IGListKit源码分析（一）

前言

这篇文章我们来分析IGListKit的基础，主要下下面几个方面

- IGListKit到底到底做了什么
- 架构和业务思考
- 代理
- 防御式编程的思考以及业务落地
- IGListKit的局限性

1. 需要注意的是，如果博客中提到全量刷新，指的是adapter中的方法performUpdates，如果提到局部刷新指的是collectionContext协议中的方法performBatchAnimated
2. 我们在讨论updater的时候，都只讨论IGListAdapterUpdater，不讨论IGListReloadDataUpdater，因为reloadDataUpdater本质上只是一层对[collectionView reloadData]的封装，它几乎所有方法最终都只是调用reloadData

1. 从一个方法总览IGListKit

SDK往往都有最关键的一个方法，看懂了这个方法就算是初步入门这个SDK的源码了，而IGListKit也有这样的方法

IGListKit除了它高度解藕的sectionController，最为重要的就是它的diff算法了，reloadData的入口就是这里，IGListKit会使用diff算法让刷新最小化，还会按照业务定制将多次全量刷新合并成一次

代码块

```
- (void)performUpdatesAnimated:(BOOL)animated completion:(IGListUpdaterCompletion)completion {
    IGAssertMainThread();

    id<IGListAdapterDataSource> dataSource = self.dataSource;
    id<IGListUpdatingDelegate> updater = self.updater;
    UICollectionView *collectionView = self.collectionView;
    if (dataSource == nil || collectionView == nil) {
        IGLKLog(@"Warning: Your call to %s is ignored as dataSource or collectionView haven't been set.", __PRETTY_FUNCTION__);
        IGLK_BLOCK_CALL_SAFE(completion, NO);
        return;
    }

```

```
13      [self _enterBatchUpdates];
14
15      __weak __typeof__(self) weakSelf = self;
16      IGListTransitionDataBlock sectionDataBlock = ^IGListTransitionData *{
17          __typeof__(self) strongSelf = weakSelf;
18          IGListTransitionData *transitionData = nil;
19          if (strongSelf) {
20              NSArray *toObjects =
    objectsWithDuplicateIdentifiersRemoved([dataSource
    objectsForListAdapter:strongSelf]);
21              transitionData = [strongSelf
    _generateTransitionDataWithObjects:toObjects dataSource:dataSource];
22          }
23          return transitionData;
24      };
25
26      IGListTransitionDataApplyBlock applySectionDataBlock =
    ^void(IGListTransitionData *data) {
27          __typeof__(self) strongSelf = weakSelf;
28          if (strongSelf) {
29              // temporarily capture the item map that we are transitioning from
    in case
30              // there are any item deletes at the same
31              strongSelf.previousSectionMap = [strongSelf.sectionMap copy];
32              [strongSelf _updateWithData:data];
33          }
34      };
35
36      IGListUpdaterCompletion outerCompletionBlock = ^(BOOL finished){
37          __typeof__(self) strongSelf = weakSelf;
38          if (strongSelf == nil) {
39              IGLK_BLOCK_CALL_SAFE(completion,finished);
40              return;
41          }
42
43          // release the previous items
44          strongSelf.previousSectionMap = nil;
45          [strongSelf _notifyDidUpdate:IGListAdapterUpdateTypePerformUpdates
    animated:animated];
46          IGLK_BLOCK_CALL_SAFE(completion,finished);
47          [strongSelf _exitBatchUpdates];
48      };
49
50      [updater performUpdateWithCollectionViewBlock:[self _collectionViewBlock]
51                                           animated:animated
52                                 sectionDataBlock:sectionDataBlock
53                            applySectionDataBlock:applySectionDataBlock
```

```
54                                                  completion:outerCompletionBlock];
55  }
```

- _enterBatchUpdates：初始化更新队列，代码自解释
- 接下来的三个block根据命名
  a. 生成一次新的事务要用的section/数据的diff信息
  b. 把生成的diff数据真正应用到UI层（提交变更）
  c. 整个数据变更/动画全部完成后调用

