第5章 内存管理

学习目标

- 了解管理内存的概念，可以简述内存管理的重要性。
- 通过分析引用计数器的工作原理，学会对单个或者多个对象进行内存管理，避免程序开发中出现内存泄漏。
- 掌握@property参数的使用场景，可以准确为不同类型的属性设定参数。
- 理解自动释放池的作用，可以阐述自动释放池何时被创建和销毁。

当OC程序运行时，会在手机的内存中产生很多临时的数据（变量，对象等），系统会将它们归纳分类，然后分配到内存的不同区域。在内存中有五大区域，具体如下：

（1）栈区域：保存局部变量。当局部变量的作用域被执行完毕以后，这个局部变量就会被系统立即回收。

（2）堆区域：保存OC对象。使用C函数申请的动态空间都是分配在堆里面的。

（3）BSS段：保存未初始化的全部变量和静态变量。一旦初始化就会被回收，并且转入到数据段中。

（4）数据段：保存已经初始化的全局变量和静态变量。直到程序结束时才会被回收。

（5）代码段：程序结束时，系统会自动回收存储在代码段中的数据。

由于移动设备的内存是有限的，一旦占用量过大，势必造成卡顿或者闪退的情况。其中，栈区、BSS段、数据段和代码段存放的数据是由系统负责回收的，而堆中的对象系统不能自动回收，所以内存管理的范围是存放在堆中的OC对象。

为了让应用程序的内存消耗到最低，需要及时地清理无用的对象，但是需要确保清除的不是正在使用的对象。Objective-C中提供了MRC（Mannul Reference Counting，手动引用计数）和ARC（Automatic Reference Counting，自动引用计数）两种机制对内存中的对象进行管理，本章主要介绍MRC内存管理的相关知识。

5.1 内存管理概述

当有人在使用某个对象时，它一定不能被回收，当没有人使用某个对象时，它才能被回收。这时需要有个计数的东西，随时统计此时使用对象的数量，为此提出了引用计数器的概念，本节将针对引用计数器进行详细介绍。

5.1.1 引用计数器

为了记录每个对象被使用的情况，Objective-C为每个对象设置了一个内部计数器，称

为引用计数器。当对象被创建时，让引用计数的值为 1，每次对象被引用时，引用计数增加 1，每次对象减少一次引用时，引用计数减 1，直到引用计数器的值为 0 时，直接释放对象占用的内存。接下来，以现实中办公室使用电灯的例子，说明引用计数的原理，如图 5-1 所示。

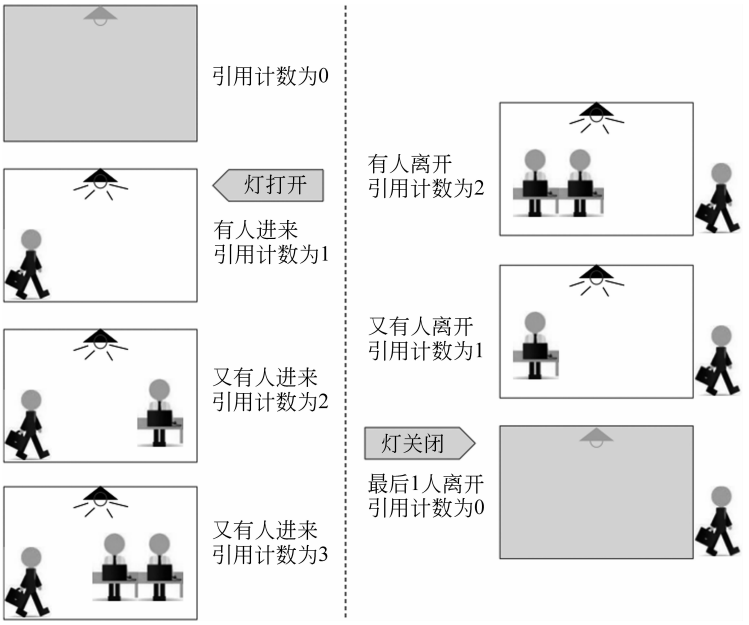

图 5-1　引用计数原理示意图

图 5-1 介绍了引用计数的原理。假设有一个办公室(内存)，有个人打开了办公室的灯(对象)，此时使用电灯照明的人数(引用计数)为 1；有人进入办公室，引用计数就增加 1；有人离开办公室，引用计数就减少 1；当最后一个人离开时，此时使用电灯照明的人数为 0，需要关闭电灯，办公室停止使用。

