# 第 7 章　内存管理——打理傲娇程序的超级拖把

周末，忙里偷闲地自己当了大厨一回，买菜洗菜烧菜，弄了满满一桌丰盛佳肴好不开心。和家人吃毕后，我们会怎么去处理饭桌上的残留呢？该洗的洗，该存的存，该倒的倒，把家里的状态打理成做饭前一样干净，必要的清理工作就算完成了。

又一次周末，嫌自己弄太麻烦太劳累，这次决定上饭店去尽情一翻，点了满满一桌丰盛佳肴好不热闹。享受完后，我们又会怎么去处理桌上的残留呢？拿抹布擦拭桌面？把碗筷归类？送去饭店的清洗间？说不定饭店老板不但不会谢谢你，还生怕你把他们家的餐具给敲碎呢。其实，在饭店吃完后，我们只需拍拍屁股走人便成，剩下收拾的事情，饭店清洁人员自然会处理干净的。

Objective-C 的内存管理机制，其实和上述的两种情况差不多，即自己申请内存自己释放（家里吃），自己申请内存由自动释放池释放（上饭店）。家里吃饭虽然麻烦，但是有着自由和省钱的优势，以 Objective－C 观点来看的话，就是想怎么用怎么用，想何时释放就何时释放。而饭店吃虽然贵些，但有着诱人的重要优点：省事。谁不想肆意而为，让他人跟后面收拾残局呢？

本章，我们以吃饭为始，细细品位！

## 7.1　内存管理机制——出色程序的重要资本

内存管理向来是高级编程语言的重要知识点，起到承前启后的作用。而 Objetive-C 作为 C 语言的超类，有着很多和 C 语言相通的特性。如果曾经有使用过 C 语言的话，会对这章内容的深刻理解很有帮助。不过没接触过 C 也没关系，毕竟 Objective-C 是一门面向对象的高级编程语言，这章会将内存管理的世界淋漓尽致地展现出来。

### 7.1.1　内存的创建和释放

让我们以 Objective－C 世界中最最简单的申请内存方式展开谈谈关于一个对象的生命周期。首先，创建一个对象：

```
//“ClassName”是任何你想写的类名，比如 NSString，NSArray 等等一切随意
 id testObject = [[ClassName alloc] init];
```

> 小贴士 7-1：
>
> “alloc”是 Objective－C 中常用来申请内存块的方式。

此时，对于对象“testObject”来说，他的引用计数就是 1 了，原因是它调用了 alloc 来创建了一块只属于它的内存，这样对象的引用计数就得+1。 另外，Objective－C 中的另两个关键字 retain，copy 也会将对象的引用计数+1。根据 Ojbective－C 的内存管理机制，我们在使用完“testObject”后需要释放它：

```
[testObject release];
```

此时，”testObject”的引用计数再次为 0，他的生命周期，也就是他的使命结束，正式寿终正寝，Game Over！这就是一个对象的生命周期，精悍而强力！

对象的内存生命周期如图 7-1：

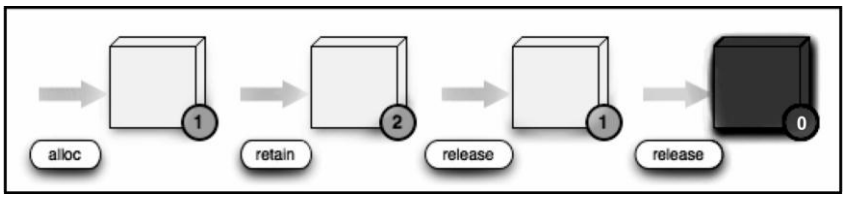


图 7-1 引用计数过程

对象 alloc 后引用计数变为 1，随后再次执行 retain，引用计数再次+1 变为 2。第三次则调用了 release 使得当前引用计数-1 又变回了 1，当再次 release 时，引用激素为 0 对象被成功释放。

小贴士 7-2：

release 后的对象，虽然已经释放，但是他的指针地址仍然存在，只是指向了一块已经释放且无用的内存。所以无论从安全释放的角度还是编码习惯上，都建议在 release 后直接赋个 nil 来置空，即：

```
[testObject release];
testObject = nil;
```

这样做仍然不算完全安全，试想在对 testObject 释放时，我们并不知 testObject 的引用计数是否已经是 0 了，如果对象引用计数已经是 0 则会造成双重释放的问题。既然这样，我们需要在释放的代码之前插入一段判断语句，即：

```
if (!testObject) {
    [testObject release];
    testObject = nil;
 }
```

好程序员都是懒人，面对纷繁的对象需要我们管理，对每个对象都写上三行释放代码实在是过于麻烦，于是，释放对象的宏出生了！即：

```
#define RELEASE(obj)      if(obj){[obj release]; obj = nil;}
```

调用一把此宏就执行一次安全释放，好用指数妥妥的！

小贴士 7-3：

一般，如果我们能知道需要实现功能所涉及的"ClassName"的具体类名，就不建议使用 init 方法去初始化，在苹果开发的 SDK 中，大部分的类都有自己一套初始化的 API，可供调用。比如以下代码：

```
NSString *strTest = [[NSString alloc] initWithString:@"test"];（推荐）
NSString *strTest = [[NSString alloc] init];（不推荐）
```

小贴士 7-4：

根据经验，如果我们找到一个类，想申请一块内存时用到了有“alloc”/“new”/“XXXCopy”/“CreateXXX”特征的这些方法时，这个创建出来的对象的引用计数一般会+1，他的释放时机需要由聪明的程序员自己来定夺，只要不忘记即可。

这些方法返回的对象是否真需要释放，最终还是以文档为准。

## 7.1.2 自动释放池和使用

除了上述的手动控制方式来管理内存使用外，苹果提供了另外一种强力的内存管理机制，即“自动释放池（NSAutoreleasePool）”。想必在之前的章节中，我们已经使用过这个机制，对于“自动释放池”有了一定了解。

所有运行在苹果 run time 环境的程序，都会在程序的主消息循环的自动释放池里面运作，也就是在 main()主函数中营造的那个自动释放池的闭包中，如下示例代码：

```
int main(int argc, char *argv[]) {
    //自动释放池
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

那么，我们平时一般怎么使用这个池子呢？

比如同样是 testObject，我们可以这么申请内存：

