内存管理编程指南

介绍

无论编写任何程序，您都需要确保能够有效和高效地管理资源。程序内存就是这些资源中的一种。在 Objective-C 程序中，您必须确保您所创建的对象，在不再需要它们的时候被销毁。

在一个复杂的系统中，可能很难精确地确定从何时起您不再需要某个对象。Cocoa 定义了一些有助于使这一抉择变得更加容易的规则和原则。

重要：在 Mac OS X v10.5 及更高的版本中，您可以通过采用垃圾回收机制来使用自动内存管理。这一部分内容在*垃圾回收编程指南*中向您介绍。iOS 不提供垃圾回收机制。

谁应该阅读本文档

如果您想要了解在引用计数环境中创建，复制，维护和销毁对象的相关技术及对象所有权的规则，您应该阅读本文档。

注意：如果您想要针对 Mac OS X v10.5 或更高的版本开发新项目，在一般情况下您应该使用垃圾回收机制，除非您有充足的理由要使用本文档描述的技术。
本文档不会向您阐述分配和初始化对象以及实现初始化方法这样的细节。这些内容会在 *Objective-C 编程语言*的"分配和初始化对象"部分中进行讨论。

本文档的组织

本文档包含以下几部分：

- "内存管理规则"总结了对象所有权和销毁的规则。
- "对象的所有权和销毁"描述了基本的对象所有权策略。
- "实用内存管理"以实用性的角度透视内存管理。
- "自动释放池"描述了自动释放池—一种延迟回收的机制—在 Cocoa 程序中的用法。
- "存取方法"向您介绍如何实现存取方法。
- "实现对象复制"讨论有关对象复制的问题，比如如何决定是执行深拷贝还是浅拷贝，在您自己的子类中如何实现对象的复制。
- "Cocoa 中 Core Foundation 对象的内存管理"介绍了 Cocoa 代码中 Core Foundation 对象的内存管理技术及使用指南。
- "Nib 对象的内存管理"讨论了与 nib 文件相关的内存管理的问题。

内存管理规则

本文总结了 Objective-C 中内存管理的规则。

以下是基本规则：

- **您只能释放或自动释放您所拥有的对象。**
  - 如果您使用名字以“alloc”或“new”开头或名字中包含“copy”的方法（例如 `alloc`，`newObject` 或 `mutableCopy`）创建了一个对象，则您会获得该对象的所有权；或者如果您向一个对象发送了一条 `retain` 消息，则您也会获得该对象的所有权。
  - 您可以使用释放 `release` 或自动释放 `autorelease` 来释放一个对象的所有权。自动释放 `autorelease` 的意思是“将会发送释放 `release` 消息”（要了解究竟何时发送，请参考“自动释放池”）。

下面的规则是基本规则的衍生，或者是用于处理特殊边界情况的规则：

- 作为基本规则的推论，如果您需要将接收到的对象存储为某个实例变量的属性，您必须保留或复制该对象。（对于弱引用来说并不是这样，详见“对象的弱引用”，但这种情况一般是很罕见的。）通常，您应该将这部分工作交给存取方法（参考“存取方法”）来处理。
- 被接收的对象通常要保证在接收它的方法中仍然有效，并且该方法也可以将对象安全地返回给它的调用者。例外的情况包括多线程应用程序和一些“分布式对象”环境，然而，此时您还必须注意自己是否修改了您用来接收其它对象的对象（见“共享对象的有效性”）。在必要的时候，将保留 `retain` 与释放 `release` 或自动释放 `autorelease` 组合使用，可以防止对象因为消息的正常的边界效应而失效。

“对象的所有权和销毁”讨论了这些规则背后的推理过程。

重要：Core Foundation 对象也有类似的内存管理规则（参考 *Core Foundation 内存管理编程指南*）。但是，Cocoa 和 Core Foundation 的命名规范是不同的。特别地，Core Foundation 的“*Core Foundation 的内存管理编程指南*”中的创建规则”并不适用于返回 Objective-C 对象的方法。例如，在下面的代码片段中，您*无需*负责释放 `myInstance` 的所有权：

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

参考“Cocoa 中 Core Foundation 对象的内存管理”。

对象的所有权和销毁

对一个程序来说，最好是尽可能少地使用内存。因此，Objective-C 环境中定义了一些机制和策略，允许您对程序的内存进行管理。虽然您可以从底层实现的角度去考虑 Objective-C 程序的内存管理（详见“幕后：保留计数”），但是通常从对象所有权的角度来考虑这个问题会更容易。

在 Objective-C 程序中，对象会被创建和销毁。为了确保您的应用程序不会使用不必要的内存，对象应该在不需要它们的时候被销毁。当然，在需要对象时保证它们不被销毁也很重要。为了满足这些需求，Cocoa 定义了一种机制—对象所有权，通过该机制您可以指定您何时需要使用一个对象，又在何时完成对该对象的使用。

为了充分理解对象所有权策略在 Cocoa 中是如何实现的，您还需要阅读“自动释放池”这部分的内容。

对象所有权策略

任何对象都可能拥有一个或多个所有者。只要一个对象至少还拥有一个所有者，它就会继续存在。如果一个对象没有所有者，则运行时系统会自动销毁它（参考“回收对象”）。为了确保您清楚自己何时拥有一个对象而何时不拥有对象，Cocoa 设置了以下策略：

- 任何您自己创建的对象都归您所有。

您可以使用名字以“alloc”或“new”开头或名字中包含“copy”的方法（例如 alloc，newObject，或 mutableCopy）来“创建”一个对象。

- 您可以使用 retain 来获得一个对象的所有权。

  请记住，一个对象的所有者可能不止一个。拥有一个对象的所有权就表示您需要保持该对象存在。（“存取方法”更详细地讨论了这部分内容。）

- 当您不再使用您所拥有的对象时，您必须释放对这些对象的所有权。

  您可以通过向一个对象发送 release 消息或 autorelease 消息（在“自动释放”这一部分更详细地讨论了 autorelease）来释放您对它的所有权。因此，用 Cocoa 的术语来说，释放对象的所有权通常被称为“释放”（releasing）对象。

- 您不能释放非您所有的对象的所有权。

  这主要是前面的策略规则的一个隐含的推论，在这里将其明确地提出。

这条策略对基于 GUI 的 Cocoa 应用程序和命令行 Foundation 工具都适用。

请仔细思考下面的代码片段：

```
{
    Thingamajig *myThingamajig = [[Thingamajig alloc] init];
    // ...
    NSArray *sprockets = [myThingamajig sprockets];
    // ...
    [myThingamajig release];
}
```

这个例子完全遵守上述策略。您使用 alloc 方法创建了 Thingamajig 对象，因此，随后您在不需要该对象时对其发送了一条 release 消息。当您通过 Thingamajig 对象获得 sprockets 数组时，您并没有“创建”这个数组，所以您也没有对其发送 release 消息。

幕后：保留计数

所有权策略是在调用 retain 方法后通过引用计数–通常被称为“保留计数”–实现的。每个对象都有一个保留计数。

- 当您创建一个对象时，该对象的保留计数为 1。
- 当您向一个对象发送 retain 消息时，该对象的保留计数加 1。
- 当您向一个对象发送 release 消息时，该对象的保留计数减 1。

当您向一个对象发送 autorelease 消息时，该对象的保留计数会在将来的某个阶段减 1。

- 如果一个对象的保留计数被减为 0，该对象就会被回收（请参考“回收对象”）。

重要：通常您不必显式地查询对象的保留计数是多少（参考 retainCount）。其结果往往容易对人产生误导，因为您可能不知道您感兴趣的对象由何种框架对象保留。在调试内存管理的问题上，您只需要确保您的代码遵守所有权规则。

自动释放

NSObject 对象定义的 autorelease 方法为后续的释放标记了接收者。通过向对象发送 autorelease 消息，您亦是声明：在您发送消息的作用域之外，您不想再保留该对象。这个作用域的范围是由当前的自动释放池定义的，可参考“自动释放池”。您可以这样实现上面提到的 sprockets 方法：

```
- (NSArray *)sprockets {



    NSArray *array = [[NSArray alloc] initWithObjects:mainSprocket,

                           auxiliarySprocket, nil];

    return [array autorelease];

}
```

使用 alloc 创建数组；因此您将拥有该数组，并负责在使用后释放所有权。而释放所有权就是使用 autorelease 完成的。

当另一个方法得到 Sprocket 对象的数组时，这个方法可以假设：当不在需要该数组时，它会被销毁，但仍可以在其作用域内的任何地方被安全地使用（请参考“共享对象的有效性”）。该方法甚至可以将数组返回给它的调用者，因为应用程序对象为您的代码定义好了调用堆栈的底部。
autorelease 方法可以使您很轻松地从一个方法返回一个对象，并且仍然遵循所有权策略。为了说明这一点，来看两个 sprockets 方法的*错误*实现：