知道了引用计数器的原理后，接着来操作引用计数器。在 MRC(手动引用计数)机制中，提供了手动管理内存的方法，具体如下。

1. retainCount 属性

每个对象内部都有 retainCount 属性，隶属于 unsigned 类型，占用 4 个字节。这就是引用计数器，用于记录当前对象被使用的次数。

2. retain 方法

当对象发送一条 retain 消息(调用 retain 方法)时，该对象的引用计数器的值加 1。例如，在办公室照明的例子中，有人进入办公室，使用电灯的人数增加 1。

3. release 方法

当对象发送一条 release 消息(调用 release 方法)时，该对象的引用计数器的值减 1。例如，有人离开办公室，使用电灯照明的人数减少 1。

4. dealloc 方法

当对象的引用计数器的值变成 0 时，其占用的内存会被系统立即回收，同时系统会发送一条 dealloc 消息。例如，最后一个人离开办公室，关闭电灯，办公室不再投入使用。

多学一招：内存管理分类——MRC 和 ARC 机制

(1) MRC(手动引用计数)

当多一个人使用对象时，要求程序员手动地发送 retain 消息；当少一个人使用对象时，要求程序员手动地发送 release 消息。

(2) ARC(自动引用计数)

当多一个人使用对象时，系统会自动在合适的位置发送 retain 消息；当少一个人使用对象时，系统会自动在合适的位置发送 release 消息。

5.1.2　第一个 MRC 程序

自 Xcode 4.2 开始支持 ARC 模式，默认不再使用 MRC 模式。假设要体验 MRC 模式开发，前提是需要在 Xcode 中关闭自动引用计数，具体步骤如下。

(1) 创建一个命令行工程，在打开的导航面板中，单击左侧的根目录，选择 Build Settings→All，如图 5-2 所示。

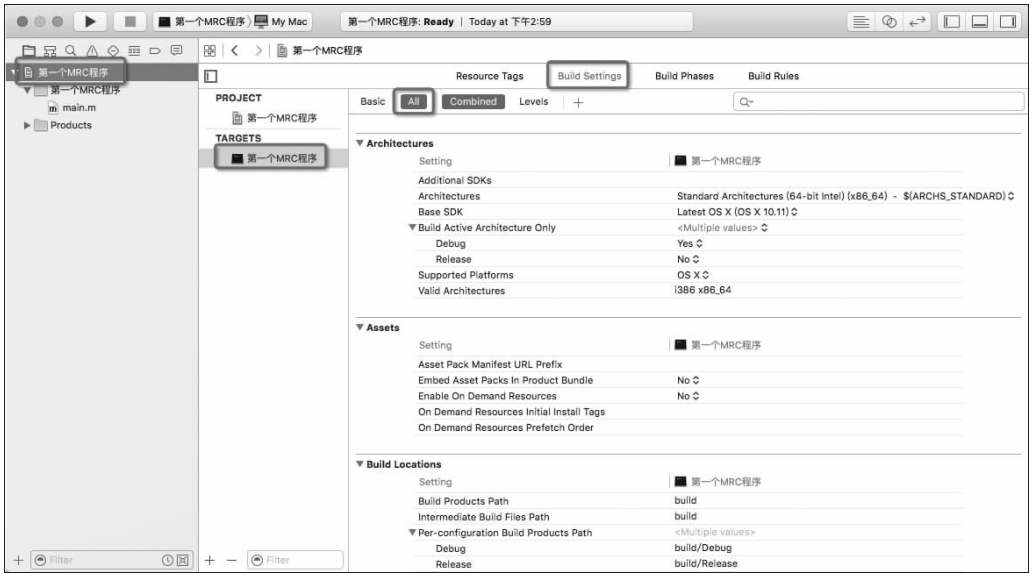

图 5-2　Build Settings 选项

(2) 在搜索栏中输入“automatic”，下面过滤出 Apple LLVM 7.1 - Language - Objective C 选项，其内部的 Objective-C Automatic Reference Counting(自动引用计数)默认为“Yes”，将其设置为 No 就行，如图 5-3 所示。

关闭“自动引用计数”功能后，就能够在 Xcode 中手动操作引用计数器了。接下来，创建一个 Person 类，Person 类的声明和实现如例 5-1 和例 5-2 所示。

