# 基本控件

## UIImageView：

### UIImageView显示问题：

### 在默认情况下，如果没有设置UIImageView的尺寸，UIImageView默认会跟图片大小一致，如果设置了尺寸，显示的图片会被压缩或者拉伸以填满整个区域。

1. 如果不想要图片被拉伸或者压缩的太严重，可以给contentMode设置UIViewContentModeScaleAspectFit。
2. 但是上面的设置，还有另外一种情况，就是图片不规则的时候，图片的宽或者高，比frame更小的时候，会出现空白，怎么解决这个问题了，可以给contentMode设置UIViewContentModeScaleAspectFill。但是这样设置还不够，因为在默认情况，图片多出来的部分还是会显示屏幕上，如果不希望超过frame的区域显示在屏幕上需要设置clipsToBounds为YES。
3. 最后一个问题，在iPhone的retina屏幕上面，必须设置,contentScaleFactor属性。这个属性默认是1，2对应retina屏,可以通过setContentScaleFactor = [[UIScreen mainScreen] scale]来设置。
4. UIImageView和UIView显示的区别：如果将控件添加到UIView中，如果控件的位置超出了UIView的尺寸，就不会显示。但是如果将控件添加到UIImageView，如果控件的位置超出了UIImageView的尺寸，还是会显示的。如果不想让它显示，必须设置clipsToBounds属性。

## UITabbleView

### UITabbleView设置组与组之间的间距

// 设置组的头高度和尾高度

self.tableView.sectionHeaderHeight = 5;

self.tableView.sectionFooterHeight = 0;

### 二．UITabbleView设置背景颜色

/\*

在groub(组)类型的tableview，tableView会被盖上一层backgroundView（一条条斜线的View），这时候设置backgroundColor会没用的，只能先清空backgroundView，然后才能设置tableview的背景颜色

默认样式是没有backgroundView的。

\*/

self.tableView.backgroundView = nil;

self.tableView.backgroundColor = [UIColor colorWithRed:230 / 250.0 green:230 / 250.0 blue:230 / 250.0 alpha:1];

# UINavgationViewController

## 设置全部导航条的背景

// 取出全部导航条

UINavigationBar \*bar = [UINavigationBar appearance];

// 设置全部导航条的背景图片

[bar setBackgroundImage:[UIImage imageName: @"navigationbar\_background.png"] forBarMetrics:UIBarMetricsDefault];

// 导航栏上有一层BackgroundImageView,不能直接设置背景颜色，设置背景颜色是无效的

// bar.backgroundColor = [UIColor colorWithPatternImage:[UIImage imageNamed:@"navigationbar\_background.png"]];

// 设置导航栏文字的主题

[bar setTitleTextAttributes:@{

UITextAttributeTextColor : [UIColor darkGrayColor] UITextAttributeTextShadowOffset : [NSValue valueWithUIOffset:UIOffsetZero]

}];

## 二．设置导航条上所有Item

// 设置全部item的背景

// 取出导航条上所有item

UIBarButtonItem \*items = [UIBarButtonItem appearance];

// 设置item的背景

[items setBackgroundImage:[UIImage imageNamed:@"navigationbar\_button\_background.png"] forState:UIControlStateNormal barMetrics:UIBarMetricsDefault];

[items setBackgroundImage:[UIImage imageNamed:@"navigationbar\_button\_background\_pushed.png"] forState:UIControlStateHighlighted barMetrics:UIBarMetricsDefault];

// 设置item的文字主题

NSDictionary \*dict = @{UITextAttributeTextColor : [UIColor darkGrayColor],

UITextAttributeTextShadowOffset : [NSValue valueWithUIOffset:UIOffsetZero],

UITextAttributeFont : [UIFont systemFontOfSize:13]

};

[items setTitleTextAttributes:dict forState:UIControlStateNormal];

[items setTitleTextAttributes:dict forState:UIControlStateHighlighted];

## 三．设置状态栏模式

// 设置状态栏,因为给导航条设置了颜色，状态栏会随着导航条的颜色改变而改变，所以我们需要设置为黑色。

[UIApplication sharedApplication].statusBarStyle = UIStatusBarStyleBlackOpaque;

## 四．导航控制器不能设置左右item和中间的文字，导航控制器本身不具备显示功能，他是通过跟控制器显示的，不要傻乎乎的给导航控制器设置左右item。

## 五．如果需要在跳转页面的时候，做一些操作效果，可以自定义导航控制器重写push或者pop方法（重写这个方法，也可以取消一些不想要的效果），或者作为导航控制器的代理，监听跳转方法。

- (void)navigationController:(UINavigationController \*)navigationController didShowViewController:(UIViewController \*)viewController animated:(BOOL)animated

- (void)navigationController:(UINavigationController \*)navigationController willShowViewController:(UIViewController \*)viewController animated:(BOOL)animated

# 通知

## 一．基本概念

1. ios提供了两种通知机制，本地通知，推送通知。

2.本地通知和推送通知可以让不在前端运行的程序告知用户程序内部发生的信息。

3.通知的展示：播放声音，更新应用程序图标上的数字，显示一个横幅。

## 二．本地通知

1.通常使用本地通知，是在应用程序退出到后台调度的。应用程序退出后台之后，会在内存中驻留10分钟的时间，过了10分钟，系统会自动清理应用程序。

2.在进入后台的方法中创建本地通知。

- (void)applicationDidEnterBackground:(UIApplication \*)application

{

[self loadNotification];

}

#pragma mark - 创建本地通知

- (void)loadNotification

{

UILocalNotification \*notification = [[UILocalNotification alloc]init];

// 5秒钟之后触发

notification.fireDate = [NSDate dateWithTimeIntervalSinceNow:5];

notification.alertBody = @"出大事了";

notification.soundName = UILocalNotificationDefaultSoundName;

notification.applicationIconBadgeNumber = 100;

notification.userInfo = @{@"userName": @"马大哈", @"age": @(18)};

// 要调用本地通知，需要通过UIApplication来统一调度

[[UIApplication sharedApplication]scheduleLocalNotification:notification];

}

3.发送的通知怎么获取本地通知

- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions

{

// 截获本地通知，用户通过通知的横幅点击进入系统，字典中才能找到本地通知

UILocalNotification \*notification = launchOptions[UIApplicationLaunchOptionsLocalNotificationKey];

// 判断通知是否存在

if (notification) {

NSLog(@"%@", notification.userInfo);

[notification.userInfo writeToFile:@"/users/apple/Desktop/123.plist" atomically:YES];

}

self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

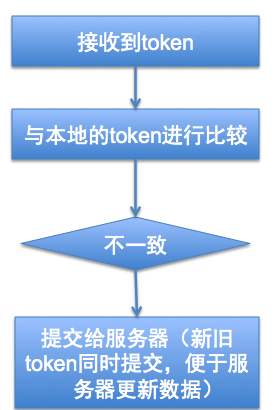
}

## 三．推送通知

1.消息推送服务，简称APNS。

2.推送流程：首先应用程序需要注册接收消息，然后APNS会返回deciceToken，将token保存到本地，提交开发商的数据服务器，服务器保存用户的token，服务器向APNS发送消息，APNS发送消息给客户的应用程序。

3.在获取到token时，发送给开发商服务器之前，需要做一步优化。



* 4.Device Token的获取，Device Token是由用户手机发起请求时，由APNS生成的。注意：DeviceToken的生成机制可能随时变化，最好方法每次获取到deviceToken后，与之前（如果有）的deviceToken进行比较，如果发生变化，及时更新自己数据服务器上的记录

4.1 向APNS发送请求，即注册应用程序接收APNS推送消息

- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions

{

// 注册应用程序接收APNS推送的消息

[application registerForRemoteNotificationTypes:UIRemoteNotificationTypeAlert | UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeSound];

self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}

4.2 获取Device Token。

#pragma mark - 获取DeviceToken

- (void)application:(UIApplication \*)application didRegisterForRemoteNotificationsWithDeviceToken:(NSData \*)deviceToken

{

NSLog(@"%@", deviceToken);

// 1. 从系统偏好取之前的token

NSData \*oldToken = [[NSUserDefaults standardUserDefaults]objectForKey:@"deviceToken"];

// 2. 新旧token进行比较

if (![oldToken isEqualToData:deviceToken]) {

// 3. 如果不一致，保存token到系统偏好

[[NSUserDefaults standardUserDefaults]setObject:deviceToken forKey:@"deviceToken"];

// 4. 使用post请求传输新旧token至服务器

// 1) url

// 具体的URL地址以及POST请求中的参数和格式，是由公司的后端程序员提供的

// 2) request POST body（包含新旧token的数据）

// 3) connection 的异步

}

}

# 新浪微博

## 一．新特性界面知识点

1. 在loadView里面设置控制器视图为UIImageView，避免多创建一个UIView，提供性能。
2. 控制器视图：UIImageView 需要允许交互，否则内部子控件不能交互
3. 添加一个UIScrollView需要设置的属性：

3.1 consize（由于不需要滚动，y轴方向可设置为0）

3.2 setShowsHorizontalScrollIndicator为NO

3.3 setPagingEnabled为YES 开启分页

4. 在UIScrollView中添加UIImageView，设置UIImageView的图片时，需要做屏幕适配，根据屏幕，加载不同的图片。（单独写一个分类用于加载图片）

5. 给NSString写一个分类，用来根据不同屏幕，加载不同图片。

5.1 如何做屏幕适配：判断屏幕的高度，是否是568，如果是，就是iPhone5.

## 二．Dock界面知识点

1. 自定义一个UIView,给外界提供一个接口，用于添加dockItem,在每次添加一个Item的时候，重新计算每个Item的位置.在添加方法中，默认选中第一个item。

2. 自定义UIButton，设置里面的UIImageView和UILabel的比例尺寸。

3. 给DockView设置一个代理，当点击dockItem时，调用代理的方法，切换界面的显示。

4. 自定义一个控制器，作为DockView的代理，实现一个方法，改变界面的显示。

5. 在手写代码的时候，创建控制器的时候，需要自己设置frame，这个不要忘记了，没创建一个新的控制器，大小都需要自己设置，其他的storyboard可以不用自己设置。

## 三．Main控制器知识点

1. 创建一个MainController控制器继承至DockController,这个控制器只管添加，需要显示多少个item和有多少个子控制器就行了，其他的代理方法不需要重写，因为控制器已经实现了相应的功能了。

2. 自定义一个UINavgationViewController，因为微博中，每个控制器都有一个导航条，我们需要将每个控制器包装成一个导航控制器，就能使每个控制器都有导航条了。

3. 自定义一个UINavgationViewController，是为了统一设置自定义导航条和导航栏上面Item的显示，使自定义UINavgationViewController的子控制器都有相同的外观。

## 四，设置更多界面知识点

1. 需要设置UITableView的组与组之间的间距
2. 每一个cell的背景不一样，右边的辅助视图显示的不一样,我们可以单独自定义cell，给外界提供一个接口，来设置。
   1. 需要提供一个设置indexPath的属性，用来设置每一行的背景图片和选中时的图片，这里需要判断一组都有多少个cell，才能设置相应的背景图片，因此需要传入一个UITableView，然后根据UITableView取出对应的组有多少行。

#pragma mark - 设置每一行的背景图片和选中图片

- (void)setIndexPath:(NSIndexPath \*)indexPath

{

\_indexPath = indexPath;

// 设置cell的背景和选中时背景

NSInteger count = [\_groupTableView numberOfRowsInSection:indexPath.section];

if (count == 1) {

// 如果只有一个

\_bg.image = [UIImage stretchImage:@"common\_card\_background.png"];

\_selectedBg.image = [UIImage stretchImage:@"common\_card\_background\_highlighted.png"];

}else if (indexPath.row == 0){

// 如果是第一行

\_bg.image = [UIImage stretchImage:@"common\_card\_top\_background.png"];

\_selectedBg.image = [UIImage stretchImage:@"common\_card\_top\_background\_highlighted.png"];

}else if (indexPath.row == count - 1){

// 如果是最后一行

\_bg.image = [UIImage stretchImage:@"common\_card\_bottom\_background.png"];

\_selectedBg.image = [UIImage stretchImage:@"common\_card\_bottom\_background\_highlighted.png"];

}else{

// 中间行

\_bg.image = [UIImage stretchImage:@"common\_card\_middle\_background.png"];

\_selectedBg.image = [UIImage stretchImage:@"common\_card\_middle\_background\_highlighted.png"];

}

}

* 1. 自定义cell中设置了一个枚举类型，用来根据枚举类型，设置右边对应的辅助视图。

#pragma mark - 每一行的右边辅助视图

- (void)setCellType:(CellType)cellType

{

\_cellType = cellType;

if (cellType == kCellTypeArrow) {

if (\_accessoryArrow == nil) {

\_accessoryArrow = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"common\_icon\_arrow.png"]];

}

self.accessoryView = \_accessoryArrow;

}else if (cellType == kCellTypeLabel){

if (\_accessoryLabel == nil) {

UILabel \*label = [[UILabel alloc] init];

label.backgroundColor = [UIColor clearColor];

label.textAlignment = NSTextAlignmentCenter;

label.textColor = [UIColor grayColor];

label.font = [UIFont systemFontOfSize:12];

// 设置尺寸

label.bounds = (CGRect){CGPointZero,{80,44}};

\_accessoryLabel = label;

}

self.accessoryView = \_accessoryLabel;

}else if (cellType == kCellTypeNone){

self.accessoryView = nil;

}else

{

if (\_accessorySwitch == nil) {

\_accessorySwitch = [[UISwitch alloc] init];

}

self.accessoryView = \_accessorySwitch;

}

}

1. 给UITableView添加尾部按钮的时候，需要注意UITableView的tableFooterView的宽度不需要设置，默认就是屏幕的宽度，反而你设置了，还不好，tableFooterView会向左边偏移你设置的宽度，一般如果我们需要让自定义的tableFooterView居中，我们有两种方式。
   1. 一种就是自定义UIButton，重写imageRectForContentRect方法，设置内部UIImageView的尺寸，这里也不能设置contentEdgeInsets属性，也会向右偏移设置的宽度。只能通过这种方法设置。

- (CGRect)imageRectForContentRect:(CGRect)contentRect

{

CGFloat x = 10;

CGFloat y = 0;

CGFloat width = contentRect.size.width - 2 \* x;

CGFloat height = contentRect.size.height;

return CGRectMake(x, y, width, height);

}

这样还不够，还需要给UITableView添加额外的滚蛋区域，使底部留出间距。

self.tableView.contentInset = UIEdgeInsetsMake(0, 0, 10, 0);

* 1. 另外一种方式是：写一个UIButton然后在创建一个UIView将按钮添加到UIView中，并居中显示即可，将UIView作为UITableView的tableFooterView。注意创建UIView时高度需要设置多一点，就能上下留出间距。

// 创建退出按钮

UIButton \*btnExit = [UIButton buttonWithType:UIButtonTypeCustom];

btnExit.frame = CGRectMake(10, 10, 300, 40);

// 设置按钮文字

[btnExit setTitle:@"退出登录" forState:UIControlStateNormal];

// 设置背景颜色

[btnExit setAllStateBgWithImageName:@"common\_button\_big\_red.png"];

// 监听按钮点击

[btnExit addTarget:self action:@selector(exit) forControlEvents:UIControlEventTouchUpInside];

// 创建一个视图，作为tableView的脚视图

UIView \*footView = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 70)];

[footView addSubview:btnExit];

self.tableView.tableFooterView = footView;

## 五．Oauth认证控制器知识点

1. 默认背景视图设置为UIWebView通过loadRequest方法访问网页.

2. 需要设置UIWebView的代理，拦截发送的请求。

3. 当请求成功，会跳转到回调界面，并且在回调界面添加请求标记，我们需要拦截到回调界面的请求，并取出对应的请求标记，判断请求的路径是否有code=，如果有代表授权成功，我们需要拿到请求标记换取access\_Token.

