**复习重点**

1. **内存管理、NSString类创建的几种方式类存情况**

OC没有像JAVA一样的垃圾回收机制，也就是说，OC编程需要程序员手动去管理内存。这就是为什么它烦的原因，苹果却一直推崇开发者在有限硬件资源内写出最优化的代码，使用CPU最少，占用内存最小。

基本原理

对象的创建：

OC在创建对象时，不会直接返回该对象，而是返回一个指向对象的指针，因此出来基本类型以外，我们在OC中基本上都在使用指针。

ClassA \*a = [[ClassA alloc] init];

在[ClassA alloc]的时候，已经发送消息通知系统给ClassA的对象分配内存空间，并且返回了指向未初始化的对象的一个指针。

未初始化的ClassA对象接手到init消息，init返回指向已初始化后的ClassA对象的一个指针，然后将其赋值给变量a。

在创建并使用完一个对象的时候，用户需要手动地去释放该对象。

[a dealloc];

如果指针a和b同时指向堆中同一块内存地址

ClassA \*a = [[ClassA alloc] init];

ClassA \*b = a;

[a dealloc];

当执行到第三行的时候，指针b就成了无头指针。这是一个在C++中也是常见的错误，我们需要避免这类错误，因为无头指针是危险的。

引用计数：

OC在内存管理上采用了引用计数（retain count），在对象内部保存一个数字，用来表示被引用的次数。init、new和copy都会让retain count加1。当销毁对象的时候，系统不会直接调用dealloc方法，而是先调用release，让retain count 减1，当retain count等于0的时候，系统才会调用dealloc方法来销毁对象。

在指针赋值的时候，retain count 是不会自动增加的，为了避免上面所说的错误，我们需要在赋值的时候手动retain一次，让retain count 增加1。

ClassA \*a = [[ClassA alloc] init]; // retain count = 1

ClassA \*b = a;

[b retain]; // retain count = 2

[a dealloc];

这样在执行到第四行的时候，对象的retain count只是减了1，并没有被销毁，指针b仍然有效。

内存泄露：

就如上面列子所示，当生成ClassA对象时，指针a拥有对该对象的访问权。如果失去了对一个对象的访问权，而又没有将retain count减至0，就会造成内存泄露。也就是说，分配出去的内存无法回收。

ClassA \*a = [[ClassA alloc] init];

a = nil;

Autorelease Pool

为了方便程序员管理内存，苹果在OC中引入了自动释放池(Autorelease Pool)。在遵守一些规则的情况下，可以自动释放对象。但即使有这么一个工具，OC的内存仍需要程序员时刻关注（这个自动释放池跟JAVA的垃圾回收机 制不是一回事，或者说，骑马都追不上JAVA的机制，可能连尘都吃不到）。

ClassA \*a = [[[ClassA alloc] init] autorelease];

//retain count = 1，但无需release

Autorelease Pool 的原理：

autorelease pool 全名叫做NSAutoreleasePool，是OC中的一个类。autorelease pool并不是天生就有的，你需要手动的去创建它

NSAutoreleasePool \*pool = [[NSAutoreleasePool alloc] init];

一般地，在新建一个iphone 项目的时候，xcode会自动地为你创建一个autorelease pool，这个pool就写在Main函数里面。

在NSAutoreleasePool中包含了一个 可变数组，用来存储被声明为autorelease的对象。当NSAutoreleasePool自身被销毁的时候，它会遍历这个数组，release数 组中的每一个成员（注意，这里只是release，并没有直接销毁对象）。若成员的retain count 大于1，那么对象没有被销毁，造成内存泄露。

默认的NSAutoreleasePool 只有一个，你可以在你的程序中创建NSAutoreleasePool，被标记为autorelease的对象会跟最近的 NSAutoreleasePool 匹配。

NSAutoreleasePool \*pool = [[NSAutoreleasePool alloc] init];

//Create some objects

//do something…

[pool  release];