```
    id testObject = [[[ClassName alloc] init] autorelease];
```

以这种形式创建的 testObject，alloc 会把它的引用计数+1，autorelease 又会把它的引用计数-1，导致了 testObject 的引用计数为 0。

那很多朋友要问了，既然 testObject 的引用计数已经为 0，会不会在之后使用它时却已经给释放掉了？答案当然是不会！此时的 testObject 该怎么用还是怎么用。

只是朋友们要注意以下两点：

- 不再需要也不再允许去手动释放 testObject，因为它的引用计数已经为 0 了。
- testObject 出了它的作用域，一般就会给“自动释放池”回收，不能再使用了。

这个就是让我们开发者最省力的地方，你只需要管好对象的创建位置和作用域，至于他的释放问题根本不用过问。

小贴士 7-5：

如果一个对象被加入自动释放池后，又以某个函数的返回值身份，返回了出去。那么他的作用域相应扩大到外面调用者那层，这个对象并不会因为 autorelease 而在返回出去的时候给系统自动回收。

比如下面的例子，并无内存泄漏问题，也不用考虑提前被系统回收问题，如预期般打印出字符串“testAutorelease”：

```
- (NSString*)testAutoReleasePool {
    NSString *test = [[NSString alloc] initWithString:@"testAutorelease"];
    return [test autorelease];
}

- (void)invokeTestMethod {
    NSString *testMethod = [self testAutoReleasePool];
    NSLog(@"%@", testMethod);
}
```

### 7.1.3 实例方法和类方法

说到自动释放池，我们就不得不提实例方法和类方法。什么叫实例方法？而什么又叫类方法呢？

简单来说，“实例方法”就是得有了具体实例（对象）后才能使用的方法，一般以减号形式作为方法的前缀来声明，比如:

```
API 定义：
- (id)initWithString:(NSString *)aString
应用例子：
NSString *strTest =    [[NSString alloc] initWithString:@"I'm an instance method"];
```

其中“initWithString”只能被具体的对象调用，例子中[NSString alloc]当场返回一个指向 NSString 对象的指针，理所当然可以调用“initWithString”这个实例方法了。

相比而言，“类方法”则无需具体对象，只需要类的名称即可调用，一般以加号形式作为方法的前缀来声明，比如：

```
API 定义：
+ (id)stringWithString:(NSString *)aString
应用例子：
NSString *strTest =    [NSString stringWithString:@"I'm an class method"];
```

其中“stringWithString”会为“strTest”对象创建一块内存，需要注意的是，这块内存一般已经被加到“自动释放池”中，因此用类方法创建出来的对象，不需要手动释放。

如果你有面向对象编程的经验，大可将类方法理解成静态方法，类方法在对象还没创建出来之前已经存在啦，生命力可比谁都强！

根据实例方法和类方法的描述，我们很容易理解如下两行代码其实是一样的：

```
Person personA = [[[Person alloc] init] autorelease];
Person personB = [Person person];
```

这两行代码做了相同的事情，即各自申请一块内存，然后将指向这块内存的指针放进自动释放池。personA 和 personB 唯一的区别是他们是两个指针。

### 7.1.4 保留（retain）对象

说到这里，很多朋友一定会觉得，“自动释放池”自有它先进的一套体系，但是它始终是丢给系统自动去回收的，控制权并不在我们手上，生怕系统意外地释放掉仍然需要使用的对象。

根据这个小小的忧虑，很容易就想到了如下场景：

```
@interface ClassName : NSObject{
    NSFileManager     *_fileMgr;
}
@end
@implementation ClassName
- (id)init {
    self = [super init];
     If (self) {
          //有没有问题？
         _fileMgr = [NSFileManager defaultManager];
    }
    return self;
}
```

```
- (void)dealloc {
    if (!_fileMgr) {
        [_fileMgr release];
        _fileMgr = nil;
    }
    [super dealloc];
}
@end
```

其中，_fileMgr 作为 ClassName 类的成员量，在 init 方法中进行初始化。上述代码中，_fileMgr 用了 NSFileManager 的类方法来创建对象。根据前两章描述的内容，类方法出了作用域即自动释放，所以 _fileMgr 虽然在创建时确实得到了有效的对象，但是当 init 方法完成后，_fileMgr 指向的内存势必被系统回收，成员量的全局可用性也名存实亡。

这时候，关键字“retain”就到了出场的时机。在成员量创建的时候，我们长臂一挥使劲把 retain 砸向它，将原来的代码改成如下的样子：

```
    _fileMgr = [[NSFileManager defaultManager] retain];
```

成员量被砸的晕头转向，立即将自己的引用计数+1。自此，即使变量出了他的生命周期，引用计数还是会剩下 1，仍然可以象普通的成员量那样使用自如。当然，别忘了将这个成员量释放代码添加到类的 dealloc 函数中去。

### 7.1.5 拷贝（copy）对象

同时，Objective－C 中还有另外一个重要的关键字 copy，对于那些实现了 NSCopying 协议的库来说，对创建出来的对象砸去一个 copy 关键字，对象的引用计数也会+1，对于避免系统池自动回收上来说，起到的效果是一样的，如下代码

```
NSString *strTestA  =  [[NSString stringWithString:@"Let's do copy"] copy];
```

注意：之前的成员量_fileMgr 无法使用 copy 关键字了，原因是 NSFileManager 此类并未实现 NSCopying 协议。

copy 和 retain 虽然犹如兄弟般你中有我我中有你，但是仍然有相当明显的区别。比如：

```
        NSMutableString *strOrigin = @"test";
        NSMutableString *strCopy = [strOrigin copy];
        NSMutableString *strRetain = [strOrigin retain];
```

三行代码执行完后，三个变量的地址变成如图 7-2 所示的样子：

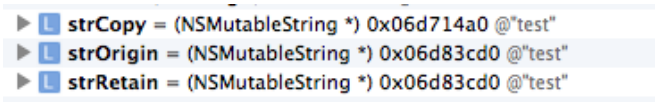


图 7-2 拷贝的变量地址

可见，strRetain 虽然将 strOrigin 保留了一把后引用计数增加了，指针仍然指向 strOrigin。此时 strCopy 却是将 strOrigin 拷贝了一份新的，指针不再和 strOrigin 有任何瓜葛，我们需要维护的是一个全新的指针！

所以，如果我们只是想把当前对象的值记录下来，之后对于原本指针指向对象的变动不再过问，那 copy 的可用优势相当明显。

小贴士 7-6：

为什么上面的例子中是 NSMutableString 而不以 NSString 来举例呢？

NSMutableString 和 NSString 同样实现了对 NSCopying 协议的实现，但是实现的内容稍有区别。对于 NSString 来说，它其实是一个不可变的常量字符串，苹果引入了“享元模式（flyweight）”，即尽可能的少用内存开销，对于某些类来说，即使使用了 copy，返回的指针仍然和原对象相同，NSString 就属于其中一类。

### 7.1.6 浅拷贝和深拷贝

假设有一个数组 NSArray *arrCopy，它有三个 NSString 类型的元素，当你要去拷贝这个数组的时候，你会如何拷贝？

如果直接申明一个新的数组 arrCopyDest，并且使用如下方法：

