xCode 4.3.3 下内存管理方法

# 一．自动引用计数(ARC)内存管理方法

## 1. 自动引用计数的基本概念

### 1)ARC下property的两个重要修饰符

strong：strong类型的指针，拥有对象所有权，可以保持对象生命周期。

weak: weak类型的指针，不拥有对象所有权，当对象释放时，该类指针会自动指向nil。

### 2)变量限定符

\_\_strong 默认限定符.一个对象，只要有strong类型的指针指向它，那么该对象的生命周期就没有结束。

\_\_weak 不能保持对象的生命周期，当对象释放时，该类指针会自动指向nil

\_\_unsafe\_unretained不能保持对象的生命周期，当对象释放时，该类指针会不会指向nil，而是变成野指针。

\_\_autoreleasing 用于修饰函数中需要返回的(id\*)类型的参数

### 3)自动释放池

使用@autoreleasepool{}创建自动释放池，不能使用NSAutoreleasePool 对象。

## 2. 自动引用计数需要遵循的规则

|  |  |  |
| --- | --- | --- |
| 编号 | 规则内容 | 备注 |
| 1 | 除了File's Owner，其他所有outlet属性都使用weak。 |  |
| 2 | dealloc方法不能调用实例对象的release，也不能调用[super dealloc]。但是可以处理其他非内存资源的释放。例如remove一个observer。 |  |
| 3 | 使用Delegate模式时，用weak @property声明delegate变量，避免所有权回环。 |  |
| 4 | 当出现循环引用时，父对象引用子对象使用strong型指针，子对象引用父对象使用weak型指针 |  |

### 1)规则1的示例

|  |
| --- |
| //将outlet定义为weak的优点是简化了viewDidUnload方法的实现：  //未将outlet定义为weak时。  - (void)viewDidUnload  {  [super viewDidUnload];  self.tableView = nil;  self.searchBar = nil;  soundEffect = nil;  }  //将outlet定义为weak 后，代码可以简化为如下  - (void)viewDidUnload  {  [super viewDidUnload];  soundEffect = nil;  }  因为tableView和searchBar这两个property定义为weak，当它们指向的对象被释放时，这两个变量会自动设置为nil。 |

### 2)规则2的示例

|  |
| --- |
| //Parent类  @interface Parent : NSObject  -(void)regedit:(id)objectObservered;//注册称为观察者，所要观察的对象是objectObservered  -(void)onScoreChanged;  @end  }  @implementation Parent  -(void)regedit:(id)objectObservered  {  //将自己注册为对象objectObservered 的观察者，事件名为score，处理函数是onScoreChanged  [[NSNotificationCenter defaultCenter]addObserver:self selector:@selector(onScoreChanged) name:@"score" object:objectObservered ];  }  -(void)onScoreChanged{  NSLog(@"Parent have seen your score\n");  }  -(void)dealloc{  //dealloc中调用removeObserver方法  [[NSNotificationCenter defaultCenter]removeObserver:self];  }  @end  //Child类  @interface Child : NSObject{  int score;  }  -(void )setScore:(int)sco;  @end  @implementation Child  //设置score  -(void)setScore:(int)sco{  if(score==sco){  return;  }  //当score发生变化时，消息中心就post一条事件名为 “score”的NSNotification  else{  score=sco;  [[NSNotificationCenter defaultCenter]postNotificationName:@"score" object:self];  }  }  @end |

### 3)规则3的示例

|  |
| --- |
| DetailViewController \*controller = [[DetailViewController alloc]  initWithNibName:@"DetailViewController" bundle:nil];  controller.delegate = self;  [self presentViewController:controller animated:YES  completion:nil];  在上面代码中，MainViewController创建一个DetailViewController，并调用presentViewController将view呈现出来，从而拥有了一个strong指针指向创建的DetailViewController对象。反过来，DetailViewController也通过delegate拥有了一个指向MainViewController的weak指针。 |

