-----------重要单条-------------------------------

@selector选择器（待深入整合）

@selector()基本可以等同C语言的中函数指针,只不过C语言中，可以把函数名直接赋给一个函数指针，而Object-C 的类不能直接应用函数指针，这样只能做一个@selector语法来取.它的结果是一个SEL类型。这个类型本质是 类方法的编号

例子表区别：

C函数指针

int add(int val)

{ return val+1; }

int (\* c\_func)(int val); //定义一个函数指针变量

c\_func = add ; //把函数addr地址直接赋给c\_func

object-c的选择器

@interface foo

-(int)add:int val;

@end

SEL class\_func ; //定义一个类方法指针

class\_func = @selector(add:int);

注意1. @selector是查找当前类的方法，而[object @selector(方法名:方法参数..) ] ;是取object对应类的

注意2. 查找类方法时，除了方法名,方法参数也查询条件之一.

注意3. 与@selector(add:int);等效： SEL　变量名　=　NSSelectorFromString(方法名字的字符串);

注意4. 可以运行中用SEL变量反向查出方法名字字符串：

NSString　\*变量名　= NSStringFromSelector(SEL参数);

执行获取的方法：

SEL sel　=　NSSelectorFromString(@"yourMethod:")//有参数

if([object　respondsToSelector:sel])

{ [object　performSelector:sel　withObject: @"test" ]; //如果有两个参数,使用两个withObject:参数; }

UIScrollView（待整顿为大块知识点） - Tab6 — 第5屏

实验结果：

1. 为navigation添加的Viewcontroller是全屏的，向ViewController加入的tableView也是全屏的

2. ableView在加载其content视图时，发现上层有navigationBar（为64），下层tabControllerBar（为49）

3. 于是将 拉扯点 的top缩小64，bottom缩小49（myTableView.contentInset = UIEdgeInsetsMake

(64, 49, 0, 0);）

4. 发现上层有navigationBar，就将 contentOffset 距离tableView上边界偏移-64（向下为负）

5. 在loadView之后viewdidAppear（系统修改inset和offset后），再修改contentInset为20时，由于 拉 扯原则 使得contentOffset由-64变为-20

6. 但在contentInset之后，再设置offset，即使偏移产生拉扯，也不会进行强行变更

7. contentSize长设置 小于 单元格显示视图的长 ， 拉扯效果以 contentSize长 为基点（拉扯回归位置 位于单元格序号19而不是29）

8. contentSize的设置，使得上一句offset被拉扯原则无效化

9. 重设bounds后，发现底部位置滑条上偏49度（这又是在加载时候被谁）

总结（核心：contentView）：

1. contentView是被实际操作的view，各种参数的作用对象

2. contentSize是contentView的长宽(长宽可大于frame，小的时候不遮罩所包含内容的显示) ：

CGSizeMake(400, 1000);

3. contenOffset是内容左上角相对于tableView左上角的偏移（右下偏为负） ： CGPointMake(50, -70);

4. contentInset是规定相对 contentView 4条边 的拉扯点，都相对于tableView缩进为正 ：

UIEdgeInsetsMake(100, 49, 100, 0); （该拉扯点为下右偏移 + 上偏移）

5. 每次单一属性变更，都会按照 size offset inset顺序依据原点重新加载视图（所以contentSize的设置，无 效化了offset的设置）

6. 无论你在哪里设定，系统都会预先根据navigation等controller占用屏幕的情况，系统性的给contentView 加偏 移，缩进 拉扯点等。但你缩小tableView避免遮挡时，系统不会帮你去除这份调整（坑爹）

比如设定tableView.bounds缩小后，下部的滑动横条，依旧相对于tableView偏移49

7. tableVeiw的addSubview是将view添加到contentView中，所以使用y=-height，height=height后， 增加的view会隐藏在 拉扯点 上方，而下拉会自动带出这个view

