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。

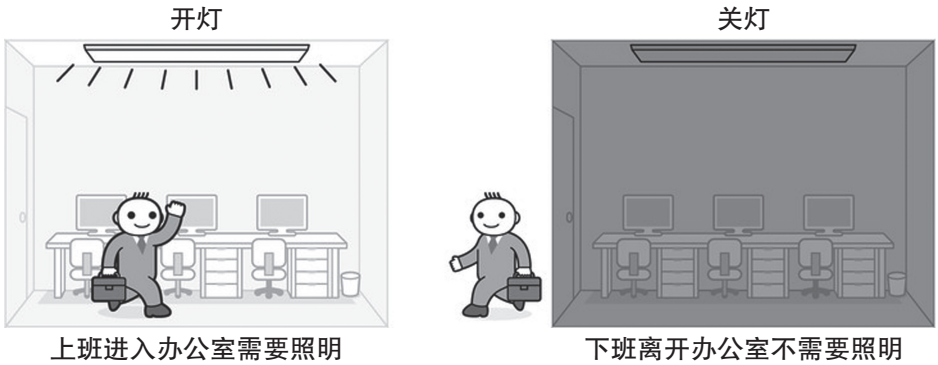

图 1-1　办公室照明

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。

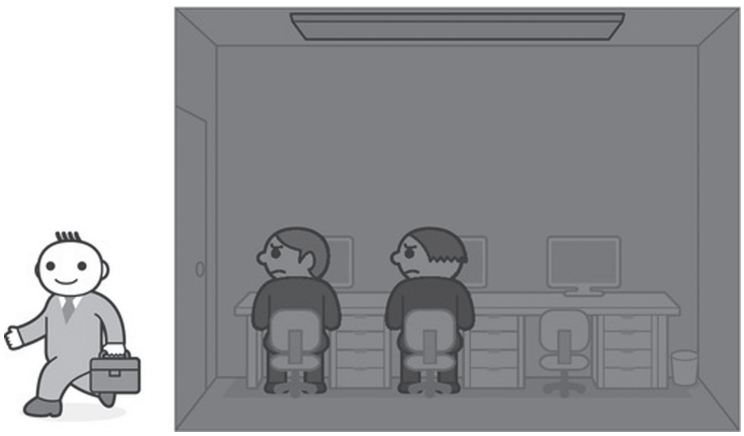

图 1-2　办公室里的照明问题

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。

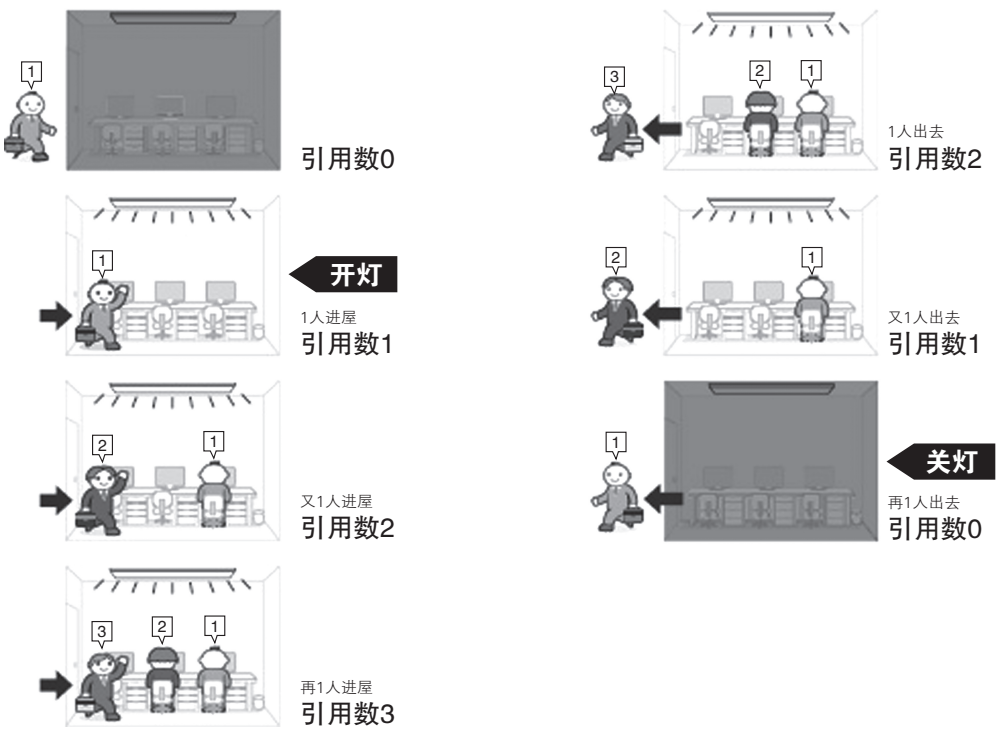

图 1-3　办公室里的照明管理

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。

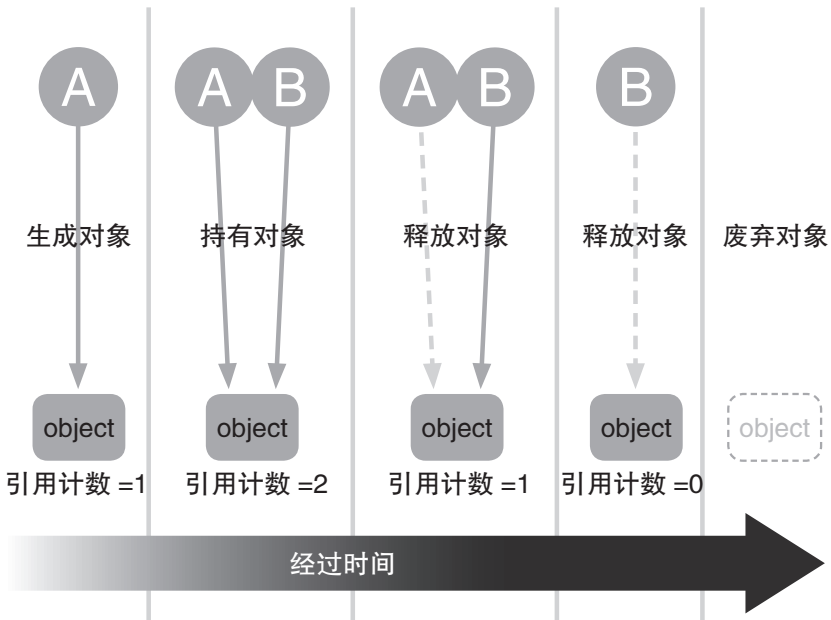

图 1-4　引用计数的内存管理

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。

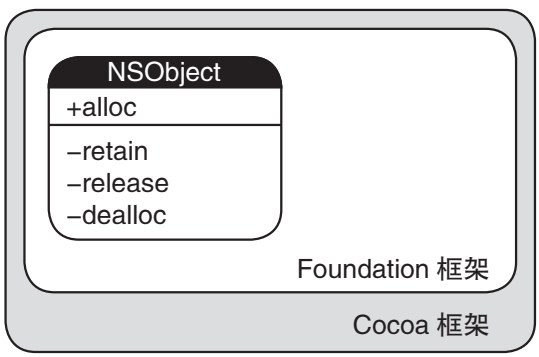

图 1-5　Cocoa 框架、Foundation 框架和 NSObject 类的关系

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

象。 autorelease 提供这样的功能，使对象在超出指定的生存范围时能够自动并正确地释放（调用 release 方法）。如图 1-6 所示。

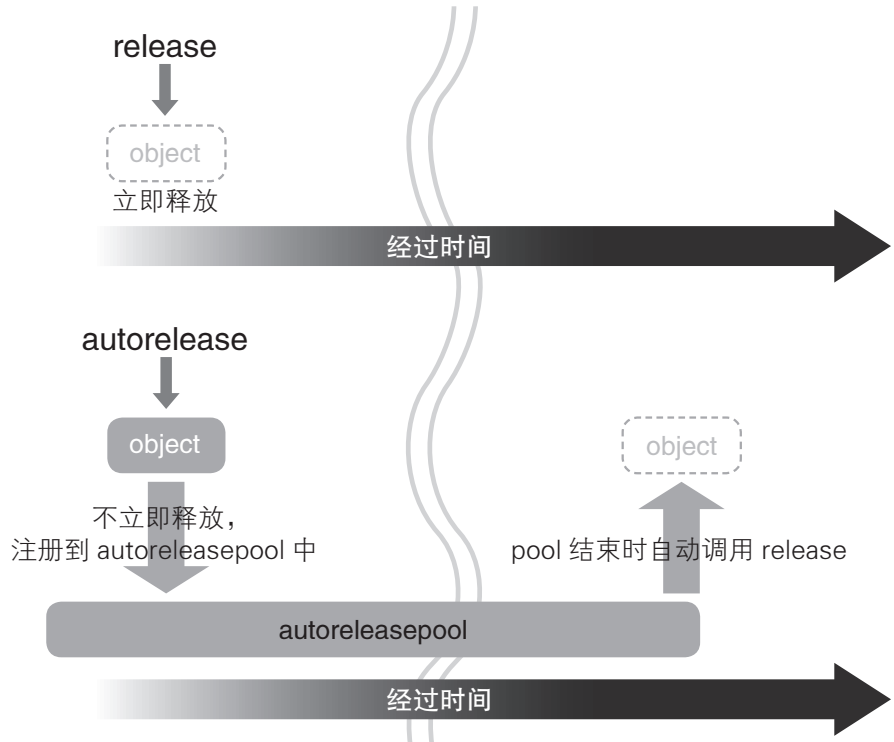

