Advanced Memory Management Programming Guide

高级内存管理编程指南

https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple\_ref/doc/uid/10000011-SW1

1. About Memory Management 关于内存管理

2. Memory Management Policy 内存管理策略

2.1 Base Memory Management Rules 基础内存管理规则

2.1.1 A Simple Example 一个简单的例子

2.1.2 Use autorelease to Send a Deferred by Reference 使用自动释放发出延迟释放请求

2.1.3 You Don’t Own Objects Returned by Reference 你不会通过引用持有返回的对象

2.2 Implement dealloc to Relinquish Ownership of Objects 实现dealloc方法以放弃对象的所有权

2.3 Core Foundation Uses Similar but Different Rules 使用Core Foundation中类似但不同的规则

3. Practical Memory Management 实践内存管理

3.1 Use Accessor Methods to Make Memory Management Easier 使用Accessor方法让内存管理更容易

3.1.1 Use Accessor Methods to Set Property Values 使用Accessor方法设置属性值

3.1.2 Don’t Use Accessor Methods in Initializer and dealloc 不要在初始化方法和dealloc方法中使用Accessor方法

3.2 Use Weak References to Avoid Retain Cycles 使用弱引用避免循环持有

3.3 Avoid Causing Deallocation of Objects You’re Using 避免你正在使用的对象被释放

3.4 Don’t Use dealloc to Manage Scarce Resources 不要使用dealloc方法管理稀有资源

3.5 Collections Own the Objects They Contain 集合会拥有它们包含的对象

3.6 Ownership Policy Is Implemented Using Retain Counts 通过引用计数实现所有权策略

4. Using Autorelease Pool Blocks 使用自动释放池block

4.1 About Autorelease Pool Blocks 关于自动释放池block

4.2 Use Local Autorelease Pool Blocks to Reduce Peak Memory Footprint 使用本地自动释放池block减少内存占用峰值

5. 修订历史

参考Object-C类

NSAutoreleasePool

NSCopying

NSMutableCopying

NSObject

NSObject

相关文档

Foundation Functions Reference

Memory Management Programming Guide for Core Foundation

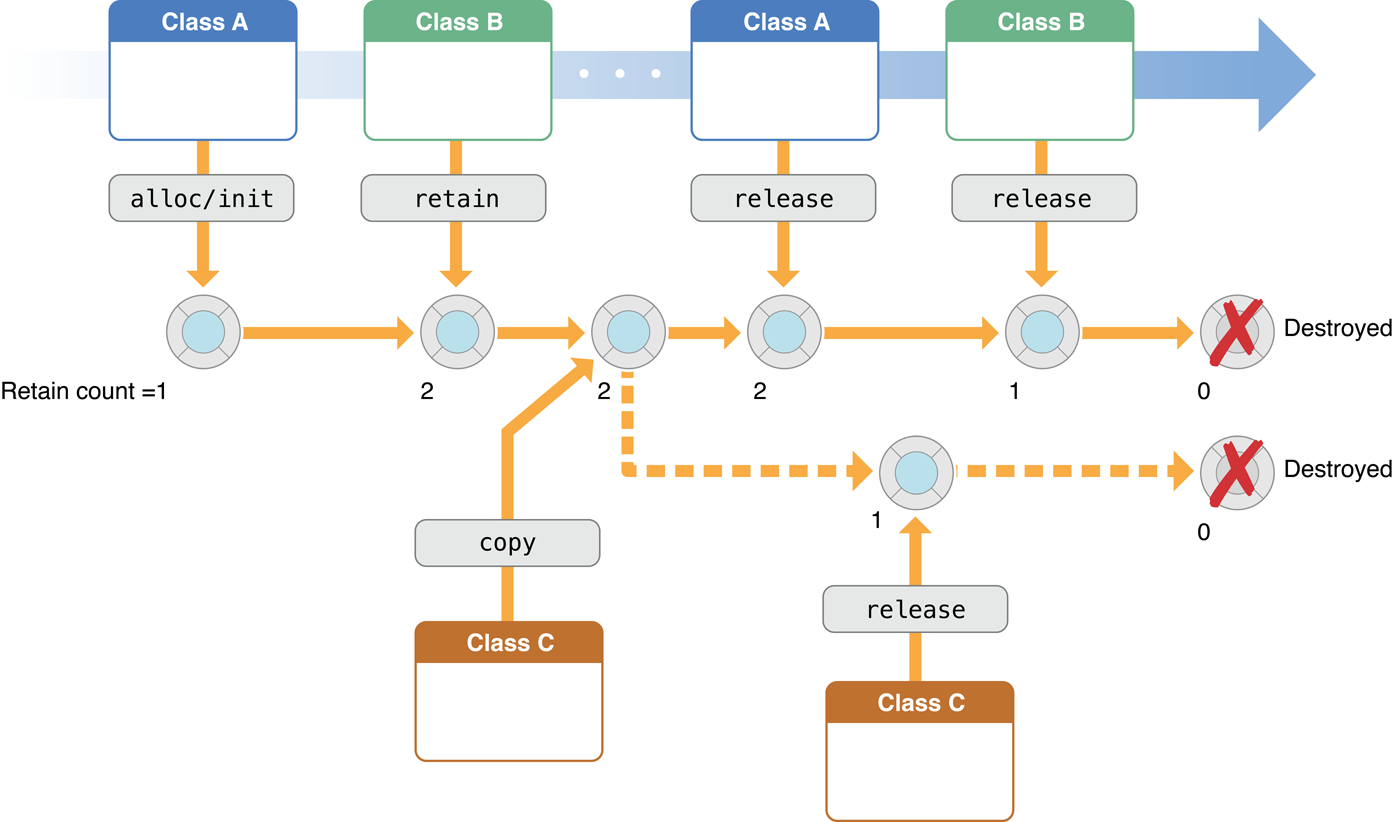
About Memory Management

Application memory management is the process of allocating memory during your program’s runtime, using it, and freeing it when you are done with it. A well-written program uses as little memory as possible. In Objective-C, it can also be seen as a way of distributing ownership of limited memory resources among many pieces of data and code. When you have finished working through this guide, you will have the knowledge you need to manage your application’s memory by explicitly managing the life cycle of objects and freeing them when they are no longer needed.

应用程序的内存管理是指，在程序运行时分配内存、使用内存、和使用后释放内存的过程。一个编写良好的程序会尽可能少使用内存。在objective - c中,它还可以被看成在许多块数据和代码分配有限内存资源的所有权的途径。当你已经通过这个向导完成了工作，你将拥有管理应用程序的内存所需的知识，可以显式的管理对象的生命周期并在不需要的时候释放它们。

Although memory management is typically considered at the level of an individual object, your goal is actually to manage object graphs. You want to make sure that you have no more objects in memory than you actually need.

尽管内存管理通常被认为是在单个对象级别的行为，但你的目标实际上是管理整个对象图。你要确保内存中不会有比你实际需要更多的对象。



## At a Glance

Objective-C provides two methods of application memory management.