8. UIEdgeInsetsMake(100, 49, 100, 0);

panGestureRecogenizer添加注意：

1. 为tableView添加自定义的gesture指示器的时候，发现滑动失败

2. 当你这么做时，手势栈中原有的指示器被你覆盖，所以其本身的panHanler没有得到执行，所以滑动失败

3. 用你的handler引用父类handler？？先不说这是个私有方法，这么复杂的handler源代码能让你看到？

4. 若是你还是希望得到手势参数，可以直接使用原有gesture对象啊

NSLog(@"observeValueForKeyPath: \t虽然我没有为scrollView添加gesture，但是我依然能检测到： （%0.1f,%0.1f）", [myTableView.panGestureRecognizer locationInView:myTableView].x, [myTableView.panGestureRecognizer locationInView:myTableView].y);

scroll状态查询

public var tracking: Bool { get } // returns YES if user has touched. may not yet have started dragging

public var dragging: Bool { get } // returns YES if user has started scrolling. this may require some time and or distance to move to initiate dragging

public var decelerating: Bool { get } // returns YES if user isn't dragging (touch up) but scroll view is still movin

1. 触屏滑动不放手时：触屏 和 下拉 为yes

2. 静止放手时： 都为0

3. 上滑或下滑漂移时：下拉 和 加速 为 YES

4. 拉扯向上漂移和向下偏移都：仅加速为yes

5.

原生动画的启动机制

触屏结束时，先修改panState，scrollView通过监听这个参数，然后获取抬离手指前的初速度，来启动弹簧动 画，该动画初速度与回弹方向相反，所以会先向下冲一段，然后向上弹回

当你监听offset的回弹时刻后，去设定你要的inset和offset。然而该动画逐帧生成时是根据inset动态的， 你的inset设定行为影响了这个正在运行中的动画（比如其他数值正在根据inset向其靠拢，你突然下撤 inset，于是动画忽略你那时设定好的offset，瞬间出现在最低点然后向新inset靠拢）

所以你监听panState，就能抢在动画发生前设定inset，阻止动画的发生（或者说让inset和offset一致使得 动画瞬间结束）

若是想让动画彻底不实例化，也许需要让触发动画的根值为零，比如初速度等

通知队列（待拓展完善） - NSNotificationQueue

原文：

//NSNotificationQueue objects (or simply notification queues) act as buffers for notification centers (instances of NSNotificationCenter). Where as a notification center distributes notifications when posted（是通知中心的buffer）

// A notification queue maintains notifications (instances of NSNotification) generally in a first in first out (FIFO) order. notifications placed into the queue can be delayed until the end of the current pass through therun loop or until the run loop is idle.（遵循先进先出原则post）

// Duplicate notifications can also be coalesced so that only one notification is sent although multiple notifications are posted.（检查队列以保证队列成员互不相同，防止多次发出同一通知）

//When a notification rises to the front of the queue, the queue posts it to the notification center, which in turn dispatches the notification to all objects registered as observers.（通知中心向观察者 分发通知）

猜想：

有许多通知队列，一般使用默认队列defaultQueue来入队，所有队列都是FIFO的

通知在队列中排到后，就向 通知中心（队列，堆？？） post该通知对象

通知中心轮询自己的堆，根据注册表筛查现有观察者（可能是通知类对多个观察者），发现匹配的就找到该观察 者，并@selecet它的回调方法

当该通知已经触发所有匹配的观察者一次后，就被释放或排出堆

若有通知被轮询到，并执行筛查后发现没人观察，也会排出堆

实例：

// The posting style indicates when the notification queue should post the notification to its notification center.

1. NSPostWhenIdle ：posted when the run loop is idle. （并发，轮询机有空就发）

2. NSPostASAP ：posted at the end of the current notification callout or timer.

3. NSPostNow ：postedimmediately after coalescing. （联合完插入到队列首）

//A mask indicating what criteria to use when matching attributes of notification to attributes of notifications in the queue. The mask is created by combining： 不合并，依据name，依据sender

//The list of modes the notification may be posted in(它的runloop只有在modes给定的 runloopmode列表中，该队列才能正常的post通知到通知中心，nil表示defaultRunloopMode)

[[NSNotificationQueue defaultQueue] enqueueNotification:notification

postingStyle:NSPostASAP coalesceMask:NSNotificationCoalescingOnName forModes:nil];

