Objective-C容易让人糊涂的知识点总结 #16

New issue

Open **abcrun** opened this issue 27 days ago · 0 comments

abcrun commented 27 days ago Owner

内存管理

目前Objective-C推荐使用 **ARC(自动引用计数)** 内存管理，ARC是编译程序时提供自动管理内存的功能；之前的Objective-C则采用的是 **自动释放池和手动处理引用计数** 进行内存管理，另外Objective-C还支持垃圾回收机制，只不过IOS系统不支持，垃圾回收不同与引用计数，垃圾回收是在程序运行后的回收机制。

自动释放池和手动处理引用计数

所谓自动释放池，就是说 **加入到** 自动释放池中的对象（*这时自动释放池获得了这些对象的所有权*），在自动释放池销毁时会自动调用一次 `release` 方法，从而使该对象的引用计数-1，而不需要手动执行对象的 `release` 方法，当这些对象的 **引用计数** 为0时，系统底层会调用 `dealloc` 方法释放掉该对象的内存。

如下代码说明了自动释放池的创建和销毁过程:

```
/**** 推荐方法 **/
@autoreleasepool{//创建自动释放池
    //statements
}//运行到这里，自动释放池销毁
```

Labels

学习笔记

未完成

Milestone

No milestone

Assignee

No one assigned

Notifications

Subscribe

You're not receiving notifications from this thread.

1 participant

```
/**** 之前的方法 **/
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];//创建自动释放池
//statements
[pool drain];//自动释放池销毁
```

自动释放池存在内存栈中，如果创建了多个自动释放池时，新的自动释放池会被放在栈顶，而池中的对象将会放在这个栈中。**但并不是说自动释放池里面的对象就加入了自动释放池**，如果以 `alloc`，`copy`，`mutableCopy`，`new` 方法为前缀创建的对象，自动释放池并不能获得对象的所有权，在自动释放池销毁时，无法对这些对象自动执行 `release` 操作。*如果对这些对象调用 `autorelease` 方法，自动释放池才会获得这些对象的所有权，此时这些对象才会被加入到自动释放池，执行相应的操作。* 但是对于这些对象，不建议使用 `autorelease` 将其加入释放池，直接通过手工方式修改引用计数的方式（如调用 `release` ）进行释放，这样便于区分和理解。

虽然有了自动释放池，但是当自动释放池销毁后，也只会对 **加入到池中的对象** 执行一次 `release` 操作，而这些对象的引用计数 `retainCount` 只会-1，并不一定能够清0，所以如果为了保证不需要的内存被释放，需要手动处理内存。

手动处理对象引用计数方法很简单,调用 `retain` 方法 +1， `release` 方法则 -1，来改变对象的引用计数值，当计数为0时，系统就会调用 `dealloc` 释放掉内存。

上面说的是单一对象的引用计数增加与消除，应用中常遇到的是不同对象之间的引用，甚至会出现循环引用，那么该如何处理？以类A和B为例。

对象引用

假设类A拥有一个属性是b(或者说setB方法)，先对A，B进行实例化后 `A *a = [[A alloc] init];B *b1 = [[B alloc] init]` ，a和b的retainCount都为1，假设A中的setB方法是这么写的：

```
-(void) setB:(B *) _b {
    b = _b;//直接赋值，不会改变引用计数
}
```

对a执行 `[a setB:b1]` 或者 `a.b = b1` ，这是实例对象a就拥有指向b1的属性，**如果这时对b1执行** `[b1`

`release]`**,那么程序出错**，因为此时b1的retainCount是0被释放掉，而a的属性b却指向b1。要避免这个错误，就得让b1不被释放掉，那么setB可以这么写：

```
-(void) setB:(B *) _b {
    b = [_b retain];//retain，引用计数+1
}
```

这样当对b1执行 `[b1 release]` 后，b1的retainCount为1，不会报错，但是此时如果对a执行 `[a release]`，还需要销毁b1,所以就得需要改造b1的 `dealloc` 方法了。

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

但是还会遇到一种情况，倘若还有一个实例化变量b2,a的属性先指向b1,然后换成b2,这是为了防止内存泄露，必须的保证a的属性指向b2之前，将b1的内存给释放掉，因此还需要对setB进行完善:

```
-(void) setB:(B *) _b {
    [b release]
    b = [_b retain];//retain，引用计数+1
}
```