// 获取全路径

NSString \*urlStr = request.URL.absoluteString;

// 查找code=范围

NSRange range = [urlStr rangeOfString:@"code="];

if (range.length > 0) {

// 说明授权成功，并返回了请求标记request\_token

// 获取request\_token

int index = range.location + range.length;

NSString \*requestToken = [urlStr substringFromIndex:index];

// 换取access\_Token

[self getAccessToken:requestToken];

// 不需要返回到回调页面

return NO;

}

1. 拿到请求标记后，需要用post请求获取access\_Token。这里需要使用第三方框架，AFN，但是这个框架我们以后可能会换成另外一个框架，所以这里我们采取一个设计思想，弄一个工具类，像外界提供一个接口，用于发送get和post请求，然后返回获取到的数据即可，至于以后用什么框架，只需要修改工具类即可，方便以后换其他框架。
2. 请求工具类的设计：

5.1提供一个httpRequestWithMethod:(NSString \*)method path:(NSString \*)path params:(NSDictionary \*)params success:(HttpSuccessBlock)success failure:(HttpFailureBlock)failure接口，传入这些参数，自动帮你发送请求，如果成功就返回数据给你，不成功返回失败给你。

5.2 + (void)httpRequestWithMethod:(NSString \*)method path:(NSString \*)path params:(NSDictionary \*)params success:(HttpSuccessBlock)success failure:(HttpFailureBlock)failure

{

// 初始化委托，用来创建请求

AFHTTPClient \*client = [AFHTTPClient clientWithBaseURL:[NSURL URLWithString:kBaseUrl]];

NSMutableDictionary \*allParams = [NSMutableDictionary dictionary];

// 这里需要判断是否有参数,否则会报错

if (params) {

// 有才需要拼接

[allParams setDictionary:params];

}

// 判断是否有access\_token,如果没有就不拼接，否则给字典传空值会报错。在很多地方都需要用到access\_token,所以就在内部拼接好

Account \*account = [AccountTool shareAccountTool].account;

if (account.accessToken) {

// 拼接access\_token

[allParams setObject:account.accessToken forKey:@"access\_token"];

}

// 创建post请求

NSMutableURLRequest \*request = [client requestWithMethod:method path:path parameters:allParams];

// 初始化AFJSONRequestOperation对象

AFJSONRequestOperation \*op = [AFJSONRequestOperation JSONRequestOperationWithRequest:request success:^(NSURLRequest \*request, NSHTTPURLResponse \*response, id JSON) {

// 在使用block时，一定要加上这句话，否则会报错.

if (success == nil) return;

success(JSON);

} failure:^(NSURLRequest \*request, NSHTTPURLResponse \*response, NSError \*error, id JSON) {

if (failure == nil) return;

failure(error);

}];

// 开始请求

[op start];

}

6. 返回的是一个JSON数据，由字典组成，直接按照解析字典的方式解析即可，我们需要拿到字典里的access\_token 和 uid , 我们需要创建一个账号模型，并创建一个工具类，用于账号的归档和解档。

## 六．主界面知识点

### 1. 创建一个微博模型。

### 2. 需要创建一个工具类，专门用来获取微博数据的，将获取到的JSON数据转化为微博模型。

### 3． 自定义一个cell，将微博中所有的内容先添加到cell中，尺寸这些先不考虑。

### 4． 默认获取20条微博数据,从返回的JSON可以看出，我们需要创建一个数组来存入获取到的微博数据。

### 5. 我们知道获取到的微博数据都是需要设置到cell上的，而每一个cell的位置尺寸需要通过每一行显示的微博计算出位置，所以我们需要在建立一个微博frame模型，用于计算出每个微博数据在cell的frame，所以需要给模型传入一个微博用于计算它每一个内容的显示。

### 6. 所以我们直接创建一个微博frame数组，在获取到微博数据的时候，就将每个微博内部的数据位置计算出来，并保存到数组，然后重新刷新UITableView。

### 7. 给cell设置一个接口，用于设置cell内部每个子控件的尺寸，并且设置显示内容。

### 8. cell中每个微博数据都显示出来之后，在慢慢细化，首先头像的业务逻辑比较复杂，有很多类型的头像，而且每一个头像在不同的节目大小也不一样，所以我们自定义一个头像类继承UIView。

8.1.头像：自定义头像类，提供两个接口，一个是设置头像类型，一个是设置头像内容。并且在初始化的时候就将所有子控件添加进去。

8.1.1 添加用户接口：

#pragma mark 设置用户

- (void)setUser:(User \*)user

{

\_user = user;

// 设置用户显示的图片

[\_icon setImageWithURL:[NSURL URLWithString:user.profileImageUrl] placeholderImage:[UIImage imageNamed:\_placehoder] options:SDWebImageDownloaderLowPriority | SDWebImageRetryFailed];

// 判断认证图片

NSString \*veritifyIcon = nil;

switch (user.verifiedType) {

case kVerifiedTypeNone:

\_vertify.hidden = YES;

break;

case kVerifiedTypePersonal:

veritifyIcon = @"avatar\_vip.png";

break;

case kVerifiedTypeDaren:

veritifyIcon = @"avatar\_grassroot.png";

break;

default:

veritifyIcon = @"avatar\_enterprise\_vip.png";

break;

}

if (user.verified) {

// 如果有认证，才需要添加认证图片

\_vertify.hidden = NO;

\_vertify.image = [UIImage imageNamed:veritifyIcon];

}

}

8.1.2 设置类型接口：

#pragma mark 设置图片类型

- (void)setType:(IconType)type

{

\_type = type;

// 设置icon和verity和视图的尺寸

// 判断类型

CGSize iconSize;

switch (type) {

case kIconTypeSmall: // 小图标

iconSize = CGSizeMake(kIconSmallW, kIconSmallH);

\_placehoder = @"avatar\_default\_small.png";

break;

case kIconTypeDefault: // 中图标

iconSize = CGSizeMake(kIconDefaultW, kIconDefaultH);

\_placehoder = @"avatar\_default.png";

break;

case kIconTypeBig: // 大图标

iconSize = CGSizeMake(kIconBigW, kIconBigH);

\_placehoder = @"avatar\_default\_big.png";

break;

}

// 设置icon的frame

\_icon.frame = (CGRect){CGPointZero,iconSize};

// 设置verity的frame

\_vertify.bounds = CGRectMake(0, 0, kVertifyW, kVertifyH);

\_vertify.center = CGPointMake(iconSize.width, iconSize.height);

// 设置视图的尺寸

CGFloat width = iconSize.width + kVertifyW \* 0.5;

CGFloat height = iconSize.height + kVertifyH \* 0.5;

self.bounds = CGRectMake(0, 0, width, height);

}

8.1.3向外提供一个类方法，传入一个类型，就返回头像的尺寸。 + (CGSize)iconSizeWithType:(IconType)type

{

// 判断类型

CGSize iconSize;

switch (type) {

case kIconTypeSmall:

iconSize = CGSizeMake(kIconSmallW, kIconSmallH);

break;

case kIconTypeBig:

iconSize = CGSizeMake(kIconBigW, kIconBigH);

break;

case kIconTypeDefault:

iconSize = CGSizeMake(kIconDefaultW, kIconDefaultH);

break;

default:

break;

}

CGFloat width = iconSize.width + kVertifyW \* 0.5;

CGFloat height = iconSize.height + kVertifyH \* 0.5;

return CGSizeMake(width, height);

}

8.2 时间

8.2.1 需要将获取到的时间数据转换成我们想要显示的内容。而这个时间是不断变化的，不是固定不变的，如何能做到随着时间的变化，显示的内容也变化了。我们知道每次给cell设置内容，是在设置frame的时候同时设置内容的，而每次获取内容都是从微博数据中获取的，所以我们的突破口就是在微博模型中重写时间的get方法，就能在让每次cell需要设置时间时，从微博模型拿到的时间，都是我们已经修改后的数据了。

#pragma mark - 重写时间的get方法，1.修正显示内容 2.因为每次拖动的时候，我们需要修改时间显示的内容

- (NSString \*)createdAt

{

// Sat Nov 02 15:08:27 +0800 2013 将这这样的字符串解析成date对象。

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

NSLocale\* local = [[NSLocale alloc] initWithLocaleIdentifier:@"en\_US"];

[fmt setLocale: local];

fmt.dateFormat = @"EEE MMM d HH:mm:ss zzzz yyyy";

// 将字符串按照日期格式转化成日期对象

NSDate \*date = [fmt dateFromString:\_createdAt];

float delta = fabs([date timeIntervalSinceNow]);

// 这里仅仅设置了文字是不够的，还需要在

if (delta < 60) {

// 一分钟内

return [NSString stringWithFormat:(@"1分钟前")];

}else if (delta < 60 \* 60){

// 一小时内

return [NSString stringWithFormat:@"%0.f分钟前",delta / 60];

}else if (delta < 60 \* 60 \* 24){

// 一天之内

return [NSString stringWithFormat:@"%0.f小时前",delta / 60 / 60 ];

}else{

fmt.dateFormat = @"MM-dd HH:mm";

return [fmt stringFromDate:date];

}

}

8.3 来源

8.3.1 来源的内容我们也需要修正，但是来源内容是固定不变的，所以我们重写set方法，如果重写get方法比较耗性能，每次cell更新都会调用。

- (void)setSource:(NSString \*)source

{

// 2013-11-02 19:24:05.519 YZ-weibo[1611:907] <a href="<http://weibo.com/>" rel="nofollow">新浪微博</a>

NSInteger begin = [source rangeOfString:@">"].location + 1;

NSInteger end = [source rangeOfString:@"</"].location;

\_source = [@"来自" stringByAppendingString:[source substringWithRange:NSMakeRange(begin , end - begin)]];

}

8.4 注意修复时间和来源的内容时，在statusCellFrame里也需要修复他们两个的尺寸，还有每次更新cell时，也需要重新计算这两个的尺寸和位置，因为时间的内容会随着cell的更新，而改变内容，这时候尺寸和位置也需要改变。

8.5 配图：配图的业务逻辑比较多，所以我们自定义一个配图View，思路跟自定义头像差不多，向外界提供一个类方法，根据有多少图返回对应的高度，并且提供一个方法，接收需要加载的image数组。

8.5.1 设置图片image数组接口,并设置里面每个图片的位置和尺寸。

- (void)setImageUrls:(NSArray \*)imageUrls

{

\_imageUrls = imageUrls;

// int count = imageUrls.count;

int count = imageUrls.count > 4 ? 4 : imageUrls.count;

// 设置每个UIImageView的位置

for (int i = 0; i < kCount; i++) {

// 取出对应位置的子控件

ImageItemView \*child = self.subviews[i];

// 判断要不要显示图片

if (i >= count ) {

// 不要显示

child.hidden = YES;

continue;

}

// 需要显示

child.hidden = NO;

// 设置图片

child.url = imageUrls[i][@"thumbnail\_pic"];

// 设置frame

if (count == 1) {

// 只有一张图片

// UIImageView默认会填充整个视图，所以想要好看点，需要设置内容模式

child.contentMode = UIViewContentModeScaleAspectFit;

child.frame = CGRectMake(0, 0, kOneW, kOneH);

continue;

}

// 设置图片显示模式

child.contentMode = UIViewContentModeScaleAspectFill;

child.clipsToBounds = YES;

// 多张图片展示

// 列数,如果总数等于4，就是2列。

int sumCols = (count == 4)?2 : 3;

CGFloat row = i / sumCols; // 行号

CGFloat col = i % sumCols; // 列号

CGFloat x = (kMultiW + kMargin) \* col;

CGFloat y = (kMultiH + kMargin) \* row;

child.frame = CGRectMake(x, y, kMultiW, kMultiH);

}

}

8.5.2 提供一个类方法，根据图片总数返回对应尺寸

+ (CGSize)imageListWithSizeCount:(int)count

{

count = count > 4 ? 4 : count;

if (count == 1) {

// 如果只有一张图片

return CGSizeMake(kOneW, kOneH);

}

// 计算出总列数

CGFloat cols = 3;

// 判断总数是否是2 或者 4，如果是 2 或者 4 列数就为2.

if (count == 4 || count == 2) cols = 2;

// 计算出总行数

int rows = (count - 1) / cols + 1;

CGFloat width = cols \* kMultiW + kMargin \* (cols - 1);

CGFloat height = rows \* kMultiH + kMargin \* (rows - 1);

return CGSizeMake(width, height);

}

8.6 配图里每个图片也需要自定义，如果是gif需要在右下角添加一个gif。

8.6.1 首先把gif图片添加到自定义ImageItemView中,注意这里不能设置gif图片的位置和尺寸。

8.6.2 给外界提供一个接口，设置显示什么图片

- (void)setUrl:(NSString \*)url

{

\_url = url;

// 设置图片

[HttpTool loadImageForImageView:self url:url placeholder:[UIImage imageNamed:@"timeline\_image\_loading.png"]];

// 判断是否显示gif

\_gifView.hidden = ![url hasSuffix:@"gif"];

}

8.6.2 重写setFrame方法，只有ImageItemView有尺寸了，我们才能设置gif的尺寸

// 重写frame的set方法，当ImageItemViewframe有值了，就能设置内部gif视图的尺寸了

- (void)setFrame:(CGRect)frame

{

// 设置gif图片的尺寸

CGFloat gifX = frame.size.width - \_gifView.bounds.size.width;

CGFloat gifY = frame.size.height - \_gifView.bounds.size.height;

\_gifView.frame = (CGRect){CGPointMake(gifX, gifY),\_gifView.bounds.size};

[super setFrame:frame];

}

8.7 调节cell的间距，重写cell的setFrame方法

#pragma mark - 设置cell的间距 重写setFrame方法

- (void)setFrame:(CGRect)frame

{

// 设置两边的间距

frame.origin.x = kTableBorderWidth;

frame.size.width -= kTableBorderWidth \* 2;

// 设置cell与tableView的间距(如何设置上下间距，可以让cell的所有y值向下移动就能腾出最上面的间距)

frame.origin.y += kTableBorderWidth;

// cell之间的间距怎么调节，可以在模型cellFrame中将所有cell的高度增加我们想要设置的间距，这样每个cell的y值就比最初的y值多了间距这么多的距离，然后在重新给cell设置高度的frame方法中，将多余的间距减掉，这样就能让每个cell保持原有高度的同时，cell之间的y值距离也有间距那么宽。

frame.size.height -= kCellMargin;

[super setFrame:frame];

}

8.7.1 调节cell的高度时，注意要在frame模型中给cell添加kCellMargin的高度。

8.8 底部操作条

8.8.1 底部操作条由于有背景，直接继承UIImageView就能设置背景图片了。这个视图的宽高不变，我们需要在内部就固定高度

// 这个视图的宽高，保持不变的，我们需要在内部就固定宽高，重写setFrame方法

- (void)setFrame:(CGRect)frame

{

frame.size.height = kStatusDockHeight;

frame.size.width = [UIScreen mainScreen].bounds.size.width - 2 \* kCellBorderWidth;

[super setFrame:frame];

}

8.8.2 底部操作条的按钮显示跟微博数据有关，我们需要给外界提供一个接口，获取微博数据，并且通过微博数据设置按钮的文字

// 重写setStatus设置每个按钮显示的内容.

- (void)setStatus:(Status \*)status

{

\_status = status;

// 设置按钮文字

// 1.转发

[self setBtn:\_repost title:@"转发" count:status.repostsCount];

// 2.评论

[self setBtn:\_comment title:@"评论" count:status.commentsCount];

// 3.赞

[self setBtn:\_attitute title:@"赞" count:status.attitudesCount];

}

#pragma mark 设置按钮文字

- (void)setBtn:(UIButton \*)button title:(NSString \*)title count:(int)count

{

// 根据count的值设置title显示的文字

if (count >= 10000) {

// 上万

CGFloat final = count / 10000.0;

title = [NSString stringWithFormat:@"%0.1f万",final];

// 如果字符串有.0,就替换成空字符

title = [title stringByReplacingOccurrencesOfString:@".0" withString:@""];

}else if(count > 0){

// 一万以内

title = [NSString stringWithFormat:@"%d",count];

}

[button setTitle:title forState:UIControlStateNormal];

}

8.8.3 由于底部操作条是紧挨着cell底部的，所以我们可以采取自适应，拉伸顶部，就能让底部操作条随着cell的拉长，一直保持到底部的位置。只要在创建的时候，设置底部操作条距离底部的位置，不管cell变多长，底部操作条始终在底部。设置autoresizingMask为UIViewAutoresizingFlexibleTopMargin。

## 七．上下拉刷新

### 1. 用MJRefresh框架实现上下拉刷新

1.1 如何使用这个框架，只需要告诉控件的scrollView是谁，就能将框架添加到我们的滚动视图中了

// 下拉刷新

MJRefreshHeaderView \*header = [MJRefreshHeaderView header];

header.scrollView = self.tableView;

header.delegate = self;

// 开始下拉刷新

[header beginRefreshing];

// 下拉刷新

MJRefreshFooterView \*footer = [MJRefreshFooterView footer];

footer.scrollView = self.tableView;

footer.delegate = self;

1.2 如何加载数据：需要知道两个参数的意思 sinceId 和 maxId ,两个不能同时传，如果传入sinceId，就会新浪服务器就会返回比sinceId大的微博数据，也就是最新的，所以我们只需要取出当前数组最大的sinceId也就是数组第一关元素的sinceId传入到请求方法中，就能获取。如果传入maxId，新浪服务器就会返回小于等于maxId的微博数据，如果想要获取以前的数据，并且不重复，需要取出数组中最小的maxId，也是最后一个元素的maxId将maxId-1，就能获取了之前的数据了。

1.3 加载最新数据，需要将获取到的数据插入最前面

#pragma mark 下拉刷新新数据

- (void)loadNewData:(MJRefreshBaseView \*)refreshView

{

// 取出第一条微博ID

// 需要判断数组是否为空。

StatusCellFrame \*f = \_statusesFrame.count?\_statusesFrame[0] :nil;

long long sinceId = f.status.ID;

[StatusTool statusesWithPath:@"2/statuses/friends\_timeline.json" sinceId:sinceId maxId:0 success:^(NSArray \*statues) {

// 显示刷新了多少条数据

[self showNewStatusCount:statues.count];

// 将获取到的数据放置最前面

NSMutableArray \*newFrames = [NSMutableArray arrayWithCapacity:statues.count];

for (Status \*status in statues) {

// 将微博数据转换成frame模型

StatusCellFrame \*f = [[StatusCellFrame alloc] init];

f.status = status;

[newFrames addObject:f];

}

// 将数据插入到数组最前面

[\_statusesFrame insertObjects:newFrames atIndexes:[NSIndexSet indexSetWithIndexesInRange:NSMakeRange(0, newFrames.count)]];

// 刷新表格

[self.tableView reloadData];

// 停止刷新状态

[refreshView endRefreshing];

} failure:^(NSError \*error) {

[refreshView endRefreshing];

}];

}

1.4 加载更多以前的数据,直接将获取到的数据添加到最后就行了。

#pragma mark 上啦刷新更多

- (void)loadMoreData:(MJRefreshBaseView \*)refreshView

{

// 取出最后一条微博ID

StatusCellFrame \*f = [\_statusesFrame lastObject];

long long maxId = f.status.ID;

// 加载比当前小的微博数据

maxId--;

[StatusTool statusesWithPath:@"2/statuses/friends\_timeline.json" sinceId:0 maxId:maxId success:^(NSArray \*statues) {

// 将获取到的数据放置最前面

NSMutableArray \*newFrames = [NSMutableArray arrayWithCapacity:statues.count];

for (Status \*status in statues) {

// 将微博数据转换成frame模型

StatusCellFrame \*f = [[StatusCellFrame alloc] init];

f.status = status;

[newFrames addObject:f];

}

// 将数据添加到最后面

[\_statusesFrame addObjectsFromArray:newFrames];

// 刷新表格

[self.tableView reloadData];

// 让刷新控件停止刷新状态

[refreshView endRefreshing];

} failure:^(NSError \*error) {

[refreshView endRefreshing];

}];

}

2. 创建显示最新微博的数目条

2.1 这个条不能直接添加到控制器的主视图中，因为控制器的主视图是tableView，而我们的条的位置设置为主视图左上角刚好看不见的位置，当拖动tableView时，数目条就会出现，不想我们想要的效果。数目条应该添加到导航控制器上，并且在导航条的下面。

#pragma mark 展示最新微博的数目

- (void)showNewStatusCount:(int)count

{

// 1.创建按钮

UIButton \*btn = [UIButton buttonWithType:UIButtonTypeCustom];

btn.enabled = NO;

// 设置了enabled默认背景会透明，我们需要设置adjustsImageWhenDisabled为NO就能取消这个效果

btn.adjustsImageWhenDisabled = NO;

[btn setBackgroundImage:[UIImage stretchImage:@"timeline\_new\_status\_background.png"] forState:UIControlStateNormal];

CGFloat w = self.view.frame.size.width;

CGFloat h = 35;

btn.frame = CGRectMake(0, 44 - h, w, h);

NSString \*title = count?[NSString stringWithFormat:@"共有%d条新的微博", count]:@"没有新的微博";

[btn setTitle:title forState:UIControlStateNormal];

[self.navigationController.view insertSubview:btn belowSubview:self.navigationController.navigationBar];