对象归档MyNotes - NSCoder

说在前面—原理理解

1 机制描述：

NSMutableData是二进制空缓存，相当于“一次性白纸”，调用wirteToFile将自身信息原封不动写入文件

Archiver (压缩)是NSCoder 子类，相当于“持有白纸引用、编码方式、协议、控制和派发笔的人”

encode时，发现传入被code对象不是基本数据类型，于是要求这个对象自encode去

encode时，在发现传入是基本类型时(int、char)，便拿过白纸，按照编码方式开始书写

decode时，从字节流匹配的keyclass信息中，发现不是基本类型，于是要求对象自decode

decode时，从class信息中发现是基本类型，于是按照解码范式开始解码2进制流

2. decode信息是向字节流发出的，为何能够调用到date的decode方法，里面记录了该值类型class吗？

3. NSKeyedArchiver名字本身说明还存在其他不同于键值对的编码方式

创建二进制文件

NSMutableData \* theData = [NSMutableData data];

NSKeyedArchiver \* archiver = [[NSKeyedArchiver alloc]

initForWritingWithMutableData:theData];

[theData writeToFile:writableDBPath atomically:YES];

实现协议 — 编码涉及到的所有对象都需要实现

note.m编码方法：

-(void)encodeWithCoder:(NSCoder \*)aCoder {

[aCoder encodeObject:\_date forKey:@"date"];

[aCoder encodeObject:\_content forKey:@"content"];}

note.m反编码方法：

-(id)initWithCoder:(NSCoder \*)aDecoder {

self.date = [aDecoder decodeObjectForKey:@"date"];

self.content = [aDecoder decodeObjectForKey:@"content"];

return self; }

启动编码&解码

[archiver encodeObject:array forKey:ARCHIVE\_KEY];

[archiver finishEncoding];

将文件读取到字节缓存中：NSData \* theData =[NSData dataWithContentsOfFile:path];

绑定解码类到字节缓存上：NSKeyedUnarchiver \* archiver = [[NSKeyedUnarchiver alloc]

initForReadingWithData:theData];

执行解码（还是不要忘了关闭IO）：

listData = [archiver decodeObjectForKey:ARCHIVE\_KEY];

[archiver finishDecoding];

Coder猜想：protobuf为什么那么快？？

Strong与retain关键字的区别——这个不同出现在了Note和ManagedObject的property中

详细网址：http://blog.csdn.net/dong\_007\_007/article/details/7722196

ARC机制：是在getter和setter方法中，根据property中的标签，自动添加retain和release方法

property，他可以提供的功能有：提供成员变量的访问方法的声明、控制成员变量的访问权限、控制多线程时 成员变量的访问环境

synthesize的理解是：实现property所声明的方法的定义。其实说直白就像是：property声明了一些成员变

量的访问方法 ，synthesize则定义了由property声明的方法

assign：setter方法直接赋值，不进行任何retain操作，为了解决原类型与环循引用问题

retain：setter方法对参数进行release旧值再retain新值，所有实现都是这个顺序

copy：setter方法进行Copy操作，与retain处理流程一样，先旧值release，再Copy出新的对象，

retainCount为1。这是为了减少对上下文的依赖而引入的机制

nonatomic：非原子性访问，不加同步，多线程并发访问会提高性能。注意，如果不加此属性，则默认是两个

访问方法都为原子型事务访问。锁被加到所属对象实例级.所以 不加nonatomic对与多线程是安全的

[\_internal lock]; // lock using an object-level lock

id result = [[value retain] autorelease];

[\_internal unlock];

return result;

strong weak是ARC引入的

strong相当于retain，但是对于有的类型，例如NSString，则使用strong相当于使用copy

weak相当于assign，但是比后者多一点：对象被干掉时会将weak引用设为nil，而对nil发送消息都不会导 致崩溃，否则weak引用为野指针，会出现问题

-----------课外单条-------------------------------

统计Crash的工具Crashlytics — davie12581@163.com

http://www.infoq.com/cn/articles/crashlytics-crash-statistics-tools （讲解如何使用这个第三方服务）

http://www.mamicode.com/info-detail-470447.html 更详细

Crash收集原理简介：

1. 在iOS中， 收集Crash主要通过两种方式， 一种是信号量机制，因为crash通常会发出信号量，标明某 某应用崩溃了， 另一种方式是每一个应用都有一个crash handle, 即崩溃钩子， 每当程序崩溃时， 都 会执行这个回调。