通过上面说明可以发现手动处理内存需要非常仔细，为了方便Objective-C内存管理，OC提供了属性修饰选项，用于在处理在 **设置属性值时** 自动管理内存。手动自动释放池模式中， `@property` 可以声明 `assign,retain,copy` 选项， `assign` 是默认,当对属性设置 `retain` 修饰选项时，类似于执行了上面代码最后一步操作，**但是并没有重写类的** `dealloc` **方法，我们还是需要手动添加上的。**另外我们还需要在主程序中对对象（a,b）进行释放（手动，或者利用自动释放池）。

下面是这三个属性修饰选项的解释：

- `assign` 默认选项：直接赋值，不改变引用计数值，适用于基本的数据类型，如init,float,char等

- retain： 释放之前对象的引用，并指向新的引用对象，新的引用对象的引用计数加1
- copy： 和 retain 相似,但是没有增加新的引用对象的计数，而是重新分配了新的内存，适用于**复制**

以 retain,copy 修饰属性选项为例，给属性设置 obj.property = newValue ，实际上相当于对属性设置时，内部执行了以下代码:

```
if(property != newValue){//retain
    [property release];
    property = [newValue retain];
}
if(property != newValue){//copy
    [property release];
    property = [newValue copy];
}
```

但是一定要记住还需要重写 dealloc 方法。

循环引用

引用计数还有个典型的问题就是循环引用，这是可以给两个引用对象的属性 @property ， 一个设置为 retain ， 另外一个设置成 assign 。当然你只用给 retain 的类重写 dealloc 方法.

ARC-自动引用计数

采用ARC编译时 自动释放池中的所有对象会被系统自动维护处理，不在引用的将会被销毁，我们不需要手动的处理 retain , release , autorelease (实际上ARC模式也不支持使用搞这些方法)，这样方便开发，但是需要注意的是ARC模式是在IOS5才开始支持的。

ARC中也有类似于手动模式的属性修饰符： unsafe_unretained,weak,strong 这些选项中, unsafe_unretained 和 weak 相当于上面的 assign , strong 相当于 retain ,而 weak 与 assign 的区别就是当对象消失时， weak 将对象指针设为 nil , weak 用于解决ARC中循环引用的问题，但是另一个对象的属性要设置成 strong 。

属性声明

接口文件中 `@property` 属性默认的取值和设置方法是getName,setName(name) - 假设属性名是name；也可以结合实现文件 `@sythesize` 用于声明属性并自动生成设值和取值方法(`@synthesize` *非必须，但是如果没有声明则生成的实体变量会以_为前缀*）。另外也可以通过属性修饰符 `setter` , `getter` 来指定取值和设置函数。

属性 `@property` 的修饰选项，可以包含以下选项以name为例：

```
@property(assign默认/retain/copy,getter=name,setter=setName,atonic默认/nonatomic,readwrite默
```

`atonic,nonatomic` 是用于线程保护的选项，前者表示互斥锁进行线程保护，后者反之。实际上对对属性使用了 `atomic` ，相当于在setter方法中执行了：

```
-(void) setterName:(id) name {
    @synchronized(self){
        _name = name;
    }
}
```

玩转控件

借助Xcode6的storyboard，我们可以在控件库中拖出需要的控件并在属性面板中设置属性，也可以不通过拖拽。不过不管是那种形式实例的控件，都会在加载完ui文件后(storyboad或者.nib文件),调用里面所有对象的 `-(void)awakeFromNib` 方法，所以如果我们想操作或者修改这些对象，可以为这些对象定义 `awakeFromNib` 方法，但是不能在这个方法中定义控件的边界(frame或bound)，边界可以在 `layoutSubviews` 中设置。关于添加控件有以下几种情况：

代码中添加控件

而在ViewController类的 `viewDidLoad` 中直接添加控件，以添加**UILabel**为例：

```
- (void)viewDidLoad {
    [super viewDidLoad];
```

```
    UILabel *label = [[UILabel alloc] initWithFrame:self.view.bounds];
    label.text = @"吃葡萄不吐葡萄皮";
    [self.view addSubview:label];
}
```

