iOS 高级内存管理编程指南

译者：张立明 baccc@sina.com

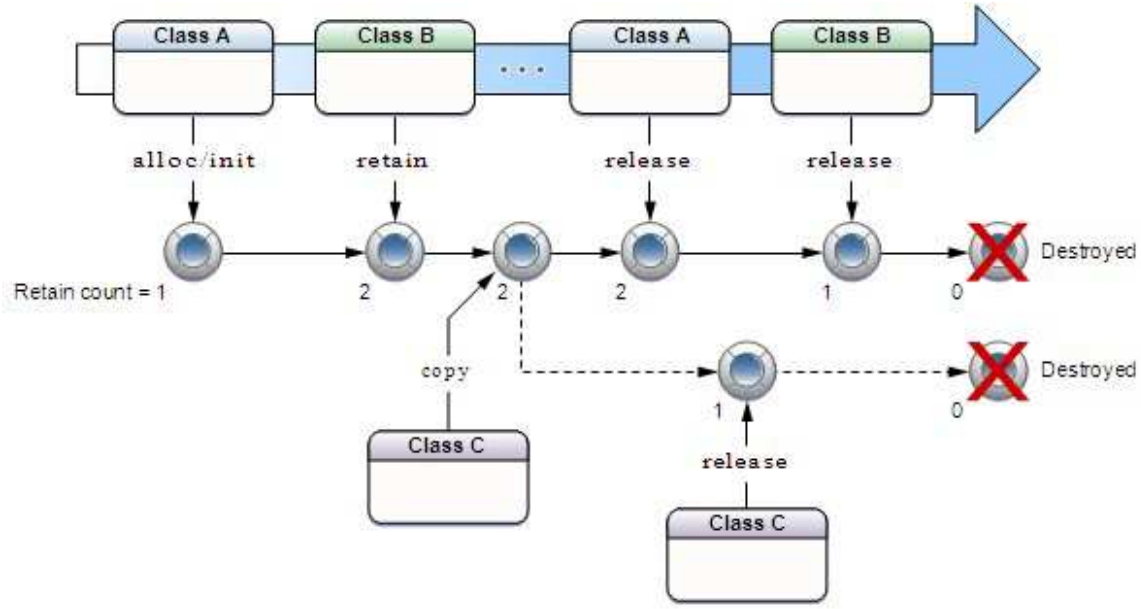
原稿为：苹果公司《Advanced Memory Management Programming Guide》 2011-9-28 版

目录

1 关于内存管理

应用程序的内存管理是一个在程序运行时进行内存分配，使用内存、结束时释放内存的过程。书写良好的程序，会尽可能少占用内存。在 Objectiv-C 中，这个过程也是一个在很多片代码或者数据中传播有限内存资源的“所有权”（Ownership）的方式。读完本指南，你将可以“显式地”管理对象的生命周期，并在不用的时候释放他们。

内存管理通常被认为是针对单个的对象进行的，但实际上我们的任务是管理“对象图”（Object Graph），你需要确保除了你真的需要的对象之外，内存中没有其它的对象。



1.1 概述

Objective-C 提供了三种内存管理方式：

1. 本文中将要讲述的一种方式，称为“手工持有-释放”（Manual Retain-Release）或 MRR。你通过跟踪你所拥有的对象来“显式地”管理内存。这种方式采用了一种称为“引用计数”的模型。该模型由基础类 NSObject 和运行时（Runtime Environment）共同提供。

2. 自动引用计数（Automatic Reference Counting），或 ARC，的方式。系统采用和 MRR 相同的引用计数系统，但是在编译时（Compile-time）插入了内存管理的方法。对于新的工程项目，强烈建议使用 ARC 方式，这样你不需要理解本文所述的底层实现。不过，个别情况下，你会受益于对这些底层实现的理解。更多关于 ARC 的讲述，请参看 *Transitioning*

To ARC Release Notes。

3. 垃圾回收的方式。系统自动跟踪对象跟对象之间的引用关系。对于没有引用的对象，自动进行回收。这种机制和前面说的 MRR 和 ARC 都不同，且只能在 Mac OS 下使用，iOS 下是不行的。更多关于垃圾回收机制的内容，请参看 *Garbage Collection Programming Guide*。

如果你正准备写 iOS 代码，你必须用显式的内存管理（也就是本文的主题）。如果你想写库函数（Library Routine）、插件（Plug-in）、或者共享代码（Shared Code）（运行于“有”或者“没有”垃圾回收的进程中），你将使用本文中的内存管理技术来写代码。

