面试题预览：

1. 怎么判断某个cell是否显示在屏幕上

今天我们来分析一下UITableViewCell的重用机制.
首先,我们要明白我们为什么需要使用这种机制,其次,这种机制的原理是什么.
我们先举个例子来说明.一个UITableView中有许多需要显示的cell,但是我们不可能每个都会浏览到,那么如果我们把这些数据全部都加载进去,是不是造成了内存的负担呢.
我们所能显示的区域通常只有一个屏幕的大小,那么那些屏幕之外的信息是不需要一次性全都加载完的,只有当我们滑动屏幕需要浏览的时候,我们才需要它加载进来.因此,就有了我们要介绍的这部分内容,UITabelViewCell的重用机制.
重用机制实现了数据和显示的分离,并不为每个数据创建一个UITableViewCell,我们只创建屏幕可显示的最大的cell个数+1,然后去循环重复使用这些cell,既节省空间,又达到我们需要显示的效果.
这种机制下系统默认有一个可变数组NSMutableArray* visiableCells,用来保存当前显示的cell.一个可变字典NSMutableDictnery* reusableTableCells,用来保存可重复利用的cell.(之所以用字典是因为可重用的cell有不止一种样式,我们需要根据它的reuseIdentifier,也就是所谓的重用标示符来查找是否有可重用的该样式的cell).
重用的写法如下:

```
// 设置单元格  indexPath :单元格当前所在位置 -- 哪个分区哪一行等
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
    (NSIndexPath *)indexPath //UITableViewDataSource
```

```
{
   static NSString *identifier = @"cell" ;
   //相当于从集合中找寻完全出屏幕的单元格.
   // identifier : 因为一个表视图中可能存在多种样式的单元格,咱们把相同样式的单元格放到
        同一个集合里面,为这个集合加标示符,当我们需要用到某种样式的单元格的时候,根据
        不同的标示符,从不同的集合中找寻单元格.
   UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:identifier] ;
   // 如果从集合中未找到单元格,也就是集合中还没有单元格,也就是还没有单元格出屏幕,那
        么我们就需要创建单元格
   if (!cell)
   {
      // 创建cell的时候需要标示符(Identifier)是因为,当该cell出屏幕的时候需要根据标示符放
         到对应的集合中.
      cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
         reuseIdentifier:@"cell"] ;
   return cell ;
}
```

系统第一次执行- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath这个方法的时候, reusableTableCells为空,[tableView dequeueReusableCellWithIdentifier:identifier]的返回值为nil,我们需要通过[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier: identifier]方式来创建.

当我们的数据过多,整个屏幕的cell显示不完全时,这个方法的执行情况是 :

(1) 先执行[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier: identifier]创建整个屏幕能显示的cell数+1的cell(当我们拖动UITableView的时候,第一个cell没有移出屏幕,最下面的cell就已经存在),并指定相同或者不同的标示符identifier.把创建出的屏幕能显示的cell全部都加入到visiableCells数组中(最后一个创建的先不加入数组)，reusableTableCells为空.

(2)当我们拖动屏幕时,顶端的cell移出屏幕并加入到reusableTableCells字典中,键为identifier ,并把之前已经创建的但是没有加入到visiableCells的cell加入到visiableCells数组中.

(3)当我们接着拖动的时候,因为reusableTableCells中已经有值，所以，当需要显示新的cell，cellForRowAtIndexPath再次被调用，执行[tableView dequeueReusableCellWithIdentifier: identifier]，返回一个标示符为identifier的cell。该cell移出reusableTableCells之后加入到visiableCells；顶端的cell移出visiableCells并加入到reusableTableCells.如果visiableCells数组中没有找到identifier类型的cell,则再次重新alloc一个.

在iOS6之后系统加入了一种单元格注册的方法.

[self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier: identifier] ;

这个方法的作用是,当我们从重用队列中取cell的时候,如果没有,系统会帮我们创建我们给定类型的cell,如果有,则直接重用. 这种方式cell的样式为系统默认样式.

在设置cell的方法中只需要:

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath: (NSIndexPath *)indexPath

```
{
    // 重用队列中取单元格 由于上面已经注册过单元格,系统会帮我们做判断,不用再次手动判
        断单元格是否存在
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier: identifier
        forIndexPath:indexPath] ;
    return cell ;
}
```

2. 进程和线程的区别

1.定义

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

2.关系

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

3.区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。**但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。**

1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

4.优缺点

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

3. TCP与UDP区别

这是两个工作在TCP/IP协议传输层的两个不同的协议，是用来传输数据用的。

TCP：Transfer Control Protocol，传输控制协议。

这是一个全双工的、面向连接的、可靠的并且是精确控制的协议。

主要是用在那些实时性不强、但要求不能出错的应用。比如说，网页的浏览、文件的下载（不是BT、电驴下载）、邮件的收发等场合，就需要TCP协议进行传输（因为不会出错）。

当然，它在网络方面的开销是昂贵的。

UDP：User Datagram Protocol，用户数据报协议。

这是一个不可靠的传输协议。因为它不排序所要发送的数据段、不关心这些数据段到达目的方的顺序（所以它才不可靠），所以它在网络的开销要比TCP小很多。因此UDP适合用在那些实时性强、允许出错的场合。

比如说：即时通信（MSN、QQ），视频，语音等方面

4. TCP流量控制

滑动窗口机制

比如发送端能发送5个数据，接收端也能收到5个数据，给个确认（ack）给发送端，确认我收到5个数据。如果网络通信出现繁忙或者拥塞的时候，接收端只能收3个数据，接受端给个确认我只能收3个数据，那么发送端就自动调整发送的窗口为3，当线路又恢复通畅的时候，接受端又可以受到5个数据，那它会给确认给发送端，告诉它我的窗口为5，那发送端就把窗口又调整会5，这样进行流量控制的

2、比如说发送端窗口为3，发送到接收端，接收端的接收窗口为5的话，接受数据，并且会给发送端一个ack（确认）告诉发送端我的窗口为5，发送端收到确认后会把自己的发送端窗口调整为5~~这样就可以加速数据传输了

5. 数组和链表的区别

二者都属于一种数据结构

从逻辑结构来看

1. 数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费；数组可以根据下标直接存取。

2. 链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项，非常繁琐）链表必须根据next指针找到下一个元素

从内存存储来看

1. (静态)数组从栈中分配空间, 对于程序员方便快速,但是自由度小

2. 链表从堆中分配空间, 自由度大但是申请管理比较麻烦

从上面的比较可以看出，如果需要快速访问数据，很少或不插入和删除元素，就应该用数组；相反， 如果需要经常插入和删除元素就需要用链表数据结构了。

6. UIView生命周期

一、 大体流程：

(loadView/nib)文件来加载view到内存-->viewDidLoad函数进一步初始化这些view-->内存不足时， 调用viewDidUnload函数释放views-->当需要使用view时又回到第一步

loadView：

永远不要主导调用这个函数。

viewController 会在view的property被请求并且当前view值为nil时调用这个函数。如果你手动创建view， 你应该重载这个函数，切不要在重载的时候调用[super loadView]。

viewDidload:

这个函数的作用主要是让你可以进一步的初始化你的views。

viewDidLoad通常负责的是view及其子view被加载进内存之后的数据初始化的工作，即视图的数据部分的初始化

viewDidUnLoad：

这个函数时viewDidLoad的对立函数。在程序内存欠缺时，这个函数被controller带哦用，来释放他的view以及view相关的对象。由于controller通常保存着view以及相关的object的引用，所以你必须使用这个函数来放弃这些对象的所有权以便内存回收，但不要释放那些难以重建的数据

viewWillAppear:

视图即将可见时调用，默认情况下不执行任何操作。

viewDidAppear：

视图已完全过渡到屏幕上时调用

viewWillDisappear：

视图被驳回时调用，覆盖或以其他方式隐藏，默认情况下不执行任何操作

viewDidDisappear:

视图被驳回后调用，覆盖或以其他方式隐藏。默认情况下不执行任何操作

didReceiveMemoryWarning:

当程序内存过度时，系统会调用该方法

二、Controller和View的生命周期

这里指的View是指Controller的View。它作为Controler的属性，生命周期在Controller的生命周期内。就是说你的Controller不能在view释放前就释放了。

当你alloc并init了一个ViewController时，这个ViewController应该是还没有创建view的。ViewController的view是使用了lazyInit方式创建，就是说你调用的view属性的getter：［self view］。在getter里会先判断view是否创建，如果没有创建，那么会调用

loadView来创建view。loadView完成时会继续调用viewDidLoad。loadView和viewDidLoad的一个区别就是：loadView时还没有view。而viewDidLoad时view以及创建好了。

当view被添加其他view中之前时，会调用viewWillAppear，而之后会调用viewDidAppear。

当view从其他view中移出之前时，会调用viewWillDisAppear，而之后会调用viewDidDisappear。

当view不在使用，而且是disappeared，受到内存警告时，那么viewController会将view释放并将其指向nil。

三、代码组织（如何设计良好的viewcontroller）

ViewController生命周期中有那么多函数，一个重要问题就是什么代码该写在什么地方。

1、init里不要出现创建view的代码。良好的设计，在init里应该只有相关数据的初始化，而且这些数据都是比较关键的数据。init里不要掉self.view，否则会导致viewcontroller创建view。（因为view是lazyinit的）。

2、loadView中只初始化view，一般用于创建比较关键的view如tableViewController的tabView，UINavigationController的navgationBar，不可掉用view的getter（在掉super loadView前），最好也不要初始化一些非关键的view。如果你是从nib文件中创建的viewController在这里一定要首先调用super的loadView方法，但建议不要重载这个方法。

3、viewDidLoad 这时候view已经有了，最适合创建一些附加的view和控件了。

4、viewWillAppear 这个一般在view被添加到superview之前，切换动画之前调用。在这里可以进行一些显示前的处理。比如键盘弹出，一些特殊的过程动画（比如状态条和navigationbar颜色）。

5、viewDidAppear 一般用于显示后，在切换动画后，如果有需要的操作，可以在这里加入相关代码。

6、viewDidUnload 这时候viewController的view已经是nil了。由于这一般发生在内存警告时，所以在这里你应该将那些不在显示的view释放了。比如你在viewcontroller的view上加了一个label，而且这个label是viewcontroller的属性，那么你要把这个属性设置成nil，以免占用不必要的内存，而这个label在viewDidLoad时会重新创建。

7、接下来看看ViewController中的view是如何被卸载的：

当系统发出内存警告时，会调用didReceiveMemoeryWarning方法，如果当前有能被释放的view，系统会调用viewWillUnload方法来释放view，完成后调用viewDidUnload方法，

至此，view就被卸载了。此时原本指向view的变量要被置为nil，具体操作是在viewDidUnload方法中调用self.myButton = nil;

小结一下：loadView和viewDidLoad的区别就是，loadView时view还没有生成，viewDidLoad时，view已经生成了，loadView只会被调用一次，而viewDidLoad可能会被调用多次（View可能会被多次加载），当view被添加到其他view中之前，会调用viewWillAppear，之后会调用viewDidAppear。当view从其他view中移除之前，调用viewWillDisAppear，移除之后会调用viewDidDisappear。当view不再使用时，受到内存警告时，ViewController会将view释放并将其指向为nil。

ViewController的生命周期中各方法执行流程如下：init—>loadView—>viewDidLoad—>viewWillApper—>viewDidApper—>viewWillDisapper—>viewDidDisapper—>viewWillUnload->viewDidUnload—>dealloc

7. 如果页面 A 跳转到 页面 B，A 的viewDidDisappear方法和 B 的viewDidAppear方法哪个先调用？

1.A -->viewWillDisappear

2.B-->viewWillAppear

3.A-->viewDidDisappear

4.B-->viewDidAppear

8. block循环引用问题

在讲block的循环引用问题之前，我们需要先了解一下iOS的内存管理机制和block的基本知识

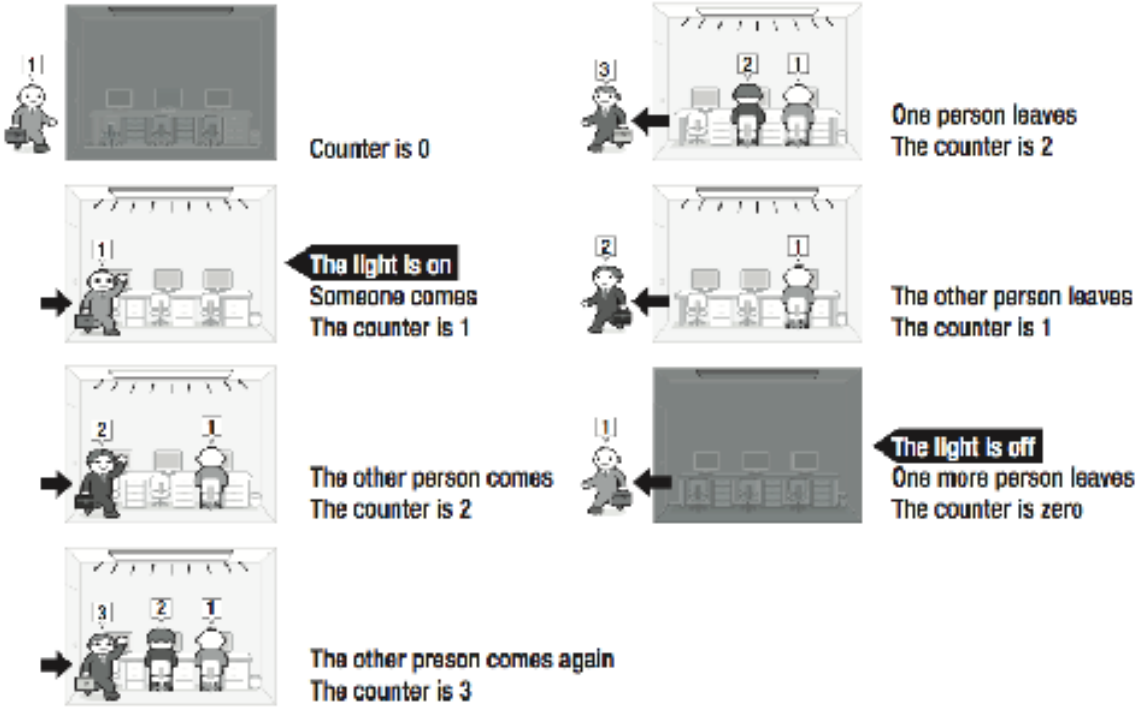
iOS的内存管理机制

Objective-C在iOS中不支持GC(垃圾回收)机制，而是采用的引用计数的方式管理内存。

引用计数(Reference Count)

在引用计数中，每一个对象负责维护对象所有引用的计数值。当一个新的引用指向对象时，引用计数器就递增，当去掉一个引用时，引用计数就递减。当引用计数到零时，该对象就将释放占有的资源。