Objective-C提供了两种应用程序内存管理方法。

1. In the method described in this guide, referred to as “manual retain-release” or MRR, you explicitly manage memory by keeping track of objects you own. This is implemented using a model, known as reference counting, that the Foundation class [NSObject](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject) provides in conjunction with the runtime environment.
2. In Automatic Reference Counting, or ARC, the system uses the same reference counting system as MRR, but it inserts the appropriate memory management method calls for you at compile-time. You are strongly encouraged to use ARC for new projects. If you use ARC, there is typically no need to understand the underlying implementation described in this document, although it may in some situations be helpful. For more about ARC, see [Transitioning to ARC Release Notes](https://developer.apple.com/library/ios/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226).
3. “手动保留释放（manual retain-release）”，或称MRR。你可以通过追踪你拥有的对象显式的管理内存。该方法通过使用一个被称为引用计数的模型来实现，基础类conjunction结合运行时环境提供了这个模型。
4. “自动引用计数（Automatic Reference Counting）”，或称ARC。系统使用了与MRR相同的引用计数系统，但是在编译时插入了适当的内存管理方法调用。强烈建议在新工程中使用ARC。如果使用ARC，通常都不需要了解本文所描述的底层实现，尽管这些知识在某些情况下是有益的。关于ARC的更多知识，请参考[Transitioning to ARC Release Notes](https://developer.apple.com/library/ios/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226)。

## Good Practices Prevent Memory-Related Problems

良好的实践可以避免内存相关的问题

There are two main kinds of problem that result from incorrect memory management:

* Freeing or overwriting data that is still in use

This causes memory corruption, and typically results in your application crashing, or worse, corrupted user data.

* Not freeing data that is no longer in use causes memory leaks

A memory leak is where allocated memory is not freed, even though it is never used again. Leaks cause your application to use ever-increasing amounts of memory, which in turn may result in poor system performance or your application being terminated.

不正确的内存管理将导致两类主要的问题：

* 释放或重写仍在使用的数据

这将导致内存损坏，通常在程序中的结果就是崩溃，或者更糟的是损坏用户数据。

* 没有释放不再使用的数据导致内存泄露

内存泄露是指已经申请的内存即使永不再使用也没有释放。泄露会导致应用使用的内存数量持续增加，从而导致系统性能变差或者应用程序被终止。

Thinking about memory management from the perspective of reference counting, however, is frequently counterproductive, because you tend to consider memory management in terms of the implementation details rather than in terms of your actual goals. Instead, you should think of memory management from the perspective of object ownership and object graphs.

从引用计数的角度思考内存管理，往往适得其反，因为人们趋向于在实现细节方面考虑内存管理而不是实际目的。与之相反的，你应该从对象所有权和对象图的角度去思考内存管理。

Cocoa uses a straightforward naming convention to indicate when you own an object returned by a method.

See [Memory Management Policy](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html#//apple_ref/doc/uid/20000994-BAJHFBGH).

当你通过一个方法拥有了一个对象时，Cocoa会使用直接了当的命名规则来指出。

请参考 [Memory Management Policy](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html#//apple_ref/doc/uid/20000994-BAJHFBGH).

Although the basic policy is straightforward, there are some practical steps you can take to make managing memory easier, and to help to ensure your program remains reliable and robust while at the same time minimizing its resource requirements.

See [Practical Memory Management](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-SW1).

尽管基础策略已经直接说了，但你还可以采用一些实践步骤使得管理内存更加简单，并且有助于确保你的程序更加可靠和健壮，同时最小化其资源需求。

请参考 [Practical Memory Management](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-SW1).

Autorelease pool blocks provide a mechanism whereby you can send an object a “deferred” release message. This is useful in situations where you want to relinquish ownership of an object, but want to avoid the possibility of it being deallocated immediately (such as when you return an object from a method). There are occasions when you might use your own autorelease pool blocks.

See [Using Autorelease Pool Blocks](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmAutoreleasePools.html#//apple_ref/doc/uid/20000047-CJBFBEDI).

自动释放池block提供了一种机制，在这个机制里你可以向一个对象发送“延迟”释放消息。当你想要放弃一个对象的所有权但又不想立即释放它的时候（例如在一个方法中返回一个对象），这非常有用。有很多场合你可能会使用到自动释放池block。

请参考 [Using Autorelease Pool Blocks](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmAutoreleasePools.html#//apple_ref/doc/uid/20000047-CJBFBEDI).

## Use Analysis Tools to Debug Memory Problems

To identify problems with your code at compile time, you can [use the Clang Static Analyzer](https://developer.apple.com/library/ios/recipes/xcode_help-source_editor/chapters/Analyze.html#//apple_ref/doc/uid/TP40009975-CH4) that is built into Xcode.

想要在编译时定位代码中的问题，你可以使用Xcode中集成的Clang静态分析器。

If memory management problems do nevertheless arise, there are other tools and techniques you can use to identify and diagnose the issues.

如果内存管理问题仍然出现，还可以使用其他的工具和技术定位和诊断问题。

Many of the tools and techniques are described in Technical Note TN2239, [iOS Debugging Magic](https://developer.apple.com/library/ios/technotes/tn2239/_index.html#//apple_ref/doc/uid/DTS40010638), in particular the use of NSZombie to help find over-released object.

在Technical Note TN2239, [iOS Debugging Magic](https://developer.apple.com/library/ios/technotes/tn2239/_index.html#//apple_ref/doc/uid/DTS40010638)一章中介绍了许多工具盒技术，尤其是NSZombie的使用可以帮助找到过度释放的对象。

You can use Instruments to track reference counting events and look for memory leaks. See [Collecting Data on Your App](https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/TheInstrumentsWorkflow.html#//apple_ref/doc/uid/TP40004652-CH5).

你可以使用Instruments跟踪引用计数事件并查看内存泄露。参考[Collecting Data on Your App](https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/TheInstrumentsWorkflow.html#//apple_ref/doc/uid/TP40004652-CH5)。

**注1：Object graph**

In an object-oriented program, groups of objects form a network through their relationships with each other—either through a direct reference to another object or through a chain of intermediate references. These groups of objects are referred to as object graphs. Object graphs may be small or large, simple or complex. An array object that contains a single string object represents a small, simple object graph. A group of objects containing an application object, with references to the windows, menus and their views, and other supporting objects, may represent a large, complex object graph.

Sometimes you may want to convert an object graph—usually just a section of the full object graph in the application—into a form that can be saved to a file or transmitted to another process or machine and then reconstructed. This process is known as archiving.

Some object graphs may be incomplete—these are often referred to as partial object graphs. Partial object graphs have placeholder objects that represent the boundaries of the graph and that may be filled in at a later stage. An example is a nib file that includes a placeholder for the File’s Owner.

Memory Management Policy

The basic model used for memory management in a reference-counted environment is provided by a combination of methods defined in the [NSObject](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intf/NSObject) protocol and a standard method naming convention. The NSObject class also defines a method, dealloc, that is invoked automatically when an object is deallocated. This article describes all the basic rules you need to know to manage memory correctly in a Cocoa program, and provides some examples of correct usage.