1.2 防止内存泄露的最佳实践

错误的内存管理往往包括两类：

1. 释放（free）或者覆盖（over-write）正在使用中的数据。

   造成内存异常，导致应用程序崩溃，甚至导致数据损坏。

2. 不用的数据却不释放，从而导致内存泄露。

   内存泄露，就是有内存分配但是不释放它，哪怕这块内存已经不用了。泄露，导致你的应用程序占用越来越多的内存，并导致整体性能的下降，或者在 iOS 平台上导致应用终止。

如果你总是考虑内存管理的实现细节，而不是你实际的管理目标，那么你会感觉到从“引用计数”的角度理解内存管理实际是极其困难的。所以，你真正应该考虑的是对象的“所有权”（Ownership）以及对象图（Object Graph）。

当一个方法所返回的对象，其所有权属于你的时候， Cocoa 用一种非常直接的命名规范来告诉你。请参看 内存管理策略

尽管最基础的策略也是最直接的，有一些有效的做法可以让内存管理更加容易，从而帮助你实现程序的稳定性和健壮性，从而使其占用更少的资源。请参看 内存管理实战

自动释放池（Autorelease Pool）使得你可以用一种不同的方式来发送 release 消息。当你想放弃对一个对象的所有权，但又不想让这个所有权的释放立即生效（比如，你在方法中要返回这个对象），这种机制就很有用了。有几种情况你应该需要使用自动释放池。 请参看 使用 Autorelease 池

1.3 使用分析工具来调试内存问题

为了发现编译时的问题，可以使用 Xcode 自带的 Clang Static Ananlyzer。

如果内存管理问题依然存在，还有其他的工具和技术可以帮助你分析问题。

这些工具和技术在技术文章 TN2239， iOS Debugging Magic 中描述。更确切地说，是使用 NSZombie 来发现 release 过多的对象。

你还可以使用 Instruments 来跟踪引用计数事件，并寻找内存泄露。参看 *Viewing and Analyzing Trace Data.*

2 内存管理策略

通过引用计数来进行内存管理的基本模型是由 NSObject 协议中的方法以及标准的方法命名规范来提供的。NSObject 还定义了一个 dealloc 方法，该方法会在对象需要析构的时候自动调用。本文描述了在 cocoa 程序中正确管理内存的基本规则，并提供了正确的示例。

2.1 基本内存管理规则

内存管理的模型，是基于对象的“所有权”（ownership）的。任何一个对象都可以有一个或者更多的“所有者”（owner）。当一个对象有至少 1 个所有者的时候，该对象存在；没有了所有者时，系统自动析构它。为了更清晰描述你何时拥有一个对象、何时放弃所有权，Cocoa 遵循下面的策略：

你拥有你所创建的对象

你如果用下面字母作为开头的方法来创建对象，那么你将拥有这个对象：alloc、new、copy、mutableCopy。比如，alloc、newObject、或者 mutableCopy 等方法。

你可以用 retain 来实现对一个对象的所有

如果你在一个方法体中，得到了一个对象，那么这个对象在本方法内部是一直都有效的。而且你还可以在本方法中将这个对象作为返回值返回给方法的调用者。在下面两种状况下，你需要用 retain：（1）在访问方法（getter、setter）或者 init 方法中，你希望将得到的返回对象作为成员变量（property）来存储。 （2）在执行某些操作时，你担心在过程中对象变得无效。（在 避免你正在使用的对象被 dealloc 中详细解释。）

你不再需要一个对象时，你必须放弃对对象的持有

通过向对象发送 release 消息或者 autorelease 消息来放弃所有权。用 Cocoa 的术语说，所谓放弃所有权，就是 release 一个对象。

对于你正在使用的对象，不要 release 它

这一点跟上一条对应，无需多言。

一个简单例子

下面的代码片段，做了示范：

```
{
    Person *aPerson =[[Person alloc] init];
    // …
    NSString *name = person.fullName;
    // …
    [aPerson release];
}
```

这里的 Person 对象是使用 alloc 来初始化，然后用 release 消息来释放的。Person 的 name 因为不是使用“所有的”方法（alloc, new, copy,…）来得到的，所以也不必 release。请注意，这里用的是 release 而非 autorelease。