1. 这种做法是*错误*的。根据所有权策略，这将会导致内存泄漏。

```
- (NSArray *)sprockets {



    NSArray *array = [[NSArray alloc] initWithObjects:mainSprocket,

                           auxiliarySprocket, nil];

    return array;

}
```

2. 对象只能在 `sprockets` 方法的内部引用新的数组对象。在该方法返回后，对象将失去对新对象的引用，导致其无法释放所有权。这本身是没有问题的。但是，按照先前提出的命名约定，调用者并没有得到任何提示，不知道它已经获得了返回的对象。因此调用者将不会释放返回的对象的所有权，最终导致内存泄漏。

3. 这种做法也是*错误*的。虽然对象正确地释放了新数组的所有权，但是在 `release` 消息发送之后，新数组将不再具有所有者，所以它会被系统立即销毁。因此，该方法返回了一个无效（已释放）的对象：

```
- (NSArray *)sprockets {



    NSArray *array = [[NSArray alloc] initWithObjects:mainSprocket,

                          auxiliarySprocket, nil];

    [array release];

    return array; // array is invalid here

}
```

最后，您还可以像这样正确地实现 `sprockets` 方法：

```
- (NSArray *)sprockets {



    NSArray *array = [NSArray arrayWithObjects:mainSprocket,

                          auxiliarySprocket, nil];

    return array;

}
```

您并没有拥有 `arrayWithObjects:`返回的数组，因此您不用负责释放所有权。不过，您可以通过 `sprockets` 方法安全地返回该数组。

重要：要理解这一点，很容易令人联想到 `arrayWithObjects:`方法本身正是使用 `autorelease` 实现的。虽然在这种情况下是正确的，但严格来讲它属于实现细节。正如您不必关心一个对象的实际保留计数一样，您同样不必关心返回给您的对象是否会自动释放。您唯一需要关心的是，您是否拥有这个对象。

共享对象的有效性

Cocoa 的所有权策略规定，被接收的对象通常应该在整个调用方法的作用域内保持有效。此外，还可以返回从当前作用域接收到的对象，而不必担心它被释放。对象的 getter 方法返回一个缓存的实例变量或者一个计算值，这对您的应用程序来说无关紧要。重要的是，对象会在您需要它的这段期间保持有效。

这一规则偶尔也有一些例外情况，主要可以总结为以下两类。

1. 当对象从一个基本的集合类中被删除的时候。

```
heisenObject = [array objectAtIndex:n];

[array removeObjectAtIndex:n];

// heisenObject could now be invalid.
```

2. 当对象从一个基本的集合类中被删除时，它会收到一条 `release`（不是 `autorelease`）消息。如果该集合是这个被删除对象的唯一所有者，则被删除的对象（例子中的 `heisenObject`）将被立即回收。

3. 当一个“父对象”被回收的时候。

```
id parent = <#create a parent object#>;

// ...

heisenObject = [parent child] ;

[parent release]; // Or, for example: self.parent = nil;

// heisenObject could now be invalid.
```

4. 在某些情况下，您通过另外一个对象得到某个对象，然后直接或间接地释放父对象。如果释放父对象会使其被回收，而且父对象是子对象的唯一所有者，那么子对象（例子中的 `heisenObject`）将同时被回收（假设它在父对象的 `dealloc` 方法中收到一条 `release` 而非 `autorelease` 消息）。

为了防止这些情况发生，您要在接收 `heisenObject` 后保留该对象，并在使用完该对象后对其进行释放，例如：

```
heisenObject = [[array objectAtIndex:n] retain];

[array removeObjectAtIndex:n];

// use heisenObject.

[heisenObject release];
```

存取方法

如果您的类中有一个实例变量本身是一个对象，那么您必须保证任何为该实例变量赋值的对象在您使用它的过程中不会被释放。因此，您必须在对象赋值时要求获取它的所有权。您还必须保证在将来会释放任何您当前持有的值的所有权。

例如，如果您的对象允许设置它的 main Sprocket，您可以这样实现 `setMainSprocket:`方法：

```
- (void)setMainSprocket:(Sprocket *)newSprocket {

    [mainSprocket autorelease];

    mainSprocket = [newSprocket retain]; /* Claim the new Sprocket. */

    return;

}
```

现在，`setMainSprocket:`可能在被调用时带一个 Sprocket 对象的参数，而调用者想要保留该 Sprocket 对象，这意味着您的对象将与其他对象共享 Sprocket。如果有其他对象修改了 Sprocket，您的对象的 main Sprocket 也会发生变化。但如果您的 Thingamajig 需要有属于它自己的 Sprocket，您可能会认为该方法应该复制一份私有的副本（您应该记得复制也会得到所有权）：

```
- (void)setMainSprocket:(Sprocket *)newSprocket {

    [mainSprocket autorelease];

    mainSprocket = [newSprocket copy]; /* Make a private copy. */

    return;

}
```

以上几种实现方法都会自动释放原来的 main sprocket。如果 `newSprocket` 和 `mainSprocket` 是同一个的对象，并且 Thingamajig 对象是它的唯一所有者的话，这样做可以避免一个可能出现的问题：在这种情况下，当 sprocket 被释放时，它会被立即回收，这样一旦它被保留或复制，就会导致错误。下面的实现也解决了这个问题：

```
- (void)setMainSprocket:(Sprocket *)newSprocket {

    if (mainSprocket != newSprocket) {

        [mainSprocket release];

        mainSprocket = [newSprocket retain]; /* Or copy, if appropriate. */

    }

}
```

在所有这些情况中，看起来好像最终为您的对象设置的 mainSprocket 泄漏了，因为您不用释放对它的所有权。这些由 `dealloc` 方法负责，在“回收对象”部分进行了介绍。“存取方法”部分更详细地描述了存取方法及其实现。

回收对象

当一个对象的保留计数减少至 0 时，它的内存将被收回–在 Cocoa 术语中，这被称为“释放”（freed）或“回收”（deallocated）。当一个对象被回收时，它的 `dealloc` 方法被自动调用。`dealloc` 方法的作用是释放对象占用的内存，释放其持有的所有资源，包括所有实例变量对象的所有权。

如果在您的类中有实例变量对象，您必须实现一个 `dealloc` 方法来释放它们，然后调用超类的 `dealloc` 实现。例如，如果 Thingamajig 类含有 `mainSprocket` 和 `auxiliarySprocket` 实例变量，您应该这样实现该类的 `dealloc` 方法：

```
- (void)dealloc {

    [mainSprocket release];

    [auxiliarySprocket release];

    [super dealloc];

}
```

重要：决不要直接调用另一个对象的 `dealloc` 方法。
您不应该让系统资源的管理依赖于对象的生命周期；参考“资源管理”。

当应用程序终止时，对象有可能没有收到 `dealloc` 消息。由于进程的内存在退出时被自动清空，因此与调用一切内存管理方法相比，简单地让操作系统清理资源效率更高。

通过引用返回的对象

Cocoa 的一些方法可以指定一个通过引用（即 `ClassName **` 或 `id *`）返回的对象。下面有几个使用 `NSError` 对象的例子，该对象包含有错误出现时的信息，例如：

- `initWithContentsOfURL:options:error:` (`NSData`)
- `initWithContentsOfFile:encoding:error:` (`NSString`)
- `executeFetchRequest:error:` (`NSManagedObjectContext`)

在这些情况下，前面描述的规则同样适用。当您调用这些方法中的任何一种时，由于您没有创建 `NSError` 对象，因此您不会拥有该对象–同样也无需释放它。

```
NSString *fileName = <#Get a file name#>;

NSError *error = nil;

NSString *string = [[NSString alloc] initWithContentsOfFile:fileName

                    encoding:NSUTF8StringEncoding error:&error];
```

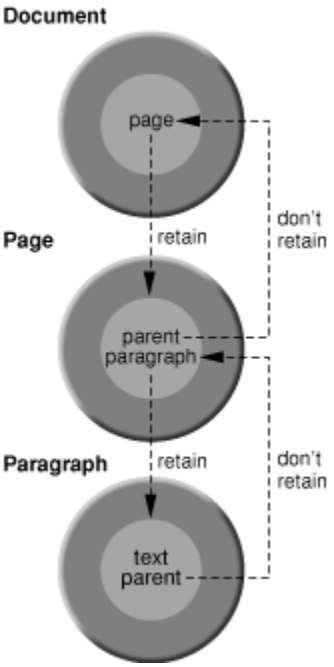
```
if (string == nil) {

    // deal with error ...

}

// ...

[string release];
```

如果因为任何原因，返回的对象的所有权不能遵守基本规则，这将在方法的文档中明确地阐明（例如，参考 `dataFromPropertyList:format:errorDescription:`）。

