寻梦记账看板架构

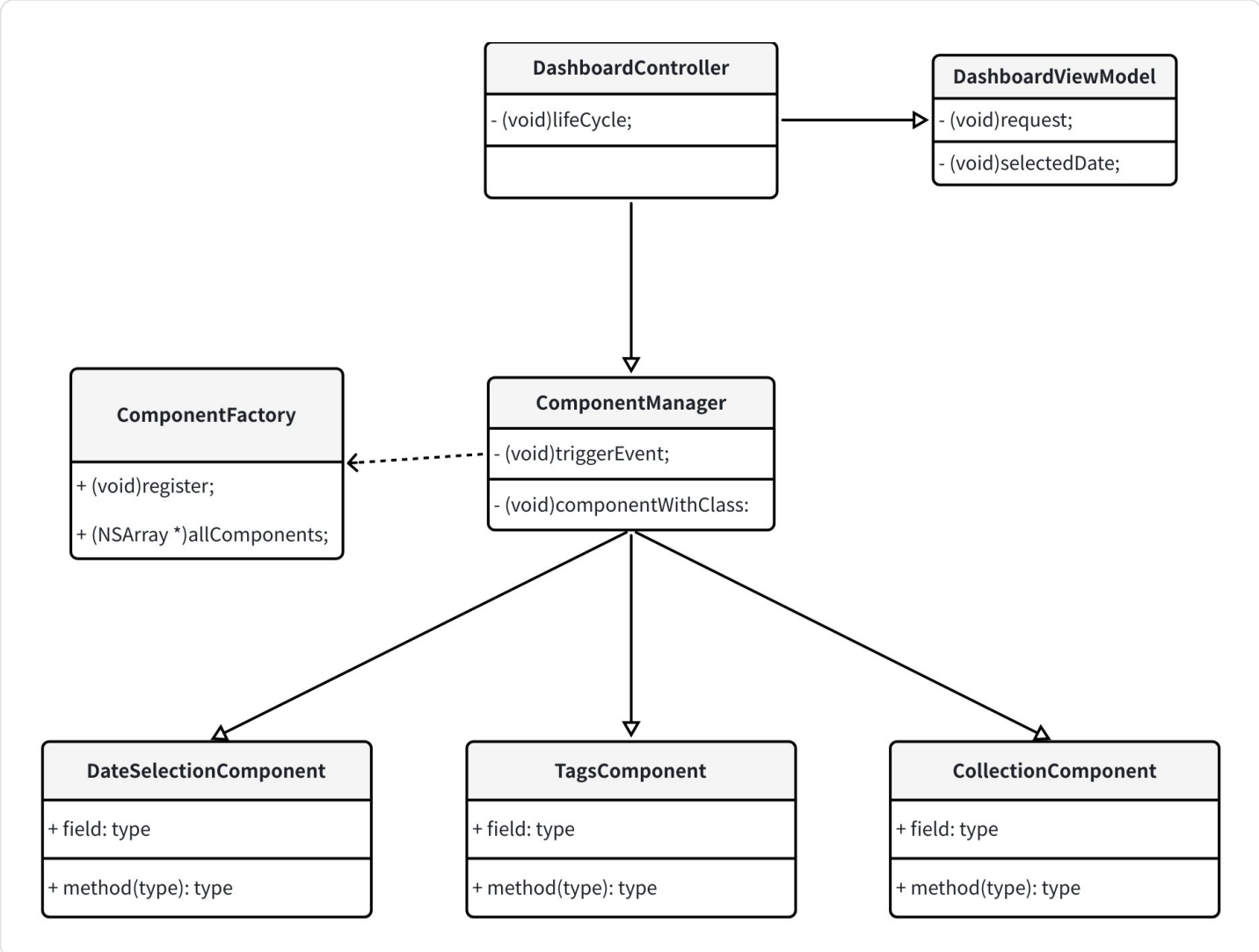
市面上类似bugly, 友盟等厂商都有成熟的埋点 - 看板功能，但是多数都需要收费，为了节约成本，寻梦记账选择

- 崩溃日志使用bugly
- 自研埋点 + 看板

也就有了这个项目

1. 架构设计

「看板」整个软件很可能只有一个页面，其它的功能性页面都会以半屏页面的形式出现（比如日期选择，查看详情等子页面），所以看板实际上非常像一个直播间，把每一个模块（component）看作是一个单独的业务，这样就可以实现不同业务之间的解藕，业务和controller解藕