2.2 延时 release—使用 autorelease

当你需要延时 release 方式时，就需要 autorelease 了，特别是当你从方法中返回一个对象的时候。比如，你可以这样来实现 fullName: 方法：

```
-(NSString *) fullName{
   NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@",
self.firstName, self.lastName] autorelease];
    return string;
}
```

上例中，你使用 alloc 方法创建了 string，所以你有该对象的所有权，因此你有义务在失去对它的引用前放弃该所有权。但如果你用 release，那么这种所有权的放弃是“即刻的”，立即生效。可是我们却要将这个对象 return，这将造成 return 时对象已经实际失效，方法实际上返回了一个无效的对象。我们采用 autorelease 来声明（译者：注意这里仅仅是一种意愿的表达，而非实际放弃的动作。）我们对所有权的放弃，但是同时允许 fullName: 方法的调用者来使用该对象。

你还可以按下面做法来实现这个方法：

```
-(NSString *) fullName{
    NSString *string = [NSString stringWithFormat:@"%@ %@", self.firstName,
self.lastName] ;
    return string;
}
```

根据我们说的基本规则，你对 stringWithFormat: 所返回的 string 没有所有权（译者：请注意到这里并没有使用 alloc，方法名也不是以 init 开始）。 因此你可以直接返回 string 给方法的调用者。

而下面的做法就是错误的了：

```
-(NSString *) fullName{
   NSString *string = [[NSString alloc] initWithFormat:@"%@ %@",
self.firstName, self.lastName];
    return string;
}
```

根据方法的命名规范，从这个方法的名字（fullName）看不出 fullName 方法的调用者将拥有返回的对象（译者：因为它并没有以 alloc、 new、 init 等开头)。因此，该方法的调用者自然也没有理由来 release 这个返回的 string。这将最终导致内存泄露。

2.3 通过引用（Reference）来返回的对象，你没有所有权

在 Cocoa 中，有些方法返回的对象是 reference(即，这些返回对象的类型是 ClassName ** 或者 id *)。 常见的情况是当出现错误异常时，一个 NSError 对象被用来承载错误的信息。比如 initWithContentsOfUrl:options:error(NSData) 和 initWithContentsOfFile:encoding:error(NSString) （译者：注意这里的 NSData 和 NSString 都没有 *)。

这种情况下，我们前面说的规则依然有效：当你调用这类方法的时候，你没有创建 NSError 对象,因此你没有对它的所有权,也不必 release 它。如下面所示(译者:注意,error 不需要 release):

```
NSString *fileName = <#Get a file name#>;
NSError * error = nil;
NSString *string =[[NSString alloc] initWithContentsOfFile:fileName
encoding:NSUTF8StringEncoding error:&error];
If(string == nil){
   //Deal with error
}
// …
[string release];
```

2.4 实现对象的 dealloc

NSObject 类定义了一个名为 dealloc 的方法。这个方法在对象无主（没有所有者）的情况下，当内存回收的时候会由系统自动调用。用 Cocoa 的术语讲，就叫 free 或者 deallocate。Dealloc 方法的作用就是释放对象的内存，并弃掉它持有的任何资源---以及它对其他对象的所有权。

下面的例子示范了你应该如何实现 Person 类的 dealloc 方法：

```
@interface Person : NSObject {}
```

```
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end
@implementation Person
@synthesize firstName=_firstName, lastName=_lastName;
// ...
- (void)dealloc
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```

重要：

你永远不需要直接调用另一个对象的 dealloc 方法。

你必须在末尾调用 super 类的实现方法。

你不可以把系统的资源和对象的生命周期进行绑定。请参看 Don’t Use Dealloc To Manage Scarce Resources。 （译者：就是说你不能等对象被 dealloc 时才放弃对关键系统资源的独占。原因见下一段。）

因为进程的内存会在退出时自动回收，当应用退出时，对象可能收不到 dealloc 这样的消息。和调用所有对象的内存管理方法的方式来比，操作系统的这种做法是高效率的。

2.5 Core Foundation 使用了类似但却不同的规则

Core Foundation 的对象采用了类似的内存管理方式（详细请参看：Memory Management Programming Guide for Core Foundation)。Cocoa 和 Core Foundation 在命名规则上是不同的。具体说，就是 Core Foundatoin 的创建规则，不适用于返回 Objective-C 对象的那些方法。比如下面的代码片段中，你没有责任或义务来释放对 myInstance 的所有权：