保留循环

在某些情况下，两个对象之间可能会出现循环引用的情况，也就是说，每一个对象都包含一个实例变量引用对方对象。例如，考虑一个文本程序，程序中对象间的关系如图 1 所示。“文档（Document）”对象为文档中的每个页面创建一个“页（Page）”对象。每个 Page 对象具有一个实例变量，用来跟踪该页所在的文档。如果 Document 对象保留了 Page 对象， 同时 Page 对象也保留 Document 对象，则这两个对象都永远不会被释放。只有 Page 对象被释放，Document 的引用计数才能变为 0，而只有 Document 对象被回收，Page 对象才能被释放。

图 1 保留循环示意图



针对保留循环问题的解决方案是“父”对象应保留其“子”对象，但子对象不应该保留其父对象。因此，在图 1 中，document 对象要保留 page 对象，但 page 对象不保留 document 对象。子对象对其父对象的引用是一个弱引用的例子，这部分内容在“对象的弱引用”有更充分的描述。

对象的弱引用

保留一个对象创建了一个对该对象的“强”引用。一个对象只有在它的所有强引用都被释放后才能被回收。因此，一个对象的生命周期取决于其强引用的所有者。在某些情况下，这种行为可能并不理想。您可能想要引用一个对象而不妨碍对象本身的回收。对于这种情况，您可以获取一个“弱”引用。弱引用是通过存储一个指向对象的指针创建的，而不是保留对象。

弱引用在可能会出现循环引用的情况下是必不可少的。例如，如果对象 A 和对象 B 互相通信，两者都需要引用对方。如果每个对象都保留对方对象，则这两个对象只有在它们之间的连接中断后才能被回收，但是它们之间的连接又只能在有对象被回收后才能中断。为了打破这种循环，其中一个对象需要扮演从属角色，得到另一个对象的一个弱引用。举个具体的例子，在视图层次中，父视图拥有其子视图，也因此能够保留子视图，但父视图并不归子视图所有；然而子视图仍需要知道谁是它的父视图，因此它保持一个对其父视图的弱引用。

Cocoa 中弱引用的其他适用情况包括：表格数据源，大纲视图项，通知观察者以及其余项目标和委托，但不仅限于上述情况。

重要：在 Cocoa 中，引用表格数据源，大纲视图项，通知观察者和委托都被看作是弱引用（例如，`NSTableView` 对象不保留其数据源，`NSApplication` 对象不保留它的委托）。本文档仅仅介绍了这一公约的例外情况。

在向您弱引用的对象发送消息时，您需要小心谨慎。如果您在一个对象被回收之后向它发送消息，您的应用程序将会崩溃。您必须为对象何时有效制定有明确界定的条件。在大多数情况下，被弱引用的对象知道其他对象对它的弱引用，这和循环引用的情况是一样的，并且它还能够在自己被回收时通知其他对象。例如，当您向通知中心注册一个对象的时候，通知中心会存储一个对该对象的弱引用，并且在适当的消息发布时，还会向该对象发送消息。当对象被回收时，您需要向通知中心解注册该对象，以防通知中心向这个已经不存在的对象继续发送消息。同样，当一个委托对象被回收时，您需要通过向其他对象发送一条带 `nil` 参数的 `setDelegate:` 消息来删除委托链接。这些消息通常由对象的 `dealloc` 方法发出。

资源管理

通常，不应该由您来管理一些稀缺资源，比如文件描述符，网络连接和 `dealloc` 方法中的缓冲区/高速缓存。特别地，您设计的类不应该让您错误地认为 `dealloc` 会在您觉得该调用的时候被调用。`dealloc` 的调用可能会因为 bug 或应用程序销毁而被延误或回避。

相反，如果您有一个类，由它的实例管理稀缺资源，则您在设计应用程序时应该让自己知道何时不再需要这些资源，并且可以在那个时刻通知实例“清理”这些资源。通常，接下来您应该释放该实例，然后调用 `dealloc`，但如果您不这样做也不会有问题。

如果您试图让 `dealloc` 肩负起资源管理的责任，会出现的一些问题：

1. 对象图销毁的顺序依赖性。

   对象图销毁机制内在是无序的。虽然通常您可能期望甚至得到一个特定的顺序，但是这样做的同时您也引入了脆弱性。如果一个对象意外落入自动释放池，则销毁顺序会改变，这可能会导致意想不到的后果。

2. 稀缺资源的未回收。

   内存泄露当然是应该被修复的 bug，但它们一般来说不会是直接致命的错误。然而，如果稀缺资源在您希望它们被释放时没有被释放，这会导致非常非常严重的问题。例如，如果您的应用程序耗尽了文件描述符，那么用户可能无法保存数据。

3. 清除在错误的线程上被执行的逻辑。

如果一个对象在意外的时刻落入自动释放池，那么无论它进入哪个线程池，它都将被回收。这对于那些本应该只由一个线程访问的资源来说很容易产生致命的错误。

实用内存管理

本文为您提供了一种透视内存管理的实用性视角。这部分内容涵盖了"对象所有权和销毁"中介绍的基本概念，不过采用了更加面向代码实现的视角。
遵从以下几条简单的规则可以使内存管理变得更加容易，而不遵守这些规则将几乎肯定会在某些时候导致内存泄漏，或者由于消息被发送给已释放的对象而导致运行时异常。

基础知识

为了让应用程序的内存消耗尽可能低，您应该清除掉不使用的对象，但是您需要确保您清除的不是正在被使用的对象。因此，您需要一种机制，可以让您标记那些仍然有用的对象。所以，从许多方面来讲，站在"对象的所有权"的角度看内存管理是最好理解的。

- 一个对象可以有一个或一个以上的所有者。

  采用类比的方式，您可以联想一个分时租用公寓。

- 当一个对象没有所有者的时候，它会被销毁。

  继续用类比的方法，您可以联想一个分时合租的寓所，但当地居民并不喜欢它。如果没有所有者，这处合租寓所将被拆除。

- 为了确保您感兴趣的对象不被销毁，您必须成为它的一个所有者。

  您可以建造一所新公寓，或入住一所现有的公寓。

- 为了让您不再感兴趣的对象能够被销毁，您应该释放它的所有权。

  您可以出售您的分时租用公寓。

为了支持这个模型，Cocoa 提供了一种被称为"引用计数"或"保留计数"的机制。每一个对象都有一个保留计数。当对象被创建的时候，其保留计数为 `1`。当保留计数减少至 `0` 时，对象会被回收（销毁）。您可以使用各种方法来操作保留计数（即获取或释放所有权）：

`alloc`

为对象分配内存并返回该对象，其保留计数为 `1`。

您拥有以单词 `alloc` 或 `new` 开头的任意方法创建的对象。

`copy`

为对象创建一份副本并返回该对象，其保留计数为 `1`。

如果您复制一个对象，您就拥有了这个对象的副本。这对于任何名字中包含单词 `copy` 的方法都是适用的，这里的"copy"是指被返回的对象。

retain

使一个对象的保留计数增加 1。

获得一个对象的所有权。

release

使一个对象的保留计数减少 1。

释放一个对象的所有权。

autorelease

使一个对象的引用计数在未来的某个阶段减少 1。

在未来的某个阶段释放一个对象的所有权。

内存管理的实用规则如下（也可以参考“内存管理规则”）：

- 您只拥有那些您使用名字以“alloc”或“new”开头或者名字中包含“copy”的方法（例如 alloc，newObject 或 mutableCopy）创建的对象，或者是那些收到了您发送的 retain 消息的对象。

  许多类提供了形如+className...的方法，您可以使用它们获得该类的一个新的实例。这些方法通常被称为“简便构造函数”，它们创建一个新的类的实例，对其进行初始化并将其返回供您使用。您并不拥有从简便构造函数或其它存取方法返回的对象。

- 一旦您使用完一个您*拥有*的对象，您应该使用 release 或 autorelease 释放这个对象的所有权。

  通常，您应该使用 release，而不是 autorelease。只有在不适合立即回收对象的情况下，您才应该使用 autorelease，比如您要从某个方法返回对象。（注意：这并不是说 release *必然*会引起对象的回收—只有当保留计数减少至 0 时才会发生回收—但它也有可能会发生，而有时您需要防止出现这种情况，请参考“从方法返回对象”中的例子。）

- 实现 dealloc 方法来释放您拥有的实例变量。

- 您不应该直接调用 dealloc（除非是您在自定义的 dealloc 方法中调用超类的实现）。

几个简单的例子

下面几个简单的例子对比说明了使用 alloc，简便构造函数和存取方法创建一个新对象。

第一个例子使用 alloc 创建了一个新的字符串对象。因此，它必须被释放。

```
- (void)printHello {

    NSString *string;

    string = [[NSString alloc] initWithString:@"Hello"];

    NSLog(string);
```

```
    [string release];

}
```