我们通过开关房间的灯为例来说明引用计数机制。



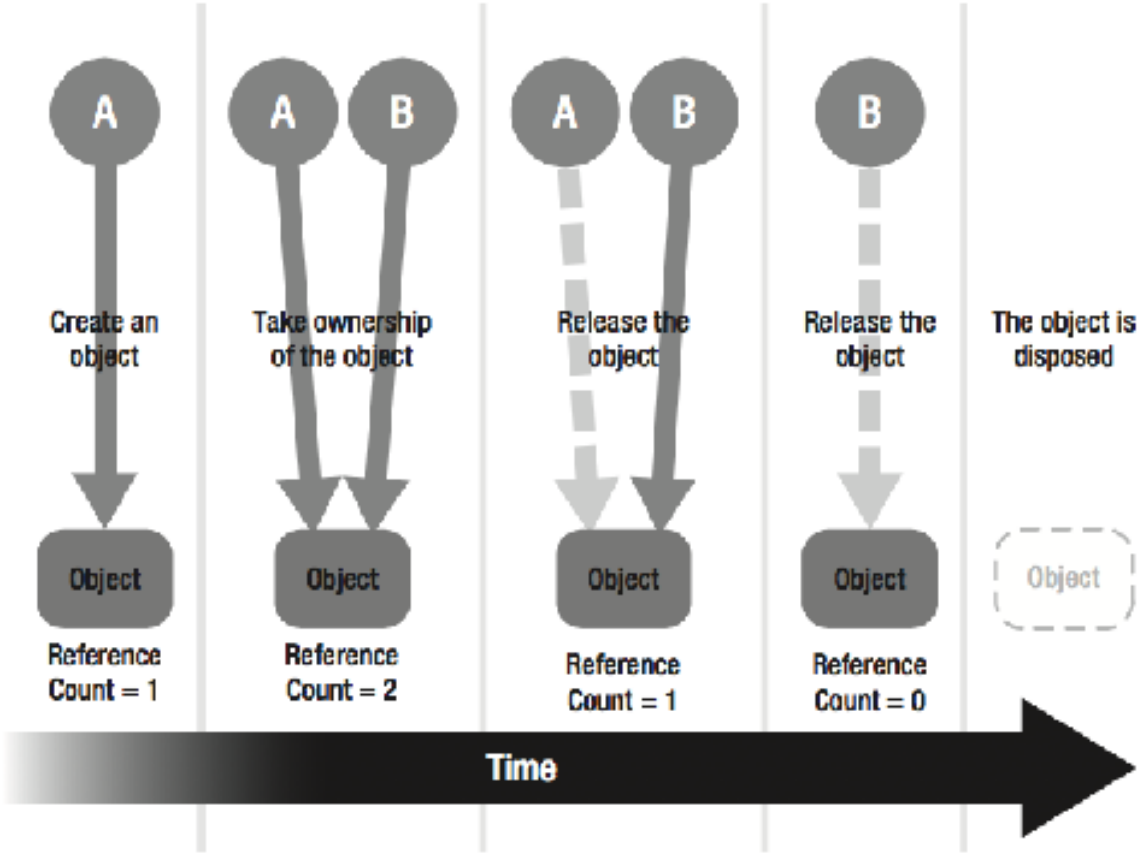
引用《Pro Multithreading and Memory Management for iOS and OS X》中的图片
图中，“需要照明的人数”即对应我们要说的引用计数值。

1. 第一个人进入办公室，“需要照明的人数”加1，计数值从0变为1，因此需要开灯；
2. 之后每当有人进入办公室，“需要照明的人数”就加1。如计数值从1变成2；
3. 每当有人下班离开办公室，“需要照明的人数”加减1如计数值从2变成1；
4. 最后一个人下班离开办公室时，“需要照明的人数”减1。计数值从1变成0，因此需要关灯。

在Objective-C中，”对象“相当于办公室的照明设备，”对象的使用环境“相当于进入办公室的人。上班进入办公室的人对办公室照明设备发出的动作，与Objective-C中的对应关系如下表

对照明设备所做的动作	对Objective-C对象所做的动作
开灯	生成对象
需要照明	持有对象
不需要照明	释放对象
关灯	废弃对象

使用计数功能计算需要照明的人数，使办公室的照明得到了很好的管理。同样，使用引用计数功能，对象也就能得到很好的管理，这就是Objective-C内存管理，如下图所示



引用《Pro Multithreading and Memory Management for iOS and OS X》中的图片

MRC(Manual Reference Counting)中引起应用计数变化的方法

Objective-C对象方法	说明
alloc/new/copy/ mutableCopy	创建对象，引用计数加1
retain	引用计数加1
release	引用计数减1
dealloc	当引用计数为0时调用
[NSArray array]	引用计数不增加，由自动释放池管理
[NSDictionary dictionary]	引用计数不增加，由自动释放池管理

自动释放池

关于自动释放，不是本文的重点，这里就不讲了。

ARC(Automatic Reference Counting)中内存管理

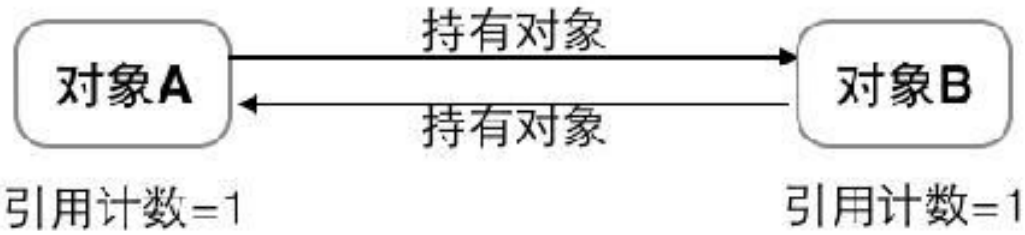
Objective-C对象所有权修饰符	说明
__strong	对象默认修饰符，对象强引用，在对象超出作用域时失效。其实就相当于retain操作，超出作用域时执行release操作
__weak	弱引用，不持有对象，对象释放时会将对象置nil。
__unsafe_unretained	弱引用，不持有对象，对象释放时不会将对象置nil。
__autoreleasing	自动释放，由自动释放池管理对象

block的基本知识

block的基本知识这里就不细说了

循环引用问题

两个对象相互持有，这样就会造成循环引用，如下图所示



两个对象相互持有
图中，对象A持有对象B，对象B持有对象A，相互持有，最终导致两个对象都不能释放。

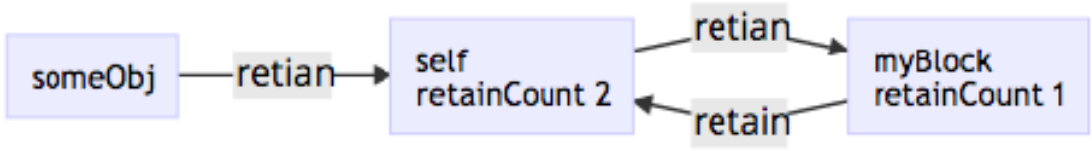
block中循环引用问题

由于block会对block中的对象进行持有操作,就相当于持有了其中的对象，而如果此时block中的对象又持有了该block，则会造成循环引用。如下，

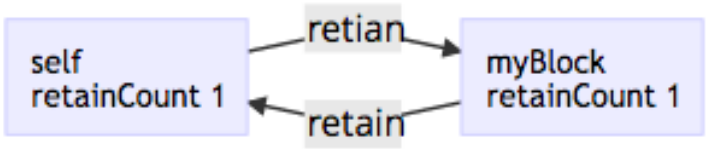
```
typedef void(^block)();

@property (copy, nonatomic) block myBlock;
@property (copy, nonatomic) NSString *blockString;

- (void)testBlock {
    self.myBlock = ^() {
        //其实注释中的代码，同样会造成循环引用
        NSString *localString = self.blockString;
          //NSString *localString = _blockString;
          //[self doSomething];
    };
```

}
注：以下调用注释掉的代码同样会造成循环引用，因为不管是通过**self.blockString**还是**_blockString**，或是函数调用**[self doSomething]**，因为只要 **block**中用到了对象的属性或者函数，**block**就会持有该对象而不是该对象中的某个属性或者函数。

当有someObj持有self对象，此时的关系图如下。



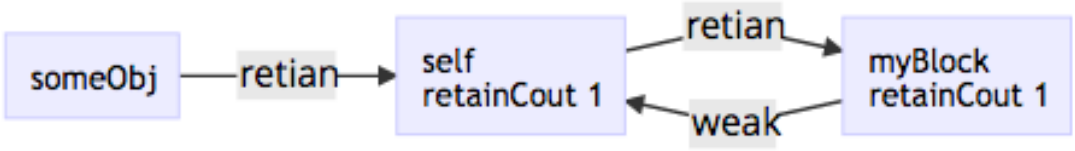
当someObj对象release self对象时，self和myblock相互引用，retainCount都为1，造成循环引用



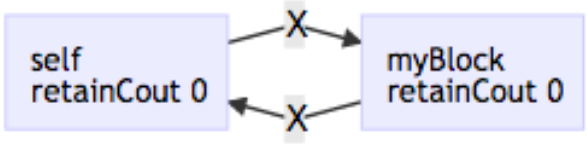
解决方法：

```
__weak typeof(self) weakSelf = self;
self.myBlock = ^() {
    NSString *localString = weakSelf.blockString;
};
```

使用__weak修饰self，使其在block中不被持有，打破循环引用。开始状态如下

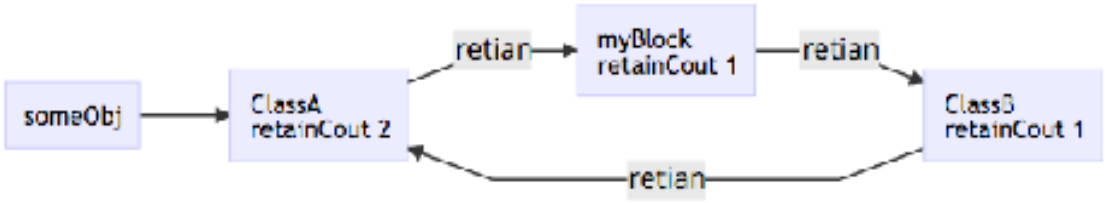


当someObj对象释放self对象时，Self的retainCount为0，走dealloc，释放myBlock对象，使其retainCount也为0。



其实以上循环引用的情况很容易发现，因为此时Xcode就会报警告。而发生在多个对象间的时候，Xcode就检测不出来了，这往往就容易被忽略。

```
//ClassB
@interface ClassB : NSObject
@property (strong, nonatomic) ClassA *objA;
- (void)doSomething;
@end

//ClassA
@property (strong, nonatomic) ClassB *objB;
@property (copy, nonatomic) block myBlock;

- (void)testBlockRetainCycle {
    ClassB* objB = [[ClassB alloc] init];
    self.myBlock = ^() {
        [objB doSomething];
    };
    objB.objA = self;
}
```



解决方法:

```
- (void)testBlockRetainCycle {
    ClassB* objB = [[ClassB alloc] init];
    __weak typeof(objB) weakObjB = objB;
    self.myBlock = ^() {
        [weakObjB doSomething];
    };
    objB.objA = self;
}
```

将objA对象weak，使其不在block中被持有

注：以上使用**__weak**打破循环的方法只在**ARC**下才有效，在**MRC**下应该使用**__block**

或者，在block执行完后，将block置nil，这样也可以打破循环引用

```
- (void)testBlockRetainCycle {
    ClassB* objB = [[ClassB alloc] init];
    self.myBlock = ^() {
        [objB doSomething];
    };
    objA.objA = self;
    self.myBlock();
```

```
    self.myBlock = nil;
}
```
这样做的缺点是，block只会执行一次，因为block被置nil了，要再次使用的话，需要重新赋值。

一些不会造成循环引用的block

在开发工程中，发现一些同学并没有完全理解循环引用，以为只要有block的地方就会要用__weak来修饰对象，这样完全没有必要，以下几种block是不会造成循环引用的。

大部分GCD方法

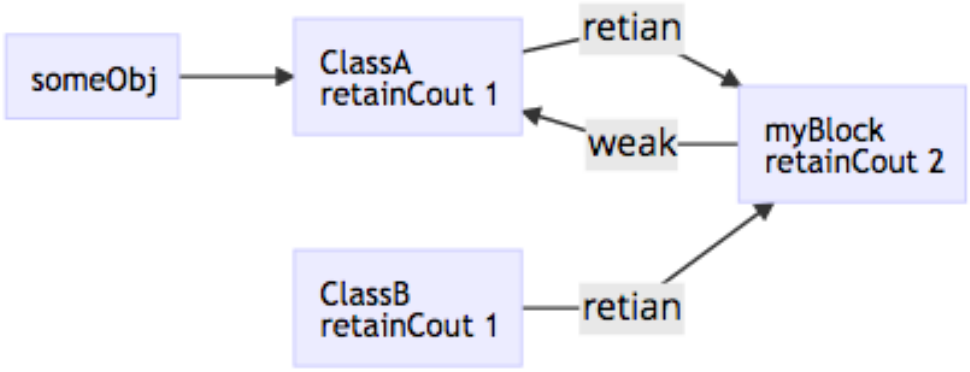
```
dispatch_async(dispatch_get_main_queue(), ^{
    [self doSomething];
});
```
因为self并没有对GCD的block进行持有，没有形成循环引用。目前我还没碰到使用GCD导致循环引用的场景，如果某种场景self对GCD的block进行了持有，则才有可能造成循环引用。

block并不是属性值，而是临时变量

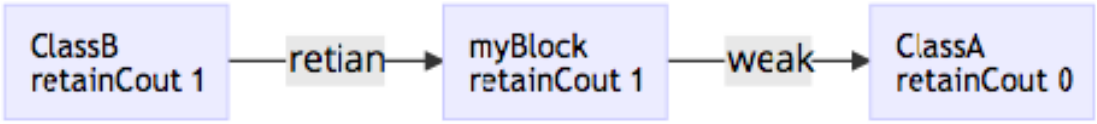
```
- (void)doSomething {
    [self testWithBlock:^{
        [self test];
    }];
}

- (void)testWithBlock:(void(^)())block {
    block();
}

- (void)test {
    NSLog(@"test");
}
```
这里因为block只是一个临时变量，self并没有对其持有，所以没有造成循环引用

block使用对象被提前释放

看下面例子，有这种情况，如果不只是ClassA持有了myBlock，ClassB也持有了myBlock。



当ClassA被someObj对象释放后



此时，ClassA对象已经被释放，而myBlock还是被ClassB持有，没有释放；如果myBlock这个时被调度，而此时ClassA已经被释放，此时访问的ClassA将是一个nil对象（使用__weak修饰，对象释放时会置为nil），而引发错误。

另一个常见错误使用是，开发者担心循环引用错误（如上所述不会出现循环引用的情况），使用__weak。比如

```
__weak typeof(self) weakSelf = self;
dispatch_async(dispatch_get_main_queue(), ^{
    [weakSelf doSomething];
});
```

因为将block作为参数传给dispatch_async时，系统会将block拷贝到堆上，而且block会持有block中用到的对象，因为dispatch_async并不知道block中对象会在什么时候被释放，为了确保系统调度执行block中的任务时其对象没有被意外释放掉，dispatch_async必须自己retain一次对象（即self），任务完成后再release对象（即self）。但这里使用__weak，使dispatch_async没有增加self的引用计数，这使得在系统在调度执行block之前，self可能已被销毁，但系统并不知道这个情况，导致block执行时访问已经被释放的self，而达不到预期的结果。

注：如果是在MRC模式下，使用__block修饰self,则此时block访问被释放的self，则会导致crash。

该场景下的代码

```
// ClassA.m
- (void)test {
```

```
    __weak MyClass* weakSelf = self;
    double delayInSeconds = 10.0f;
    dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW,
(int64_t)(delayInSeconds * NSEC_PER_SEC));
    dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
        NSLog(@"%@", weakSelf);
    });
}

// ClassB.m
- (void)doSomething {
    NSLog(@"do something");
    ClassA *objA = [[ClassA alloc] init];
    [objA test];
}
```

运行结果

```
[5988:435396] do something
[5988:435396] self:(null)
```

解决方法:

对于这种场景，就不应该使用__weak来修饰对象，让dispatch_after对self进行持有，保证block执行时self还未被释放。

block执行过程中对象被释放

还有一种场景，在block执行开始时self对象还未被释放，而执行过程中，self被释放了，此时访问self时，就会发生错误。

对于这种场景，应该在block中对 对象使用__strong修饰，使得在block期间对 对象持有，block执行结束后，解除其持有。

```
- (void)testBlockRetainCycle {
    ClassA* objA = [[ClassA alloc] init];
    __weak typeof(objA) weakObjA = objA;
    self.myBlock = ^() {
        __strong typeof(weakObjA) strongWeakObjA = weakObjA;
        [strongWeakObjA doSomething];
    };
    objA.objA = self;
}
```

注：此方法只能保证在block执行期间对象不被释放，如果对象在block执行执行之前已经被释放了，该方法也无效。

9. ARC的本质

ARC是编译器（时）特性，而不是运行时特性，更不是垃圾回收器(GC)。

Automatic Reference Counting (ARC) is a compiler-level feature that simplifies the process of managing object lifetimes (memory management) in Cocoa applications.