在引用计数环境中用于内存管理的基础模型，由定义在NSObject协议（注2）中的方法和标准方法命名规范相结合而来。NSObject类也定义了dealloc方法，当对象被销毁的时候该方法会自动调用。本文介绍了在Cocoa编程中正确的管理内存需要知道的所有基础规则，并提供了一些正确使用的例子。

## Basic Memory Management Rules

The memory management model is based on object ownership. Any object may have one or more owners. As long as an object has at least one owner, it continues to exist. If an object has no owners, the runtime system destroys it automatically. To make sure it is clear when you own an object and when you do not, Cocoa sets the following policy:

内存管理模型的基础是对象的所有权。每个对象可能有一个或多个所有者。只要一个对象有了至少一个所有者，它就继续存在。如果一个对象没有任何所有者，运行时系统会自动销毁它。为了确保我们清楚什么时候拥有了一个对象而什么时候又没有，Cocoa设定了以下规则：

* **You own any object you create**

You create an object using a method whose name begins with “alloc”, “new”, “copy”, or “mutableCopy” (for example, [alloc](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/clm/NSObject/alloc), newObject, or [mutableCopy](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/instm/NSObject/mutableCopy)).

* **你拥有你创建的任何对象**

你可以使用以“alloc”、“new”、“copy”或者“mutableCopy”开头的方法创建对象。（例如alloc，newObject或mutableCopy）

* **You can take ownership of an object using retain**

A received object is normally guaranteed to remain valid within the method it was received in, and that method may also safely return the object to its invoker. You use retain in two situations: (1) In the implementation of an accessor method or an init method, to take ownership of an object you want to store as a property value; and (2) To prevent an object from being invalidated as a side-effect of some other operation (as explained in [Avoid Causing Deallocation of Objects You’re Using](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/20000043-1000922)).

* 你可以使用retain获得一个对象的所有权。

返回对象通常保证在收到的方法里仍然有效，并且该方法也可以安全的返回给它的调用者。在下面两种情况下你需要使用retain：

1. 在accessor方法或init方法的实现中，获取你想要存为属性值的对象的所有权；
2. 放置一个对象由于某些其他操作的副作用被置为无效的（在[Avoid Causing Deallocation of Objects You’re Using](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/20000043-1000922)中会详细解释）

* **When you no longer need it, you must relinquish ownership of an object you own**

You relinquish ownership of an object by sending it a [release](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/release) message or an [autorelease](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/autorelease) message. In Cocoa terminology, relinquishing ownership of an object is therefore typically referred to as “releasing” an object.

* 当你不再需要一个你拥有的对象时，你必须放弃其所有权

通过向一个对象发送release或autorelease消息可以放弃对它的所有权。因此，在Cocoa术语中，放弃一个对象的所有权通常也被称为“释放”一个对象。

* **You must not relinquish ownership of an object you do not own**

This is just corollary of the previous policy rules, stated explicitly.

* 你不能放弃不是你拥有的对象的所有权

这只是前一条规则的推论，只是明确的指出来。

## A Simple Example

To illustrate the policy, consider the following code fragment:

为了阐明这些策略，请看以下代码片段：

|  |
| --- |
| { |
| Person \*aPerson = [[Person alloc] init]; |
| // ... |
| NSString \*name = aPerson.fullName; |
| // ... |
| [aPerson release]; |
| } |

The Person object is created using the alloc method, so it is subsequently sent a release message when it is no longer needed. The person’s name is not retrieved using any of the owning methods, so it is not sent a release message. Notice, though, that the example uses release rather than autorelease.

Person对象是使用alloc方法创建的，所以随后不再需要它时向它发送了release消息。Person的name取得时没有被任何所有权方法，所以不用向它发送release消息。注意，本例中使用的是release而不是autorelease。

## Use autorelease to Send a Deferred release

You use autorelease when you need to send a deferred release message—typically when returning an object from a method. For example, you could implement the fullName method like this:

当你需要发送一个延迟release消息时你可以使用autorelease——通常是在从方法中返回一个对象的时候。例如，你可以像这样实现fullName方法：

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

You own the string returned by alloc. To abide by the memory management rules, you must relinquish ownership of the string before you lose the reference to it. If you use release, however, the string will be deallocated before it is returned (and the method would return an invalid object). Using autorelease, you signify that you want to relinquish ownership, but you allow the caller of the method to use the returned string before it is deallocated.

你拥有了通过alloc返回的string。为了遵守内存管理规则，你必须在你失去对它的引用之前放弃对它的所有权。如果你使用release，这个string就会在返回之前被销毁（这个方法也会返回一个无效的对象）。使用autorelease，你既表示了将要放弃所有权，又允许方法的调用者在返回的string被销毁前使用它。

You could also implement the fullName method like this:

你也可以像这样实现fullName方法：

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

Following the basic rules, you don’t own the string returned by stringWithFormat:, so you can safely return the string from the method.

按照基本规则，你不会拥有stringWithFormat:方法返回的string对象，所以你也可以安全的从该方法中返回string。

By way of contrast, the following implementation is wrong:

作为对照，下面的实现方式是错误的：

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

According to the naming convention, there is nothing to denote that the caller of the fullName method owns the returned string. The caller therefore has no reason to release the returned string, and it will thus be leaked.

根据命名规则，没有任何地方说明了fullName方法的调用者拥有返回的string。因此调用者没有任何理由释放返回的string，这样它将被泄露。

## You Don’t Own Objects Returned by Reference

Some methods in Cocoa specify that an object is returned by reference (that is, they take an argument of type ClassName \*\* or id \*). A common pattern is to use an NSError object that contains information about an error if one occurs, as illustrated by [initWithContentsOfURL:options:error:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSData_Class/index.html#//apple_ref/occ/instm/NSData/initWithContentsOfURL:options:error:) (NSData) and [initWithContentsOfFile:encoding:error:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/index.html#//apple_ref/occ/instm/NSString/initWithContentsOfFile:encoding:error:) (NSString).

Cocoa中的某些方法指明了对象是通过引用返回的（比如，参数类型是ClassName \*\*或id \*的方法）。常见的模式是在发生错误时使用一个包含了错误信息的NSError对象，例如[initWithContentsOfURL:options:error:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSData_Class/index.html#//apple_ref/occ/instm/NSData/initWithContentsOfURL:options:error:) (NSData)方法和[initWithContentsOfFile:encoding:error:](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/index.html#//apple_ref/occ/instm/NSString/initWithContentsOfFile:encoding:error:) (NSString)方法。

In these cases, the same rules apply as have already been described. When you invoke any of these methods, you do not create the NSError object, so you do not own it. There is therefore no need to release it, as illustrated in this example:

在这些情况下，前面介绍的规则同样适用。当你调用任何这样的方法，你没有创建NSError对象，所以你不拥有它。因此你不需要释放它。如下面的例子所示：

|  |
| --- |
| NSString \*fileName = <#Get a file name#>; |
| NSError \*error; |
| NSString \*string = [[NSString alloc] initWithContentsOfFile:filename |
| encoding:NSUTF8StringEncoding error:&error]; |
| if (string == nil) { |
| // Deal with error... |
| } |
| // ... |
| [string release]; |

## Implement dealloc to Relinquish Ownership of Objects

The NSObject class defines a method, [dealloc](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/instm/NSObject/dealloc), that is invoked automatically when an object has no owners and its memory is reclaimed—in Cocoa terminology it is “freed” or “deallocated.”. The role of the dealloc method is to free the object's own memory, and to dispose of any resources it holds, including ownership of any object instance variables.