第二个例子使用简便构造函数创建了一个新的字符串对象。此外没有额外的工作要做。

```
- (void)printHello {

    NSString *string;

    string = [NSString stringWithFormat:@"Hello"];

    NSLog(string);

}
```

第三个例子使用存取方法获取一个字符串对象。与简便构造函数一样，没有额外的工作要做。

```
- (void)printWindowTitle {

    NSString *string;

    string = [myWindow title];

    NSLog(string);

}
```

使用存取方法

虽然使用存取方法有时看似繁琐，有故意卖弄之嫌，但如果您坚持使用存取方法，则内存管理方面出现问题的可能性将大大减小。如果您在代码中对实例变量全部使用 `retain` 和 `release`，那么几乎可以肯定您在做错误的事情。

考虑一个"计数器"对象（Counter），您要设置它的计数。

```
@interface Counter : NSObject {

    NSNumber *count;

}
```

为了获取和设置计数的值，您需要定义了两个存取方法。（下面的例子给出了存取方法的简单实现。在"存取方法"中有对它们更加详细的介绍。）在 get 方法中，您只是回传了一个变量，所以没有必要进行 `retain` 或 `release`：

```
- (NSNumber *)count {

    return count;
```

```
}
```

在 set 方法中，如果其他人都按照同样的规则进行操作，则您需要假设新的计数可能会在任何时刻被销毁，因此您需要获得对象的所有权–向它发送一条 `retain` 消息–来确保它不会被销毁。在这里您还需要通过向旧的计数对象发送一条 `release` 消息来释放它的所有权。（Objective-C 中允许向 `nil` 发送消息，因此在计数尚未被设置时仍然可以这么做。）您必须在`[newCount retain]`之后发送这个消息，以防这两者是同一个对象–您肯定不希望由于疏忽造成对象意外被回收。

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

    [newCount retain];

    [count release];

    // make the new assignment

    count = newCount;

}
```

只有在两处地方您不该使用存取方法来设置实例变量–`init` 方法和 `dealloc`。为了用一个表示零的数字对象初始化一个计数对象，您可以按照下面的方式实现一个 `init` 方法：

```
- init {

    self = [super init];

    if (self) {

        count = [[NSNumber alloc] initWithInteger:0];

    }

    return self;

}
```

为了用一个非零的计数初始化计数器，您可以这样实现一个 `initWithCount:`方法：

```
- initWithCount:(NSNumber *)startingCount {

    self = [super init];

    if (self) {

        count = [startingCount copy];

    }

    return self;
```

```
}
```

由于“计数器”类（Counter）有一个对象实例变量，您还必须实现一个 `dealloc` 方法。该方法应该可以通过向实例变量发送 `release` 消息来释放任何实例变量的所有权，并且最终调用超类的 `dealloc` 实现：

```
- (void)dealloc {

    [count release];

    [super dealloc];

}
```

实现重置方法

假设您想实现一个方法来重置计数器。那么您有两种选择，第一种是使用简便构造函数创建一个新的 `NSNumber` 对象—也因此没有必要发送任何 `retain` 或 `release` 消息。请注意，两种方法都使用了类的 set 存取方法。

```
- (void)reset {

    NSNumber *zero = [NSNumber numberWithInteger:0];

    [self setCount:zero];

}
```

第二种是使用 `alloc` 创建 `NSNumber` 实例，因此您要相应地使用 `release`。

```
- (void)reset {

    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];

    [self setCount:zero];

    [zero release];

}
```

常见错误

下面几小节举例说明常见的错误。

没有使用存取方法

下面的例子在一些简单的情况下几乎肯定可以正常工作，但这个例子避免使用存取方法，这样做几乎肯定会在某个阶段（当您忘记保留或释放，或者当您的实例变量的内存管理语义发生变化的时候）导致错误。

```
- (void)reset {

    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];

    [count release];

    count = zero;

}
```

另外请注意，如果您正在使用键值观察（参考*键值观察编程指南*），那么用这种方式改变变量是不兼容 KVO 的。

实例泄露

```
- (void)reset {

    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];

    [self setCount:zero];

}
```

新数字的保留计数是 1（来自 `alloc`），而且在该方法释放的作用域内没有与之对应的 release。新数字是不可能被释放的，这将导致内存泄漏。

向非您所有的实例发送 release

```
- (void)reset {

    NSNumber *zero = [NSNumber numberWithInteger:0];

    [self setCount:zero];

    [zero release];

}
```

如果没有调用 `retain`，则在当前的自动释放池被释放后，下一次您访问 `count` 会失败。简便构造方法返回一个会自动释放的对象，所以您不必再发送 release。这样做意味着，当因 `autorelease` 而产生的 `release` 被发送后，保留计数会被减为 0，且对象将被释放。当您下次想要访问计数时，您将向一个已经被释放的对象发送消息（这时通常您会得到一个 `SIGBUS 10` 错误）。

会造成混乱的情况

使用集合

当您把一个对象添加到一个集合，比如数组，字典或集合，集合拥有对象的所有权。当对象从集合中删除或集合本身被释放时，集合会释放所有权。因此，举例来说，如果您想创建一个数字数组，您可以选择以下方法中的一种：

```
NSMutableArray *array;

NSUInteger i;

// ...

for (i = 0; i < 10; i++) {

    NSNumber *convenienceNumber = [NSNumber numberWithInteger:i];

    [array addObject:convenienceNumber];

}
```

在这段代码中，您没有调用 `alloc`，因此也没有必要调用 `release`。没有必要保留新的数字对象（`convenienceNumber`），因为数组会为您代劳。

```
NSMutableArray *array;

NSUInteger i;

// ...

for (i = 0; i < 10; i++) {

    NSNumber *allocedNumber = [[NSNumber alloc] initWithInteger: i];

    [array addObject:allocedNumber];

    [allocedNumber release];

}
```

在这段代码中，您*需要*在 `for` 循环的作用域内向 `allocedNumber` 发送 `release` 消息，以抵消之前的 `alloc`。由于数组在用 `addObject:`方法添加数字时对其进行了保留，因此只要它还在数组中就不会被释放。

要理解这一点，您要把自己放在实现这种集合类的作者的位置。您要确保交给您管理的对象不能在您的眼皮底下消失，所以您要在这些对象被加入集合中时向它们发送 `retain` 消息。如果它们被删除，您还必须相应地发送 `release` 消息，并且在您自己的 `dealloc` 方法中，您还应该向其余的对象发送 `release` 消息。

从方法返回的对象

当您从一个方法中返回一个局部变量时，您不仅要保证自己遵守了内存管理规则，而且要保证接收方在对象被释放之前一直有机会使用该对象。当您返回一个新创建的（您拥有的）对象时，您应该是用 `autorelease` 而不是 `release` 来释放所有权。

请考虑一个很简单的 `fullName` 方法，用它来连接 `firstName` 和 `lastName`。一种可行的正确的实现方法（仅仅从内存管理的角度而言—当然从功能性的角度考虑，它仍有很多不足之处）可能如下面的代码所示：

```
- (NSString *)fullName {

    NSString *string = [NSString stringWithFormat:@"%@ %@", firstName, lastName];

    return string;

}
```

按照最基本的规则，您并不拥有 `stringWithFormat` 返回的字符串，所以它可以安全地从该方法中返回。

下面这种实现方法也是正确的：

```
- (NSString *)fullName {

    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@", firstName, lastName] autorelease];

    return string;

}
```

您拥有 `alloc` 返回的字符串，但您随后向它发送了一条 `autorelease` 消息，因此在您失去它的引用之前，您已经放弃了所有权，并且这样做也是满足内存管理规则的。这种实现方法的精髓在于使用了 `autorelease` 而不是 `release`，要意识到这一点。

相比之下，*下面的代码是错误的*：

```
- (NSString *)fullName {

    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@", firstName, lastName] release];

    return string;

}
```

纯粹从内存管理的角度来讲，它看起来是正确的：您拥有 `alloc` 返回的字符串，并向它发送一条 `release` 的信息来释放所有权。然而从实用角度来看，该字符串很有可能在那一步就被回收了（它可能没有任何其他的所有者），因此该方法的调用者会接收到一个无效的对象。这说明了为什么 `autorelease` 非常实用—它能让您推迟释放，您可以在未来的某一时刻过后再释放对象。

为了追求完整性，*下面的代码也是错误的*：

```
- (NSString *)fullName {
```

```
    NSString *string = [[NSString alloc] initWithFormat:@"%@ %@", firstName, lastName];

    return string;

