Object-C有多继承吗？没有的话用什么代替？

cocoa 中所有的类都是NSObject 的子类, 多继承在这里是用protocol 委托代理 来实现的  
你不用去考虑繁琐的多继承 ,虚基类的概念. 多态特性 在 obj-c 中通过委托来实现.

Object-C有私有方法吗？私有变量呢？

objective-c – 类里面的方法只有两种, 静态方法和实例方法. 这似乎就不是完整的面向对象了,按照OO的原则就是一个对象只暴露有用的东西. 如果没有了私有方法的话, 对于一些小范围的代码重用就不那么顺手了. 在类里面声名一个私有方法  
@interface Controller : NSObject { NSString \*something; }  
+ (void)thisIsAStaticMethod; (工厂方法)  
– (void)thisIsAnInstanceMethod;(实例方法)  
@end  
@interface Controller (private)

-(void)thisIsAPrivateMethod;  
@end

@private可以用来修饰私有变量  
在Objective‐C中，所有实例变量默认都是私有的，所有实例方法默认都是公有的

关键字const什么含义

const意味着”只读”，下面的声明都是什么意思？  
const int a;  
int const a;  
const int \*a; //修饰值  
int \* const a; //修饰指针  
int const \* a const;

前两个的作用是一样，a是一个常整型数。

第三个意味着a是一个指向常整型数的指针

（也就是，整型数是不可修改的，但指针可以）。

第四个意思a是一个指向整型数的常指针

（也就是说，指针指向的整型数是可以修改的，但指针是不可修改的）。

最后一个意味着a是一个指向常整型数的常指针（也就是说，指针指向的整型数是不可修改的，同时指针也是不可修改的）。

结论：  
• 关键字const的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果  
你曾花很多时间清理其它人留下的垃圾，你就会很快学会感谢这点多余的信息。

（当然，懂得用const的程序员很少会留下的垃圾让别人来清理的。）  
• 通过给优化器一些附加的信息，使用关键字const也许能产生更紧凑的代码。  
• 合理地使用关键字const可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少bug的出现。

欲阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初  
始化，因为以后就没有机会再去改变它了；  
（2）对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指  
定为 const；  
（3）在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；  
（4）对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量；  
（5）对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。

关键字volatile有什么含义？并给出三个不同例子？

一个定义为volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。

精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是volatile变量的几个例子：  
• 并行设备的硬件寄存器（如：状态寄存器）  
• 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)  
• 多线程应用中被几个任务共享的变量

• 一个参数既可以是const还可以是volatile吗？解释为什么。  
• 一个指针可以是volatile 吗？解释为什么。

下面是答案：  
• 是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。  
• 是的。尽管这并不很常见。一个例子是当一个中服务子程序修改一个指向一个buffer的指针时。

static作用？

函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；  
（2）在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；  
（3）在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；  
（4）在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；  
（5）在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的static 成员变量。

#import和#include的区别，@class代表什么？

@class一般用于头文件中需要声明该类的某个实例变量的时候用到，在m文件中还是需要使用#import , 而#import比起#include的好处就是不会引起重复包含.

线程和进程的区别？

进程和线程都是由操作系统所体会的程序运行的基本单元，系统利用该基本单元实现系统对应用的并发性。  
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。

堆和栈的区别？

管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。  
申请大小：  
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。  
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。  
碎片问题：对于堆来讲，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。对于栈来讲，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alloca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。

Object-C的内存管理？

1.当你使用new,alloc和copy方法创建一个对象时,该对象的保留计数器值为1.当你不再使用该对象时,你要负责向该对象发送一条release或autorelease消息.这样,该对象将在使用寿命结束时被销毁.  
2.当你通过任何其他方法获得一个对象时,则假设该对象的保留计数器值为1,而且已经被设置为自动释放,你不需要执行任何操作来确保该对象被清理.如果你打算在一段时间内拥有该对象,则需要保留它并确保在操作完成时释放它.  
3.如果你保留了某个对象,你需要(最终)释放或自动释放该对象.必须保持retain方法和release方法的使用次数相等.