NSObject类定义了一个方法，dealloc，当对象没有拥有者并且其内存被回收时会自动调用——在Cocoa术语中称为它被“释放”或“销毁”。Dealloc方法的角色是释放对象自己的内存，并且清除它持有的所有资源，包括所有对象实例变量的所有权。

The following example illustrates how you might implement a dealloc method for a Person class:

下面的例子展示了应该如何实现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 |
| // ... |
| - (void)dealloc |
| [\_firstName release]; |
| [\_lastName release]; |
| [super dealloc]; |
| } |
| @end |

**Important:**

Never invoke another object’s dealloc method directly.

You must invoke the superclass’s implementation at the end of your implementation.

You should not tie management of system resources to object lifetimes; see [Don’t Use dealloc to Manage Scarce Resources](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-SW13).

When an application terminates, objects may not be sent a dealloc message. Because the process’s memory is automatically cleared on exit, it is more efficient simply to allow the operating system to clean up resources than to invoke all the memory management methods.

注意：

永远不要直接调用另一个对象的dealloc方法。

你必须在你的实现末尾调用父类的实现。

你不应该将系统资源的管理与对象生命周期绑在一起；参考[Don’t Use dealloc to Manage Scarce Resources](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-SW13)。

当应用被终止时，对象可能不会收到dealloc消息。因为进程的内存在退出时自动清理了，允许操作系统清理资源比调用所有的内存管理方法更加高效。

## Core Foundation Uses Similar but Different Rules

There are similar memory management rules for Core Foundation objects (see [Memory Management Programming Guide for Core Foundation](https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/CFMemoryMgmt.html#//apple_ref/doc/uid/10000127i)). The naming conventions for Cocoa and Core Foundation, however, are different. In particular, Core Foundation’s Create Rule (see [The Create Rule](https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/Concepts/Ownership.html#//apple_ref/doc/uid/20001148-103029)) does not apply to methods that return Objective-C objects. For example, in the following code fragment, you are not responsible for relinquishing ownership of myInstance:

Core Foundation对象有一些类似的内存管理规则（参考[Memory Management Programming Guide for Core Foundation](https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/CFMemoryMgmt.html#//apple_ref/doc/uid/10000127i)）。但是Cocoa与Core Foundation的命名规范是不同的。特别是Core Foundation的创建规则（参考[The Create Rule](https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/Concepts/Ownership.html#//apple_ref/doc/uid/20001148-103029)），它不接受返回返回Objective-C对象的方法。例如，在下面的代码片段中，你并不负责放弃myInstance的所有权：

|  |
| --- |
| MyClass \*myInstance = [MyClass createInstance]; |

Practical Memory Management

Although the fundamental concepts described in [Memory Management Policy](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html#//apple_ref/doc/uid/20000994-BAJHFBGH) are straightforward, there are some practical steps you can take to make managing memory easier, and to help to ensure your program remains reliable and robust while at the same time minimizing its resource requirements.

尽管[Memory Management Policy](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html#//apple_ref/doc/uid/20000994-BAJHFBGH)中直接介绍了基本的概念，但你还可以采用一些实践步骤使得管理内存更加简单，并且有助于确保你的程序更加可靠和健壮，同时最小化其资源需求。

## Use Accessor Methods to Make Memory Management Easier

If your class has a property that is an object, you must make sure that any object that is set as the value is not deallocated while you’re using it. You must therefore claim ownership of the object when it is set. You must also make sure you then relinquish ownership of any currently-held value.

如果你的类有一个属性是对象，你就必须确保设置成它的值的任何对象在你使用时没有被销毁。因此在设置时你必须声明该对象的所有权。你也必须确保你最后放弃了所有当前持有的值的所有权。

Sometimes it might seem tedious or pedantic, but if you use accessor methods consistently, the chances of having problems with memory management decrease considerably. If you are using retain and release on instance variables throughout your code, you are almost certainly doing the wrong thing.

尽管有时显得多余或者过于迂腐，但是如果你坚持使用accessor方法，你遇到内存管理问题的机会将会大大减少。如果你整个代码自始至终都在使用retain和release处理实例变量，那你几乎肯定做了错误的事情。

Consider a Counter object whose count you want to set.

考虑有一个计数器对象，你可以设置其计数。

|  |
| --- |
| @interface Counter : NSObject |
| @property (nonatomic, retain) NSNumber \*count; |
| @end; |

The property declares two accessor methods. Typically, you should ask the compiler to synthesize the methods; however, it’s instructive to see how they might be implemented.

该属性（注3）声明了两个accessor方法。通常，都是编译器合成这两个方法；不过，看看它是如何实现的会非常有益。

In the “get” accessor, you just return the synthesized instance variable, so there is no need for retain or release:

在“get”方法中，只是返回了合成实例变量，所以不需要retain或release：

|  |
| --- |
| - (NSNumber \*)count { |
| return \_count; |
| } |

In the “set” method, if everyone else is playing by the same rules you have to assume the new count may be disposed of at any time so you have to take ownership of the object—by sending it a retain message—to ensure it won’t be. You must also relinquish ownership of the old count object here by sending it a release message. (Sending a message to nil is allowed in Objective-C, so the implementation will still work if \_count hasn’t yet been set.) You must send this after [newCount retain] in case the two are the same object—you don’t want to inadvertently cause it to be deallocated.

在“set”方法中，如果其他人按照相同的规则操作，你就不得不考虑新的count对象可能在任何时候被清理，所以你不得不获取该对象的所有权——通过向其发送retain消息——以确保其不被清理。你也必须在这里放弃旧的count对象的所有权，需要向它发送一个release消息。（在Objective-C中允许向nil发送消息，所以该实现在\_count从未设置时也仍然可以工作。）你必须在[newCount retain]之后发送这个release消息，如果这两个count是同一个对象——你一定不希望无意中将它销毁吧。

|  |
| --- |
| - (void)setCount:(NSNumber \*)newCount { |
| [newCount retain]; |
| [\_count release]; |
| // Make the new assignment. |
| \_count = newCount; |
| } |

## Use Accessor Methods to Set Property Values

Suppose you want to implement a method to reset the counter. You have a couple of choices. The first implementation creates the NSNumber instance with alloc, so you balance that with a release.

假如你想要实现一个方法重置计数器。你有两种选择。第一种实现方法用alloc创建了一个NSNumber实例，因此你需要相应的用release取得平衡。

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

The second uses a convenience constructor to create a new NSNumber object. There is therefore no need for retain or release messages

第二种实现方法使用了一个便捷构造器创建新的NSNumber对象。因此不需要retain或者release消息。

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

Note that both use the set accessor method.