你也可以嵌套使用NSAutoreleasePool ，就像你嵌套使用for一样。

即使NSAutoreleasePool 看起来没有手动release那么繁琐，但是使用NSAutoreleasePool 来管理内存的方法还是不推荐的。因为在一个NSAutoreleasePool 里面，如果有大量对象被标记为autorelease，在程序运行的时候，内存会剧增，直到NSAutoreleasePool 被销毁的时候才会释放。如果其中的对象足够的多，在运行过程中你可能会收到系统的低内存警告，或者直接crash。

Autorelease Pool 扩展：

如果你极具好奇心，把Main函数中的NSAutoreleasePool  代码删除掉，然后再自己的代码中把对象声明为autorelease，你会 发现系统并不会给你发出错误信息或者警告。用内存检测工具去检测内存的话，你可能会惊奇的发现你的对象仍然被销毁了。

其实在新生成一个Run Loop的时候，系统会自动的创建一个NSAutoreleasePool ，这个NSAutoreleasePool 无法被删除。

在做内存测试的时候，请不要用NSString。OC对字符串作了特殊处理

NSString \*str =[ [NSString alloc] stringWithString:@”123”];

在输出str的retain count 的时候，你会发现retain count 大于1。

四、手动管理内存

使用alloc、new、copy创建一个对象，该对象的retain count 都等于1，需要用release来释放该对象。谁创建，谁去释放。在这3钟方法以外的方法创建的对象，都被系统默认的声明为autorelease。

ClassA \*a = [[ClassA alloc] init];

ClassA \*b = a;

[b retain];//do smoething

[b release];

b = nil;

把一个指针赋值给另外一个指针的时候，a 指针所指向的对象的引用次数并没有增加，也就是说，对象的retain count依然等于1。只有在retain了之后，retain count 才会加1。那么，如果这时候执行[a  release]，只是a指针放弃了对对象的访问权，对象的retain count 减1，对象没有被销毁。只有当b也执行了release方法之后，才会将对象销毁掉。因此，谁retain了，谁就要release。

在对象被销毁之后，指针依然是存在的。所以在release了之后，最好把指针赋为空，防止无头指针的出现。顺便一说，release一个空指针是合法的，但是不会发生任何事情。

如果你在一个函数中创建并返回一个对象，那么你需要把这个对象声明为autorelease

(ClassA  \*)Function()

{

ClassA \*a = [[[ClassA alloc] init] autorelease];

return a;

}

不这样做的话，会造成内存泄露。

五、属性与内存管理

苹果一直没有强调的一点是，关于属性中的retain。事实上，属性中带有retain的，在赋值的时候可能已经在合成的setter中retain了一次，因此，这里也需要release。

@property实际上是getter和setter，@synthesize是合成这2个方法。为什么在声明了属性之后可以用“.”来直接调用成员变 量呢？那是因为声明属性以后系统根据你给的属性合成了一个set方法和一个get方法。使用“.”与属性并没有直接关联，如果你不嫌麻烦，在你的程序里面 多写一个set和get方法，你也可以使用“.”来调用变量。

@property()，如果你里面什么都不写，那么系统会默认的把你的属性设置为：

@property(atomic, assign)…..

关于nonatomic：

这个属性没有对应的atomic关键字，即使我上面是这么写，但atomic只是在你没有声明这个特性的时候系统默认，你无法主动去声明这一特性。

如果你的程序只有一个主线程，或者你确定你的程序不会在2个或者以上线程运作的时候访问同一个变量，那么你可以声明为nonatomic。指定 nonatomic特性，编译器合成访问器的时候不会去考虑线程安全问题。如果你的多个线程在同一时间会访问到这个变量的话，可以将特性声明为 atomic(通过省略关键字nonatomic)。在这种特性的状态下，编辑器在合成访问器的时候就会在访问器里面加一个锁 （@synchronized），在同一时间只能有一个线程访问该变量。

但是使用锁是需要付出代价的，一个声明为atomic的属性，在设置和获取这个变量的时候都要比声明为nonatomic的慢。所以如果你不打算编写多线程代码，最好把变量的属性特性声明为nonatomic。