这三个block非常重要，先来看看具体实现

1.1 sectionDataBlock

因为代码太长，所以删除一些防御式编程的代码

代码块

```
1   - (IGListTransitionData *)_generateTransitionDataWithObjects:(NSArray
    *)objects dataSource:(id<IGListAdapterDataSource>)dataSource {
2       IGListSectionMap *map = self.sectionMap;
3
4       NSMutableArray<IGListSectionController *> *sectionControllers =
    [[NSMutableArray alloc] initWithCapacity:objects.count];
5       NSMutableArray *validObjects = [[NSMutableArray alloc]
    initWithCapacity:objects.count];
6
7       // push the view controller and collection context into a local thread
    container so they are available on init
8       // for IGListSectionController subclasses after calling [super init]
9       IGListSectionControllerPushThread(self.viewController, self);
10
11      [objects enumerateObjectsUsingBlock:^(id object, NSUInteger idx, BOOL
    *stop) {
12          // infra checks to see if a controller exists
13          IGListSectionController *sectionController = [map
    sectionControllerForObject:object];
14
15          // if not, query the data source for a new one
16          if (sectionController == nil) {
17              sectionController = [dataSource listAdapter:self
    sectionControllerForObject:object];
18          }
19
20          if (sectionController == nil) {
```

```
            IGLKLog(@"WARNING: Ignoring nil section controller returned by data source %@ for object %@.",
                    dataSource, object);
            return;
        }

        if ([sectionController isMemberOfClass:[IGListSectionController class]]) {
            IGFailAssert(@"Ignoring IGListSectionController that's not a subclass from data source %@ for object %@", NSStringFromClass([dataSource class]), NSStringFromClass([object class]));
            return;
        }

        // in case the section controller was created outside of -listAdapter:sectionControllerForObject:
        sectionController.collectionContext = self;
        sectionController.viewController = self.viewController;

        [sectionControllers addObject:sectionController];
        [validObjects addObject:object];
    }];

    // clear the view controller and collection context
    IGListSectionControllerPopThread();

    return [[IGListTransitionData alloc] initFromObjects:map.objects
                                               toObjects:validObjects
                                  toSectionControllers:sectionControllers];
}
```

1. 把当前的viewController推入mainThread的堆栈，方便之后sectionController创建的时候能够正确取到
2. 根据object添加或创建sectionController，并存储有效的object和sectionController
3. 出栈，返回结果

这之前还有一个步骤，就是会对objects根据diffable协议进行去重，把去重后的objects当作参数传入这个方法，然后返回最后的结果

1.2 applySectionDataBlock

同样删除防御式编程的代码后

代码块

```
- (void)_updateWithData:(IGListTransitionData *)data {
```

```
    // Should be the first thing called in this function.
    _isInObjectUpdateTransaction = YES;

    IGListSectionMap *map = self.sectionMap;

    // Note: We use an array, instead of a set, because the updater should have dealt with duplicates already.
    NSMutableArray *updatedObjects = [NSMutableArray new];

    for (id object in data.toObjects) {
        // check if the item has changed instances or is new
        const NSInteger oldSection = [map sectionForObject:object];
        if (oldSection == NSNotFound || [map objectForSection:oldSection] != object) {
            [updatedObjects addObject:object];
        }
    }

    [map updateWithObjects:data.toObjects sectionControllers:data.toSectionControllers];

    // now that the maps have been created and contexts are assigned, we consider the section controller "fully loaded"
    for (id object in updatedObjects) {
        [[map sectionControllerForObject:object] didUpdateToObject:object];
    }

    [self _updateBackgroundView];

    // Should be the last thing called in this function.
    _isInObjectUpdateTransaction = NO;
}
```

1. 标记正在进行更新事务
2. 根据object顺序如果不存在或者不相等添加进数据，这里要注意的是updatedObjects并不参与map的变化，只是用来通知sectionController，让刷新最小化
3. 更新sectionMap，在这里会重置map，重新设置key-value，让section和controller关联，让controller和object关联
    a. 这里会给section、isFirstSection、isLastSection赋值
4. 通知所有的sectionController数据已经更新了
5. 更新事务结束

1.3 outerCompletionBlock

代码块

```
    IGListUpdaterCompletion outerCompletionBlock = ^(BOOL finished){
        // release the previous items
        strongSelf.previousSectionMap = nil;
        [strongSelf _notifyDidUpdate:IGListAdapterUpdateTypePerformUpdates
animated:animated];
        IGLK_BLOCK_CALL_SAFE(completion,finished);
        [strongSelf _exitBatchUpdates];
    };
```

1. 释放previousSectionMap，通知listener更新，调用block，退出update

现在就只剩下一行代码，也就是updater的执行更新，业务中传入的updater一般都是IGListAdapterUpdater，所以还是从这里入手

1.4 IGListAdapterUpdater

updater是真正处理更新的实例，删除防御式编程的代码后可以看到，这里有一个transactionBuilder接收了所有的block，然后执行了一个方法

代码块

```
- (void)performUpdateWithCollectionViewBlock:
(IGListCollectionViewBlock)collectionViewBlock
                                    animated:(BOOL)animated
                                sectionDataBlock:
(IGListTransitionDataBlock)sectionDataBlock
                            applySectionDataBlock:
(IGListTransitionDataApplyBlock)applySectionDataBlock
                                  completion:(nullable
IGListUpdatingCompletion)completion {
    [self.transactionBuilder addSectionBatchUpdateAnimated:animated
                                      collectionViewBlock:collectionViewBlock
                                         sectionDataBlock:sectionDataBlock

applySectionDataBlock:applySectionDataBlock
                                                completion:completion];

    [self _queueUpdateIfNeeded];
}
```