ARC只是相对于MRC（Manual Reference Counting或称为非ARC，下文中我们会一直使用MRC来指代非ARC的管理方式）的一次改进，但它和之前的技术本质上没有区别。具体信息可以参考ARC编译器官方文档。

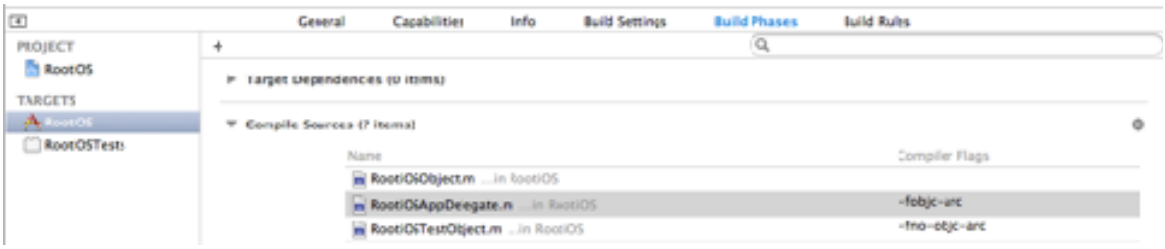
ARC的开启与关闭

不同于XCode4可以在创建工程时选择关闭ARC，XCode5在创建的工程是默认开启ARC，没有可以关闭ARC的选项。

如果需要对特定文件开启或关闭ARC，可以在工程选项中选择Targets -> Compile Phases -> Compile Sources，在里面找到对应文件，添加flag:

- 打开ARC：-fobjc-arc
- 关闭ARC：-fno-objc-arc

如图：



ARC的修饰符

ARC主要提供了4种修饰符，他们分别是:__strong,__weak,__autoreleasing,__unsafe_unretained。

__strong

表示引用为强引用。对应在定义property时的"strong"。所有对象只有当没有任何一个强引用指向时，才会被释放。

注意：如果在声明引用时不加修饰符，那么引用将默认是强引用。当需要释放强引用指向的对象时，需要将强引用置nil。

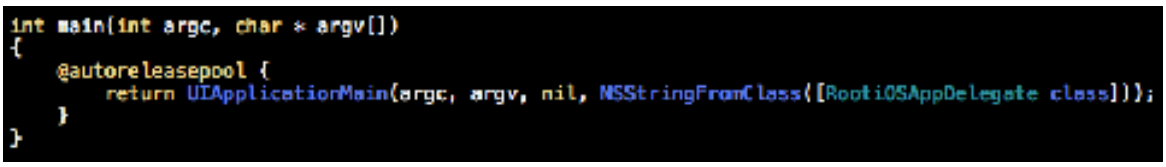
__weak

表示引用为弱引用。对应在定义property时用的"weak"。弱引用不会影响对象的释放，即只要对象没有任何强引用指向，即使有100个弱引用对象指向也没用，该对象依然会被释放。不过好在，对象在被释放的同时，指向它的弱引用会自动被置nil，这个技术叫zeroing weak pointer。这样有效得防止无效指针、野指针的产生。__weak一般用在delegate关系中防止循环引用或者用来修饰指向由Interface Builder编辑与生成的UI控件。

__autoreleasing

表示在autorelease pool中自动释放对象的引用，和MRC时代autorelease的用法相同。定义property时不能使用这个修饰符，任何一个对象的property都不应该是autorelease型的。

一个常见的误解是，在ARC中没有autorelease，因为这样一个"自动释放"看起来好像有点多余。这个误解可能源自于将ARC的"自动"和autorelease"自动"的混淆。其实你只要看一下每个iOS App的main.m文件就能知道，autorelease不仅好好的存在着，并且变得更fashion了：不需要再手工被创建，也不需要再显式得调用[drain]方法释放内存池。



```
int main(int argc, char * argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([RootiOSAppDelegate class]));
    }
}
```

以下两行代码的意义是相同的。

```
NSString *str = [[[NSString alloc] initWithFormat:@"hehe"]
autorelease]; // MRC
NSString *__autoreleasing str = [[NSString alloc]
initWithFormat:@"hehe"]; // ARC
```

这里关于autoreleasepool就不做展开了，详细地信息可以参考官方文档或者其他文章。

__autoreleasing在ARC中主要用在参数传递返回值（out-parameters）和引用传递参数（pass-by-reference）的情况下。

`__autoreleasing` is used to denote arguments that are passed by reference (`id *`) and are autoreleased on return.

比如常用的NSError的使用：

```
NSError *__autoreleasing error;
 if (![data writeToFile:filename options:NSDataWritingAtomic
error:&error])
 {
    NSLog(@"Error: %@", error);
}
```

（在上面的writeToFile方法中error参数的类型为(NSError *__autoreleasing *)）

注意，如果你的error定义为了strong型，那么，编译器会帮你隐式地做如下事情，保证最终传入函数的参数依然是个__autoreleasing类型的引用。

```
NSError *error;
NSError *__autoreleasing tempError = error; // 编译器添加
if (![data writeToFile:filename options:NSDataWritingAtomic
error:&tempError])
 {
    error = tempError; // 编译器添加
    NSLog(@"Error: %@", error);
}
```

所以为了提高效率，避免这种情况，我们一般在定义error的时候将其（老老实实地=。=）声明为__autoreleasing类型的：

```
NSError *__autoreleasing error;
```

在这里，加上__autoreleasing之后，相当于在MRC中对返回值error做了如下事情：

```
*error = [[[NSError alloc] init] autorelease];
```

*error指向的对象在创建出来后，被放入到了autoreleasing pool中，等待使用结束后的自动释放，函数外error的使用者并不需要关心*error指向对象的释放。

另外一点，在ARC中，所有这种指针的指针 （NSError **）的函数参数如果不加修饰符，编译器会默认将他们认定为__autoreleasing类型。

比如下面的两段代码是等同的：

```
- (NSString *)doSomething:(NSNumber **)value
{
        // do something
}
- (NSString *)doSomething:(NSNumber * __autoreleasing *)value
{
        // do something
}
```

除非你显式得给value声明了__strong，否则value默认就是__autoreleasing的。

最后一点，某些类的方法会隐式地使用自己的autorelease pool，在这种时候使用__autoreleasing类型要特别小心。

比如NSDictionary的[enumerateKeysAndObjectsUsingBlock]方法：

```
- (void)loopThroughDictionary:(NSDictionary *)dict error:(NSError
**)error
{
    [dict enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL
*stop){

          // do stuff
          if (there is some error && error != nil)
          {
                *error = [NSError errorWithDomain:@"MyError"
code:1 userInfo:nil];
          }

    }];
```

```
}
```

会隐式地创建一个autorelease pool，上面代码实际类似于：

```
- (void)loopThroughDictionary:(NSDictionary *)dict error:(NSError
**)error
{
    [dict enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL
*stop){

            @autoreleasepool  // 被隐式创建
            {
                if (there is some error && error != nil)
                {
                    *error = [NSError errorWithDomain:@"MyError"
code:1 userInfo:nil];
                }
            }
    }];

    // *error 在这里已经被dict的做枚举遍历时创建的autorelease pool释放掉
了 : (
 }
```

为了能够正常的使用*error，我们需要一个strong型的临时引用，在dict的枚举Block中是用这个临时引用，保证引用指向的对象不会在出了dict的枚举Block后被释放，正确的方式如下：

```
- (void)loopThroughDictionary:(NSDictionary *)dict error:(NSError
**)error
{
    __block NSError* tempError; // 加__block保证可以在Block内被修改
    [dict enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL
*stop)
    {
       if (there is some error)
       {
```

```
            *tempError = [NSError errorWithDomain:@"MyError"  code:1
userInfo:nil];
        }

    }]

    if (error != nil)
    {
        *error = tempError;
    }
}
```

__unsafe_unretained

ARC是在iOS 5引入的，而这个修饰符主要是为了在ARC刚发布时兼容iOS 4以及版本更低的设备，因为这些版本的设备没有weak pointer system，简单的理解这个系统就是我们上面讲weak时提到的，能够在weak引用指向对象被释放后，把引用值自动设为nil的系统。这个修饰符在定义property时对应的是"unsafe_unretained"，实际可以将它理解为MRC时代的assign：纯粹只是将引用指向对象，没有任何额外的操作，在指向对象被释放时依然原原本本地指向原来被释放的对象（所在的内存区域）。所以非常不安全。

现在可以完全忽略掉这个修饰符了，因为iOS 4早已退出历史舞台很多年。

*使用修饰符的正确姿势（方式=。=）

这可能是很多人都不知道的一个问题，包括之前的我，但却是一个特别要注意的问题。

苹果的文档中明确地写道：

You should decorate variables correctly. When using qualifiers in an object variable declaration,

the correct format is:

```
ClassName * qualifier variableName;
```

按照这个说明，要定义一个weak型的NSString引用，它的写法应该是：

```
NSString * __weak str = @"hehe"; // 正确!
```
而不应该是：

```
__weak NSString *str = @"hehe";  // 错误!
```
我相信很多人都和我一样，从开始用ARC就一直用上面那种错误的写法。

那这里就有疑问了，既然文档说是错误的，为啥编译器不报错呢？文档又解释道：

Other variants are technically incorrect but are “forgiven” by the compiler. To understand the issue, seehttp://cdecl.org/.

好吧，看来是苹果爸爸（=。=）考虑到很多人会用错，所以在编译器这边贴心地帮我们忽略并处理掉了这个错误：）虽然不报错，但是我们还是应该按照正确的方式去使用这些修饰符，如果你以前也常常用错误的写法，那看到这里记得以后不要这么写了，哪天编译器怒了，再不支持错误的写法，就要郁闷了。

栈中指针默认值为nil

无论是被strong，weak还是autoreleasing修饰，声明在栈中的指针默认值都会是nil。所有这类型的指针不用再初始化的时候置nil了。虽然好习惯是最重要的，但是这个特性更加降低了“野指针”出现的可能性。

在ARC中，以下代码会输出null而不是crash:)

```
- (void)myMethod
{
    NSString *name;
    NSLog(@"name: %@", name);
}
```

ARC与Block

在MRC时代，Block会隐式地对进入其作用域内的对象（或者说被Block捕获的指针指向的对象）加retain，来确保Block使用到该对象时，能够正确的访问。

这件事情在下面代码展示的情况中要更加额外小心。

```
MyViewController *myController = [[MyViewController alloc] init…];

// 隐式地调用[myController retain];造成循环引用
myController.completionHandler =  ^(NSInteger result) {
   [myController dismissViewControllerAnimated:YES
completion:nil];
};

[self presentViewController:myController animated:YES
completion:^{
   [myController release]; // 注意，这里调用[myController release];是在MRC中的一个常规写法，并不能解决上面循环引用的问题
}];
```

在这段代码中，myController的completionHandler调用了myController的方法[dismissViewController...]，这时completionHandler会对myController做retain操作。而我们知道，myController对completionHandler也至少有一个retain（一般准确讲是copy），这时就出现了在内存管理中最糟糕的情况：循环引用！简单点说就是：myController retain了completionHandler，而completionHandler也retain了myController。循环引用导致了myController和completionHandler最终都不能被释放。我们在delegate关系中，对delegate指针用weak就是为了避免这种问题。

不过好在，编译器会及时地给我们一个警告，提醒我们可能会发生这类型的问题：

对这种情况，我们一般用如下方法解决：给要进入Block的指针加一个__block修饰符。

这个__block在MRC时代有两个作用：

- 说明变量可改
- 说明指针指向的对象不做这个隐式的retain操作

一个变量如果不加__block，是不能在Block里面修改的，不过这里有一个例外：static的变量和全局变量不需要加__block就可以在Block中修改。

使用这种方法，我们对代码做出修改，解决了循环引用的问题：

```
MyViewController * __block myController = [[MyViewController
alloc] init…];
// ...
myController.completionHandler =  ^(NSInteger result) {
    [myController dismissViewControllerAnimated:YES
completion:nil];
};
//之后正常的release或者retain
```

在ARC引入后，没有了retain和release等操作，情况也发生了改变：在任何情况下，__block修饰符的作用只有上面的第一条：说明变量可改。即使加上了__block修饰符，一个被block捕获的强引用也依然是一个强引用。这样在ARC下，如果我们还按照MRC下的写法，completionHandler对myController有一个强引用，而myController对completionHandler有一个强引用，这依然是循环引用，没有解决问题：（

于是我们还需要对原代码做修改。简单的情况我们可以这样写：

```
__block MyViewController * myController = [[MyViewController
alloc] init…];
// ...
myController.completionHandler =  ^(NSInteger result) {
    [myController dismissViewControllerAnimated:YES
completion:nil];
    myController = nil;  // 注意这里，保证了block结束myController强引用
的解除
};
```

在completionHandler之后将myController指针置nil，保证了completionHandler对myController强引用的解除，不过也同时解除了myController对myController对象的强引用。这种方法过于简单粗暴了，在大多数情况下，我们有更好的方法。

这个更好的方法就是使用weak。（或者为了考虑iOS4的兼容性用unsafe_unretained，具体用法和weak相同，考虑到现在iOS4设备可能已

经绝迹了，这里就不讲这个方法了）（关于这个方法的本质我们后面会谈到）

为了保证completionHandler这个Block对myController没有强引用，我们可以定义一个临时的弱引用weakMyViewController来指向原myController的对象，并把这个弱引用传入到Block内，这样就保证了Block对myController持有的是一个弱引用，而不是一个强引用。如此，我们继续修改代码：

```
MyViewController *myController = [[MyViewController alloc] init…];
// ...
MyViewController * __weak weakMyViewController = myController;
myController.completionHandler =  ^(NSInteger result) {
    [weakMyViewController dismissViewControllerAnimated:YES
completion:nil];
};
```

这样循环引用的问题就解决了，但是却不幸地引入了一个新的问题：由于传入completionHandler的是一个弱引用，那么当myController指向的对象在completionHandler被调用前释放，那么completionHandler就不能正常的运作了。在一般的单线程环境中，这种问题出现的可能性不大，但是到了多线程环境，就很不好说了，所以我们需要继续完善这个方法。