// 2.开始执行动画

CGFloat duration = 0.5;

[UIView animateWithDuration:duration animations:^{ // 下来

btn.transform = CGAffineTransformMakeTranslation(0, h);

} completion:^(BOOL finished) {

[UIView animateWithDuration:duration delay:1.0 options:UIViewAnimationOptionCurveLinear animations:^{// 上去

btn.transform = CGAffineTransformIdentity;

} completion:^(BOOL finished) {

[btn removeFromSuperview];

}];

}];

}

## 八．跳转到微博详情界面

1.利用导航控制器实现界面的跳转。在跳转的时候，想实现二个效果，1.导航控制器的界面拉长为main视图的高度，2.当界面跳转的时候，dock能随着当前控制器视图向左移动。需要监听到导航控制器的跳转事件。由于需要拿到dock，不能重写push方法，只能在main控制器中，让main作为导航控制器的代理。监听界面开始跳转的方法。在跳转完成后，系统会自带返回按钮，太丑了，我们需要在每次跳转的时候，自定义一个返回按钮，也需要在界面开始跳转的方法里实现。

注意点：1.并不是每个控制器之间的跳转都需要实现这些功能，只有导航控制器的根控制器跳转到其他控制器才需要实现这些功能，所以在做这些功能的时候，需要先取出当前跟控制器，然后判断需要跳转的控制器是不是跟控制器，如果不是跟控制器，就做这两个效果。

2.由于根控制器的视图是滚动视图，所以需要添加contentOffset即滚动的距离，计算出dock的实际y值。

#pragma mark WBNavigationViewController的代理方法

// 下一个控制器即将要出现时调用

- (void)navigationController:(UINavigationController \*)navigationController willShowViewController:(UIViewController \*)viewController animated:(BOOL)animated

{

/\*

我们想要导航控制器推出控制器的时候，dock也会随着控制器移走，所以我们可以将dock加入到需要移走的控制器的里，就能有这种效果了。

\*/

// 0.取出导航控制器的跟控制器

UIViewController \*root = navigationController.viewControllers[0];

// 1.判断是否是根控制器

if (viewController != root) {

// 如果不是根控制器

// 2.拉长导航控制器的view

CGRect frame = navigationController.view.frame;

frame.size.height = [UIScreen mainScreen].applicationFrame.size.height;

navigationController.view.frame = frame;

// 3.添加dock到跟控制器的view上面，就能让dock随着根控制器滑动。

CGRect dockFrame = self.dockView.frame;

// dock的y值等于当前跟控制器的高度 - dock的高度

dockFrame.origin.y = root.view.frame.size.height - self.dockView.frame.size.height;

// 判断根控制的视图是否是可滚动的视图，如果是可滚动的视图，dock的y值需要加入偏移量才对

if ([root.view isKindOfClass:[UIScrollView class]]) {

UIScrollView \*scroll = (UIScrollView \*)root.view;

dockFrame.origin.y += scroll.contentOffset.y;

}

self.dockView.frame = dockFrame;

[root.view addSubview:self.dockView];

// 4.添加左上角的返回按钮

viewController.navigationItem.leftBarButtonItem = [UIBarButtonItem barButtonItemWithIcon:@"navigationbar\_back.png" target:self action:@selector(back)];

}

}

2.在从其他控制器跳转到跟控制器时，需要将控制器缩短，这个操作不能放在开始跳转的方法里实现，必须放在跳转完成之后，原因1.如果在开始跳转的时候，就缩短导航控制器，就会出现一块白的在屏幕下，因为dock添加到跟控制器上，他的y值在根控制器的高度的下面，所以就看不到了。不要认为dock的y值不是刚好在屏幕上吗，怎么看不见了，视图的显示是相对视图的尺寸的，如果你添加的视图尺寸很小，即使你的x和y都在屏幕上，也是看不见的。原因2.当跳转到跟控制器的时候，我们想要dock随着跟控制器往右挪动，而挪动，所以只能在完成跳转才缩短导航控制器的高度，才有这样的效果。

3．跳转到跟控制器时，dock必须重新添加到main控制器，否则会被跟控制器拖动。

- (void)navigationController:(UINavigationController \*)navigationController didShowViewController:(UIViewController \*)viewController animated:(BOOL)animated

{

// 0.取出导航控制器的跟控制器

UIViewController \*root = navigationController.viewControllers[0];

if (root == viewController) {

// 如果是根控制器就还原，如果不还原，dock会被根控制器的视图拖动。而我们想要的是，让dock在底部保持不变。

// 让导航控制器还原

CGRect frame = navigationController.view.frame;

frame.size.height = [UIScreen mainScreen].applicationFrame.size.height - self.dockView.frame.size.height;

navigationController.view.frame = frame;

// 将dock添加到主控制器上

CGRect dockFrame = self.dockView.frame;

dockFrame.origin.y = self.view.frame.size.height - self.dockView.frame.size.height;

self.dockView.frame = dockFrame;

[self.view addSubview:self.dockView];

}

}

## 九．微博详情页知识点

### 1.微博详情页的cell和微博首页的cell非常相似，只是dock不相似，一个dock显示在当前微博cell的下面，一个是显示在被转发微博的下面。所以可以抽出一个父类，自定义一个cell直接继承这个父类，添加一点自有属性，就变成一个微博详情cell了。因为微博详情页需要添加dock到被转发微博的底部，所以我们需要将转发微博的整体框架开发出来，让子类能够拿到它。

父cell跟之前微博cell一模一样，只需要将被转发的整体框架开放出来。

@interface BaseStatusCell : UITableViewCell

{

// 1.被转发的微博整体框架

UIImageView \*\_retweet;

}

* 1. 微博首页cell,添加并设置自己的dock显示的内容

- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString \*)reuseIdentifier

{

self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];

if (self) {

// Initialization code

// 添加操作条

CGFloat y = self.frame.size.height - kStatusDockHeight;

\_dock = [[StatusDock alloc] initWithFrame:CGRectMake(0, y, 0, 0)];

[self.contentView addSubview:\_dock];

}

return self;

}

- (void)setCellFrame:(BaseStatusFrame \*)cellFrame

{

[super setCellFrame:cellFrame];

\_dock.status = cellFrame.status;

}

* 1. 1.2 微博详情，cell添加并设置自己的dock显示的内容

- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString \*)reuseIdentifier{

self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];

if (self) {

// Initialization code

// 1.操作条

RetweetedDock \*dock = [[RetweetedDock alloc] init];

dock.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin | UIViewAutoresizingFlexibleTopMargin;

CGFloat x = \_retweet.frame.size.width - dock.frame.size.width;

CGFloat y = \_retweet.frame.size.height - dock.frame.size.height;

dock.frame = CGRectMake(x, y, 0, 0);

[\_retweet addSubview:dock];

\_dock = dock;

// 2.监听被转发微博的点击

[\_retweet addGestureRecognizer:[[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(showRetweet)]];

}

return self;

}

- (void)showRetweet

{

// 展示被转发的微博

DatailViewController \*detail = [[DatailViewController alloc] init];

detail.status = \_dock.status;

MainViewController \*main = (MainViewController \*)self.window.rootViewController;

UINavigationController \*nav = (UINavigationController \*)main.selectedController;

[nav pushViewController:detail animated:YES];

}

- (void)setCellFrame:(BaseStatusFrame \*)cellFrame

{

[super setCellFrame:cellFrame];

\_dock.status = cellFrame.status.retweetedStatus;

}

### 2.自定义cell还需要抽出一个共同的frame父类。这个父类跟之前微博的frame，只需要修改cell的高度，一开始不用先添加dock的高度，其他计算都一样。然后将cell的高度属性和被转发微博的整体框架的frame开发出来。让子类自己决定自己的dock多高，然后计算出最后cell的高度。

@interface BaseStatusFrame : NSObject

{

CGFloat \_cellHight;

CGRect \_retweetFrame;

}

2.1 微博首页frame,只需要在setStatus方法中设置cell的高度即可

- (void)setStatus:(Status \*)status

{

[super setStatus:status];

\_cellHight += kStatusDockHeight;

}

2.2 微博详情frame，需要判断是否有转发的微博，如果有，给被转发的微博和cell添加高度

- (void)setStatus:(Status \*)status

{

[super setStatus:status];

if (status.retweetedStatus) {

// 如果有配图，才需要添加被转发配图的评论等详细信息的高度

\_retweetFrame.size.height += kRetweetedDockHeight;

\_cellHight += kRetweetedDockHeight;

}

}

### 3.status模型数据都一样，不需要抽父类。

1. 微博详情控制器。
   1. 微博详情控制器视图是分组的tableView，但是在初始化控制器的时候，直接使用init，不要在初始化的时候初始化分组类型的控制器，initWithStyle:UITableViewStyleGrouped，因为我们需要自定义分组，如果在初始化选中分组类型，就是系统自带的分组效果，不是我们想要的。
   2. 我们需要设置第一组的cell为微博详情页。
   3. 第一组cell不能被选中，注意选中和不允许交互的区别，不允许选中不代表不能交互。
   4. 第一组cell的高度为微博详情cell的高度
   5. 第一组的头视图nil
   6. 第一组的头视图高度为0
   7. 第一组cell为一行
   8. 总共有2组
   9. 自定义头部视图，自定义头部视图，对外提供一个接口告诉当前选中的按钮类型，需要根据这个按钮类型显示不同的数据，头部视图中的按钮默认选中评论，在awakeFromNib里实现。

- (void)awakeFromNib

{

self.backgroundColor = kGlobalBg;

// 第一次创建视图，不会通知代理加载默认选中的评论数据，因为awakeFromNib方法的调用，在设置delegate前面。

[self btnClick:\_comment];

}

* 1. 自定义头部视图添加一个代理，当点击按钮，就会调用代理的方法，通知控制器加载对应的数据。注意在第一次创建的头视图的时候，虽然默认选中了评论按钮，但是一开始并不会自动获取评论数据，因为awakeFromNib中调用按钮点击方法时，这时候代理还没设置。因为当视图一加载完毕，就会调用awakeFromNib方法，这时候还没有代理，也就不会加载数据。如果想要加载数据，需要在控制器的viewDidLoad中调用代理的加载数据方法.
  2. 在初始化头视图的时候，需要在viewDidLoad中初始化，不能在tableView设置头部视图的代理方法创建（tableView:viewForHeaderInSection:），因为tableView的cell和头部都是懒加载的，假如一个微博详情cell太长，头部不出现，这时候就不会创建头部，即使在viewDidLoad中调用代理加载数据，也不能现象评论数据。

// 处理header时注意:

/\*

1.header只需要初始化一次，否则在每次刷新的时候，都调用一次这个方法，会产生很多header，不是我们想要的.假如我们点击了转发，获得数据后，又会调用一次reloadData，重新刷新数据，这时候如果没有做处理，又会重新创建一个header对象，作为头视图，而awakefromnib中默认会调用点击评论的方法，这时候就会修改当前选中的按钮类型为评论，并且加载评论数据，但是我们并没有获取评论数据，所以会爆角标越界。

2.如果在viewForHeaderInSection创建，会有一种情况，当cell中微博正文内容比较长，由于tableView的cell和头部header只有出现在视野中才会创建，所以以至于header没有出现的时候，header为空，因此\_detailHeader.currentBtnType也为空，而我们程序中写的1代表显示评论数据，空代表tableView显示转发的内容，而我们默认加载的是评论的，因此会出现cell过长，评论数据不显示的情况。

\*/

* 1. 评论和转发显示的cell相同，同微博详情和微博首页差不多，抽取相同的到父类。就能顺序扩展两个的内容了。
  2. 细节处理，在每次点击按钮加载数据之前，先刷新下表格

#pragma mark header的代理方法

- (void)detailHeader:(DetailHeader \*)header btnClick:(DetailHeaderBtnType)type

{

// 在更新数据之前，需要先刷新下tableView，因为如果网速很慢的时候，我们点击另外一个按钮，网速来不及加载另外一个按钮的数据，当前tableView还是停留在原来按钮的数据，这样用户体验不好，所以考虑到网速慢的原因，我们需要先刷新tableView，将数据更新为另一个按钮的数据，即使网速再慢，也能第一时间看到另一个按钮的数据

[self.tableView reloadData];

// 根据当前的点击的按钮加载不同的数据

if (type == kDetailHeaderBtnTypeComment) {

// 评论

[self loadNewComment];

}else

{

// 转发

[self loadNewRepost];

}

}

4.14 上下拉刷新细节，在每次刷新数据时，需要重新修改头视图显示的评论\转发，头视图的显示跟微博模型有关，而每次加载数据我们都会收到一个totalNumber数据，表示返回的评论或者转发总数。然后我们给微博模型的评论总数或者转发总数重新赋值，即可改变了。因为每次加载数据后，会重新刷新表格。

4.15 上下拉刷新细节2，在每次刷新的时候，会返回一个nextCursor,表示下一页从这个ID开始加载，如果为0，表示下一页没有数据，我们需要将上拉刷新隐藏。

#pragma mark - 加载更多评论数据

- (void)loadMoreComments

{

// 评论

long long lastID = [[\_commentFrames lastObject] baseTest].ID - 1;

[StatusTool commentsWithSinceId:0 maxId:lastID statusId:\_status.ID success:^(NSArray \*comments, int totalNumber, long long nextCursor) {

// nextCursor 表示下一页从哪个ID开始加载，如果等于0，意味着下一页没有数据，没有数据，我们就需要隐藏上拉控件。

\_footer.hidden = (nextCursor == 0);

// 将模型数据转换为frame

NSArray \*newFrames = [self framesWithModels:comments class:[CommentCellFrame class]];

[\_commentFrames addObjectsFromArray:newFrames];

// 更新微博的评论数据

\_status.commentsCount = totalNumber;

[self.tableView reloadData];

[\_footer endRefreshing];

} failure:^(NSError \*error) {

[\_footer endRefreshing];

}];

}

#pragma mark - 加载更多转发数据

- (void)loadMoreReposts

{

// 转发

long long lastID = [[\_repostFrames lastObject] baseTest].ID - 1;

[StatusTool repostsWithSinceId:0 maxId:lastID statusId:\_status.ID success:^(NSArray \*comments, int totalNumber, long long nextCursor) {

// nextCursor 表示下一页从哪个ID开始加载，如果等于0，意味着下一页没有数据，没有数据，我们就需要隐藏上拉控件。

\_footer.hidden = (nextCursor == 0);

// 将模型数据转换为frame

NSArray \*newFrames = [self framesWithModels:comments class:[RepostCellFrame class]];

[\_repostFrames addObjectsFromArray:newFrames];

// 更新微博的评论数据

\_status.repostsCount = totalNumber;

[self.tableView reloadData];

[\_footer endRefreshing];

} failure:^(NSError \*error) {

[\_footer endRefreshing];

}];

}

* 1. 在下拉刷新新的微博数据时，需要给两个属性赋值，才能更新最新界面显示。

#pragma mark 加载当前微博数据

- (void)loadNewStatus

{

[StatusTool statusWithId:\_status.ID success:^(Status \*statues) {

// 修改frame中的status，就能修改显示的内容了。

\_detailFrame.status = statues;

// 因为头部的显示跟当前status有关，所以还需要给\_status赋值

\_status = statues;

[\_header endRefreshing];

} failure:^(NSError \*error) {

[\_header endRefreshing];

}];

}

# 团购项目

## 一．Dock界面知识点

### 1.自适应属性，dock需要设置自适应属性，在屏幕旋转的时候，重新调用setframe修改尺寸。Dock在内部固定宽度。

// 这样设置表示随着父视图的拉伸而拉伸。autoresizingMask的设置是相对父视图的。父视图变化，子类只要设置了这个属性，也会变化，内部会重新调用setframe方法。

// 这里必须设置UIViewAutoresizingFlexibleHeight，虽然刚开始传入的高度为1024，足够高，怎么dock都会全屏显示，但是考虑到dock内部的子控件也需要跟屏幕方向自适应，如果这里这里不设置UIViewAutoresizingFlexibleHeight，那dock的高度永远为1024，这样在宽屏的时候，dock的子视图就不能显示出来了。因为dock的子视图算位置，是根据dock的高度计算的，如果屏幕横着，高度如果设置了自适应就会变成768，这样子控件的位置就会根据768算，如果dock不设置UIViewAutoresizingFlexibleHeight，始终都是依据1024，即使横屏了，也是，这就导致dock的子视图在横屏的时候看不见了。

self.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleRightMargin;

### 2.item父类的抽取共性，发现(更多，定位，和团购)这类的item，有许多共性（分割线，尺寸，image都有两种状态）

2.1 父类dockitem的设计，向外界提供两个属性，正常状态下的图片名称，和不能交互状态下的图片名称。内部在初始化方法中添加分割线，重写setFrame设置固定尺寸。重写setHighlighted，取消高亮时做的事情。

### 3.tabitem,继承父类dockitem，自有特性，就是当选中（disable）时，有背景图片，重写setEnabled，当为不允许交互时，需要隐藏分割线。

moreitem 继承父类dockitem，自有属性，需要设置自有属性，始终在dock的底部，也就意味着需要拉长与顶部的距离，设置autoresizingMask为UIViewAutoresizingFlexibleTopMargin;

### 4.locationitem 需要自己排布内部imageView和title的位置，需要自己设置title。

### 5.将item类创建好了之后，将item添加到dock界面。

## 二．搭建dock与main控制器的联系

### 1.监听tabitem的点击，用于跳转主界面的显示。给dock定义代理，当点击tabitem时，通知代理做事情。

#pragma mark 监听tab点击

- (void)tabClick:(TGTabItem \*)tab

{

// 0.通知代理

if ([\_delegate respondsToSelector:@selector(dock:tabChangeFrom:to:)]) {

[\_delegate dock:self tabChangeFrom:\_selectedItem.tag to:tab.tag];

}

// 1.控制状态

\_selectedItem.enabled = YES;

tab.enabled = NO;

\_selectedItem = tab;

}

### 2.主控制器中，需要初始化一个内容视图，用于解决屏幕适配问题。之后main控制器的子控制器只要将子视图添加到内容视图，并且frame设置为内容视图的bound，就可以解决屏幕适配问题了。

// 2.添加内容view

\_contentView = [[UIView alloc] init];

CGFloat w = self.view.frame.size.width - kDockItemW;

CGFloat h = self.view.frame.size.height;

\_contentView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

\_contentView.frame = CGRectMake(kDockItemW, 0, w, h);

[self.view addSubview:\_contentView];