- addSectionBatchUpdateAnimated：这个方法很简单，就是做了一些对应的判断，然后使用属性持有了这些block

Builder

代码块

```
- (void)addSectionBatchUpdateAnimated:(BOOL)animated
                   collectionViewBlock:(IGListCollectionViewBlock)collectionViewBlock
                      sectionDataBlock:(IGListTransitionDataBlock)sectionDataBlock
                  applySectionDataBlock:(IGListTransitionDataApplyBlock)applySectionDataBlock
                             completion:(IGListUpdatingCompletion)completion {
    self.mode = MAX(self.mode, IGListUpdateTransactionBuilderModeBatchUpdate);

    self.animated = self.animated && animated;
    self.collectionViewBlock = collectionViewBlock;

    self.sectionDataBlock = sectionDataBlock;

    self.applySectionDataBlock = applySectionDataBlock;

    IGListUpdatingCompletion localCompletion = completion;
    if (localCompletion) {
        [self.completionBlocks addObject:localCompletion];
    }
}
```

- 可以看到builder收集了上下文的blocks，最重要的是它给mode进行了判断赋值，优先级问题在下篇讨论，这里只说结论
  - 如果不配置coalescer，IGListKit不会合并更新，因为更新是在主线程上的，所以它们会按照顺序进行，这个mode将不会对更新造成任何影响
  - 如果是合并更新的，changeDataSource > reloadData > batchUpdate，在coalescer决定执行update的时候，会根据mode创建优先级最高的transaction进行更新
- 分析一下三个block的时序，首先这几个block会在updater真正执行更新的时候被调用，具体取决于coalescer接收到的策略
  - sectionDataBlock：这个block返回了当前sectionData去重之后的对象，它会被updater用来做diff
  - applySectionBlock：_applyDiff的时候被performUpdates调用（collectionView的更新方法）；在reloadData的时候在collectionView执行更新前被调用
  - completionBlock：会在更新最后被调用
- 最后再分析一下applySecitonBlock、sectionData、diffResult的关系
  - applySectionBlock：使用sectionData为建立object - sectionController之间的映射，本质上是adpater中的数据源，在collectionViewDataSource中更新数据

- diffResult：这是对sectionData执行diffKit之后的结果，包括deletes、inserts、move关系，用它来指挥collectionView做最小更新动画，告知collectionView每个cell应该去什么地方

applySectionBlock的本质就是更新sectionMap，而sectionMap的本质就是数据源，代码如下：

代码块

```
- (NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView {
    _assertNotInMiddleOfObjectUpdate(self.isInObjectUpdateTransaction);
    return self.sectionMap.objects.count;
}
```

1.4.1 Coalescer的意义

coalescer是调度的意思，下面是config字段的含义

字段	中文含义	实际用途与意义
enabled	启用自适应合并	是否用动态收敛批量更新策略
minInterval	最小间隔	合并窗口的下限，限制太频繁批量刷新
intervalIncrement	间隔递增	高频操作时动态增加合并窗口，防止过度刷新
maxInterval	最大间隔	合并窗口上限，保证不会长时间不刷新
useMaxIntervalWhenViewNotVisible	不可见时用最大间隔	界面不可见时合并窗口强制用最大值，省性能

- minInterval：两个update时间间隔小于这个值，就不执行update，等一会收敛更多变更再执行
- intervalIncrement：如果有持续的update请求，每次批量合并时，下一次的间隔会递增这个值，形成自适应加长合并窗口
- maxInterval：最长延迟，不管后面有没有新的update，请求合并窗口不会超过这个时间，保证不会卡着不刷新

调节器可以根据配置，进行延迟刷新，如果没有配置就会直接调用updater的更新方法

来具体分析一下coalescer是如何合并多次更新的

代码块

```
- (void)_adaptiveDispatchUpdateForView:(nullable UIView *)view {
    const IGListAdaptiveCoalescingExperimentConfig config = _adaptiveCoalescingExperimentConfig;
    const NSTimeInterval timeSinceLastUpdate = -[_lastUpdateStartDate timeIntervalSinceNow];
    const BOOL isViewVisible = _isViewVisible(view, config);
    const NSTimeInterval currentCoalescenceInterval = _coalescenceInterval;

    if (isViewVisible) {
```

```
        if (!_lastUpdateStartDate || timeSinceLastUpdate > currentCoalescenceInterval) {
            _coalescenceInterval = config.minInterval;
            [self _performUpdate];
            return;
        } else {
            _coalescenceInterval = MIN(currentCoalescenceInterval + config.intervalIncrement, config.maxInterval);
        }
    }

    const NSTimeInterval remainingTime = isViewVisible ? (currentCoalescenceInterval - timeSinceLastUpdate) : config.maxInterval;
    const NSTimeInterval remainingTimeCapped = MAX(remainingTime, 0);

    _hasQueuedUpdate = YES;
    __weak __typeof__(self) weakSelf = self;
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(remainingTimeCapped * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
        [weakSelf _performUpdate];
    });
}
```

我们假设view是可见的，如果不可见就是会延迟maxInterval后在主线程performUpdate

1. 如果之前没有更新过直接更新
2. 如果上次更新的间隔 > _coalescenceInterval就直接更新
3. _coalescenceInterval每次更新后被初始化为最小间隔，每次递增intervalIncrement，当递增超过maxInterval的时候就是maxInterval
4. 其实就是一个窗口，每次递增这个[minInterval, _coalescenceInterval]的窗口，只有timeSinceLastUpdate大于这个窗口值的时候会直接更新其它情况会延迟更新
5. 计算出延迟更新的时间戳，执行更新

到这里我们会发现，coalescer并没有合并多次更新，只是首次更新以后会维护一个递增窗口，上次更新时间大于这个窗口的最大值，才会直接更新，不然就延迟更新

所以合并更新的逻辑在什么地方呢?