1.1 ComponentFactory

类的单一性原则，把创建逻辑从Manager拆分出来，manager只负责管理components，而Factory只负责创建component

代码块

```
@implementation XMDashboardComponentFactory

static NSMutableDictionary<NSString *, Class> *_factory;
static dispatch_semaphore_t _lock;

+ (void)initialize
{
    _factory = [NSMutableDictionary dictionary];
    _lock = dispatch_semaphore_create(1);
}

+ (void)registerComponentClass:(Class)componentClass {
    if (![componentClass conformsToProtocol:@protocol(XMDashboardComponent)]) {
        return;
    }

    NSString *identifier = [componentClass xm_identifier];
    if (StringUtil.isBlank(identifier)) {
        return;
    }

    dispatch_semaphore_wait(_lock, DISPATCH_TIME_FOREVER);
    _factory[identifier] = componentClass;
    dispatch_semaphore_signal(_lock);
}

@end

@implementation XMTagHeaderComponent

+ (void)load {
    [XMDashboardComponentFactory registerComponentClass:self];
}

@end
```

- components将会在+ load的时候实现自动注册
- 这里有趣的是，+ load在类对象被加载进内存后立即调用，initialized通常在类对象收到第一条消息前调用，而这个时候，如果有component先于factory被加载进内存，factory将会收到消息从而调用initialized，这时候+initialized的调用有可能早于+load，实验如下

代码块

```
@implementation XMDashboardComponentFactory

+ (void)load {
    NSLog(@"load factory");
}

+ (void)initialize
{
    NSLog(@"initialized factory");
    _factory = [NSMutableDictionary dictionary];
    _lock = dispatch_semaphore_create(1);
}

@end

@implementation XMTagHeaderComponent

+ (void)load {
    NSLog(@"load Component");
    [XMDashboardComponentFactory registerComponentClass:self];
}

@end

// initialized factory
// load factory
// load Component
```

- 这里可以看到，虽然打印日志的component晚于factory，但非常幸运的是factory的initialize的确早于load调用了，应该是其它的component调用了注册方法是以复现了这个实验

📌 这里留个疑问吧，问了GPT，这一次GPT没有产生幻觉，始终非常确定同一个类的+load加载一定早于+initialized，即使打了log发给GPT，它还是非常确定

1.2 通信

单向数据流动，点击事件的传递，多模块之间的通知如何实现，这些自研的功能将会让模块之间更加解藕，让component/viewController的数据流动更加清晰，更易于测试

1.2.1 EventBus

组建之间总免不了相互依赖，比如TagComponent可能依赖CollectionView展示，需要知道CollectionView目前的contentOffset

- NSNotification可以解决问题，但是NSNotification有一些问题
  - 需要手动remove
  - 消息在哪个线程响应?
- EventBus关键代码

代码块

```
- (XMEventToken *)subscribeEventKey:(NSString *)key
                             target:(id)target
                            handler:(XMEventHandler)handler
{
    if (!key || !target || !handler) return nil;

    XMEventToken *token = [[XMEventToken alloc] init];
    XMEventWrapper *wrapper = [[XMEventWrapper alloc] init];
    wrapper.handler = [handler copy];
    wrapper.target = target;
    wrapper.token = token;

    pthread_mutex_lock(&_lock);
    NSMutableArray *events = _subscribers[key];
    if (!events) {
        events = [NSMutableArray array];
        _subscribers[key] = events;
    }
    [events addObject:wrapper];
    pthread_mutex_unlock(&_lock);

    objc_setAssociatedObject(token, @"unbind_block", ^{
        [self unsubscribe:token];
    }, OBJC_ASSOCIATION_COPY_NONATOMIC);
    objc_setAssociatedObject(target, (__bridge const void *)(token), token,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);

    return token;
}

- (void)postOnMainThread:(id)event {
    if ([NSThread isMainThread]) {
        [self post:event];
    } else {
        dispatch_async(dispatch_get_main_queue(), ^{
            [self post:event];
        });
    }
}
```

- 使用associatedObject的特性，让业务强持有token，当业务释放的时候，因为wrapper持有token的弱引用，如果没有业务方持有token就会被release，这个时候retainCount == 0，将会调用dealloc，dealloc的时候会进行unsubscribe，释放所有wrapper
    - 使用associatedObject注入token，做到业务无感知，自动解绑

1.2.2 Observable

数据单向流动的viewController

传统：点击日期选择按钮 -> 确定选择 -> 请求服务端/更新UI -> 请求成功再更新UI（UI重复刷新，UI状态混乱，到处刷新）

单向数据流：点击日期选择按钮 -> 确定选择 -> 请求服务端 -> 请求成功更新UI

- 对比NSNotification和KVC，Observable的优势
    - 强类型传递，callback的时候获取到model，而不是字典形式的userInfo
    - 不使用kvc
    - 无需手动remove，生命周期跟随model
    - 防抖
    - 支持willChange, initBlock, didChange几种变化，支持strong, copy类型赋值
- 关于防抖和防抖实验