```
arrCopyDest = [arrCopy copy];,
```

这个新的数组的指针地址确实为全新的，里面的三个元素也都还在，表面看上去这样做并无任何问题。其实，新的数组只是拷贝了原数组的指针，但里面的元素完全没有进行拷贝，arrCopyDest 中三个 NSString 元素仍然指向的是原来 arrCopy 中的三个元素。这就是所谓的浅拷贝，一种治标不治本式的“表面拷贝”。真正的深拷贝是不仅对象本身，还包括对象内的对象都需要执行拷贝的操作。

关于深拷贝，下面实现了数组（NSArray）和字典（NSDictionary）的深拷贝具体做法，其他基本库的深拷贝做法类似：

首先是 NSDictionary 的深拷贝，代码如下：

```
#import "NSDictionary+deepCopy.h"
@implementation NSDictionary (deepCopy)
- (NSMutableDictionary*)mutableDeepCopy {
    //申请一个新的字典
     NSMutableDictionary *dictReturn    = [NSMutableDictionary dictionaryWithCapacity:self.count];
    NSArray                    *arrKeys       = [self allKeys];

    //原字典循环
    for (id key in arrKeys)   {
        id oneObj     = [self objectForKey:key];
        id oneCopy    = nil;

        //没有获得元素
        if (!oneObj) {
            continue;
        }

        //如果元素实现了：mutableDeepCopy
        if ([oneObj respondsToSelector:@selector(mutableDeepCopy)]) {
            oneCopy = [oneObj mutableDeepCopy];
        }
        //如果元素实现了：mutableCopy
        else if([oneObj respondsToSelector:@selector(mutableCopy)]) {
            oneCopy = [oneObj mutableCopy];
        }
        //如果元素实现了 copying 协议
```

```
            else if([oneObj conformsToProtocol:@protocol(NSCopying)]){
                oneCopy = [oneObj copy];
            }
            //如果元素无法进行 copy
            else {
                NSLog(@"ClassName:[%@] couldn't be copied", [oneObj class]);
                continue;
            }

            [dictReturn setObject:oneCopy
                           forKey:key];
        }

        return dictReturn;
}
```

关于数组的深拷贝，代码如下：

```
#import "NSArray+deepCopy.h"
@implementation NSArray (deepCopy)
- (NSMutableArray*)mutableDeepCopy {
        //申请一个新的数组
         NSMutableArray    *arrReturn    = [NSMutableArray arrayWithCapacity:self.count];

        //原数组循环
        for (id obj in self)    {
            id oneCopy    = nil;

            //如果元素实现了：mutableDeepCopy
            if ([(NSObject*)obj respondsToSelector:@selector(mutableDeepCopy)]) {
                oneCopy = [obj mutableDeepCopy];
            }
            //如果元素实现了：mutableCopy
            else if([(NSObject*)obj respondsToSelector:@selector(mutableCopy)]) {
                oneCopy = [obj mutableCopy];
            }
            //如果元素实现了 copying 协议
            else if([obj conformsToProtocol:@protocol(NSCopying)]){
                oneCopy = [obj copy];
            }
            //如果元素无法进行 copy
            else {
                NSLog(@"ClassName:[%@] couldn't be copied", [obj class]);
                continue;
            }

            [arrReturn addObject:oneCopy];
        }

        return arrReturn;
}
```

实现 NSCopying 的常用类如表 7-1 所示：

表 7-1　实现NSCopying的常用类

| 不可变类型 | 可变类型 |
| --- | --- |
| NSString | NSMutableString |
| NSArray | NSMutableArray |
| NSDictionary | NSMutableDictionary |
| NSData | NSMutableData |
| NSSet | NSMutableSet |

### 7.1.7　自动保留（retain）

retain 的使用方法我们已经知悉，而有些类的 API，会在内部对传入的参数自动进行 retain 操作。这些类的独特特性，对 Objective－C 的初学者造成很大困扰。

以如下代码为例：

```
UIView *rootView      = [[[UIView alloc] init] autorelease];
UIView *subView       = [[UIView alloc] init];

[rootView addSubview:subView];
[subView release];
```

各位兄台所见不错，上述代码中的 subView 在最后句竟然被释放了！难道 rootView 作为父视图还能对释放掉的子视图进行管理和操作么？

其实，这样做毫无问题，原因就在于：UIView 这个类的 addSubview 方法会对子视图进行 retain 操作，致使之后即使 subView 被释放了，subView 的引用计数仍然剩下 1，不会被系统回收掉，所以 rootView 中的那个 subView 仍然还活奔乱跳的。

而在释放父视图 rootView 时，只需要对 rootView 发送释放的消息调用 release 即可，rootView 的所有子视图均会按前后次序逐个释放出内存。

> 小贴士 7-7
>
> 有些集合类比如 NSMutableArray / NSMutableDictionary 等，当使用那些用于“添加或者修改元素”的 API 时（比如 addObject / insertObject / setObject：ForKey：等），会有自动保留的操作执行。

### 7.1.8　其他创建和释放对象的方式

Objective－C 提供了许多方式来创建新对象，具体的方式会随着不同类不同框架而不同，无法一概而论。如第一节所说，如果创建对象的方法名中有以“alloc”/“new”/“XXXCopy”/“CreateXXX”来命名的特征，这个创建出来的对象的引用计数就会+1，和类方法创建的对象不同，我们需要负责经由我们手使得引用计数增加的那些对象的释放工作。

而作为早期苹果开发框架 Carbon 的基础库 Core Foundation，我们必须使用 C 语言调用函数的风格来调用它的 API。

举个例子来说，Carbon 中也有一个字符串的类：CFStringRef，其创建方法为：