### 3.在主控制器中viewdidload添加所有子控制器，被导航控制器包装的子控制器。

### 4.实现dock的代理方法。

#pragma mark TGDockDelegate代理方法

- (void)dock:(TGDock \*)dock tabChangeFrom:(int)from to:(int)to

{

// 移除之前的视图

// 取出旧控制器

UIViewController \*old = self.childViewControllers[from];

[old.view removeFromSuperview];

// 添加新视图

UIViewController \*new = self.childViewControllers[to];

// 直接让控制器的尺寸为\_contentView就行了，必须要设置，否则当横屏的时候，控制器的默认宽度为748.

new.view.frame = \_contentView.bounds;

[\_contentView addSubview:new.view];

}

### 自适应属性（autoresizingMask）心得，当一个控件的尺寸，会根据屏幕的变化而变化时，比如窗口的跟控制器，在每次屏幕变化时，都会在即将进入窗口时，重新调用setFrame调整尺寸，这时候跟控制器内部的子视图，就需要设置自适应属性，设置了这个属性，就会在跟控制器改变的尺寸的时候，子视图也会重新调用setFrame调整尺寸。根据情况，设置自定义属性，情况一.如果是一直在父控件的底部，可以设置顶部拉长。情况二。如果想让视图跟父视图一样，随着屏幕变化而变化，比如内容视图，需要设置宽和高拉长。本项目的左边dock，想一直保持在左边，且高度和屏幕一样，需要设置高拉长和距离右边距离拉长。

## 三．更多item点击知识点。

### 1、modal出来的更多控制器，可以设置modal的属性，弹出来是个表单类型。

- (void)moreClick

{

self.enabled = NO;

// 创建more控制器

TGMoreViewController \*more = [[TGMoreViewController alloc] init];

// 设置这个属性是为了当modal出more控制器，让moreitem显示白色的文字。

more.moreItem = self;

// modal出more控制器

// 取出跟控制器

TGNavigationController \*nav = [[TGNavigationController alloc] initWithRootViewController:more];

// 设置modal出来的控制器视图的类型为表单类型。

nav.modalPresentationStyle = UIModalPresentationFormSheet;

// 这里不要传错控制器了，如果传入more控制器，会报错的，如果一个控制器是导航控制器的跟控制器，只能modal导航控制器，modal跟控制器会报错。

[self.window.rootViewController presentViewController:nav animated:YES completion:nil];

}

## 四．定位item点击知识点

### 1.点击定位按钮，弹出一个popover控制器，这个控制器是唯一一个不继承UIViewController的控制器，它继承NSObject。因此他本身没有view，因此他只能基于其他控制器显示。

### 2.自定义一个显示城市的控制器，继承UIViewController，添加两个子视图（UISearchBar,UITableView），注意这里需要设置两个控制器的自适应属性，因为默认控制器在viewdidload的尺寸是屏幕的尺寸，只有进入窗口的时候，才会调整frame。

- (void)addSearchBar

{

UISearchBar \*search = [[UISearchBar alloc] init];

search.frame = CGRectMake(0, 0, self.view.frame.size.width, kSearchH);

// 设置代理

search.delegate = self;

// 设定自适应属性，因为ipad默认控制器的宽度为768，所以我们需要当控制器frame变小时，内部的子控件也需要变小，控制器在viewdidload时宽高如果没有设置，就是默认属性的宽高，当即将进入窗口的时候，会重新设置frame，city控制器在即将进入窗口时，frame会变化，因为他会添加到popover控制器上面，popover的大小我们设置了320\*480，所以city控制器在即将进入窗口的时候frame会变小，所以子控件在那个时候也变小，就必须要设置自适应属性了。

search.autoresizingMask = UIViewAutoresizingFlexibleWidth;

\_search = search;

[self.view addSubview:search];

}

#pragma mark 添加tableView

- (void)addTableView

{

UITableView \*tableView = [[UITableView alloc] init];

CGFloat h = self.view.frame.size.height - kSearchH;

tableView.frame = CGRectMake(0, kSearchH, self.view.frame.size.width, h);

tableView.dataSource = self;

tableView.delegate = self;

// 设置自适应

tableView.autoresizingMask = UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight;

\_tableView = tableView;

[self.view addSubview:tableView];

}

### 3.在按钮点击事件中，初始化popover，还需要监听通知中心的旋转通知，当屏幕改变的时候，popover的箭头需要重新调整位置。

- (void)locationClick

{

// 显示白色

self.enabled = NO;

TGCityListViewController \*cityList = [[TGCityListViewController alloc] init];

// 创建popover控制器,必须基于其他控制器才能显示

\_popover = [[UIPopoverController alloc] initWithContentViewController:cityList];

// 设置popover的尺寸

\_popover.popoverContentSize = CGSizeMake(320, 480);

// 设置popover代理

\_popover.delegate = self;

// 展现出popover,注意这里的尺寸是相对于谁的原点计算的，self.bounds 坐标是相对于自己的原点，如果传入frame，坐标是相对父类的原点，inView应该传入self.superView

[self showPopover];

// 每次旋转的时候，系统都会传一个通知到通知中心，我们就可以监听屏幕的旋转通知。object填空表示任何对象发出这个通知，都拦截

// 在添加通知之前，需要将之前的通知移除，在添加，否则重复添加会有问题的。因为多次添加，意味着会多次调用screenRotate这个方法。

[[NSNotificationCenter defaultCenter] removeObserver:self name:UIDeviceOrientationDidChangeNotification object:nil];

// 只有在展示的时候，才需要监听通知，如果在创建对象的时候就监听，必须考虑到一点，就是每次旋转都会调用监听的方法，需要添加一个判断只有\_popover正在展示，就销毁之前的，并创建新的popover，这个判断不能少，因为只要监听旋转通知没有移除，这个方法在每次旋转都会调用，如果不加这个判断，当调用第一次展现出popover了，当第二次调用这个方法，之前的popover还在展示，这时就会报错，因为之前的popover对象没有指针引用它了，会被自动释放，但是它还在展示中，就会报错。所以必须加这个判断。

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(screenRotate) name:UIDeviceOrientationDidChangeNotification object:nil];

}

#pragma mark 展示popover

- (void)showPopover

{

[\_popover presentPopoverFromRect:self.bounds inView:self permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];

}

- (void)screenRotate

{

// 只有popover展示的时候，才需要关闭之前的。

if (\_popover.popoverVisible) {

// 1.关闭之前的

[\_popover dismissPopoverAnimated:NO];

// 2.0.5秒后创建新的

[self performSelector:@selector(showPopover) withObject:nil afterDelay:0.5];

}

}

### 4．注意：监听通知中心中系统自带的通知时，需要在popover不显示的时候，移除这个通知，还有当监听对象被销毁的时候也需要移除。

#pragma mark popover的代理

- (void)popoverControllerDidDismissPopover:(UIPopoverController \*)popoverController

{

// 让按钮能够点击

self.enabled = YES;

// 移除旋转监听，因为只有popover出现的时候，才需要监听旋转的。这里不需要移除城市值改变通知。因为当popover再次出现的时候，我又要监听城市值改变，而监听城市值改变只在item初始化（init）的时候，添加了一次，如果这里移除了，下次popover出现的时候，就不能在监听了，因为下次popover出现，item不会在初始化。而旋转通知不同，旋转通知在每次点击按钮的时候，都会添加一次监听。

[[NSNotificationCenter defaultCenter] removeObserver:self name:UIDeviceOrientationDidChangeNotification object:nil];

}

#pragma mark 一旦监听了通知，需要在对象消耗后，移除监听通知，否则会报野指针错误，因为当对象销毁了，但是监听通知没有移除，当屏幕再次旋转，通知中心就会通知监听对象，调用对应的方法，这时候监听对应已经销毁，就会报野指针错误。

- (void)dealloc

{

// 将监听对象移除

[[NSNotificationCenter defaultCenter]removeObserver:self];

}

## 五．城市控制器知识点

### 1.完善searchbar，监听searchbar的代理方法

#pragma mark UISearchBar代理方法

// 开始编辑的时候，就会调用这个方法，即键盘出现的时候,键盘出现的时候，我们需要出现取消按钮

- (void)searchBarTextDidBeginEditing:(UISearchBar \*)searchBar

{

// 添加取消按钮

[searchBar setShowsCancelButton:YES];

// 添加蒙版

UIView \*cover = [[UIView alloc] init];

// 设置背景颜色

cover.backgroundColor = [UIColor blackColor];

cover.frame = \_tableView.frame;

// 动画添加蒙版

cover.alpha = 0;

[UIView animateWithDuration:0.5 animations:^{

cover.alpha = 0.7;

}];

\_cover = cover;

// 添加手势

[\_cover addGestureRecognizer:[[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(coverClick)]];

[self.view addSubview:cover];

}

#pragma mark 手势监听

- (void)coverClick

{

// 隐藏取消按钮

[\_search setShowsCancelButton:NO];

// 清除蒙版

[UIView animateWithDuration:0.3 animations:^{

\_cover.alpha = 0.0;

} completion:^(BOOL finished) {

[\_cover removeFromSuperview];

}];

// 退出键盘

[\_search endEditing:YES];

}

// 点击取消按钮的时候

- (void)searchBarCancelButtonClicked:(UISearchBar \*)searchBar

{

[self coverClick];

}

// 当键盘取消的时候，即停止编辑的时候，需要退出键盘

- (void)searchBarTextDidEndEditing:(UISearchBar \*)searchBar

{

[self coverClick];

}

### 2.完善tableView，由于需要获取元数据，我们可以定义一个工具类，管理我们需要的所有元数据，以后我们需要什么数据，就从工具类里面拿，由于工具类需要用到成员属性，所以我们设置为单例。

### 2.1从字典里取值给模型赋值时，我们需要用到运行时机制，这个效率高，且不用我们一个一个写。

### 2.2 向外界提供一个城市组接口，这个城市组先添加最热门城市，在添加所有城市组，最后将最近访问城市插入到第0个角标中，添加最近访问城市时，需要判断最近访问城市是否数据，如果没有，就不需要添加.

- (id)init

{

self = [super init];

if (self) {

// 加载plist文件

NSArray \*array = [NSArray arrayWithContentsOfFile:[[NSBundle mainBundle]pathForResource:@"Cities.plist" ofType:nil]];

// 初始化字典，存放所有的城市

\_totalCities = [NSMutableDictionary dictionary];

NSMutableArray \*tempArray = [NSMutableArray array];

// 添加热门城市组

TGCitySection \*hotSection = [[TGCitySection alloc] init];

hotSection.name = @"热门城市";

[tempArray addObject:hotSection];

hotSection.cities = [NSMutableArray array];

// 初始化所有城市组数组

for (NSDictionary \*dict in array) {

TGCitySection \*citySection = [[TGCitySection alloc] init];

[citySection setValues:dict];

// 判断城市是否是热门城市

for (TGCity \*city in citySection.cities) {

if (city.hot) {

// 如果是热门城市，就添加到热门城市组中

[hotSection.cities addObject:city];

}

// 将城市存储到字典

[\_totalCities setObject:city forKey:city.name];

}

[tempArray addObject:citySection];

}

\_totalCitySections = tempArray;

// 添加最近访问城市组

\_visitedSection = [[TGCitySection alloc] init];

\_visitedSection.name = @"最近访问";

\_visitedSection.cities = [NSMutableArray array];

// 从沙盒中读取之前访问过的城市名称

\_visitedCityNames = [NSKeyedUnarchiver unarchiveObjectWithFile:kFilePath];

if (\_visitedCityNames == nil) {

\_visitedCityNames = [NSMutableArray array];

}

// 怎样通过名称，拿到对应的城市对象了，可以用一个字典，记录所有城市对象。

// 遍历城市名称数组

for (NSString \*name in \_visitedCityNames) {

TGCity \*city = \_totalCities[name];

[\_visitedSection.cities addObject:city];

}

// 将最近访问城市组插入数组最前面

[tempArray insertObject:\_visitedSection atIndex:0];

}

return self;

}

### 2.3 当城市控制器的cell被点击的时候，我们需要将点击的城市记录下来，放进最近访问的城市中，并且需要将这个最近访问的城市归档，因此工具类向外界提供一个当前城市接口。在点击城市cell的时候，记录下来，并归档。重写setcurrentCity方法。最后在方法中，要记得向通知中心发送通知，告诉需要监听这个通知的对象做出一些事情。

- (void)setCurrentCity:(TGCity \*)currentCity

{

\_currentCity = currentCity;

#warning 这样设计的好处，为什么只归档城市名称了，因为如果归档整个城市对象，就太冗余了，我们归档只需要归档有用的数据。

// 移除之前的城市名，避免重复添加

[\_visitedCityNames removeObject:currentCity.name];

// 将点击的城市的名称加入最近访问城市的name数组最前面

[\_visitedCityNames insertObject:currentCity.name atIndex:0];

// 将城市名称归档

[NSKeyedArchiver archiveRootObject:\_visitedCityNames toFile:kFilePath];

// 发现当再次点击按钮时，刚刚点击的城市，没有进入到最近访问城市，是因为只有在每次调用init的时候，才会解档，将刚刚点击的城市添加到数组，为了解决这个问题，我们可以自己添加城市到最近访问城市数组中。

// 移除之前的城市,避免重复添加

[\_visitedSection.cities removeObject:currentCity];

// 将新的城市放到\_visitedSection的最前面

[\_visitedSection.cities insertObject:currentCity atIndex:0];

// 每次点击城市，需要发送通知，通知locationitem和团购控制器刷新界面显示。

// 发送通知，object传nil表示匿名发送通知.

[[NSNotificationCenter defaultCenter] postNotificationName:kCityChangeNote object:nil];

}

### 2.4 注意点，归档只归档有用数据，否则会让数据冗余。

## 六．搜索结果控制器知识点。

### 1.当在searchBar输入文字时，我们需要监听searchBar的一个方法，来告诉用户搜索结果。

// 监听UISearchBar的输入,一旦输入的文字，我们都需要做出一些响应。

- (void)searchBar:(UISearchBar \*)searchBar textDidChange:(NSString \*)searchText

{

if (searchText.length == 0) {

// 隐藏搜索界面

[\_searchResultController.view removeFromSuperview];

}else{

// 显示搜索界面

if (\_searchResultController == nil) {

// 初始化搜索结果控制器

\_searchResultController = [[TGSearchResultController alloc] init];

\_searchResultController.view.frame = \_cover.frame;

\_searchResultController.view.autoresizingMask = \_cover.autoresizingMask;

// 一个控制器的视图是另外一个控制器的子视图，则这个控制器一定也要成为另外一个控制器的子控制器

[self addChildViewController:\_searchResultController];

}

\_searchResultController.searchText = searchText;

[self.view addSubview:\_searchResultController.view];

}

}

### 2.搜索结果控制器向外界提供一个接口，用来接收搜索框输入的文字，然后搜索结果控制器根据搜索框输入的内容刷新控制器。判断是否满足搜索条件，需要用到第三方框架，PinYin4Objc

#pragma mark 重写setSearchText方法，根据搜索文本，刷新搜索结果控制器的显示

- (void)setSearchText:(NSString \*)searchText

{

\_searchText = searchText;

// 清除上一次的搜索结果,否则会造出数据冗余。

[\_results removeAllObjects];

// 取出所有城市

NSDictionary \*dict = [TGMetaDataTool sharedTGMetaDataTool].totalCities;

// 筛选城市

HanyuPinyinOutputFormat \*fmt = [[HanyuPinyinOutputFormat alloc] init];

// 将文字转换成大写拼音

fmt.caseType = CaseTypeUppercase;

// 不需要声调

fmt.toneType = ToneTypeWithoutTone;

// v按v来显示

fmt.vCharType = VCharTypeWithV;

// 遍历字典

[dict enumerateKeysAndObjectsUsingBlock:^(NSString \*key, TGCity \*obj, BOOL \*stop) {

// 城市名称转换成拼音,第三个参数是分隔符，比如石家庄转换成 SHI#JIA#ZHUANG

NSString \*pinyin = [PinyinHelper toHanyuPinyinStringWithNSString:key withHanyuPinyinOutputFormat:fmt withNSString:@"#"];

// 拼音首字母

NSArray \*words = [pinyin componentsSeparatedByString:@"#"];

NSMutableString \*pinyinHeader = [NSMutableString string];

for (NSString \*word in words) {

// substringToIndex 截取从0到哪的索引 这里是从0 到 1.即第一个字母

[pinyinHeader appendString:[word substringToIndex:1]];

}

pinyin = [pinyin stringByReplacingOccurrencesOfString:@"#" withString:@""];

// 判断所有城市名称中是否包含了搜索文本

/\*

判断城市名中包含了搜索条件

判断拼音中包含了搜索条件

判断拼音首字母中包含了搜索条件

注意：搜索文本必须转换为大写，跟中文转换拼音的格式一致，这里转换拼音的都是转换成大写的，这样就能忽略大小写了，

\*/

if (([key rangeOfString:searchText].length != 0) ||

([pinyin rangeOfString:searchText.uppercaseString].length != 0) ||

([pinyinHeader rangeOfString:searchText.uppercaseString].length != 0)) {

// 将满足条件的对象添加到搜索结果

[\_results addObject:obj];

}

}];

// 刷新表格

[self.tableView reloadData];

}

## 七．搭建团购界面

### 1.导航栏的搭建。

1.1左边是一个自定义UIView上面有3个按钮，右边是一个搜索框。

### 2.自定义顶部View

2.0 顶部按钮内部设置宽高

2.1首先观察，顶部3个按钮都是相同的特征（右边有箭头，分割线，点击背景视图会改变）,因此我们自定义一个top按钮，向外界提供一个接口，传入一个字符串进来，就设置按钮文字。自定义按钮内部固定高度和宽度。

2.2 给自定义UIView添加自定义top按钮，并且监听他们的点击事件。

2.3 按钮三步曲，1.取消上一次按钮的选中状态2.当前按钮状态选中3。将当前按钮赋值。

2.4 底部视图应该添加到团购控制器上，且大小和团购控制器一样大，因此顶部视图需要提供一个接口，将控制器视图传进来。

### 3.自定义底部View（注意添加自动拉伸属性）

3.0 观察可知道，底部视图有3种类型（分类，商区，排序），每种类型的底部菜单按钮和底部子标题按钮都是不一样的，需要根据顶部按钮的点击判断，因此我们可以抽取他们的相同之处，抽取父类，然后在创建3个子类继承这个父底部视图，3个子底部视图，分别在各自的类里设置自己的内容。

3.1 首先分析底部视图的整体框架，底部视图分为3部分（底部菜单，底部子标题菜单，蒙版）底部菜单是放在UIScrollView中，底部又是一个自定义view，里面有自己的业务逻辑，蒙版就是一个半透明的UIView。