如果用户在搜索中输入文字，输入顺序：zao, 早，早饭，实际上客户端并不需要响应每次输入，只需要响应最后一次早饭，搜索一般会在characterDidChange类似的方法里调用，早饭相当于要响应2-6次事件，为了防止每次点击都进行搜索，设定了防抖功能

代码块

```
- (void)xm_setValue:(id)value {
    pthread_mutex_lock(&_lock);
    _pendingValue = value;
    if (_debounceBlock) {
        // 取消队列中的block，有无法取消的可能性
        dispatch_block_cancel(_debounceBlock);
        _debounceBlock = nil;
    }
    if (_debounceTimeInterval > 0) {
        WS
        XMBlock block = dispatch_block_create(0, ^{
            [weak_self _commitValue:weak_self.pendingValue];
        });
        _debounceBlock = block;
```

```
15          dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)
    (_debounceTimeInterval * NSEC_PER_SEC)), dispatch_get_main_queue(), block);
16          pthread_mutex_unlock(&_lock);
17          return;
18      }
19      pthread_mutex_unlock(&_lock);
20
21      [self _commitValue:value];
22  }
23
24  - (void)_commitValue:(id)value {
25      pthread_mutex_lock(&_lock);
26      if (value == _value) {
27          // 必须是新的实例才返回，不使用isEqual:
28          pthread_mutex_unlock(&_lock);
29          return;
30      }
31
32      pthread_mutex_unlock(&_lock);
33
34      id newValue = _valuePolicy == XMObservableValuePolicyStrong ? value :
    [value copy];
35
36      [self _performWillChangeObserverCallback:newValue];
37
38      pthread_mutex_lock(&_lock);
39
40      id oldValue = _value;
41      _value = newValue;
42      pthread_mutex_unlock(&_lock);
43
44      [self _performInitObserverCallbackIfNeeded:newValue];
45
46      [self _performDidChangeObserverCallback:newValue oldValue:oldValue];
47  }
```

- dispatch_block_create：创建的block会被放进队列里，如果直接创建block不会被系统放进队列里
- dispatch_block_cancel：取消队列里的block，可能会失败
- _pendingValue：缓存最新的值

为什么不用value，block会捕获这个value，为什么还要缓存？实验如下：

代码块

```
1   for (int i = 0; i < 10; i++) {
```

```
    dispatch_queue_t queue;
    if (i % 2 == 1) {
        queue = dispatch_get_global_queue(0, 0);
    } else {
        queue = dispatch_get_main_queue();
    }
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(1 * NSEC_PER_SEC)), queue, ^{
        NSLog(@"block %d", i);
    });
}
```

- 代码打印结果是乱序
- 因为dispatch_block_cancel可能会失败，所以还是要防止多线程的情况下dispatch_after的乱序行为
- setValue这个操作可能会在多个线程执行，dispatch_after并不会保证block的执行顺序，所以缓存value的值，保证最后不管执行的block是谁，都能拿到正确的值

优化：Observable虽然能够很好的实现对对象的观察，但是却破坏了类的封装性，考虑以下情况

代码块

```
@property (nonatomic, strong, readonly) XMObservable<NSDate *> *selectedDate;
```

- 是的，即使已经被修饰为ro，外部还是可以setValue

解决方案

代码块

```
@protocol XMObservableReadonly <NSObject>

- (void)addObserver:(id)observer
           forEvent:(XMObservableEventType)type
           callback:(XMObservableCallback)block;
- (void)addObserver:(id)observer
           callback:(XMObservableCallback)block;
- (void)removeObserver:(id)observer forEvent:(XMObservableEventType)type;
- (id)xm_getValue;

@end

- (id<XMObservableReadonly>)selectedDateObservable;
```

- 通过协议，外部将只能调用ro相关的方法

1.2.3 UIResponder+EventHandler

- 在开发寻梦记账的时候，经常碰到block多层传递的情况，场景如下（箭头表示持有）

viewController -> subview -> subviewA

这个时候，子类的子类需要传递事件，将会嵌套block，事件多的时候管理非常麻烦

- UIResponder+EventHandler：冒泡传递事件，自动寻找事件响应者
- 优势
  - 业务解藕，响应者对注册无感知，使用runtime机制，自动维护事件表，响应事件
  - 无需中间人传递block或delegate
  - 无需维护一大堆block，自动查找响应者
- 劣势
  - 需要维护命名表
  - 会比block略慢
  - 参数过多的情况下只能通过userInfo传递参数
- 可能的问题
  - 不要在子线程调用xm_router方法，可能会出现资源竞争
  - 不要在响应链的多个可能的responder响应同一个方法，会发生事件拦截