关于assign、retain和copy：

assign是系统默认的属性特性，它几乎适用于OC的所有变量类型。对于非对象类型的变量，assign是唯一可选的 特性。但是如果你在引用计数下给一个对象类型的变量声明为assign，那么你会在编译的时候收到一条来自编译器的警告。因为assign对于在引用计数 下的对象特性，只创建了一个弱引用（也就是平时说的浅复制）。这样使用变量会很危险。当你release了前一个对象的时候，被赋值的对象指针就成了无头 指针了。因此在为对象类型的变量声明属性的时候，尽量少（或者不要）使用assign。

关于assign合成的setter，看起来是这样的：

-(void)setObjA:(ClassA \*)a

{

objA = a;

}

在深入retain之前，先把声明为retain特性的setter写出来：

-(void)setObjA:(ClassA \*)a

{

If(objA != a)

{

[objA release];

objA = a;

[objA retain]; //对象的retain count 加1

}

}

明显的，在retain的setter中，变量retain了一次，那么，即使你在程序中

self.objA = a;

只写了这么一句，objA仍然需要release，才能保证对象的retain count 是正确的。但是如果你的代码

objA = a；

只写了这么一句，那么这里只是进行了一次浅复制，对象的retain count 并没有增加，因此这样写的话，你不需要在后面release objA。

这2句话的区别是，第一句使用了编译器生成的setter来设置objA的值，而第二句只是一个简单的指针赋值。

copy的setter看起来是这样的:

-(void)setObjA:(ClassA  \*)a

{

ClassA \* temp = objA;

objA = [a copyWithZone:nil];

[temp  release];

}

复制必须通过实现copyWithZone：这个方法，因次copy这个特性只适用于拥有这个方法的类型，也就是说，必须这个类支持复制。复制是把 原来的对象release掉，然后让指针指向一个新的对象的副本。因此即使在setter里面release了原来的对象，你仍然需要在后面 release新指向的对象（副本）。

苹果在Mac OS X 雪豹(v10.5)系统里面添加了另外一种内存管理方式(垃圾收集)，但目前不适用于IOS。

1. **NSString类创建的几种方式类存情况？**

NSString是一个不可变的字符串对象。这不是表示这个对象声明的变量的值不可变，而是表示它初始化以后，你不能改变该变量所分配的内存中的值，但你可以重新分配该变量所处的内存空间。生成一个NSString类型的字符串有三种方法:

方法1.直接赋值:     NSString \*str1 = @"my string";

方法2.类函数初始化生成:     NSString \*str2 = [NSString stringWithString:@"my string"];

方法3.实例方法初始化生成:　  NSString \*str3 = [[NSString alloc] initWithString:@"my string"];

NSString \*str4 = [[NSString alloc]initWithFormat:@"my string"];

区别1: 方法一生成字符串时,不会初始化内存空间,所以使用结束后不会释放内存;

而其他三个都会初始化内存空间,使用结束后要释放内存;

在释放内存时方法2和3也不同,方法2是autorelease类型,内存由系统释放;方法3则必须手动释放

区别2:用Format初始化的字符串，需要初始化一段动态内存空间，如：**0x6a42a40;**

而用String声明的字符串，初始化的是常量内存区，如：**0x46a8**，常量内存区的地址，只要值相同，占用的地址空间是一致的。

所以str3和str1的地址一致，但是str4和str1的地址不一致。

对常量进行操作不会造成内存泄漏，而对对象进行alloc,copy,format.要release

1、initWithFormat是实例方法

只能通过 NSString\* str = [[NSString alloc] initWithFormat:@"[%@",@"Hello](mailto:%25@%22,@%22Hello) World"] 调用，但是必须手动release来释放内存资源

2、stringWithFormat是类方法

可以直接用 NSString\* str = [NSString stringWithFormat:@"[%@",@"Hello](mailto:%25@%22,@%22Hello) World"] 调用，内存管理上是autorelease的，不用手动显式release