代码修改storyboard控件属性

对于拖拽到storyboard面板上的控件，我们可能希望通过代码来更好的修改属性(毕竟属性面板并不能包含空间所有的属性),这时我们可以自定义一个继承当前控件或者UIView的类，然后将拖拽的这个控件绑定到新建的类上，在新建的类里面进行修改。

- 可以在 `-(void)awakeFromNib` 中设置控件的属性值，但是不能设置控件的frame和bounds边界；
- 如果想重新设置控件的边界，需要在 `-(void)layoutSubviews` 里设置，这里面也可以修改控件的属性。

关于这两个方法的区别： `awakeFromNib` 是在初始化就会调用的，无法设置控件的边界，
而 `layoutSubViews` 则是在 `viewDidLoad` 之后调用的，主要用于在控件边界变换时重新布局。

自定义绘制控件

另外如果控件库不存在我们需要的控件，这时我们就需要自定义控件了。自定义控件时，可以先从控件库面板拖拽出一个UIView控件，同上新建一个UIView子类，在此空间上添加已知的控件，或者通过重写 `-(void)drawRect:(CGRect)aRect` 绘制控件。需要说明的是 `drawRect` 也是发生在viewController生命周期 `viewDidLoad` 之后才会触发。

当自定义的view某个属性发生改变时，如果view需要重绘，则需要在这个属性的设置函数中调用 `-(void)setNeedsDisplay` 或 `-(void)setNeedsDisplayInRect:(CGRect)aRect` 方法，这时当这个属性值发生改变时，系统会自动调用 `-(void) drawRect:(CGRect)aRect` ，进行重绘，当然也可以通过设置view的 `contentMode` 为 `UIViewContentModeRedraw` ，这时表示全局的，只要修改机会重绘。

ViewController生命周期

storyboard - outlet setting

在storyboard进行初始化后(拖拽空间，设置输出口)，会调用 `-(void)viewDidLoad` 方法，这里可以放置视图控制器初始化代码，需要注意的是不能在这里添加任何关于视图形状相关的初始化代码，因为此时视图边界尚未确定。

Appearing and disappearing

在屏幕显示之前将会调用 `-(void)viewWillAppear:(BOOL) animated` ，由于app可能包含多个视图，所以同一个视图可能会多次的出现或者隐藏，所以当视图隐藏前会调 `-(void)viewWillDisappear(BOOL)animated` ，由于 `viewWillAppear` 可能会被多次加载，在这期间Model数据可能会改变，所以可以在这里设置加载Data，当然也可以处理几何形状的改变等。

另外还有两个和这两个相似的函数 `viewDidAppear` , `viewDidDisappear` 是在视图出现或者消失之后调用的。

Geometry changes

当我们横屏或者竖屏时会触发集合形状的改变，尽管系统会自动的对元素进行布局，但是自动布局并不一定是我们想要的结果，这时可以在 `-(void)view{Will,Did}LayoutSubviews` 这两个函数中设置。

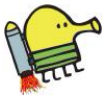
Low-memory situations

`didRecieveMemoryWarning` 用于处理当内存不足时的情况。

对象复制

一般对于Foundation对象，可以直接调用 `copy` 或 `mutableCopy` 进行复制，而自定义对象需要在定义类时，实现 `<NSCopying>` 协议，在实现中定义 `copyWithZone` 方法，在执行程序中通过调用 `copy` 方法，来实现简单的复制。但是对于复杂的涉及到可修改的对象时，需要考虑是 **深度复制** 还是 **浅复制**，这种情况下如果要实现深度复制，可以通过归档的方式来处理，同理自定义归档的类，需要实现 `<NSCoding>` 协议实现 `encodeWithCoder,initWithCoder` 方法，将数据编码归档，解码传递给 `NSData` 对象，并最终复制给新的引用，从而实现深度复制。

  **abcrun** added 未完成 学习笔记 labels 27 days ago



Write Preview Markdown supported Edit in fullscreen

Leave a comment

Attach images by dragging & dropping, selecting them, or pasting from the clipboard.

Comment