代码块

```
#define TO_NSSTRING(x) @#x

#define XM_EVENT_HANDLER(SELECTOR_SUFFIX) \
    - (void)handleEvent_##SELECTOR_SUFFIX:(id)data

#define XM_EVENT_HANDLER_PREFIX_STRING @"handleEvent_"

NS_ASSUME_NONNULL_BEGIN

/**
 @brief 注册事件，沿着UIResponder链传递，自动寻找接收者
 */
@interface UIResponder (EventHandler)

- (void)xm_routerEvent:(NSString *)eventName data:(nullable id)data;

@end

```

```objc
NS_ASSUME_NONNULL_END

@implementation UIResponder (EventHandler)

- (void)xm_routerEvent:(NSString *)eventName data:(id)data {
    SEL sel = [self.eventStrategy[eventName] pointerValue];
    if (sel && [self respondsToSelector:sel]) {
        ((void (*)(id, SEL, id))objc_msgSend)(self, sel, data);
        return;
    }
    [[self nextResponder] xm_routerEvent:eventName data:data];
}

- (NSDictionary *)eventStrategy {
    NSDictionary *dict = objc_getAssociatedObject(self, @selector(eventStrategy));
    if (!dict) {
        NSMutableDictionary *result = [NSMutableDictionary dictionary];
        unsigned int methodCount = 0;
        Method *methodList = class_copyMethodList([self class], &methodCount);
        for (unsigned int i = 0; i < methodCount; i++) {
            SEL sel = method_getName(methodList[i]);
            NSString *methodName = NSStringFromSelector(sel);
            if ([methodName hasPrefix:XM_EVENT_HANDLER_PREFIX_STRING]) {
                NSString *eventName = [methodName substringFromIndex:XM_EVENT_HANDLER_PREFIX_STRING.length];
                if ([eventName hasSuffix:@":"]) {
                    eventName = [eventName substringToIndex:eventName.length-1];
                }
                result[eventName] = [NSValue valueWithPointer:sel];
            }
        }
        free(methodList);
        dict = result.copy;
        objc_setAssociatedObject(self, @selector(eventStrategy), dict, OBJC_ASSOCIATION_COPY_NONATOMIC);
    }
    return dict;
}

- (void)setEventStrategy:(NSDictionary *)eventStrategy {
    objc_setAssociatedObject(self, @selector(eventStrategy), eventStrategy, OBJC_ASSOCIATION_COPY_NONATOMIC);
}

@end
```

- 经过实验，有如下关系ViewController -> TestViewController -> TestView，点击TestView，在事件里打印self，实验的时候把return语句注释了
  - 响应链顺序：AppDelegate -> UIApplication -> UINavigationController -> navigationController的子类 -> ViewController -> TestViewController -> TestView
- eventStrategy：响应表，懒加载创建，通过AssociatedObject挂载到响应者上，和业务解藕
- NSValue封装SEL：SEL本质上是指向objc_selector的指针，无法存储到NSDictionary中

1.2.4 后续还能做什么？

- 数据结构：使用CFDictionary和CFArray，但是不能替代NSMapTable

1.3 Context

在业务组件化的设计里，BaseController掌握着上下文所有的细节，包括ViewModel，Manager，EventBus等，Component没必要知道BaseController接口的全部细节，甚至没必要知道BaseController到底是谁，只需要知道它是一个有自己需要的东西的ViewController（UIViewController<Context>）

代码块

```
@class XMDashboardViewModel;
@protocol XMDashboardComponent;

@protocol XMComponentContext <NSObject>

@property (nonatomic, strong, readonly) XMDashboardViewModel *dashboardViewModel;
@property (nonatomic, strong, readonly) UIView *containerView;

- (id<XMDashboardComponent>)componentWithClass:(Class)componentClass;

@end

@implementation XMDashboardViewController (Context)

- (XMDashboardViewModel *)dashboardViewModel {
    return _viewModel;
}

- (id<XMDashboardComponent>)componentWithClass:(Class)componentClass {
    return [_componentManager componentWithClass:componentClass];
}

- (UIView *)containerView {
```

```
24        return self.view;
25    }
26
27    @end
```