2. 信号量比起崩溃句柄的区别有点像ios开发中的通知和delegate。 信号量抛出后，可以被多个捕获crash 的工具获取到，然后取当前的堆栈信息， 再利用该堆栈信息与原app的dsym文件进行比对， 就可以找 到崩溃的代码行。

3. 理论上讲， 这个信号量机制优秀于crash句柄， 因为这样的话，可以有多个收集工具并行收集， 前提是， 每个收集工具收集后，继续抛出这个异常，而不是截断这个异常，当截断后后续的其它工具就收集不到这 个异常了， 会导致其它工具收集不全的问题。 而友盟正是这样做的， 小小地ＢＳ一下。

4. 利用crash 句柄这种方式使得crash信息只能被一个收集工具所收集到，因为句柄只有一个。如果一个应 用中有多个收集工具都设置了这个句柄， 这里就得看谁最后设置这个句柄， 谁就有效。

Protocol Buffers数据传输协议

http://blog.csdn.net/caisini\_vc/article/details/5599468

写在前面：

代码关于网络层数据接收转化方面，需要单分离出“数据通信协议转化”层，且该层关于不同协议有统一接口

网络通信机制基本为：Model实例据指定方法序列化 —> 传输与接收 —> 用给定方法解析序列生成Model

简介：

是Google的一个开源项目，和XML 差不多，也就是把某种数据结构的信息，以某种格式保存起来。主要 用于数据存储、传输协议格式等场合。

和XML区别（性能问题）：

时间开销：XML格式化（序列化）的开销倒还好；但是XML解析（反序列化）的开销就不敢恭维啦。俺之前 经常碰到一些时间性能很敏感的场合，由于不堪忍受XML解析的速度，弃之如敝履。

　　 空间开销：熟悉XML语法的同学应该知道，XML格式为了有较好的可读性，引入了一些冗余的文本信息。所 以空间开销也不是太好

优势：

代码生成机制：

假设一个在网络传输的订单数据格式为……；

如果使用protobuf实现，首先要写一个proto文件（不妨叫Order.proto），在该文件中添加一个名为 "Order"的message结构，用来描述通讯协议中的结构化数据；

然后，使用protobuf内置的编译器编译 该proto。由于本例子的模块是C++，你可以通过protobuf 编译器的命令行参数（看“这里 ”），让它生成C++语言的“订单包装类”；

然后调用某种socket的通讯库把序列化实例之后的字符串发送出去

修改：万一将来需求发生变更，要求给订单再增加一个“状态”的属性，那只需要在Order.proto文件中增加 一行代码。对于发送方（模块A），只要增加一行设置状态的代码；对于接收方（模块B）只要增加一行读 取状态的代码。

支持“向后兼容”和“向前兼容”：

当模块B升级了之后，它能够正确识别模块A发出的老版本的协议。由于老版本没有“状态”这个属性， 在扩充协议时，可以考 虑把“状态”属性设置成非必填 的，或者给“状态”属性设置一个缺省值

　　 当模块A升级了之后，模块B能够正常识别模块A发出的新版本的协议。这时候，新增加的“状态”属 性会被忽略。

　　 向后兼容”和“向前兼容”有啥用捏？俺举个例子：当你维护一个很庞大的分布式系统时，由于你无法同 时 升级所有 模块，为了保证在升级过程中，整个系统能够尽可能不受影响，就需要尽量保证通讯协 议的“向后兼容”或“向前兼容”。

缺点：可读性极差（二进制文件），不同于XML的自描述性，需要配备对应的protobuf文件

JSPatch APP动态更新服务平台

http://jspatch.com/Docs/intro

简介：

只需要在项目里引入极小的引擎文件，就可以使用 JavaScript 调用任何 Objective-C 的原生接口，替换任意 Objective-C 原生方法。目前主要用于下发 JS 脚本替换原生 Objective-C 代码，实时修复线上 bug