```
MyClass *myInstance = [MyClass createInstance];
```

3 内存管理实战

我们前面介绍了内存管理策略的一些基本概念。这些概念非常简单直接，但有些实战方面的做法可以使得内存管理更加容易，进而保证你的应用稳定而健壮，减少对系统资源的占用。

3.1 使用访问方法（Accessor Method）使得内存管理更加容易

如果你的类（Class），其成员为对象（Object），你必须保证这个对象的值在你需要使用它的时候，没有被 dealloc。所以你必须在 set 值的时候获得该对象的所有权。你还必须保证对这些对象所有权的放弃。

这个工作看似非常枯燥乏味。如果你坚持用 get 和 set 这种方法方法来实现，那么内存管理的问题出现的几率就大大增加了。如果你在代码中对成员对象到处使用 retain 和 release，那你这么做是错误的。

我们假定有一个 Counter 对象，你用它来实现计数。

```
@interface Counter : NSObject {
    NSNumber *_count;
}
@property (nonatomic, retain) NSNumber *count;
@end;
```

这里的 property 定义了两个访问方法（get set）。通常而言，这些方法由编译器来合成，但如果你知道这些方法是如何实现的，还是会帮助你理解问题的。

在 get 方法中，就是返回实例的变量，所以不必 retain 和 release：

```
- (NSNumber *)count {
    return _count;
}
```

在 set 方法中，你必须考虑到的是：新的值可能随时被 dealloc。因此你必须通过发送 retain 消息来取得对新值的所有权，进而保证 dealloc 不会发生。你还必须对旧值发送 release 消息。在 Objective-c 中，对一个 nil 发送消息是没问题。因此就算_count 还没有旧值，也不会出错。你必须在[newCount retain]之后再（对旧值）发送 release，因为你不想因为意外而造成 dealloc（译者：意思是说，如果你先调用旧值的 dealloc，再回过头来调用新值的 retain，恐怕为时已晚。）。

```
- (void)setCount:(NSNumber *)newCount {
```

```
    [newCount retain];
    [_count release];
    // Make the new assignment.
    _count = newCount;
}
```

3.2 使用访问方法（get set）来设置 property 属性的值

假定你要实现一个方法来复位（reset）计数器，好几个可行的做法。第一种做法就是用 alloc 来新建一个 NSNumber 实例，然后再对应一个 release。

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [self setCount:zero];
    [zero release];
}
```

第二种做法是使用快速构造器（Convenience Constructor）来创建一个新的 NSNumber。这种情况下你不需要 release 和 retain。

```
- (void)reset {
    NSNumber *zero = [NSNumber numberWithInteger:0];
    [self setCount:zero];
}
```

请注意，上面这些都用到了 set 方法。

下面的做法，对于简单的情况而言，肯定是没问题的。但是，因为它的实现绕开了 set 方法，那么在特定情况下（比如当你忘记了 retain 或者 release；再比如，当对这个变量的内存管理发生了变化。）会导致内存泄露：

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [_count release];
    _count = zero;
}
```

还需要注意到的是，如果你使用了 Key-Value Observing，那么这种对于值的复位就跟 KVO 不兼容了。 （译者： Key –Value Observing，即后面的 KVO）。

3.3 不要在初始化的方法中，或者 dealloc 方法中使用访问方法（get set）

不允许是用访问方法（get 和 set）的地方就限于初始化方法和 dealloc 方法。为了初始化一个 counter，并将值设置为 0，你需要这么来实现初始化方法：

```
- init {
    self = [super init];
    if (self) {
        _count = [[NSNumber alloc] initWithInteger:0];
    }
    return self;
}
```

为了让 counter 的初始化值为非 0 值，你可以实现一个名为 initWithCount:的方法：