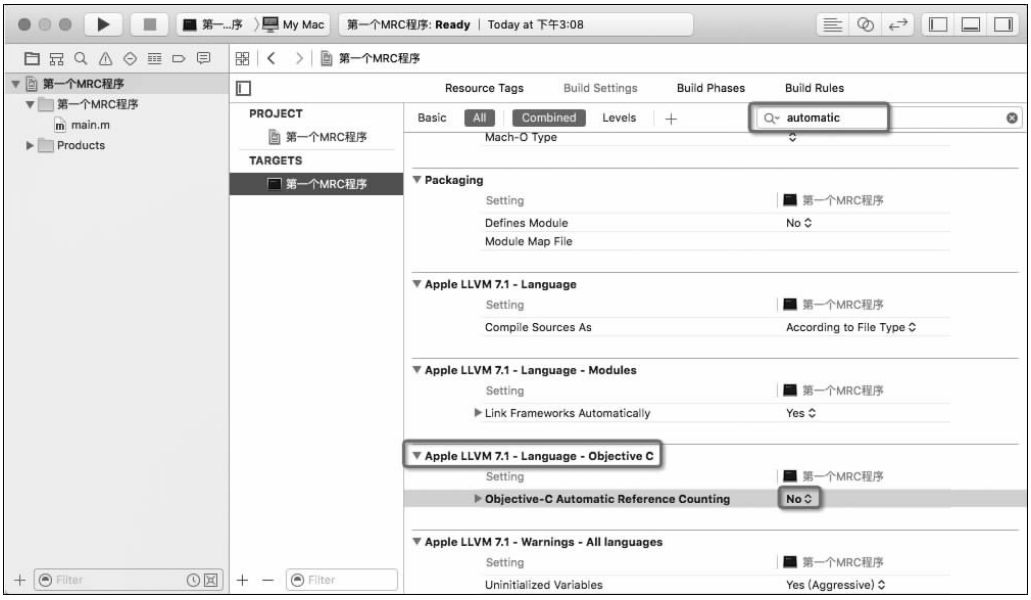

图 5-3 关闭“自动引用计数”功能

例 5-1 Person. h

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
@property NSString * name;          //姓名
@property int age;                  //年龄
-(void)sayHi;                       //打招呼
@end
```

例 5-2 Person. m

```
#import "Person.h"
@implementation Person
-(void)sayHi
{
    NSLog(@"大家好,才是真的好!");
}
-(void)dealloc
{
    NSLog(@"名字叫做%@的人遇难.", _name);
    [super dealloc];
}
@end
```

在例 5-1 和例 5-2 中,定义了 name 和 age 两个属性、一个 sayHi 方法,接着实现了 sayHi 方法。为了能够监测到 Person 对象的回收,需要重写 dealloc 方法。在重写 dealloc 方法时,必须在末尾位置添加[super dealloc]代码,调用父类的 dealloc 方法。

在 main. m 中调用 alloc 方法创建 person 对象,然后对指向新对象的指针变量 p 分别做一次 retain 操作、两次 release 操作,并在每次操作之后打印输出对象当前时刻的引用计数器的值。具体代码如例 5-3 所示。

例 5-3　main. m

```
1   #import <Foundation/Foundation.h>
2   #import "Person.h"
3   int main(int argc, const char * argv[]) {
4       Person * p=[[Person alloc] init];
5       p.name=@"小明";
6       NSLog(@"count=%lu", p.retainCount);
7       [p retain];
8       NSLog(@"count=%lu", p.retainCount);
9       [p release];
10      NSLog(@"count=%lu", p.retainCount);
11      [p release];
12      return 0;
13  }
```

运行程序，控制台的输出结果如图 5-4 所示。

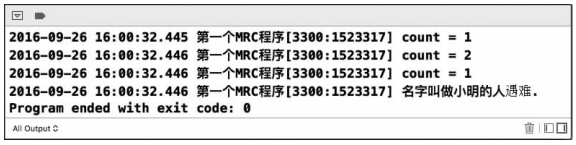

图 5-4　控制台的输出结果

从图 5-4 的结果可以看出，当程序执行完第 5 行时，创建了一个 Person 类对象，并且设置 name 的值为“小明”，此时引用计数器的值为 1；

当程序执行完第 7 行时，由于调用了 retain 方法，使得引用计数器的值变成 2；

当程序执行完第 9 行时，由于调用了 release 方法，使得引用计数器的值变成 1；

当程序执行完第 11 行时，继续调用了 release 方法，使得计数器的值变成 0。此时，系统会自动调用重写的 dealloc 方法，执行里面的打印语句。

脚下留心：野指针与僵尸对象

在例 5-3 中，当执行完第 11 行代码后，p 指针指向的对象会立即被回收，p 指针此时就成为了野指针。在 Objective-C 中，野指针是指指针指向的对象已经被回收了。

当对象被回收以后，它占用的内存空间可以分配给其他对象使用。但是这个空间在没有被分配给其他对象前，对象的数据仍然存放在这个空间内。为了大家更好地理解，在例 5-3的第 11 行代码后面，增加调用 sayHi 方法的代码，具体如下。

