无感知记账

无感知记账就是在记账或者数据产生的时候，没有loading+没有卡顿，在后台同步数据，无网络的情况下缓存数据，有网络之后再进行上传，用户感知不到数据同步，通过技术方案优化产品体验。

无感知记账并不是寻梦记账独创的，像竞品鲨鱼记账等记账软件都有类似的功能，但是网上没有公开的技术方案，需要自己「设计和实现」

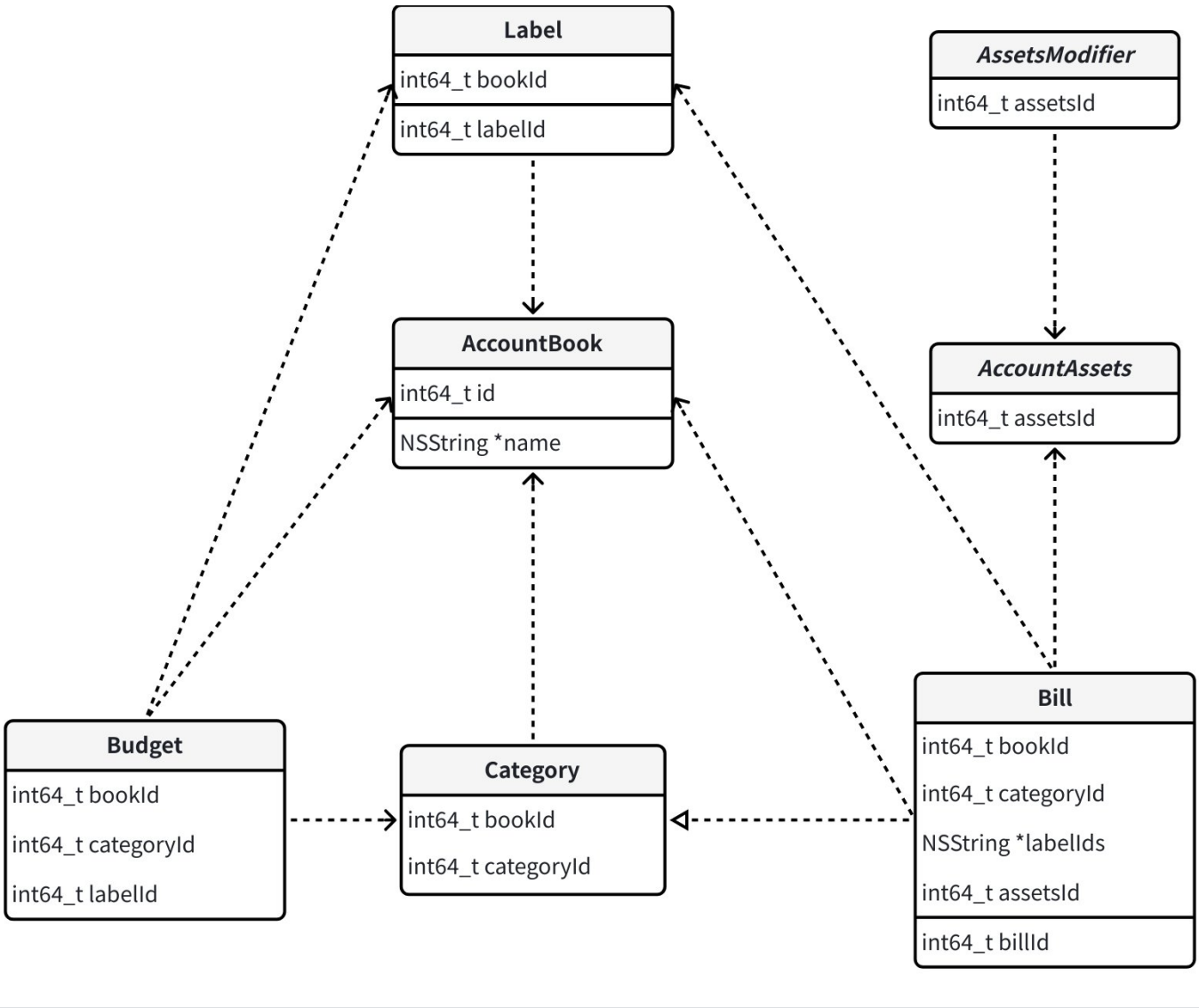
思路

数据操作实际上只有四种，增删改查，而「无感知记账」最重要的问题就是同步id

- 查：寻梦记账在登录的时候查询入库，后续不再做任何查询操作（不再查询服务端数据）
- 增：优先级最高，同步成功后会替换相关对象的id
- 改：不涉及id变化，优先级较低，有网同步即可
- 删：优先级最低，已经同步服务端的数据缓存id，网络恢复后同步；未同步服务端的数据直接删除

数据之间的关系

无感知记账的难度就在于如何正确的同步和更新id，id就是不同model之间的关系，寻梦记账中重要的model之间的关系如图：



- 指向的线最多，说明数据同步最重要，依赖于它的id变化的对象就更多，需要优先进行同步
- 线最少，依赖越少，可以看到budget、modeifier、bill没有任何依赖，优先级最低，可以放在最后进行同步
- 实际上同步先后并不影响准确性，正确的顺序影响的是同步的次数，而不是准确性
  - 同步次数：假设先同步bill，再同步book，bills中的bookId还是可以被正确同步只不过需要再次update，也就是多了一次请求

标记状态

如何知道model的状态？什么样的状态需要标记？不同的状态处理有什么区别？

代码块

```
@interface Bill : NSObject <NSCopying, AccountAssetsDetailsItemDelegate>

@property (nonatomic, assign) int64_t billId; /// 自动填
@property (nonatomic, assign) int64_t categoryId; /// 必填
@property (nonatomic,   copy) NSString *notes; /// 非必填
```

```objc
@property (nonatomic, assign) NSTimeInterval timeInterval; /// 默认当天
@property (nonatomic, assign) NSTimeInterval creationTime; /// 自动填
@property (nonatomic, strong, nullable) NSNumber *amount; /// 必填
@property (nonatomic,   copy, nullable) NSString *labelIds;
@property (nonatomic, assign) int64_t bookId;