CDN：

http://baike.baidu.com/link?url=tSsHuGWuPDFYBbBwGaluPc0UhRAFfhEfVlwVAo2jWdKaixmgjgpofNMx0gRByh58zL4TsUGG9D-v8Nj2oBAT3K

Content Delivery Network，即内容分发网络。其基本思路是尽可能避开互联网上有可能影响数据传输速度和 稳定性的瓶颈和环节，使内容传输的更快、更稳定。通过在网络各处放置节点服务器所构成的在现有的互 联网基础之上的一层智能虚拟网络，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到 用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。其目的是使用户可就 近取得所需内容，解决 Internet网络拥挤的状况，提高用户访问网站的响应速度。

-----------边际单条-------------------------------------------------------

无法归类的单条

1. frame的center是相对于其父视图的坐标，layer的position是相对于其父layer的坐标，但是rootLayer的坐 标和在锚点为（0.5，0.5）的情况下，和center的数据一致

2. 类族/类簇示范 - GCD-Demo-1（photo为实体）：

是一个类簇，将普通继承实现为类簇，方便init时不用太多的不同的类，其他好处没发现。

有 通过asset对象初始化 、通过网络下载初始化 两种初始化方式，每次初始化返回的实际对象都不同

ALAsset：照片和video描述对象（包含缩略图，全屏图，高清图，url等属性）

获取图片uURL：

String类型： NSString \*url = [[[result defaultRepresentation]url]description];

URL类型： NSURL \*url = [[result defaultRepresentation]url];

获取缩略图：

CGImageRef ref = [result thumbnail];

UIImage \*img = [[UIImage alloc]initWithCGImage:ref];

获取全屏相片：

CGImageRef ref = [[result defaultRepresentation]fullScreenImage];

UIImage \*img = [[UIImage alloc]initWithCGImage:ref];

获取高清相片：

CGImageRef ref = [[result defaultRepresentation]fullResolutionImage];

UIImage \*img = [[UIImage alloc]initWithCGImage:ref];

实例：

需要的包：

#import <AssetsLibrary/AssetsLibrary.h>

#import "ELCImagePickerController.h"

弹出选择器（设置委托）：

ELCImagePickerController \*imagePickerController = [[ELCImagePickerController alloc] init];

[imagePickerController setImagePickerDelegate:self];

[self presentViewController:imagePickerController animated:YES completion:nil];

两个委托协议：

- (void)elcImagePickerController:(ELCImagePickerController \*)picker didFinishPickingMediaWithInfo:(NSArray \*)info

{

//通过遍历 字典对象的数组info info中存了被选择的图片的：地址、获取选择完后的信息

for (NSDictionary \*dictionary in info)

{

[self.library assetForURL:dictionary[UIImagePickerControllerReferenceURL]

//成功获取result的回调

resultBlock:^(ALAsset \*asset){

Photo \*photo = [[Photo alloc] initWithAsset:asset];

[[PhotoManager sharedManager] addPhoto:photo];

}

//失败的回调

failureBlock:^(NSError \*error)

{

UIAlertView \*alert = …… ;

[alert show];

}];

}

[self dismissViewControllerAnimated:YES completion:nil];

}

//取消选择后的回调方法（iphone中，让自身controller消失）

- (void)elcImagePickerControllerDidCancel:(ELCImagePickerController \*)picker

{

[self dismissViewControllerAnimated:YES completion:nil];

}

堆栈分析

范例：

0 CoreFoundation 0x0000000102155e65 \_\_exceptionPreprocess + 165

1 libobjc.A.dylib 0x0000000101bccdeb objc\_exception\_throw + 48

2 CoreFoundation 0x000000010215e48d -[NSObject(NSObject) doesNotRecognizeSelector:] + 205

1 2 3 4 5 1. 堆栈输出序号，序号越大表示越早被调用

2. 调用方法所属的框架（或库），或者工程名

3. 调用方法所属的类名

4. 调用方法

5. 调用方法编译后的代码偏移量（不是行号），没用处

分析方法：

堆栈信息需要从下往上看，程序运行是从下往上调用的

项目MyNotes - plist文件的使用

分层架构设计初探

1. 持久层包含实体Bean和Dao两个类文件

持久文件的初始化必须出现在代码某处，并且在安装期间唯一执行

Bean相当于某个表中的一行，是一个实体类，数据单元，Model

Dao相当于由许多行组成的实际数据表，面向持久化文件负责。有的文件通过string持久，有的是二进 制，Dao使用合适的数据类型，完成和持久文件的通信（plist全是字符串，所以数字需要转为字符）