图 1-6　release 和 autorelease 的区别

在后面，对 autorelease 做了更为详细的解说，具体可参看 1.2.5 节。使用 NSMutableArray 类的 array 类方法等可以取得谁都不持有的对象，这些方法都是通过 autorelease 而实现的。此外，根据上文的命名规则，这些用来取得谁都不持有的对象的方法名不能以 alloc/new/copy/mutableCopy 开头，这点需要注意。

```
id obj1 = [obj0 object];

/*
 * 取得的对象存在，但自己不持有对象
 */
```

当然，也能够通过 retain 方法将调用 autorelease 方法取得的对象变为自己持有。

```
id obj1 = [obj0 object];

/*
 * 取得的对象存在，但自己不持有对象
```

```
 */

[obj1 retain];
/*
 * 自己持有对象
 */
```

无法释放非自己持有的对象

对于用 alloc/new/copy/mutableCopy 方法生成并持有的对象，或是用 retain 方法持有的对象，由于持有者是自己，所以在不需要该对象时需要将其释放。而由此以外所得到的对象绝对不能释放。倘若在应用程序中释放了非自己所持有的对象就会造成崩溃。例如自己生成并持有对象后，在释放完不再需要的对象之后再次释放。

```
/*
 * 自己生成并持有对象
 */

id obj = [[NSObject alloc] init];

/*
 * 自己持有对象
 */

[obj release];

/*
 * 对象已释放
 */

[obj release];

/*
 * 释放之后再次释放已非自己持有的对象!
 * 应用程序崩溃!
 *
 * 崩溃情况：