```
- initWithCount:(NSNumber *) startingCount {
    self = [super init];
    if (self) {
        _count = [startingCount copy];
}
    return self;
}
```

因为 Counter 类有一个成员对象实例，你就必须实现 dealloc 方法。这个方法通过发 release 消息来放弃对于所有其他对象的所有权，然后，在最后时刻调用 super 对象的实现：

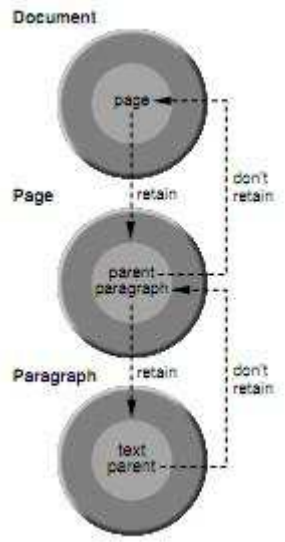
```
- (void)dealloc {
    [_count release];
    [super dealloc];
}
```

3.4 使用弱引用来避免所有权的死锁

Retain 一个对象，实际是对一个对象的强引用（strong reference）。一个对象在所有的强引用都解除之前，是不能被 dealloc 的，这导致一个被称为“环形持有”的问题：两个对象相互强引用（可能是直接引用，也可能是通过其他对象间接地引用。）

下图所示的对象关系就构成了一个环形持有。Document 对象持有多个 Page 对象，每个 Page 对象又具有一个 Document 引用来指示它归属的文档。这样每个对象都不能被 dealloc。在全部 Page 对象都 release 之前，Document 对象的引用数永远不会为 0；而如果 Document 对象存在，Page 对象也无法被 release。

图 1 环形持有



使用弱引用是避免环形持有的有效办法。所谓的弱引用，就是一种非持有的关系，已经被引用的对象不对它的属主进行持有。

为了实现上面的对象图，肯定是需要强引用的（如果只有弱引用，那么 Page 和 Paragraph 就没有了属主，造成他们不会被 dealloc）。Cocoa 设定了一个规则，那就是：父对象建立对子对象的强引用，而子对象只对父对象建立弱引用。

在 Cocoa 中，弱引用的例子包括（但不限于）Table 表格的数据源、Outline 视图项目（Outline view item）、通知观察者（Notification Observer）和其他的 target 以及 delegate。

对你弱引用的对象发送消息时，需要注意：当你发送消息给一个被 dealloc 的弱引用对象时，你的程序会崩溃。因此，你必须细致地判断对象是否有效。多数情况下，被弱引用的对象是知道其他对象对它的弱引用的（比如环形持有的情形），所以需要通知其他对象它自己的 dealloc。举例，当你向 Notification Center 注册一个对象时，Notification Center 对这个对象是弱引用的，并且在有消息需要通知到这个对象时，就发送消息给这个对象。当这个对象 dealloc 的时候，你必须向 Notification Center 取消这个对象的注册。这样，这个 Notification Center 就不会再发送消息给这个不存在的对象了。同样，当一个 delegate 对象被 dealloc 的时候，必须向其他对象发送一个 setDelegate:消息，并传递 nil 参数，从而将代理的关系撤销。这些消息通常在对象的 dealloc 方法中发出。

3.5 避免你正在使用的对象被 dealloc

Cocoa 的所有权策略是这么说的：返回的对象，在调用者的调用方法中，始终保持有效。所以说，在当前方法内部，不必担心你收到的返回对象会被 dealloc。对于你的代码而言，通过 getter 方法收到的返回对象是一个被缓存的实例，还是计算出来的实例，这并不重要，重要的是这个对象在你使用它的时候会一直有效。

这个策略并非放之四海而皆准，在有些情况下是特例，特别是下面两种情况：

1. **当一个对象从 collection 中删除的时候**

```
heisenObject = [array objectAtIndex:n];
[array removeObjectAtIndex:n];
// heisenObject could now be invalid.
```

当一个对象从 collection 中删除的时候，系统立刻调用了该对象的 release 方法（而不是 autorelease）。如果这时候，这个 collection 是该对象的唯一属主，那么这个对象（本例中的 heisenObject）会立刻被 dealloc。

2. **当父对象被 dealloc 的时候**