```
CFStringRef str       = NULL;
const UInt8 cTest[] = {5, 'H', 'e', 'l', 'l', 'o'};

str = CFStringCreateWithPascalString(NULL, cTest, kCFStringEncodingMacRoman);
```

释放的方法为：

```
CFRelease(str);
```

和 cocoa 框架的 release 方法相同，我们依然可以自己制作一个适用于 carbon 框架对象的释放宏用来释放对象，如下：

```
#define SAFE_RELEASE_CARBON(obj)  if(obj){ CFRelease(obj); obj = NULL;}
```

而在 iOS 和 MacOS 的图像编程开发中，我们将会遇到一个常见面的老朋友：Quartz 库。其中的大部分的 API 也都是有着 C 语言风格。Quartz 库的使用本书也会有所涉及，之后会有所详解。

## 7.2　单例模式

单例模式是一种代表着坚定不移，唯我独尊的霸气模式！

如果常常使用面向对象开发语言编程，你一定对单例模式这个名词耳熟能详。如果没有听说过也没关系，看了本节后就会认识 Objective－C 中的单例模式。

单例对象一旦出生了，就成为了不死之物。除非运行时（runtime）毁灭，否则它能改变的最多也就是它的认知，性格和三观。

换句话说，单例的对象只能修改无法释放，直到程序结束。

让我们想想，Objective－C 中平时我们有没有碰到过单例模式的库呢？其实，cocoaTouch 和 cocoa 的库中，存在着大量的单例模式。不知有没有记得，很多类的使用直接调用一句 shareXXX 的类方法就出来一个强大的对象，这些类大部分都是使用了单例模式。比如

```
[UIApplication sharedApplication]
[NSFileManager defaultManager]
[NSWorkspace sharedWorkspace]
```

有没有一下子恍然大悟的感觉？这些单例模式如果要实现的话，代码应该如何写才好呢？不用着急，让我们一步一步来：

（1）首先单例一旦创建，是永远存在于内存中的。所以需要创建一个全局量：

```
static MySingletonClass *sharedSingletonObj = nil;
```

（2）既然是单例，一定有一个构造方法直接忽略跳过实例对象的生成过程。据此看来“类方法”最合适不过了：

```
+ (MySingletonClass*)sharedSingleton
    {
  //多线程安全的关键字，相关概念可以参考多线程编程章节
  @synchronized(self) {
                //创建
          if (sharedSingletonObj == nil) {
                sharedSingletonObj = [[super allocWithZone:NULL] init];
           }
                //发出必要警告
```

```
            else{
              NSLog(@"单例对象已经存在！");
                }
          }

      return sharedSingletonObj;
 }
```

（3）如果对 sharedSingletonObj 执行了 copy 呢？我们需要重写 copy 方法：

```
- (id)copyWithZone:(NSZone *)zone {
      return self;
}
```

（4）如果对 sharedSingletonObj 执行了 retain 呢？我们同样需要重写 retain 方法：

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

（5）继续，release 和 autorelease 的方法重写：

```
- (void)release {
}

- (id)autorelease {
    return self;
}
```

（6）重要的一点！我们需要实现 NSObject 里面，关于引用计数 API 的重写以避免因为引用计数为 0 导致 dealloc 的触发：

```
- (NSUInteger)retainCount {
      //是一个无限大的 int 数，避免了系统自动触发单例的 dealloc 方法
      //也可以明确告知调用者，此为单例对象。
      return NSUIntegerMax;
}
```

（7）最后，你会发现就这样让人使用的话，如果不通过类方法创建对象转而调用 alloc 创建，则每次会分配新内存且引用计数+1！显然 alloc 方法势必也需要重写：

```
+ (id)allocWithZone:(NSZone *)zone {
      //直接套用 sharedSingleton，  retain 符合 alloc 惯例，
      //使类方法返回的对象的引用计数+1，此处 retain 根据上面的重写内容，不做任何事情。
      return [[MySingletonClass sharedSingleton] retain];
 }
```

以下是单例模式的全部代码实现：

```
static MySingletonClass *sharedSingletonObj = nil;

+ (MySingletonClass*)sharedSingleton {
    @synchronized(self) {
          //创建
        if (sharedSingletonObj == nil) {
            sharedSingletonObj = [[super allocWithZone:NULL] init];
        }
        else{
            NSLog(@"单例对象已经存在！");
```

```
            }
        }
    return sharedSingletonObj;
}

+ (id)allocWithZone:(NSZone *)zone {
    return [[MySingletonClass sharedSingleton] retain];
}

- (id)copyWithZone:(NSZone *)zone {
    return self;
}

- (id)retain {
    return self;
}

- (void)release {
}

- (id)autorelease {
    return self;
}

- (NSUInteger)retainCount {
    return NSUIntegerMax;
}
```

小贴士 7-8：

我们需要注意的是：往往只需要一个单例对象而已，但是如果仍然想用 alloc 和 init 创建这个类的其他对象，那上述写法中，allocWithZone：之后的所有方法，我们不要重写即可。

## 7.3 取值方法(getter)和赋值方法(setter)

在 Objective－C 2.0 规范推出之前，苹果开发的程序员光在类成员量的管理上就非常可怜！根据编码规范，他们需要为每一个可供外部存取的变量实现 setter 和 getter 方法，一旦碰上超大的类，维护起来何止一个“痛苦”可以描述。

然而万事开头难，苹果也发现了 Objective－C 1.0 的挫样，直接在 2.0 中加入了属性的特性。属性以 @property 的形式展开，直接自动生成对象的存取方法。使用时，直接 self.XXX（属性变量名）就能获取相应的变量值。这样，不仅省去了程序员烦人的复制粘贴，提高成员量管理的合理性和健壮度，也使得代码的可读性大大增强。

不过，在研究 Objective－C 的属性之前，我们有必要先去历史看看，以前是怎么做的？

### 7.3.1 取值方法和赋值方法

如果我们拥有一个类 A，他有一个成员变量 int B。在这个类 A 的外部，如果我们要访问其成员变量 B，我们应该如何去访问？

如果根据 C++的做法，我们一般会使用类 A 的对象 ObjectA->B 来访问 B 的变量，但是这么使用有一个前提：变量 B 是 public 的类型。而在 Objective－C 中如果没有明确的指定，所有的变量均为 protected 类型，即外部不可直接访问，继承的子类可以访问。

所以，为了访问内部的保护变量，我们会有两个方案可供选择：

- 对 int B 申明为 public 类型
- 增加一个函数，来当作返回 B 值的接口。

对于这两种解决方案，应该选择哪一种更理想呢？首先来看看每个方案的优点：

方案 1：更符合 C++的编程习惯

方案 2：对类进行了封装，保护了成员量的访问。如果另外还需要提供赋值的接口，这样做又可以使成员量赋值受自己类控制以避免被外部的胡乱修改。甚至在 MacOSX 编程中和 Binding 技术相辅相成。

显然根据分析方案 2 是我们的首选，具体实现如下：