Dao核心在于和持久文件通信的方法，所以一般是单例的

Dao接收实体Bean，转义完成持久存放；读取的持久信息，需要转义而返回bean

plist增删，需要NSArray完成，所以每次需要生成新NSArray，操作这个array，再写回plist）

2. BusinessLogicLayer处理数据逻辑，单例访问Dao接口

该层是Dao层与实际项目的转接口，Bean数据类型和Dao持久通信已经封装好，但返回的Bean的数据 形式需要在该层做一个包装

该层才应该持有NSArray<Bean>，而不是表现层，并且也是单例的，内存中至少维持一份常用数据表的 BeanArray形式，然后同步增删，或异步增删

3. 表现层，单例访问Business接口

由于tableView需要一份NSArray来刷新cell，所以作者把持有放在这里了

数据类型转换

1. http://blog.csdn.net/bailu66/article/details/7665357 Data 、String 、Byte 、Image 、mutableData

plist文件的使用

1. 如何创建看着看着就明白了

2. 读取：NSMutableArray \*array = [[NSMutableArray alloc] initWithContentsOfFile:path];

3. 写入：[array writeToFile:path atomically:YES];

atomically：原子性，先写入附件，完成写入后再重命名该文件为目标文件。若为否，则直接写入目标文件

NSDateFormatter格式类 + 字典字面量初始化

NSDateFormatter \*dateFormat = [[NSDateFormatter alloc] init];

[dateFormat setDateFormat: @"yyyy-MM-dd HH:mm:ss"];

[dateFormat stringFromDate: model.date]

NSDate \*date = [dateFormatter dateFromString:[dict objectForKey:@"date"]];

比较日期相等：[date isEqualToDate:model.date]

NSDictionary\* dict = [NSDictionary dictionaryWithObjects:

@[ [dateFormat stringFromDate: model.date] , model.content ]

forKeys:@[ @"date",@"content" ] ];

----------单条未完成知识-------------------------------

单条问题

1. 多线程与编译运行猜想 与 Runloop源码与猜想：

Darwin系统的XUN内核，是基于Mach和BSD的

**enum** CFRunLoopActivity

使用select系统调用写过程序你便可以快速的理解runloop事件源的概念

事件源的机制和select 一样是一种多路复用IO的实现

runloop有时运行经过其他模式？？

源不仅可以成为消息的被影响者，也可以发出消息被runloop监听？？

runloop回调任务的过程中，source接收到port的信号该怎么办？放入source持有的信号队列？处理时又 该怎么办？处理所有信号还是队列最开头的信号？？

runloop回调任务的过程中，多个source同时接到port又怎么办？待处理的source队列？？

2.

FMDB第三方（待学）

here：

SQLite基础知识（待学）

here：

FMDB第三方（待学）

here：

用到的库的依赖层次（待研究）

NS系列/Cocoa/Foundatio /提供GCD

/UIKit/CA系列是哪一层的

CoreFoundation/CF系列

C语言库/PThread协议（导入PThread.h）/

Mach Port相关函数 又是哪一层的？



-----------触摸事件（待学，有链接）--------------------------------------------

-----------通知推送（待学，有链接）----------------------------------------------

-----------绘图（待学，有链接）：Quartz 2D-----------------------------------------

提及处（待研究）：

Tab6 — 第二屏一个方法 、GCD-Demo-1中detailcontroller大量提及

Tab6 — 第二屏及KCLayer绘图研究

CIContext处理图片

提及：Tab6—pageView中翻转和模糊图片

-----------网络（主学习和整理到单独文档）---------------------------------------------------

-----------KVO（网址待深入）-------------------------------

KVO基本概述

KVO是Object-C中定义的一个通知机制，其定义了一种对象间监控对方状态的改变，并做出反应的机制。对 象可 以为自己的属性注册观察者，当这个属性的值发生了改变，系统会对这些注册的观察者做出通知。其用途 十分广泛，比方说，你的下载进度条是根据下载百分比决定的，那么，可以通过观察下载百分比的改变，刷新 进度条的样式，来直观的反应下载进度等等。