为了保证在Block内能够访问到正确的myController，我们在block内新定义一个强引用strongMyController来指向weakMyController指向的对象，这样多了一个强引用，就能保证这个myController对象不会在completionHandler被调用前释放掉了。于是，我们对代码再次做出修改：

```
MyViewController *myController = [[MyViewController alloc] init…];
// ...
MyViewController * __weak weakMyController = myController;
myController.completionHandler =  ^(NSInteger result) {
    MyViewController *strongMyController = weakMyController;

    if (strongMyController) {
        // ...
        [strongMyController dismissViewControllerAnimated:YES
completion:nil];
```

```
        // ...
    }
    else {
        // Probably nothing...
    }
};
```

到此，一个完善的解决方案就完成了：）

官方文档对这个问题的说明到这里就结束了，但是可能很多朋友会有疑问，不是说不希望Block对原myController对象增加强引用么，这里为啥堂而皇之地在Block内新定义了一个强引用，这个强引用不会造成循环引用么？理解这个问题的关键在于理解被Block捕获的引用和在Block内定义的引用的区别。为了搞得明白这个问题，这里需要了解一些Block的实现原理，但由于篇幅的缘故，本文在这里就不展开了，详细的内容可以参考其他的文章，这里特别推荐唐巧的文章和另外2位作者的博文：这个和这个，讲的都比较清楚。

这里假设大家已经对Block的实现原理有所了解了。我们就直入主题了！注意前方高能（=。=）

为了更清楚地说明问题，这里用一个简单的程序举例。比如我们有如下程序：

```
#include <stdio.h>

int main()
{
    int b = 10;

    int *a = &b;

    void (^blockFunc)() = ^(){

        int *c = a;

    };

    blockFunc();
```

```
    return 1;
}
```

程序中，同为int型的指针，a是被Block捕获的变量，而c是在Block内定义的变量。我们用clang -rewrite-objc处理后，可以看到如下代码：

原main函数：

```
int main()
{
    int b = 10;

    int *a = &b;

    void (*blockFunc)() = (void (*)())&__main_block_impl_0((void
*)__main_block_func_0, &__main_block_desc_0_DATA, a);

    ((void (*)(__block_impl *))((__block_impl *)blockFunc)-
>FuncPtr)((__block_impl *)blockFunc);

    return 1;
}
```

Block的结构：

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;

  int *a; // 被捕获的引用 a 出现在了block的结构体里面

  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc,
int *_a, int flags=0) : a(_a) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

实际执行的函数：

```
static void __main_block_func_0(struct __main_block_impl_0
*__cself) {
  int *a = __cself->a; // bound by copy


        int *c = a; // 在block中声明的引用 c 在函数中声明，存在于函数栈上

    }
```

我们可以清楚得看到，a和c存在的位置完全不同，如果Block存在于堆上（在ARC下Block默认在堆上），那么a作为Block结构体的一个成员，也自然会存在于堆上，而c无论如何，永远位于Block内实际执行代码的函数栈内。这也导致了两个变量生命周期的完全不同：c在Block的函数运行完毕，即会被释放，而a呢，只有在Block被从堆上释放的时候才会释放。

回到我们的MyViewController的例子中，同上理，如果我们直接让Block捕获我们的myController引用，那么这个引用会被复制后（引用类型也会被复制）作为Block的成员变量存在于其所在的堆空间中，也就是为Block增加了一个指向myController对象的强引用，这就是造成循环引用的本质原因。对于MyViewController的例子，Block的结构体可以理解是这个样子：（准确的结构体肯定和以下这个有区别，但也肯定是如下这种形式：）

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;

  MyViewController * __strong myController;  // 被捕获的强引用
myController

  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc,
int *_a, int flags=0) : a(_a) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

而反观我们给Block传入一个弱引用weakMyController，这时我们Block的结构：

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;

  MyViewController * __weak weakMyController;  // 被捕获的弱引用weakMyController

  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int *_a, int flags=0) : a(_a) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

再看在Block内声明的强引用strongMyController，它虽然是强引用，但存在于函数栈中，在函数执行期间，它一直存在，所以myController对象也一直存在，但是当函数执行完毕，strongMyController即被销毁，于是它对myController对象的强引用也被解除，这时Block对myController对象就不存在强引用关系了！加入了strongMyController的函数大体会是这个样子：

```
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {

  MyViewController * __strong strongMyController = __cself->weakMyController;

    // ....
    }
```

综上所述，在ARC下（在MRC下会略有不同），Block捕获的引用和Block内声明的引用无论是存在空间与生命周期都是截然不同的，也正是这种不同，造成了我们对他们使用方式的区别。

以上就解释了之前提到的所有问题，希望大家能看明白：）

好的，最后再提一点，在ARC中，对Block捕获对象的内存管理已经简化了很多，由于没有了retain和release等操作，实际只需要考虑循环引用的问题就行了。比如下面这种，是没有内存泄露的问题的：

```
TestObject *aObject = [[TestObject alloc] init];

aObject.name = @"hehe";

self.aBlock = ^(){

    NSLog(@"aObject's name = %@",aObject.name);

};
```

我们上面提到的解决方案，只是针对Block产生循环引用的问题，而不是说所有的Block捕获引用都要这么处理，一定要注意！

ARC与Toll-Free Bridging

There are a number of data types in the Core Foundation framework and the Foundation framework that can be used interchangeably. This capability, called toll-free bridging, means that you can use the same data type as the parameter to a Core Foundation function call or as the receiver of an Objective-C message.

Toll-Free Briding保证了在程序中，可以方便和谐的使用Core Foundation类型的对象和Objective-C类型的对象。详细的内容可参考官方文档。以下是官方文档中给出的一些例子：

```
NSLocale *gbNSLocale = [[NSLocale alloc]
initWithLocaleIdentifier:@"en_GB"];
CFLocaleRef gbCFLocale = (CFLocaleRef) gbNSLocale;
CFStringRef cfIdentifier = CFLocaleGetIdentifier (gbCFLocale);
NSLog(@"cfIdentifier: %@", (NSString *)cfIdentifier);
// logs: "cfIdentifier: en_GB"
CFRelease((CFLocaleRef) gbNSLocale);

CFLocaleRef myCFLocale = CFLocaleCopyCurrent();
NSLocale * myNSLocale = (NSLocale *) myCFLocale;
[myNSLocale autorelease];
NSString *nsIdentifier = [myNSLocale localeIdentifier];
```

```
CFShow((CFStringRef) [@"nsIdentifier: "
stringByAppendingString:nsIdentifier]);
// logs identifier for current locale
```

在MRC时代，由于Objective-C类型的对象和Core Foundation类型的对象都是相同的release和retain操作规则，所以Toll-Free Bridging的使用比较简单，但是自从ARC加入后，Objective-C类型的对象内存管理规则改变了，而Core Foundation依然是之前的机制，换句话说，Core Foundation不支持ARC。

这个时候就必须要要考虑一个问题了，在做Core Foundation与Objective-C类型转换的时候，用哪一种规则来管理对象的内存。显然，对于同一个对象，我们不能够同时用两种规则来管理，所以这里就必须要确定一件事情：哪些对象用Objective-C（也就是ARC）的规则，哪些对象用Core Foundation的规则（也就是MRC）的规则。或者说要确定对象类型转换了之后，内存管理的ownership的改变。

If you cast between Objective-C and Core Foundation-style objects, you need to tell the compiler about the ownership semantics of the object using either a cast (defined in `objc/runtime.h`) or a Core Foundation-style macro (defined in`NSObject.h`)

于是苹果在引入ARC之后对Toll-Free Bridging的操作也加入了对应的方法与修饰符，用来指明用哪种规则管理内存，或者说是内存管理权的归属。

这些方法和修饰符分别是：

__bridge（修饰符）

只是声明类型转变，但是不做内存管理规则的转变。

比如：

```
CFStringRef s1 = (__bridge CFStringRef) [[NSString alloc]
initWithFormat:@"Hello, %@!", name];
```

只是做了NSString到CFStringRef的转化，但管理规则未变，依然要用Objective-C类型的ARC来管理s1，你不能用CFRelease()去释放s1。

__bridge_retained（修饰符） or CFBridgingRetain（函数）

表示将指针类型转变的同时，将内存管理的责任由原来的Objective-C交给Core Foundation来处理，也就是，将ARC转变为MRC。

比如，还是上面那个例子

```
NSString *s1 = [[NSString alloc] initWithFormat:@"Hello, %@!",
name];
 CFStringRef s2 = (__bridge_retained CFStringRef)s1;
 // do something with s2
//...
 CFRelease(s2); // 注意要在使用结束后加这个
```

我们在第二行做了转化，这时内存管理规则由ARC变为了MRC，我们需要手动的来管理s2的内存，而对于s1，我们即使将其置为nil，也不能释放内存。

等同的，我们的程序也可以写成：

```
NSString *s1 = [[NSString alloc] initWithFormat:@"Hello, %@!",
name];
 CFStringRef s2 = (CFStringRef)CFBridgingRetain(s1);
 // do something with s2
//...
 CFRelease(s2); // 注意要在使用结束后加这个
```

__bridge_transfer（修饰符） or CFBridgingRelease（函数）

这个修饰符和函数的功能和上面那个__bridge_retained相反，它表示将管理的责任由Core Foundation转交给Objective-C，即将管理方式由MRC转变为ARC。

比如：

```
CFStringRef result =
CFURLCreateStringByAddingPercentEscapes(. . .);
 NSString *s = (__bridge_transfer NSString *)result;
//or NSString *s = (NSString *)CFBridgingRelease(result);
 return s;
```

这里我们将result的管理责任交给了ARC来处理，我们就不需要再显式地将CFRelease()了。

对了，这里你可能会注意到一个细节，和ARC中那个4个主要的修饰符（__strong,__weak,...）不同，这里修饰符的位置是放在类型前面的，虽然官方文档中没有说明，但看官方的头文件可以知道。小伙伴们，记得别把位置写错哦：）

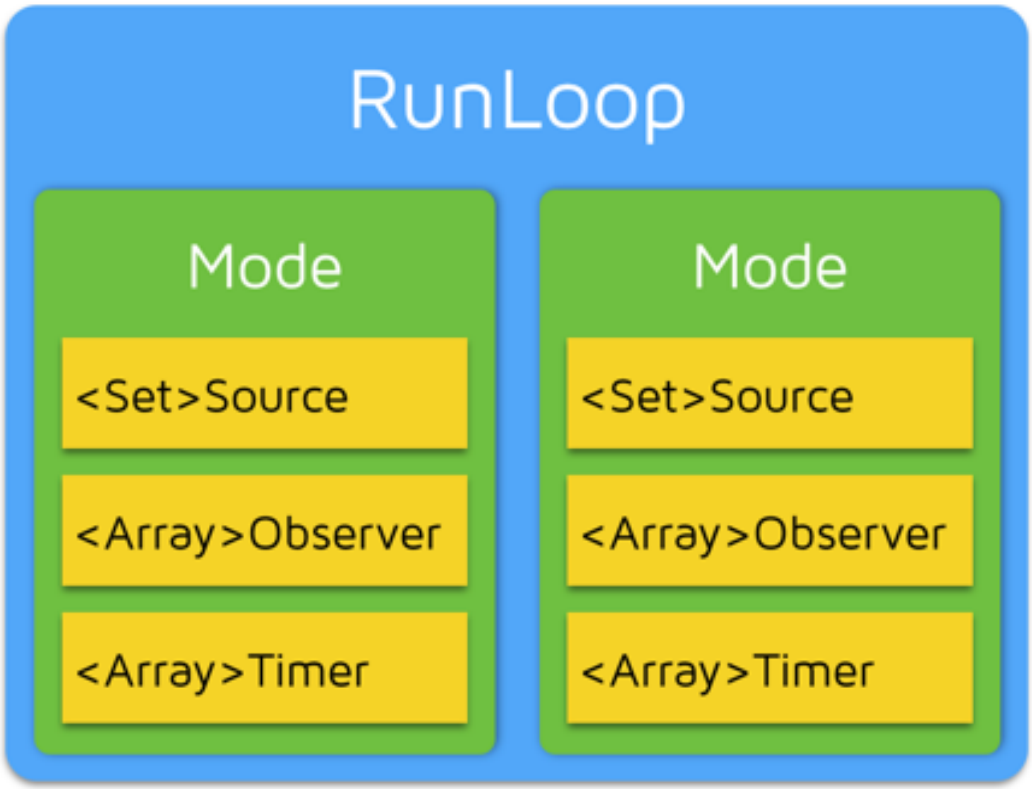
```
NS_INLINE CF_RETURNS_RETAINED CFTypeRef CFBridgingRetain(id X) {
    return (__bridge_retained CFTypeRef)X;
}

NS_INLINE id CFBridgingRelease(CFTypeRef CF_CONSUMED X) {
    return (__bridge_transfer id)X;
}
```

10.RunLoop的基本概念，它是怎么休眠的?

RunLoop是ios中一个非常重要的机制，ios系统底层很多模块都是通过RunLoop机制实现的，例如界面更新、事件响应等。本质上RunLoop是一种用于循环处理事件，而又不至于使CPU无意义空转的方式。

一、基本概念（了解过的可以跳过这一节）

1、NSRunLoop对象



（1）CFRunLoopRef

NSRunLoop对象是OC对象，是对CFRunLoopRef的封装，可以通过getCFRunLoop方法获取其对应的CFRunLoopRef对象。注意，NSRunLoop不是线程安全的，但CFRunLoopRef是线程安全的。

（2）RunLoopMode

NSRunLoop对象是一系列RunLoopMode的集合,每个mode包括有这个模式下所有的Source源、Timer源和观察者。每次RunLoop调用的时候都只能调用其中的一个mode，接收这个mode下的源，通知这个mode下的观察者。这样设计的主要目的就是为了隔离各个模式下的源和观察者，使其不相互影响。

其中系统默认注册的5个mode有：

- **kCFRunLoopDefaultMode**：App默认的mode，一般情况下App都是运行在这个mode下的。

- **UITrackingRunLoopMode**： 界面跟踪时的mode，一般用于ScrollView滚动的时候追踪的，保证滑动的时候不受其他事件影响。
- **UIInitializationRunLoopMode**： 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用。
- **GSEventReceiveRunLoopMode**： 接受系统事件的内部 Mode，一般用不到。
- **kCFRunLoopCommonModes**： 占位mode，可以向其中添加其他mode用以检测多个mode的事件

（3）CFRunLoopSourceRef

CFRunLoopSourceRef是事件源产生的地方，主要有两种：

- **Source0** :只包含一个函数指针（回调方法），不能自动触发，只能手动触发，触发方式是先通过CFRunLoopSourceSignal(source)将这个Source标记为待处理，然后再调用CFRunLoopWakeUp(runloop) 来唤醒RunLoop处理这个事件。

- **Source1** :基于port的Source源，包含一个port和一个函数指针（回调方法）。该Source源可通过内核和其他线程相互发送消息，而且可以主动唤醒RunLoop。

（4）CFRunLoopTimerRef

CFRunLoopTimerRef是基于事件的触发器，其中包含一段时间长度、延期容忍度和一个函数指针（回调方法）。当其加入到RunLoop中时，RunLoop会注册一个时间点，当到达这个时间点后，会触发对应的事件。

（5）performSEL

performSEL其实和NSTimer一样，是对CFRunLoopTimerRef的封装。因此，当调用performSelecter:afterDelay: 后，实际上内部会转化成CFRunLoopTimerRef并添加到当前线程的RunLoop中去，因此，如果当前线程中没有启动RunLoop的时候，该方法会失效。