@property (nonatomic, assign) BOOL liked; // 已废弃
@property (nonatomic, assign) BOOL deleted;
@property (nonatomic, assign) NSTimeInterval deletedTimeInterval;
@property (nonatomic, assign) NSTimeInterval likedTimeInterval; // 已废弃
@property (nonatomic, assign) BOOL autoRecord;
@property (nonatomic, assign) int64_t assetsId; // 当前默认值是-1，可以优化为null
@property (nonatomic, assign) FeaturesSyncStatus syncStatus; // 是否已经同步到服务端
@property (nonatomic, assign) BOOL notIncludeInBudget;
@property (nonatomic, assign) BOOL notIncludeInSummary;
@property (nonatomic,   copy) NSString *remark;

@property (nonatomic, copy) NSString *notesPinyin;
@property (nonatomic, copy) NSString *notesPinyinPrefix;
@property (nonatomic, copy) NSString *remarkPinyin;
@property (nonatomic, copy) NSString *remarkPinyinPrefix;
@property (nonatomic, assign) int64_t originId;

@end

typedef NS_ENUM(NSInteger, FeaturesSyncStatus) {
    FeaturesSyncStatusNormal = 0,     // 正常同步
    FeaturesSyncStatusNeedInsert = 1, // 需要insert
    FeaturesSyncStatusNeedUpdate = 2, // 需要update
};
```

账单的数据结构，记录了一笔账的详细信息；其它的数据结构类似，都有一个syncStatus

- syncStatus：同步状态，分为normal、needInsert、needUpdate，在insert/update的时候标记，网络请求成功的回调后改成normal
- 删除：删除没有状态，不进行任何标记，而是把id缓存到NSUserDefaults里，如果对象的syncStatus是insert，说明还没有上传到服务端，直接删除
- 不同状态同步的时候会有细微的差别，之后会说明