### 4)规则4的示例

|  |
| --- |
| //wheel类  @class car;  @interface wheel : NSObject  @property(weak)car \*ca;  @end  @implementation wheel  @synthesize ca;  @end  //car类  @interface car : NSObject  @property (strong)wheel \*w;  @end  @implementation car  @property (strong)wheel \*w;  @end  int main(int argc, const char \* argv[])  {  @autoreleasepool {  car \* ca=[[car alloc]init];  wheel \*wh=[[wheel alloc ]init];  [ca setW2:wh];  [wh setCa:ca];  }  return 0;  } |

# 二．手动内存管理方法

## 1． 手动内存管理的基本概念

-(id)retain; 引用计数加1

-(void) release ; 返回对象的引用计数

-(NSUInteger)retainCount; 引用计数

-(void)dealloc：销毁对象。

自动释放池(autoreleasepool):当自动释放池释放时，会给每个放在池中的对象发送一条autorelease消息。

-(id) autorelease ; 将对象放入自动释放池中，引用计数不变。

## 2． 手动内存管理需要遵循的规则

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 情况 | 规则内容 | |
| 池化模式 | 非池化模式 |
|  | 对象只在创建他的函数使用，没有被其他函数引用。 | 初始化后调用autorelease。 | 这种情况下可以明确的知道对象的生命周期，所以使用retain和release,比较方便。 |
|  | 对象需要返回给其他函数。 | 对象初始化后，调用autorelease。被赋值对象要调用一次retain。 | 这种情况下，不推荐使用引用计数的方式进行手动释放对象，因为要知道发送方对对象的处理是很麻烦的事情。 |
|  | 复合对象 | 1. 复合对象的构建顺序:父对象-〉子对象   重载init函数，先初始化父对象，在初始化子对象。  2．复合对象释放的顺序:子对象->父对象。  重载dealloc函数，在dealloc函数中先调用子对象的release,再调用父对象的dealloc方法。 | |
| 在池化状态下，复合对象的初始化后，调用autorelease。 | 在非池化状态下，初始化复合对象后。如果进行复合对象的retain和release方法的调用，只对父对象进行操作。 |

### 1)规则1的示例

|  |
| --- |
| @interface core : NSObject  @end  @implementation core  @end  @interface CallClass : NSObject  -(void) callMethod;  @end  @implementation CallClass  -(void)callMethod{  core\* co=[[core alloc]init];//count=1  [co retain]; // count=2;    [co release]; //count=1;    [co release]; //count=0;自动调用了［co dealloc］  return ;  }  @end |

### 2)规则2的示例

|  |
| --- |
| / /core类  @interface core : NSObject  @end  @implementation core  @end  //CalledClass类  @interface CalledClass : NSObject  -(core \*)calledMethod;  @end  @implementation CalledClass  -(core\*)calledMethod  {  core \*co;  co=[[core alloc]init] ; //count=1  return [co autorelease];//count=1  }  @end  int main(int argc, const char \* argv[])  {  @autoreleasepool {  core \*co;  CalledClass \*called = [[CalledClass alloc]init] ;  co = [called calledMethod];//count=1;  [co retain]; //赋值完后需要将被赋值对象retain，count=2    [co release]; //使用完后将被赋值对象释放 ,coun =1  }//count=0  return 0;  } |