3.2 先添加蒙版，铺盖整个底部视图，这样就不需要管蒙版的高度了，因为之后添加的视图都会在蒙版上边。

3.3 分析底部菜单和底部子标题的结构，当点击底部菜单时，显示子标题菜单。当点击顶部菜单或者蒙版的时候，底部菜单和底部子标题同时动画向上移走，这点我们就能判断底部菜单和底部子标题同时存在一个UIView中。因此我们在底部视图中创建一个内容视图容器，用来装底部菜单和底部子菜单。

3.4 父底部视图的搭建，先不管底部子菜单，先将底部菜单完成，创建一个UIScrollView将它添加到内容视图中

- (id)initWithFrame:(CGRect)frame

{

self = [super initWithFrame:frame];

if (self) {

// Initialization code

// 底部视图需要设置自适应属性

self.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

// 添加蒙版,为什么先添加蒙版了，因为我们直接设置蒙版为整个底部的尺寸，就不需要之后在计算蒙版的尺寸了。

\_cover = [[UIView alloc] init];

\_cover.alpha = 0.7;

\_cover.frame = self.bounds;

[\_cover addGestureRecognizer:[[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(hide)]];

// 需要设置自适应属性，因为父控件的尺寸可能会改变

\_cover.autoresizingMask = self.autoresizingMask;

\_cover.backgroundColor = [UIColor blackColor];

[self addSubview:\_cover];

// 2.添加内容view

// 为什么要弄一个内容view，因为我们观察，底部scrollView和子标题动画是一致的，所以我们弄一个内容视图，将底部scrollView和子标题放在同一个view中，然后操作一个view就能达到同时动画的效果，如果分开管理动画，达不到我们这样的效果。

\_contentView = [[UIView alloc] init];

\_contentView.frame = CGRectMake(0, 0, self.frame.size.width, kBottomMenuItemH);

\_contentView.autoresizingMask = UIViewAutoresizingFlexibleWidth;

[self addSubview:\_contentView];

// 添加scrollView

UIScrollView \*scrollView = [[UIScrollView alloc] init];

scrollView.showsHorizontalScrollIndicator = NO;

scrollView.autoresizingMask = UIViewAutoresizingFlexibleWidth;

scrollView.frame = CGRectMake(0, 0, self.frame.size.width, kBottomMenuItemH);

scrollView.backgroundColor = [UIColor whiteColor];

[\_contentView addSubview:scrollView];

\_scrollView = scrollView;

}

return self;

}

3.5 父底部菜单按钮的添加，底部菜单按钮是根据顶部按钮来显示对应类型的按钮的，有3种类型的按钮，他们也有自己的共性（右边分割线，被选中的背景，宽高一致）因此我们又能抽取一个底部菜单按钮父类。

if (self) {

// Initialization code

// 1.右边的分割线

UIImage \*img = [UIImage imageNamed:@"separator\_filter\_item.png"];

UIImageView \*divider = [[UIImageView alloc] initWithImage:img];

divider.bounds = CGRectMake(0, 0, 2, kBottomMenuItemH \* 0.7);

divider.center = CGPointMake(kBottomMenuItemW, kBottomMenuItemH \* 0.5);

[self addSubview:divider];

// 2.文字颜色

[self setTitleColor:[UIColor blackColor] forState:UIControlStateNormal];

// 3.设置被选中时的背景

[self setBackgroundImage:[UIImage resizedImage:@"bg\_filter\_toggle\_hl.png"] forState:UIControlStateSelected];

}

3.5 搭建数据模型，我们之前创建了一个工具类用来管理数据模型，因此这里也是一样的，同样将数据交给工具类管理。这样的设计的好处是，数据统一管理，因为一个项目中有很多数据模型，这时候我们就需要一个工具类来管理我们的数据模型。

之前已经加载了所有城市数据，现在需要加载分类数据，排序数据。并且向外界提供数据接口。

#pragma mark 初始化排序数据

- (void)loadOrderData

{

NSArray \*array = [NSArray arrayWithContentsOfFile:[[NSBundle mainBundle] pathForResource:@"Orders.plist" ofType:nil]];

int count = array.count;

NSMutableArray \*temp = [NSMutableArray array];

for (int i = 0; i<count; i++){

TGOrder \*o = [[TGOrder alloc] init];

o.name = array[i];

o.index = i;

[temp addObject:o];

}

\_totalOrders = temp;

}

#pragma mark 初始化分类数据

- (void)loadCategoryData

{

NSArray \*array = [NSArray arrayWithContentsOfFile:[[NSBundle mainBundle] pathForResource:@"Categories.plist" ofType:nil]];

NSMutableArray \*temp = [NSMutableArray array];

for (NSDictionary \*dict in array) {

TGCategory \*c = [[TGCategory alloc] init];

[c setValues:dict];

[temp addObject:c];

}

\_totalCategories = temp;

}

3.6 分类底部视图的搭建

3.6.1 首先我们需要获取分类的总数据，然后根据分类模型，来添加分类底部视图中UIScrollView的分类按钮，每个分类按钮需要设置一个接口，将分类模型传进去，然后设置按钮的文字。

3.6.2 自定义分类按钮

- (id)initWithFrame:(CGRect)frame

{

self = [super initWithFrame:frame];

if (self) {

// Initialization code

// 2.文字

self.titleLabel.textAlignment = NSTextAlignmentCenter;

// 3.图片

self.imageView.contentMode = UIViewContentModeCenter;

}

return self;

}

- (void)setCategory:(TGCategory \*)category

{

\_category = category;

// 1.图标

[self setImage:[UIImage imageNamed:category.icon] forState:UIControlStateNormal];

// 2.标题

[self setTitle:category.name forState:UIControlStateNormal];

}

#pragma mark 设置按钮标题的frame

- (CGRect)titleRectForContentRect:(CGRect)contentRect {

CGFloat titleHeight = contentRect.size.height \* kTitleRatio;

CGFloat titleY = contentRect.size.height - titleHeight;

return CGRectMake(0, titleY, contentRect.size.width, titleHeight);

}

#pragma mark 设置按钮图片的frame

- (CGRect)imageRectForContentRect:(CGRect)contentRect {

return CGRectMake(0, 0, contentRect.size.width, contentRect.size.height \* (1 - kTitleRatio));

}

3.6.3 分类底部视图

- (id)initWithFrame:(CGRect)frame

{

self = [super initWithFrame:frame];

if (self) {

// Initialization code

NSArray \*categories = [TGMetaDataTool sharedTGMetaDataTool].totalCategories;

// 1.往scrollView里面添加内容

int count = categories.count;

for (int i = 0; i<count; i++) {

// 创建item

TGCategoryMenuItem \*item = [[TGCategoryMenuItem alloc] init];

item.category = categories[i];

[item addTarget:self action:@selector(itemClick:) forControlEvents:UIControlEventTouchUpInside];

item.frame = CGRectMake(i \* kBottomMenuItemW, 0, 0, 0);

[\_scrollView addSubview:item];

}

\_scrollView.contentSize = CGSizeMake(count \* kBottomMenuItemW, 0);

}

return self;

}

3.7 商区底部视图的搭建

3.7.1 自定义商区按钮

- (void)setDistrict:(TGDistrict \*)district

{

\_district = district;

[self setTitle:district.name forState:UIControlStateNormal];

}

3.7.2 商区底部视图，需要拿到当前点击的城市数据，构建视图

- (id)initWithFrame:(CGRect)frame

{

self = [super initWithFrame:frame];

if (self) {

// 1.往scrollView中添加内容

// 获取当前选中的城市

TGCity \*city = [TGMetaDataTool sharedTGMetaDataTool].currentCity;

NSInteger count = city.districts.count;

for (int i = 0; i<count; i++) {

// 创建区域item

TGDistrictMenuItem \*item = [[TGDistrictMenuItem alloc] init];

item.district = city.districts[i];

[item addTarget:self action:@selector(itemClick:) forControlEvents:UIControlEventTouchUpInside];

item.frame = CGRectMake(i \* kBottomMenuItemW, 0, 0, 0);

[\_scrollView addSubview:item];

}

\_scrollView.contentSize = CGSizeMake(count \* kBottomMenuItemW, 0);

}

return self;

}

3.8 排序底部视图的搭建

3.8.1 自定义排序按钮

- (void)setOrder:(TGOrder \*)order

{

\_order = order;

[self setTitle:order.name forState:UIControlStateNormal];

}

3.8.2 排序底部视图，拿到排序的数据

- (id)initWithFrame:(CGRect)frame

{

self = [super initWithFrame:frame];

if (self) {

// 1.往UIScrollView添加内容

NSArray \*orders = [TGMetaDataTool sharedTGMetaDataTool].totalOrders;

int count = orders.count;

for (int i = 0; i<count; i++) {

// 创建排序item

TGOrderMenuItem \*item = [[TGOrderMenuItem alloc] init];

item.order = orders[i];

[item addTarget:self action:@selector(itemClick:) forControlEvents:UIControlEventTouchUpInside];

item.frame = CGRectMake(i \* kBottomMenuItemW, 0, 0, 0);

[\_scrollView addSubview:item];

}

\_scrollView.contentSize = CGSizeMake(count \* kBottomMenuItemW, 0);

}

return self;

}

### 4.菜单动画

4.1底部视图搭建好之后，接着就是现实动画了，如果顶部菜单点击的按钮不相同，显示底部菜单按钮。如果顶部菜单点击的按钮相同，隐藏底部按钮菜单按钮。

4.2 顶部按钮点击事件

#pragma mark 监听按钮的点击

- (void)itemClick:(TGDealTopMenuItem \*)item

{

// 取消上次按钮的选中状态

\_selectedItem.selected = NO;

if (\_selectedItem == item) {

// 点击相同按钮，即取消按钮的选中效果，当前就没有选中按钮了。

\_selectedItem = nil;

// 移除底部视图

[self hideBottomMenu];

}else{

item.selected = YES;

\_selectedItem = item;

[self showBottomMenu:item];

}

}

4.3 显示底部视图，是根据当前按钮的tag对应哪个按钮，就添加哪个类型按钮的底部视图。为了让每次团购控制器只有一个底部视图，避免重复添加底部视图，而造成蒙版越来越深，我们可以用一个父类的底部视图，记录这当前正在显示的底部视图，然后在下次点击的时候，先移除当前正在显示的底部视图，然后在添加新的底部视图。而且每个类型的底部视图没必要每次点击都创建，因为每个类型出现的底部视图都一样，如果每次点击都创建，这样消耗性能 ，我们可以把每个类型的底部视图作为成员变量，保存起来。

#pragma mark 显示底部视图

- (void)showBottomMenu:(TGDealTopMenuItem \*)item

{

// 是否需要执行动画（没有正在展示的菜单时，就需要执行动画）

BOOL animted = \_showingMenu == nil;

// 为避免重复添加显示的底部菜单，需要先移除之前的菜单

[\_showingMenu removeFromSuperview];

// 添加底部视图

if (item.tag == 0) {

// 分类

if (\_cMenu == nil) {

\_cMenu = [[TGCategoryMenuView alloc] init];

}

\_showingMenu = \_cMenu;

}else if (item.tag == 1){

// 商区

if (\_dMenu == nil) {

\_dMenu = [[TGDistrictMenu alloc] init];

}

\_showingMenu = \_dMenu;

}else{

// 排序

if (\_oMenu == nil) {

\_oMenu = [[TGOrderMenu alloc] init];

}

\_showingMenu = \_oMenu;

}

// 将底部视图添加到控制器视图上。

[\_contentView addSubview:\_showingMenu];

// 设置frame

\_showingMenu.frame = \_contentView.bounds;

// 解决block中修改自己成员属性产生循环retain的情况，只要加这个属性就好了。

\_\_unsafe\_unretained TGDealTopMenuView \*menu = self;

\_showingMenu.hideBlock = ^{

// 注意在block中需要修改自己的成员属性时，会发送循环retain的情况。

// 取消当前按钮选中状态 menu ->\_selectedItem 表示访问成员属性\_selectedItem，将\_selectedItem的selected设置为NO，这样就能取消循环retain事件了。menu是\_\_unsafe\_unretained的策略。

menu -> \_selectedItem.selected = NO;

// 清空正在显示的底部视图

menu->\_showingMenu = nil;

// 清空当前选中按钮

menu->\_selectedItem = nil;

};

// 不管点击了哪个菜单，只要当第一次出现底部菜单，就需要动画。所以我们统一在下面执行动画，在最开始的时候判断下是否是第一次，这样就能省了在每个判断里，都执行一次动画。巧妙的设计。

if (animted) {

// 由于动画，需要拿到底部视图内部的控件执行动画，所以我们可以这样设计，在底部视图中提供一个动画的接口，我们直接调用就好了。

// 执行动画

[\_showingMenu show];

}

}

4.4 每次第一次展现底部视图，和隐藏底部视图的时候都需要动画，我们可以当底部视图内部提供一个显示动画接口和隐藏动画接口，因为只有底部视图才最清楚自己有多少控件，哪些控件需要动画，因此我们采取提供接口的方法。

4.4.1在底部视图中提供的动画接口

#pragma mark 显示

- (void)show

{

\_contentView.transform = CGAffineTransformMakeTranslation(0, -\_contentView.frame.size.height);

\_contentView.alpha = 0;

\_cover.alpha = 0;

[UIView animateWithDuration:kDefaultAnimDuration animations:^{

// 1.scrollView从上面 -> 下面

\_contentView.transform = CGAffineTransformIdentity;

\_contentView.alpha = 1;

// 2.遮盖（0 -> 0.4）

\_cover.alpha = kCoverAlpha;

}];

}

#pragma mark 注意隐藏的时候，需要让顶部视图的按钮取消选中状态，并将当前显示的底部视图设置为nil，当前选中的按钮也设置为nil

- (void)hide

{

if (\_hideBlock) {

\_hideBlock();

}

[UIView animateWithDuration:kDefaultAnimDuration animations:^{

// 1.scrollView从下面 -> 上面

\_contentView.transform = CGAffineTransformMakeTranslation(0, -\_contentView.frame.size.height);

\_contentView.alpha = 0;

// 2.遮盖（0.4 -> 0）

\_cover.alpha = 0;

} completion:^(BOOL finished) {

// 从父控件中移除

[self removeFromSuperview];

// 恢复属性,注意动画完成之后，必须复原，因为show方法，只有在底部视图第一次出现的时候，才会调用，当底部视图已经出现了，就不会在调用了，而如果点击了分类按钮，然后在点击排序按钮，然后点击排序按钮隐藏底部视图，这时候排序按钮底部的transfrom是-\_contentView.frame.size.height的高度，而当我们点击分类按钮，这时候是第一次出现，所以会调用show，但我们再次点击排序按钮时，就不会调用show了，而这时候排序按钮的transfrom是\_contentView.frame.size.height，也就不会出现在我们视野，因为它的位置不对。所以我们必须在每次hide之后，将transfrom赋值。

\_contentView.transform = CGAffineTransformIdentity;

\_contentView.alpha = 1;

\_cover.alpha = kCoverAlpha;

}];

}

### 5.展现子标题

5.1当点击底部菜单，出现自定义子标题视图。因为子标题视图每次出现的动画都是不一样的，根据各个底部菜单按钮中子标题个数决定，如果从标题多的底部菜单按钮过度到标题少的底部菜单按钮，动画效果是高度慢慢减少，因此我们能断定，每次出现的子标题都是同一个视图，也就意味着所有的子标题都可以共用。

5.2 分析完子标题始终都是一个，不会被销毁后，我们还知道每个底部菜单按钮，做的事情都是一样，显示子标题。因此我们可以统一在父类中实现，监听按钮点击的方法，子类只要不实现，就会去父类找。这种思想，要多用。在父类中，实现按钮点击的方法。

#pragma mark 按钮点击事件

// 为什么要在父类中实现点击事件了，因为我们观察子类（分类，商区，排序）他们的点击事件都是一样的，所以我们统一放在父类实现，就不要一个一个子类实现了，只要子类没有这个按钮监听方法，就会到父类找，因此我们利用这个思想做一个共性的事情

- (void)itemClick:(TGDealBottomMenuItem \*)item

{

// 1.控制item的状态

\_selectedItem.selected = NO;

item.selected = YES;

\_selectedItem = item;

// 2.查看是否有子分类，由于各个子类的item的显示都跟它对应的模型有关，所以只要能拿到模型，我们就能判断模型中是否有子分类。而这里是在底部父类方法中实现点击事件，传入的item也只能是父类item，因此只有让item父类提供一个获取titles的接口，然后分别在子类item中实现，子类item的titles方法，返回自己模型的子分类总数。

if (item.titles.count) { // 显示所有的子标题

[self showSubtitlesView:item];

} else { // 隐藏所有的子标题

[self hideSubtitlesView:item];

}

}

#pragma mark 隐藏子标题

- (void)hideSubtitlesView:(TGDealBottomMenuItem \*)item

{

// 隐藏子标题

[\_subtitlesView hide];

// 更改\_contentView的尺寸

CGRect f = \_contentView.frame;

f.size.height = kBottomMenuItemH;

\_contentView.frame = f;

// 将没有子标题的分类记录下来。

[self settingTitlesWithString:[item titleForState:UIControlStateNormal]];

}

#pragma mark 显示子标题

- (void)showSubtitlesView:(TGDealBottomMenuItem \*)item

{

// 将这个改变子标题的代码包起来，每次更换子标题的时候都会有动画。

[UIView beginAnimations:nil context:nil];

[UIView setAnimationDuration:kDefaultAnimDuration];

if (\_subtitlesView == nil) {

\_subtitlesView = [[TGSubtitlesView alloc] init];

// 因为block只需要创建一次，因此放在这。

[self settingSubtitlesView];

}

#warning ...

