浙江大學

硕 士 研 究 生 读 书 报 告

题目 iOS 内存管理研究

作者姓名 刘凯敏

作者学号 21651074

指导教师 李启雷

学科专业 软件工程

所在学院 软件学院

提交日期 二〇一七年一月

Research Of IOS Memory Management Mechanism

A Dissertation Submitted to
Zhejiang University
in partial fulfillment of the requirements for
the degree of
Master of Engineering

Major Subject: Software Engineering
Advisor: Li Qilei

By:Liu Kaimin

Zhejiang University, P.R. China
2017

摘要

随着移动设备的内存越来越大，程序员也已经度过了为了那一两 M 的内存在系统的抽丝剥茧的年代，对于 JAVA 的开发者，对内存更是伸手即取，并且从不关心什么时候还回去。但是，程序的掌控度对程序员来说是至关重要的，任何语言的内存管理机制的初衷也是在有限的空间里完成最精致的逻辑。

我们知道在程序运行过程中要创建大量的对象，和其他高级语言类似，在 Objective-C 中对象时存储在堆中的，系统并不会自动释放堆中的内存（注意基本类型是由系统自己管理的，放在栈上）。如果一个对象创建并使用后没有得到及时释放那么就会占用大量内存。其他高级语言如 C#、Java 都是通过垃圾回收来（GC）解决这个问题的，但在 Objective-C 中并没有类似的垃圾回收机制，因此它的内存管理就需要由开发人员手动维护。这篇报告将着重介绍 Objective-C 内存管理。

关键词：内存管理；Objective-C；垃圾回收

Abstract

As the mobile memory is more and more big, the programmer has spent for a year or two M the presence of the whole system in s, for JAVA developers, to memory is reached that, and never care about what time it back. But the control program are vital for programmers, the purpose of the memory management mechanism of any language is in limited space to complete the most delicate logic.

We know that in the process of the program is run to create a large number of objects, and other high-level languages are similar, in the Objective - C when the object is stored in the heap, in the system does not automatically release in the heap memory (pay attention to the basic types is run by the system, on the stack). If an object is created and used has not been timely release after then will take up a lot of memory. Other high-level language such as C #, Java is through garbage collection (GC) to solve the problem, but in Objective - C and there is no similar garbage collection mechanism, so its memory management will need to manually maintained by the developers. This report will focus on Objective - C memory management.

Keywords: Memory management; Objective-C; The garbage collection

1.引言

对于面向对象的编程语言而言，程序需要不断地创建对象。开始创建时，程序创建的所有对象都有指针指向它，程序可能需要访问这些对象的实例对象或者调用这些对象的方法，总之，这些对象是有用的。随着时间的流逝，程序再次创建了一些新的对象，而那些老的对象已经不会再被调用，也不再有指针指向它们，但如果程序没有回收它们占用的内存，就会出现内存泄漏。如果程序一直泄漏内存，那么可用的内存就会越来越少，知道没有足够的内存去做更多的事情，应用程序可能会崩溃。

对于有效的内存管理，通常认为包括两方面内容。

- 内存分配：当程序创建对象时需要为对象分配内存。采用合理的设计，尽量减少对象的创建，并减少创建过程中的内存开销。
- 内存回收：当程序不在需要对象时，程序必须及时回收这些对象所占用的内存，以便程序可以再次使用这些内存。

一般来说，内存的分配工作对程序影响小一些，即使程序在一段时间内创建了过多的对象，造成较大的内存开销，但是只要这些对象占用的内存及时得到了回收，程序也依然可以正常运行。

而且，内存的分配操作相对比较容易，当程序创建对象时，系统会自动为这些对象分配内存。相比之下，内存的回收操作就显得更加复杂，管理起来也更加困难，因此，人们开发了多种回收策略，典型的内存回收策略有两种：①自动回收，在这种策略下，系统会自动跟踪所有的对象，并对这些对象失去作用时回收它们占用的内存；②混合回收，在这种策略下，系统做一些工作，但同时也需要程序员在对象失去作用时通知系统。

在 Xcode4.2 之前，Objective-C 的内存回收需要程序员花费大量的精力去理解 Objective-C 内存回收的相关理论，并且必须在程序中通过 retain、release、

autorelease 方法去管理对象的引用计数，这样才能让程序正常回收内存。稍有不慎，程序就可能会造成内存泄漏，或者可能让对象过早地被释放，从而引起程序崩溃。

Xcode4.2 之后引入了一个重要的新特性：自动引用计数（Automatic Reference Counting，ARC)。通过启用 ARC 特性，程序员不再需要重点关注内存回收相关的内容，它大大降低了程序员进入 iOS 应用开发的门槛。

