orilme
2019年04月14日 阅读 314

关注

iOS 内存管理相关面试题

内存管理的一些概念

- 为什么要使用内存管理?

    1. 严格的内存管理，能够是我们的应用程在性能上有很大的提高
    2. 如果忽略内存管理，可能导致应用占用内存过高，导致程序崩溃

- OC的内存管理主要有三种方式:

    1. ARC（自动内存计数）
    2. 手动内存计数
    3. 内存池

- OC中内存管理的基本思想:
  保证任何时候指向对象的指针个数和对象的引用计数相同，多一个指针指向这个对象这个对象的引用计数就加1，少一个指针指向这个对象这个对象的引用计数就减1。没有指针指向这个对象对象就被释放了。

    1. 每个对象都有一个引用计数器，每个新对象的计数器是1，当对象的计数器减为0时，就会被销毁
    2. 通过retain可以让对象的计数器+1、release可以让对象的计数器-1
    3. 还可以通过autorelease pool管理内存
    4. 如果用ARC，编译器会自动生成管理内存的代码

- 苹果官方基础内存管理规则:

    1. 你拥有你创建的任何对象
    2. 你可以使用retain获取一个对象的拥有权

3. 当你不再需要它，你必须放弃你拥有的对象的拥有权
4. 你一定不能释放不是你拥有的对象的拥有权

自动内存管理

- 谈谈你对 `ARC` 的认识和理解？ `ARC` 是iOS 5推出的新功能。编译器在代码里适当的地方自动插入 `retain` / `release` 完成内存管理（引用计数）。

- ARC机制中,系统判断对象是否被销毁的依据是什么?
  指向对象的强指针是否被销毁

引用计数器

1. 给对象发送一条retain消息，可以使引用计数器+1(retain方法返回对象本身)
2. 给对象发送一条release消息，可以使引用计数器-1(注意release并不代表销毁/回收对象，仅仅是计数器-1)
3. 给对象发送retainCount消息，可以获得当前的引用计数值

自动释放池

- 自动释放池底层怎么实现?
  (以栈的方式实现的)(系统自动创建，系统自动释放)栈里面的(先进后出)
  内存里面有栈，栈里面有自动释放池。
  自动释放池以栈的形式实现：当你创建一个新的自动释放池时，它将被添加到栈顶。当一个对象收到发送autorelease消息时,它被添加到当前线程的处于栈顶的自动释放池中，当自动释放池被回收时，它们从栈中被删除，并且会给池子里面所有的对象都会做一次release操作。

- 什么是自动释放池?
  答：自动释放池是用来存储多个对象类型的指针变量

- 自动释放池对池内对象的作用? 被存入到自动释放池内的对象，当自动释放池被销毁时，会对池内的对象全部做一次release操作

- 对象如何放入到自动释放池中? 当你确定要将对象放入到池中的时候，只需要调用对象的 `autorelease` 对象方法就可以把对象放入到自动释放池中

- 多次调用对象的autorelease方法会导致什么问题?

答：多次将地址存到自动释放池中,导致野指针异常

- 自动释放池作用
  将对象与自动释放池建立关系，池子内调用 `autorelease` 方法，在自动释放池销毁时销毁对象，延迟 `release` 销毁时间

- 自动释放池，什么时候创建?

  1. 程序刚启动的时候，也会创建一个自动释放池
  2. 产生事件以后，运行循环开始处理事件，就会创建自动释放池

- 什么时候销毁的?

  1. 程序运行结束之前销毁
  2. 事件处理结束以后，会销毁自动释放池
  3. 还有在池子满的时候，也会销毁

- 自动释放池使用注意：
  不要把大量循环操作放在释放池下，因为这会导致大量循环内的对象没有被回收，这种情况下应该手动写 `release` 代码。尽量避免对大内存对象使用 `autorelease` ，否则会延迟大内存的回收。

- autorelease的对象是在什么时候被release的?
  答：autorelease实际上只是把对release的调用延迟了，对于每一个Autorelease，系统只是把该Object放入了当前的 Autoreleasepool中，当该pool被释放时，该pool中的所有Object会被调用Release。对于每一个Runloop，系统会隐式创建一个Autoreleasepool，这样所有的releasepool会构成一个象CallStack一样的一个栈式结构，在每一个 Runloop结束时，当前栈顶的Autoreleasepool会被销毁，这样这个pool里的每个Object（就是autorelease的对象）会被release。那什么是一个Runloop呢？一个UI事件，Timer call，delegate call， 都会是一个新的Runloop。