代码块

```
- (void)performDataSourceChange:(IGListDataSourceChangeBlock)block {
    if (!self.transaction
        && ![self.transactionBuilder hasChanges]
        && !IGListExperimentEnabled(self.experiments, IGListExperimentRemoveDataSourceChangeEarlyExit)) {
```

```
        block();
        return;
    }

    IGListUpdateTransactionBuilder *builder = [IGListUpdateTransactionBuilder new];
    [builder addDataSourceChange:block];

    if ([self.transaction cancel] && self.lastTransactionBuilder) {
        [builder addChangesFromBuilder:(IGListUpdateTransactionBuilder *)self.lastTransactionBuilder];
    }

    [builder addChangesFromBuilder:self.transactionBuilder];

    self.transaction = nil;
    self.lastTransactionBuilder = nil;
    self.transactionBuilder = builder;

    [self update];
}
```

- 如果在更新中，先退出等待当前更新完成后的queueUpdateIfNeeded
- 如果能取消(还没有执行_didDiff)，就合并之前的builder一块更新
- 可以看到只有dataSourceChange相当于直接update，不直接走coaclescer，但是如果被queueUpdateIfNeeded了还是会走

代码块

```
- (void)addSectionBatchUpdateAnimated:(BOOL)animated
                   collectionViewBlock:(IGListCollectionViewBlock)collectionViewBlock
                      sectionDataBlock:(IGListTransitionDataBlock)sectionDataBlock
                 applySectionDataBlock:(IGListTransitionDataApplyBlock)applySectionDataBlock
                            completion:(IGListUpdatingCompletion)completion {
    self.mode = MAX(self.mode, IGListUpdateTransactionBuilderModeBatchUpdate);

    self.animated = self.animated && animated;
    self.collectionViewBlock = collectionViewBlock;

    self.sectionDataBlock = sectionDataBlock;

    self.applySectionDataBlock = applySectionDataBlock;
```

```
14
15      IGListUpdatingCompletion localCompletion = completion;
16      if (localCompletion) {
17          [self.completionBlocks addObject:localCompletion];
18      }
19  }
```

当调用reloadData、performUpdates、performBatchAnimated的时候，都会先添加一个transactionBuilder，而它就是合并更新的原因

- mode：多次更新取优先级最高的
- animated：多次更新必须都是animated才做动画

因为dispatch_after相当于在主线程dispatch_async，当time到了的时候，会等runLoop空闲的时候执行尝试queueUpdate

或者不使用coalescer也可能会出现两次update时序很接近的情况

如果多次更新排的很近，会返回并合并到同一个builder里，等当前更新执行完之后会调用

代码块

```
1   - (void)update {
2       if (![self.transactionBuilder hasChanges]) {
3           return;
4       }
5
6       if (self.transaction && self.transaction.state !=
    IGListBatchUpdateStateIdle) {
7           return;
8       }
9       // 其它逻辑
10      {
11          [transaction addCompletionBlock:^(BOOL finished) {
12          __typeof__(self) strongSelf = weakSelf;
13          if (strongSelf == nil) {
14              return;
15          }
16          if (strongSelf.transaction == weakTransaction) {
17              strongSelf.transaction = nil;
18              strongSelf.lastTransactionBuilder = nil;
19
20              [strongSelf _queueUpdateIfNeeded];
21          }
22      }];
23  }
```

- 当前transaction执行完会在队列中尝试更新，这个时候如果有多次更新，会被builder合并成一次
  - 多次更新取优先级最高的
  - 它们的更新blocks会被builder收集，在合适的时机调用（比如reloadDataTransaction就会在collectionView更新前先调用所有的itemUpdates，如果有的话）

1.4.2 Update

代码块

```
- (void)update {
    if (![self.transactionBuilder hasChanges]) {
        return;
    }

    if (self.transaction && self.transaction.state != IGListBatchUpdateStateIdle) {
        return;
    }

    IGListUpdateTransactationConfig config = (IGListUpdateTransactationConfig){
        .sectionMovesAsDeletesInserts = _sectionMovesAsDeletesInserts,
        .singleItemSectionUpdates = _singleItemSectionUpdates,
        .preferItemReloadsForSectionReloads = _preferItemReloadsForSectionReloads,
        .allowsReloadingOnTooManyUpdates = _allowsReloadingOnTooManyUpdates,
        .allowsBackgroundDiffing = _allowsBackgroundDiffing,
        .experiments = _experiments,
        .adaptiveDiffingExperimentConfig = _adaptiveDiffingExperimentConfig,
    };

    id<IGListUpdateTransactable> transaction = [self.transactionBuilder buildWithConfig:config delegate:_delegate updater:self];
    self.transaction = transaction;
    self.lastTransactionBuilder = self.transactionBuilder;
    self.transactionBuilder = [IGListUpdateTransactionBuilder new];

    if (!transaction) {
        // If we don't have enough information, we might not be able to create a transaction.
        self.lastTransactionBuilder = nil;
        return;
    }

    __weak __typeof__(self) weakSelf = self;
    __weak __typeof__(transaction) weakTransaction = transaction;
```

```
33      [transaction addCompletionBlock:^(BOOL finished) {
34          __typeof__(self) strongSelf = weakSelf;
35          if (strongSelf == nil) {
36              return;
37          }
38          if (strongSelf.transaction == weakTransaction) {
39              strongSelf.transaction = nil;
40              strongSelf.lastTransactionBuilder = nil;
41
42              // queue another update in case something changed during batch
    updates. this method will bail next runloop if
43              // there are no changes
44              [strongSelf _queueUpdateIfNeeded];
45          }
46      }];
47      [transaction begin];
48  }
```