}
```

您拥有 `alloc` 返回的字符串，但是在您有机会释放所有权之前，您就失去了对该对象的引用。根据内存管理规则，这将导致内存泄漏，因为调用者没有得到任何迹象表明他们拥有返回的对象。

自动释放池

本章向您介绍如何微调您的应用程序对自动释放池的控制；有关使用自动释放机制的一般介绍请参考文档“对象的所有权和销毁”。

自动释放池概述

自动释放池是一个 `NSAutoreleasePool` 实例，其中“包含”已经收到 `autorelease` 消息的其他对象；当自动释放池被回收时，它会向其中的每个对象发送一条 `release` 消息。一个对象可以被数次放入一个自动释放池中，并且在每次被放入池中的时候都会收到一条 `release` 消息。因此，向对象发送 `autorelease` 消息（而不是 `release` 消息）可以至少将该对象的生命周期延长至自动释放池本身被释放的时候（如果在此期间对象被保留，则它可以存活更久）。

Cocoa 总是期望有一个自动释放池可用。如果自动释放池不可用，那么自动释放对象就无法得到释放，您也就泄漏了内存。如果当自动释放池不可用的时候，您发送了 autorelease 消息，那么 Cocoa 会记录相应的错误信息。

您可以使用常见的 `alloc` 和 `init` 消息来创建一个 `NSAutoreleasePool` 对象，并使用 `drain`（如果您向一个自动释放池发送 `autorelease` 或 `retain` 消息，会引发异常—要了解 `release` 和 `drain` 之间的差异，请参考“垃圾回收”）销毁它。自动释放池应该总是在与它被创建时所处的相同上下文环境（方法或函数的调用，或循环体）中被销毁。
自动释放池被置于一个堆栈中，虽然它们通常被称为被“嵌套”的。当您创建一个新的自动释放池时，它被添加到堆栈的顶部。当自动释放池被回收时，它们从堆栈中被删除。当一个对象收到送 `autorelease` 消息时，它被添加到当前线程的目前处于栈顶的自动释放池中。

嵌套自动释放池的能力是指，您可以将它们包含进任何函数或方法中。例如，`main` 函数可以创建一个自动释放池，并调用另一个创建了另外一个自动释放池的函数。或者，一个方法可以有一个自动释放池用于外循环，而有另一个自动释放池用于内循环。嵌套自动释放池的能力是一种很显著的优势，但是，当发生异常时也会有副作用（见“自动释放池的作用域和嵌套自动释放池的含义”）。
Application Kit 会在一个事件周期（或事件循环迭代）的开端—比如鼠标按下事件—自动创建一个自动释放池，并且在事件周期的结尾释放它，因此您的代码通常不必关心。但是，有三种情况您应该使用您自己的自动释放池：

- 如果您正在编写一个不是基于 Application Kit 的程序，比如命令行工具，则没有对自动释放池的内置支持；您必须自己创建它们。
- 如果您生成了一个从属线程，则一旦该线程开始执行，您必须立即创建您自己的自动释放池；否则，您将会泄漏对象。（详情请参考“自动释放池和线程”。）

- 如果您编写了一个循环，其中创建了许多临时对象，您可以在循环内部创建一个自动释放池，以便在下次迭代之前销毁这些对象。这可以帮助减少应用程序的最大内存占用量。

自动释放池是“按序”使用的。*一般情况下，您不应该将自动释放池作为某个对象的实例变量。*

非 Application Kit 程序中的自动释放池

在不是基于 Application Kit 的程序中，启用自动释放机制是很容易的。您可以简单地在 `main()` 函数的开始创建一个自动释放池，并在 `main()` 函数的最后释放它–这是 Xcode 中的 Foundation Tool 模板使用的模式。这样就为任务的生命周期建立了一个自动释放池。但是，这也意味着在任务的生命周期中创建的所有的自动释放对象，都要等到任务完成时才会被销毁。这可能会导致任务的内存占用量不必要地增加。您也可以考虑创建具有更窄作用域的自动释放池。

许多程序具有深层循环，并在其中完成程序的大部分工作。为了减少内存占用峰值，您可以在该循环中一次迭代的开始创建一个自动释放池，并在本次迭代的最后销毁它。

您的 `main` 函数可能看起来类似清单 1 中的代码。

清单 1 非 AppKit 程序的 `main` 函数的例子

```
void main()

{

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



   NSArray *args = [[NSProcessInfo processInfo] arguments];



   for (NSString *fileName in args) {



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



       NSError *error = nil;

      NSString *fileContents = [[[NSString alloc] initWithContentsOfFile:fileName

                                    encoding:NSUTF8StringEncoding error:&error] autorelease];
```

```
        /* Process the string, creating and autoreleasing more objects. */


        [loopPool drain];
    }

    /* Do whatever cleanup is needed. */
     [pool drain];

    exit (EXIT_SUCCESS);
}
```

for 循环每次处理一个文件。NSAutoreleasePool 对象在循环开始的时候被创建，并在循环结束时被释放。因此，在循环内部，任何收到 autorelease 消息的对象（比如 fileContents）都会被添加到 loopPool 中，并且，当 loopPool 在循环的最后被释放的时候，那些对象也会被释放（通常会引发这些对象的回收，从而减少了程序的内存占用量）。

自动释放池和线程

Cocoa 应用程序中的每个线程都会维护一个自己的 NSAutoreleasePool 对象的堆栈。当一个线程终止时，它会自动地释放所有与自身相关的自动释放池。在基于 Application Kit 的应用程序中，自动释放池会在程序的主线程中被自动创建和销毁，所以，您的代码通常无需处理它们。但是，如果您在 Application Kit 的主线程之外发起 Cocoa 调用，则您需要创建自己的自动释放池。如果您正在编写一个 Foundation 应用程序，或者如果您拆分了一个线程，则是属于这种情况。

如果您的应用程序或线程是长期存在的，并且会潜在地生成大量的自动释放对象，那么您应该定期地销毁和创建自动释放池（像 Application Kit 在主线程中所做的那样），否则，自动释放对象会产生堆积且您的内存占用量也会增长。如果被拆分的线程不会发起 Cocoa 调用，那么您无需创建自动释放池。

注意：如果您使用 POSIX 线程 API（而不是 NSThread）创建从属线程，则您无法使用 Cocoa–包括 NSAutoreleasePool 在内–除非 Cocoa 处于多线程模式。Cocoa 只有在拆分它的第一个 NSThread 对象之后才进入多线程模式。要在从属 POSIX 线程中使用 Cocoa，您的应用程序必须首先至少拆分 1 个可以立即退出的 NSThread 对象。您可以使用 NSThread 类的 isMultiThreaded 方法测试 Cocoa 是否处于多线程模式。

自动释放池的作用域和嵌套自动释放池的含义

我们通常会提及自动释放池是被嵌套的，如清单 1 所示。但是，您也可以认为嵌套自动释放池位于一个堆栈中，其中，“最内层”的自动释放池位于栈顶。如前所述，嵌套自动释放池实际上是这样实现的：程序中的每个线程都维护一个自动释放池的堆栈。

当您创建一个自动释放池时，它被压入当前线程的堆栈的栈顶。*当一个对象被自动释放时—也就是说，当一个对象收到一条* `autorelease` *消息或者当它作为一个参数被传入* `addObject:` *类方法时—它总是被放入堆栈顶部的自动释放池中。*
因此，自动释放池的作用域是由它在堆栈中的位置以及它的存在情况定义的。自动释放对象被添加至栈顶的自动释放池中。如果另一个自动释放池被创建，则当前位于栈顶的池就超出其作用域，直到新的池被释放为止（此时原来的自动释放池再次成为栈顶的自动释放池）。当自动释放池本身被释放的时候，它（显然）就永久地超出其作用域。

如果您释放了一个不是位于堆栈顶部的自动释放池，则这会导致堆栈中所有位于它上面的（未释放的）自动释放池，连同它们包含的所有对象一起被释放。当您用完自动释放池时，如果您一时疏忽，忘记向它发送 `release` 消息（不推荐您这样做），那么，当嵌套在它外层的自动释放池中的某个被释放时，它也会被释放。

这种行为对于异常的处理很有意义。如果发生异常，并且线程突然转移出当前的上下文环境，则与该上下文相关联的自动释放池将被释放。但是，如果被释放的池不是线程堆栈顶部的池，则所有位于该自动释放池之上的自动释放池也会被释放（并在这个过程中释放其中所有的对象）。然后，先前位于被释放的池下面的自动释放池则成为线程堆栈最顶端的自动释放池。由于这种行为，异常处理程序则不需要释放收到 `autorelease` 消息的对象。对于异常处理程序来说，没有必要也不值得向它的自动释放池发送 `release`，除非异常处理程序重新引发该异常。

保证基础所有权策略

通过创建一个自动释放池，而不是简单地释放对象，您可以将临时对象的生命周期延长至自动释放池的生命周期。在自动释放池被回收之后，您应该将该池有效期间被自动释放的任何对象视为“已被销毁”，而不应向这个对象发送消息或将它返回给方法的调用者。

如果您必须超出自动释放的上下文范围，使用其中的一个临时对象，您可以这样做：在该上下文范围内向这个对象发送一条 `retain` 消息，然后，在自动释放池被释放之后向它发送 `autorelease`，例如：