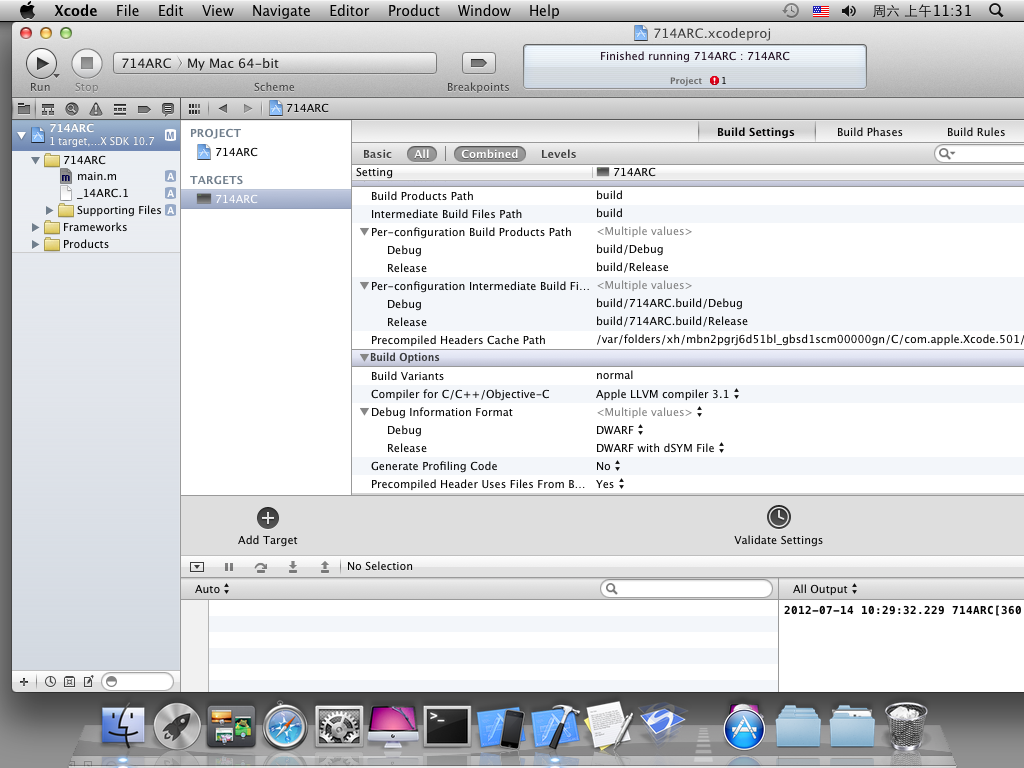
### 3)规则3的示例

|  |
| --- |
| //Wheel 类  @interface Wheel : NSObject  @end  @implementation Wheel  @end  @interface Car : NSObject  {  Wheel \_\_weak \*wheel;  }  -(void)setWheel:(Wheel \*)w;  @end  @implementation Car  -(id)init  {  if (self=[super init])  {  //wheel=[[Wheel alloc]init];  }  return self;  }  -(void)setWheel:(Wheel \*)w  {  [w retain];  if (wheel !=nil){  [wheel release];  }  wheel=w;  NSLog(@"%lu",[wheel retainCount]);  }  -(void)dealloc  {  [wheel release];  [super dealloc];  NSLog(@"WELL DONE when car destory!\n");  }  @end  int main(int argc, const char \* argv[])  {  @autoreleasepool {  Wheel \*wheel=[[Wheel alloc]init];  Car \*car=[[Car alloc]init];  [car setWheel:wheel];  [car release];  [wheel release];  }  return 0;  } |