- If we don’t create any autorelease pool in our application then is there any autorelease pool already provided to us?
  系统会默认会不定时地创建和销毁自动释放池

- When you will create an autorelease pool in your application?
  当不需要精确地控制对象的释放时间时，可以手动创建自动释放池

@property内存管理策略的选择

读写属性：readwrite 、readonly setter语意：assign 、retain/copy 原子性（多线程管理）：atomic 、 nonatomic 强弱引用：strong 、 weak

- 读写属性：
  `readwrite` ： 同时生成 `set` 和 `get` 方法(默认)
  `readonly` ： 只会生成 `get` 方法

- 控制set方法的内存管理：
  `retain`： `release` 旧值， `retain` 新值。希望获得源对象的所有权时，对其他 `NSObject` 和其子类(用于 `OC` 对象)
  `copy` ： `release` 旧值， `copy` 新值。希望获得源对象的副本而不改变源对象内容时(一般用于 `NSString` ， `block`)
  `assign` ： 直接赋值，不做任何内存管理(默认属性)，控制需不需生成 `set` 方法。对基础数据类型 （ `NSInteger` ， `CGFloat` ）和C数据类型（ `int` , `float` , `double` , `char` , 等等)

- 原子性（多线程管理）：
  - atomic
    默认属性，访问方法都为原子型事务访问。锁被加到所属对象实例级，性能低。原子性就是说一个操作不可以中途被 cpu 暂停然后调度, 即不能被中断, 要不就执行完, 要不就不执行. 如果一个操作是原子性的,那么在多线程环境下, 就不会出现变量被修改等奇怪的问题。原子操作就是不可再分的操作，在多线程程序中原子操作是一个非常重要的概念，它常常用来实现一些同步机制，同时也是一些常见的多线程 Bug 的源头。当然，原子性的变量在执行效率上要低些。
  - nonatomic
    非原子性访问。不加同步，尽量避免多线程抢夺同一块资源。是直接从内存中取数值，因为它是从内存中取得数据，它并没有一个加锁的保护来用于cpu中的寄存器计算Value，它只是单纯的从内存地址中，当前的内存存储的数据结果来进行使用。 多线程并发访问会提高性能，但无法保证数据同步。尽量避免多线程抢夺同一块资源，否则尽量将加锁资源抢夺的业务逻辑交给服务器处理，减少移动客户端的压力。
    当有多个线程需要访问到同一个数据时，OC中，我们可以使用 `@synchronized` （变量）来对该变量进行加锁（加锁的目的常常是为了同步或保证原子操作）。

- 强指针(strong)、弱指针(weak)

- `strong`

  `strong` 系统一般不会自动释放，在 `oc` 中，对象默认为强指针。作用域销毁时销毁引用。在实际开放中一般属性对象一般 `strong` 来修饰（`NSArray`，`NSDictionary`），在使用懒加载定义控件的时候，一般也用strong。
- `weak`

  `weak` 所引用对象的计数器不会加一，当对象被释放时指针会被自动赋值为 `nil`，系统会立刻释放对象。
- `__unsafe_unretained` 弱引用 当对象被释放时指针不会被自动赋值为 `ni`

  在ARC时属性的修饰符是可以用 `assign` 的（相当于 `__unsafe_unretained`）

  在ARC时属性的修饰符是可以用 `retain` 的(相当于 `__strong`)
- 假定有N个指针指向同一个对象，如果至少有一个是强引用，这个对象只要还在作用域内就不会被释放。相反，如果这N个指针都是弱引用，这个对象马上就被释放
- 在使用 `sb` 或者 `xib` 给控件拖线的时候，为什么拖出来的先属性都是用 weak 修饰呢?

  由于在向 `xib` 或者 `sb` 里面添加控件的时候,添加的子视图是添加到了跟视图 `View` 上面，而 控制器 `Controller` 对其根视图 `View` 默认是强引用的，当我们的子控件添加到 `view` 上面的时候，`self.view addSubView:` 这个方法会对添加的控件进行强引用，如果在用 `strong` 对添加的子控件进行修饰的话,相当于有两条强指针对子控件进行强引用, 为了避免这种情况,所以用 `weak` 修饰。