为什么很多内置的类，如TableViewController的delegate的属性是assign不是retain？

循环引用  
所有的引用计数系统，都存在循环应用的问题。例如下面的引用关系：  
    •    对象a创建并引用到了对象b.  
    •    对象b创建并引用到了对象c.  
    •    对象c创建并引用到了对象b.  
这时候b和c的引用计数分别是2和1。当a不再使用b，调用release释放对b的所有权，因为c还引用了b，所以b的引用计数为1，b不会被释放。b不释放，c的引用计数就是1，c也不会被释放。从此，b和c永远留在内存中。  
这种情况，必须打断循环引用，通过其他规则来维护引用关系。比如，我们常见的delegate往往是assign方式的属性而不是retain方式 的属性，赋值不会增加引用计数，就是为了防止delegation两端产生不必要的循环引用。如果一个UITableViewController 对象a通过retain获取了UITableView对象b的所有权，这个UITableView对象b的delegate又是a， 如果这个delegate是retain方式的，那基本上就没有机会释放这两个对象了。自己在设计使用delegate模式时，也要注意这点。

定义属性时，什么情况使用copy、assign、retain？

assign用于简单数据类型，如 NSInteger, double, bool;

retain和copy则用于对象;  
copy用于当a指向一个对象，b也想指向同样的对象的时候，如果用assign，a如果释放，再调用b会crash,如果用copy 的方式，a和b各自有自己的内存，就可以解决这个问题。  
retain 会使计数器加一，也可以解决assign的问题。  
另外：atomic和nonatomic用来决定编译器生成的getter和setter是否为原子操作。在多线程环境下，原子操作是必要的，否则有可能引起错误的结果。  
加了atomic，setter函数会变成下面这样：  
if (property != newValue) {  
[property release];  
property = [newValue retain];  
}🡪其实就相当于做了retain的操作

对象是什么时候被release的？

引用计数为0时。  
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Autorelease pool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Autorelease pool，这样所有的release pool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Autorelease pool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

iOS有没有垃圾回收？

Objective-C 2.0也是有垃圾回收机制的，但是只能在Mac OS X Leopard 10.5 以上的版本使用。

tableView的重用机制？

　查看UITableView头文件，会找到NSMutableArray\*  visiableCells，和NSMutableDictnery\* reusableTableCells两个结构。visiableCells内保存当前显示的cells，reusableTableCells保存可重用的cells。

TableView显示之初，reusableTableCells为空，那么tableView dequeueReusableCellWithIdentifier:CellIdentifier返回nil。开始的cell都是通过[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]来创建，而且cellForRowAtIndexPath只是调用最大显示cell数的次数。

比如：有100条数据，iPhone一屏最多显示10个cell。程序最开始显示TableView的情况是：

1. 用[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]创建10次cell，并给cell指定同样的重用标识(当然，可以为不同显示类型的cell指定不同的标识)。并且10个cell全部都加入到visiableCells数组，reusableTableCells为空。

2. 向下拖动tableView，当cell1完全移出屏幕，并且cell11(它也是alloc出来的，原因同上)完全显示出来的时候。cell11加入到visiableCells，cell1移出visiableCells，cell1加入到reusableTableCells。

3. 接着向下拖动tableView，因为reusableTableCells中已经有值，所以，当需要显示新的cell，cellForRowAtIndexPath再次被调用的时候，tableView dequeueReusableCellWithIdentifier:CellIdentifier，返回cell1。cell1加入到visiableCells，cell1移出reusableTableCells；cell2移出visiableCells，cell2加入到reusableTableCells。之后再需要显示的Cell就可以正常重用了。

ViewController 的loadView、viewDidLoad、viewDidUnload分别是什么时候调用的，在自定义ViewCointroller时在这几个函数中应该做什么工作？

由init、loadView、viewDidLoad、viewDidUnload、dealloc的关系说起:  
init方法  
在init方法中实例化必要的对象  
init方法中初始化ViewController本身

loadView方法  
当view需要被展示而它却是nil时，viewController会调用该方法。不要直接调用该方法。  
如果手工维护views，必须重载重写该方法  
如果使用IB维护views，必须不能重载重写该方法

loadView和IB构建view  
你在控制器中实现了loadView方法，那么你可能会在应用运行的某个时候被内存管理控制调用。 如果设备内存不足的时候， view 控制器会收到didReceiveMemoryWarning的消息。 默认的实现是检查当前控制器的view是否在使用。 如果它的view不在当前正在使用的view hierarchy里面，且你的控制器实现了loadView方法，那么这个view将被release, loadView方法将被再次调用来创建一个新的view。

viewDidLoad方法  
viewDidLoad 此方法只有当view从nib文件初始化的时候才被调用。  
重载重写该方法以进一步定制view  
在iPhone OS 3.0及之后的版本中，还应该重载重写viewDidUnload来释放对view的任何索引  
viewDidLoad后调用数据Model

viewDidUnload方法  
当系统内存吃紧的时候会调用该方法（注：viewController没有被dealloc）  
内存吃紧时，在iPhone OS 3.0之前didReceiveMemoryWarning是释放无用内存的唯一方式，但是OS 3.0及以后viewDidUnload方法是更好的方式  
在该方法中将所有IBOutlet（无论是property还是实例变量）置为nil（系统release view时已经将其release掉了）