总结起来，Objective-C 程序员可用的内存回收机制有如下 3 种：

- 手动引用计数（MRC）和自动释放池。
- 自动引用计数（ARC）
- 自动垃圾回收

很遗憾的是,iOS 系统不支持自动垃圾回收。iOS 应用目前只支持手动引用计数和 ARC 两种机制。相比之下，ARC 已经足够简单、易用。

2.Objective-C 内存管理的对象

在 iOS 开发中，内存中的对象主要有两类，一类是值类型，比如 int、float、struct 等基本数据类型，另一类是引用类型，也就是继承自 NSObject 类的所有的 OC 对象。前一种值类型不需要我们管理，后一种引用类型是需要我们管理内存的，一旦管理不好，就会产生非常糟糕的后果。

为什么值类型不需要管理，而引用类型需要管理呢？那是因为他们分配内存方式不一样。值类型会被放入栈中，他们依次紧密排列，在内存中占有一块连续的内存空间，遵循先进后出的原则。引用类型会被放到堆中，当给对象分配内存空间时，会随机的从内存当中开辟空间，对象与对象之间可能会留有不确定大小的空白空间，因此会产生很多内存碎片，需要我们管理。

栈内存与堆内存从性能上比较，栈内存要优于堆内存，这是因为栈遵循先进后出的

原则，因此当数据量过大时，存入栈会明显的降低性能。因此，我们会把大量的数据存入堆中，然后栈中存放堆的地址，当需要调用数据时，就可以快速的通过栈内的地址找到堆中的数据。

值类型和引用类型之间是可以相互转化的，把值类型转化为引用类型的过程叫做装箱，比如把 int 包装为 NSNumber，这个过程会增加程序的运行时间，降低性能。而把引用类型转为值类型的过程叫做拆箱，比如把 NSNumber 转为 float，在拆箱的过程中，我们一定要注意数据原有的类型，如果类型错误，可能导致拆箱失败，因此会存在安全性的问题。手动的拆箱和装箱，都会增加程序的运行时间，降低代码可读性，影响性能。在 IOS 开发过程中，栈内存中的值类型系统会自动管理，堆内存中的引用类型是需要我们管理的。

3. 与内存有关的修饰符

strong：强引用，ARC 中使用，与 MRC 中 retain 类似，使用之后，计数器+1。

weak：弱引用 ，ARC 中使用，如果指向的对象被释放了，其指向 nil，可以有效的避免野指针，其引用计数为 1。

readwrite：可读可写特性，需要生成 getter 方法和 setter 方法时使用。

readonly：只读特性，只会生成 getter 方法 不会生成 setter 方法，不希望属性在类外改变。

assign：赋值特性，不涉及引用计数，弱引用，setter 方法将传入参数赋值给实例变量，仅设置变量时使用。

retain：表示持有特性，setter 方法将传入参数先保留，再赋值，传入参数的 retainCount 会+1。

copy:表示拷贝特性,setter 方法将传入对象复制一份,需要完全一份新的变量时。

nonatomic ：非原子操作，不加同步，多线程访问可提高性能，但是线程不安全的。决定编译器生成的 setter getter 是否是原子操作。

atomic ： 原子操作，同步的，表示多线程安全，与 nonatomic 相反。

4.Objective-C 管理内存的方式

Objective-C 提供了两种内存管理机制手动引用计数 MRC（Mannul Reference Counting）和自动引用计数 ARC（Automatic Reference Counting），分别提供对内存的手动和自动管理，来满足不同的需求。

4.1 手动引用计数

4.1.1 对象的引用计数

程序创建一个对象时，可能在一段时间内访问该对象的实例变量，也可能在这段时间内调用该对象的方法。当程序不再需要该对象时，就希望系统及时回收该对象所占用的内存。系统如何才能知道何时需要回收该对象呢？

Objective-C 采用了一种被称为引用计数（Reference Counting）的机制来跟踪对象的状态：每个对象都有一个与之关联的整数，这个整数被称为引用计数。在正常情况下，当一段代码需要使用某个对象时，应将该对象的引用计数加 1；当这段代码不再需要该对象时，应将该对象的引用计数减 1，表示这段代码不再访问该对象。当对象的引用计数为 0 时，表明没有任何程序需要该对象，系统就会回收该对象所占用的内存。