```
- (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 release];
    }

    return [match autorelease];   /* Let match go and return it. */
}
```

在 `subpool` 有效期间向 `match` 发送 `retain`，并且在 `subpool` 被释放之后向 `match` 发送 `autorelease`，如此一来，`match` 就被有效地从 `subpool` 移至先前有效的池中。这样就延长了 `match` 的生命周期，并允许它接收循环之外的消息，而且被返回给 `findMatchingObject:`的调用者。

垃圾回收

虽然垃圾回收系统（详见*垃圾回收编程指南*）不使用自动释放池本身，但是，如果您正在开发一种混合框架（即一种可以同时用于垃圾回收和引用计数环境的框架），自动释放池可以为回收者提供一些提示信息。

当您想要释放那些已经被添加到自动释放池中的对象的所有权时，自动释放池就要被释放。这往往会导致当时堆积的临时对象被销毁–例如，在事件周期的结尾，或者在一个创建了大量临时对象的循环期间。一般来说，这些也是有助于提示垃圾回收者需要进行垃圾回收的时间点。

在垃圾回收环境中，`release` 是一个空操作。因此，`NSAutoreleasePool` 提供了 `drain` 方法，在引用计数环境中，该方法的作用等同于调用 `release`，但在垃圾回收环境中，它会触发垃圾回收（如果自上次垃圾回收以来分配的内存大于当前的阈值）。因此，在通常情况下，您应该使用 `drain` 而不是 `release` 来销毁自动释放池。

存取方法

本章向您介绍为什么应该使用存取方法，以及您应该如何声明和实现它们。

使用存取方法的一个主要原因是封装（见 *Objective-C 面向对象编程*中的“封装”部分）。在引用计数环境中，使用存取方法还有一个特别的好处，它们可以为您的类处理大部分的基本内存管理。

声明存取方法

通常，您应该使用 Objective - C 的属性声明功能来声明存取方法，例如：

```
@property (copy) NSString *firstName;
```

```
@property (readonly) NSString *fullName;

@property (retain) NSDate *birthday;

@property NSInteger luckyNumber;
```

上述声明为属性指定了明确的内存管理语义。

实现存取方法

在很多情况下，您可以（而且应该）避免实现您自己的存取方法，而使用 Objective-C 的属性声明功能，并要求编译器为您合成存取方法：

```
@synthesize firstName;

@synthesize fullName;

@synthesize birthday;

@synthesize luckyNumber;
```

即使您需要提供您自己的实现，您也应该使用声明属性来声明存取方法–当然，您必须保证您的实现符合您给出的规范。（要特别注意的是，在默认情况下，声明属性是原子的；如果您不提供原子的实现，那么您应该在声明中指定 `nonatomic`。）

对于简单的对象值来说，大致有三种方式来实现存取方法：

1. Getter 在返回值之前保留并自动释放该值；setter 释放旧的值并保留（或复制）新的值。
2. Getter 返回值；setter 自动释放旧的值并保留（或复制）新的值。
3. Getter 返回值；setter 释放旧的值并保留（或复制）新的值。

方法 1

在方法 1 中，getter 返回的值在调用作用域内被自动释放：

```
- (NSString*) title {

    return [[title retain] autorelease];

}



- (void) setTitle: (NSString*) newTitle {

    if (title != newTitle) {
```

```
        [title release];

        title = [newTitle retain]; // Or copy, depending on your needs.

    }

}
```

由于 get 存取器返回的对象在当前作用域内被自动释放，因此，如果属性的值发生改变，它仍然有效。这使得存取器更加健壮，但额外开销更多。如果您希望您的 getter 方法能够被频繁调用，则您也应该考虑到，相比于这种做法带来的性能开销，保留和自动释放对象所增加的额外开销是很不值得的。

方法 2

和方法 1 一样，方法 2 也使用了自动释放技术，但这次是在 setter 方法中完成：

```
- (NSString*) title {

    return title;

}



- (void) setTitle: (NSString*) newTitle {

    [title autorelease];

    title = [newTitle retain]; // Or copy, depending on your needs.

}
```

在 getter 比 setter 调用更频繁的情况下，方法 2 的性能明显好于方法 1。

方法 3

方法 3 完全没有使用自动释放技术：

```
- (NSString*) title {

    return title;

}



- (void) setTitle: (NSString*) newTitle {
```

```
    if (newTitle != title) {

        [title release];

        title = [newTitle retain]; // Or copy, depending on your needs.

    }

}
```

方法 3 所采用的方式非常适合于 getter 和 setter 方法被频繁调用的情况。它也非常适合于那些不希望延长值的生命周期的对象，比如集合类。它的缺点是，旧的值可能被立即回收（如果没有其他的所有者），如果另一个对象持有指向该值的非所有性引用，那么这将导致一个问题。例如：

```
NSString *oldTitle = [anObject title];

[anObject setTitle:@"New Title"];

NSLog(@"Old title was: %@", oldTitle);
```

如果 `anObject` 是拥有原始标题字符串的唯一对象，那么该字符串会在新的标题被设置之后被回收。随后，日志语句将引起程序崩溃，因为 `oldTitle` 是一个已被释放的对象。

值对象和复制

在 Objective-C 代码中，复制值对象–表示属性的对象–是一种很普遍的做法。C-类型的变量通常可以取代值对象，但值对象具有封装常用操作的优势。例如，`NSString` 对象被用来代替字符指针，因为它们封装了编码和存储。

当值对象作为方法的参数被传递或者从一个方法被返回时，通常会使用对象的副本而不是对象本身。例如，请仔细思考下面的方法，该方法将一个字符串赋值给对象的 `name` 实例变量。

```
- (void)setName:(NSString *)aName {

    [name autorelease];

    name = [aName copy];

}
```

存储 `aName` 的一个副本，其效果是产生一个独立于原始对象，但与原始对象具有相同内容的对象。副本的后续变化不会影响原始对象，并且原始对象的变化也不会影响副本。类似地，一种常见的做法是返回实例变量的副本，而不是实例变量本身。例如，下面的方法返回 `name` 实例变量的一个副本：

```
- (NSString *)name {

    return [[name copy] autorelease];
```

```
}
```

实现对象复制

本文介绍了实现 NSCopying 协议中的 copyWithZone:方法的两种方式，都可以达到复制对象的目的。

有两种基本方式可以通过实现 NSCopying 协议的 copyWithZone:方法来创建副本。您可以使用 alloc 和 init...，也可以使用 NSCopyObject。要选择一种更适合于您的类的方式，您需要考虑以下问题：

- 我需要深拷贝还是浅拷贝？
- 我从超类继承 NSCopying 的行为了吗？

这些内容将在以下章节进行介绍。

深拷贝与浅拷贝

一般来说，复制一个对象包括创建一个新的实例，并以原始对象中的值初始化这个新的实例。复制非指针型实例变量的值很简单，比如布尔，整数和浮点数。复制指针型实例变量有两种方法。一种方法称为浅拷贝，即将原始对象的指针值复制到副本中。因此，原始对象和副本共享引用数据。另一种方法称为深拷贝，即复制指针所引用的数据，并将其赋给副本的实例变量。

实例变量的 set 方法的实现应该能够反映出您需要使用的复制类型。如果相应的 set 方法复制了新的值，如下面的方法所示，那么您应该深拷贝这个实例变量：

```
- (void)setMyVariable:(id)newValue

{

    [myVariable autorelease];

    myVariable = [newValue copy];

}
```

如果相应的 set 方法保留了新的值，如下面的方法所示，那么您应该浅拷贝这个实例变量：

```
- (void)setMyVariable:(id)newValue

{

    [myVariable autorelease];

    myVariable = [newValue retain];
```

```
}
```

类似地，如果实例变量的 set 方法只是简单地将新的值赋给实例变量，而没有复制或保留它，那么您应该浅拷贝这个实例变量，正如下面的例子–虽然这通常很罕见：

```
- (void)setMyVariable:(id)newValue

{

    myVariable = newValue;