 *   再度废弃已经废弃了的对象时崩溃
 *   访问已经废弃的对象时崩溃
 */
```

或者在“取得的对象存在，但自己不持有对象”时释放。

```
id obj1 = [obj0 object];

/*
 * 取得的对象存在，但自己不持有对象
 */

[obj1 release];
```

```
/*
 * 释放了非自己持有的对象!
 * 这肯定会导致应用程序崩溃!
 */
```

如这些例子所示，释放非自己持有的对象会造成程序崩溃。因此绝对不要去释放非自己持有的对象。

以上四项内容，就是“引用计数式内存管理”的思考方式。

1.2.3 alloc/retain/release/dealloc 实现

接下来，以 Objective-c 内存管理中使用的 alloc/retain/release/dealloc 方法为基础，通过实际操作来理解内存管理。

OS X、iOS 中的大部分作为开源软件公开在 Apple Open Source[①] 上。虽然想让大家参考 NSObject 类的源代码，但是很遗憾，包含 NSObject 类的 Foundation 框架并没有公开。不过，Foundation 框架使用的 Core Foundation 框架的源代码，以及通过调用 NSObject 类进行内存管理部分的源代码是公开的。但是，没有 NSObject 类的源代码，就很难了解 NSObject 类的内部实现细节。为此，我们首先使用开源软件 GNUstep[②] 来说明。

GNUstep 是 Cocoa 框架的互换框架。也就是说，GNUstep 的源代码虽不能说与苹果的 Cocoa 实现完全相同，但是从使用者角度来看，两者的行为和实现方式是一样的，或者说非常相似。理解了 GNUstep 源代码也就相当于理解了苹果的 Cocoa 实现。