数据同步如何保证正确性

客户端使用的数据库是SQLite，服务端使用的数据库是MySQL，主键也就是id都是自增主键，自增主键的变化原则如下

- 每个表维护一个自增计数器，默认从1开始，每插入行自增1

- 主键删除后，id不会回填，也就是不会因为删除而回退
- 如果插入时手动指定id，且比当前自增值大，则自增计数器自动跳到更大的值
- 重启数据库后，自增计数器默认以表中最大id+1为准
- 并发插入时，有“gap”，不是绝对连续，gap的意思就是一个并发的id将会被跳过
  - 假如自增主键现在是100：

  a. 线程A、B、C并发插入，分别获得ID 101、102、103。

  b. 线程B插入回滚，A和C插入成功。

  c. 此时表里的ID变成：101, 103（**102缺失**）。

  d. 下次插入会分配104，不会回到102。

SQLite和MySQL数据库自增主键行为对比如下

	MySQL AUTO_INCREMENT	SQLite AUTOINCREMENT
自增起始值	1，除非手动指定	1（或最大值+1，绝不回收）
删除后复用	不会复用	永不复用
手动插入大id	后续自增会从更大值继续	同左
并发/回滚	可能跳号（id丢失）	可能跳号
计数器持久化	是	是，维护在sqlite_sequence表

也就是说MySQL和SQLite在自增主键上行为保持一致

- 因为客户端和数据库表的主键也就是id都是自增的，且双方数据库主键自增的行为保持一致，所以服务端表的id一定大于等于客户端的id
- SQLite数据库在服务端返回的数据进行insert的时候，表里已经存在相同id的行会直接崩溃

以同步insert的分类为例

代码块

```
1  + (void)syncInsertCategoryListWithCompletion:(void(^)(BOOL
   returnedDueToEmptyList, KTBaseResponseObject *rs, NSError *error))completion {
2      NSArray *insertCategoryList = [[[CategoryDatabaseManager sharedInstance]
   queryAllBillCategories] kt_select:^BOOL(BillCategory *category) {
3          return category.syncStatus == FeaturesSyncStatusNeedInsert;
4      }];
5
6      if (CollectionsUtils.isEmpty(insertCategoryList)) {
7          XMLog(@"[SyncManager] no categories need insert");
8          SafeBlock(completion, YES, nil, nil);
9      } else {
10         [CategoryRequestManager requestInsertCategoryList:insertCategoryList
   completion:^(KTBaseResponseObject *rs, CategoryListResponseObject *categoryRS,
```

```
    NSError *error) {
11              if (rs.code == CommonResponseResultSuccess) {
12                  NSArray *categoryListAfterUpdate = [categoryRS.categoryList
    kt_safe_map:^id(id obj) {
13                      return [BillCategory categoryWithResponseObject:obj];
14                  }];
15                  NSSet *deletedIdSet = [CategoryManager
    sharedInstance].getDeletedIdSet;
16                  NSMutableArray *finalInsertionList = [NSMutableArray array];
17                  NSMutableArray *finalUpdatedList = [NSMutableArray array];
18                  [insertCategoryList enumerateObjectsUsingBlock:^(BillCategory
    *category, NSUInteger idx, BOOL *stop) {
19                      if (![deletedIdSet containsObject:@(category.categoryId)])
    {
20                          BillCategory *finalUpdatedCategory =
    [categoryListAfterUpdate safe_objectAtIndex:idx];
21                          [finalInsertionList safe_addObject:category];
22                          [finalUpdatedList safe_addObject:finalUpdatedCategory];
23                      }
24                  }];
25                  NSArray *fromIds = [finalInsertionList
    kt_safe_map:^id(BillCategory *category) {
26                      return @(category.categoryId);
27                  }];
28                  NSArray *toIds = [finalUpdatedList
    kt_safe_map:^id(BillCategory *category) {
29                      return @(category.categoryId);
30                  }];
31                  [[CategoryManager sharedInstance]
    updateInsertionIdMapWithKeys:fromIds values:toIds];
32                  [BudgetManager updateCategoryIdFromIds:fromIds toIds:toIds];
33                  [BillManager updateWhenCategoryIdsChangeFrom:fromIds
    toIds:toIds];
34                  [finalInsertionList enumerateObjectsUsingBlock:^(BillCategory
    *oldCategory, NSUInteger idx, BOOL *stop) {
35                      [[CategoryDatabaseManager sharedInstance]
    deleteBillCategory:oldCategory completion:nil];
36                  }];
37                  [finalUpdatedList kt_each:^(BillCategory *obj, NSUInteger idx)
    {
38                      obj.syncStatus = FeaturesSyncStatusNormal;
39                      [[CategoryDatabaseManager sharedInstance]
    insertBillCategoryWithCategoryId:obj];
40                  }];
41                  XMLog(@"[SyncManager] success to insert categories %@",
    finalUpdatedList);
42              } else {
```

```
                XMLog(@"[SyncManager] insert categories error %lu",
    error.code);
            }
            SafeBlock(completion, NO, rs, error);
        }];
    }
}
```