}
```

独立副本

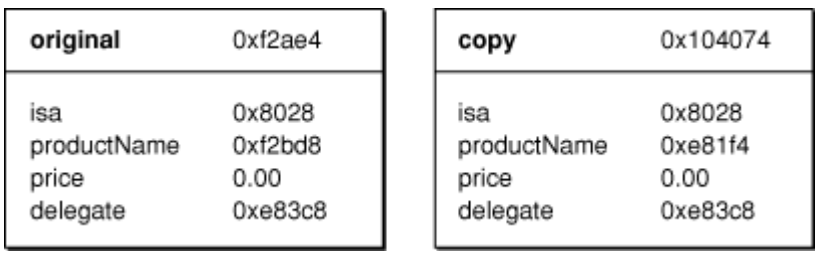
为了产生真正独立于原始对象的对象副本，必须对整个对象进行深拷贝。每个实例变量都必须被复制。如果实例变量本身具有实例变量，则它们也必须被复制，依此类推。在许多情况下，混合使用两种复制方式会更加有用。一般情况下，可以被视为数据容器的指针型实例变量往往被深拷贝，而更复杂的实例变量（如委托）则被浅拷贝。

```
@interface Product : NSObject <NSCopying>

{

    NSString *productName;

    float price;

    id delegate;

}



@end
```

例如，Product 类继承了 `NSCopying`。正如这个接口中所声明的，Product 实例含有名称，价格和委托三个变量。

复制 Product 实例会产生 `productName` 的一份深拷贝，这是因为它表示一个扁平的数据值。但是，`delegate` 实例变量是一个更复杂的对象，对于原始 Product 和副本 Product 都能够正常运行。因此，副本和原始对象应该共享这个委托。图 1 表示了 Product 实例及其副本在内存中的映像。

图 1 浅拷贝与深拷贝实例变量



productName 的不同指针值说明，原始对象和副本各自都有自己的 productName 字符串对象。而 delegate 的指针值是相同的，这表明这两个 product 对象共享同一个对象作为它们的委托。

从超类继承 NSCopying

如果超类没有实现 NSCopying，则您的类的实现必须复制它所继承的实例变量，以及那些在您的类中声明的实例变量。一般来说，完成这一任务的最安全的方式是使用 alloc,init...和 set 方法。

另一方面，如果您的类继承了 NSCopying 的行为，并声明了额外的实例变量，那么您也需要实现 copyWithZone:。在这个方法中，调用超类的实现来复制继承的实例变量，然后复制其他新声明的实例变量。您要如何处理新的实例变量，取决于您对超类的实现的熟悉程度。如果超类使用了或者有可能使用过 NSCopyObject，那么您必须有别于使用 alloc 和 init...函数的情况，用不同的方式处理实例变量。

使用“alloc, init...”方式

如果一个类没有继承 NSCopying 行为，则您应该使用 alloc, init...和 set 方法来实现 copyWithZone:。例如，对于“独立副本”中提到的 Product 类，它的 copyWithZone:方法可能会采用以下方式实现：

```
- (id)copyWithZone:(NSZone *)zone

{

    Product *copy = [[[self class] allocWithZone: zone]

            initWithProductName:[self productName]

            price:[self price]];

    [copy setDelegate:[self delegate]];



    return copy;

}
```

由于与继承实例变量相关的实现细节被封装在超类中，因此，一般情况下最好通过 alloc, init... 的方式实现 NSCopying，这样一来就可以使用 set 方法中实现的策略来确定实例变量所需的复制类型。

使用 NSCopyObject()

如果一个类继承了 NSCopying 行为，则您必须考虑到超类的实现有可能会使用 NSCopyObject 函数。NSCopyObject 通过复制实例变量的值而不是它们指向的数据，来创建对象的浅拷贝。例如，NSCell 的 copyWithZone:实现可能按照以下方式定义：

```
- (id)copyWithZone:(NSZone *)zone

{

    NSCell *cellCopy = NSCopyObject(self, 0, zone);

    /* Assume that other initialization takes place here. */



    cellCopy->image = nil;

    [cellCopy setImage:[self image]];



    return cellCopy;

}
```

在上面的实现中，`NSCopyObject` 创建了原始 cell 对象的一份浅拷贝。这种行为适合于复制非指针型的实例变量，以及指向浅拷贝的非保留数据的指针型实例变量。指向保留对象的指针型实例变量还需要额外的处理。

在上面的 `copyWithZone:`例子中，`image` 是一个指向保留对象的指针。保留 image 的策略反映在下面 `setImage:`存取方法的实现中。

```
- (void)setImage:(NSImage *)anImage

{

    [image autorelease];

    image = [anImage retain];

}
```

请注意，`setImage:`在重新对 `image` 赋值之前自动释放了 `image`。如果上述 `copyWithZone:`的实现没有在调用 `setImage:`之前，显式地将副本的 `image` 实例变量设置为空，则副本和原始对象所引用的 image 将被释放，而没有被保留。

即使 `image` 指向了正确的对象，在概念上它也是未初始化的。不同于使用 `alloc` 和 `init...`创建的实例变量，这些未初始化的变量的值并不是 `nil` 值。您应该在使用它们之前显式地为这些变量赋予初始值。在这种情况下，`cellCopy` 的 `image` 实例变量应该被设置为空，然后使用 `setImage:`方法对它进行设置。

`NSCopyObject` 的作用会影响到子类的实现。例如，`NSSliderCell` 的实现可能会按照下面的方式复制一个新的 `titleCell` 实例变量。

```
- (id)copyWithZone:(NSZone *)zone

{
```

```
    id cellCopy = [super copyWithZone:zone];

    /* Assume that other initialization takes place here. */



    cellCopy->titleCell = nil;

    [cellCopy setTitleCell:[self titleCell]];



    return cellCopy;

}
```

其中，假设 `super` 的 `copyWithZone:`方法完成这样的操作：

```
id copy = [[[self class] allocWithZone: zone] init];
```

超类的 `copyWithZone:`方法被调用，以复制继承而来的实例变量。当您调用超类的 `copyWithZone:`方法时，如果超类的实现有可能使用 `NSCopyObject`，则假定新的对象的实例变量是未初始化的。您应该在使用这些实例变量之前显式地为它们赋值。在这个例子中，在 `setTitleCell:`被调用之前，`titleCell` 被显式地设置为 `nil`。

当使用 `NSCopyObject` 时，对象的保留计数的实现是另一个应该考虑的问题。如果一个对象将它的保留计数存储在一个实例变量中，则 `copyWithZone:`的实现必须正确地初始化副本的保留计数。图 2 说明了这个过程。

图 2 复制过程中引用计数的初始化

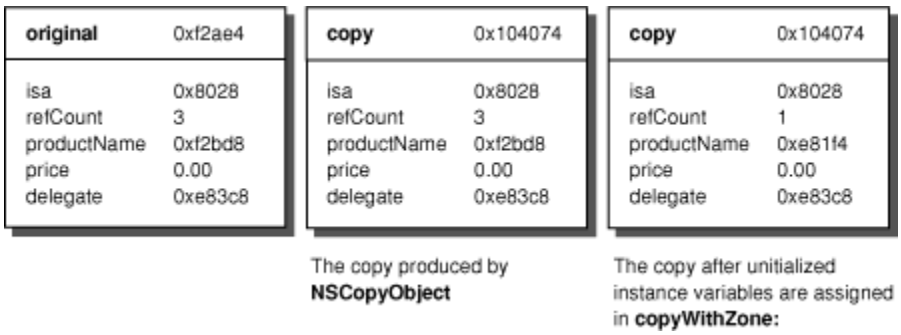


图 2 中的第一个对象表示内存中的一个 Product 实例。`refCount` 中的值表明该对象已经被保留了 3 次。第二个对象是通过 `NSCopyObject` 产生的 Product 实例的一份副本。它的 `refCount` 值与原始对象一致。第三个对象表示从 `copyWithZone:` 返回的一份副本，它的 `refCount` 已被正确地初始化。`copyWithZone:`在使用 `NSCopyObject` 创建了副本之后，它将 `refCount` 实例变量的值赋为 1。`copyWithZone:`的调用者隐式地保留了该副本，并负责释放它。

复制可变的和不可变的对象

当“不可变的和可变的”这一概念应用于某个对象时，不论原始对象是不可变的还是可变的，`NSCopying` 总是产生不可变的副本。不可变的类可以非常有效地实现 `NSCopying`。由于不可变的对象不会发生改变，因此没有必要复制它们。相反，`NSCopying` 可以实现为 `retain` 原始对象。例如，对于一个不可变的字符串类，`copyWithZone:`可以按照下列方式实现。

```
- (id)copyWithZone:(NSZone *)zone {

    return [self retain];

}
```

要使用 NSMutableCopying 协议创建对象的可变副本。为了支持可变的复制，对象本身并不需要是可变的。该协议声明了 mutableCopyWithZone:方法。通常，可变的复制是通过 NSObject 的便捷方法 mutableCopy 调用的，该方法调用了默认 zone 的 mutableCopyWithZone:。

Cocoa 中 Core Foundation 对象的内存管理

许多 Core Foundation 对象和 Cocoa 实例可以简单地相互进行类型转换，比如 CFString 和 NSString 对象。本文介绍如何管理 Cocoa 中的 Core Foundation 对象。有关对象所有权的一般信息请参考"对象的所有权和销毁"。

重要：本章介绍了 Cocoa 和 Core Foundation 对象在引用计数环境中的用法。如果您使用垃圾回收机制—见*垃圾回收编程指南*，则语义有所不同。

Core Foundation 对象的内存分配策略是，您需要释放那些由名字中包含"Copy"或"Create"的函数返回的值；您不应该释放那些由名字中不包含"Copy"或"Create"的函数返回的值。

Core Foundation 对象和 Cocoa 使用的公约非常相似，而且由于分配（allocation）/保留（retain）/释放（release）的实现是兼容的—每种环境中等价的函数和方法可以混合使用。因此，

```
NSString *str = [[NSString alloc] initWithCharacters: ...];

...