我们来看看 GNUstep 源代码中 NSObject 类的 alloc 类方法。为明确重点，有的地方对引用的源代码进行了摘录或在不改变意思的范围内进行了修改。

```
id obj = [NSObject alloc];
```

上述调用 NSObject 类的 alloc 类方法在 NSObject.m 源代码中的实现如下。

▼ GNUstep/modules/core/base/Source/NSObject.m alloc

```
+ (id) alloc
{
   return [self allocWithZone: NSDefaultMallocZone()];
}

+ (id) allocWithZone: (NSZone*) z
{
   return NSAllocateObject (self, 0, z);
}
```

① Apple Open Source　http://opensource.apple.com/。
② GNUstep　http://gnustep.org/。

通过 allocWithZone：类方法调用 NSAllocateObject 函数分配了对象。下面我们来看看 NSAllocateObject 函数。

▼ GNUstep/modules/core/base/Source/NSObject.m NSAllocateObject

```
struct obj_layout {
    NSUInteger retained;
};

inline id
NSAllocateObject ( Class aClass, NSUInteger extraBytes, NSZone *zone )
{
    int size = 计算容纳对象所需内存大小 ;
    id new = NSZoneMalloc ( zone,size ) ;
    memset ( new, 0, size ) ;
    new = ( id ) & (( struct obj_layout * ) new ) [1];
}
```

NSAllocateObject 函数通过调用 NSZoneMalloc 函数来分配存放对象所需的内存空间，之后将该内存空间置 0，最后返回作为对象而使用的指针。

专栏　区域

NSDefaultMallocZone、NSZoneMalloc 等名称中包含的 NSZone 是什么呢？它是为防止内存碎片化而引入的结构。对内存分配的区域本身进行多重化管理，根据使用对象的目的、对象的大小分配内存，从而提高了内存管理的效率。

但是，如同苹果官方文档 Programming With ARC Release Notes 中所说，现在的运行时系统只是简单地忽略了区域的概念。运行时系统中的内存管理本身已极具效率，使用区域来管理内存反而会引起内存使用效率低下以及源代码复杂化等问题。如图 1-7 所示。

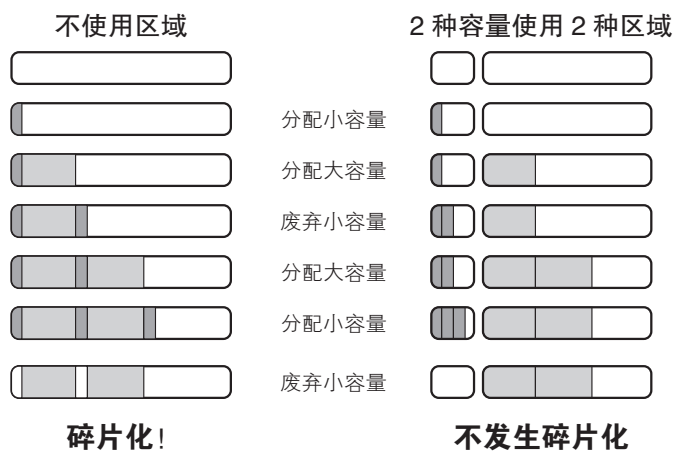

图 1-7　使用多重区域防止内存碎片化的例子

以下是去掉 NSZone 后简化了的源代码：

▼ GNUstep/modules/core/base/Source/NSObject.m alloc 简化版

```
struct obj_layout {
    NSUInteger retained;
};

+ ( id ) alloc
{
    int size = sizeof ( struct obj_layout ) + 对象大小 ;
    struct obj_layout *p = ( struct obj_layout * ) calloc ( 1,size ) ;
    return ( id )( p+1 ) ;
}
```

alloc 类方法用 struct obj_layout 中的 retained 整数来保存引用计数，并将其写入对象内存头部，该对象内存块全部置 0 后返回。以下用图示来展示有关 GNUstep 的实现，alloc 类方法返回的对象。如图 1-8 所示。