- 网络请求回调后，先删除旧的，再插入新的，防止oldId == newId造成的崩溃
- 不同状态的处理差异：根据fromId也就是oldId找到需要变化的关联对象，把关联对象的categoryId改成newId，如果该对象的状态为normal，把关联对象的状态改成needUpdate，其它状态不做处理

这一行代码比较特殊，后面再分析

代码块

```
[[CategoryManager sharedInstance] updateInsertionIdMapWithKeys:fromIds
values:toIds];
```

SourceManager

寻梦记账中，所有可以被UINavigationController Push的页面都继承自KTBaseViewController，这个baseController对navigationController的一些行为做了统一的处理，还有就是做了埋点、通用UI之类的事情

代码块

```
@interface SourceManager ()

@property (nonatomic, weak) UIViewController *currentController;

@end

@implementation SourceManager

+ (instancetype)sharedInstance {
    static SourceManager *sharedInstance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}

```

```
@end
```

- KTBaseViewController：多数页面的基类，做一些通用的事情，其中就包括，在viewDidAppear的时候更新sourceManager

代码块

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];

    [[SourceManager sharedInstance] updateCurrentController:self];
}
```

同步完成后

代码块

```
+ (void)syncModelsIfNeededWithCompletion:(void(^)(BOOL didSync))completion {
    if ([self canPerformSync] == NO) {
        SafeBlock(completion, NO);
        return;
    }
    BOOL shouldSync = [KittenUserManager sharedInstance].didLogin && [KittenUserManager sharedInstance].isGuest == NO;
    if (isSyncing || !shouldSync) {
        SafeBlock(completion, NO);
        return;
    }
    isSyncing = YES;
    [KTNetworkManager checkNetworkReachabilityWithCompletion:^(BOOL isReachable) {
        if (isReachable) {
            dispatch_group_t group = dispatch_group_create();

            dispatch_group_enter(group);
            [self syncDeleteModelsIfNeededWithCompletion:^{
                dispatch_group_leave(group);
            }];

            dispatch_group_enter(group);
            [self syncInsertModelsIfNeededWithCompletion:^{
                [self syncUpdateModelsIfNeededWithCompletion:^{
                    dispatch_group_leave(group);
                }];
            }];
```

```
27
28              dispatch_group_notify(group, dispatch_get_main_queue(), ^{
29                  isSyncing = NO;
30                  SafeBlock(completion, YES);
31              });
32          } else {
33              isSyncing = NO;
34  #ifdef DEBUG
35              [[SourceManager sharedInstance] showToastWithCurrentController:@"网
    络未连接"];
36  #endif
37          }
38      }];
39  }
```

- dispatch_gorup_t：保证全部同步完成后再通知相关的currentController刷新数据

代码块

```
1       [FeaturesSyncManager syncModelsIfNeededWithCompletion:^(BOOL didSync) {
2           if (didSync) {
3               dispatch_async(dispatch_get_main_queue(), ^{
4                   [[SourceManager sharedInstance] reloadData];
5               });
6
7               [[UIApplication sharedApplication]
    endBackgroundTask:backgroundTaskIdentifier];
8           }
9       }];
```

- 在数据同步完成后，通过sourceManager刷新当前BaseController的数据

不是KTBaseViewController怎么办

很多页面不一定是navigationController控制的，比如编辑页面，很多编辑页面都是半屏页面，并不是KTBaseViewController，这样的页面无法被reloadData，如何保证数据同步后，这些页面数据的一致性

而保证数据一致性的问题本质只有一个，也就是保证关联数据id的一致性，因为其它字段（属性）在记录的时候就已经确定，同步服务端只是根据客户端的数据落表，所以两端是一致的

- 办法也很简单，建立fromId到toId的映射

代码块