1. **内存警告http://blog.163.com/l1\_jun/blog/static/14386388201352410233131/**

iPhone下每个app可用的内存是被限制的，如果一个app使用的内存超过20M，则系统会向该app发送Memory Warning消息。收到此消息后，app必须正确处理，否则可能出错或者出现内存泄露。

app收到Memory Warning后会调用：

UIApplication::didReceiveMemoryWarning -> UIApplicationDelegate::applicationDidReceiveMemoryWarning,然后调用当前所有的viewController进行处理。因此处理的主要工作是在viewController。

当我们的程序在第一次收到内存不足警告时，应该释放一些不用的资源，以节省部分内存。否则，当内存不足情形依然存在，iOS再次向我们程序发出内存不足的警告时，我们的程序将会被iOS kill掉。

iOS的UIViewController 类给我们提供了处理内存不足的接口。在iOS 3.0 之前，当系统的内存不足时，UIViewController的didReceiveMemoryWarining 方法会被调用，我们可以在didReceiveMemoryWarining 方法里释放掉部分暂时不用的资源。

从iOS3.0 开始，UIViewController增加了viewDidUnload方法。该方法和viewDIdLoad相配对。当系统内存不足时，首先UIViewController的didReceiveMemoryWarining 方法会被调用，而didReceiveMemoryWarining 会判断当前ViewController的view是否显示在window上，如果没有显示在window上，则didReceiveMemoryWarining 会自动将viewcontroller 的view以及其所有子view全部销毁，然后调用viewcontroller的viewdidunload方法。如果当前UIViewController的view显示在window上，则不销毁该viewcontroller的view，当然，viewDidunload也不会被调用了。但是到了ios6.0之后，这里又有所变化，ios6.0内存警告的viewDidUnload 被屏蔽，即又回到了ios3.0的时期的内存管理方式。

iOS3-iOS5.0以前版本收到内存警告：  
调用didReceiveMemoryWarning内调用super的didReceiveMemoryWarning会将controller的view进行释放。所以我们不能将controller的view再次释放。  
处理方法：

-(void)didReceiveMemoryWarning

{

[super didReceiveMemoryWarning];//如没有显示在window上，会自动将self.view释放。

// ios6.0以前，不用在此做处理，self.view释放之后，会调用下面的viewDidUnload函数，在viewDidUnload函数中做处理就可以了。

}

-(void)viewDidUnload

{

// Release any retained subviews of the main view.不包含self.view

//处理一些内存和资源问题。

[super viewDidUnload];

}

iOS6.0及以上版本的内存警告：  
调用didReceiveMemoryWarning内调用super的didReceiveMemoryWarning调只是释放controller的resouse，不会释放view  
处理方法：  
-(void)didReceiveMemoryWarning  
{  
[super didReceiveMemoryWarning];//即使没有显示在window上，也不会自动的将self.view释放。  
// Add code to clean up any of your own resources that are no longer necessary.  
// 此处做兼容处理需要加上ios6.0的宏开关，保证是在6.0下使用的,6.0以前屏蔽以下代码，否则会在下面使用self.view时自动加载viewDidUnLoad

if ([[UIDevice currentDevice].systemVersion floatValue] >= 6.0) {

//**需要注意的是self.isViewLoaded是必不可少的，其他方式访问视图会导致它加载 ，在WWDC视频也忽视这一点。**

if (self.isViewLoaded && !self.view.window)// 是否是正在使用的视图  
{  
// Add code to preserve data stored in the views that might be  
// needed later.  
// Add code to clean up other strong references to the view in  
// the view hierarchy.  
self.view = nil;// 目的是再次进入时能够重新加载调用viewDidLoad函数。  
}