系统在销毁该对象之前，会自动调用该对象的 dealloc 方法(该方法继承自 NSObject，因此所有的对象都有这个方法）来执行一些回收操作。如果该对象还持有其他对象的引用，此时必须重写 dealloc 方法，在该方法中释放该对象所持有的其他对象（通常就是调用被持有对象的 release 方法将引用计数减 1)。

需要注意的是，对象的 dealloc 方法是不能手动调用的，当一个对象的引用计数为

0 时，系统会自动调用该对象的 dealloc 方法来销毁它。

在引用计数中，改变对象的引用计数的方式如下。

- 当程序调用方法名以 alloc、new、copy、mutableCopy 开头的方法来创建对象时，该对象的引用计数加 1。
- 程序调用对象的 retain 方法时，该对象的引用计数加 1。
- 程序调用对象的 release 方法时，该对象的引用计数减 1。

NSObject 中提供了有关引用计数的如下方法。

- -retain：将该对象的引用计数加 1。
- -release：将该对象的引用计数减 1。
- -autorelease：不改变该对象的引用计数的值，只是将对象添加到自动释放池中。
- -retainCount：返回该对象的引用计数的值。

4.1.2 对象所属权

下面假设有如下场景。

- 在 main()函数中创建一个 Item 对象
- 程序中 User 对象持有一个 Item 对象，程序可调用 User 的 setItem 对象来设置该 User 持有的 Item 对象。

此时的问题是：该 Item 对象属于 User 还是属于 main()函数？，应该有 main()函数来释放 Item 对象还是由 User 来释放。

如果让 main()函数来销毁 Item 对象，那么接下来 User 对象所持有的 Item 对象其实被销毁了，因此，User 对象中指向 Item 对象的指针变成了悬空指针。如果在 User 对象的 dealloc 方法中就销毁了 Item 对象，那么可能出现的情况是：当 User 的引用计数

变为 0 时，User 的 dealloc 方法被调用，这样它所持有的 Item 也被销毁了。界限了 main()函数指向该对象的指针就变成了悬空指针，这也是危险的。

由此可见，上面两种思路都不可行，常用的做法是，在 setItem:方法中传入 Item 对象的引用计数加 1，这样就可以保证：只有当 main()函数调用了 Item 的 release 方法将其引用计数减 1，且 User 的 alloc 方法再次调用 Item 的 release 方法将其引用计数减 1（二者不分先后）时，Item 对象的引用计数为 0，系统才会真正释放该 Item 对象。

4.1.3 自动释放池

初学者可能会认为：如果在函数、方法开始处将对象的引用计数加 1，接下来在函数、方法不在需要该对象时就应该将该对象的引用计数减 1。这个理论基本正确，可问题在于，有些函数、方法需要返回一个对象，那么问题就产生了：该函数、方法需要返回的对象肯定不能立即将引用计数减 1，这样系统可能在该对象被返回之前，就已经销毁了该对象。

为了保证函数、方法返回的对象不会在被返回之前就被销毁，就需要保证被函数、方法返回的对象能被延迟销毁。为了实现这种延时销毁，有如下两种方法。

- 程序每次获取并使用完其他函数、方法返回的对象之后，立即调用该对象的 release 方法将函数、方法返回对象的引用计数减 1。
- 使用自动释放池进行延迟销毁。

第一种方式比较容易理解，但编程显得比较复杂。例如，如下程序。

```objc
#import <Foundation/Foundation.h>
#import "FKItem.h"

FKItem* productItem()
{
    FKItem* item = [FKItem new]; // 引用计数为1
    NSLog(@"函数返回之前的引用计数：%ld" , item.retainCount);
    // 返回的对象的引用计数为1
    return item;
}

int main(int argc , char * argv[])
{
    // it的引用计数为1
    FKItem* it = productItem();
    // 接下来可以调用it的方法
    NSLog(@"%ld" , it.retainCount);
    // ...
    // 程序执行完成后,将it（引用productItem()函数返回值）的引用计数减1
    // 程序使用productItem()函数返回值完成后，延迟销毁it所指向的对象
    [it release];
}
```