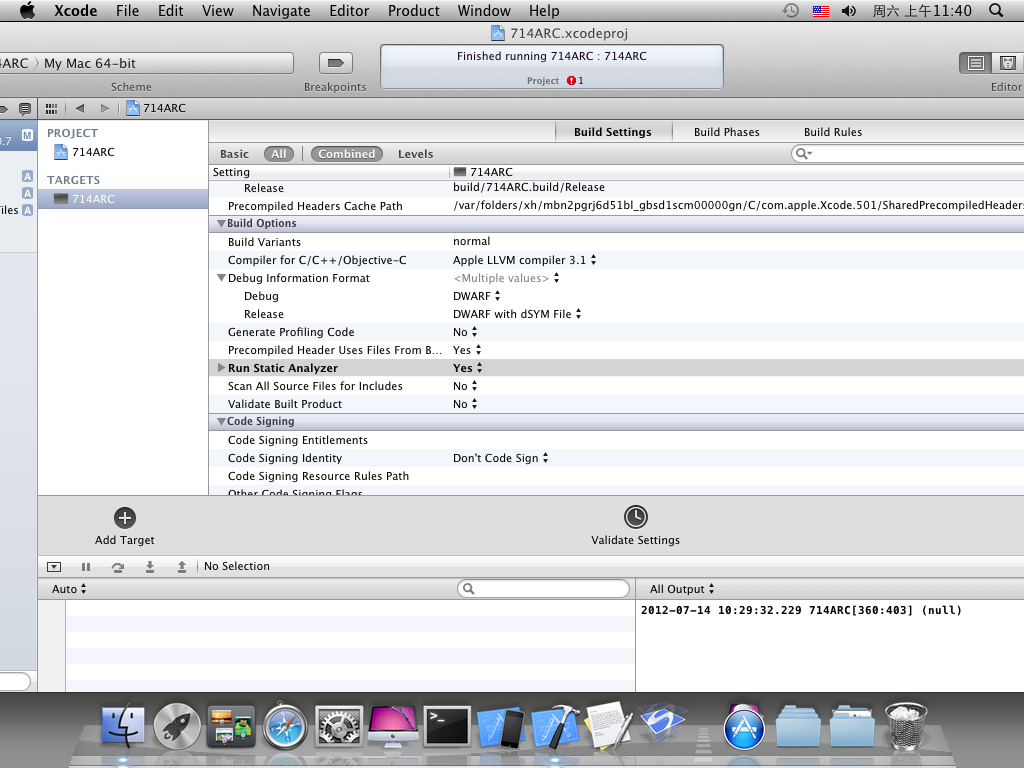
# 三．内存泄漏检测工具

## 1．静态内存泄漏检测工具

1）单击项目名(例如”714ARC”),选中”Bulid Settings”.



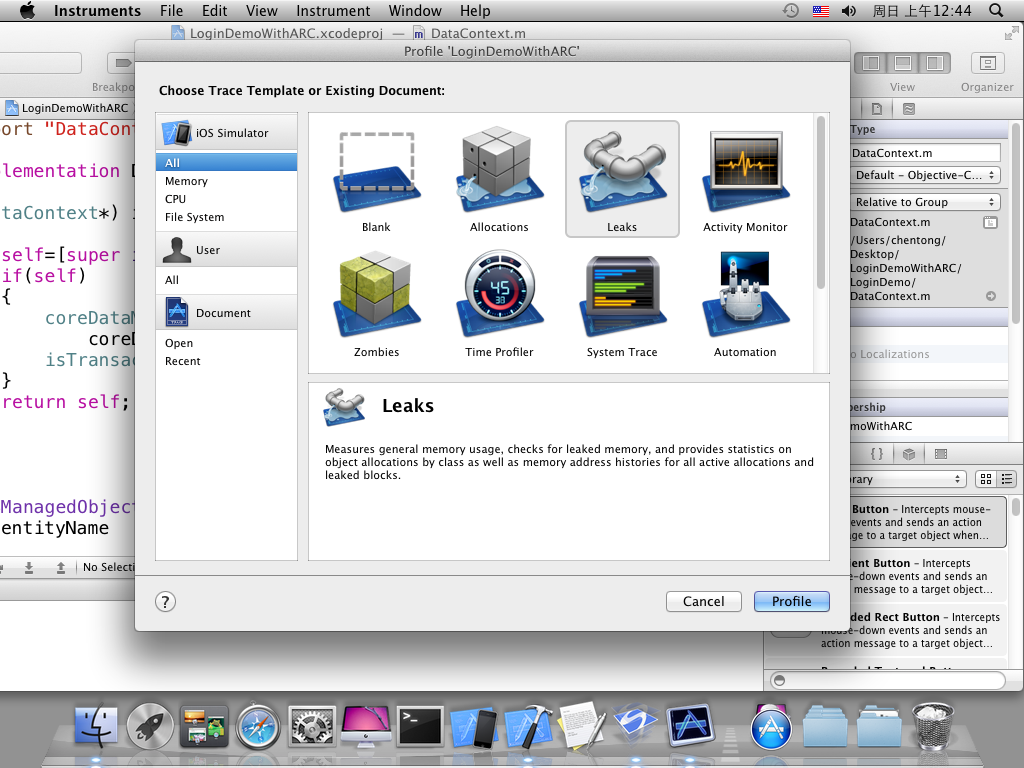
2) 选择”All”,然后在”Build Options”下找到”Run Static Analyzer”,并将值改为”Yes”.