（6）CFRunLoopObserverRef

CFRunLoopObserverRef是RunLoop的观察者。每个观察者都可以观察RunLoop在某个模式下事件的触发并处理。可以观察的时间点包括以下几点：

- **kCFRunLoopEntry**：即将进入RunLoop
- **kCFRunLoopBeforeTimers**：即将处理Timer
- **kCFRunLoopBeforeSources**：即将处理Source
- **kCFRunLoopBeforeWaiting**：即将进入休眠
- **kCFRunLoopAfterWaiting**：刚从休眠中被唤醒
- **kCFRunLoopExit**：即将退出RunLoop

（7）modeItem

上面的Source/Timer/Observer被统称为mode item，一个item可以被同时加入多个mode。但一个item被重复加入同一个mode时是不会有效果的。如果一个mode中一个item都没有，则RunLoop会直接退出，不进入循环。

2、RunLoop的驱动

```
BOOL isRunning = NO;
do {
        isRunning = [[NSRunLoop currentRunLoop]
runMode:NSDefaultRunLoopMode beforeDate:
[NSDatedistantFuture]];
} while (isRunning);
```

RunLoop本身是不能循环的，要通过外部的while循环循环驱动。

3、RunLoop内部的基本流程

每次运行RunLoop，内部都会处理之前没有处理的消息，并且在各个阶段通知相应的观察者。大致步骤如下：

（1）通知观察者RunLoop启动

（2）通知观察者即将处理Timer

（3）通知观察者即将处理Source0

（4）触发Source0回调

（5）如果有Source1（基于port）处于ready状态，直接处理该Source1然后跳转到（）去处理消息

（6）如果没有待处理消息，则通知观察者RunLoop所在线程即将进入休眠。

（7）休眠前，RunLoop会添加一个dispatchPort，底层调用mach_ msg接收mach_ port的消息。线程进入休眠，直到下面某个事件触发唤醒线程

- 基于port的Source1事件到达
- Timer时间到达
- RunLoop启动时设置的最大超时时间到了
- 手动唤醒

（8）唤醒后，将休眠前添加的dispatchPort移除，并通知观察者RunLoop已经被唤醒

（9）通过handle_ msg处理消息

（10）如果消息是Timer类型，则触发该Timer的回调

（11）如果消息是dispatch到main_ queue的block，执行block

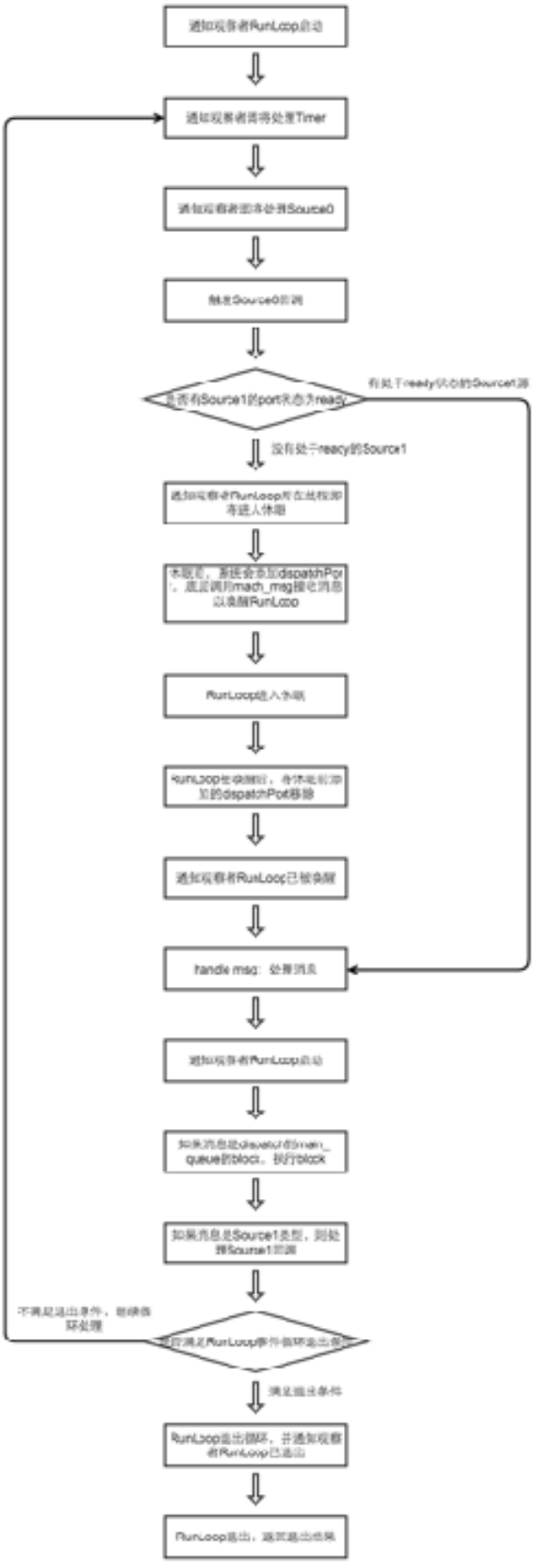
（12）如果消息是Source1类型，则处理Source1回调

（13）以下条件中满足时候退出循环，否则从（2）继续循环

- 事件处理完毕而且启动RunLoop的时候参数设置为一次性执行
- 启动RunLoop时设置的最大运行时间到期
- RunLoop被外部调用强行停止
- 启动RunLoop的mode items为空

（14）上一步退出循环后退出RunLoop，通知观察者RunLoop退出

流程图如下：


通知观察者RunLoop启动
通知观察者即将处理Timer
通知观察者即将处理Source0
触发Source0回调
是否有Source1的port状态为ready
有处于ready状态的Source1源
没有处于ready的Source1
通知观察者RunLoop即将进入休眠
休眠后，系统会添加dispatchPor
，底层调用mach_msg接收消息
以唤醒RunLoo
RunLoop进入休眠
RunLoop生命周期后，将休眠前添
加的dispatchPort移除
通知观察者RunLoop已被唤醒
handle msg：处理消息
通知观察者RunLoop启动
如果消息处dispatch_的main_
queue的block，执行block
如果消息是Source1类型，则处
理Source1回调
不满足退出条件，继续循
环处理
判断是否满足RunLoop事件循环退出条件
满足退出条件
RunLoop退出循环，并通知观察
者RunLoop已退出
RunLoop退出，返回退出结果

二、系统利用RunLoop处理事件

1、自动释放池的创建与释放

程序启动后，通过打印主线程的RunLoop，我们可以看到，系统在主线程自动生成了两个观察者。

```
<CFRunLoopObserver 0x7fad438034b0 [0x10c874a40]>{valid = Yes, activities = 0x1, repeats = Yes, order = -2147483647, callout = _wrapRunLoopWithAutoreleasePoolHandler (0x10c9fb4c2), context = <CFArray 0x7fad43803070 [0x10c874a40]>{type = mutable-small, count = 1, values = (
    0 : <0x7fad44000048>
)}}
...
<CFRunLoopObserver 0x7fad43809d30 [0x10c874a40]>{valid = Yes, activities = 0xa0, repeats = Yes, order = 2147483647, callout = _wrapRunLoopWithAutoreleasePoolHandler (0x10c9fb4c2), context = <CFArray 0x7fad43803070 [0x10c874a40]>{type = mutable-small, count = 1, values = (
    0 : <0x7fad44000048>
)}}
```

一个观察者监测RunLoop的kCFRunLoopEntry状态，最高优先级order = -2147483647。设置这个观察者的目的在于在RunLoop启动刚进入的时候生成自动释放池。由于优先级最高，因此先于所有其他回调操作生成。

另一个观察者监测RunLoop的kCFRunLoopBeforeWaiting状态和kCFRunLoopExit状态，此时是最低优先级order = 2147483647。该观察者的任务是当RunLoop在即将休眠时调用_objc_autoreleasePoolPop（）销毁自动释放池，根据池中的记录向池中所有对象发送realse方法真正地减少其引用计数，并调用_objc_autoreleasePoolPush（）创建新的自动释放池，当RunLoop即将退出的时候调用_objc_autoreleasePoolPop（）销毁池子。

而主线程中执行的代码，通常都是写在RunLoop事件回调，Timer回调中，处于自动释放池范围内，因此不会出现内存泄漏，开发人员也不需要显示在主线程创建自动释放池了。

oc中的自动释放池本质上就是延迟释放，将向自动释放池中对象发送的释放消息存在pool中，当pool即将被销毁的时候向其中所有对象发送realse消息使其计数减小。而自动释放池的创建和销毁也是由RunLoop控制的。

2、识别硬件和手势

当一个硬件事件（触摸、锁屏、摇晃、加速等）发生后，先由IOKit.framework生成一个IOHIDEEvent事件并由SpringBoard接收，然后由mach port转发到需要处理的App进程中。

```
<CFRunLoopSource 0x7fad43801e30 [0x10c874a40]>{signalled = No, valid = Yes, order = 0, context = <CFMachPort 0x7fad43801970 [0x10c874a40]>{valid = Yes, port = 1e03, source = 0x7fad43801e30, callout = __IOHIDEventSystemClientAvailabilityCallback (0x10eb2a444), context = <CFMachPort context 0x7fad42508470>}}
...
<CFRunLoopSource 0x7fad43801cb0 [0x10c874a40]>{signalled = No, valid = Yes, order = 0, context = <CFMachPort 0x7fad43801690 [0x10c874a40]>{valid = Yes, port = 1d03, source = 0x7fad43801cb0, callout = __IOHIDEventSystemClientQueueCallback (0x10eb2a293), context = <CFMachPort context 0x7fad42508470>}}
```

```
2
3
```

上面第一个Source源中是对硬件可用性变化敏感的，当某个硬件可用性发生变化就会触发这个Source源，进程就会在__IOHIDEventSystemClientAvailabilityCallback回调方法中处理

另一个是对硬件事件进行处理的，当有硬件事件发生时，底层就会包装成一个IOHIDEvent事件，通过mach port转发到该Source1源中，触发__IOHIDEventSystemClientQueueCallback回调，并调用_UIApplicationHandleEventQueue（）进行内部分发。这个方法会把IOHIDEvent进行预处理并包装成UIEvent进行处理和分发，包括UIGesture/处理屏幕旋转/发送给 UIWindow。

注意：当用户点击屏幕的时候最先是产生了一个硬件事件，底层封装后触发RunLoop的Source1源，回调__IOHIDEventSystemClientQueueCallback函数处理，然后再触发RunLoop中的一个Source0的源，回调_UIApplicationHandleEventQueue函数将事件包装成UIEvent处理并分发，通过函数调用栈可以看到：

```
0   test2
0x000000010024356d -[UITestGestureRecognizer
touchesBegan:withEvent:] + 157
1   UIKit
0x0000000101575bcf -[UIGestureRecognizer
_touchesBegan:withEvent:] + 113
2   UIKit
0x00000001010fb196 -[UIWindow _sendGesturesForEvent:] +
377
3   UIKit
0x00000001010fc6c4 -[UIWindow sendEvent:] + 849
4   UIKit
0x00000001010a7dc6 -[UIApplication sendEvent:] + 263
5   UIKit
0x0000000101081553 _UIApplicationHandleEventQueue + 6660
6   CoreFoundation
0x0000000100bf6301
```

```
__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION_
_ + 17
7   CoreFoundation
0x0000000100bec22c __CFRunLoopDoSources0 + 556
8   CoreFoundation
0x0000000100beb6e3 __CFRunLoopRun + 867
9   CoreFoundation
0x0000000100beb0f8 CFRunLoopRunSpecific + 488
10  GraphicsServices
0x0000000104550ad2 GSEventRunModal + 161
11  UIKit
0x0000000101086f09 UIApplicationMain + 171
12  test2
0x0000000100243adf main + 111
13  libdyld.dylib
0x00000001034aa92d start + 1
```

调用栈显示，处理事件相应的时候是由Source0分发的event。而实际上这个事件的识别和包装是由Source0处理的。

3、手势识别

```
1   test2
0x000000010c34e55f -[ViewController touchedButton] + 63
2   UIKit
0x000000010d68ab28 _UIGestureRecognizerSendTargetActions
+ 153
3   UIKit
0x000000010d68719a _UIGestureRecognizerSendActions + 162
4   UIKit
0x000000010d685197 -[UIGestureRecognizer
_updateGestureWithEvent:buttonEvent:] + 843
5   UIKit
0x000000010d68d655
___UIGestureRecognizerUpdate_block_invoke898 + 79
```

```
6   UIKit
0x000000010d68d4f3
_UIGestureRecognizerRemoveObjectsFromArrayAndApplyBlocks
+ 342
7   UIKit
0x000000010d67ae75 _UIGestureRecognizerUpdate + 2634
8   UIKit
0x000000010d20748e -[UIWindow _sendGesturesForEvent:] +
1137
9   UIKit
0x000000010d2086c4 -[UIWindow sendEvent:] + 849
10  UIKit
0x000000010d1b3dc6 -[UIApplication sendEvent:] + 263
11  UIKit
0x000000010d18d553 _UIApplicationHandleEventQueue + 6660
12  CoreFoundation
0x000000010cd02301
__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION_
_ + 17
13  CoreFoundation
0x000000010ccf822c __CFRunLoopDoSources0 + 556
14  CoreFoundation
0x000000010ccf76e3 __CFRunLoopRun + 867
15  CoreFoundation
0x000000010ccf70f8 CFRunLoopRunSpecific + 488
16  GraphicsServices
0x000000011065cad2 GSEventRunModal + 161
17  UIKit
0x000000010d192f09 UIApplicationMain + 171
18  test2
0x000000010c34eaef main + 111
19  libdyld.dylib
0x000000010f5b692d start + 1
```

上述是点击事件的调用栈，可见，当发生点击并包装好事件后RunLoop会进行事件分发，然后会触发_UIGestureRecognizerUpdate，并调用其中的回调函数，最后发送给对应GestureRecognizer的target。

再回看RunLoop，我们也可以看到：

```
...
<CFRunLoopSource 0x7fad4380a9b0 [0x10c874a40]>{signalled = No, valid = Yes, order = -1, context = <CFRunLoopSource context>{version = 0, info = 0x7fad43900630, callout = _UIApplicationHandleEventQueue (0x10c9fbb4f)}}
...
<CFRunLoopObserver 0x7fad4388e010 [0x10c874a40]>{valid = Yes, activities = 0x20, repeats = Yes, order = 0, callout = _UIGestureRecognizerUpdateObserver (0x10ceea41f), context = <CFRunLoopObserver context 0x0>}
```

系统默认注册了一个Source0源用以分发事件，回调_UIApplicationHandleEventQueue方法，然后系统会将对应的UIGestureRecognizer标记为待处理，然后还注册了一个观察者，监听RunLoop即将进入休眠的事件，回调_UIGestureRecognizerUpdateObserver（）函数，该函数内部会获取到所有被标记为待处理的UIGestureRecognizer，执行相应的回调。

4、界面刷新

当在操作 UI 时，比如改变了 Frame、更新了 UIView/CALayer 的层次时，或者手动调用了 UIView/CALayer 的 setNeedsLayout/setNeedsDisplay方法后，这个 UIView/CALayer 就被标记为待处理，并被提交到一个全局的容器去。

系统注册了一个观察者：

```
<CFRunLoopObserver 0x7fad42419d40 [0x10c874a40]>{valid =
Yes, activities = 0xa0, repeats = Yes, order = 2000000,
callout =
_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObse
rvermPv (0x110afb79c), context = <CFRunLoopObserver
context 0x0>}
```

这个观察者监听RunLoop即将进入休眠和即将退出的事件，回调_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv（）方法，该方法中会遍历所有待处理的UIView或CALayer以执行绘制和调整，更新UI界面。

