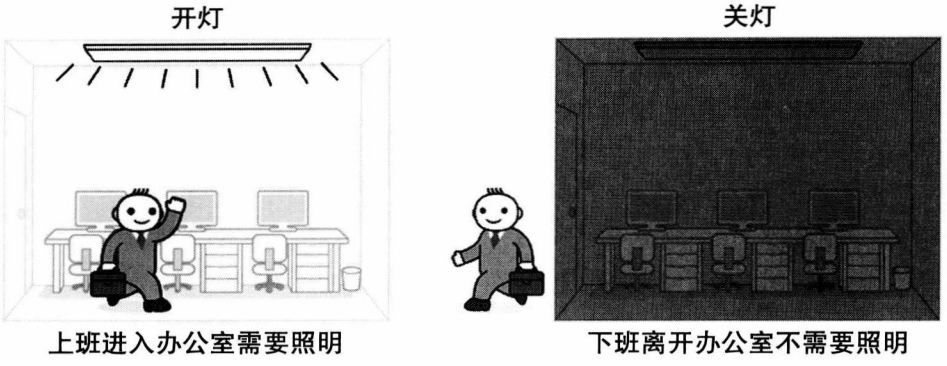


图 1-1 办公室照明

假设办公室里的照明设备只有一个。上班进入办公室的人需要照明，所以要把灯打开。而对于下班离开办公室的人来说，已经不需要照明了，所以要把灯关掉。若是很多人上下班，每个人都开灯或是关灯，那么办公室的情况又将如何呢？最早下班离开的人如果关了灯，那就会像图 1-2 那样，办公室里还没走的所有人都将处于一片黑暗之中。

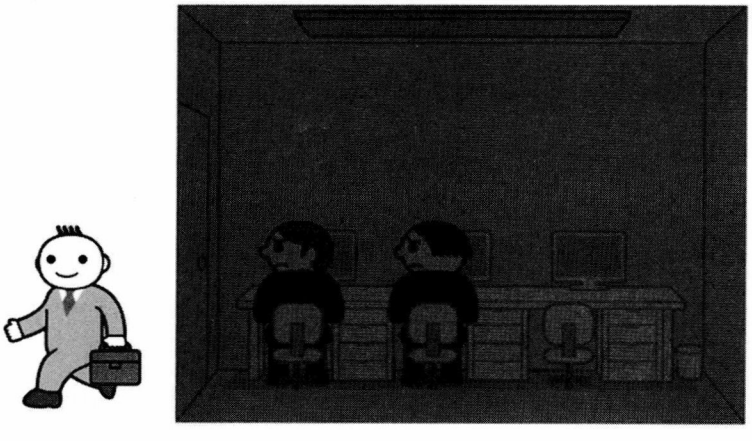


图 1-2 办公室里的照明问题

解决这一问题的办法是使办公室在还有至少 1 人的情况下保持开灯状态，而在无人时保持关灯状态。

（1）最早进入办公室的人开灯。
（2）之后进入办公室的人，需要照明。
（3）下班离开办公室的人，不需要照明。
（4）最后离开办公室的人关灯（此时已无人需要照明）。