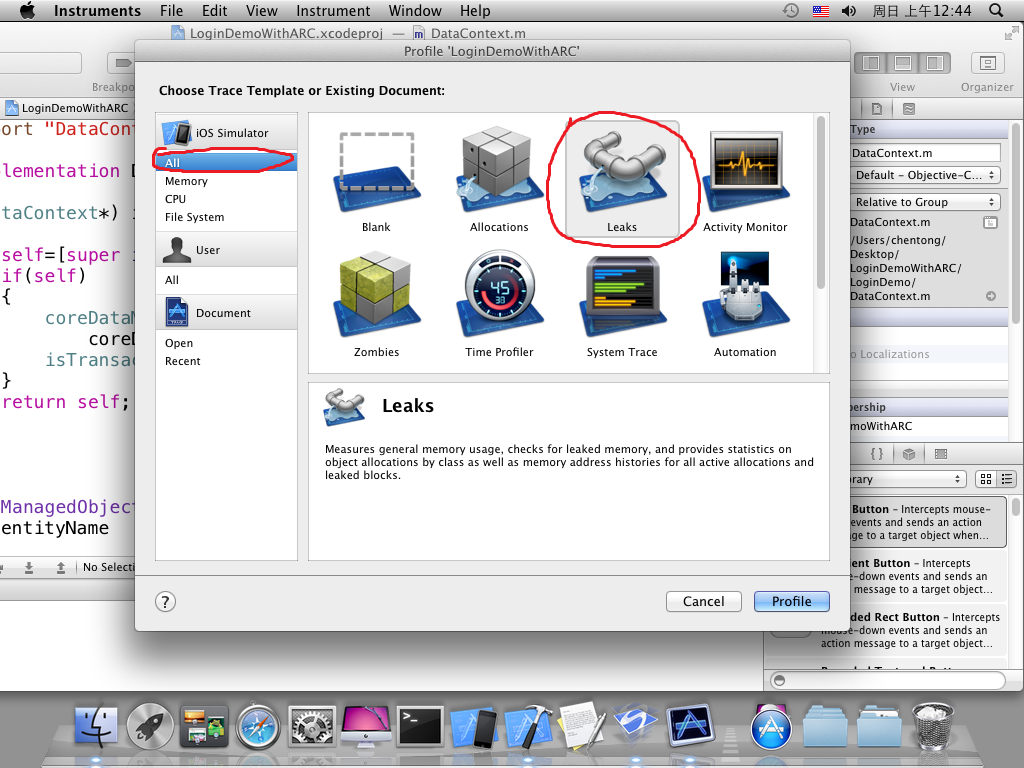
3)这样就开启了静态内存检测工具

## 2.内存泄漏动态检测工具(Instruments)

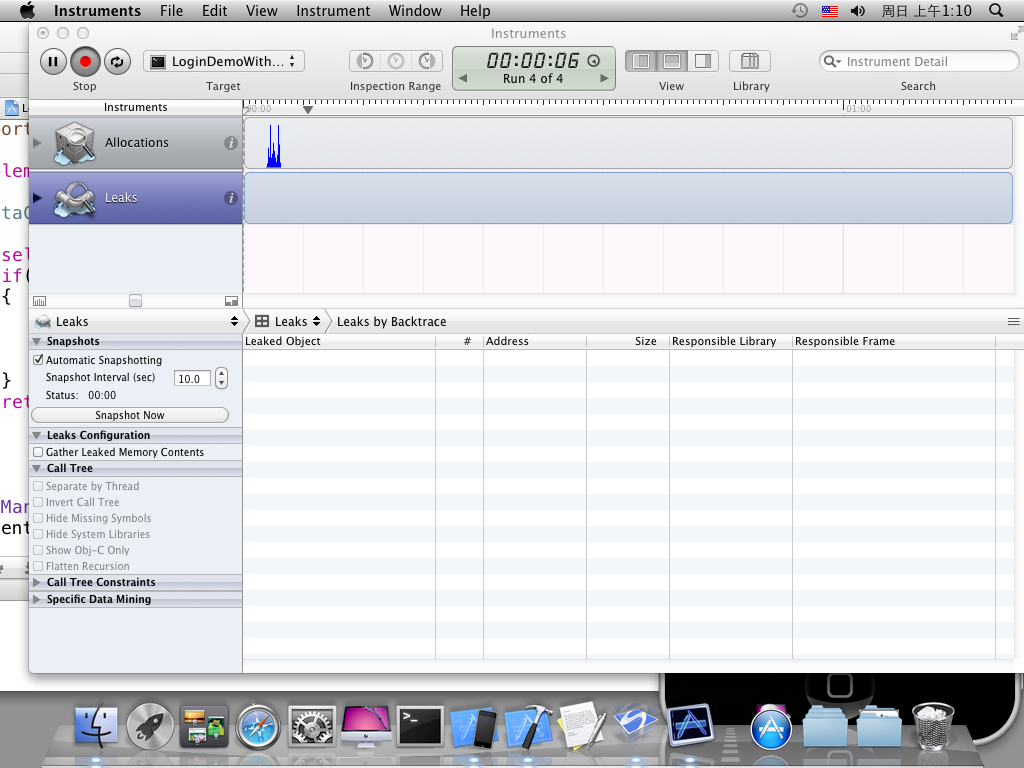
1) 　　进入Xcode4，选择菜单“Product”->“Profile”。