```
id parent = <#create a parent object#>;
// ...
heisenObject = [parent child] ;
[parent release]; // Or, for example: self.parent = nil;
// heisenObject could now be invalid.
```

有时候，你是从一个对象来获取另一个对象的，然后你直接或间接地 release 了父对象。如果对父对象的 release 造成了它被 dealloc，且这时该父对象恰好是子对象的唯一属主，那么子对象（本例中的 heisenObject）会同时也被 dealloc（这里我们假定在父对象的 dealloc 方法中，对子对象发送的是 release 消息，而非 autorelease）。

3.6 不要使用 dealloc 来管理关键系统资源

通常，你不应该在 dealloc 中来管理稀缺系统资源，比如文件句柄、网络连接、缓存等。更具体地说，你不可能设计出一个类，你想让系统什么时候调用 dealloc，系统就什么时候调用（译者：因为你能做的是 release，至于 release 是否会导致系统一定调用 dealloc，还要看这个对象有没有其他属主）。因为系统性能的下降、系统自身的 Bug，有可能 dealloc 的调用被推迟搁置。

正确做法是，如果你的对象管理了稀缺资源，它就必须知道它什么时候不再需要这些

资源，并在此时立即释放资源。通常情况下，此时，你会调用 release 来 dealloc，但是因为此前你已经释放了资源，这里就不会遇到任何问题。

如果你把资源管理的职能交给 dealloc，那么会暴露好多问题，比如：

1. 对象图的拆除顺序问题

实际上，对象图的拆除是没有任何顺序保证的。也许你认为、你希望有一个具体明确的顺序，但事实是没有。如果对象被放到了 autorelease 池，这个拆除的过程也会发生变化，并导致你无法预见的后果。

2. 系统稀缺资源不能回收

内存泄露问题，是系统的缺陷，应该被修正。但问题是通常这个问题不是立刻暴露出来的。如果没有保留的时候，你认为某个资源已经释放，而实际上没有释放，你就会面临更加严重的问题了。比如，如果文件句柄被用光了，其结果将是你无法保存数据。

3. 释放资源的操作由其他线程来做

如果对象在一个不确定的时刻被放到了 autorelease 池中，它将被线程池中的线程来 dealloc。这对于有些只能供单一线程来访问的资源而言，是致命的错误。

3.7 Collection 容器拥有其包容的对象的所有权

如果一个对象被放到了 collection 容器（Array、Dictionary、Set）中，Collection 会取得对该对象的所有权。当容器自己 release 的时候，或者该对象从容器中删除时，collection 会放弃对该对象的所有权。举例说明，如果要建一个存放数字的数组，你可以按下面方法来做：

```
NSMutableArray *array = <#Get a mutable array#>;
NSUInteger i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *convenienceNumber = [NSNumber numberWithInteger:i];
    [array addObject:convenienceNumber];
}
```

这时你并没有使用 alloc，因此你也不必用 release。你不需要调用新数值（convenienceNumber）的 retain，因为数组会这么做：