The following will almost certainly work correctly for simple cases, but as tempting as it may be to eschew accessor methods, doing so will almost certainly lead to a mistake at some stage (for example, when you forget to retain or release, or if the memory management semantics for the instance variable change).

注意上述两种方法都使用了set accessor方法。

接下来的方法在简单的情况下几乎肯定是正确执行的，但是尽管避开accessor方法是多么诱人，这么做在某些场景几乎肯定会导致错误（例如，当你忘记retain或者release时，或者如果这个实例变量的内存管理语义发生变化）。

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

Note also that if you are using key-value observing, then changing the variable in this way is not KVO compliant.

也要注意如果你使用键值观察，采用这种方法改动变量是不符合KVO的。

## Don’t Use Accessor Methods in Initializer Methods and dealloc

The only places you shouldn’t use accessor methods to set an instance variable are in initializer methods and dealloc. To initialize a counter object with a number object representing zero, you might implement an init method as follows:

唯一一个不应该使用accessor方法设置实例变量的地方是初始化方法和dealloc方法中。要初始化一个表示0的number的计数器对象，你应该像这样实现init方法：

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

To allow a counter to be initialized with a count other than zero, you might implement an initWithCount: method as follows:

要允许计数器被另一个数字而不是0来初始化，你应该实现initWithCount:方法如下：

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

Since the Counter class has an object instance variable, you must also implement a dealloc method. It should relinquish ownership of any instance variables by sending them a release message, and ultimately it should invoke super’s implementation:

因为Counter类有一个对象实例变量，你也必须实现dealloc方法。需要向所有实例变量的发送release消息放弃它们的所有权，最后需要调用父类的实现：

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

## Use Weak References to Avoid Retain Cycles

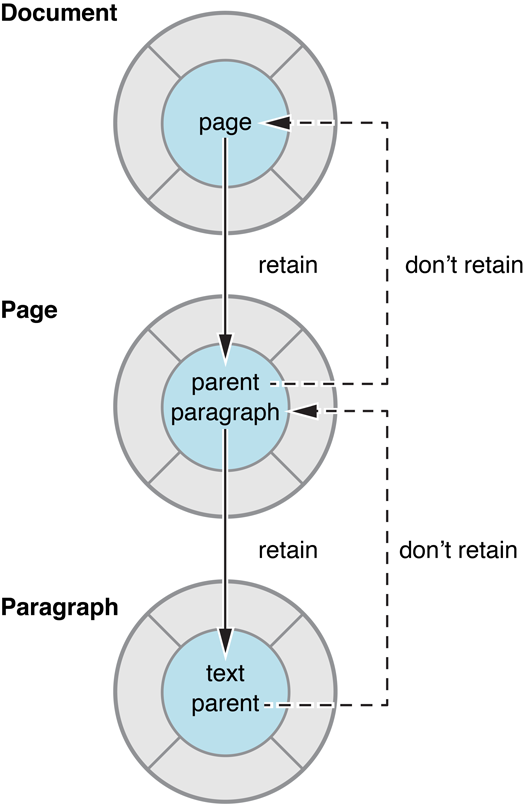
Retaining an object creates a strong reference to that object. An object cannot be deallocated until all of its strong references are released. A problem, known as a retain cycle, can therefore arise if two objects may have cyclical references—that is, they have a strong reference to each other (either directly, or through a chain of other objects each with a strong reference to the next leading back to the first).

Retain一个对象创建了一个对该对象的强引用。对象只有在它的所有强引用都被释放时才会被销毁。有这样一个问题，被称为循环持有，如果两个对象循环引用就会发生——也就是说他们互相发生强引用（直接互相引用，或者通过其他对象组成一个链条，每一个强引用下一个，最后一个引用第一个）。

The object relationships shown in Figure 1 illustrate a potential retain cycle. The Document object has a Page object for each page in the document. Each Page object has a property that keeps track of which document it is in. If the Document object has a strong reference to the Page object and the Page object has a strong reference to the Document object, neither object can ever be deallocated. The Document’s reference count cannot become zero until the Page object is released, and the Page object won’t be released until the Document object is deallocated.

图1展示了一个永久循环持有的对象关系。Document对象有一个Page对象表示文章里的每一页。每一个Page对象又有一个属性跟踪是在哪个文章里。如果Document对象强引用了Page对象，而Page对象也强引用了Document对象，两个对象都永远不会被销毁。Document对象的引用计数不会变成0除非Page对象被释放，而Page对象不会释放除非Document对象被销毁。

**Figure 1**  An illustration of cyclical references



The solution to the problem of retain cycles is to use weak references. A weak reference is a non-owning relationship where the source object does not retain the object to which it has a reference.

循环持有问题的解决办法就是使用弱引用。弱引用是一种非拥有的关系，在这种关系里源对象不持有被引用对象。

To keep the object graph intact, however, there must be strong references somewhere (if there were only weak references, then the pages and paragraphs might not have any owners and so would be deallocated). Cocoa establishes a convention, therefore, that a “parent” object should maintain strong references to its “children,” and that the children should have weak references to their parents.

为了保持对象图完成，我们必须在某些地方使用强引用（如果这些地方只有弱引用，Page和Paragraph可能没有任何所有者而被销毁）。因此，Cocoa建立了一个约定，“父对象”应该强引用持有其“子对象”，而“子对象”只能弱引用它们的“父对象”。

So, in Figure 1 the document object has a strong reference to (retains) its page objects, but the page object has a weak reference to (does not retain) the document object.

所以，在图1中，Document对象强引用（持有）它的Page对象，但是Page对象只是弱引用（不持有）Document对象。

Examples of weak references in Cocoa include, but are not restricted to, table data sources, outline view items, notification observers, and miscellaneous targets and delegates.

在Cocoa中弱引用的例子很多，包括但不限于table数据源、离线视图项目、通知观察者以及各种各样的target和delegate。

You need to be careful about sending messages to objects for which you hold only a weak reference. If you send a message to an object after it has been deallocated, your application will crash. You must have well-defined conditions for when the object is valid. In most cases, the weak-referenced object is aware of the other object’s weak reference to it, as is the case for circular references, and is responsible for notifying the other object when it deallocates. For example, when you register an object with a notification center, the notification center stores a weak reference to the object and sends messages to it when the appropriate notifications are posted. When the object is deallocated, you need to unregister it with the notification center to prevent the notification center from sending any further messages to the object, which no longer exists. Likewise, when a delegate object is deallocated, you need to remove the delegate link by sending a setDelegate: message with a [nil](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/c_ref/nil) argument to the other object. These messages are normally sent from the object’s dealloc method.