- transactionBuilder：根据传入的参数，创建transaction
- transaction：builder在这里会根据自己的mode返回对应的transaction，然后transaction根据传入的参数初始化，这里会是batchTransaction，把block添加进队列，然后开始执行更新事务

1.4.3 IGListBatchUpdateTransaction

代码块

```
1   - (void)begin {
2       self.state = IGListBatchUpdateStateQueuedBatchUpdate;
3
4       [self _diff];
5   }
```

- 删除防御式编程的代码，代码只有两行，设置状态，执行diff

代码块

```
1   - (void)_diff {
2       IGListTransitionData *data = self.sectionData;
3       [self.delegate listAdapterUpdater:self.updater
    willDiffFromObjects:data.fromObjects toObjects:data.toObjects];
4
5       __weak __typeof__(self) weakSelf = self;
6       IGListPerformDiffWithData(data,
7                                 self.collectionView,
8                                 self.config.allowsBackgroundDiffing,
```

```
                                self.config.adaptiveDiffingExperimentConfig,
                                ^(IGListIndexSetResult * _Nonnull result, BOOL onBackground) {
        [weakSelf _didDiff:result onBackground:onBackground];
    });
}
```

- 根据sectionDataBlock的数据进行diff算法，然后执行_didDiff
- 如果没有sectionDataBlock就不会去做diff
- listAdapterUpdater:willDiffFromObjects:toObjects:协议方法被调用

代码块

```
- (void)_didDiff:(IGListIndexSetResult *)diffResult onBackground:(BOOL)onBackground {
    if (self.mode == IGListBatchUpdateTransactionModeCancelled) {
        // Cancelling should have already taken care of the completion blocks
        return;
    }

    // After this point, we can assume that the update has began and there's no turning back.
    self.mode = IGListBatchUpdateTransactionModeNotCancellable;

    [self.delegate listAdapterUpdater:self.updater didDiffWithResults:diffResult onBackgroundThread:onBackground];

    @try {
        // Keeping a pointer to self.collectionView.dataSource, because it can get deallocated before the UICollectionView and crash
        id<UICollectionViewDataSource> const collectionViewDataSource = self.collectionView.dataSource;

        if (collectionViewDataSource == nil) {
            // If the data source is nil, we should not call any collection view update.
            [self _bail];
        } else if (diffResult.changeCount > 100 && self.config.allowsReloadingOnTooManyUpdates) {
            [self _reload];
        } else if (self.sectionData && [self.collectionView numberOfSections] != (NSInteger)self.sectionData.fromObjects.count) {
            // If data is nil, there are no section updates.
            IGWarnAssert(@"The UICollectionView's section count (%li) didn't match the IGListAdapter's count (%li), so we can't performBatchUpdates.
```

```
    Falling back to reloadData.",
24                          (long)[self.collectionView numberOfSections],
25                          (long)self.sectionData.fromObjects.count);
26              [self _reload];
27          } else {
28              [self _applyDiff:diffResult];
29          }
30      } @catch (NSException *exception) {
31          [self.delegate listAdapterUpdater:self.updater
32                           collectionView:self.collectionView
33                     willCrashWithException:exception
34                                fromObjects:self.sectionData.fromObjects
35                                  toObjects:self.sectionData.toObjects
36                                 diffResult:diffResult
37                                    updates:(id)_actualCollectionViewUpdates];
38          @throw exception;
39      }
40  }
```

- 如果太多，执行reload，如果前后不相等，执行_reload，不然就执行_applyDiff
  - 这就是为什么涉及大量数据更新的时候不会再更新部分数据，而是直接reloadData
- 这里不同的是，changeCount > 100的时候，section也可能相同，因为有可能这些diffableIdentifier变了
- 而apply和reloadData最大的不同就是，apply是执行部分更新，而reload是会调用collectionView的reloadData

1.4.4 diff算法

上面的执行过程跳过了diff算法部分，入口函数其实很简单，做了简单的判断后就调用diffKit进行diff，唯一不同的就是要不要在backgroundThread执行这个diff算法

diff算法非常长，细致的分析会在IGListKit源码分析（二）中进行

代码块

```
1   static id IGListDiffing(BOOL returnIndexPaths,
2                           NSInteger fromSection,
3                           NSInteger toSection,
4                           NSArray<id<IGListDiffable>> *oldArray,
5                           NSArray<id<IGListDiffable>> *newArray,
6                           IGListDiffOption option)
```