  注意：

  （1）addSubView 默认对其 subView 进行了强引用

  （2）在纯手码实现界面布局时，如果通过懒加载处理界面控件，需要使用strong强指针

- ARC管理内存是用 `assign` 还是用 `weak` ？

  `assign` : 如果由于某些原因代理对象被释放了，代理指针就变成了野指针。

  `weak` : 如果由于某些原因代理对象被释放了，代理指针就变成了空指针，更安全（`weak` 不能修饰基本数据类型，只能修饰对象）。

内存分析

- 静态分析(Analyze)
  1. 不运行程序, 直接检测代码中是否有潜在的内存问题(不一定百分百准确, 仅仅是提供建议)
  2. 结合实际情况来分析, 是否真的有内存问题
- 动态分析(Profile == Instruments)
  1. 运行程序, 通过使用app，查看内存的分配情况(Allocations)：可以查看做出了某个操作后(比如点击了某个按钮\显示了某个控制器)，内存是否有暴增的情况(突然变化)
  2. 运行程序, 通过使用app, 查看是否有内存泄漏(Leaks)：红色区域代表内存泄漏出现的地

方

什么情况下会发生内存泄漏和内存溢出?

内存泄漏：堆里不再使用的对象没有被销毁，依然占据着内存。
内存溢出：一次内存泄露危害可以忽略，但内存泄露多了，内存迟早会被占光，最终会导致内存溢出！当程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如数据长度比较小的数据类型 存储了数据长度比较大的数据。

关于图片占用内存管理

- 图片加载占用内存对比

    1. 使用 `imageName:` 加载图片：
        - 加载到内存当中后，占据内存空间较大
        - 相同的图片，图片不会重复加载
        - 加载内存当中之后，会一直停留在内存当中，不会随着对象销毁而销毁
        - 加载进去图片之后，占用的内存归系统管理，我们无法管理
    2. 使用 `imageWithContentsOfFile:` 加载图片
        - 加载到内存当中后,占据内存空间较小
        - 相同的图片会被重复加载内存当中
        - 对象销毁的时候,加载到内存中图片会随着一起销毁
    3. 结论:
        1. 图片较小，并且使用频繁，使用 `imageName:` 来加载(按钮图标/主页里面图片)
        2. 图片较大，并且使用较少，使用 `imageWithContentsOfFile:` 来加载(版本新特性/相册)

- 图片在沙盒中的存在形式

    1. 部署版本在>=iOS8的时候，打包的资源包中的图片会被放到Assets.car。图片有被压缩；
       部署版本在<iOS8的时候，打包的资源包中的图片会被放在MainBudnle里面。图片没有被压缩
    2. 没有放在Images.xcassets里面的所有图片会直接暴露在沙盒的资源包(main Bundle), 不会压缩到Assets.car文件，会被放到MainBudnle里面。图片没有被压缩
    3. 结论：
        - 小图片\使用频率比较高的图片放在Images.xcassets里面

    - 大图片\使用频率比较低的图片(一次性的图片, 比如版本新特性的图片)不要放在Images.xcassets里面

内存管理问题

单个对象内存管理的问题

- 关于内存我们主要研究的问题是什么? 野指针：对象的retainCount已经为0,保存了对象指针地址的变量就是野指针。使用野指针调用对象的方法,会导致野指针异常,导致程序直接崩溃
  内存泄露：已经不在使用的对象,没有正确的释放掉,一直驻留在内存中,我们就说是内存泄漏
- 僵尸对象? retainCount = 0的对象被称之为僵尸对象,也就是不能够在访问的对象
  1. 是什么问题导致,访问僵尸对象,时而正确时而错误?
  2. 如何开始xcode的时时检测僵尸对象功能?
- 当对象的retainCount = 0 时 能否调用 retain方法使对象复活? 已经被释放的对象是无法在复活的
- 如何防止出现野指针操作? 通常在调用完release方法后，会把保存了对象指针地址的变量清空，赋值为nil 在oc中没有空指针异常,所以使用[nil retain]调用方法不会导致异常的发生
- 内存泄漏有几种情况?
  1. 没有配对释放，不符合内存管理原则
  2. 对象提前赋值为nil或者清空，导致release方法没有起作用

多个对象内存管理的问题

- 对象与对象之间存在几种关系?
  1. 继承关系
  2. 组合关系
  3. 对象作为方法参数传递
- 对象的组合关系中,如何确保作为成员变量的对象,不会被提前释放? 重写set方法，在set方法中，retain该对像，使其retainCount值增加 1
- 组合关系导致内存泄漏的原因是什么? 在set方法中，retain了该对象，但是并没有配对释放
- 作为成员变量的对象,应该在那里配对释放? 在dealloc函数中释放