```
[p sayHi];
```

再次运行程序，控制台的输出如图 5-5 所示。

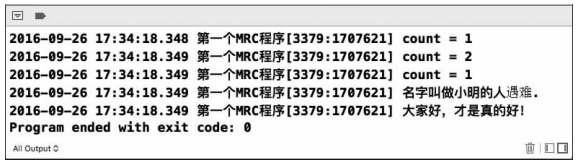

图 5-5　控制台的输出结果

从图 5-5 中可以看出，使用野指针访问已经被释放的对象，仍然能够调用 sayHi 方法。像这样一个已经被释放的对象，但这个对象所占用的空间还没有分配给别人，这样的对象称为僵尸对象。但是如果多次调用 sayHi 方法，Xcode 就会出现报错信息。

通常我们认为，只要对象变成了僵尸对象，无论空间是否分配给别人，都是不允许访问的。也就是说，只要访问了僵尸对象，就会出现报错信息。默认情况下，Xcode 不会管理僵尸对象，需要按照如下操作开启对僵尸对象的监控。

(1) 在一个项目中，单击工具栏左侧的项目名称按钮，按钮的具体位置如图 5-6 所示。

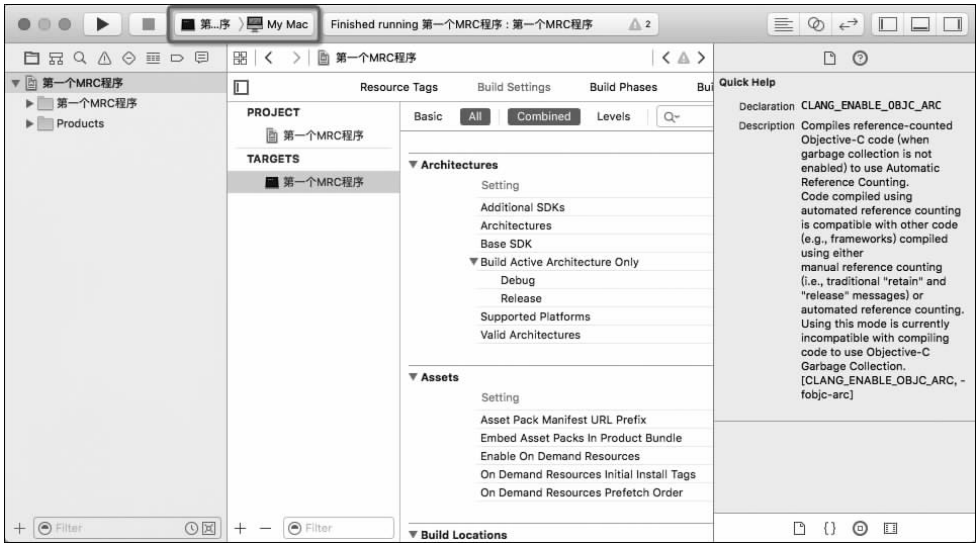

图 5-6　项目名称按钮的位置

(2) 单击上述按钮后，出现一个下拉列表，接着选择“Edit Scheme”会弹出图 5-7 所示的窗口。

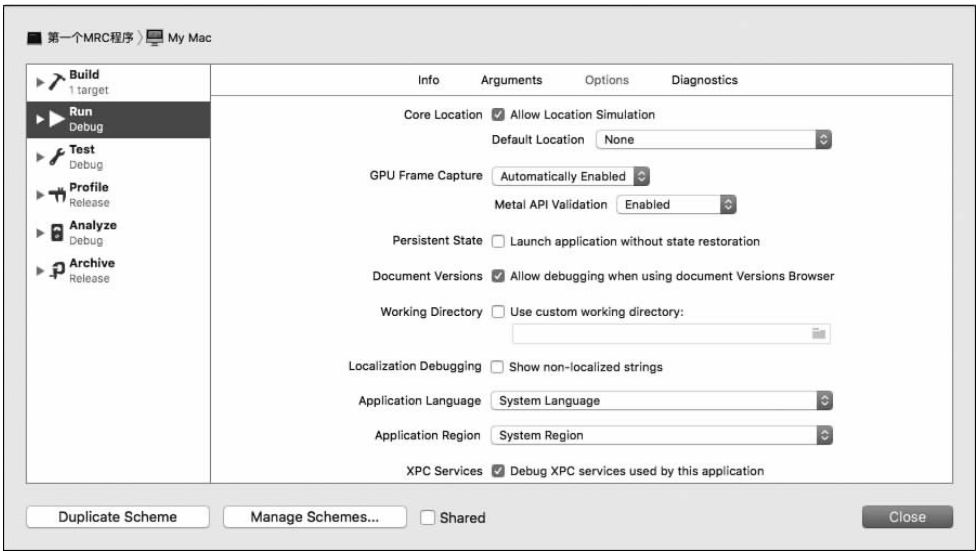

图 5-7　设置结构属性的窗口

(3) 在右侧的对话框中切换至 Diagnostics 分页，用来设置程序诊断。勾选 Enable Zombie Objects 选项的复选框，就会开启僵尸对象的监控，如图 5-8 所示。

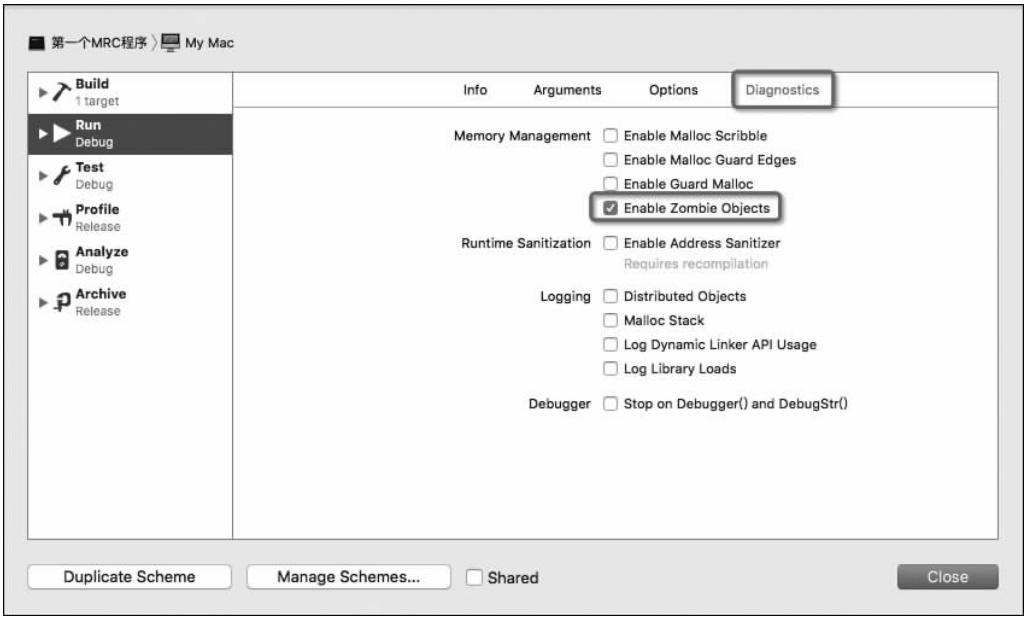

图 5-8　勾选监控僵尸对象

多学一招：内存管理的原则

1. 有对象的创建，就要匹配一个 release。
2. retain 的次数要和 release 的次数相匹配。