从上面的程序可以看出，对于返回 FKItem 对象的 productItem()函数而言，它只是负责创建 FKItem 对象，并将它的引用计数加 1，该函数并没有将它的引用计数减 1。接下来在 main()函数中调用 productItem()函数时，将会使用一个 FKItem 变量来保存 productItem()函数的返回值，当该函数的返回值使用完成后，程序调用[it release]代码将 FKItem 对象的引用计数减 1，使它的引用计数变为 0，从而使系统销毁该对象。

上面的做法虽然简洁、易懂，但违背了前面介绍的原则：谁负责将对象的引用计数加 1，谁就应该负责将该对象的引用计数减 1——因此，这种做法并不是一种优雅的设计。

更好地做法是使用自动释放池来解决这个问题。所谓自动释放池，就是一个存放对象的容器（比如集合）。自动释放池会保证延迟释放该池中所有的对象。出于自动释放的考虑，所有的对象都应该添加到自动释放池中，这样可以让自动释放池在销毁之前，先销毁池中的所有对象。为了把一个对象添加到自动释放池中，可调用该对象的autorelease 方法，其签名如下。

- - (id) autorelease：该方法不会修改对象的引用计数，只是将该对象添加到自动释放池中。该对象将会返回调用该方法的对象本身。

由于 autorelease 方法预设了一个在将来某个时间将要调用的 release 方法，该方法会把对象的引用计数减 1。至于在将来哪个时间，这取决于自动释放池额释放时间，当程序在自动释放池上下文中调用某个对象的 autorelease 方法时，该方法只是将该对象添加到自动释放池中，当该自动释放池释放时，自动释放池会让池中所有的对象执行 release 方法。

借助自动释放池的功能，可以控制当函数、方法需要返回一个对象时，先调用该对象的 autorelease 方法。这样只要在自动释放池上下文中调用该方法时，这个对象就会被添加到自动释放池中。程序只要控制自动释放池的销毁即可。

4.1.4 自动释放池的销毁时机与工作流程

自动释放池的销毁时机与其他普通对象的销毁时机相同，只要自动释放池的引用计数为 0，系统就会自动销毁自动释放池对象。从这个意义上可以看出，自动释放池并没与任何特别之处，它也只是一个 Objective-C 对象，同样遵守相同的对象回收机制。

通常来说，创建和销毁自动释放池的代码模板如下：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool allic] init];
...
```

//自动释放池的引用计数减 1

[pool release];

从上面的代码可以看出，当程序创建 NSAutoreleasePool 对象之后，该对象的引用计数为 1，接下来当程序调用该对象的 release 方法后，该对象的引用计数变为 0，系统准备调用 NSAutoreleasePool 的 dealloc 方法来回收该对象，系统会在调用 NSAutoreleasePool 的 dealloc 方法时回收该池中所有的对象。

下面使用 productItem()函数将 main()函数改为如下形式。