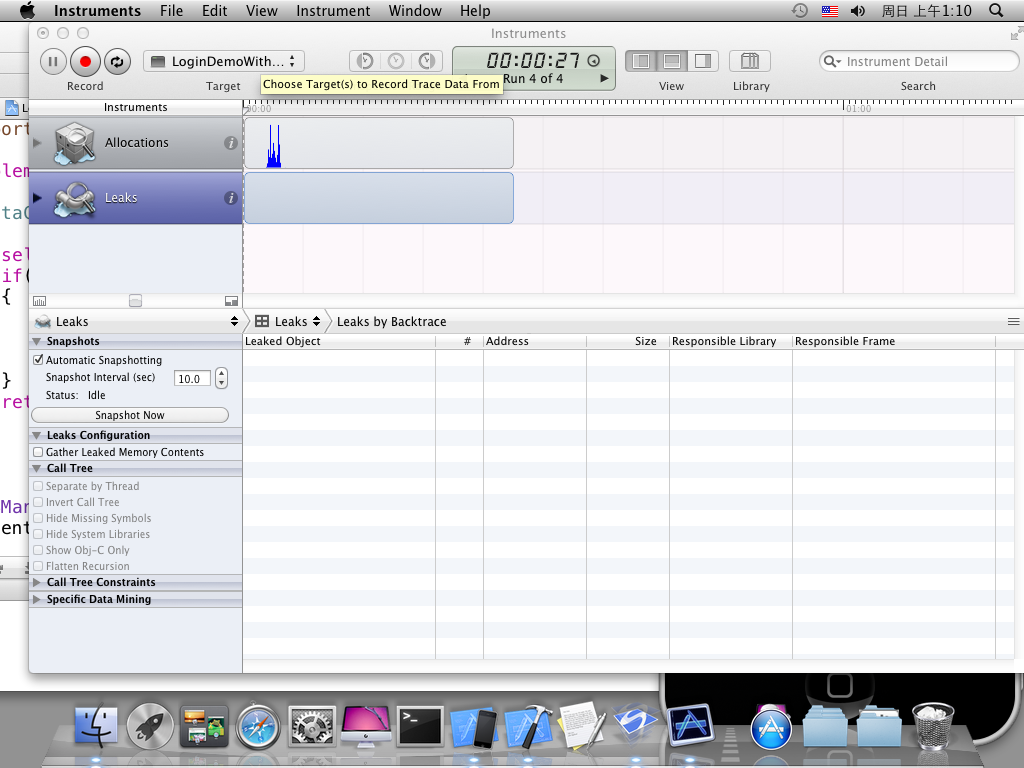
2) 在左侧选择”iOS Simulator” ，然后选中右侧的”Leaks”,点击”Profile”按钮。



3)开始自动监控运行程序的内存泄漏，同时会自动开启运行程序。



4)点击左上角的”stop”,停止监控，同时会自动关闭运行程序。



5)若出现内存泄漏，窗口右下角会提示出现内存泄漏的对象的详细信息