```
@interface A : NSObject {
    int B;
}
- (void)setB:(int)newB;
- (int)B;
@end

@implementation A
- (void)setB:(int)newB {
    B = newB;
}

- (int)B {
    return B;
}
@end
```

上面的代码示例只是简单类型成员变量的存取，接下来是以对象存在的成员变量的存取，以任意对象类型为例，新的类 A 的申明如下：

```
@interface A : NSObject
{
    int          B;
    id           classObj;
}

- (void)setB:(int)newB;
- (int)B;

- (void)setClassObj:(id)aClassObj;
- (id)classObj;

@end
```

对于 classObj 变量的取值方法，直接把成员量“classObj”丢出去即可。

我们需要注意的一定是 classObj 的赋值方法，具体实现如下：

```
- (void)setClassObj:(id)aClassObj;
{
    //参数检查
    if (aClassObj == nil || classObj == aClassObj) {
        return;
    }

    //释放原对象
    if (classObj != nil) {
        [classObj release];
        classObj = nil;
    }

    //赋值
    classObj = [aClassObj retain];
}
```

如果新对象和老对象是同个指针的话，明显是外部的调用者在忽悠我们，需要以明确的不做任务事情的态度去反驳他的胡闹！

否则根据正常来说，首先就需要清空原对象的内容来准备好迎接自己的新生！既然他是一个类中的成员变量，最后一句势必要 retain 一把，这样就能避免那个“aClassObj”哪天又不知道出什么岔子给释放了，我们的成员对象给连累。

### 7.3.2 属性关键字

一整套的 setter 和 getter 方法有没有让人很头疼？只有经历过那个黑暗的时代，我们才懂得珍惜现在的光明。光明从来都是与你我相伴，请看如下使用属性关键字@property 的例子：

```
@interface A : NSObject {
    int          B;
    id           _classObj;
}

@property (nonatomic, assign) int     B;
@property (nonatomic, retain, setter=setMyBeatifulObj:) id       classObj;

@end

@implementation A
@synthesize B;
@synthesize classObj = _classObj;

- (void)setMyBeatifulObj:(id)aClassObj;
{
    //参数检查
    if (aClassObj == nil || _classObj == aClassObj) {
        return;
```

```
    }

    //释放原对象
    if (_classObj != nil) {
        [_classObj release];
        _classObj = nil;
    }

    //赋值
    _classObj = [aClassObj retain];
}

@end
```

上面这段代码涵盖了属性的大部分特征，我们开始细细品位。

可以看到关键字@property 后的括号出现了四个特性关键字：non atomic, assign, retain, setter。这些关键字直接告诉编译器后面的变量用何种方式来存取。

这些关键字分为四类，具体如表 7-2 所示：

表 7-2　关键字分类

| 属性关键字 | 使用范围 | 释义 | 是否是默认值 | 小贴士 |
|---|---|---|---|---|
| assign | 赋值方式 | 不拷贝不保留，直接赋值 | ○ | 基本数据类型和本类不直接拥有的对象 |
| retain | 赋值方式 | 将新值保留一份赋覆盖原值 | × | 大部分对象可使用 |
| copy | 赋值方式 | 将新值拷贝一份赋覆盖原值 | × | 字符串选择性使用 |
| readwrite | 读写权限 | 生成getter，setter的两个方法 | ○ | 变量可读取可修改 |
| readonly | 读写权限 | 只生成getter方法 | × | 变量只读不可修改 |
| atomic | 原子性 | 原子操作 | ○ | 可以保证在多线程的环境下，能安全的存取值 |
| nonatomic | 原子性 | 非原子操作 | × | 不生成多线程同步内容 |
| getter | 存取方法 | 自定义取方法 | × | |
| setter | 存取方法 | 自定义赋值方法 | × | |

对于两个“property”进行分析：

1）

```
@property (nonatomic, assign) int     B;
```

这行代码生成关于 B 的 setter 和 getter 方法，非原子操作且不拷贝不保留，直接赋值。

对于属性变量 B 而言，使用方法如下：

int test = self.B; (整型变量 test 将被赋值成"B"的值)

self.B = 10; （"B"的值被赋值整型 10）

2）

```
@property (nonatomic, retain, setter=setMyBeatifulObj:) id       classObj;
```

生成关于 classObj 的 getter 方法，而 setter 方法 setMyBeatifulObj:需要自定义实现，

由于属性变量 classObj 和成员量_classObj 不同名，我们需要在@ synthesize 中将两者关联即可。