}  
}

但是似乎这么写相对于以前并不省事。最终我们找到一篇文章，文章中说其实并不值得回收这部分的内存，原因如下：

1. UIView是UIResponder的子类，而UIResponder有一个CALayer的成员变量，CALayer是具体用于将自己画到屏幕上的。

2. CALayer是一个bitmap图象的包装类，当UIView调用自身的drawRect时，CALayer才会创建这个bitmap图象类。

3. 具体占内存的其实是一个bitmap图象类，CALayer只占48bytes, UIView只占96bytes。而一个iPad的全屏UIView的bitmap类会占到12M的大小！

4.在iOS6时，当系统发出MemoryWarning时，系统会自动回收bitmap类。但是不回收UIView和CALayer类。这样即回收了大部分内存，又能在需要bitmap类时，根据CALayer类重建。

所以，iOS6这么做的意思是：我们根本没有必要为了几十byte而费力回收内存。

1. **内存泄露，**

1.静态分析

通过静态分析我们可以最初步的了解到代码的一些不规范的地方或者是存在的内存泄漏，这是我们第一步对内存泄漏的检测。当然有一些警告并不是我们关心的可以略过。

2.通过instruments来检查内存泄漏

这个方法能粗略的定位我们在哪里发生了内存泄漏。方法是完成一个循环操作，如果内存增长为0就证明我们程序在该次循环操作中不存在内存泄漏，如果内存增长不为0那证明有可能存在内存泄漏，当然具体问题需要具体分析。

3.代码测试内存泄漏

在做这项工作之前我们要注意一下，在dealloc的方法中我们是否已经释放了该对象所拥有的所有对象。观察对象的生成和销毁是否配对。准确的说就是 init（创建对象的方法）和dealloc是否会被成对触发（简单说来就是走一次创建对象就有走一次dealloc该对象）。

1. **内存检查工具**

编译和分析工具Analyze

内存泄漏检测工具—Leak

内存猛增检测工具—Allocations

1. **性能优化（离线缓存）**

**入门：**

**1. 用ARC管理内存**

**2. 在正确的地方使用reuseIdentifier **

**3. 尽可能使Views透明 **

**4. 避免庞大的XIB **

**5. 不要block主线程**

**6. 在Image Views中调整图片大小 **

**7. 选择正确的Collection **

**8. 打开gzip压缩**

**中级**

**9. 重用和延迟加载Views **

**10. Cache, Cache, 还是Cache！ **

**11. 权衡渲染方法 **

**12. 处理内存警告 **

**13. 重用大开销的对象 **

**14. 使用Sprite Sheets **

**15. 避免反复处理数据 **

**16. 选择正确的数据格式**

**17. 正确地设定Background Images **

**18. 减少使用Web特性 **

**19. 设定Shadow Path **

**20. 优化你的Table View **

**21. 选择正确的数据存储选项**

**进阶：**

**22. 加速启动时间 **

**23. 使用Autorelease Pool **

**24. 选择是否缓存图片 **

**25. 尽量避免日期格式转换**

1. **基类、父类、分类、继承、**

分类，用来扩展类的方法，不能定义新成员，但是可以访问到私有成员  
子类，可以通过覆盖和定义新方法来扩展父类，可以新增成员，但是不能访问父类的私有成员。

1. **Block、代理、通知**
2. **设计模式（单例、适配器、工厂、组合、责任链、观察者、代理） MVC**
3. **KVC/KV0**
4. **视图控制器的生命周期**
5. **运行时机制runtime**
6. **响应链**
7. **屏幕适配**
8. **数据存储:NSCoder、SQLite、Core Data**
9. **核心动画**
10. **多线程 GCD 同步异步 加锁**
11. **网络交互**
12. **JSON、XML解析**
13. **第三方框架**
14. **消息推送**
15. **发布流程**
16. **蓝牙**
17. **音频、音效**
18. **支付、加密**
19. **二维码**
20. **Socket**
21. **正则表达**
22. **Svn、Git**
23. **了解HTTP协议了解各种请求方式（post、get、head、put、delete）如果对这个不了解 那么像上传/ 下载这些功能你都没法去做，还有像断点续传功能也不能做，所以你们有空的时候可以看一下这块。**
24. **TCP/IP**
25. **XMPP、群聊功能**