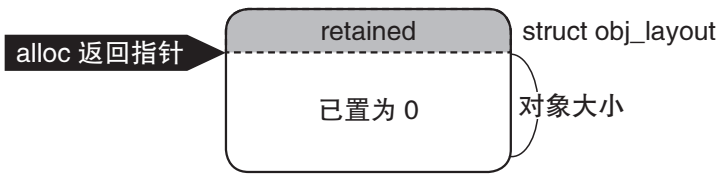

图 1-8　alloc 返回对象的内存图

对象的引用计数可通过 retainCount 实例方法取得。

```
id obj = [[NSObject alloc] init];
NSLog ( @"retainCount=%d", [obj retainCount] ) ;

/*
 * 显示 retainCount=1
 */
```

执行 alloc 后对象的 retainCount 是“1”。下面通过 GNUstep 的源代码来确认。

▼ GNUstep/modules/core/base/Source/NSObject.m retainCount

```
- ( NSUInteger ) retainCount
{
    return NSExtraRefCount ( self ) + 1;
}

inline NSUInteger
NSExtraRefCount ( id anObject )
{
    return (( struct obj_layout * ) anObject ) [-1].retained;
}
```

由对象寻址找到对象内存头部，从而访问其中的 retained 变量。如图 1-9 所示。

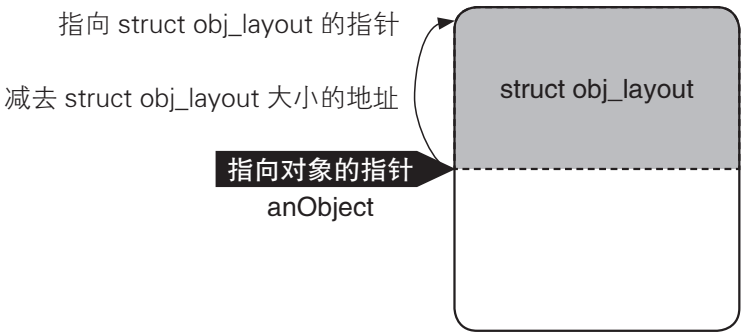

图 1-9　通过对象访问对象内存头部

因为分配时全部置 0，所以 retained 为 0。由 NSExtraRefCount（self）+ 1 得出，retainCount 为 1。可以推测出，retain 方法使 retained 变量加 1，而 release 方法使 retained 变量减 1。

```
[obj retain];
```

下面来看一下像上面那样调用出的 retain 实例方法。

▼ GNUstep/modules/core/base/Source/NSObject.m retain

```
- (id) retain
{
    NSIncrementExtraRefCount(self);
    return self;
}

inline void
NSIncrementExtraRefCount(id anObject)
{
    if (((struct obj_layout *)anObject)[-1].retained == UINT_MAX - 1)
        [NSException raise: NSInternalInconsistencyException
            format: @"NSIncrementExtraRefCount() asked to increment too far");

    ((struct obj_layout *)anObject)[-1].retained++;
}
```

虽然写入了当 retained 变量超出最大值时发生异常的代码，但实际上只运行了使 retained 变量加 1 的 retained++ 代码。同样地，release 实例方法进行 retained-- 并在该引用计数变量为 0 时做出处理。下面通过源代码来确认。

```
[obj release];
```

以下为此 release 实例方法的实现。

▼ GNUstep/modules/core/base/Source/NSObject.m release

```
- (void) release
{
    if (NSDecrementExtraRefCountWasZero(self))
        [self dealloc];
}

BOOL
NSDecrementExtraRefCountWasZero(id anObject)
{
    if (((struct obj_layout *)anObject)[-1].retained == 0) {
        return YES;
    } else {
        ((struct obj_layout *)anObject)[-1].retained--;
        return NO;
    }
}
```

同预想的一样，当 retained 变量大于 0 时减 1，等于 0 时调用 dealloc 实例方法，废弃对象。以下是废弃对象时所调用的 dealloc 实例方法的实现。

▼ GNUstep/modules/core/base/Source/NSObject.m dealloc

```
- (void) dealloc
{
    NSDeallocateObject(self);
}

inline void
NSDeallocateObject(id anObject)
{
    struct obj_layout *o = &((struct obj_layout *)anObject)[-1];
    free(o);
}
```

上述代码仅废弃由 alloc 分配的内存块。

以上就是 alloc/retain/release/dealloc 在 GNUstep 中的实现。具体总结如下：

- 在 Objective-C 的对象中存有引用计数这一整数值。
- 调用 alloc 或是 retain 方法后，引用计数值加 1。
- 调用 release 后，引用计数值减 1。
- 引用计数值为 0 时，调用 dealloc 方法废弃对象。