2. 架构

2.1 业务组件化

- 解藕

IGListKit通过SectionController的机制，实现了高度解藕。每个sectionController只关心自己Section的数据和视图渲染逻辑，不关心其它section的实现。这样即使业务变动，各个sectionController直接也互不影响，便于维护和拓展

业务组件化：本质上就是把复杂的业务场景拆分为职责单一、互不依赖的组件（Component），降低了业务耦合度，从架构层面提升了工程的可维护性与灵活性。

- 可插拔

服务端只需要通过AB测试、switch、conf的方式，就可以让一个sectionController展示或者不展示，还可以在服务端配置展示顺序，无需客户端发板即可灵活控制业务功能

业务组件化：可插拔是组件化架构的核心思想之一，只要各业务组件遵循同一协议/接口，暴露标准的处理能力，上层就可以做到组件的动态组合和调度，提升业务响应变化的能力

- 与传统对比

MVC或MVVM体系中，一个ViewController负责整个页面的所有逻辑，导致业务逻辑交错，Controller混乱，后期改动困难。

组件化的思想把业务拼图化，不同业务之间耦合度低，可替换性高，把大的业务进行分治，降低了维护难度，如果一个业务出现问题，可以直接在服务端配置下架，不会影响整体app的使用

IGListKit和业务组件化的思路非常像，无论是在快手直播，还是寻梦记账的看板，都用到了这样的方式对不同的业务进行解藕，让不同业务可以单独控制（可插拔）

2.2 IGListKit架构

从一个performUpdate方法已经大概了解了IGListKit的结构

这里要注意的是，updater和transaction实际上是协议对象，这里只是用最常用类来描述结构

- IGListAdapter：适配器，协调管理SectionController，当数据的变化，UI状态变化时通知SectionController进行更新
- IGListAdapterUpdater：负责管理更新相关的逻辑，协调UpdateTransaction，Coalescer，IGListDiff
  - IGListUpdateCoalescer：负责合并多次update，或者按照业务配置对多次update进行延迟更新
  - UpdateTransactionBuilder：根据上下文生成transaction，是一个工厂对象
  - transaction：事务，负责根据上下文，使用IGListDiff更新并通知adapterDelegate

3. 协议和NSProxy

3.1 setCollectionViewDelegate和setScrollViewDelegate

代码块

```
- (void)setCollectionViewDelegate:(id<UICollectionViewDelegate>)collectionViewDelegate {
    if (_collectionViewDelegate != collectionViewDelegate) {
        _collectionViewDelegate = collectionViewDelegate;
        [self _createProxyAndUpdateCollectionViewDelegate];
    }
}

- (void)setScrollViewDelegate:(id<UIScrollViewDelegate>)scrollViewDelegate {
    if (_scrollViewDelegate != scrollViewDelegate) {
        _scrollViewDelegate = scrollViewDelegate;
        [self _createProxyAndUpdateCollectionViewDelegate];
    }
}

- (void)_createProxyAndUpdateCollectionViewDelegate {
    _collectionView.delegate = nil;

    self.delegateProxy = [[IGListAdapterProxy alloc] initWithCollectionViewTarget:_collectionViewDelegate

scrollViewTarget:_scrollViewDelegate

interceptor:self];
    [self _updateCollectionViewDelegate];
}

- (void)_updateCollectionViewDelegate {
    _collectionView.delegate = (id<UICollectionViewDelegate>)self.delegateProxy ?: self;
}
```

- 无论是设置UICollectionViewDelegate，还是设置UIScrollViewDelegate，本质上都是让adapter持有这个delegate的弱引用，重置collection.delegate，然后把delegate委托给proxy

> 如果直接 `_collectionView.delegate = newDelegate`，有些情况下（比如旧 delegate 被 KVO、或者是 NSProxy，或 Accessibility 相关的引用还未释放），UIKit 内部的引用关系没有完全断开，可能导致「僵尸代理」、「未生效」、「偶现崩溃」等问题，尤其是在有 VoiceOver/Accessibility 机制启用的设备上。

3.2 IGListAdapterProxy

IGListAdapterProxy实际上就是一个代理对象，它会判断这个协议是不是需要被转发的，如果是，就走转发流程走adapter的实现，如果不是就使用_scrollViewTart和_collectionViewTarget

adapter实现的相关delegate方法中，实际上也会带上业务实现，只不过业务实现会在adapter的实现之后被调用

代码块

```
- (BOOL)respondsToSelector:(SEL)aSelector {
    return isInterceptedSelector(aSelector)
    || [_collectionViewTarget respondsToSelector:aSelector]
    || [_scrollViewTarget respondsToSelector:aSelector];
}

- (id)forwardingTargetForSelector:(SEL)aSelector {
    if (isInterceptedSelector(aSelector)) {
        return _interceptor;
    }

    return [_scrollViewTarget respondsToSelector:aSelector] ? _scrollViewTarget : _collectionViewTarget;
}

- (void)forwardInvocation:(NSInvocation *)invocation {
    void *nullPointer = NULL;
    [invocation setReturnValue:&nullPointer];
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)selector {
    return [NSObject instanceMethodSignatureForSelector:@selector(init)];
}
```