```
int main(int argc , char * argv[])
{
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    // it的引用计数为1
    FKItem* it = productItem();
    // 接下来可以调用it的方法
    NSLog(@"%ld" , it.retainCount);
    // ...
    // 创建一个FKUser对象，并将它添加到自动释放池
    FKUser* user = [[[FKUser alloc] init] autorelease];
    // 接下来可以调用user的方法
    NSLog(@"%ld" , user.retainCount);
    // ...
    // 系统将因为池的引用计数变为0而回收自动释放池，
    // 回收自动释放池时会调用池中所有对象的release方法
    [pool release];
}
```

在上面的程序中，第一行代码创建了一个 NSAutoreleasePool 对象，第六行代码将自动释放池的引用计数减 1，系统将因为池的引用计数变为 0 而回收自动释放池，回收自动释放池时会调用池中所有对象的 release 方法。

上面的程序在自动释放池上下文中一共得到了两个对象，其中 FKItem 对象通过 productItem()函数获得，该函数最后一行代码在返回 FKItem 对象是调用了 autorelease 方法，该方法会把该 FKItem 对象添加到自动释放池中；而 FKUser 对象则在创建完成后立即调用 autorelease 方法，该方法也会把该 FKUser 对象添加到自动释放池中。

从程序的运行结果来看，使用自动释放池后，程序只要把所有通过以 alloc、new、copy、mutableCopy 等开头的方法来创建出来的对象添加到自动释放池中，接下来程序只要控制释放自动释放池即可,自动释放池就会负责把容器中所有对象的引用计数减 1。

4.1.5 手动内存管理的规则总结

1.调用对象的 release 方法并不是销毁该对象，只是想该对象的引用计数减 1；当一个对象的引用计数为 0 时，系统会自动调用该对象的 dealloc 方法来销毁该对象。

2.当自动释放池被回收时，自动释放池会依次调用池中每个对象的 release 方法。如果该方法调用 release 方法后引用计数变为 0，那么该对象将要被销毁；否则该对象可以从自动释放池中“活”下来。

3.当程序使用以 alloc、new、copy、mutableCopy 开头的方法创建对象后，该对象的引用计数为 1，当不再使用该对象时，需要调用该对象的 release 方法或者 autorelease 方法。

4.如果使用 retain 方法为对象增加过引用计数，则用完该对象后需要调用 release 方法来减少该对象的引用计数，并保证 retain 次数与 release 次数相等。

5.如果通过其他方法过去了对象，且该对象是一个临时对象，若在自动释放池上下文中使用该对象，那么使用完成后无须理会该对象的回收，系统会自动回收该对象。如果程序需要保留这个临时对象，则需要手动调用 retain 来增加该对象的引用计数；或

者将该临时对象赋值给 retain、strong、或 copy 指示符修饰的属性。

6.在 Cocoa 或 iOS 的事件循环中，在每个事件处理方法执行之前都会创建自动释放池，方法执行完成后会自动回收自动释放池。如果希望自动释放池被回收后，池中的某些对象能“活”下来，程序必须增加该对象的引用计数，保证该对象的引用计数大于它调用 autorelease 的次数。

4.2 自动引用计数

自动引用计数（ARC）可以避免手动引用计数可能引入的一些潜在的陷阱，自动引用计数同样是基于引用计数的，只不过由系统来负责管理对象的引用计数，系统可以判断何时需要保持对象，何时需要自动释放对象。开发者不需要担心方法内创建的对象，编译器会管理好对象的内存，通过生成正确的代码去释放或保存返回的对象。

一旦启用了 ARC 机制，程序员就不允许在程序中调用对象的 retain、release 方法来改变对象的引用计数。如果在最新的 Xcode 中创建 iOS 项目，则默认已经启用了 ARC 机制。当然也可以关闭项目的 ARC 机制，或者项目需要单独对某个项目启用手动引用计数，也就是关闭 ARC 机制，则可以单独关闭某个文件的 ARC 机制。

在 ARC 模式下，只要没有强指针（强引用）指向对象，对象就会被释放。在 ARC 模式下，不允许使用 retain、release、retainCount 等方法。并且，如果使用 dealloc 方法，不允许调用[super dealloc]方法。ARC 模式下的 property 变量修饰词为 strong、weak，相当于 MRC 模式下的 retain、assign。strong 代替 retain 表示强引用；weak 代替 assign，声明了一个可以自动设置 nil 的若引用，但是比 assgin 多一个功能，指针指向的地址被释放之后，指针本身也会自动被释放。

5.总结

要想掌握 iOS 的内存管理，首先要深入理解引用计数(Reference Count)这个概念

以及内存管理的规则，Objective-C 的内存管理采用引用计数来控制对象的回收，当一个对象的应用计数为 0 时，系统就会调用该对象的 dealloc 方法来回收它。在没引入 ARC 之前，为了管理对象的引用计数，早起开发者可能需要手动管理对象的引用计数，通过 retain 和 release 方法来手动管理内存；引入 ARC（自动引用计数）机制之后，我们可以借助编译器来帮忙自动调用 retain 和 release 方法来简化内存管理和减低出错的可能性。虽然 strong 修饰符能够执行大多数内存管理，但它不能解决引用循环（Reference Cycle）问题，于是又引入另一个修饰符 weak。被 strong 修饰的变量都持有对象的所有权，而被 weak 修饰的变量并不持有对象所有权。

总而言之，引入 ARC（自动引用计数）机制之后，Objective-C 对象的引用计数已经变得非常简单，ARC 大大简化了 Objective-C 的内存管理。

参考文献

[1] Lonely_（简书作者）. iOS 内存管理机制解析. 2015(6)

[2] 李刚. 疯狂 iOS 讲义. 2015(4)

[3] （日)坂一树 （日)古本智彦. Objective-C 高级编程 iOS 与 OS X 多线程和内存管理. 2013(6)

[4] 刘乐廷 李敬兆. iOS 内存开发管理机制的研究. 2013(3)