// 注意\_subtitlesView每次高度都不是0，因为\_subtitlesView并不是每次都创建，这里的情况是特殊的，\_subtitlesView虽然移除父控件，但是他还是被强引用着，不会释放，这也就意味着，我们这里能拿到\_subtitlesView之前的高度。这样我们就能实现跳转按钮时，\_subtitlesView的尺寸变化时，有动画效果，且是相对之前的尺寸的动画。

\_subtitlesView.frame = CGRectMake(0, kBottomMenuItemH, self.frame.size.width, \_subtitlesView.frame.size.height);

// 将底部菜单的名称（主标题）传入进去，注意这个一定要放在子标题内容的前面传进去，因为子标题内容中，需要用到当前住标题，如果放在后面，子标题会拿到之前的主标题，因此不是我们想要的

\_subtitlesView.mainTitle = [item titleForState:UIControlStateNormal];

MyLog(@"%@", [item titleForState:UIControlStateNormal]);

// 将子标题内容传进去

\_subtitlesView.titles = item.titles;

// 这句一定要放在插入\_contentView之前，因为这里是拿父视图判断的。我们需要判断的是在插入\_contentView之前，有没有父视图，如果没有，就执行动画。

if (\_subtitlesView.superview == nil) { // 没有父控件

// [\_subtitlesView show];

}

// 必须将子标题插入到scrollView的后面，要不在执行动画的时候，子标题会把scrollView覆盖，我们需要他从scrollView的里面专出来。

[\_contentView insertSubview:\_subtitlesView belowSubview:\_scrollView];

// 这里计算\_contentView的尺寸必须放在子标题添加到\_contentView后面，因为子标题在第一次添加到\_contentView，才会调用layoutSubviews调用\_subtitlesView的尺寸，因此只有添加完\_subtitlesView，才能获取\_subtitlesView的尺寸。

// 重新计算\_contentView的步骤必不可少，一。如果不设置，当点击蒙版，动画效果移动的距离只有一个scorllView的高度。

// 子标题中的按钮不可点击，因为\_contentView的尺寸实际上没有那么多。

CGRect f = \_contentView.frame;

f.size.height = kBottomMenuItemH + \_subtitlesView.frame.size.height;

\_contentView.frame = f;

[UIView commitAnimations];

}

5.3 子标题里面的显示，根据底部菜单按钮内部的模型中有多少子分类判断，由于我们子标题是在父类底部视图按钮点击方法中创建的，而每个子类底部视图中按钮里面的模型的子分类都不一样，因此只有子类底部按钮才最清楚自己的子分类标题的内容，父类底部按钮视图中是不知道子标题内部的显示内容，那有什么方法能在父类中清楚知道各个子类中的信息了，我们可以给父类底部按钮定义一个方法，但是方法的具体实现交给子类底部按钮。就能获取到每个子类按钮的详细子分类信息了。我们当初点击顶部按钮时，创建的底部视图按钮，都是各个分类底部视图按钮，因此可以采取这种方法，获取子类的详细信息。只要子类实现了方法，就不会调用父类的方法了。

父类item：

- (NSArray \*)titles

{

return nil;

}

分类item

- (NSArray \*)titles

{

return \_category.subcategories;

}

商区item

- (NSArray \*)titles

{

return \_district.neighborhoods;

}

排序item，不需要实现，因为他们按钮内部的模型没有子分类。

5.4 有了数据，我们就能自定义子标题。子标题需要给外界提供一个包含所有子标题内容的数组接口，然后重写这个方法给子标题按钮设置文本内容。由于子标题视图是一直存在的，所以我们并不需要每次都创建新的按钮，我们可以将创建的按钮保存起来，循环利用。

#pragma mark 设置子标题内的所有button，这里需要做一个优化，

- (void)setTitles:(NSArray \*)titles

{

NSMutableArray \*array = [NSMutableArray array];

[array addObject:kAll];

[array addObjectsFromArray:titles];

\_titles = array;

int count = \_titles.count;

// 设置按钮的文字

for (int i = 0; i<count; i++) {

// 1.取出i位置对应的按钮

UIButton \*btn = nil;

if (i >= self.subviews.count) { // 按钮个数不够

btn = [UIButton buttonWithType:UIButtonTypeCustom];

[btn setTitleColor:[UIColor blackColor] forState:UIControlStateNormal];

// 给按钮监听点击事件

[btn addTarget:self action:@selector(titleClick:) forControlEvents:UIControlEventTouchUpInside];

[btn setBackgroundImage:[UIImage resizedImage:@"slider\_filter\_bg\_active.png"] forState:UIControlStateSelected];

[self addSubview:btn];

} else {

btn = self.subviews[i];

}

// 2.设置按钮文字

btn.hidden = NO;

[btn setTitle:\_titles[i] forState:UIControlStateNormal];

// 根据按钮文字来决定要不要选中

if (\_getTitleBlcok) {

NSString \*current = \_getTitleBlcok();

if ([current isEqualToString:\_mainTitle] && i == 0) {

// 选中主标题并且是第一个按钮

btn.selected = YES;

\_selectedItem = btn;

}else{

btn.selected = [\_titles[i] isEqualToString:current];

if (btn.selected) {

\_selectedItem = btn;

}

}

} else {

btn.selected = NO;

}

}

// 隐藏后面多余的按钮

for (int i = count; i<self.subviews.count; i++) {

UIButton \*btn = self.subviews[i];

btn.hidden = YES;

}

// 这里必须强制调用一下layoutSubviews，调整下宽高，因为每次都需要根据文字算出尺寸，这样就能达到每次点击子标题按钮而达到子标题视图，变产生动画。layoutSubviews这个方法，不会随意调用的，只要控件的尺寸不变，就不会调用。而第一次点击一个底部菜单按钮时，就会确认底部子标题的尺寸，由于底部子标题不会销毁，因此在下次点击其他菜单按钮时，传给底部子标题的高度还是之前的，因此即使子标题不一样，但是高度还是一样的，并没有重新计算，所以这里我们需要强制调整高度。

[self layoutSubviews];

}

5.5 设置按钮的位置，每当父视图的尺寸调整的时候，都会调用layoutSubviews方法，因此我们可以在这个方法中设置按钮的位置，因为子标题会随着数组的变化，而导致尺寸变化。

// 第一次加入的时候和控件本身的宽高发生改变的时候就会调用这个方法，因此我们可以在这个方法中调整每次旋转后子视图的位置。

- (void)layoutSubviews

{

// 一定要调用super

[super layoutSubviews];

int columns = self.frame.size.width / kTitleW;

for (int i = 0; i<\_titles.count; i++) {

UIButton \*btn = self.subviews[i];

// 设置位置

CGFloat x = i % columns \* kTitleW;

CGFloat y = i / columns \* kTitleH;

btn.frame = CGRectMake(x, y, kTitleW, kTitleH);

}

// 虽然在这里可以添加动画，但是因为layoutSubviews在旋转的时候，也会调用这个方法，我们在旋转的过程中并不需要添加动画，因此不能在这里添加动画，那怎么能够在点击菜单按钮的时候，更换子标题的时候产生动画了，可以在点击底部菜单按钮时，将整个代码包起来，就能有动画效果了，这样会发现按钮会从左往右变大，因为把整个代码包装起来，其中就包括了执行layoutSubviews，而这里设置了按钮的尺寸。刚开始按钮的尺寸为（0，0），因此会有这样的效果。

// UIView animateWithDuration:<#(NSTimeInterval)#> animations:<#^(void)animations#>

// 设置子标题正确的高度。

int rows = (\_titles.count + columns - 1) / columns;

CGRect frame = self.frame;

frame.size.height = rows \* kTitleH;

self.frame = frame;

}

5.6 实现子标题的动画。当点击底部菜单按钮时，需要动画显示子标题。所以子标题向外界提供一个显示动画和隐藏动画方法。

- (void)show

{

// 子标题加入到\_contentView里尺寸改变了才会调用layoutSubviews方法，设置内部的高度，因此前面调用show的时候，高度为0

// 这里必须调用这个方法，因为调用show的时候，self.frame.size.height为0，只有真正因此不会有动画效果，我们需要动画效果，就必须调用layoutSubviews，就能强制获取当前的正确高度。以后如果系统不给我们正确的高度，我们可以通过这个方法，强制获取我们想要的尺寸。

[self layoutSubviews];

self.transform = CGAffineTransformMakeTranslation(0, -self.frame.size.height);

self.alpha = 0;

[UIView animateWithDuration:kDuration animations:^{

self.transform = CGAffineTransformIdentity;

self.alpha = 1;

}];

}

// 没有子分类标题的时候，执行这个方法

- (void)hide

{

[UIView animateWithDuration:kDuration animations:^{

self.transform = CGAffineTransformMakeTranslation(0, -self.frame.size.height);

self.alpha = 0;

} completion:^(BOOL finished) {

// 将子标题从父控件中移除

[self removeFromSuperview];

// 这里必须做一个这样的操作，每次隐藏将子标题移除父控件后，还需要将子标题的高度清零，因为每次点击菜单按钮都会调用layoutSubviews方法，而当点美食，这时候的子标题的高度很高，而当我们点击抽奖这类没有子标题的按钮时，子标题按钮会移除，而当我们再次点击比美食菜单少的子标题按钮时，就会出现从长度高的到长度低的动画，这不是我们想要的，出现这个问题的原因是之前的子标题的高度很高，而我们当前点击的子标题高度低，而在每次更换子标题的时候，都会产生动画效果，隐藏为了避免这个原因，我们需要每次将子标题移除的时候，将它的高度清零，就能保证每次子标题第一次出现在屏幕，是从0开始慢慢变高。

CGRect frame = self.frame;

frame.size.height = 0;

self.frame = frame;

self.transform = CGAffineTransformIdentity;

self.alpha = 1;

}];

}

5.7 解决动画bug，当点击底部菜单按钮时，背景视图高度慢慢添加的时候，文字先出现了，我们需要做一些处理，让文字和背景视图出现达到一致。

if (self) {

// Initialization code

self.autoresizingMask = UIViewAutoresizingFlexibleWidth;

self.image = [UIImage resizedImage:@"bg\_subfilter\_other.png"];

// 背景视图在执行动画效果的时候，我们发现文字比白色的背景视图先出现，而背景视图表示的是整个子标题的尺寸，是什么原因导致的了，我们知道子视图的出现，并不是根据尺寸有多大，才出现多少，而是只要添加到父视图就能出现，只要不超过屏幕，因此我们可以利用一个属性，来解决文字比背景先出现的情况。

self.clipsToBounds = YES;

// 允许用户交互

self.userInteractionEnabled = YES;

}

### 6.实现子标题的点击，当点击子标题时，需要记录住当前选中了哪个子标题，所以我们这里又需要用到工具类，让工具类帮我们记录住。

6.1 工具类需要分别记录住当前选中的分类，商区，和排序，因此提供这3个接口。

@property (nonatomic, strong) NSString \*currentCategory; // 当前选中的类别

@property (nonatomic, strong) NSString \*currentDistrict; // 当前选中的区域

@property (nonatomic, strong) TGOrder \*currentOrder; // 当前选中的排序

6.2 如何知道当点击子标题的时候，将子标题按钮的文字归为哪个类型。子标题按钮的文字归为哪个类型，只有他父视图（层次关系父子）最清楚，因此我们自定义一个block，回调给父视图，然后父视图提供一个方法，用来将子标题的文本归类，具体的实现由各个子底部视图（继承关系）实现。

#pragma mark 按钮点击事件

- (void)titleClick:(UIButton \*)button

{

// 获取按钮的文字，比较特殊不能使用button.titleLabel.text,因为按钮的文字是根据状态来显示的[button titleForState:UIControlStateNormal],这样才能获取到

\_selectedItem.selected = NO;

button.selected = YES;

\_selectedItem = button;

// 因为子标题视图是不会被销毁的，因此在切换文字标题的时候，之前选中的按钮会重复利用到新的文字标题中，这就会造成本来新的文字标题应该没有按钮被选中，确有按钮选中，这种情况在哪解决了，我们知道每次给文字标题视图设置文字的时候，会重新给按钮设置文字，而且按钮是重复利用的，就意味着之前是选中状态下的按钮，在新的文字界面也是被选中的，这时候我们可以在重新设置title的方法中判断，是否是文字按钮是否和当前选中的按钮文字相同，如果不同，就取消选中状态。

// 记录住当前选中的按钮文字，并将当前选中按钮的文字传给工具类对应的类别中记录起来（有可能是分类的子标题，商区的子标题）

// 当前点击了什么类型的按钮文字，只有父类菜单视图才知道，因此我们需要将数据传递给父类，让父类来归类。

// 设置当前选中的分类文字

if (\_setTitleBlock) {

// 判断是否点击了全部，如果点击了全部就替换成主标题

NSString \*title = [button titleForState:UIControlStateNormal];

if ([title isEqualToString:kAll]) {

title = \_mainTitle;

}

\_setTitleBlock(title);

}

}

6.3 在每次重新刷新子标题的时候，需要判断有没有按钮是之前选中的按钮，如果是之前选中的按钮，就默认选中。那如何判断了，因为子分类按钮分为两种类型（分类，商区），且子标题刷新的时候，并不知道当前正在刷新属于哪个分类的子标题按钮，因为只是传入一个包含所有子标题内容的数组，并不知道当前的按钮属于哪个分类，比如当前的子标题按钮属于商区，但是每次刷新子标题按钮都与当前选中分类比较，那无论你之前选中商区中的哪一个按钮，当重新点击到之前子标题界面时，之前选中的按钮都不会默认点击，因为你始终拿的是当前选中分类按钮来与商区比较，商区按钮显示的文字肯定和分类不同的。因此我们还需要创建一个block，用来告诉子标题当前正在刷新哪个类型的子标题按钮，然后就与当前选中的那个类型比较，就能达到之前选中哪个按钮，重新返回到子标题页面时，还是选中了那个按钮。

@property (nonatomic, copy) void (^setTitleBlock)(NSString \*title); // 将当前选中的标题记录到对应的分类。

@property (nonatomic, copy) NSString \*(^getTitleBlcok)(); // 返回当前记录的选中按钮标题，用于判断当前切换的子标题中有没有之前选中的按钮。

父类底部视图提供方法

- (void)settingSubtitlesView; // 有子标题的归类

分类底部视图

- (void)settingSubtitlesView

{

\_subtitlesView.setTitleBlock = ^(NSString \*title) {

[TGMetaDataTool sharedTGMetaDataTool].currentCategory = title;

};

\_subtitlesView.getTitleBlcok = ^{

return [TGMetaDataTool sharedTGMetaDataTool].currentCategory;

};

}

商区底部视图

- (void)settingSubtitlesView

{

\_subtitlesView.setTitleBlock = ^(NSString \*title) {

[TGMetaDataTool sharedTGMetaDataTool].currentDistrict = title;

};

\_subtitlesView.getTitleBlcok = ^{

return [TGMetaDataTool sharedTGMetaDataTool].currentDistrict;

};

}

排序底部视图不需要实现这个方法，因为他没有子分类。

### 点击没有子分类的底部菜单按钮时，需要将没有子标题的分类记录下来，方法和之前子标题一样。只不过有一点区别，当前操作直接在父类底部视图中操作，无须定义block回调，因为底部按钮没有子分类，就直接到父类底部视图的隐藏子标题方法中处理了，因此我们不需要将按钮文本回调给底部视图，只需要在这个方法中，提供一个设置没有子标题内容的方法，然后在各个子底部视图中实现就好了。

父类底部视图。负责调用，具体实现交给子类

// 将没有子标题的分类记录下来。

[self settingTitlesWithString:[item titleForState:UIControlStateNormal]];

分类底部视图

- (void)settingTitlesWithString:(NSString \*)title

{

[TGMetaDataTool sharedTGMetaDataTool].currentCategory = title;

}

商区底部视图

- (void)settingTitlesWithString:(NSString \*)title

{

[TGMetaDataTool sharedTGMetaDataTool].currentDistrict = title;

}

排序底部视图

- (void)settingTitlesWithString:(NSString \*)title

{

[TGMetaDataTool sharedTGMetaDataTool].currentOrder = [[TGMetaDataTool sharedTGMetaDataTool] orderWithName:title];

}

### 8.顶部菜单接收4个通知（当城市，分类，商区，排序改变时），修改顶部菜单的显示，并且隐藏底部视图。在viewdidload里监听。

8.0 发出通知，我们知道之前城市改变发出城市改变通知，实在重写setCurrentCity里发出的。因此这里也一样，重写3个当前选中哪个按钮的方法，这这些方法中发出通知

// 分类

- (void)setCurrentCategory:(NSString \*)currentCategory

{

\_currentCategory = currentCategory;

// 发出通知

[[NSNotificationCenter defaultCenter] postNotificationName:kCategoryChangeNote object:nil];

}

// 商区

- (void)setCurrentDistrict:(NSString \*)currentDistrict

{

\_currentDistrict = currentDistrict;

// 发出通知

[[NSNotificationCenter defaultCenter] postNotificationName:kDistrictChangeNote object:nil];

}

// 排序

- (void)setCurrentOrder:(TGOrder \*)currentOrder

{

\_currentOrder = currentOrder;

// 发出通知

[[NSNotificationCenter defaultCenter] postNotificationName:kOrderChangeNote object:nil];

}

8.1 由于4个通知有很多地方需要监听，所以写成宏定义。

#define kAddAllNotes(method) \

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(method) name:kCityChangeNote object:nil]; \

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(method) name:kCategoryChangeNote object:nil]; \

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(method) name:kDistrictChangeNote object:nil]; \

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(method) name:kOrderChangeNote object:nil];

8.2 监听到通知后的操作。

- (void)dataChange

{

\_selectedItem.selected = NO;

\_selectedItem = nil;

// 1.分类按钮

NSString \*c = [TGMetaDataTool sharedTGMetaDataTool].currentCategory;

if (c) {

\_cItem.title = c;

}

// 2.商区按钮

NSString \*d = [TGMetaDataTool sharedTGMetaDataTool].currentDistrict;

if (d) {

\_dItem.title = d;

}

// 3.排序按钮

NSString \*o = [TGMetaDataTool sharedTGMetaDataTool].currentOrder.name;

if (o) {

\_oItem.title = o;

}

// 4.隐藏底部菜单，为什么要在这隐藏底部菜单了，因为无论哪个通知触发了，都需要隐藏底部菜单，放在这里实现隐藏底部菜单，就不需要那么麻烦的去判断，哪种类型的按钮点击了，才需要隐藏底部菜单。（子标题按钮和没有子标题的底部菜单按钮需要隐藏底部视图），这两种类型按钮的点击都会触发通知。最终都会调用这个方法。 [\_showingMenu hide];

\_showingMenu = nil;

}

### 9.团购控制器接收通知，请求对应的数据。在监听到通知的方法中，直接调用上拉刷新即可。

- (void)dataChange

{

[\_header beginRefreshing];

}

### 10.团购控制器界面：

10.1 考虑到需要横竖屏切换，且横屏和竖屏没一行显示的个数不一样，用tableView不方便，还要计算每一行的间距，比较麻烦，我们采用UICollectionView,就能轻松搞定这个效果，而且支持横竖屏切换。

10.2 将控制器继承UICollectionViewController，默认视图就是UICollectionView了，注意创建UICollectionViewController的时候必须传入一个参数(UICollectionViewLayout),这个参数必不可少,用来指定UICollectionView中的布局。

10.3 团购控制器在Main控制器中调用init创建的，而如果我们在外面设置UICollectionViewLayout，不符合面向对象思想，我们应该把他封装到团购控制器中，可以重写init方法.注意UICollectionViewLayout是抽象类，因此我们需要创建UICollectionViewLayout的子类来设置布局。

- (id)init

{

UICollectionViewFlowLayout \*layout = [[UICollectionViewFlowLayout alloc] init];

layout.itemSize = CGSizeMake(250, 250);

layout.minimumLineSpacing = 20; // 每一行之间的间距

// 因为子类没有实现initWithCollectionViewLayout这个方法，因此会自动调用父类的方法.

return [self initWithCollectionViewLayout:layout];

}

10.4 创建xib来描述UICollectionView中的每个cell，并在viewdidload中注册。

10.5 控制器中要监听屏幕旋转不要那么麻烦，不需要添加通知，控制器有一个方法可以监听屏幕旋转，view里没有这个方法，只能用通知的方法，监听屏幕旋转。监听屏幕旋转是为了调整cell上下左右的间距.