[str release];
```

等同于

```
CFStringRef str = CFStringCreateWithCharacters(...);

 ...

CFRelease(str);
```

和

```
NSString *str = (NSString *)CFStringCreateWithCharacters(...);

 ...

[str release];
```

和

```
 NSString *str = (NSString *)CFStringCreateWithCharacters(...);
```

```
...

[str autorelease];
```

正如这些代码示例所示，一旦被创建，类型转换对象可以被视为 Cocoa 或 Core Foundation 对象，而且，在每种环境中看起来都是“本地”的。

Nib 对象的内存管理

在 Cocoa 应用程序运行时生命周期中，会有一个或多个 nib 文件被加载，而且它们所包含的对象会被解压。当不再需要它们时，谁来负责释放这些对象取决于您在哪种平台进行开发，而且，如果是在 Mac OS X 的话，还取决于您的 File's Owner 继承自哪个类。

关于 nib 文件及其内存管理语义的基本讨论，以及与 nib 相关的术语的定义，比如插座对象，File's Owner 和顶层对象，请参考*资源编程指南*中的“Nib 文件”一节。

插座对象

当一个 nib 文件被加载且插座对象被建立的时候，如果存取方法（在 Mac OS X 和 iOS 上都）存在，则 nib 加载机制一定会使用存取方法。因此，无论您在哪个平台上进行开发，一般情况下，您都应该使用 Objective-C 的属性声明功能声明插座对象。

声明插座对象的一般形式应该是：

```
@property (attributes) IBOutlet UserInterfaceElementClass *anOutlet;
```

插座对象的行为取决于平台的类型（请参考“Mac OS X”和“iOS”），因此，实际的声明会有所不同：

- 对于 Mac OS X，您应该使用：

  ```
  @property (assign) IBOutlet UserInterfaceElementClass *anOutlet;
  ```

- 对于 iOS，您应该使用：

  ```
  @property (nonatomic, retain) IBOutlet UIUserInterfaceElementClass *anOutlet;
  ```

接下来，您应该做的是合成相应的存取方法，或者是根据声明实现这些方法，（在 iOS 平台）还要在 `dealloc` 中释放相应的变量。

如果您使用现代运行时并合成实例变量，这种模式同样适用，因此它能够在所有情况下保持一致性。

Mac OS X

在默认情况下，nib 文件的 File's Owner 负责释放 nib 文件中的顶层对象和 nib 中的对象创建的所有非对象资源。对象图中的根对象的释放会促使所有依赖它的对象也被释放。对应用程序的主 nib 文件（其中包含应用程序菜单和其他可能的项目）来说，它

的“文件所有者”是全局应用程序对象 NSApp。然而，当 Cocoa 应用程序终止时，主 nib 中的顶层对象不会因为 NSApp 被回收（请参考“回收对象”）而自动获得 dealloc 消息。换句话说，即使是在主 nib 文件中，您也必须管理顶层对象的内存。
应用程序套件提供了一系列功能来帮助您确保 nib 对象能够被正确地释放：

- NSWindow 对象（包括面板）有一项 isReleasedWhenClosed 属性，如果该属性被设为 YES，则窗口会在关闭的时候释放它自己（和它的视图层次中所有相关的对象）。在 Interface Builder 中，您可以通过勾选检视器的属性面板上的“在关闭时释放”复选框来设置此选项。

- 如果 nib 文件的“文件所有者”是一个 NSWindowController 对象（它是基于文档的应用程序中的文档默认 nib–前面介绍过，NSDocument 负责管理 NSWindowController 实例），它会自动销毁它所管理的窗口。

因此，在一般情况下，您负责释放 nib 文件中的顶层对象。但实际上，如果您的 nib 文件的所有者是一个 NSWindowController 实例，则它会为您释放顶层对象。如果您的某个对象加载了 nib 本身（而且它的所有者不是 NSWindowController 实例），则您可以为每一个顶层对象定义插座对象，以便您可以在适当的时候使用这些引用释放它们。如果您不希望为所有的顶层对象定义插座对象，那么您可以使用 NSNib 类的 instantiateNibWithOwner:topLevelObjects:方法来获得一个包含 nib 文件顶层对象的数组。

当您考虑到应用程序的各种类型时，销毁 nib 对象的责任问题就变得更清晰了。大多数 Cocoa 应用程序归属于两类：单一窗口应用程序和基于文档的应用程序。在这两种情况下，系统会在一定程度上为您自动处理 nib 对象的内存管理。在单一窗口应用程序中，主 nib 文件中的对象在应用程序的整个运行周期内持续存在，并在应用程序终止时被释放；但是，系统并不保证在应用程序终止时自动地对主 nib 文件中的对象调用 dealloc。在基于文档的应用程序中，每一个文档窗口都由一个 NSWindowController 对象管理，该对象处理文档 nib 文件的内存管理。

有些应用程序可能具有更复杂的 nib 文件和顶层对象的分布。例如，应用程序可能有多个 nib 文件，并带有多个窗口控制器，可加载的面板和检视器。但是，在大多数情况下，如果您使用 NSWindowController 对象来管理窗口和面板，或者如果您设置了“在关闭时释放”这一窗口属性，那么系统会为您自动处理大部分的内存管理。如果您决定不使用窗口控制器，也不想设置“在关闭时释放”属性，则您应该在窗口关闭时显式地释放您的 nib 文件的窗口和其他顶层对象。此外，如果您的应用程序使用检视器面板，那么（在延迟加载之后）该面板通常应该在整个应用程序生命周期内持续存在–没有必要销毁检视器和它的资源。

iOS

顶层对象

当 nib 文件中的对象被创建时，它们的保留计数为 1，而且随后它们会被自动释放。由于重建了对象层次结构，UIKit 会使用 setValue:forKey:重新建立对象之间的连接，setValue:forKey:使用现有的 setter 方法，如果没有 setter 方法可用，那么它会默认保留这个对象。这就意味着（假设您遵循“插座对象”中所示的模式）*具有插座对象的任何对象*都保持有效。但是，如果存在您没有存储在插座对象中的*顶层*对象，则您必须保留 loadNibNamed:owner:options:方法返回的数组或数组中的对象，以防止这些对象过早地被释放。.

内存警告

当视图控制器收到内存警告（didReceiveMemoryWarning）的时候，它应该释放那些当前不需要的，和那些如果有需要可以稍后重新创建的资源的所有权。其中一种资源就是视图控制器的视图本身。如果它没有父视图，则该视图应该被销毁（在它的 didReceiveMemoryWarning 实现中，UIViewController 调用[self setView:nil]）。

但是，由于 nib 文件中的元素的插座对象通常会被保留（见“插座对象”），因此，即使主视图被销毁，如果没有采取任何进一步的行动，插座对象也不会被销毁。这并不是其本身的问题–如果当主视图被重新载入时，它们只是被简单地替换掉–但这确实意味着，`didReceiveMemoryWarning` 的效果被削弱了。为了确保您正确地释放插座对象的所有权，在您的自定义视图控制器类中，您可以实现 `viewDidUnload` 来调用您的存取方法，以便将插座对象设置为 `nil`。

```
- (void)viewDidUnload {

    self.anOutlet = nil;

    [super viewDidUnload];

}
```

注意：在 iOS 3.0 版本之前，`viewDidUnload` 方法是不可用的。相反，您应该在 `setView:`中将插座对象设置为 `nil`，正如该例所示：

```
- (void)setView:(UIView *)aView {

    if (!aView) { // View is being set to nil.

        // Set outlets to nil, e.g.

        self.anOutlet = nil;

    }

    // Invoke super's implementation last.

    [super setView:aView];

}
```

此外，由于 `UIViewController` 中的 `dealloc` 的一个实现细节，您还应该在 `dealloc` 中将插座对象变量设置为 `nil`：

```
- (void)dealloc {

    // Release outlets and set outlet variables to nil.

    [anOutlet release], anOutlet = nil;

    [super dealloc];

}
```