1.2.4 苹果的实现

在看了 GNUstep 中的内存管理和引用计数的实现后，我们来看看苹果的实现。因为 NSObject 类的源代码没有公开，此处利用 Xcode 的调试器（lldb）和 iOS 大概追溯出其实现过程。

在 NSObject 类的 alloc 类方法上设置断点，追踪程序的执行。以下列出了执行所调用的方法和函数。

```
+alloc
+allocWithZone:
class_createInstance
calloc
```

alloc 类方法首先调用 allocWithZone: 类方法，这和 GNUstep 的实现相同，然后调用 class_createInstance 函数[①]，该函数在 Objective-C 运行时参考中也有说明，最后通过调用 calloc 来分配内存块。这和前面讲述的 GNUstep 的实现并无多大差异。class_createInstance 函数的源代码可以通过 objc4 库[②] 中的 runtime/objc-runtime-new.mm 进行确认。

retainCount/retain/release 实例方法又是怎样实现的呢？同刚才的方法一样，下面列出各个方法分别调用的方法和函数。

```
-retainCount
__CFDoExternRefOperation
CFBasicHashGetCountOfKey
```

```
-retain
__CFDoExternRefOperation
CFBasicHashAddValue
```

```
-release
__CFDoExternRefOperation
CFBasicHashRemoveValue
（CFBasicHashRemoveValue 返回 0 时，-release 调用 dealloc）
```

各个方法都通过同一个调用了 __CFDoExternRefOperation 函数，调用了一系列名称相似的函数。如这些函数名的前缀“CF”所示，它们包含于 Core Foundation 框架源代码中，即是 CFRuntime.c 的 __CFDoExternRefOperation 函数。为了理解其实现，下面简化了 __CFDoExternRefOperation 函数后的源代码。

▼ CF/CFRuntime.c __CFDoExternRefOperation

```
int __CFDoExternRefOperation（uintptr_t op, id obj）{
    CFBasicHashRef table = 取得对象对应的散列表（obj）;
    int count;

    switch（op）{
    case OPERATION_retainCount:
```

① http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/ObjCRuntimeRef/reference.html。
② http://www.opensource.apple.com/source/objc4/。

```
            count = CFBasicHashGetCountOfKey(table, obj);
            return count;

        case OPERATION_retain:
            CFBasicHashAddValue(table, obj);
            return obj;

        case OPERATION_release:
            count = CFBasicHashRemoveValue(table, obj);
            return 0 == count;
        }
    }
```

__CFDoExternRefOperation 函数按 retainCount/retain/release 操作进行分发，调用不同的函数。NSObject 类的 retainCount/retain/release 实例方法也许如下面代码所示：

```
- (NSUInteger) retainCount
{
    return (NSUInteger) __CFDoExternRefOperation(OPERATION_retainCount, self);
}

- (id) retain
{
    return (id) __CFDoExternRefOperation(OPERATION_retain, self);
}

- (void) release
{
    return __CFDoExternRefOperation(OPERATION_release, self);
}
```

可以从 __CFDoExternRefOperation 函数以及由此函数调用的各个函数名看出，苹果的实现大概就是采用散列表（引用计数表）来管理引用计数。如图 1-10 所示。

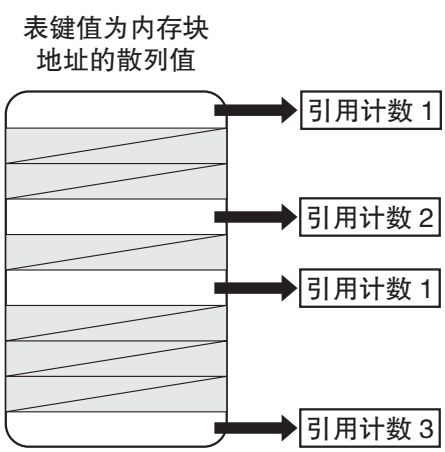

图 1-10 通过散列表管理引用计数

GNUstep 将引用计数保存在对象占用内存块头部的变量中，而苹果的实现，则是保存在引用计数表的记录中。GNUstep 的实现看起来既简单又高效，而苹果如此实现必然有它的好处。下面我们来讨论一下。