三、日常用法

日常情况下，除了系统使用RunLoop，某些情况下我们也可以在底层合理使用RunLoop机制达到功能要求。

1、线程池

线程池的使用是为了避免程序使用线程的时候频繁地创建和销毁线程从而造成不必要的过度消耗，因此通过线程池机制维持几个常驻线程,避免使用的时候频繁创建和销毁，达到节省资源和加快响应速度的目的。

但是除了主线程外的线程默认是创建后执行完毕销毁的，这个时候，如果还需要维护住该线程，则需要手动创建该线程对应的RunLoop。RunLoop启动后，可以向该RunLoop添加一个port或者Timer用以触发线程接收自己生成的事件，从而分发处理。

在这种方式中，RunLoop执行的主要任务是维持线程不退出，然后应用就可以利用performSelector:onThread:等方式将自己的事件传递给RunLoop处理了。著名的开源框架AFNetworking就是这样生成网络线程的。

```
+ (void)networkRequestThreadEntryPoint:(id)__unused
object {
    @autoreleasepool {
```

```
        [[NSThread currentThread]
setName:@"AFNetworking"];
        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        [runLoop addPort:[NSMachPort port]
forMode:NSDefaultRunLoopMode];
        [runLoop run];
    }
}

+ (NSThread *)networkRequestThread {
    static NSThread *_networkRequestThread = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _networkRequestThread = [[NSThread alloc]
initWithTarget:self
selector:@selector(networkRequestThreadEntryPoint:)
object:nil];
        [_networkRequestThread start];
    });
    return _networkRequestThread;
}

...
- (void)start {
    [self.lock lock];
    if ([self isCancelled]) {
        [self performSelector:@selector(cancelConnection)
onThread:[[self class] networkRequestThread]
withObject:nil waitUntilDone:NO modes:[self.runLoopModes
allObjects]];
    } else if ([self isReady]) {
        self.state = AFOperationExecutingState;
        [self
performSelector:@selector(operationDidStart) onThread:
[[self class] networkRequestThread] withObject:nil
waitUntilDone:NO modes:[self.runLoopModes allObjects]];
```

```
    }
    [self.lock unlock];
}
```

2、通过监听RunLoop处理事件

通过添加Observer监听主线程RunLoop，可以仿造主线程RunLoop进行UI更新，例如Facebook的开源框架AsyncDisplayKit。

AsyncDisplayKit框架就是仿造QuartzCore/UIKit 框架的模式，实现了一套类似的界面更新的机制。平时的时候将UI对象的创建、设置属性的事件放在队列中，通过在主线程添加Observer监听主线程RunLoop的kCFRunLoopBeforeWaiting 和 kCFRunLoopExit事件，在收到回调时，遍历队列里的所有事件并执行。

因此，通过合理利用RunLoop机制，可以将很多不是必须在主线程中执行的操作放在子线程中实现，然后在合适的时机同步到主线程中，这样可以节省在主线程执行操作的时间，避免卡顿。

11. Autoreleasepool什么时候释放，在什么场景下使用?

autorelease 基本用法

1，对象执行autorelease方法时会将对象添加到自动释放池中
2，当自动释放池销毁时自动释放池中所有对象作release操作
3，对象执行autorelease方法后自身引用计数器不会改变，而且会返回对象本身

autoreleased 对象什么时候释放

autorelease 本质上就是延迟调用 release ，那 autoreleased 对象究竟会在什么时候释放呢？为了弄清楚这个问题，我们先来做一个小实验。这个小实验分 3 种场景进行，请你先自行思考在每种场景下的 console 输出，以加深理解。

注：本实验的源码可以在这里 AutoreleasePool 找到。

```
__weak NSString *string_weak_ = nil;
- (void)viewDidLoad {
    [super viewDidLoad];
    // 场景 1
    NSString *string = [NSString stringWithFormat:@"le
ichunfeng"];
    string_weak_ = string;
    // 场景 2
//    @autoreleasepool {
//        NSString *string = [NSString stringWithForma
t:@"leichunfeng"];
//        string_weak_ = string;
//    }
    // 场景 3
//    NSString *string = nil;
//    @autoreleasepool {
//        string = [NSString stringWithFormat:@"leichu
nfeng"];
//        string_weak_ = string;
//    }
    NSLog(@"string: %@", string_weak_);
}
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    NSLog(@"string: %@", string_weak_);
}
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    NSLog(@"string: %@", string_weak_);
}
```

思考得怎么样了？相信在你心中已经有答案了。那么让我们一起来看看 console 输出：

```
// 场景 1
2015-05-30 10:32:20.837 AutoreleasePool[33876:1448343]
 string: leichunfeng
2015-05-30 10:32:20.838 AutoreleasePool[33876:1448343]
 string: leichunfeng
2015-05-30 10:32:20.845 AutoreleasePool[33876:1448343]
 string: (null)
// 场景 2
2015-05-30 10:32:50.548 AutoreleasePool[33915:1448912]
 string: (null)
2015-05-30 10:32:50.549 AutoreleasePool[33915:1448912]
 string: (null)
2015-05-30 10:32:50.555 AutoreleasePool[33915:1448912]
 string: (null)
// 场景 3
2015-05-30 10:33:07.075 AutoreleasePool[33984:1449418]
 string: leichunfeng
2015-05-30 10:33:07.075 AutoreleasePool[33984:1449418]
 string: (null)
2015-05-30 10:33:07.094 AutoreleasePool[33984:1449418]
 string: (null)
```

跟你预想的结果有出入吗？Any way ，我们一起来分析下为什么会得到这样的结果。

分析：3 种场景下，我们都通过 [NSString stringWithFormat:@"leichunfeng"] 创建了一个 autoreleased 对象，这是我们实验的前提。并且，为了能够在 viewWillAppear 和 viewDidAppear 中继续访问这个对象，我们使用了一个全局的 __weak 变量 string_weak_ 来指向它。因为 __weak 变量有一个特性就是它不会影响所指向对象的生命周期，这里我们正是利用了这个特性。

场景 1：当使用 [NSString stringWithFormat:@"leichunfeng"] 创建一个对象时，这个对象的引用计数为 1 ，并且这个对象被系统自动添加到了当前的

autoreleasepool 中。当使用局部变量 string 指向这个对象时，这个对象的引用计数 +1 ，变成了 2 。因为在 ARC 下 NSString *string 本质上就是 __strong NSString *string 。所以在 viewDidLoad 方法返回前，这个对象是一直存在的，且引用计数为 2 。而当 viewDidLoad 方法返回时，局部变量 string 被回收，指向了 nil 。因此，其所指向对象的引用计数 -1 ，变成了 1 。

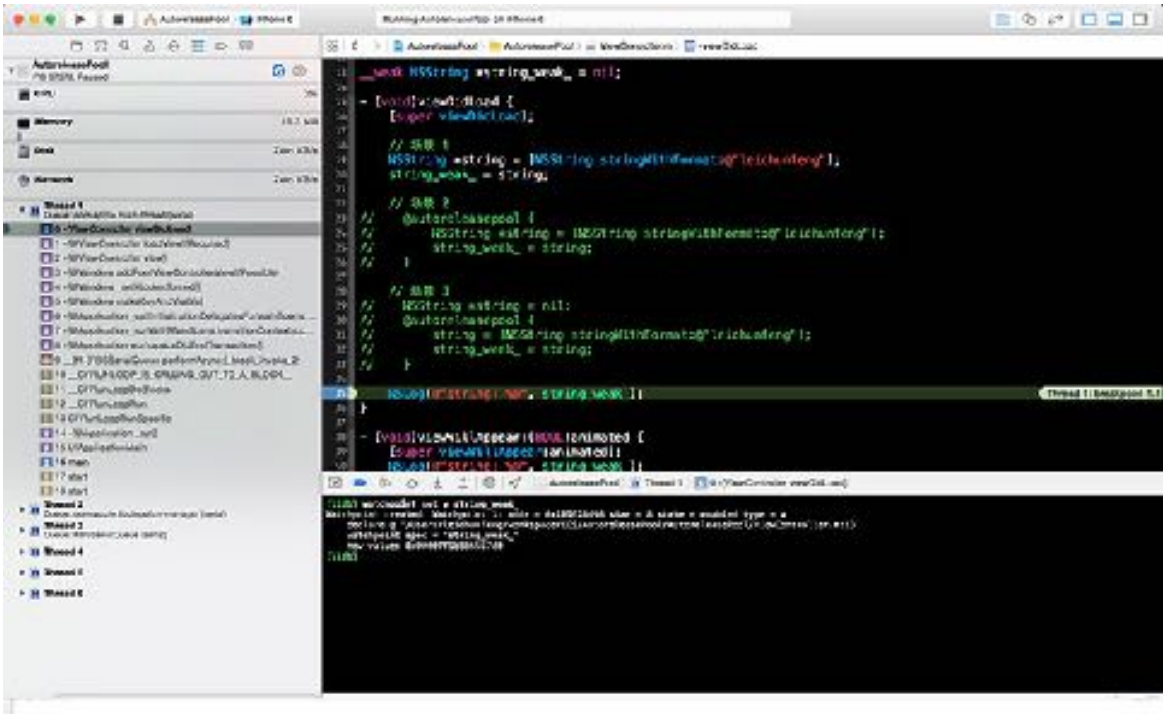
而在 viewWillAppear 方法中，我们仍然可以打印出这个对象的值，说明这个对象并没有被释放。咦，这不科学吧？我读书少，你表骗我。不是一直都说当函数返回的时候，函数内部产生的对象就会被释放的吗？如果你这样想的话，那我只能说：骚年你太年经了。开个玩笑，我们继续。前面我们提到了，这个对象是一个 autoreleased 对象，autoreleased 对象是被添加到了当前最近的 autoreleasepool 中的，只有当这个 autoreleasepool 自身 drain 的时候，autoreleasepool 中的 autoreleased 对象才会被 release 。

另外，我们注意到当在 viewDidAppear 中再打印这个对象的时候，对象的值变成了 nil ，说明此时对象已经被释放了。因此，我们可以大胆地猜测一下，这个对象一定是在 viewWillAppear 和 viewDidAppear 方法之间的某个时候被释放了，并且是由于它所在的 autoreleasepool 被 drain 的时候释放的。

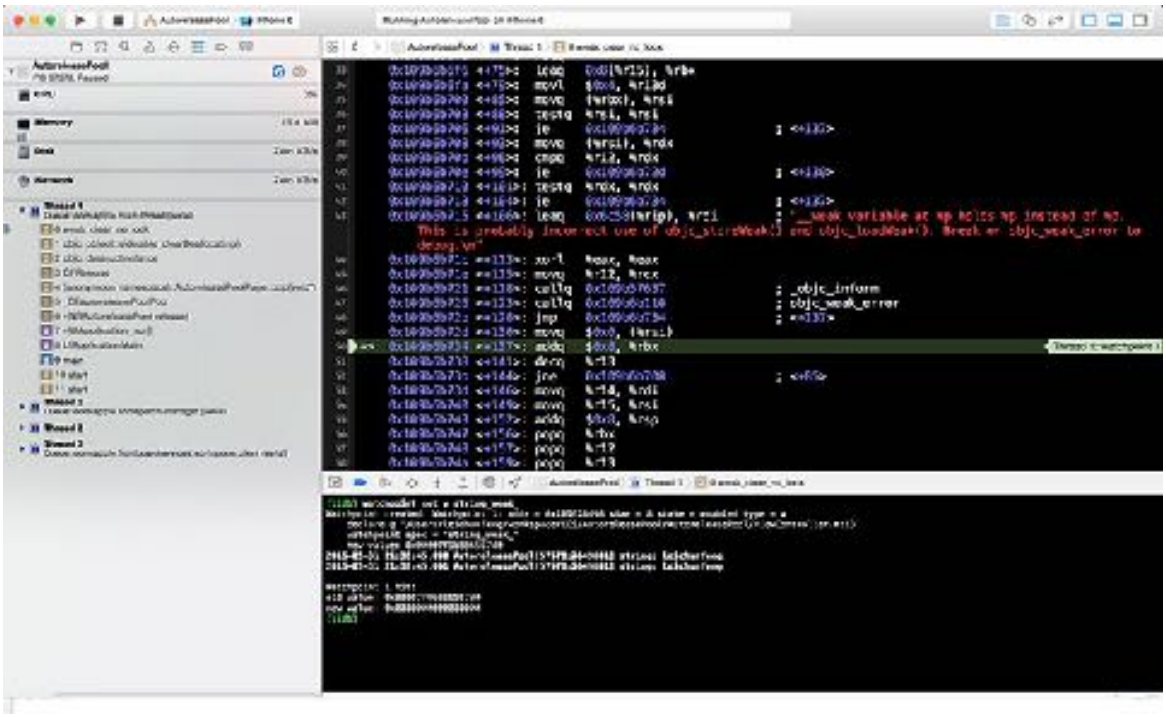
你说什么就是什么咯？有本事你就证明给我看你妈是你妈。额，这个我真证明不了，不过上面的猜测我还是可以证明的，不信，你看！

在开始前，我先简单地说明一下原理，我们可以通过使用 lldb 的 watchpoint 命令来设置观察点，观察全局变量 string_weak_ 的值的变化，string_weak_ 变量保存的就是我们创建的 autoreleased 对象的地址。在这里，我们再次利用了 __weak 变量的另外一个特性，就是当它所指向的对象被释放时，__weak 变量的值会被置为 nil 。了解了基本原理后，我们开始验证上面的猜测。

我们先在第 35 行打一个断点，当程序运行到这个断点时，我们通过 lldb 命令 watchpoint set v string_weak_ 设置观察点，观察 string_weak_ 变量的值的变化。如下图所示，我们将在 console 中看到类似的输出，说明我们已经成功地设置了一个观察点：



设置好观察点后，点击 Continue program execution 按钮，继续运行程序，我们将看到如下图所示的界面：



我们先看 **console** 中的输出，注意到 string_weak_ 变量的值由 0x00007f9b886567d0 变成了 0x0000000000000000 ，也就是 nil 。说明此时它所指向的对象被释放了。另外，我们也可以注意到一个细节，那就是 console 中打印了两次对象的值，说明此时 viewWillAppear 也已经被调用了，而 viewDidAppear 还没有被调用。

接着，我们来看看左侧的线程堆栈。我们看到了一个非常敏感的方法调用 -[NSAutoreleasePool release] ，这个方法最终通过调用 AutoreleasePoolPage::pop(void *) 函数来负责对 autoreleasepool 中的 autoreleased 对象执行 release 操作。结合前面的分析，我们知道在 viewDidLoad 中创建的 autoreleased 对象在方法返回后引用计数为 1 ，所以经过这里的 release 操作后，这个对象的引用计数 -1 ，变成了 0 ，该 autoreleased 对象最终被释放，猜测得证。

另外，值得一提的是，我们在代码中并没有手动添加 autoreleasepool ，那这个 autoreleasepool 究竟是哪里来的呢？看完后面的章节你就明白了。

场景 2：同理，当通过 [NSString stringWithFormat:@"leichunfeng"] 创建一个对象时，这个对象的引用计数为 1 。而当使用局部变量 string 指向这个对象时，这个对象的引用计数 +1 ，变成了 2 。而出了当前作用域时，局部变量 string 变成了 nil ，所以其所指向对象的引用计数变成 1 。另外，我们知道当出了 @autoreleasepool {} 的作用域时，当前 autoreleasepool 被 drain ，其中的 autoreleased 对象被 release 。所以这个对象的引用计数变成了 0 ，对象最终被释放。