需要小心的向你仅仅是弱引用的对象发消息。如果你在一个对向被销毁后向它发送消息，你的应用将会崩溃。当对象可用时，你必须有明确定义的先决条件。在大多数情况下，被弱引用的对象知道其它对象弱引用了它，正如循环引用的例子中那样，它就有责任在它被销毁时通知其它对象。例如，当你在通知中心注册一个对象，通知中心就保存了一个该对象的弱引用，并且会在适当的通知发出时向它发消息。当这个对象被销毁时，你需要在通知中心进行反注册，以避免通知中心以后再向它发送任何消息，它已经不存在了。同样的，当delegate对象被释放时，你需要移除代理连接，想其它对象发送一个setDelegate:消息并附带nil参数。这些消息通常都在对象的dealloc方法中发出。

## Avoid Causing Deallocation of Objects You’re Using

Cocoa’s ownership policy specifies that received objects should typically remain valid throughout the scope of the calling method. It should also be possible to return a received object from the current scope without fear of it being released. It should not matter to your application that the getter method of an object returns a cached instance variable or a computed value. What matters is that the object remains valid for the time you need it.

Cocoa的所有权策略指定接收到的对象应该在贯穿整个调用方法的范围内都是有效的。应该也可以从当前的作用域中返回一个接收到的对象，而不用担心它被释放。对于应用程序来说，对象的getter方法返回缓存的实例变量或者返回计算的值都没有关系。重要的是在你需要它们的时候，它们仍然是有效的。

There are occasional exceptions to this rule, primarily falling into one of two categories.

按照这条规则，偶然会出现异常情况，主要是下面两种情形之一：

1. When an object is removed from one of the fundamental collection classes.

1. 当对象从一个基础集合类中移除的时候。

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

When an object is removed from one of the fundamental collection classes, it is sent a release (rather than autorelease) message. If the collection was the only owner of the removed object, the removed object (heisenObject in the example ) is then immediately deallocated.

当对象从一个基础集合类中移除的时候，它会收到一个release消息（而不是autorelease消息）。如果这个集合是被移除对象的唯一所有者，被移除对象（例子中的heisenObject）就会立即被销毁。

1. When a “parent object” is deallocated.

2. 当“父对象”被销毁时。

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

In some situations you retrieve an object from another object, and then directly or indirectly release the parent object. If releasing the parent causes it to be deallocated, and the parent was the only owner of the child, then the child (heisenObject in the example) will be deallocated at the same time (assuming that it is sent a release rather than an autorelease message in the parent’s dealloc method).

在某些情况下，你会从另一个对象取回一个对象，然后又直接或间接的释放了父对象。如果释放父对象导致父对象被销毁，而这个父对象又是这个子对象的唯一的所有者，那么这个子对象（例子中的heisenObject）就会同时被销毁（就好像在父对象的dealloc方法中向子对象发送了一个release消息而不是autorelease消息）。

To protect against these situations, you retain heisenObject upon receiving it and you release it when you have finished with it. For example:

为了避免这些情况，你应该在拿到heisenObject时就retain它，然后在用完以后release它。例如：

|  |
| --- |
| heisenObject = [[array objectAtIndex:n] retain]; |
| [array removeObjectAtIndex:n]; |
| // Use heisenObject... |
| [heisenObject release]; |

## Don’t Use dealloc to Manage Scarce Resources

You should typically not manage scarce resources such as file descriptors, network connections, and buffers or caches in a dealloc method. In particular, you should not design classes so that dealloc will be invoked when you think it will be invoked. Invocation of dealloc might be delayed or sidestepped, either because of a bug or because of application tear-down.

通常，不应该再dealloc方法中管理稀有资源，例如文件描述符、网络连接、缓冲区或者缓存。特别要注意，dealloc方法不会在你想要调用它时它就会被调用，不应该这样设计一个类。Dealloc的调用可能被延迟或者回避，可能是因为bug，也可能是因为应用被关闭。

Instead, if you have a class whose instances manage scarce resources, you should design your application such that you know when you no longer need the resources and can then tell the instance to “clean up” at that point. You would typically then release the instance, and dealloc would follow, but you will not suffer additional problems if it does not.

相反，你如果有一个管理稀有资源的类，你应该将你的应用设计成你知道你什么时候不再使用这些资源了，然后告知这个类的实例在那个时候就“清理”掉。通常是释放掉实例，接着就会调用dealloc方法，但是这样做的话，即使dealloc方法没有被调用也不会出现另外的问题。

Problems may arise if you try to piggy-back resource management on top of dealloc. For example:

如果你试图在dealloc顶部背负起资源管理，那问题可能会增加。例如：

1.Order dependencies on object graph tear-down.

The object graph tear-down mechanism is inherently non-ordered. Although you might typically expect—and get—a particular order, you are introducing fragility. If an object is unexpectedly autoreleased rather than released for example, the tear-down order may change, which may lead to unexpected results.

1. 对象图关闭的顺序依赖。

对象图关闭机制本质上是无序的。尽管你通常期望——并得到了——一个特殊的顺序，但你需要知道这个排序的脆弱性。如果一个对象出乎意料的自动释放而不是直接释放，那关闭顺序也可能改变，这就可能导致预料之外的结果了。

2.Non-reclamation of scarce resources.

Memory leaks are bugs that should be fixed, but they are generally not immediately fatal. If scarce resources are not released when you expect them to be released, however, you may run into more serious problems. If your application runs out of file descriptors, for example, the user may not be able to save data.

1. 未回收稀有资源

内存泄露通常不是立即致命的，但这类bug也应该被修正。如果稀有资源在你希望它们释放的时候没有被释放，你可能会走入更严重的问题中。例如，如果应用程序在文件描述符之外运行，用户可能无法保存数据。

3.Cleanup logic being executed on the wrong thread.

If an object is autoreleased at an unexpected time, it will be deallocated on whatever thread’s autorelease pool block it happens to be in. This can easily be fatal for resources that should only be touched from one thread.

1. 清理逻辑在错误的线程执行。

如果对象在意外的时间自动释放，它可能在任何线程的自动释放池block里被销毁。这对于只能在一个线程中接触的资源是极易致命的。

## Collections Own the Objects They Contain

When you add an object to a collection (such as an array, dictionary, or set), the collection takes ownership of it. The collection will relinquish ownership when the object is removed from the collection or when the collection is itself released. Thus, for example, if you want to create an array of numbers you might do either of the following:

当把对象添加到集合（例如Array、Dictionary或Set）中，集合就获得了它的所有权。当对象从集合中移除或者集合自己被释放时，集合都会放弃所有权。那么，举个例子说，如果你想要创建一个数字对象的数组，你可能像下面其中一种这样做：

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

In this case, you didn’t invoke alloc, so there’s no need to call release. There is no need to retain the new numbers (convenienceNumber), since the array will do so.

在这个案例中，你没有调用alloc，所以不需要调用release。不要要持有新的数字对象（convenienceNumber），因为数组持有了它们。

|  |
| --- |
| 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]; |
| } |

In this case, you do need to send allocedNumber a release message within the scope of the for loop to balance the alloc. Since the array retained the number when it was added by addObject:, it will not be deallocated while it’s in the array.