通过内存块头部管理引用计数的好处如下：

- 少量代码即可完成。
- 能够统一管理引用计数用内存块与对象用内存块。

通过引用计数表管理引用计数的好处如下：

- 对象用内存块的分配无需考虑内存块头部。
- 引用计数表各记录中存有内存块地址，可从各个记录追溯到各对象的内存块。

这里特别要说的是，第二条这一特性在调试时有着举足轻重的作用。即使出现故障导致对象占用的内存块损坏，但只要引用计数表没有被破坏，就能够确认各内存块的位置。如图 1-11 所示。

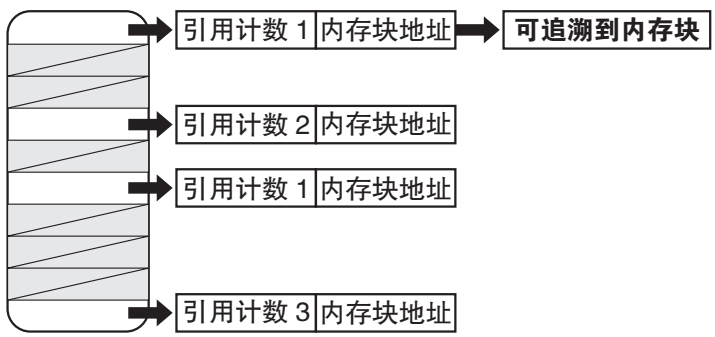

图 1-11　通过引用计数表追溯对象

另外，在利用工具检测内存泄漏时，引用计数表的各记录也有助于检测各对象的持有者是否存在。

通过以上解说即可理解苹果的实现。

1.2.5　autorelease

说到 Objective-C 内存管理，就不能不提 autorelease。

顾名思义，autorelease 就是自动释放。这看上去很像 ARC，但实际上它更类似于 C 语言中自动变量[①]（局部变量）的特性。

我们来复习一下 C 语言的自动变量。程序执行时，若某自动变量超出其作用域，该自动变量将被自动废弃。

① http://en.wikipedia.org/wiki/Automatic_variable。

```
{

    int a;

}   /*
     * 因为超出变量作用域，
     * 自动变量“int a”被废弃，不可再访问
     */
```

autorelease 会像 C 语言的自动变量那样来对待对象实例。当超出其作用域（相当于变量作用域）时，对象实例的 release 实例方法被调用。另外，同 C 语言的自动变量不同的是，编程人员可以设定变量的作用域。

autorelease 的具体使用方法如下：

（1）生成并持有 NSAutoreleasePool 对象；
（2）调用已分配对象的 autorelease 实例方法；
（3）废弃 NSAutoreleasePool 对象。

图 1-12　NSAutoreleasePool 对象的生存周期

NSAutoreleasePool 对象的生存周期相当于 C 语言变量的作用域。对于所有调用过 autorelease 实例方法的对象，在废弃 NSAutoreleasePool 对象时，都将调用 release 实例方法。如图 1-12 所示。

用源代码表示如下：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

id obj = [[NSObject alloc] init];

[obj autorelease];

[pool drain];
```

上述源代码中最后一行的“[pool drain]”等同于“[obj release]”。

在 Cocoa 框架中，相当于程序主循环的 NSRunLoop 或者在其他程序可运行的地方，对 NSAutoreleasePool 对象进行生成、持有和废弃处理。因此，应用程序开发者不一定非得使用 NSAutoreleasePool 对象来进行开发工作。如图 1-13 所示。

图 1-13　NSRunLoop 每次循环过程中 NSAutoreleasePool 对象被生成或废弃

尽管如此，但在大量产生 autorelease 的对象时，只要不废弃 NSAutoreleasePool 对象，那么生成的对象就不能被释放，因此有时会产生内存不足的现象。典型的例子是读入大量图像的同时改变其尺寸。图像文件读入到 NSData 对象，并从中生成 UIImage 对象，改变该对象尺寸后生成新的 UIImage 对象。这种情况下，就会大量产生 autorelease 的对象。