场景 3：同理，当出了 @autoreleasepool {} 的作用域时，其中的 autoreleased 对象被 release ，对象的引用计数变成 1 。当出了局部变量 string 的作用域，即 viewDidLoad 方法返回时，string 指向了 nil ，其所指向对象的引用计数变成 0 ，对象最终被释放。

理解在这 3 种场景下，autoreleased 对象什么时候释放对我们理解 Objective-C 的内存管理机制非常有帮助。其中，场景 1 出现得最多，就是不需要我们手动添加 @autoreleasepool {} 的情况，直接使用系统维护的 autoreleasepool ；场景 2 就是需要我们手动添加 @autoreleasepool {} 的情况，手动干预 autoreleased 对象的释放时机；场景 3 是为了区别场景 2 而引入的，在这种场景下并不能达到出了 @autoreleasepool {} 的作用域时 autoreleased 对象被释放的目的。

PS：请读者参考场景 1 的分析过程，使用 lldb 命令 watchpoint 自行验证下在场景 2 和场景 3 下 autoreleased 对象的释放时机，you should give it a try yourself 。

AutoreleasePoolPage

细心的读者应该已经有所察觉，我们在上面已经提到了 -[NSAutoreleasePool release] 方法最终是通过调用 AutoreleasePoolPage::pop(void *) 函数来负责对 autoreleasepool 中的 autoreleased 对象执行 release 操作的。

那这里的 AutoreleasePoolPage 是什么东西呢？其实，autoreleasepool 是没有单独的内存结构的，它是通过以 AutoreleasePoolPage 为结点的**双向链表**来实现的。我们打开 runtime 的源码工程，在 NSObject.mm 文件的第 438-932 行可以找到 autoreleasepool 的实现源码。通过阅读源码，我们可以知道：

每一个线程的 autoreleasepool 其实就是一个指针的堆栈；

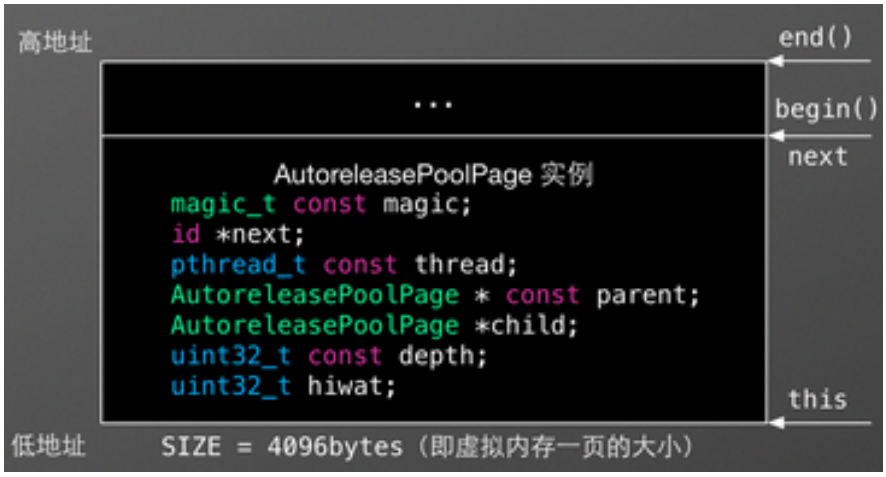
每一个指针代表一个需要 release 的对象或者 POOL_SENTINEL（哨兵对象，代表一个 autoreleasepool 的边界）；

一个 pool token 就是这个 pool 所对应的 POOL_SENTINEL 的内存地址。当这个 pool 被 pop 的时候，所有内存地址在 pool token 之后的对象都会被 release；

这个堆栈被划分成了一个以 page 为结点的双向链表。pages 会在必要的时候动态地增加或删除；

Thread-local storage（线程局部存储）指向 hot page，即最新添加的 autoreleased 对象所在的那个 page 。

一个空的 AutoreleasePoolPage 的内存结构如下图所示：



. magic 用来校验 AutoreleasePoolPage 的结构是否完整；

. next 指向最新添加的 autoreleased 对象的下一个位置，初始化时指向 begin()；

. thread 指向当前线程；

. parent 指向父结点，第一个结点的 parent 值为 nil；

. child 指向子结点，最后一个结点的 child 值为 nil；

. depth 代表深度，从 0 开始，往后递增 1；

. hiwat 代表 high water mark 。

另外，当 next == begin() 时，表示 AutoreleasePoolPage 为空；当 next == end() 时，表示 AutoreleasePoolPage 已满。

Autorelease Pool Blocks

我们使用 clang -rewrite-objc 命令将下面的 Objective-C 代码重写成 C++ 代码：

```
@autoreleasepool {
}
```

将会得到以下输出结果（只保留了相关代码）：

```
extern "C" __declspec(dllimport) void * objc_autoreleasePoolPush(void);
extern "C" __declspec(dllimport) void objc_autoreleasePoolPop(void *);
struct __AtAutoreleasePool {
  __AtAutoreleasePool() {atautoreleasepoolobj = objc_autoreleasePoolPush();}
  ~__AtAutoreleasePool() {objc_autoreleasePoolPop(atautoreleasepoolobj);}
  void * atautoreleasepoolobj;
};
/* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;
}
```

不得不说，苹果对 @autoreleasepool {} 的实现真的是非常巧妙，真正可以称得上是代码的艺术。苹果通过声明一个 __AtAutoreleasePool 类型的局部变量 __autoreleasepool 来实现 @autoreleasepool {} 。当声明 __autoreleasepool 变量时，构造函数 __AtAutoreleasePool() 被调用，即执行 atautoreleasepoolobj = objc_autoreleasePoolPush(); ；当出了当前作用域时，析构函数 ~__AtAutoreleasePool() 被调用，即执行 objc_autoreleasePoolPop(atautoreleasepoolobj); 。也就是说

@autoreleasepool {} 的实现代码可以进一步简化如下：

```
/* @autoreleasepool */ {
    void *atautoreleasepoolobj = objc_autoreleasePoolPush();
    // 用户代码，所有接收到 autorelease 消息的对象会被添加到这个 autoreleasepool 中
    objc_autoreleasePoolPop(atautoreleasepoolobj);
}
```

因此，单个 autoreleasepool 的运行过程可以简单地理解为 objc_autoreleasePoolPush()、[对象 autorelease] 和 objc_autoreleasePoolPop(void *) 三个过程。

push 操作

上面提到的 objc_autoreleasePoolPush() 函数本质上就是调用的 AutoreleasePoolPage 的 push 函数。

```
void *
objc_autoreleasePoolPush(void)
{
    if (UseGC) return nil;
    return AutoreleasePoolPage::push();
}
```

因此，我们接下来看看 AutoreleasePoolPage 的 push 函数的作用和执行过程。一个 push 操作其实就是创建一个新的 autoreleasepool ，对应 AutoreleasePoolPage 的具体实现就是往 AutoreleasePoolPage 中的 next 位置插入一个 POOL_SENTINEL ，并且返回插入的 POOL_SENTINEL 的内存地址。这个地址也就是我们前面提到的 pool token ，在执行 pop 操作的时候作

为函数的入参。

```
static inline void *push()
{
    id *dest = autoreleaseFast(POOL_SENTINEL);
    assert(*dest == POOL_SENTINEL);
    return dest;
}
```

push 函数通过调用 autoreleaseFast 函数来执行具体的插入操作。

```
static inline id *autoreleaseFast(id obj)
{
    AutoreleasePoolPage *page = hotPage();
    if (page && !page->full()) {
        return page->add(obj);
    } else if (page) {
        return autoreleaseFullPage(obj, page);
    } else {
        return autoreleaseNoPage(obj);
    }
}
```

autoreleaseFast 函数在执行一个具体的插入操作时，分别对三种情况进行了不同的处理：

. 当前 page 存在且没有满时，直接将对象添加到当前 page 中，即 next 指向的位置；

. 当前 page 存在且已满时，创建一个新的 page ，并将对象添加到新创建的 page 中；

. 当前 page 不存在时，即还没有 page 时，创建第一个 page ，并将对象添加到新创建的 page 中。

每调用一次 push 操作就会创建一个新的 autoreleasepool ，即往 AutoreleasePoolPage 中插入一个 POOL_SENTINEL ，并且返回插入的 POOL_SENTINEL 的内存地址。

autorelease 操作

通过 NSObject.mm 源文件，我们可以找到 -autorelease 方法的实现：

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

通过查看 ((id)self)->rootAutorelease() 的方法调用，我们发现最终调用的就是 AutoreleasePoolPage 的 autorelease 函数。

```
__attribute__((noinline,used))
id
objc_object::rootAutorelease2()
{
    assert(!isTaggedPointer());
    return AutoreleasePoolPage::autorelease((id)this);
}
```

AutoreleasePoolPage 的 autorelease 函数的实现对我们来说就比较容量理解了，它跟 push 操作的实现非常相似。只不过 push 操作插入的是一个 POOL_SENTINEL ，而 autorelease 操作插入的是一个具体的 autoreleased 对象。

```
static inline id autorelease(id obj)
{
    assert(obj);
    assert(!obj->isTaggedPointer());
    id *dest __unused = autoreleaseFast(obj);
    assert(!dest  ||  *dest == obj);
    return obj;
}
```

pop 操作

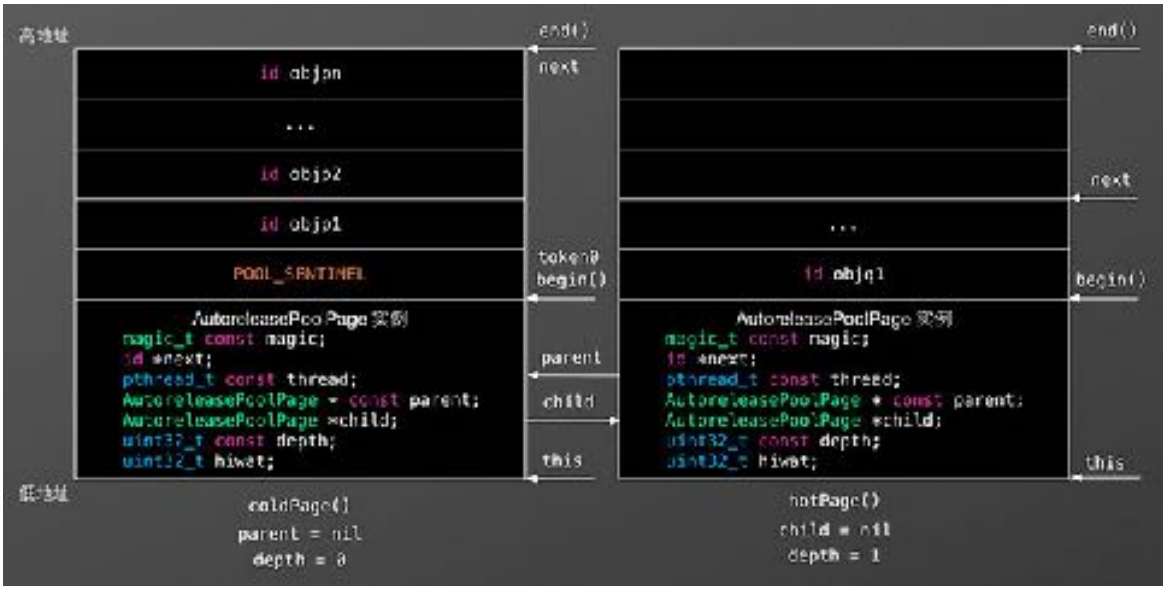
同理，前面提到的 objc_autoreleasePoolPop(void *) 函数本质上也是调用的 AutoreleasePoolPage 的 pop 函数。

```
void
objc_autoreleasePoolPop(void *ctxt)
{
    if (UseGC) return;
    // fixme rdar://9167170
    if (!ctxt) return;
    AutoreleasePoolPage::pop(ctxt);
}
```

pop 函数的入参就是 push 函数的返回值，也就是 POOL_SENTINEL 的内存地址，即 pool token 。当执行 pop 操作时，内存地址在 pool token 之后的所有 autoreleased 对象都会被 release 。直到 pool token 所在 page 的 next 指向 pool token 为止。

下面是某个线程的 autoreleasepool 堆栈的内存结构图，在这个 autoreleasepool 堆栈中总共有两个 POOL_SENTINEL ，即有两个 autoreleasepool 。该堆栈由三个 AutoreleasePoolPage 结点组成，第一个 AutoreleasePoolPage 结点为 coldPage() ，最后一个 AutoreleasePoolPage 结点为 hotPage() 。其中，前两个结点已经满了，最后一个结点中保存了最新添加的 autoreleased 对象 objr3 的内存地址。

此时，如果执行 pop(token1) 操作，那么该 autoreleasepool 堆栈的内存结构将会变成如下图所示：



NSThread、NSRunLoop 和 NSAutoreleasePool

根据苹果官方文档中对 NSRunLoop 的描述，我们可以知道每一个线程，包括主线程，都会拥有一个专属的 NSRunLoop 对象，并且会在有需要的时候自动创建。

Each NSThread object, including the application's main thread, has an NSRunLoop object automatically created for it as needed.

同样的，根据苹果官方文档中对 NSAutoreleasePool 的描述，我们可知，在主线程的 NSRunLoop 对象（在系统级别的其他线程中应该也是如此，比如通过 dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0) 获取到的线程）的每个 event loop 开始前，系统会自动创建一个 autoreleasepool ，并在 event loop 结束时 drain 。我们上面提到的场景 1 中

创建的 autoreleased 对象就是被系统添加到了这个自动创建的 autoreleasepool 中，并在这个 autoreleasepool 被 drain 时得到释放。

The Application Kit creates an autorelease pool on the main thread at the beginning of every cycle of the event loop, and drains it at the end, thereby releasing any autoreleased objects generated while processing an event.

另外，NSAutoreleasePool 中还提到，每一个线程都会维护自己的 autoreleasepool 堆栈。换句话说 autoreleasepool 是与线程紧密相关的，每一个 autoreleasepool 只对应一个线程。

Each thread (including the main thread) maintains its own stack of NSAutoreleasePool objects.