在这个案例中，你需要在for循环的作用域内向allocedNumber发送release消息以平衡alloc。因为数字在被addObject:方法添加到数组时被数组持有，所以当它在数组中时不会被销毁。

To understand this, put yourself in the position of the person who implemented the collection class. You want to make sure that no objects you’re given to look after disappear out from under you, so you send them a retain message as they’re passed in. If they’re removed, you have to send a balancing release message, and any remaining objects should be sent a release message during your own dealloc method.

要理解这些，请将自己放在集合类的实现者的位置。你想要确保没有任何交给你照看的对象在你的面前消失，因此你需要在它们传过来时就向它们发一个retain消息。如果它们被移除，你需要相应的发一个release消息；而且在你调用自己的dealloc方法时，要给所有还在的对象都发一个release消息。

## Ownership Policy Is Implemented Using Retain Counts

The ownership policy is implemented through reference counting—typically called “retain count” after the [retain](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/retain) method. Each object has a retain count.

所有权策略通过引用计数来实现——引用计数通常在retain方法后被称为“retain count”。每一个对象有一个retain count。

* When you create an object, it has a retain count of 1.
* When you send an object a retain message, its retain count is incremented by 1.
* When you send an object a [release](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/release) message, its retain count is decremented by 1.

When you send an object a [autorelease](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/autorelease) message, its retain count is decremented by 1 at the end of the current autorelease pool block.

* If an object’s retain count is reduced to zero, it is deallocated.
* 当创建一个对象时，它的retain count就是1。
* 当向一个对象发送retain消息，它的retain count加1。
* 当向一个对象发送release消息，它的retain count减1。

当向一个对象发送autorelease消息，它的retain count在当前自动释放池block的末尾减1。

* 如果对象的retain count被减为0，它就会被销毁。

**Important:** There should be no reason to explicitly ask an object what its retain count is (see [retainCount](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/retainCount)). The result is often misleading, as you may be unaware of what framework objects have retained an object in which you are interested. In debugging memory management issues, you should be concerned only with ensuring that your code adheres to the ownership rules.

注意：没必要明确的向对象查询它的retain count是多少（参考[retainCount](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/retainCount)）。这样做的结果通常会令人误入歧途，因为你可能不知道哪个框架对象持有了你关注的对象。在调试内存管理问题时，你关注的应该只是确保你的代码符合所有权规则。

Using Autorelease Pool Blocks

Autorelease pool blocks provide a mechanism whereby you can relinquish ownership of an object, but avoid the possibility of it being deallocated immediately (such as when you return an object from a method). Typically, you don’t need to create your own autorelease pool blocks, but there are some situations in which either you must or it is beneficial to do so

自动释放池block提供了一种机制，在这种机制例你可以放弃一个对象的所有权而避免它立即被销毁的可能性（例如当你从一个方法返回一个对象时）。通常你不需要创建你自己的自动释放池block，但是在某些情况下你必须这么做或者最好这么做。

## About Autorelease Pool Blocks

An autorelease pool block is marked using @autoreleasepool, as illustrated in the following example:

自动释放池使用@autoreleasepool标记，如下面的例子所阐述的：

|  |
| --- |
| @autoreleasepool { |
| // Code that creates autoreleased objects. |
| } |

At the end of the autorelease pool block, objects that received an autorelease message within the block are sent a release message—an object receives a release message for each time it was sent an autorelease message within the block.

在自动释放池block的末尾，向block里接收了autorelease消息的对象发送了一条release消息——在block里的对象每被发送一次autolease消息此时就会接到一次release消息。

Like any other code block, autorelease pool blocks can be nested:

就像所有其他的代码block一样，自动释放池block也可以嵌套：

|  |
| --- |
| @autoreleasepool { |
| // . . . |
| @autoreleasepool { |
| // . . . |
| } |
| . . . |
| } |

(You wouldn’t normally see code exactly as above; typically code within an autorelease pool block in one source file would invoke code in another source file that is contained within another autorelease pool block.) For a given autorelease message, the corresponding release message is sent at the end of the autorelease pool block in which the autorelease message was sent.

（你可能不常看到代码恰好像上面那样；常见的是一个源文件中的自动释放池block中的代码调用另一个源文件中的自动释放池中的代码。）对于一个给定的autorelease消息，相应的release消息在autorelease消息被发送的自动释放池block的末尾被发送。

Cocoa always expects code to be executed within an autorelease pool block, otherwise autoreleased objects do not get released and your application leaks memory. (If you send an autorelease message outside of an autorelease pool block, Cocoa logs a suitable error message.) The AppKit and UIKit frameworks process each event-loop iteration (such as a mouse down event or a tap) within an autorelease pool block. Therefore you typically do not have to create an autorelease pool block yourself, or even see the code that is used to create one. There are, however, three occasions when you might use your own autorelease pool blocks:

Cocoa总是希望代码在自动释放池block中执行，否则自动释放的对象不会被释放，应用就会泄露内存。（如果你在自动释放池block之外发送一个autorelease消息，Cocoa会将其作为一个错误消息记入日志。）AppKit和UIKit框架都在自动释放池block中处理每一个事件循环迭代（比如鼠标点击事件或触碰事件）。因此你通常不需要自己创建自动释放池block，甚至看不到用于创建的代码。然而，有三个场合你可能要使用自己的自动释放池block：

* If you are writing a program that is not based on a UI framework, such as a command-line tool.
* If you write a loop that creates many temporary objects.

You may use an autorelease pool block inside the loop to dispose of those objects before the next iteration. Using an autorelease pool block in the loop helps to reduce the maximum memory footprint of the application.

* If you spawn a secondary thread.

You must create your own autorelease pool block as soon as the thread begins executing; otherwise, your application will leak objects. (See Autorelease Pool Blocks and Threads for details.)

* 如果你正在写的程序不是基于UI框架的，例如写一个命令行工具。
* 如果你正在写一个创建了大量临时对象的循环。

你可以在循环里使用自动释放池block，以便在下一个迭代之前处理这些对象。在循环中使用自动释放池block有助于降低应用程序的最大内存占用。

* 如果你大量创建第二线程。

你必须在线程执行的开始就创建你自己的自动释放池；否则，应用程序将会泄露对象。（详情参考Autorelease Pool Blocks and Threads）

## Use Local Autorelease Pool Blocks to Reduce Peak Memory Footprint

Many programs create temporary objects that are autoreleased. These objects add to the program’s memory footprint until the end of the block. In many situations, allowing temporary objects to accumulate until the end of the current event-loop iteration does not result in excessive overhead; in some situations, however, you may create a large number of temporary objects that add substantially to memory footprint and that you want to dispose of more quickly. In these latter cases, you can create your own autorelease pool block. At the end of the block, the temporary objects are released, which typically results in their deallocation thereby reducing the program’s memory footprint.

许多程序都会创建要自动释放的临时对象。这些对象会加入到程序的内存占用直到block的结尾。在多数情况下，允许临时对象积累到当前事件循环迭代的末尾不会导致过多的开销；然而，在某些情况下，你可能会创建大量的临时对象，它们实际上已添加到内存占用，而你又想更快速的清理它们。在后一种情况，你可以创建自己的自动释放池block。在block的末尾，临时对象就被释放了，这通常结果就是销毁，因此减少了程序的内存占用。

The following example shows how you might use a local autorelease pool block in a for loop.