```
for ( int i = 0; i < 图像数 ; ++i ) {

    /*
     * 读入图像
     * 大量产生 autorelease 的对象。
     * 由于没有废弃 NSAutoreleasePool 对象
     * 最终导致内存不足!
     */
}
```

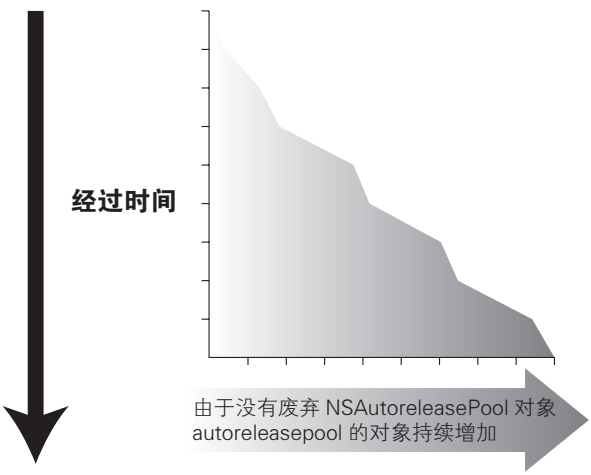

图 1-14　大量产生 autorelease 的对象

在此情况下，有必要在适当的地方生成、持有或废弃 NSAutoreleasePool 对象。

```
for ( int i = 0; i < 图像数 ; ++i ) {

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    /*
     * 读入图像
     * 大量产生 autorelease 的对象。
     */

    [pool drain];

    /*
     * 通过 [pool drain],
     * autorelease 的对象被一起 release。
     */
}
```

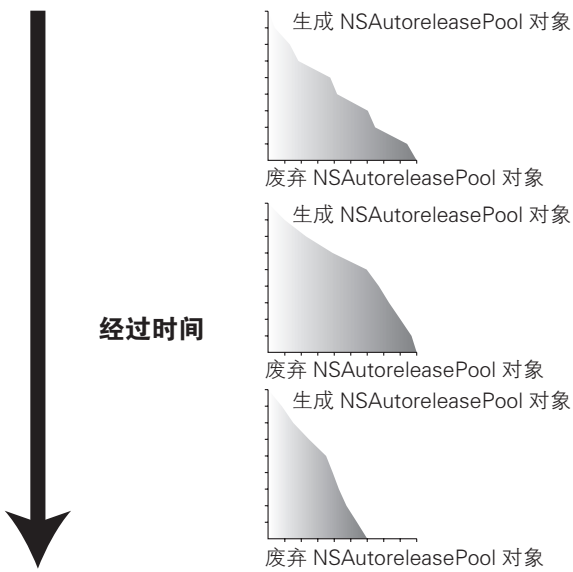

图 1-15　适当地释放 autorelease 的对象

另外，Cocoa 框架中也有很多类方法用于返回 autorelease 的对象。比如 NSMutableArray 类的 arrayWithCapacity 类方法。

```
id array = [NSMutableArray arrayWithCapacity:1];
```

此源代码等同于以下源代码。

```
id array = [[[NSMutableArray alloc] initWithCapacity:1] autorelease];
```

1.2.6　autorelease 实现

autorelease 是怎样实现的呢？为了加深理解，同 alloc/retain/release/dealloc 一样，我们来查看一下 GNUstep 的源代码。

```
[obj autorelease]
```

此源代码调用 NSObject 类的 autorelease 实例方法。

▼ GNUstep/modules/core/base/Source/NSObject.m autorelease

```
- ( id ) autorelease
{
   [NSAutoreleasePool addObject:self];
}
```

autorelease 实例方法的本质就是调用 NSAutoreleasePool 对象的 addObject 类方法。

专栏　提高调用 Objective–C 方法的速度

GNUstep 中的 autorelease 实际上是用一种特殊的方法来实现的。这种方法能够高效地运行 OS X、iOS 用应用程序中频繁调用的 autorelease 方法，它被称为“IMP Caching”。在进行方法调用时，为了解决类名 / 方法名以及取得方法运行时的函数指针，要在框架初始化时对其结果值进行缓存。

```
id autorelease_class = [NSAutoreleasePool class];
SEL autorelease_sel = @selector ( addObject: ) ;
IMP autorelease_imp = [autorelease_class methodForSelector:autorelease_sel];
```

实际的方法调用就是使用缓存的结果值。

```
- ( id ) autorelease
{
    ( *autorelease_imp )( autorelease_class, autorelease_sel, self ) ;
}
```

这就是 IMP Caching 的方法调用。虽然同以下源代码完全相同，但从运行效率上看，即使它依赖于运行环境，一般而言速度也是其他方法的 2 倍。

```
- ( id ) autorelease
{
    [NSAutoreleasePool addObject:self];
}
```