- 消息查找到基类后，发现没有消息，会调用NSObject的respondsToSelector:再给这个实例一次转发的机会，如果返回true，就走消息转发
- _interceptor：可以看到消息先是被转发给了这个对象，如果没有实现，就会走业务的实现逻辑，有实现就会重新在这个实例上走消息查询的一套逻辑，而_interceptor就是adapter
- 原因：IGListKit 通过 NSProxy 实现了 delegate 方法的智能分发：只拦截 Adapter 关心的方法并实现逻辑，其他未实现的方法自动透传给外部业务的 delegate。这样可以极大简化 Adapter 代码，无需重复实现所有协议方法，实现了优雅、解耦的代理分发。

代码块

```
- (void)scrollViewDidScroll:(UIScrollView *)scrollView {
    id<IGListAdapterPerformanceDelegate> performanceDelegate = self.performanceDelegate;
    [performanceDelegate listAdapterWillCallScroll:self];

```

```
    // forward this method to the delegate b/c this implementation will steal the message from the proxy
    id<UIScrollViewDelegate> scrollViewDelegate = self.scrollViewDelegate;
    if ([scrollViewDelegate respondsToSelector:@selector(scrollViewDidScroll:)]) {
        [scrollViewDelegate scrollViewDidScroll:scrollView];
    }
    NSArray<IGListSectionController *> *visibleSectionControllers = [self visibleSectionControllers];
    for (IGListSectionController *sectionController in visibleSectionControllers) {
        [[sectionController scrollDelegate] listAdapter:self didScrollSectionController:sectionController];
    }

    [performanceDelegate listAdapter:self didCallScroll:scrollView];
}
```

- IGListAdapterProxy中就有转发这个方法，流程如下：
  - performanceDelegate回调相关方法
  - 尝试调用业务实现的delegate方法
  - 通知sectionController滑动已经开始了
  - performanceDelegate回调相关方法

其它的delegate代理方法基本上都是类似实现

4. 自动化注册ReuseIdentifier

代码块

```
- (__kindof UICollectionViewCell *)dequeueReusableCellOfClass:(Class)cellClass
                                         withReuseIdentifier:(NSString *)reuseIdentifier
                                                forSectionController:(IGListSectionController *)sectionController
                                                             atIndex:(NSInteger)index {
    IGAssertMainThread();
    IGParameterAssert(sectionController != nil);
    IGParameterAssert(cellClass != nil);
    IGParameterAssert(index >= 0);
    UICollectionView *collectionView = self.collectionView;
    IGAssert(collectionView != nil, @"Dequeueing cell of class %@ with reuseIdentifier %@ from section controller %@ without a collection view at
```

```
    index %li", NSStringFromClass(cellClass), reuseIdentifier, sectionController,
    (long)index);
11      NSString *identifier = IGListReusableViewIdentifier(cellClass, nil,
    reuseIdentifier);
12      NSIndexPath *indexPath = [self
    indexPathForSectionController:sectionController index:index
    usePreviousIfInUpdateBlock:NO];
13      if (![self.registeredCellIdentifiers containsObject:identifier]) {
14          [self.registeredCellIdentifiers addObject:identifier];
15          [collectionView registerClass:cellClass
    forCellWithReuseIdentifier:identifier];
16      }
17      return [self _dequeueReusableCellWithReuseIdentifier:identifier
    forIndexPath:indexPath forSectionController:sectionController];
18  }
19
20  - (__kindof UICollectionViewCell *)dequeueReusableCellOfClass:(Class)cellClass
21                                                   forSectionController:
    (IGListSectionController *)sectionController
22                                                                atIndex:(NSInteger)index
    {
23      return [self dequeueReusableCellOfClass:cellClass withReuseIdentifier:nil
    forSectionController:sectionController atIndex:index];
24  }
25
26  NS_INLINE NSString *IGListReusableViewIdentifier(Class viewClass, NSString *
    _Nullable kind, NSString * _Nullable givenReuseIdentifier) {
27      return [NSString stringWithFormat:@"%@%@%@", kind ?: @"",
    givenReuseIdentifier ?: @"", NSStringFromClass(viewClass)];
28  }
```

- 传入reuseIdentifier：注册的id将会是CellClass和reuseIdentifier
- 不传入：将会是CellClass
- 在这里会使用一个NSMubtaleSet存储identifiers，如果有就直接使用，没有会先注册

5. IGListSectionMap

在之前的performUpdate方法中可以看到，SectionMap在SectionController的管理中也有着一定的分量

代码块

```
1   @interface IGListSectionMap ()
2
3   // both of these maps allow fast lookups of objects, list objects, and indexes
```

```
@property (nonatomic, strong, readonly, nonnull) NSMapTable<id, IGListSectionController *> *objectToSectionControllerMap;
@property (nonatomic, strong, readonly, nonnull) NSMapTable<IGListSectionController *, NSNumber *> *sectionControllerToSectionMap;

@property (nonatomic, strong, nonnull) NSMutableArray *mObjects;

@end
```