接下来的例子展示了应该如何在一个for循环中使用自动释放池block。

|  |
| --- |
| NSArray \*urls = <# An array of file URLs #>; |
| for (NSURL \*url in urls) { |
|  |
| @autoreleasepool { |
| NSError \*error; |
| NSString \*fileContents = [NSString stringWithContentsOfURL:url |
| encoding:NSUTF8StringEncoding error:&error]; |
| /\* Process the string, creating and autoreleasing more objects. \*/ |
| } |
| } |

The for loop processes one file at a time. Any object (such as fileContents) sent an autorelease message inside the autorelease pool block is released at the end of the block.

这个for循环一次处理一个文件。每一个在自动释放池block中被发送了autorelease消息的对象会在block的末尾被释放。

After an autorelease pool block, you should regard any object that was autoreleased within the block as “disposed of.” Do not send a message to that object or return it to the invoker of your method. If you must use a temporary object beyond an autorelease pool block, you can do so by sending a retain message to the object within the block and then send it autorelease after the block, as illustrated in this example:

在自动释放池block之后，你应该注意到在block中自动释放的对象已经被“清理”了。不要再向这些对象发消息或者将它们返回给你的方法的调用者。如果你必须要在自动释放池block之外使用临时对象，你可以在block中向该对象发送一条retain消息，然后在block之后向它发送一条autorelease消息，如下面例子所示：

|  |
| --- |
| – (id)findMatchingObject:(id)anObject { |
|  |
| id match; |
| while (match == nil) { |
| @autoreleasepool { |
|  |
| /\* Do a search that creates a lot of temporary objects. \*/ |
| match = [self expensiveSearchForObject:anObject]; |
|  |
| if (match != nil) { |
| [match retain]; /\* Keep match around. \*/ |
| } |
| } |
| } |
|  |
| return [match autorelease]; /\* Let match go and return it. \*/ |
| } |

Sending retain to match within the autorelease pool block the and sending autorelease to it after the autorelease pool block extends the lifetime of match and allows it to receive messages outside the loop and be returned to the invoker of findMatchingObject:

在自动释放池block里面向match发送retain，然后再自动释放池block之后向match发送autorelease以延长其生命周期，允许它在循环之外接收消息，并将它返回给findMatchingObject:方法的调用者。

## Autorelease Pool Blocks and Threads

Each thread in a Cocoa application maintains its own stack of autorelease pool blocks. If you are writing a Foundation-only program or if you detach a thread, you need to create your own autorelease pool block.

Cocoa应用的每一个线程都维护了自己的自动释放池栈。如果你正在写一个只有Foundation的程序或者分离一个线程，你需要创建自己的自动释放池block。

If your application or thread is long-lived and potentially generates a lot of autoreleased objects, you should use autorelease pool blocks (like AppKit and UIKit do on the main thread); otherwise, autoreleased objects accumulate and your memory footprint grows. If your detached thread does not make Cocoa calls, you do not need to use an autorelease pool block.

如果你的应用或线程长期潜在的生成大量自动释放的对象，你应该使用自动释放池block（像AppKit和UIKit在主线程做的一样）；否则，自动释放的对象会积累起来，内存占用也会增长。如果你分离出的线程没有调用Cocoa方法，那你不需使用自动释放池block。

**Note:** If you create secondary threads using the POSIX thread APIs instead of NSThread, you cannot use Cocoa unless Cocoa is in multithreading mode. Cocoa enters multithreading mode only after detaching its first NSThread object. To use Cocoa on secondary POSIX threads, your application must first detach at least one NSThread object, which can immediately exit. You can test whether Cocoa is in multithreading mode with the NSThread class method [isMultiThreaded](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSThread_Class/index.html#//apple_ref/occ/clm/NSThread/isMultiThreaded).

注意：如果你使用POSIX线程API而不是NSThread创建辅助线程，你将不能使用Cocoa，除非Cocoa处在多线程模式。Cocoa只有在分离出第一个NSThread对象才会进入多线程模式。想要在辅助的POSIX线程中使用Cocoa，你的应用必须首先分离至少一个NSThread对象，可以在NSThread线程创建之后立即退出。你可以使用NSThread类的[isMultiThreaded](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSThread_Class/index.html#//apple_ref/occ/clm/NSThread/isMultiThreaded)方法测试Cocoa是否在多线程模式。

This table describes the changes to Advanced Memory Management Programming Guide.

|  |  |
| --- | --- |
| **Date** | **Notes** |
| 2012-07-17 | Updated to describe autorelease in terms of @autoreleasepool blocks. |
| 2011-09-28 | Updated to reflect new status as a consequence of the introduction of ARC. |
| 2011-03-24 | Major revision for clarity and conciseness. |
| 2010-12-21 | Clarified the naming rule for mutable copy. |
| 2010-06-24 | Minor rewording to memory management fundamental rule, to emphasize simplicity. Minor additions to Practical Memory Management. |
| 2010-02-24 | Updated the description of handling memory warnings for iOS 3.0; partially rewrote "Object Ownership and Disposal." |
| 2009-10-21 | Augmented section on accessor methods in Practical Memory Management. |
| 2009-08-18 | Added links to related concepts. |
| 2009-07-23 | Updated guidance for declaring outlets on OS X. |
| 2009-05-06 | Corrected typographical errors. |
| 2009-03-04 | Corrected typographical errors. |
| 2009-02-04 | Updated "Nib Objects" article. |
| 2008-11-19 | Added section on use of autorelease pools in a garbage collected environment. |
| 2008-10-15 | Corrected missing image. |
| 2008-02-08 | Corrected a broken link to the "Carbon-Cocoa Integration Guide." |
| 2007-12-11 | Corrected typographical errors. |
| 2007-10-31 | Updated for OS X v10.5. Corrected minor typographical errors. |
| 2007-06-06 | Corrected minor typographical errors. |
| 2007-05-03 | Corrected typographical errors. |
| 2007-01-08 | Added article on memory management of nib files. |
| 2006-06-28 | Added a note about dealloc and application termination. |
| 2006-05-23 | Reorganized articles in this document to improve flow; updated "Object Ownership and Disposal." |
| 2006-03-08 | Clarified discussion of object ownership and dealloc. Moved discussion of accessor methods to a separate article. |
| 2006-01-10 | Corrected typographical errors. Updated title from "Memory Management." |
| 2004-08-31 | Changed Related Topics links and updated topic introduction. |
| 2003-06-06 | Expanded description of what gets released when an autorelease pool is released to include both explicitly and implicitly autoreleased objects in [Using Autorelease Pool Blocks](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmAutoreleasePools.html#//apple_ref/doc/uid/20000047-CJBFBEDI). |
| 2002-11-12 | Revision history was added to existing topic. It will be used to record changes to the content of the topic. |