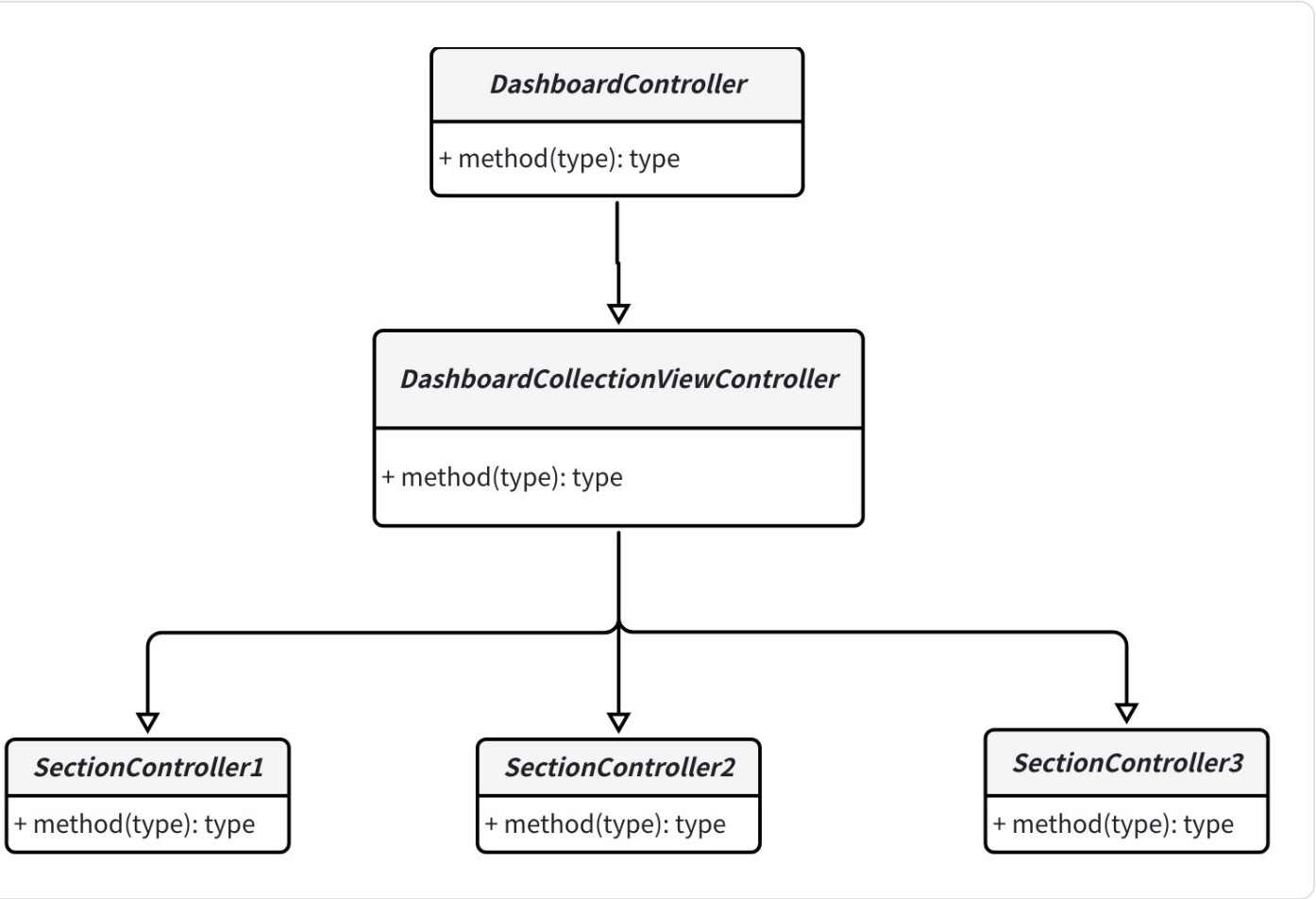
- componentWithClass：假设一个component的继承层级依赖于另一个component实现的view/controller，可以这样获取

1.4 更适合的架构

寻梦记账看板实际上更加适合MVVM+IGListKit的架构，IGListKit中实现的ListSectionController本质上就是一种component的结构。

之所以设计成components而不是MVVM+IGListKit，是因为寻梦记账项目本身不可公开，所以想通过公开看板项目的方式来体现我个人一直在写代码和学习，在日后的工作中也能够快速上手，写出较为优质的代码

其实这个项目有更加简单/合适的架构方式



- DashboardController：负责其它控件的展示和事件逻辑
- CollectionViewController+IGListKit：负责UICollectionView相关模块的逻辑

	优点	缺点

业务组件化	解藕更加彻底，模块清晰，可插拔	结构略微复杂，初期开发成本较高
MVVM+IGListKit	更加轻量级，易于开发	业务复杂后耦合度高，很容易出现巨大的 controller