```
@synthesize classObj = _classObj;
```

属性 classObj 非原子，但是会以“保留(retain)”的形式赋值.

id aObject = self.classObj; (aObject 对象的指针指向我们的成员量_classObj)

self.classObj = (id)newObject；（_classObj 由系统自动清理，之后将 newOjbect retain 一把后赋给 _classObj）;

小贴士 7-9：

属性变量声明一般放在头文件的类声明之后，是要放出去给外部调用者看到的。

并且，在自动生成 setter 和 getter 方法时，方法的取名直接受变量名字影响。比如属性变量 int A 的默认 setter 方法为：

- (int)A;

而默认 setter 方法为：

- (void) setA:(int)newVar;

所以属性变量的取名需要特别注意。

开发者可以使用“@ synthesize”将属性变量和成员变量进行关联，关联后的两者就是一个指针但是成员变量没有属性变量的特性而已

小贴士 7-10：

关于 nonatomic，如果我们能确定不需要多线程访问时，强烈推荐使用这个关键字，因为 atomic 对于性能的损失相对较大。

小贴士 7-11：

如果是类的 delegate，推荐使用 assign 关键字。原因是避免了 retain 的死循环造成的对象无法被真正释放。

比如：有一个类 A，其中有一个成员对象是 B。同时，A 又是 B 的 delegate。

如果 delegate 用了 retain 的话，A 被 B retain 了一把。

此时，即使 A 被释放了，他之前被 B retain 的那个 1 也不会变成 0，造成了 A 的 dealloc 方法不会被调用。A 的成员对象 B 也就没有机会释放。造成无法释放的引用计数无限循环。

### 7.3.3 synthesize 和 dynamic 的区别

如果使用@property 声明了一个属性变量，仍然需要多做一步才能使其自动生成 setter 和 getter 方法，即在之后类实现中，对属性变量声明 @synthesize。对于关键字@synthesize 来说，对应的属性变量的 setter 和 getter 方法一定会存在，无论是程序员自定义的还是系统为该类自动生成的。

注意：Xcode4 的 Apple LLVM compiler 4.0 新编译器会自动为属性变量添加@syntesize，也就是说，如果开发者自己不写@synthesize 的话，编译器会自动生成属性变量的 setter 和 getter 方法，并且默认将变量和成员变量关联（如果属性变量是 A，则关联的成员变量默认就是_A，如果_A 不存在则编译器负责创建。）

而@dynamic 不同，他会告诉编译器，setter 和 getter 方法由其他方式生成，比如 super 类，比如程序员自己定义等等。

> 小贴士 7-12：
> @dynamic 在开发中，常可以在 NSManagedObject 的子类的使用中看到。

## 7.4 自动拖把 ARC

没有人爱收拾打理屋子，面对满地杂物，头疼的我们，谁不希望往床上一倒，醒来后屋子就全部变得干干净净？苹果的工程师看来也是懒人居多，在 iOS 5 SDK 和 MacOSX 10.7SDK 发布的时候，高调地宣布：作为伟大苹果开发者的我们，从此以后家里的家务不用我们再管了，自动拖把全部为我们搞定！他的全名叫 Automatic Reference Counting（ARC），后勤英雄！

### 7.4.1 ARC 的使用方法

ARC 机制其实相当简单，我们原先程序中由于未使用 ARC 机制而大量使用的 release／retain 等方法均不需要了，所有的内存管理事宜均由 ARC 机制在程序编译时自动完成。

放出两段代码我们来看一下，第一个是没有使用 ARC，第二个是使用 ARC：

没有使用 ARC：

```
@interface NoARCClassName : NSObject {
    NSString *strTest;
}

-(id)initWithString:(NSString *)aString;
@end

@implementation NoARCClassName
-(id)initWithString:(NSString *) aString {
    self = [super init];
    if (self) {
        strTest = [aString retain];
    }
```

```
    return self;
}

-(void)dealloc {
    [strTest release];
    [super dealloc];
}
@end
```

使用 ARC 的版本如下：

```
@interface ARCClass : NSObject {
    NSString * strTest;
}
-(id)initWithString:(NSString *)aString;
@end

@implementation ARCClass
-(id)initWithString:(NSString *) aString {
    self = [super init];
    if (self) {
        strTest = aString;
    }
    return self;
}
@end
```

很明显，那句关键的 strTest = aString；编译器给你自动 retian 了。并且再不用写 dealloc 方法。

ARC 机制在使用中有两个基本原则：

（1）retain, release, autorelease, dealloc 由编译器生成；

（2）dealloc 可以在子类中重写，但是不允许调用，即不可在 dealloc 中写上［super dealloc］。

### 7.4.2 ARC 新增关键字

作为内存管理国家的新国王，ARC 有两个武功高强的左右护法：strong 和 weak。

这两个大内护卫，实力不仅比原本的金刚三人众（assign，retain 和 copy）强，还掌握了吸心大法，能够去修饰变量，并且改变其处理内存管理相关问题的具体行为。那接下来让我们正式拜见下这两位少年英雄。

我们假设新建一个代表“人”的 Man 类，此类的 ARC 使用如下代码，首先是头文件的声明和 @synthesize：

```
@interface Person : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSNumber * years;
@property (nonatomic, strong) Woman *wife;

@end

@implementation Person
```

```
@synthesize name, years, wife;