为判断是否还有人在办公室里，这里导入计数功能来计算“需要照明的人数”。下面让我们

来看看这一功能是如何运作的吧。

（1）第一个人进入办公室，“需要照明的人数”加 1。计数值从 0 变成了 1，因此要开灯。

（2）之后每当有人进入办公室，“需要照明的人数”就加 1。如计数值从 1 变成 2。

（3）每当有人下班离开办公室，“需要照明的人数”就减 1。如计数值从 2 变成 1。

（4）最后一个人下班离开办公室时，“需要照明的人数”减 1。计数值从 1 变成了 0，因此要关灯。

这样就能在不需要照明的时候保持关灯状态。办公室中仅有的照明设备得到了很好的管理，如图 1-3 所示。

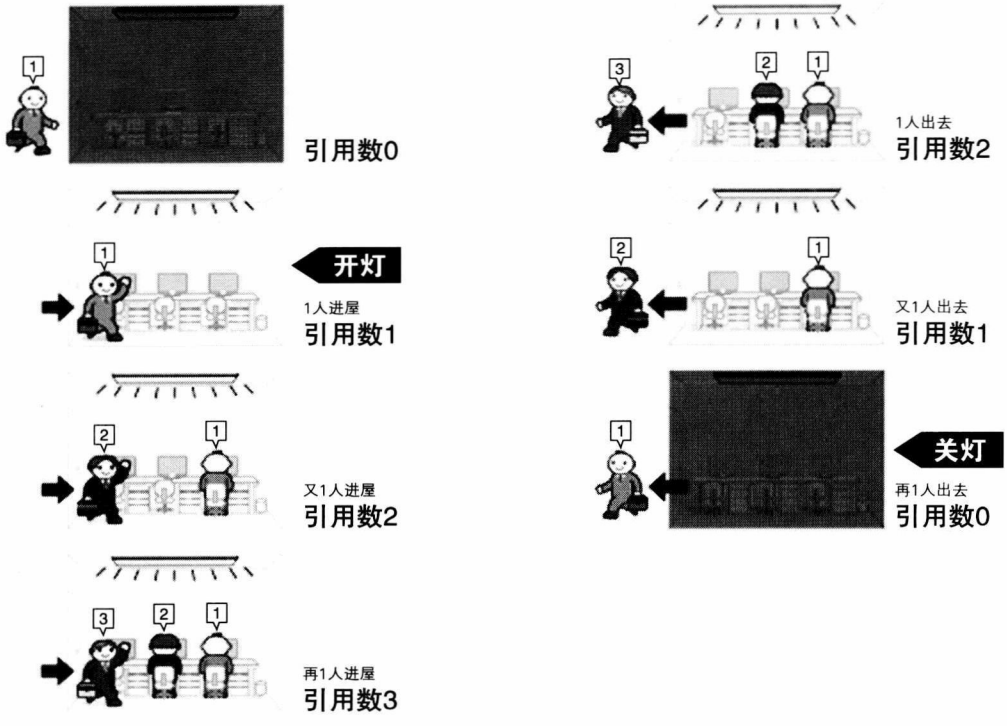


图 1-3　办公室里的照明管理

在 Objective-C 中，“对象”相当于办公室的照明设备。在现实世界中办公室的照明设备只有一个，但在 Objective-C 的世界里，虽然计算机资源有限，但一台计算机可以同时处理好几个对象。

此外，“对象的使用环境”相当于上班进入办公室的人。虽然这里的“环境”有时也指在运行中的程序代码、变量、变量作用域、对象等，但在概念上就是使用对象的环境。上班进入办公室的人对办公室照明设备发出的动作，与 Objective-C 中的对应关系则如表 1-1 所示。

表 1-1 对办公室照明设备所做的动作和对 Objective-C 的对象所做的动作

对照明设备所做的动作	对 Objective-C 对象所做的动作
开灯	生成对象
需要照明	持有对象
不需要照明	释放对象
关灯	废弃对象

使用计数功能计算需要照明的人数，使办公室的照明得到了很好的管理。同样，使用引用计数功能，对象也就能够得到很好的管理，这就是 Objective-C 的内存管理。如图 1-4 所示。