1. 为对象的属性注册观察者：对象通过调用下面这个方法为属性添加观察者

- (void)addObserver:(NSObject \*)observer

forKeyPath:(NSString \*)keyPath

options:(NSKeyValueObservingOptions)options

context:(void \*)context

observer: 观察者对象. 其必须实现方法observeValueForKeyPath: ofObject: change: context:.

keyPath: 被观察的属性的路径，其不能为nil.

options: 设定通知观察者时传递的属性值，是传改变前的呢，还是改变后的，具体的设定可以这儿：

NSKeyValueObservingOptionNew

NSKeyValueObservingOptionOld

NSKeyValueObservingOptionInitial

NSKeyValueObservingOptionPrior

可以多个（所以change是字典类型）：(NSKeyValue……OptionNew | NSKeyValueObservingOptionOld)

context: 一些其他的需要传递给观察者的上下文信息，通常设置为nil

2. 观察者接收通知，并做出处理:观察者通过实现下面的方法，完成对属性改变的响应：

- (void)observeValueForKeyPath:(NSString \*)keyPath

ofObject:(id)object

change:(NSDictionary \*)change

context:(void \*)context

{

// CGFloat newCurveHeight = [(NSNumber\*)change[NSKeyValueChangeNewKey] floatValue];

CGFloat newCurveHeight = [(NSNumber\*)[change objectForKey:@"NSKeyValueChangeNewKey"] floatValue];

}

keyPath: 被观察的属性的路径，其不能为nil.

ofObject: 被观察者的对象.

change: 属性值，根据上面提到的Options设置，给出对应的属性值

context: 上面传递的context对象。

注意：在使用该方法并用if判断捕获需要监听的属性后，不要忘了在else调用super的该方法

3. 清除观察者:对象通过下面这个方法移除观察者：

- (void)removeObserver:(NSObject \*)anObserver forKeyPath:(NSString \*)keyPath

使用手动通知（深入点）

在值放生变化是一般框架使用自动通知（对任何被监控的成员发生改变时，父类automatically

NotifiesObserversForKey方法返回YES，可以覆盖这个方法针对需要的属性返回NO）

//重新实现NSObject类中的automaticallyNotifiesObserversForKey:方法，返回yes表示自动通知。

+ (BOOL)automaticallyNotifiesObserversForKey:(NSString\*)key

{

if ([key isEqualToString:@"isFinished"]) { return NO; }

else return [super automaticallyNotifiesObserversForKey:key];

}

手动通知需要在值变化时调用 willChangeValueForKey:和didChangeValueForKey: 方法通知调用者

[self willChangeValueForKey:@"earthquakeList"];

[self.earthquakeList addObjectsFromArray:earthquakes];

[self didChangeValueForKey:@"earthquakeList"];

所以又是才需要通过getter和setter访问属性

注意事项：

使用KVO消息传递机制有两个要求：

（1)观察者必须知道被观察对象，即在同一作用域。

（2）观察者还需要知道被观察对象的生命周期，因为在销毁发送者对象之前，需要取消观察者的注册。

另外：如果计划在Core Data对象上使用KVO，需要知道这跟一般的KVO使用方法有点不同。还必须结合 Core Data的故障机制(faulting mechanism)，一旦core data出现了故障，它将会触发其属性对应的观察者 (即使这些属性值没有发生改变)。

一些好的实践

当一个观察者观察多个对象的相同属性（即不同Object，但是KeyPath相同），可通过设定静态的Context 变量来区分不同的通知。

使用NSStringFromSelector(@selector(method))来获取KeyPath，而不是直接通过NSString写属性 名，这样编译器可以帮助发现属性名中的Type。

通过方法：+ (NSSet \*)keyPathsForValuesAffectingValueForKey:(NSString \*)key，通过一个Key观 察多个属性值的改变。

的

这是开始