#pragma mark 监听屏幕旋转

-(void)didRotateFromInterfaceOrientation:(UIInterfaceOrientation)fromInterfaceOrientation

{

// 调整cell的上下左右间距

// 上下间距

CGFloat v = 20;

// 竖屏左右间距

CGFloat h = (self.view.frame.size.width - 2 \* kItemW) / 3;

// 判断是横屏还是竖屏

if (UIInterfaceOrientationIsLandscape(self.interfaceOrientation)) {

// 横屏左右间距

h = (self.view.frame.size.width - 3 \* kItemW) / 4;

}

// 拿到当前的UICollectionViewFlowLayout

UICollectionViewFlowLayout \*layout = (UICollectionViewFlowLayout \*)self.collectionView.collectionViewLayout;

layout.sectionInset = UIEdgeInsetsMake(v, h, v, h);

}

10.6 第一次collection中cell的位置不对，我们需要在第一次出现的时候主动调用didRotateFromInterfaceOrientation方法将位置调好，但是不能在viewDidLoad中调用，viewDidLoad中控制器的尺寸是默认的，只有在即将出现的时候，控制器的宽高才会调整，因此我们在viewWillAppear方法中调用。

#pragma mark 在这能拿到控制器的正确高度,在viewdidload里控制器的高度和宽度是默认的。

- (void)viewWillAppear:(BOOL)animated

{

// 主动调用旋转方法调整一开始的位置。

[self didRotateFromInterfaceOrientation:0];

}

10.7 位置调整好之后，自定义一个DealCell与xib连线，并提供一个deal模型接口，设置xib内容显示的内容。重写setDeal方法

10.8 发送请求，获取数据，一般涉及到请求数据，我们就自定义一个请求工具类，向外界提供一个接口，外界只需要告诉工具类需要加载什么样的数据，工具类就返回什么样的数据。

10.9 工具类的设计，由于需要将工具类作为代理才能获取数据，因此需要设计一个对象方法，请求方法向外界提供一个请求第几页，和提供一个请求成功的block和失败的block。

10.10 工具类难点，由于考虑到外界可能会多次请求，所以内部必须实例化一个字典，用来管理请求产生的多个block，一次请求对应一个block，因此需要弄一个字典保存。

### 11.团购界面细节完善。

11.1 给分类数据添加全部分类按钮，修改模型

11.2 当城市没有选中的时候，点击顶部按钮，不发生任何反应。这个操作在监听顶部按钮中实现。

11.3 给商区添加全部商区按钮，在商区底部视图中修改。先拿到当前城市，然后在添加全部商区

11.4 当点击全部商区，全部分类时，默认不传分类参数和商区参数就是加载全部商区和全部分类，修改请求工具类，在里面多加一个判断

11.5当改变城市的时候，商区也需要改变。否则请求会报错，而显示的也不是当前城市的商区，且商区子标题也不一样，因为点击顶部按钮，底部视图，都不会重新创建，因此我们的商区底部视图，需要单独监听城市改变通知。

11.6 当城市改变，就通知商区底部视图改变所有底部菜单按钮的显示，但是没必要全部重新创建，可以采用和子标题一样的方式创建，但是子标题又有一点区别，就是子标题的父类是UIView内部没有子视图，可以用subViews来判断当前有多少子按钮，但是通知商区的底部按钮的父视图是UIScrollView，而UIScrollView的内部本身就有两个子视图，就是两个滚动条。因此不能跟子标题一样来判断，需要自己创建一个数组来判断当前有多少按钮。

11.7 城市改变需要隐藏子标题，却将不是默认选中的按钮设为非选中

11.8 城市改变，需要改变顶部菜单的显示，而且加载当前城市的全部商区，为了不报区域错误，而且又能修改顶部标题，可以在发出城市改变通知之前，修改标题，即在工具类设置当前城市的方法中修改

11.9 给子标题添加全部按钮

11.10 当点击全部按钮时，需要根据当前子标题的按钮属于哪个分类，就将哪个分类传给当前选中分类。比如选中美食的全部，就把美食传给当前分类。因此我们需要在子标题按钮点击的方法里，判断当前按钮的文字，如果当前文字是全部，我们需要做处理，将他替换成当前分类名称，然后在设置给当前分类，在发出通知，从而通知控制器加载正确的数据。

11.11 怎么拿到当前子标题的主标题了，可以向外界提供一个接口，当底部菜单按钮点击的时候，就将底部菜单的item的名称传进来。

11.12 当点击了全部按钮的时候，修改了当前选中标题的文字，而当我们点击其他分类按钮后，在返回回来的时候，之前选中的全部按钮，没有选中，是因为我们在点击全部按钮后，将当前分类的文字改成了主标题，而不是全部，而我们之前的判断是看两个按钮的文字是否相同，相同就被选中，但是由于我们在点击全部按钮后，将当前分类的文字改成了主标题，而不是全部，所以返回回来，之前的全部按钮并不是当前选中分类的名称，就不会被选中。因此我们当重新刷新子标题时，我们需要判断，如果当前选中的分类名称等于主标题并且是第一个按钮，就默认选中。就能恢复之前选中的全部按钮了。

11.13 小数点解决，由于在deal模型中，我们将字典中当前价格的number对象转换成了doubel类型，导致我们在价格显示上面和返回的数据有差别，返回数据中显示0.3等等数据，我们在显示的时候都是显示了0，怎么让价格显示和返回数据一样了，我们给NSString添加一个分类，专门用来解决小数点问题。

11.14 刷新有bug，当下拉的时候，如果拖到当前有一行cell不见了，然后松手就会出现bug。原因在于松手的时候执行了上拉刷新，将原来数组中的元素都清空了，而松手的时候，滚动视图会往上面恢复原有模样，之前由于上拉，移除屏幕的cell会重新出现在屏幕上，我们知道每当有cell出现在屏幕上的时候，都会调用返回当前cell外观的方法，返回cell的样子，而此时模型数组已经被清空，就会报角标错误。cell.deal = \_deals[indexPath.row];这里indexPath.row是当前cell对应的角标，而数组是都没有元素，怎么取出对应角标的元素。



11.15 解决刷新bug。

#pragma mark 刷新代理方法

- (void)refreshViewBeginRefreshing:(MJRefreshBaseView \*)refreshView

{

if ([refreshView isKindOfClass:[MJRefreshHeaderView class]]) {

// 清理缓存，因为使用SD框架，会讲网络数据保存在内存中，且ipad图片资源比较大，非常耗内存，所以我们应该在每次刷新之前，将之前的缓存数据全部清空

[TGImageTool clear];

// 当上拉刷新的时候，创建新的可变数组，相当于把之前的数组清空了，但是这里又会引发一个bug，就是当下拉的时候，有一行cell不见了，这时候松手，就会报数组越界错误，就是因为在数组清空的时候，且还没有加载数据时，调用了返回每一行cell的外观方法,为什么会调用了，因为当下拉的时候，会产生弹簧效果，之前被移除屏幕的cell要重新回到屏幕，所以就会调用返回cell的方法。

// 因此我们不能在这清空之前的数组，应该在获取到数据的时候，清空之前的数组。

// \_deals = [NSMutableArray array];

\_page = 1; // 第一页

}else{

// 上拉加载更多

\_page++;

}

// 加载第\_page页的数据

[[TGDealTool sharedTGDealTool]dealWithPage:\_page success:^(NSArray \*deals,int totalCount) {

// 0.创建数组,但是只有上拉的时候，才需要清空之前的数组，所以需要添加一个判断

if ([refreshView isKindOfClass:[MJRefreshHeaderView class]]) {

\_deals = [NSMutableArray array];

}

// 1.添加数据

[\_deals addObjectsFromArray:deals];

// 2.刷新表格

[self.collectionView reloadData];

// 3.恢复刷新状态

[refreshView endRefreshing];

// 4.根据数量判断是否需要隐藏上拉控件

\_footer.hidden = \_deals.count >= t otalCount;

} error:^(NSError \*error) {

[refreshView endRefreshing];

}];

}

### 12.点击cell显示团购详情控制器。

12.0 当点击item时，显示遮盖和右边的导航控制器。

12.1 遮盖，由于很多地方用到遮盖，我们可以封装一个遮盖类。内部设置宽高自动伸缩和透明度，背景颜色

if (self) {

// Initialization code

// 背景色

self.backgroundColor = [UIColor blackColor];

// 设置自动伸缩

self.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

// 透明度

self.alpha = kAlpha;

12.2 添加遮盖到右边导航控制器的上面，遮盖的宽度就是右边整一个导航控制器的宽度。

12.3 添加团购详情控制器到右边导航控制器的上面。团购详情控制器需要被包装成一个导航控制器，因此我们直接设置被包装成导航控制器的团购详情控制器的尺寸，视图的宽度自己定义，高度跟右边整一个导航控制器一样。并且设置宽度和高度自适应。将团购详情控制器的视图添加到右边整一个导航控制器上边的时候，还需要添加为它的子控制器。

#pragma mark 显示详情控制器

- (void)showDetail:(TGDeal \*)deal

{

// 显示遮盖

if (\_cover == nil) {

\_cover = [TGCover coverWithTarget:self action:@selector(coverClick)];

}

\_cover.frame = self.navigationController.view.bounds;

\_cover.alpha = 0;

[UIView animateWithDuration:kDefaultAnimDuration animations:^{

[\_cover reset];

}];

[self.navigationController.view addSubview:\_cover];

// 添加团购详情控制器

TGDealDetailController \*detail = [[TGDealDetailController alloc] init];

detail.navigationItem.leftBarButtonItem = [UIBarButtonItem itemWithIcon:@"btn\_nav\_close.png" highlightedIcon:@"btn\_nav\_close\_hl.png" target:self action:@selector(coverClick)];

detail.deal = deal;

TGNavigationController \*nav = [[TGNavigationController alloc] initWithRootViewController:detail];

nav.view.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleLeftMargin;

nav.view.frame = CGRectMake(\_cover.frame.size.width, 0, kDetailWidth, \_cover.frame.size.height);

// 当2个控制器互为父子关系时，它们的view也是互为父子关系

[self.navigationController.view addSubview:nav.view];

[self.navigationController addChildViewController:nav];

[UIView animateWithDuration:kDefaultAnimDuration animations:^{

CGRect f = nav.view.frame;

f.origin.x -= kDetailWidth;

nav.view.frame = f;

}];

}

12.4 给遮盖添加手势，添加遮盖，移除遮盖和团购详情控制器。

#pragma mark 点击遮盖

- (void)coverClick

{

UIViewController \*nav = [self.navigationController.childViewControllers lastObject];

// 隐藏遮盖

[UIView animateWithDuration:kDefaultAnimDuration animations:^{

\_cover.alpha = 0;

// 2.隐藏控制器

CGRect f = nav.view.frame;

f.origin.x += kDetailWidth;

nav.view.frame = f;

} completion:^(BOOL finished) {

[\_cover removeFromSuperview];

[nav.view removeFromSuperview];

[nav removeFromParentViewController];

}];

[\_cover removeFromSuperview];

}

### 13.团购详情控制器的搭建。

13.1 先设置导航条上的内容显示，由于导航条上左边的返回按钮，需要调用团购控制器的方法，因此我们直接在团购控制器中添加，其他按钮都在团购详情控制器内添加。注意在哪里添加需要看调用方法中用到什么成员变量，如果用到其他控制器的变量，就选择在其他控制器添加，方便。

#pragma 基本设置

- (void)baseSetting

{

// 1.背景色

self.view.backgroundColor = kGlobalBg;

// 2.设置标题

self.title = \_deal.title;

// 3.右上角的按钮

self.navigationItem.rightBarButtonItems = @[

[UIBarButtonItem itemWithIcon:@"btn\_share.png" highlightedIcon:@"btn\_share\_pressed.png" target:nil action:nil],

[UIBarButtonItem itemWithIcon:@"ic\_deal\_collect.png" highlightedIcon:@"ic\_deal\_collect\_pressed.png" target:nil action:nil]];

}

13.2 顶部购买栏，自定义一个购买栏view，用xib来描述这个视图，描述xib时，需要设置自动拉伸属性。由于宽度跟控制器的宽度一样，因此内部不需要固定宽度，由外界来设置。购买栏view需要提供一个接口，让外界传递一个团购模型来，设置当前价格和之前价格的显示。在设置团购模型的时候，设置显示内容，由于购买栏是个view，只能通过绘图的方式，给view设置背景图片，并且需要在xib中将view的颜色设置为clear。

- (void)setDeal:(TGDeal \*)deal

{

\_deal = deal;

\_listPrice.text = [NSString stringWithFormat:@"%@ 元", deal.list\_price\_text];

\_currentPrice.text = deal.current\_price\_text;

}

+ (id)buyDock

{

return [[NSBundle mainBundle] loadNibNamed:@"TGBuyDock" owner:nil options:nil][0];

}

- (void)drawRect:(CGRect)rect

{

[[UIImage resizedImage:@"bg\_buyBtn.png"] drawInRect:rect];

}

13.2.1 以前价格label需要自定义一个label，用绘图方法在显示的数字上面画一根线。

- (void)drawRect:(CGRect)rect

{

// 一定要调用父类的方法，否则控件的原始内容不会显示，因为一个控件中所有内容都是画上去的。

[super drawRect:rect];

// 获取上下文

CGContextRef ctx = UIGraphicsGetCurrentContext();

// 设置颜色

[self.textColor setStroke];

// 画线

CGFloat y = rect.size.height \* 0.5;

CGContextMoveToPoint(ctx, 0, y);

CGFloat endX = [self.text sizeWithFont:self.font].width;

CGContextAddLineToPoint(ctx, endX, y);

// 渲染

CGContextStrokePath(ctx);

}

13.3 添加顶部购买栏

#pragma 顶部购买栏

- (void)addBuyDock

{

TGBuyDock \*dock = [TGBuyDock buyDock];

dock.deal = \_deal;

dock.frame = CGRectMake(0, 0, self.view.frame.size.width, 60);

[self.view addSubview:dock];

}

13.4 右边dock搭建，自定义右边dock，用xib描述，需要设置自动拉伸属性。自定义右边dock，当按钮点击的时候，需要调整3个按钮的层次关系，当点击最上边按钮时，需要设置最下边按钮到最下面，且当前点击的按钮在最上面。点击中间按钮的时候，直接设置当前按钮在最上面，点击下边按钮时，需要设置上边按钮在最下面，当前按钮在最前面。其他设置跟之前的dock设置一样，设置代理，默认选中，和自定义dock上面的按钮，重写高亮方法。由于不需要用到外界的尺寸，所以内部固定宽高。

- (void)setFrame:(CGRect)frame

{

frame.size = self.frame.size;

[super setFrame:frame];

}

+ (id)detailDock

{

return [[NSBundle mainBundle] loadNibNamed:@"TGDetailDock" owner:nil options:nil][0];

}

- (void)awakeFromNib

{

[self btnClick:\_infoBtn];

}

- (IBAction)btnClick:(UIButton \*)sender {

// 0.通知代理

if ([\_delegate respondsToSelector:@selector(detailDock:btnClickFrom:to:)]) {

[\_delegate detailDock:self btnClickFrom:\_selectedBtn.tag to:sender.tag];

}

// 1.控制按钮状态

\_selectedBtn.enabled = YES;

sender.enabled = NO;

\_selectedBtn = sender;

// 2.添加被点击的按钮在最上面

if (sender == \_infoBtn) { // 点击了第1个按钮

[self insertSubview:\_merchantBtn atIndex:0];

} else if (sender == \_merchantBtn) { // 点击了第3个按钮

[self insertSubview:\_infoBtn atIndex:0];

}

[self bringSubviewToFront:sender];

}

@end

@implementation TGDetailDockItem

- (void)setHighlighted:(BOOL)highlighted {}

@end

13.5 添加右边dock和实现代理方法

#pragma mark - dock的代理方法

- (void)detailDock:(TGDetailDock \*)detailDock btnClickFrom:(int)from to:(int)to

{

// 1.移除旧控制器的view

UIViewController \*old = self.childViewControllers[from];

[old.view removeFromSuperview];

// 2.添加新控制器的view

UIViewController \*new = self.childViewControllers[to];

// 设置宽度和高度自动拉伸，就会随着父视图拉伸而拉伸，且跟父视图的宽度始终保持一个dock的宽度。

new.view.autoresizingMask = UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight;

CGFloat w = self.view.frame.size.width - \_detailDock.frame.size.width;

CGFloat h = self.view.frame.size.height;

new.view.frame = CGRectMake(0, 0, w, h);

[self.view insertSubview:new.view atIndex:0];

}

#pragma mark 添加右边选项卡

- (void)addDetailDock

{

TGDetailDock \*dock = [TGDetailDock detailDock];

CGSize size = dock.frame.size;

CGFloat x = self.view.frame.size.width - size.width;

CGFloat y = self.view.frame.size.height - size.height - 100;

dock.frame = CGRectMake(x, y, 0, 0);

dock.delegate = self;

[self.view addSubview:dock];

\_detailDock = dock;

}

### 14. 搭建图文详情控制器

14..1 主视图直接设置为webView，就好了。

14.2 让外界提供一个接口，传入团购模型进来，模型中有我们想要加载的页面的url。

14.3 加载网页后，我们发现并不是我们想要的网页，因此我们需要做一些处理。我们需要的网页是点击更多网文详情之后显示网页，因此我们需要拿到他的url，可以通过设置webView代理，获取他的url。

- (BOOL)webView:(UIWebView \*)webView shouldStartLoadWithRequest:(NSURLRequest \*)request navigationType:(UIWebViewNavigationType)navigationType