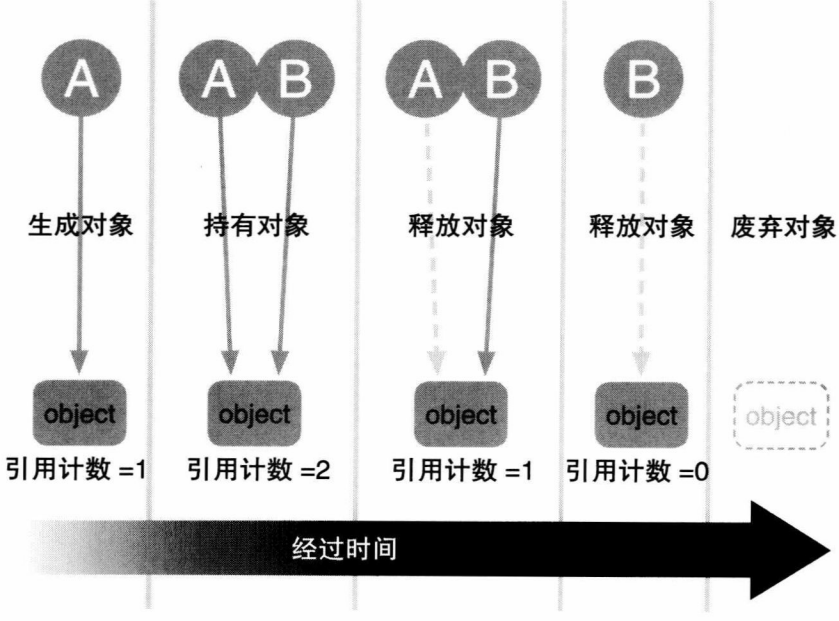


图 1-4 引用计数的内存管理

现在对 Objective-C 的内存管理多少理解一些了吧。下面，我们将学习“引用计数式内存管理”的思考方式，并在此基于实现进一步加深理解。

1.2.2 内存管理的思考方式

首先来学习引用计数式内存管理的思考方式。看到“引用计数”这个名称，我们便会不自觉地联想到“某处有某物多少多少”而将注意力放到计数上。但其实，更加客观、正确的思考方式是：

- 自己生成的对象，自己所持有。
- 非自己生成的对象，自己也能持有。

- 不再需要自己持有的对象时释放。
- 非自己持有的对象无法释放。

引用计数式内存管理的思考方式仅此而已。按照这个思路，完全不必考虑引用计数。

上文出现了“生成”、“持有”、“释放”三个词。而在 Objective-C 内存管理中还要加上“废弃”一词，这四个词将在本书中频繁出现。各个词表示的 Objective-C 方法如表 1-2。

表 1-2　对象操作与 Objective-C 方法的对应

对象操作	Objective-C 方法
生成并持有对象	alloc/new/copy/mutableCopy 等方法
持有对象	retain 方法
释放对象	release 方法
废弃对象	dealloc 方法

这些有关 Objective-C 内存管理的方法，实际上不包括在该语言中，而是包含在 Cocoa 框架中用于 OS X、iOS 应用开发。Cocoa 框架中 Foundation 框架类库的 NSObject 类担负内存管理的职责。Objective-C 内存管理中的 alloc/retain/release/dealloc 方法分别指代 NSObject 类的 alloc 类方法、retain 实例方法、release 实例方法和 dealloc 实例方法。

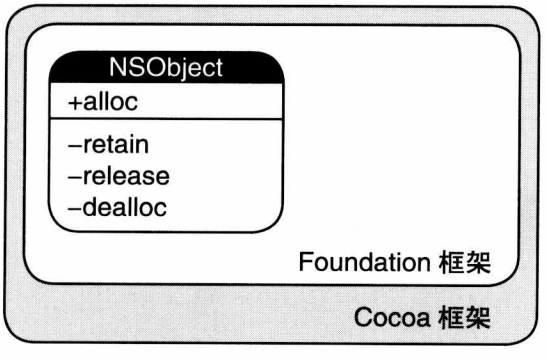


图 1-5　Cocoa 框架、Foundation 框架和 NSObject 类的关系

接着来详细了解“内存管理的思考方式”中出现的各个项目。

自己生成的对象，自己所持有

使用以下名称开头的方法名意味着自己生成的对象只有自己持有：

- alloc
- new
- copy
- mutableCopy

上文出现了很多“自己”一词。本书所说的“自己”固然对应前文提到的“对象的使用环境”，但将之理解为编程人员“自身”也是没错的。下面写出了自己生成并持有对象的源代码。为生成并持有对象，我们使用 alloc 方法。

```
/*
 * 自己生成并持有对象
 */

id obj = [[NSObject alloc] init];

/*
 * 自己持有对象
 */
```

使用 NSObject 类的 alloc 类方法就能自己生成并持有对象。指向生成并持有对象的指针被赋给变量 obj。另外，使用如下 new 类方法也能生成并持有对象。[NSObject new] 与 [[NSObject alloc] init] 是完全一致的。

```
/*
 * 自己生成并持有对象
 */

id obj = [NSObject new];

/*
 * 自己持有对象
 */
```

copy 方法利用基于 NSCopying 方法约定，由各类实现的 copyWithZone：方法生成并持有对象的副本。与 copy 方法类似，mutableCopy 方法利用基于 NSMutableCopying 方法约定，由各类实现的 mutableCopyWithZone：方法生成并持有对象的副本。两者的区别在于，copy 方法生成不可变更的对象，而 mutableCopy 方法生成可变更的对象。这类似于 NSArray 类对象与 NSMutableArray 类对象的差异。用这些方法生成的对象，虽然是对象的副本，但同 alloc、new 方法一样，在“自己生成并持有对象”这点上没有改变。