@end
```

其中，strong 的含义和 retain 相同，weak 和 assign 相同，修饰完的属性变量用法也是完全没变。不过 strong 和 weak 只能修饰对象。

另外，对于普通变量，我们又多了四种关键字可以修饰：

__strong

__weak

__unsafe_unretained

__autoreleasing

先不管那么多新出来的特殊命名关键字，我们首先只来看看 strong 和 weak 两个最关键的东西，一旦我们理解了他们俩，剩余其他的关键字概念均所差无几。

而在介绍 strong 和 weak 之前，苹果官方对于 ARC 机制中对象的内存引用规则有二：

- 任何对象，如果仍有持有者，就不会销毁
- 任何对象，已经没有任何持有者，即自动销毁。

我们可以看到，苹果反复在强调持有者，它就是指向对象的指针，如果是 strong 修饰的，即是对象的持有者，如果是 weak 属性的，则不是持有者！

强参照（strong）：内力深厚，一骑当千，如图 7-3 所示：

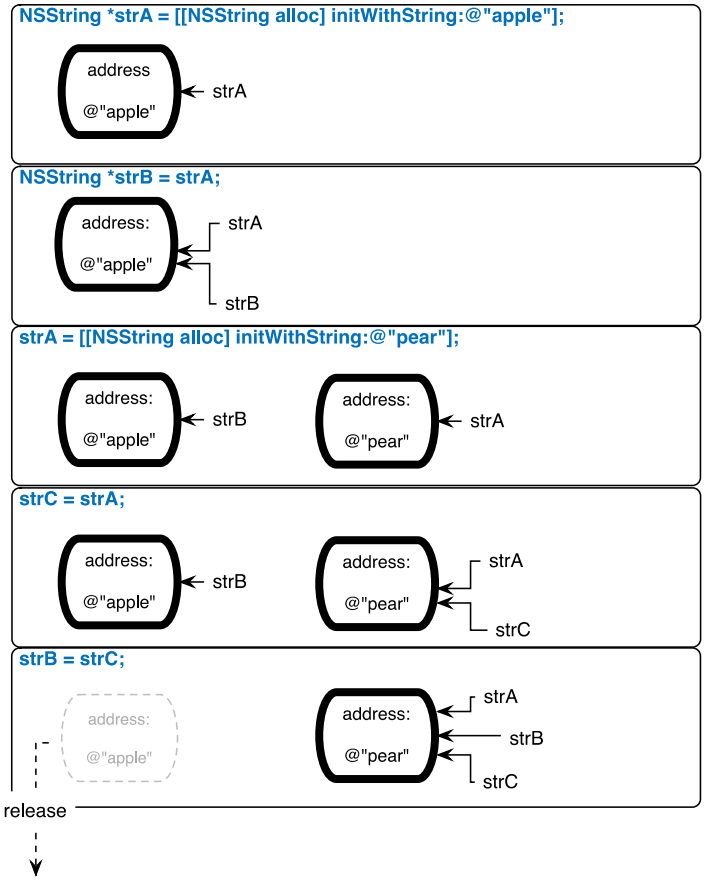


图 7-3　强参照图示

上图的 strong 使用分成 5 步，分别作如下解释：

1) 根据默认的变量参照规则是__strong，strA 作为”apple”字符串对象的最初持有者。

2) 根据默认的变量参照规则是__strong， strB 又成为了”apple”字符串对象的另一个持有者。

3) 根据默认的变量参照规则是__strong，strA 又成为”pear”字符串对象的最初持有者。此时，两个字符串对象“apple”/“pear”都拥有一个持有者。

4) 根据默认的变量参照规则是__strong， strC 又变成了“pear”字符串对象的另一个持有者。

5) 根据默认的变量参照规则是__strong，strB 也成为了“pear”字符串对象的持有者之一。此时，“apple”对象没有持有者了，即被释放。

弱参照（weak）：灵活多变，游刃有余，如图 7-4 所示：

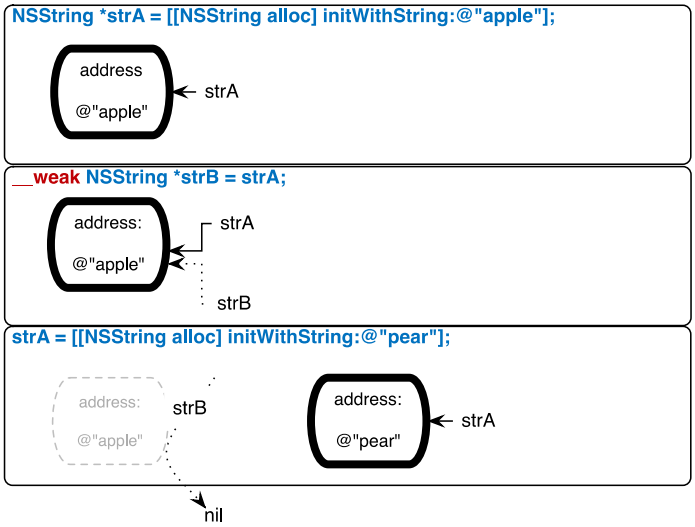


图 7-4 弱参照图示

上图的 weak 使用分成 3 步，分别作如下解释：

1) 根据默认的变量参照规则是__strong，strA 作为”apple”字符串对象的最初持有者

2) 关键字__weak 出现!strB 虽然指向“apple”字符串对象，但是不是其持有者，只是这个时候“apple”存在，所以 strB 一切正常。

3) 根据默认的变量参照规则是__strong，strA 作为”pear”字符串对象的最初持有者。此时，“pear”没有任何一个持有者，即自动销毁。strB 因为指向的内存地址被回收，也会自动置空，黯然退隐。

基于图示，我们对于其他关键字的新近认识，适逢时机！

__strong：默认修饰变量的关键字，为对象的持有者

__weak：简单的指针指向对象，但并非对象的持有者。对象如果被销毁，指针自动置空。

> 小贴士 7-13：
> weak 在 iOS 5/ Mac OS X 10.7 之后的 SDK 才能够被使用。

__unsafe_unretained：和__weak 一样，唯一的区别便是，对象即使被销毁，指针也不会自动置空，此时指针指向的是一个无用的野地址。如果使用此指针，程序会抛出 BAD_ACCESS 的异常。

__autoreleasing：介于__strong 和__weak 之间，使指向的对象延迟销毁，我们常可以在函数的输出参数和返回值上见到使用，如下两个示例：

例 1：（输出参数）

```
- (void)fetchOtherString:(__autoreleasing NSString**)aString {
        aString = [[NSString alloc] initWithString:@"testString"];
```

```
}
```

出了这个函数，拥有"testString"内容的 aString 对象也暂时不会销毁，他的自动销毁区域会扩大到外面调用的那层。

例 2：（返回值）

```
- (NSString*)testString
{
    NSString __autoreleasing *returnString =
    [[NSString alloc] initWithString:@"testString"];
    return returnString;
}
```

如果不加__autoreleasing，testString 会在返回时自动销毁。

### 7.4.3 ARC 机制的使用规则

对于 ARC 机制的使用，苹果发布了几条重要的规则需要开发者遵守。单单看那些生搬硬套的东西难免生涩，根据开发经验将规则总结如下：

（1）不能调用 dealloc， 不能重写和调用 retain，release，retainCount 和 autorelease，同理，@selector（retina），@selector（release）这些曲线救国的方法也不能调用。

> 小贴士 7-14：
> 但是由于释放由 ARC 为我们完成了，对象的销毁过程我们仍然需要把除了释放之外的事情做掉，比如 delegate 的置空之类的安全规范。

dealloc 虽然能够重写，但是不能调用［super dealloc］之类的方法。CoreFoundation 框架由于非从属 cocoa 框架，所以 CFRetain 和 CFRelease 仍然正常使用。

（2）不能使用 NSAllocateObject 或 NSDeallocateObject 函数来创建对象。比如 id Obj = NSAllocateObje([NSObject class], 0, nil);

（3）不能在 C 语言的结构体中使用对象指针，同时建议用 Objective－C 的类来管理数据而不是结构体。

（4）id 和 void＊的转换的正确对待。

我们都知道，NSString 和 CFStringRef 是桥接的关系，可以直接互通使用，这下在 ARC 模式中，当我们互相转换的时候，则需要再加上__bridge 的关键字，这个关键字分为几种，如下所示：

__bridge