```
NSMutableArray *array = <#Get a mutable array#>;
NSUInteger i;
```

```
// ...
for (i = 0; i < 10; i++) {
    NSNumber *allocedNumber = [[NSNumber alloc] initWithInteger: i];
    [array addObject:allocedNumber];
    [allocedNumber release];
}
```

这种做法，我们在 for 循环内部向 allocedNumber 发送了与 alloc 相对应的 release 消息。因为 Array 的 addObject:方法实际上对这个对象做了 retain 处理，那么这个对象（allocedNumber）不会因此而被 dealloc。

我们换位思考一下，假定你自己就是这个 Collection 类的作者。你要确保加入的对象只要继续存在于 Collection 里，就不应该被 dealloc，因此你在添加这个对象时，向它发送了 retain 消息，删除这个对象时，向它发送了 release 消息。当你这个 collection 类自己 dealloc 时，对容器内所有的对象发 release。

3.8 所有权策略通过引用计数来实现

所有权策略是通过引用计数来实现的，通常称之为“retain count”。每个对象都有一个 retain count。

- ✓ 当新建一个对象时，它的 retain count 为 1；
- ✓ 发送 retain 消息给一个对象时，它的 retain count 加 1；
- ✓ 发送 release 消息给一个对象时，它的 retain count 减 1；
- ✓ 发送 autorelease 消息，它的 retain count 将在未来某个时候减 1；
- ✓ 如果 retain count 是 0，就会被 dealloc。

重要：其实你应该没有理由想知道一个对象的 retain count。这个数值有时候会造成对你的误导：你不知道实际上有些系统框架的对象会对你关注的那个对象进行 retain。在调试内存问题的时候，你只需要遵守所有权规则就行了。

4 使用 Autorelease 池

Autorelease 池的机制，为你提供了一个“延时”release 对象的机制。当你既想放弃对象所有权，又不想立即发生放弃行为的时候（比如你在方法中，把一个持有的对象作为返回值 return）。这时候，你不是必须要创建一个 autorelease 池，但个别情况下，你就必须这么做，或者如果你这么做了，是有益的。

4.1 关于 Autorelease 池

Autorelease 池是 NSAutorelease 类的一个实例，它是得到了 autorelease 消息的对象的容器。在 autorelease 池被 dealloc 的时候，它自己会给容纳的所有对象发送 release 消息。一个对象可以被多次放到同一个 autorelease 池，每一次放入（发送 autorelease 消息）都会造成将来收到一次 release。

多个 autorelease 池之间的关系，虽然通常的描述是“嵌套关系”，实际上它们是按照栈（stack）的方式工作的（译者：即类似于后进先出的队列）。当一个新的 autorelease 池创建后，它就位于这个栈的最顶端。池被 dealloc 的时候，就从栈中删除。当对象收到 autorelease 消息时，实际上它会被放到“这个线程”“当时”位于栈的最顶端的那个池中（译者：由此可以推定，每个线程都有一个私有的 autorelease 池的栈）。

Cocoa 希望程序中长期存在一个 autorelease 池。如果池不存在，autorelease 的对象就无从 release 了，从而造成内存泄露。当程序中没有 autorelease 池，你的程序还给对象发送 autorelease 消息，这是 Cocoa 会发出一个错误日志。AppKit 和 UIKit 框架自动在每个消息循环的开始都创建一个池（比如鼠标按下事件、触摸事件）并在结尾处销毁这个池。正因为如此，你实际上不需要创建 autorelease 池，甚至不需要知道创建 autorelease 池的代码如何写。下面三种情形下，你却应该使用你自己的 autorelease 池：

1. 如果你写的程序，不是基于 UI Framwork。例如你写的是一个基于命令行的程序。

2. 如果你程序中的一个循环，在循环体中创建了大量的临时对象。

你可以在循环体内部新建一个 autorelease 池，并在一次循环结束时销毁这些临时对象。这样可以减少你的程序对内存的占用峰值。

3. 如果你发起了一个 secondary 线程（译者：main 线程之外的线程）。这时你“必须”在线

程的最初执行代码中创建 autorelease 池，否则你的程序就内存泄露了。（参看“Autorelease 池和线程”）

通常我们用 alloc 和 init 来新建一个 NSAutoreleasePool 对象，用 drain 消息来销毁这个池。如果你发送 autorelease 或者 retain 消息给一个 autorelease 池对象，就会出现程序异常。如果要理解 release 和 drain 之间的区别，请参看“垃圾内存回收”。 Autorelease 池必须在其所“诞生”的上下文环境（比如对方法或函数的调用处，或者循环体的内部）中进行 drain。

Autorelease 池必须以 inline 的方式使用，你永远不需要把一个这样的池作为对象的成员变量来处理。

4.2 使用本地 Autorelease 池来减少内存占用峰值

许多程序所使用的临时对象都是 autorelease 的，因此这些对象在池被销毁前是占用内存的。要想减少对内存的占用峰值，就应该使用本地的 autorelease 池。当池被销毁时，那些临时对象都将被 release，进而系统占用内存情况得以改善。

下面的例子，是在 for 循环中使用本地 autorelease 池：

```
NSArray *urls = <# An array of file URLs #>;
for (NSURL *url in urls) {
    NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];
    NSError *error = nil;
    NSString *fileContents = [[[NSString alloc] initWithContentsOfURL:url
                                        encoding:NSUTF8StringEncoding
error:&error] autorelease];
    /* Process the string, creating and autoreleasing more objects. */
    [loopPool drain];
}
```

这个 for 循环每次处理一个文件。在循环体的开始，创建了一个池，结束时销毁了这个池。那么任何收到了 autorelease 消息的对象（本例中的 fileContents）都会被存放于池（loopPool）中。当池在单次循环结束时，进行销毁，这些对象也就 release 了。

在 autorelease 池已经 dealloc 之后，那些曾经收到 autorelease 消息对象，只能被视为失效，而不要再给他们发消息，或者把他们作为返回值进行返回。如果你必须在 autorelease 之后还要使用某个临时对象，你可以先发一个 retain 消息，然后等到这时的池已经调用了 drain 之后，再发送 autorelease 消息。示例如下：