另外，根据上述“使用以下名称开头的方法名”，下列名称也意味着自己生成并持有对象。

- allocMyObject
- newThatObject
- copyThis
- mutableCopyYourObject

但是对于以下名称，即使用 alloc/new/copy/mutableCopy 名称开头，并不属于同一类别的方法。

- allocate
- newer

- copying
- mutableCopyed

这里用驼峰拼写法（CamelCase[①]）来命名。

非自己生成的对象，自己也能持有

用上述项目之外的方法取得的对象，即用 alloc/new/copy/mutableCopy 以外的方法取得的对象，因为非自己生成并持有，所以自己不是该对象的持有者。我们来使用 alloc/new/copy/mutableCopy 以外的方法看看。这里试用一下 NSMutableArray 类的 array 类方法。

```
/*
 * 取得非自己生成并持有的对象
 */

id obj = [NSMutableArray array];

/*
 * 取得的对象存在，但自己不持有对象
 */
```

源代码中，NSMutableArray 类对象被赋给变量 obj，但变量 obj 自己并不持有该对象。使用 retain 方法可以持有对象。

```
/*
 * 取得非自己生成并持有的对象
 */

id obj = [NSMutableArray array];

/*
 * 取得的对象存在，但自己不持有对象
 */

[obj retain];

/*
 * 自己持有对象
 */
```

通过 retain 方法，非自己生成的对象跟用 alloc/new/copy/mutableCopy 方法生成并持有的对象一样，成为了自己所持有的。

不再需要自己持有的对象时释放

自己持有的对象，一旦不再需要，持有者有义务释放该对象。释放使用 release 方法。

① 驼峰拼写法是将第一个词后每个词的首字母大写来拼写复合词的记法。例如 CamelCase 等。

```
/*
 * 自己生成并持有对象
 */

id obj = [[NSObject alloc] init];

/*
 * 自己持有对象
 */

[obj release];

/*
 * 释放对象
 *
 * 指向对象的指针仍然被保留在变量 obj 中，貌似能够访问，
 * 但对象一经释放绝对不可访问。
 */
```

如此，用 alloc 方法由自己生成并持有的对象就通过 release 方法释放了。自己生成而非自己所持有的对象，若用 retain 方法变为自己持有，也同样可以用 release 方法释放。

```
/*
 * 取得非自己生成并持有的对象
 */

id obj = [NSMutableArray array];

/*
 * 取得的对象存在，但自己不持有对象
 */

[obj retain];

/*
 * 自己持有对象
 */

[obj release];

/*
 * 释放对象
 * 对象不可再被访问
 */
```

用 alloc/new/copy/mutableCopy 方法生成并持有的对象，或者用 retain 方法持有的对象，一旦不再需要，务必要用 release 方法进行释放。

如果要用某个方法生成对象，并将其返还给该方法的调用方，那么它的源代码又是怎样的呢？

```
- (id) allocObject
{
```

```
    /*
     * 自己生成并持有对象
     */

    id obj = [[NSObject alloc] init];

    /*
     * 自己持有对象
     */

    return obj;
}
```

如上例所示，原封不动地返回用 alloc 方法生成并持有的对象，就能让调用方也持有该对象。请注意 allocObject 这个名称是符合前文命名规则的。

```
/*
 * 取得非自己生成并持有的对象
 */

id obj1 = [obj0 allocObject];

/*
 * 自己持有对象
 */
```

allocObject 名称符合前文的命名规则，因此它与用 alloc 方法生成并持有对象的情况完全相同，所以使用 allocObject 方法也就意味着“自己生成并持有对象”。

那么，调用 [NSMutableArray array] 方法使取得的对象存在，但自己不持有对象，又是如何实现的呢？根据上文命名规则，不能使用以 alloc/new/copy/mutableCopy 开头的方法名，因此要使用 object 这个方法名。

```
- (id)object
{
    id obj = [[NSObject alloc] init];

    /*
     * 自己持有对象
     */

    [obj autorelease];

    /*
     * 取得的对象存在，但自己不持有对象
     */

    return obj;
}
```

上例中，我们使用了 autorelease 方法。用该方法，可以使取得的对象存在，但自己不持有对