在该方法中释放其他与view有关的对象、其他在运行时创建（但非系统必须）的对象、在viewDidLoad中被创建的对象、缓存数据等 release对象后，将对象置为nil（IBOutlet只需要将其置为nil，系统release view时已经将其release掉了）

一般认为viewDidUnload是viewDidLoad的镜像，因为当view被重新请求时，viewDidLoad还会重新被执行

viewDidUnload中被release的对象必须是很容易被重新创建的对象（比如在viewDidLoad或其他方法中创建的对象），不要release用户数据或其他很难被重新创建的对象

dealloc方法  
viewDidUnload和dealloc方法没有关联，dealloc还是继续做它该做的事情

ViewController的didReceiveMemoryWarning是在什么时候调用的？默认的操作是什么？

当程序接到内存警告时View Controller将会收到这个消息：didReceiveMemoryWarning

从iOS3.0开始，不需要重载这个函数，把释放内存的代码放到viewDidUnload中去。

这个函数的默认实现是:检查controller是否可以安全地释放它的view(这里加粗的view指的是controller的view属性)，比如view本身没有superview并且可以被很容易地重建（从nib或者loadView函数）。

如果view可以被释放，那么这个函数释放view并调用viewDidUnload。

你可以重载这个函数来释放controller中使用的其他内存。但要记得调用这个函数的super实现来允许父类（一般是UIVIewController）释放view。

如果你的ViewController保存着view的子view的引用，那么，在早期的iOS版本中，你应该在这个函数中来释放这些引用。而在iOS3.0或更高版本中，你应该在viewDidUnload中释放这些引用。

列举Cocoa中常见的集中多线程的实现，并谈谈多线程安全的几种解决办法，一般什么地方会用到多线程？

NSOperation NSThread  
@sychonized

怎么理解MVC，在Cocoa中MVC是怎么实现的？

MVC设计模式考虑三种对象：模型对象、视图对象、和控制器对象。模型对象代表特别的知识和专业技能，它们负责保有应用程序的数据和定义操作数据的逻辑。视图对象知道如何显示应用程序的模型数据，而且可能允许用户对其进行编辑。控制器对象是应用程序的视图对象和模型对象之间的协调者。  
ViewCotroller  
Xib

delegate和notification区别，分别在什么情况下使用？

KVC(Key-Value-Coding)  
KVO（Key-Value-Observing）  
理解KVC与KVO（键-值-编码与键-值-监看）

当通过KVC调用对象时，比如：[self valueForKey:@”someKey”]时，程序会自动试图通过几种不同的方式解析这个调用。首先查找对象是否带有 someKey 这个方法，如果没找到，会继续查找对象是否带有someKey这个实例变量（iVar），如果还没有找到，程序会继续试图调用 -(id) valueForUndefinedKey:这个方法。如果这个方法还是没有被实现的话，程序会抛出一个NSUndefinedKeyException异常错误。

(Key-Value Coding查找方法的时候，不仅仅会查找someKey这个方法，还会查找getsomeKey这个方法，前面加一个get，或者\_someKey以及\_getsomeKey这几种形式。同时，查找实例变量的时候也会不仅仅查找someKey这个变量，也会查找\_someKey这个变量是否存在。)

设计valueForUndefinedKey:方法的主要目的是当你使用-(id)valueForKey方法从对象中请求值时，对象能够在错误发生前，有最后的机会响应这个请求。

self.跟self什么区别？

id、nil代表什么？

id

id和void \*并非完全一样。在上面的代码中，id是指向struct objc\_object的一个指针，这个意思基本上是说，id是一个指向任何一个继承了Object（或者NSObject）类的对象。需要注意的是id是一个指针，所以你在使用id的时候不需要加星号。比如id foo=nil定义了一个nil指针，这个指针指向NSObject的一个任意子类。而id \*foo=nil则定义了一个指针，这个指针指向另一个指针，被指向的这个指针指向NSObject的一个子类。

nil

nil和C语言的NULL相同，在objc/objc.h中定义。nil表示一个Objctive-C对象，这个对象的指针指向空（没有东西就是空）。

内存管理 Autorelease、retain、copy、assign的set方法和含义？

1，你初始化(alloc/init)的对象，你需要释放(release)它。例如：

　　NSMutableArray aArray = [[NSArray alloc] init];