3. 谁要使用对象谁就 retain，谁不再使用对象谁负责 release。

总而言之，有始有终，有加有减。

多学一招：dealloc 方法的写法

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

5.2　单个对象的内存管理

所谓内存泄漏，指的是一个对象在该回收时没有被及时回收，该对象一直驻留在内存中，直到程序结束时才会回收。我们了解到引用计数器的使用，如何能够确保不会产生内存泄漏呢？

总结可知，产生内存泄漏的原因具体分为如下几种情况：

(1) 有对象的创建，而没有对应的 release 来释放。

(2) retain 的次数和 release 的次数是不匹配的。

(3) 在不恰当的位置,将指针赋值为 nil。

(4) 在方法中为传入的对象进行不适当的 retain。

根据上述阐述的情况,要确保单个对象能够及时地回收,防止产生内存泄漏,需要做到以下几点:

(1) 只要有对象的创建,就必须为其匹配一个 release。

(2) retain 的次数和 release 的次数一定要匹配。

(3) 只有当指针变成野指针时,才能赋值为 nil。

(4) 在方法中不要随意地为传入的对象做 retain 操作。

5.3 多个对象的内存管理(setter 方法内存管理)

假设有如下一个案例:凤姐开车去拉萨。基于面向对象的思维模拟上述案例,采用名词提炼法可分为如下三类:

- 人类:
  属性:车;
  行为:开车。
- 车类:
  属性:速度;
  行为:行驶。
- 城市类:

其中,城市类仅用到了一个地名,这里可以直接忽略。接下来,创建一个工程,再新建一个 Car 类,Car 类的声明代码和实现代码如例 5-4 和例 5-5 所示。

例 5-4 Car. h