```
NSString      *string       = [NSString stringWithString:@"test"];
CFStringRef cfString     = (__bridge CFStringRef)string;
```

__bridge_retained

```
NSString      *string       = [NSString stringWithString:@"test"];
CFStringRef cfString      = (__bridge_retained CFStringRef)string;
// 由于 Core Foundation 的对象不属于 ARC 的管理范畴，所以需要自己 release
CFRelease(cfString);
```

__bridge_transfer

```
CFStringRef cfString      = CFStringCreateWithCString( kCFAllocatorSystemDefault,
```

```
                                                    "abc",
                                                    kCFStringEncodingMacRoman);
    NSString      *string       = (__bridge_transfer NSString *)cfString;
    // 因为已经用 __bridge_transfer 转移了对象的所有权，所以不需要调用 release
    // CFRelease(cfString);
```

（5）不能使用 NSAutoreleasePool 对象。ARC 中，全部使用@autorelease 关键字代替，且比 NSAutorealsePool 更高效。

（6）不得使用内存 Zone，那些牵涉到 NSZone 的方法都不得使用。

（7）不得对一个属性变量的取值方法命名以 new 开头。比如：

```
@property (strong, nonatomic) NSString *newName;（错误，取值方法为 newName）
@property(strong, non atomic, getter = theNewName) NSString *newName;（正确，取值方法为 theNewName）
```

（8）outlet 均用 weak 关键字修饰，除非它是 xib 中最顶部的界面元素，则需要用 strong 否则 xib 初始化出来后就不销毁了嘛？！

（9）Core Foundation 不适用 ARC。该创建的仍然创建！该释放的仍然释放！

### 7.4.4 ARC 机制的注意事项

ARC 的优缺点非常明显，取舍与否一般需要根据项目的具体情况而定。总之，对于入门的开发朋友来说，ARC 是你学习苹果开发的良好帮手。而对于有经验的朋友来说，如果你始终能自信控制好你的内存，不转换 ARC 也无妨。表 7-3 为 ARC 的优缺点。

表 7-3　ARC的优缺点

| | |
|---|---|
| 优点 | 无序担心内存泄漏，只许专注于开发。 |
| | 大量的内存管理代码灰飞烟灭，代码量减少 |
| | 编译器编译优化，比如变量未初始化，且未用__unsafe_unretained修饰，编译器会自动给你置空。避免了野指针的活跃。 |
| 缺点 | 移植至ARC需要工作量和学习过程，且很多第三方框架暂不支持ARC |
| | 一切都是自动化，大块内存使用完需要尽早释放的情况如果不特别去考虑编码方式，会有性能损失 |

### 7.4.5 迁移程序到 ARC 的做法

苹果总体来说是鼓励开发者往 ARC 上转的，在 Xcode4.2 中，体贴地植入了“转 ARC”的功能，具体操作流程见图 7-5 至 7-8 所示：


图 7-5 迁移 ARC 的 Xcode 菜单

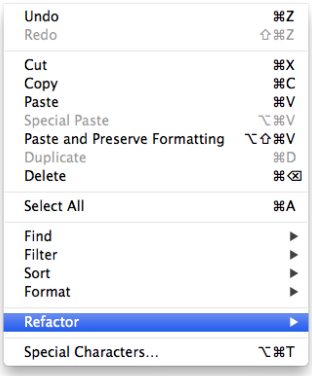


图 7-6 迁移 ARC 的 Edit 菜单

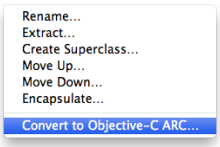


图 7-7 迁移 ARC 的 Edit 二级菜单

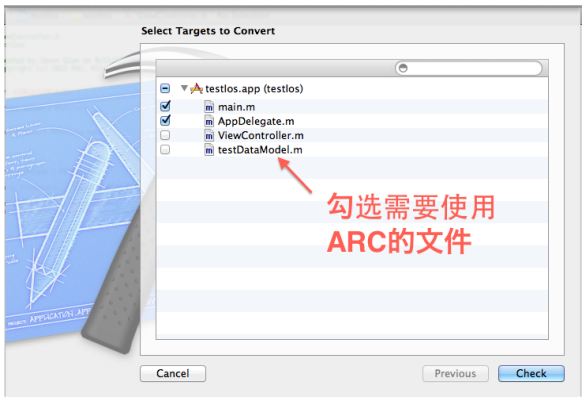


图 7-8 迁移 ARC 的工程配置

转完之后的情景可以料想，一定会面对无数的错误警告，大部分是和内存管理相关，请有耐心的一个一个解决。

如果当前整个工程都没有使用 ARC 机制，而想对于特定文件在编译时使用 ARC，请对此文件添上预编译项(-objc-arc)。具体操作流程如图 7-9：

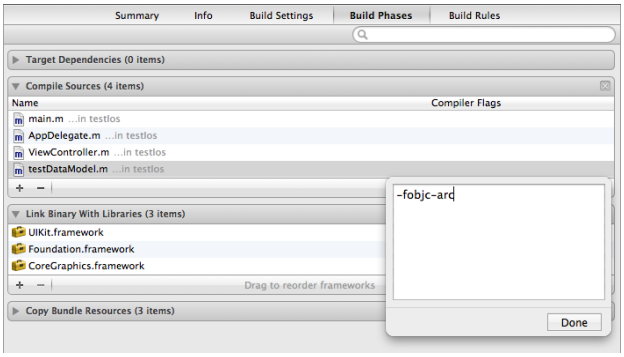


图 7-9 避免使用 ARC 的文件编译标志位

## 7.5 本章小结

本章全面但不失细节地介绍了苹果开发中内存管理的大部分知识。笔者一直认为，好的程序有很多重要点需要覆盖到，诸如界面美观，用户体验，程序性能等。其中的程序性能部分，同样很大部分依赖于内存的良好管理。iOS 作为移动设备的系统，对于内存利用优良率更是注重，每个程序都会有个内存使用量上限，一旦超过那个值，系统会毫不客气地将你关闭。并且在低内存的运行环境中，很多 UI 上的效果常会出现顿卡的现象甚至发生布局错误，更是将用户体验降低到极致。MacOSX 虽然有更多的资源可供客户端使用，但是请不要忘记，MacOSX 上的往往也拥有更多的并行程序，更长的运行清醒时间。

提升内存管理的意识，仅是一个合格的苹果开发者的第一步。