{

// 获取图文详情的url--<http://m.dianping.com/tuan/deal/moreinfo/5113330>

MyLog(@"%@",request.URL);

return YES;

}

14.4 获取图文详情url后，h**ttp://m.dianping.com/tuan/deal/moreinfo/5113330，**我们发现了一个规律，就是最后面是一个团购ID，因此我们可以采用拼接的方式，来获取每个团购商品的图文详情URL。

14.5 处理图文详情URL，我们发现团购ID都是8-**5113330这种样子，因此我们拼接的时候，还需要将ID做处理，将8-去掉。**

- (void)viewDidLoad

{

[super viewDidLoad];

// 最原始的url，需要做一些处理

// NSURL \*url = [NSURL URLWithString:\_deal.deal\_h5\_url];

// 处理ID

NSRange range = [\_deal.deal\_id rangeOfString:@"-"];

// from是从当前角标开始，因此需要角标+1，才不是-

NSString \*ID = [\_deal.deal\_id substringFromIndex:(range.location + 1)];

// 拼接图文详情url

NSString \*urlStr = [NSString stringWithFormat:@"http://m.dianping.com/tuan/deal/moreinfo/%@",ID];

NSURL \*url = [NSURL URLWithString:urlStr];

NSURLRequest \*request = [NSURLRequest requestWithURL:url];

[\_webView loadRequest:request];

}

**14.6 加载图文详情后，我们发现图文详情有很多显示的东西，不是我们想要的，我们需要通过脚本语言javascript,获取我们想要的网页内容。这个操作，我们必须在web加载完网页之后，才能操作，和javascript操作html必须放在内容最后面执行角标语言。**

**14.7 获取我们想要的网页部分，如果知道我们想要的网页部分，首先必须用火狐打开网页，用火狐bug显示网页源码，但是由于我们拿到的url是移动端的，因此不能在浏览器中打开，所以我们需要先要拿到图文详情的html，在把它生成为我们自己的静态网页，就能在浏览器中打开了。**

**14.7.1 在webView加载完后，添加脚本语言，获取图文详情的HTML，并将他打印出来，复制，粘贴到UE，生成自己的静态网页。**

#pragma mark 一定要在加载完成，执行脚本语言，才能获取到html

- (void)webViewDidFinishLoad:(UIWebView \*)webView

{

// 执行脚本语言，获取图文详情的HTML。

NSString \*html = [webView stringByEvaluatingJavaScriptFromString:@"(document.getElementsByTagName('html')[0]).innerHTML;"];

MyLog(@"\n%@",html);

}

**14.8 在网页中打开我们自己生成的图文详情静态网页，打开源码，找到我们想要的部分，通过脚本语言将网页转换为我们想要显示的界面。**

**14.8.1 我们发现我们想要的部分在section里面，因此我们只要让webView显示section就可以了。**

**14.8.2 利用脚本语言生成我们想要的界面。需要注意的是，脚本一定要写在最后面。**

**14.8.3 在我们自己的静态网页中添加脚本语言，试试能不能达到我们想要的效果。**

**<script type="text/javascript">**

**// 取出body var body = document.body;**

**// 取出section var section = document.getElementById('J\_section');**

**// 清空body里的HTML body.innerHTML = '';**

**// 添加section到body body.appendChild(section);**

**</script>**

14.8.4 重新刷新网页，发现网页只剩我们最后想要的东西了，所以我们可以在webView中执行我们的脚本语言了。

// 1.拼接脚本

NSMutableString \*script = [NSMutableString string];

// 1.1.取出body

[script appendString:@"var body = document.body;"];

// 1.2.取出section

[script appendString:@"var section = document.getElementById('J\_section');"];

// 1.3.清空body

[script appendString:@"body.innerHTML = '';"];

// 1.4.添加section到body

[script appendString:@"body.appendChild(section);"];

// 2.执行脚本

[webView stringByEvaluatingJavaScriptFromString:script];

14.9 webView细节处理

14.9.1 我们发现最终webView展示了我们想要的结果页面，但是在之前还是会显示最原始的，没有动过手脚的网页信息，原因很简单，因为我们是在网页加载完毕的时候，才用脚本进行处理的，因此我们可以在网页开始加载的时候先给整个控制器先添加一层遮盖，当脚本执行完之后，我们在移除遮盖，就能屏蔽这一效果了。

14.9.2 加载的时候看见一层白的，用户体验不好，我们添加一个指示器。

#pragma mark webView代理方法

- (void)webViewDidStartLoad:(UIWebView \*)webView

{

\_cover = [[UIView alloc] init];

\_cover.frame = \_webView.bounds;

// 颜色必须保持一致

\_cover.backgroundColor = kGlobalBg;

\_cover.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

[\_webView addSubview:\_cover];

// 指示器

UIActivityIndicatorView \*indicator = [[UIActivityIndicatorView alloc] initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleGray];

CGFloat centerX = \_cover.frame.size.width \* 0.5;

CGFloat centerY = \_cover.frame.size.height \* 0.5;

indicator.center = CGPointMake(centerX, centerY);

// 设置上下拉伸，就能保证在中间显示了

indicator.autoresizingMask = UIViewAutoresizingFlexibleTopMargin | UIViewAutoresizingFlexibleBottomMargin;

// 必须调用这个方法才能显示指示器

[indicator startAnimating];

[\_cover addSubview:indicator];

}

14.10 webView左上角默认显示在购买栏那，我们需要让webView向下移动，但是webView的尺寸必须设置在左上角，才能滚动的时候，能穿过购买栏下部的感觉，为了达到这个效果，而且webView又能出现在下面，我们可以给webView添加额外的滚动区域，然后设置contentOffset，注意设置contentOffset，只有在webView加载完毕设置才有用。

14.11 webView上有个灰灰的阴影，怎么挪掉了，这个灰灰的阴影，其实是webView里ScrollView里的UIImageView造成的，因此我们可以遍历scrollView里所有的UIImageView，将所有的UIImageView移除就行了。

- (void)webViewDidFinishLoad:(UIWebView \*)webView

{

// 执行脚本语言，获取图文详情的HTML。

// NSString \*html = [webView stringByEvaluatingJavaScriptFromString:@"(document.getElementsByTagName('html')[0]).innerHTML;"];

// MyLog(@"\n%@",html);

// 移除webView灰灰的阴影

for (UIView \*view in \_webView.scrollView.subviews) {

if ([view isKindOfClass:[UIImageView class]]) {

[view removeFromSuperview];

}

}

// 1.拼接脚本

NSMutableString \*script = [NSMutableString string];

// 1.1.取出body

[script appendString:@"var body = document.body;"];

// 1.2.取出section

[script appendString:@"var section = document.getElementById('J\_section');"];

// 1.3.清空body

[script appendString:@"body.innerHTML = '';"];

// 1.4.添加section到body

[script appendString:@"body.appendChild(section);"];

// 2.执行脚本

[webView stringByEvaluatingJavaScriptFromString:script];

// 给webView里的scrollView添加额外的滚动区域，顶部为购买栏的高度

\_webView.scrollView.contentInset = UIEdgeInsetsMake(60, 0, 0, 0);

// 注意给webView设置移动位置contentOffset，只有在webView加载完毕才有用。

// 并设置contentOffset为-100，contentOffset指的是屏幕移动，并不是scorllView的位置移动，因此如果我们想要scorllView往下移动，我们需要设置屏幕往上移动，就能达到效果了。

\_webView.scrollView.contentOffset = CGPointMake(0, -60);

// 执行完脚本后移除遮盖

[\_cover removeFromSuperview];

// 遮盖对象只用一次，因此用完，我们就释放它，直接设置为nil，就会被释放

\_cover = nil;

}

15.搭建团购信息简介

15.1 添加UIScrollView,高度和控制器的高度一样，但是宽度不需要那么宽，这样方便在scrollView中添加子控件的时候，不需要计算每个控件的x，y值。我们只需要将UIScrollView添加到控制器的中间就好了。且需要设置自动拉伸属性，为了保证位置居中，且宽度不拉伸，需要设置高度和左右间距拉伸。

15.2 设置头部视图，用xib描述，这里有需要自动拉伸的细节。

首先顶部的UIImageView需要保持上左右位置不变，只拉伸宽度。

中间的label，需要拉伸宽度，上下左右位置不变。

中间分割线，宽度需要拉伸，且保持下边位置不变，上边位置不确定。

底部右边按钮需要保持右边和下边的位置不变，右边按钮需要保持左边和下边的位置不变。按钮内部的显示设置也需要注意，为了不让内部的图片，随着文字的增多或者减少，而改变位置，需要修改按钮内部内容向左靠齐，而不是居中显示。然后调节content的距离，自己将内部控件的位置往右边挪。也能和居中显示同样的效果。

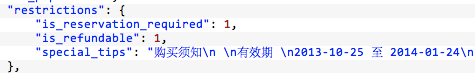
底部按钮中的分割线，需要保持底部位置不变。就能永远在中间了。

15.3 自定义头部视图，需要提供一个团购模型接口，根据模型设置视图的显示。先将模型中已有的数据添加到头部视图中显示。

15.4 将头部视图添加到scrollView中，然后在加载更详细的数据，为什么不一开始就加载数据了，因为本身就有一些数据了，可以先展示给用户看，然后在加载更详细的数据。

15.5 加载更详细数据，返回的数据中有比较特殊的。看到有带点的结果字段，说明点前面的是模型，因为我们知道标识符是不能加.的，因此不能作为我们的成员属性解析，我们需要在创建一个restriction模型，然后里面包含3个点后面的成员属性。还有一个地方也能看出，必须要这样解析。直接看返回的JSON数据。





15.5.1 分析完数据之后，我们就能给deal模型，添加一些更详细的成员属性了，并且赋值。这里由于deal模型中多了一个restriction模型，且json返回的是字典，因此我们需要重写setrestriction将字典转换为模型。

15.5.2 模型建立好了之后，我们就需要请求数据了，给之前写好的请求工具类中添加一个专门请求一个团购详细信息的方法，用来请求详细的团购信息。

15.5.3 头视图细节处理

15.5.4 自定义内容视图

15.5.5 添加内容视图到scrollView中，设置他们的y值和滚动视图的时候，第一个想法是下一个内容视图的y值等于UIScrollView最后一个子视图的y值+一定的间距。这样的想法，在UIView中是适用的，但是在UIScrollView中，是不适用的，因为当UIScrollView的contentSize达到可以滚动的时候，他会自动添加右边的滚动条，这时候UIScrollView的最后一个子视图就是滚动条了，因此算出的位置就不准了，解决一：用一个变量，记住每一个内容视图的最大y值+一定间距，这个变量就是下一个内容视图的y值. 解决二: 由于我们每次添加新的内容视图，都需要重新设置UIScrollView的contentSize，contentSize的值就是每一个内容视图的最大y值+一定间距

## 八．实现收藏功能

1.监听团购详情右上角收藏按钮。由于收藏是有顺序的，因此我们用一个数组来管理所有收藏的团购详情模型，

当收藏点击的时候，判断是否收藏过了，如果没有收藏过，将当前团购详情模型添加到收藏数组并归档，如果已经收藏过了，删除当前收藏的模型，然后再次将收藏的数组归档。

#pragma mark 收藏

- (void)collect

{

// 当模型收藏状态改变的时候，我们需要修改barItem的图片显示，barItem是一个自定义custom，而这个custom是一个按钮，因此修改barItem的图片，就变成了修改按钮的图片，因此我们需要拿到这个按钮。

UIButton \*btn = (UIButton \*)[self.navigationItem.rightBarButtonItems[1] customView];

// 注意这里取消收藏或者收藏团购模型的时候，我们不能只修改图片显示，还需要修改当前模型的收藏状态，修改收藏状态可以直接封装到收藏工具内部。

if (\_deal.collected) {

// 取消收藏

[[TGCollectTool sharedTGCollectTool] uncollectDeal:\_deal];

[btn setBackgroundImage:[UIImage imageNamed:@"ic\_deal\_collect.png"] forState:UIControlStateNormal];

}else{

// 收藏

[[TGCollectTool sharedTGCollectTool] collectDeal:\_deal];

[btn setBackgroundImage:[UIImage imageNamed:@"ic\_collect\_suc.png"] forState:UIControlStateNormal];

}

}

2.由于用到归档，我们需要创建一个工具类，专门处理归档，向外界提供4个接口，一个获取收藏的所有团购详情信息，一个添加收藏，一个取消收藏，一个判断团购是否收藏。

- (id)init

{

if (self = [super init]) {

// 1.加载沙盒中的收藏数据

\_collectedDeals = [NSKeyedUnarchiver unarchiveObjectWithFile:kFilePath];

// 2.第一次没有收藏数据

if (\_collectedDeals == nil) {

\_collectedDeals = [NSMutableArray array];

}

}

return self;

}

#pragma mark 判断当前团购是否被收藏了。

- (void)handleDeal:(TGDeal \*)deal

{

deal.collected = [\_collectedDeals containsObject:deal];

}

- (void)collectDeal:(TGDeal \*)deal

{

// 收藏之后，需要修改模型的收藏状态。

deal.collected = YES;

// 没收藏一个，都放在数组第一个元素。

[\_collectedDeals insertObject:deal atIndex:0];

[NSKeyedArchiver archiveRootObject:\_collectedDeals toFile:kFilePath];

}

- (void)uncollectDeal:(TGDeal \*)deal

{

// 取消收藏之后，修改模型的收藏状态

deal.collected = NO;

[\_collectedDeals removeObject:deal];

// 将数组归档。

[NSKeyedArchiver archiveRootObject:\_collectedDeals toFile:kFilePath];

}

3 数组内部删除元素，是通过判断地址是否相等来删除相应的元素。而当我们每次收藏控制器生成的团购模型和团购列表控制器生成的团购模型肯定不是一个对象，只不过他们的内容是一样的，但是地址肯定不一样。那如何在收藏数组中删除指定的团购模型了，首先这个团购模型地址肯定是不一样的，但是内容可能会有一样的。我们只需要删除数组中内容和指定模型一样内容的模型就可以了。方法一，遍历数组中的团购模型，拿每一个团购模型的团购ID来和指定模型的ID对比，如果相同，就删除。方法二，数组删除元素，内部会调用模型中的isEqual,判断两个模型的地址是否相等，因此我们可以在模型中重写这个方法，在这个方法中用团购ID来判断两个对象是否相等。

4. 每次进入团购详情界面的时候，右边收藏按钮需要根据当前模型是否归档，来决定收藏按钮显示什么图片。

// 3.处理团购的收藏属性

// 右上角的收藏按钮的图片需要根据当前团购详情模型是否被收藏决定的。而是否被收藏只有收藏工具类才最清楚，因此工具类需要提供一个方法，判断当前团购详情是否被收藏了。

[[TGCollectTool sharedTGCollectTool]handleDeal:\_deal];

// 4.右上角的按钮

NSString \*collection = \_deal.collected ? @"ic\_collect\_suc.png" : @"ic\_deal\_collect.png";

self.navigationItem.rightBarButtonItems = @[

[UIBarButtonItem itemWithIcon:@"btn\_share.png" highlightedIcon:@"btn\_share\_pressed.png" target:nil action:nil],

[UIBarButtonItem itemWithIcon:collection highlightedIcon:@"ic\_deal\_collect\_pressed.png" target:self action:@selector(collect)]];

## 九．收藏控制器界面搭建

1.由于收藏控制器和团购列表控制器一样，因此我们抽父类。然后父类定义一个方法，获取子类中所有团购数据，然后在子类中具体实现这个方法。在收藏中的所有团购数据，在收藏工具类里面。

2.当点击收藏按钮时，需要发送通知，然后收藏控制器接收通知，刷新UIColloectionView。

## 十．地图控制器搭建

1.首先创建mapView添加到控制器view中。

2.让地图自动定位到用户的位置，并且设置区域，让地图显示的尺寸是我们想要看见的范围，只需要定位一次，用一个成员变量记录住。

3.设置地图的代理，当地图的区域改变的时候，发送请求，获取周边区域的商家信息，然后在地图上添加商家大头针。

3.1 发送请求，需要给请求工具类定义一个获取数据的方法。

3.2 添加自定义大头针视图，首先需要自定义一个大头针类。然后在实现mapView的代理方法，设置大头针的中心位置。大头针的中心位置，需要给团购模型添加一个商家模型，每个团购模型对应一个商家模型数组，因为一个团购的商家有可能有很多分店。我们需要给每个商家的分店都设置大头针，每个商家模型中有经纬度，我们取出来直接设置大头针的中心位置即可。

3.3 处理地图的细节。为了避免重复添加大头针，需要判断一下有没有重复添加，怎么判断了，由于每个大头针对应一个团购对象，我们可以用一个数组存储添加到地图上的大头针的团购模型，然后当有一个大头针需要添加到地图上显示的时候，我们直接判断数组中有没有包含当前大头针的团购模型，因为如何判断团购模型一致，我们已经重写isequale的方法了，根据团购ID判断。

3.4 设置大头针视图的图片，大头针的图片我们发现和分类的图片很像，大头针的图片名称是ic\_category\_1 分类图片的名称是ic\_filter\_category\_1，我们可以这样思考，大头针的图片名称跟分类图片名称那么相似，我们完全可以先找到团购模型中的分类，我们分类模型中有对应分类的名称和图片，然后取出分类中的图片，然后将分类图片转换为大头针的图片，因此我们可以给自定义大头针类提供一个团购模型，和一个图片名称接口，团购模型用于判断当前团购属于哪个分类然后找到对应分类的图片，如何判断了，我们知道元数据工具类中有所有分类的数据，因此在元数据工具类中我们提供一个接口，传入一个分类名称进来，就返回对应的分类图片回去。图片名称接口用于给大头针视图提供相应的图片名称。然后我们可以这样想，给大头针设置一个图片， 想给大头针视图设置图片，必须知道大头针中团购模型属于什么分类。

# 识别复杂的plist文件。

## 读取复杂的plist文件心得，如果将复杂的plist文件转化成数据模型，首先plist文件中是一个数组，一开始就知道，数组中的每个元素都是一个模型（城市组信息模型），这个模型包含这个组有多少城市（cities），组名（name）两个属性，有的城市可能会有hot属性，点击城市数组，我们就能判断城市数组每个元素，又是一个模型（城市模型）,城市模型包含这个组有城市有多少区（districts）和城市名（name），有点击区数组，我们知道，每个区数组元素又是一个模型，区模型包含街道（neighborhoods）和name（区名），点击街道数组，发现里面是字符串类型，就不需要在建立模型了。因为我创建模型，是因为需要把复杂的东西，转化简单，一般都是把字典对象转化成模型对象。其他对象一般不需要转化为模型。