```
- (id)findMatchingObject:(id)anObject {
    id match = nil;
    while (match == nil) {
        NSAutoreleasePool *subPool = [[NSAutoreleasePool alloc] init];
        /* Do a search that creates a lot of temporary objects. */
        match = [self expensiveSearchForObject:anObject];
        if (match != nil) {
            [match retain]; /* Keep match around. */
        }
        [subPool drain];
    }
    return [match autorelease];   /* Let match go and return it. */
}
```

在 subPool 有效的时候，我们给 match 对象发送了 retain 消息。在 subPool 被 drain 之后，我们又给 match 发送了 autorelease 消息。这样做的结果，就是 match 没有进入 subPool 池，而是进入了 subPool 的更早一个入栈的池。这样实际上是延长了 match 对象的生命周期，使得 match 可以在循环体之外还能接收消息，还使得 match 可以作为返回值来返回给 findMatchingObject 的调用者。

4.3 Autorelease 池和线程

Cocoa 程序的每一个线程都维护着一个自己的 NSAutorelease 对象的栈。当线程结束的时候，这些池对象就会被 release。如果你写的程序仅仅是一个基于 Foundation 的程序，又或者你 detach 一个线程（译者：关于 detached thread，请参考 Threading Programming Guide），你需要新建一个你自己的 autorelease 池。

如果你的程序是一个要长期运行的程序，可能会产生大量的临时对象，这是你必须周期性地销毁、新建 autorelease 池（Kit 在主线程中就是这么做的），否则 autorelease 对象就会累积并吃掉大量内存。如果你 detached 线程不调用 Cocoa，你就不必新建 autorelease 池。

注意：除非是 Cocoa 运行于多线程模式，否则如果你使用 POSIX 线程 API 来启动一个 secondary 线程，而不是使用 NSThread，你是不能使用 Cocoa 的，当然也就不能使用 NSAutorelease 池。Cocoa 只有在 detach 了它第一个 NSThread 对象之后，才能进入多线程模式。为了在 secondary POSIX 线程中使用 Cocoa，你的程序首先要做的是 detach 至少 1 个 NSThread，然后立刻结束这个线程。你可以用 NSThread 的 isMultiThreaded 方法来检测 Cocoa 是否处于多线

程模式。

4.4 Autorelease 池的作用域（Scope）和嵌套

Autorelease 池常常被称为“嵌套”的。但实际上也可以把这些嵌套的池理解为在一个栈（stack）中：嵌套在最里面的池，位于栈的最顶端。每个线程都有一个 autorelease 池的栈，而你每新建一个池，它都会被放置在这个栈的最顶端。当一个对象收到了 autorelease 消息时，又或者这个对象作为 addObject:方法的参数传递的时候，这个对象被放到了当时这个栈中最顶端的那个池中。

一个 autorelease 池的“作用域”（Scope）实际是由它在栈中的位置所决定的。最顶上的池，就是当下存放 autorelease 对象的池。如果这是新建了一个新的池，原有的池就离开了 Scope，知道这个新池被 drain 后，原有的池再次回到最顶端，进入 scope。Drain 后的池，就永远不再 Scope 了。

如果你 drain 一个池，但是这个池却不在栈顶，那么栈内位于它上面的所有池就都 drain 了（这意味着所有他们容纳的对象，都收到 release 消息）。如果你不小心忘记了调用一个池的 drain，那么从嵌套结构上看，更外一层的池在 drain 的时候，会销毁这个池。

这种特性，对于出现程序运行异常的情形是有用的。当异常出现，系统立刻从当前执行的代码中跳出，当前现场有效的池被 drain。然而一旦这个池不是栈顶的那个池，那么所有它上面的池都被 drain。最终，比这个池更早的那个池成为栈顶。这种行为机制，是的 Exception Handler 不必去处理异常发生所在现场 autorelease 对象的 release 工作。对 Exception Handler 而言，既不希望也不必要去给 autorelease 池发送 release---Unless the heandler is re-raising the exception。（译者：后半句理解不了啊）。

4.5 内存垃圾回收

尽管内存垃圾回收系统（Garbage Collection Programming Guide 中讲述 ）并不使用 autorelease 池，但如果你开发的是一个混合框架（既用到了内存垃圾回收，还用到了引用计数），那么 autorelease 池为垃圾内存回收者提供了线索。当 autorelease 池进行 release 的时候，恰恰是提示垃圾内存回收者现在需要回收内存。

在垃圾内存回收的环境中，release 实际上什么都不做。正因为如此，NSAutoreleasePool 才

提供了一个 drain 方法—这个方法在引用计数环境下等同于 release，但是在垃圾内存回收环境下，触发了内存回收行为（前提是此时内存新分配的数量，超过了阈值）。所以，如果要销毁 autorelease 池，你应该用 drain，而不是 release。

[The End.]