内存相关的一些数据结构的对比

- 简述内存分区情况

  1. 代码区：存放函数二进制代码
  2. 数据区：系统运行时申请内存并初始化，系统退出时由系统释放。存放全局变量、静态

变量、常量

3. 堆区：通过malloc等函数或new等操作符动态申请得到，需程序员手动申请和释放
4. 栈区：函数模块内申请，函数结束时由系统自动释放。存放局部变量、函数参数

- 手机的存储空间分为内存（RAM）和闪存（Flash）两种

1. 内存一般较小：1G、2G、3G、4G。闪存空间相对较大16G、32G、64G；
2. 内存的读写速度较快、闪存的读写速度相对较慢；
3. 内存里的东西掉电后全部丢失、闪存里的东西掉电也不丢；
4. 内存相当于电脑的内存条、闪存相当于电脑的硬盘；

- 堆和栈的区别?

  - 管理方式：
    堆释放工作由程序员控制，容易产生memory leak；
    栈是由编译器自动管理，无需我们手工控制。
  - 申请大小：
    堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。
    栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 Windows下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。
  - 碎片问题：
    堆：频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。
    栈：则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出
  - 分配方式：
    堆都是动态分配的，没有静态分配的堆。
    栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloc函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。
  - 分配效率：
    栈：是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。

堆：则是C/C++函数库提供的，它的机制是很复杂的。

- 每个App有个内存空间，假定是4G，分为堆和栈两大部分。一般来说每个进程有一个堆（这个进程的所有线程共用这个堆），进程中的线程有自己栈。
  通过alloc、new或malloc获得的内存在堆中分配，堆中的内存需要写相应的代码释放。
  如果进程结束了在堆中分配的内存会自动释放。
  局部变量、函数参数是在栈空间中分配，如果函数返回这个函数中的局部变量、参数所占的内存系统自动释放（回收）。
  程序在编译期对变量和函数分配内存都在栈上进行，且程序运行过程中函数调用时参数的传递也在栈上进行。

- 队列和栈有什么区别：
  队列和栈是两种不同的数据容器。从”数据结构”的角度看，它们都是线性结构，即数据元素之间的关系相同。
  队列是一种先进先出的数据结构，它在两端进行操作，一端进行入队列操作，一端进行出列队操作。
  栈是一种先进后出的数据结构，它只能在栈顶进行操作，入栈和出栈都在栈顶操作。

- 链表和数组的区别在哪里?
  二者都属于一种数据结构。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。

  - 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. 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

面试题

- 如何让程序尽量减少内存泄漏

  - 非ARC

`Foundation` 对象(`OC` 对象) : 只要方法中包含了 `alloc\new\copy\mutableCopy\retain` 等关键字，那么这些方法产生的对象, 就必须在不再使用的时候调用1次 `release` 或者1次 `autorelease` 。
`CoreFoundation` 对象(`C` 对象) : 只要函数中包含了 `create\new\copy\retain` 等关键字, 那么这些方法产生的对象, 就必须在不再使用的时候调用1次 `CFRelease` 或者其他 `release` 函数。

- ARC(只自动管理OC对象, 不会自动管理C语言对象)
  `CoreFoundation` 对象(`C` 对象) : 只要函数中包含了 `create\new\copy\retain` 等关键字, 那么这些方法产生的对象, 就必须在不再使用的时候调用1次 `CFRelease` 或者其他 `release` 函数。

- block的注意

```
// block的内存默认在栈里面(系统自动管理)
void (^test)() = ^{

};
// 如果对block进行了Copy操作, block的内存会迁移到堆里面(需要通过代码管理内存)
Block_copy(test);
// 在不需要使用block的时候, 应该做1次release操作
Block_release(test);
[test release];
```

- 野指针举例
  建了个视图控制器(ARC时)某个函数里写了如下代码。当这个函数返回时因为没有指针指向b所以b会被释放、但是b.view不会被释放。如果在b里有需要操作b的地方（比如代理的方法），就会产生野指针（提前释放）

```
B *b = [[B alloc]init];
[self.view addSubview:b.view];
```

- set方法

1. 在对象的组合关系中,导致内存泄漏有几种情况? 1.set方法中没有retain对象 2.没有release掉旧的对象 3.没有判断向set方法中传入的是否是同一个对象
2. 该如何正确的重写set方法? 1.先判断是否是同一个对象 2.release一次旧的对象 3.retain新的对象
   写一个setter方法用于完成@property （nonatomic,retain）NSString *name,