```
1   [[CategoryManager sharedInstance] updateInsertionIdMapWithKeys:fromIds
```

```
    values:toIds];
```

在CategoryManager中建立映射

代码块

```
- (void)updateInsertionIdMapWithKeys:(NSArray<NSNumber *> *)keys values:(NSArray<NSNumber *> *)values {
    [_lock lock];
    [keys enumerateObjectsUsingBlock:^(NSNumber *key, NSUInteger idx, BOOL *stop) {
        NSNumber *value = [values safe_objectAtIndex:idx];
        self.insertionIdMap[key] = value;
    }];
    [_lock unlock];
}
```

- CategoryManager是一个单例，因为在Objective-C中，对象被存储在内存中（单例比较特殊，static修饰的对象被存储在数据段，但是还是在内存里），所以insertionIdMap被保存在内存中，和进程的生命周期保持一致，在这一次进程生命周期中出现的同步情况都能正确的映射
  - 当进程被杀死，下一次重新加载的时候，所有已同步model会重新加载，model之间的关系将会被正确建立

获取的时候是这样的

代码块

```
- (NSNumber *)valueForKeyInInsertionMap:(NSNumber *)key {
    id value;
    [_lock lock];
    value = self.insertionIdMap[key];
    [_lock unlock];
    return value;
}

- (BillCategory *)categoryWithId:(int64_t)categoryId {
    BillCategory *category = [self _getById:categoryId];
    if (!category) {
        NSNumber *insertionId = [self valueForKeyInInsertionMap:@(categoryId)];
        if (insertionId) {
            category = [self _getById:insertionId.longLongValue];
        }
    }
    if (!category) {
        NSLog(@"[CategoryManager] 没有找到对应的本地数据 %lld", categoryId);
```

```
    }
    return category;
}

- (BillCategory *)_getById:(int64_t)categoryId {
    [_lock lock];
    NSArray *allCategories = _allCategories.copy;
    [_lock unlock];
    for (BillCategory *category in allCategories) {
        if (category.categoryId == categoryId) {
            return category;
        }
    }
    return nil;
}
```

- 查询category全部通过这个方法，先查本地缓存的数据_allCategories，如果找不到说明id发生了变化，再查insertionIdMap
- 锁：保证一致性，防止遍历过程中被其他线程修改

架构

SourceManager和FeaturesSyncManager全部和业务功能解藕，被拆分到单独的类中，也实现了类的单一职责

同步时机和优化

目前的同步时机是applicationDidFinishLaunch和applicationDidEnterForground，且时间间隔在5min以上

因为这个功能非常基础，开发的也比较早，因为这之后个人技术的成长，在文档总结过程中还是发现了几个可以优化的点，具体如下

锁性能优化

在这里，查询是一个量较大的操作，且性能敏感，可以改成性能更好的dispatch_mutex_t，或者可以直接改成os_unfair_lock，因为寻梦记账项目支持的版本是iOS13，所以os_unfair_lock完全可用

同步过程的优化

可以使用FIFO进行同步，自增主键就相当于一个队列，id从小到大进行同步，限制单次同步的数量，提升同步的频

- 同步数量可以写一个合理的值，写死在客户端，或者也可以通过服务端下发配置
- 同步频率也就是时间，也是同理

可以参考YYCache的实现，以下为优化后的伪代码

代码块

```
+ (void)_syncRecursively {
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW,
_timeInterval*NSEC_PER_SEC)), dispatch_get_global_queue(0, 0)) {
        [self _syncFeaturesData];
        [self _syncRecursively];
    }
}

+ (void)_syncFeaturesData {
    // 根据条件同步数据
}
```