```
#import <Foundation/Foundation.h>
@interface Car : NSObject
{
    int _speed;            //速度
}
-(void)setSpeed:(int)speed;
-(int)speed;
-(void)run;                //行驶
@end
```

例 5-5 Car. m

```
#import "Car.h"
@implementation Car
-(void)setSpeed:(int)speed
{
    _speed=speed;
}
-(int)speed
{
```

```
9       return _speed;
10  }
11  //行驶
12  -(void)run
13  {
14      NSLog(@"时速为%d的车子在行驶.", _speed);
15  }
16  -(void)dealloc
17  {
18      NSLog(@"时速为%d的车子报废了.", _speed);
19      [super dealloc];
20  }
21  @end
```

在例 5-4 中直接定义了一个_speed 属性，同时为其手动编写对应的 setter 和 getter 方法，并且声明了表示行驶的 run 方法。在例 5-5 中同样手动编写了 setter 和 getter 方法的实现。此外重写了 dealloc 方法，用来监测 Car 对象的销毁。

新建一个 Person 类，继承自 NSObject 基类，Person 类的声明代码和实现代码如例 5-6 和例 5-7 所示。

例 5-6　Person. h

```
1   #import <Foundation/Foundation.h>
2   #import "Car.h"
3   @interface Person : NSObject
4   {
5       Car * _car;          //车
6   }
7   -(void)setCar:(Car *)car;
8   -(Car *)car;
9   -(void)drive;           //开车
10  @end
```

例 5-7　Person. m

```
1   #import "Person.h"
2   @implementation Person
3   -(void)setCar:(Car *)car
4   {
5       _car=car;
6   }
7   -(Car *)car
8   {
9       return _car;
10  }
11  -(void)drive          //开车
12  {
13      NSLog(@"走,去拉萨,开车");
```

```
14      [_car run];
15  }
16  -(void)dealloc
17  {
18      NSLog(@"Person dealloc");
19      [super dealloc];
20  }
21  @end
```

在例 5-6 中直接定义了一个 Car 类型的_car 属性，同时为其手动编写对应的 setter 和 getter 方法，并且声明了表示开车的 drive 方法。在例 5-7 中实现了 setter 和 getter 方法，重写了 dealloc 方法来监测 Person 对象的销毁。在 drive 方法的实现中，调用了 Car 对象的 run 方法。

在 main.m 文件中，先创建一个 Person 类对象，再次创建一个 Car 类对象，并且将 Car 对象赋值给_car 属性，然后调用 drive 方法进行开车行为，最后为两个对象匹配相应的 release 方法，具体代码如例 5-8 所示。

例 5-8　main.m

```
1   #import <Foundation/Foundation.h>
2   #import "Person.h"
3   int main(int argc, const char * argv[]) {
4       @autoreleasepool {
5           //创建人物对象,凤姐
6           Person *fj=[[Person alloc] init];
7           //创建车对象,并设置速度
8           Car *bmw=[[Car alloc] init];
9           bmw.speed=100;
10          fj.car=bmw; //赋值给 car 属性
11          //凤姐开车
12          [fj drive];
13          [bmw release];
14          [fj release];
15      }
16      return 0;
17  }
```

在例 5-8 中，fj 指针引用了 Person 类的实例，bmw 指针引用了 Car 类的实例，并且给 speed 属性赋值为 100，让 bmw 指针引用的对象的地址赋值给_car 指针。因此，bmw 指针和_car 指针都指向了同一个 Car 实例。此时，程序运行期间对象在内存中的状态如图 5-9 所示。

然后调用 drive 方法，同时调用 Car 类的 run 方法，将打印方法内部的输出语句。依据内存管理的原则，只要创建了对象，必须有对应的 release 操作，为此分别给每个对象发送 release 消息。

运行程序，控制台的输出结果如图 5-10 所示。