写一个setter方法用于完成@property（nonatomic，copy）NSString *name。

```
@property (nonatomic, retain) NSString *name;
- (void)setName:(NSString *)name {
    if (_name != name) {
        [_name release];
        _name = [name retain];
    }
}

@property(nonatomic, copy) NSString *name;
- (void)setName:(NSString *)name {
    if (_name != name) {
        [_name release];
        _name = [name copy];
    }
}

- (void)dealloc {
        self.name = nil;
        // 上边这句相当于下边两句
        [_name release];
        _name = nil;
}
```

- 引用计数的使用

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // 1
        Person *p = [[Person alloc] init];

        p.age = 20;

        // 0 (p指向的内存已经是坏内存, 称person对象为僵尸对象)
        // p称为野指针, 野指针: 指向僵尸对象(坏内存)的指针
        [p release];

        // p称为空指针
        p = nil;

        p.age = 40;
//        [0 setAge:40];
```

```
        // message sent to deallocated instance 0x100201950
        // 给空指针发消息不会报错
        [p release];
    }
    return 0;
}
```

堆和栈

```
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int a = 10; // 栈

        int b = 20; // 栈

        // c : 栈
        // Car对象(计数器==1) : 堆
        Car *c = [[Car alloc] init];
    }

    // 当autoreleasepool执行完后后, 栈里面的变量a\b\c都会被回收
    // 但是堆里面的Car对象还会留在内存中, 因为它是计数器依然是1

    return 0;
}
```

- 看下面的程序,三次NSLog会输出什么？为什么?
  结果：3、2、1

```
NSMutableArray* ary = [[NSMutableArray array] retain];
NSString *str = [NSString stringWithFormat:@"test"];  // 1
[str retain];   // 2
[ary addObject:str]; // 3
NSLog(@"%d", [str retainCount]);
[str retain];  // 4
[str release];   // 3
[str release];   // 2
NSLog(@"%d", [str retainCount]);
```

```
 [ary removeAllObjects]; // 1
 NSLog(@"%d", [str retainCount]);
```

- [NSArray arrayWithobject:]后需要对这个数组做释放操作吗?
  答： 不需要，这个对象被放到自动释放池中

- 老版本的工程是可以转换成使用ARC的工程,转换规则包括：

  1. 去掉所有的retain，release，autorelease
  2. 把NSAutoRelease替换成@autoreleasepool{}块
  3. 把assign的属性变为weak使用ARC的一些强制规定
  4. dealloc方法来管理一些资源，但不能用来释放实例变量，也不能在dealloc方法里面去掉[super dealloc]方法，在ARC下父类的dealloc同样由编译器来自动完成
  5. Core Foundation类型的对象任然可以用CFRetain，CFRelease这些方法
  6. 不能在使用NSAllocateObject和NSDeallocateObject对象
  7. 不能在c结构体中使用对象指针，如果有类似功能可以创建一个Objective-c类来管理这些对象
  8. 在id和void *之间没有简便的转换方法，同样在Objective-c和core Foundation类型之间的转换都需要使用编译器制定的转换函数
  9. 不能使用内存存储区（不能再使用NSZone）
  10. 不能以new为开头给一个属性命名
  11. 声明outlet时一般应当使用weak，除了对StoryBoard，这样nib中间的顶层对象要用strong
  12. weak 相当于老版本的assign，strong相当于retain

评论

输入评论...

相关推荐

专栏老司机iOS周报 · 1天前 · iOS / Swift

老司机 iOS 周报 #63 | 2019-04-15

15 2

专栏QiShare · 21小时前 · iOS

iOS 图标&启动图生成器（一）

20 11

专栏白衣哥 · 1天前 · iOS

JPTagView-多样化的标签View

4 1

专栏即将成为型男的涛 · 1天前 · iOS

iOS 多线程记录（一）

4

专栏orilme · 1天前 · iOS

iOS 内存管理相关面试题

9

专栏RyanLeeLY · 3天前 · iOS

让UINavigationController更好用

22 1

专栏orilme · 1天前 · iOS

iOS 启动速度优化和安装包优化简单总结

4

专栏我是繁星 · 1天前 · iOS

《YYModel源码分析（一）YYClassInfo》

4 1

专栏小东邪 · 2天前 · iOS

iOS视频采集实战(AVCaptureSession)

3

热 · 稀土君 · 14天前 · iOS

专栏

声网 Agora SDK 使用体验征文大赛

36 14