弄清楚 NSThread、NSRunLoop 和 NSAutoreleasePool 三者之间的关系可以帮助我们从整体上了解 Objective-C 的内存管理机制，清楚系统在背后到底为我们做了些什么，理解整个运行机制等。

总结

看到这里，相信你应该对 Objective-C 的内存管理机制有了更进一步的认识。通常情况下，我们是不需要手动添加 autoreleasepool 的，使用线程自动维护的 autoreleasepool 就好了。根据苹果官方文档中对 Using Autorelease Pool Blocks 的描述，我们知道在下面三种情况下是需要我们手动添加 autoreleasepool 的：

. 如果你编写的程序不是基于 UI 框架的，比如说命令行工具；

. 如果你编写的循环中创建了大量的临时对象；

. 如果你创建了一个辅助线程。

12.如何找到字符串中第一个不重复的字符

我在写这个程序时突然想起一个问题，就是C语言里给int一维数组初始化赋值的问题，比如我写：

int index[11]={0}; 那么此时数组中所有的元素初始化值均为0；但是我写：

int index[11]={-1};此时我以为所有元素的值均为-1了，其实不是这样的，当{}中的值的个数小于数组元素个数时，只有数组前面对应的元素被赋值了，而后面多余的元素被默认初始化为0了，所以这句话执行后，只有index[0]的值为-1，而其余元素均为0；但是如果我这样写：

int index[11]; 那么你也不要天真地以为所有元素默认值均为0，其实不然，这样写的话，所有元素的值均为随机值。

这就是数组初始化时我们应该注意的地方。好了，下面说说这个算法：

源代码如下：

```
#include <iostream>

using namespace std;

char findIt(const char *str);
int main()
{
   char str[]="iamastudenti";

   cout << findIt(str) << endl;
   return 0;
}

char findIt(const char *str)
{
   int count[26]={0};
```

```
    int index[26]={0}; //注意int数组初始化赋值时，如果写成={-1}是不能给所有元素初始化为-1的，只有第一个元素是-1，其余为默认值0
    unsigned int i;
    int pos;
    for(i=0;i<strlen(str);i++)
    {
        count[str[i]-'a']++;  //记录该字母出现的次数
       // cout<<count[str[i]-'a']<<endl;
        if(index[str[i]-'a']==0)
        {
            index[str[i]-'a']=i;  //记住该字母第一次出现时的索引
        }
    }
    pos=strlen(str);
    for(i=0;i<26;i++)
    {
        if(count[i]==1)  //找到只出现一次的字母
        {
            if(index[i]!=-1&&index[i]<pos)  //在只出现一次的字母中找出索引值最小的即可
            {
                pos=index[i];
            }
        }
    }
    if(pos<strlen(str))
        return str[pos];
    return '\0';
}
```

13.哈希表如何处理冲突

哈希法又称散列法、杂凑法以及关键字地址计算法等，相应的表称为哈希表。这种方法的基本思想是：首先在元素的关键字k和元素的存储位置p之间建立一个对应关系f，使得p=f(k)，f称为哈希函数。创建哈希表时，把关键字为k的元素直接存入地址为f(k)的单元；以后当查找关键字为k的元素时，再利用哈希函数计算出该元素的存储位置p=f(k)，从而达到按关键字直接存取元素的目的。

当关键字集合很大时，关键字值不同的元素可能会映象到哈希表的同一地址上，即 k1≠k2 ，但 H（k1）=H（k2），这种现象称为冲突，此时称k1和k2为

同义词。实际中，冲突是不可避免的，只能通过改进哈希函数的性能来减少冲突。

综上所述，哈希法主要包括以下两方面的内容：

1）如何构造哈希函数

2）如何处理冲突。

8.4.1 哈希函数的构造方法

构造哈希函数的原则是：①函数本身便于计算；②计算出来的地址分布均匀，即对任一关键字k，f(k) 对应不同地址的概率相等，目的是尽可能减少冲突。

下面介绍构造哈希函数常用的五种方法。

1．数字分析法

如果事先知道关键字集合，并且每个关键字的位数比哈希表的地址码位数多时，可以从关键字中选出分布较均匀的若干位，构成哈希地址。例如，有80个记录，关键字为8位十进制整数$d_1d_2d_3…d_7d_8$，如哈希表长取100，则哈希表的地址空间为：00~99。假设经过分析，各关键字中 d_4和d_7的取值分布较均匀，则哈希函数为：$h(key)=h(d_1d_2d_3…d_7d_8)=d_4d_7$。例如，$h(81346532)=43$，$h(81301367)=06$。相反，假设经过分析，各关键字中 d_1和d_8的取值分布极不均匀， d_1 都等于5，d_8 都等于2，此时，如果哈希函数为：$h(key)=h(d_1d_2d_3…d_7d_8)=d_1d_8$，则所有关键字的地址码都是52，显然不可取。

2．平方取中法

当无法确定关键字中哪几位分布较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为哈希地址。这是因为：平方后中间几位和关键字中每一位都相关，故不同关键字会以较高的概率产生不同的哈希地址。

例：我们把英文字母在字母表中的位置序号作为该英文字母的内部编码。例如K的内部编码为11，E的内部编码为05，Y的内部编码为25，A的内部编码为01，B的内部编码为02。由此组成关键字“KEYA”的内部代码为11052501，同理我们可以得到关键字“KYAB”、“AKEY”、“BKEY”的内部编码。之后对关键字进行平方运算后，取出第7到第9位作为该关键字哈希地址，如图8.23所示。

关键字	内部编码	内部编码的平方值	H(k)关键字的哈希地址
KEYA	11050201	122157778355001	778
KYAB	11250102	126564795010404	795
AKEY	1110525	1233265775625	265
BKEY	2110525	4454315775625	315

图8.23平方取中法求得的哈希地址

3．分段叠加法

这种方法是按哈希表地址位数将关键字分成位数相等的几部分（最后一部分可以较短），然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。具体方法有**折叠法**与**移位法**。移位法是将分割后的每部分低位对齐相加，折叠法是从一端向另一端沿分割界来回折叠（奇数段为正序，偶数段为倒序），然后将各段相加。例如：key=12360324711202065,哈希表长度为1000，则应把关键字分成3位一段，在此舍去最低的两位65，分别进行移位叠加和折叠叠加，求得哈希地址为105和907，如图8.24所示。

```
    1 2 3          1 2 3
    6 0 3          3 0 6
    2 4 7          2 4 7
    1 1 2          2 1 1
+)  0 2 0      +)  0 2 0
-----------    -----------
1 1 0 5        9 0 7
```

（a）移位叠加　　(b) 折叠叠加

图8.24 由叠加法求哈希地址

4．除留余数法

假设哈希表长为m，p为小于等于m的最大素数，则哈希函数为

h（k）=k ％ p，其中%为模p取余运算。

例如，已知待散列元素为（18，75，60，43，54，90，46），表长m=10，p=7，则有

h(18)=18 % 7=4　h(75)=75 % 7=5　h(60)=60 % 7=4

h(43)=43 % 7=1　h(54)=54 % 7=5　h(90)=90 % 7=6

h(46)=46 % 7=4

此时冲突较多。为减少冲突，可取较大的m值和p值，如m=p=13，结果如下:

h(18)=18 % 13=5　h(75)=75 % 13=10　h(60)=60 % 13=8

h(43)=43 % 13=4　h(54)=54 % 13=2　h(90)=90 % 13=12

h(46)=46 % 13=7

此时没有冲突，如图8.25所示。

0	1	2	3	4	5	6	7	8	9	10	11	12
		54		43	18		46	60		75		90

图8.25 除留余数法求哈希地址

5．伪随机数法

采用一个伪随机函数做哈希函数，即h(key)=random(key)。

在实际应用中，应根据具体情况，灵活采用不同的方法，并用实际数据测试它的性能，以便做出正确判定。通常应考虑以下五个因素：

- 计算哈希函数所需时间（简单）。
- 关键字的长度。
- 哈希表大小。
- 关键字分布情况。
- 记录查找频率

8.4.2 处理冲突的方法

通过构造性能良好的哈希函数，可以减少冲突，但一般不可能完全避免冲突，因此解决冲突是哈希法的另一个关键问题。创建哈希表和查找哈希表都会遇到冲突，两种情况下解决冲突的方法应该一致。下面以创建哈希表为例，说明解决冲突的方法。常用的解决冲突方法有以下四种：

1.　开放定址法

这种方法也称**再散列法**，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。这种方法有一个通用的再散列函数形式：

Hi=（H（key）+d_i）% m　i=1，2，…，n

其中H（key）为哈希函数，m 为表长，d_i称为**增量序列**。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下三种：

l　线性探测再散列

d_ii=1，2，3，…，m-1

这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

l　二次探测再散列

d_i=1^2，-1^2，2^2，-2^2，…，k^2，$-k^2$　(k<=m/2)

这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

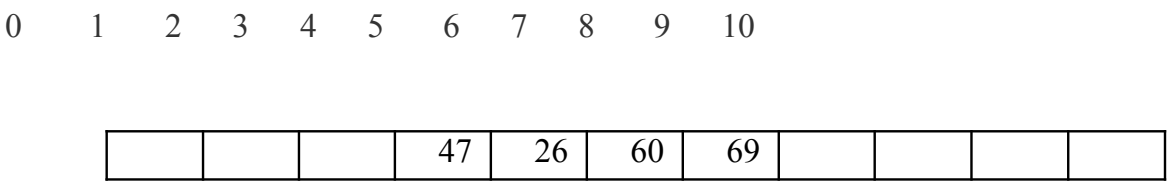
l　伪随机探测再散列

d_i=伪随机数序列。

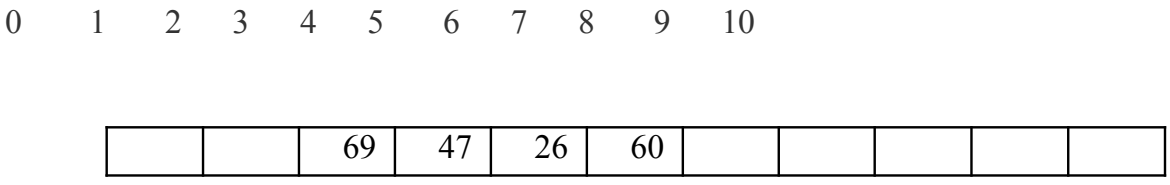
具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），并给定一个随机数做起点。

例如，已知哈希表长度m=11，哈希函数为：H（key）= key % 11，则H（47）=3，H（26）=4，H（60）=5，假设下一个关键字为69，则H（69）=3，与47冲突。如果用线性探测再散列处理冲突，下一个哈希地址为H1=（3 + 1）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 + 2）% 11 = 5，还是冲突，继续找下一个哈希地址为H3=（3 + 3）% 11 = 6，此时不再冲突，将69填入5号单元，参图8.26 (a)。如果用二次探测再散列处理冲突，下一个哈希地

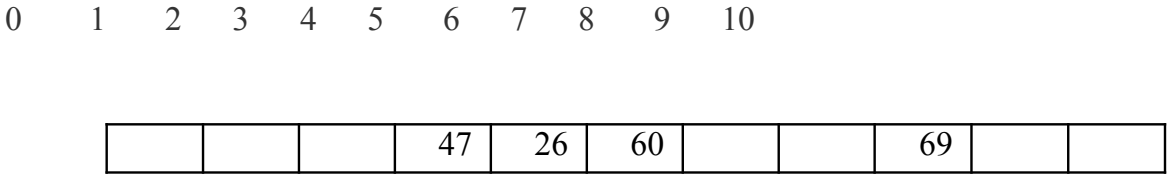
址为H1=（3 + 1^2）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 - 1^2）% 11 = 2，此时不再冲突，将69填入2号单元，参图8.26 (b)。如果用伪随机探测再散列处理冲突，且伪随机数序列为：2，5，9，……..，则下一个哈希地址为H1=（3 + 2）% 11 = 5，仍然冲突，再找下一个哈希地址为H2=（3 + 5）% 11 = 8，此时不再冲突，将69填入8号单元，参图8.26 (c)。



（a） 用线性探测再散列处理冲突



（b） 用二次探测再散列处理冲突



（c） 用伪随机探测再散列处理冲突

图8.26开放地址法处理冲突

从上述例子可以看出，线性探测再散列容易产生“二次聚集”，即在处理同义词的冲突时又导致非同义词的冲突。例如，当表中i，i+1 ,i+2三个单元已满时，下一个哈希地址为i，或i+1 ,或i+2，或i+3的元素，都将填入i+3这同一个单元，而这四个元素并非同义词。线性探测再散列的优点是：只要哈希表不满，

就一定能找到一个不冲突的哈希地址，而二次探测再散列和伪随机探测再散列则不一定。

2.　再哈希法

这种方法是同时构造多个不同的哈希函数：

$H_i=RH_1$（key）　i=1，2，…，k

当哈希地址$H_i=RH_1$（key）发生冲突时，再计算$H_i=RH_2$（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

3.　链地址法

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

例如，已知一组关键字（32，40，36，53，16，46，71，27，42，24，49，64），哈希表长度为13，哈希函数为：H（key）= key % 13，则用链地址法处理冲突的结果如图8.27所示：

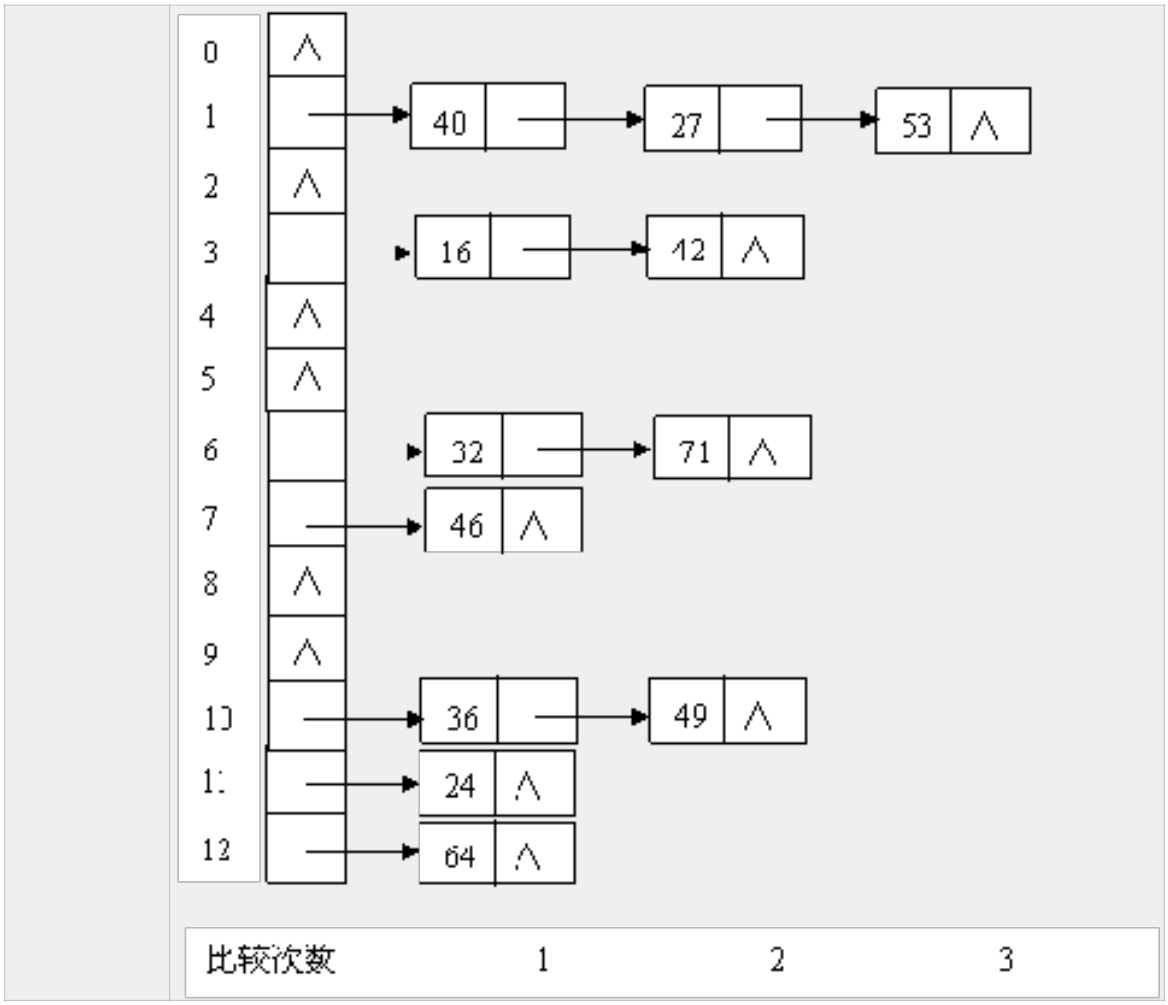


图8.27 链地址法处理冲突时的哈希表

本例的平均查找长度 ASL=(1*7+2*4+3*1)=1.5

4、建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表