　　后，需要

　　[aArray release];

　　2，你retain或copy的，你需要释放它。例如：

　　[aArray retain]

　　后，需要

　　[aArray release];

　　3，被传递(assign)的对象，你需要斟酌的retain和release。例如：

　　obj2 = [[obj1 someMethod] autorelease];

　　对象2接收对象1的一个自动释放的值，或传递一个基本数据类型(NSInteger，NSString)时： 你或希望将对象2进行retain，以防止它在被使用之前就被自动释放掉。但是在retain后，一定要在适当的时候进行释放。

　　关于索引计数(Reference Counting)的问题

　　retain值 = 索引计数(Reference Counting)

　　NSArray对象会retain(retain值加一)任何数组中的对象。当NSArray被卸载(dealloc)的时候，所有数组中的对象会被执行一次释放(retain值减一)。不仅仅是NSArray，任何收集类(Collection Classes)都执行类似操作。例如NSDictionary，甚至UINavigationController。

　　Alloc/init建立的对象，索引计数为1。无需将其再次retain。

　　[NSArray array]和[NSDate date]等“方法”建立一个索引计数为1的对象，但是也是一个自动释放对象。所以是本地临时对象，那么无所谓了。如果是打算在全Class中使用的变量(iVar)，则必须retain它。

　　缺省的类方法返回值都被执行了“自动释放”方法。(\*如上中的NSArray)

　　在类中的卸载方法“dealloc”中，release所有未被平衡的NS对象。(\*所有未被autorelease，而retain值为1的)

类别的作用？

有时我们需要在一个已经定义好的类中增加一些方法，而不想去重写该类。比如，当工程已经很大，代码量比较多，或者类中已经包住很多方法，已经有其他代码调用了该类创建对象并使用该类的方法时，可以使用类别对该类扩充新的方法。

  注意：类别只能扩充方法，而不能扩充成员变量。

委托（举例）

委托代理（degegate），顾名思义，把某个对象要做的事情委托给别的对象去做。那么别的对象就是这个对象的代理，代替它来打理要做的事。反映到程序中，首先要明确一个对象的委托方是哪个对象，委托所做的内容是什么。  
委托机制是一种设计模式，在很多语言中都用到的，这只是个通用的思想，网上会有很多关于这方面的介绍。  
那么在苹果开发过程中，用到委托的程序实现思想如下，我主要拿如何在视图之间传输信息做个例子。  
譬如：在两个页面（UIIview视图对象）实现传值，用委托（delegate）可以很好做到！  
方法：  
类A  
@interface A：UIView  
        id transparendValueDelegate;  
       @property(nomatic, retain) id transparendValueDelegate;  
@end  
  
@implemtion A  
@synthesize transparendValueDelegate  
-(void)Function  
{   
      NSString\* value = @"hello";  
      //让代理对象执行transparendValue动作  
      [transparendValueDelegate transparendValue: value];  
}  
@end  
  
类B  
@interface B：UIView  
      NSString\* value;  
@end  
  
@implemtion B  
-(void)transparendValue:(NSString\*)fromValue  
{  
      value = fromValue;  
      NSLog(@"the value is %@ ",value);   
}  
@end  
  
//下面的设置A代理委托对象为B  
//在定义A和B类对象处：  
  
A\* a = [[A alloc] init];  
B\* b = [[B alloc] init];  
a. transparendValueDelegate = b;//设置对象a代理为对象b  
  
这样在视图A和B之间可以通过委托来传值！  
  
下面这个例子委托有两类：  
1、一个视图类对象的代理对象为父视图，子视图用代理实现让父视图显示别的子视图  
2、同一父视图下的一个子视图为另一个子视图的代理对象，让另一个子视图改变自身背景色为给定的颜色  
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝  
规范格式如下：  
@protocol TransparendValueDelegate;  
  
@interface A：UIView  
id< TransparendValueDelegate > m\_dTransparendValueDelegate;  
@property(nomatic, retain) id m\_dTransparendValueDelegate;  
@end  
//代理协议的声明  
@protocol TransparendValueDelegat<NSObject>  
{  
      -(void)transparendValue:(NSString\*)fromValue;  
}

retainCount？