- 结构：sectionMap使用两个map维护object-sectionController之间的映射和sectionController-section之间的映射，还保留了mObjects这个有序集合

更新和reset是非常重要的，其它方法都是对这三个集合类型对象操作后的返回值

代码块

```
- (void)updateWithObjects:(NSArray *)objects sectionControllers:(NSArray *)sectionControllers {
    IGParameterAssert(objects.count == sectionControllers.count);

    [self reset];

    self.mObjects = [objects mutableCopy];

    id firstObject = objects.firstObject;
    id lastObject = objects.lastObject;

    [objects enumerateObjectsUsingBlock:^(id object, NSUInteger idx, BOOL *stop) {
        IGListSectionController *sectionController = sectionControllers[idx];

        // set the index of the list for easy reverse lookup
        [self.sectionControllerToSectionMap setObject:@(idx) forKey:sectionController];
        [self.objectToSectionControllerMap setObject:sectionController forKey:object];

        sectionController.isFirstSection = (object == firstObject);
        sectionController.isLastSection = (object == lastObject);
        sectionController.section = (NSInteger)idx;
    }];
}

- (void)reset {
    [self enumerateUsingBlock:^(id  _Nonnull object, IGListSectionController * _Nonnull sectionController, NSInteger section, BOOL * _Nonnull stop) {
```

```
        sectionController.section = NSNotFound;
        sectionController.isFirstSection = NO;
        sectionController.isLastSection = NO;
    }];

    [self.sectionControllerToSectionMap removeAllObjects];
    [self.objectToSectionControllerMap removeAllObjects];
}
```

- reset：重置sectionController和section相关的数据
- update：存储并建立sectionController到section相关数据的映射
- 按照单一职责原则（SRP），理论上 SectionMap 不应该关心 sectionController 的属性赋值，理想状态下这部分职责应拆分出去。不过，考虑到 SectionMap 的核心作用几乎都和 sectionController 相关，将赋值逻辑放在这里其实更符合当前工程的实际需求，也是对设计原则的一种灵活应用

代码块

```
- (nullable IGListSectionController *)sectionControllerForSection:(NSInteger)section {
    return [self.sectionMap sectionControllerForSection:section];
}

- (NSInteger)sectionForSectionController:(IGListSectionController *)sectionController {
    return [self.sectionMap sectionForSectionController:sectionController];
}

- (IGListSectionController *)sectionControllerForObject:(id)object {
    return [self.sectionMap sectionControllerForObject:object];
}

- (id)objectForSectionController:(IGListSectionController *)sectionController {
    const NSInteger section = [self.sectionMap sectionForSectionController:sectionController];
    return [self.sectionMap objectForSection:section];
}

- (id)objectAtSection:(NSInteger)section {
    return [self.sectionMap objectForSection:section];
}

- (NSInteger)sectionForObject:(id)item {
    return [self.sectionMap sectionForObject:item];
}
```

```
25
26  - (NSArray *)objects {
27      return self.sectionMap.objects;
28  }
```

大量方法使用到了sectionMap，可见其重要性，虽然内部很简单😄，但是map、map、array的结构还是值得学习的

6. 防御式编程

虽然本文删除了大量防御式编程的代码，但是防御式编程的思想还是值得学习和在工程中使用

- 不相信任何外部输入和上下文，始终做好最坏打算，这是一种好的编程习惯，也是代码安全，健壮，易维护的保障手段

但是防御式编程也需要区分场景，OSTEP中有一句话很有意思，重要的是做对事。所以不能把一种方式完全带入所有场景，考虑下面这样的情况，这是yymodel的一段代码，而作者并没有为这段代码设置断言

代码块

```
1  - (BOOL)modelSetWithJSON:(id)json {
2      NSDictionary *dic = [NSObject _yy_dictionaryWithJSON:json];
3      return [self modelSetWithDictionary:dic];
4  }
```

- 如果设置断言会怎么样：所有碰到json为nil的地方都会crash，而有的时候数据可能就是空的，这也是程序的正常行为，毕竟nil的存在就是为了表示没有数据

而IGListKit使用防御式编程的地方

代码块

```
1  - (void)setCollectionView:(UICollectionView *)collectionView {
2      IGAssertMainThread();
3  }
```

- 在这里如果collectionView为空，整个collectionView会消失，这一定是懒加载写了if (_collectionView)之类导致没有正确初始化的情况
- 如果业务不需要collectionView出现，就应该不要同时不要初始化adapter以防止错误调用带来的性能损失
- 所以在这里crash app是正确的选择

如果不能特别确定怎么办？或者我们就需要debug模式下crash发现问题，但是在线上又有一定的容忍度，毕竟线上最好不要影响整个app的运行

代码块

```
#if DEBUG
#define XMAssert(condition, fmt, ...) NSAssert((condition), (fmt), ##__VA_ARGS__)
#else
#define XMAssert(condition, fmt, ...) \
    do { \
        if (!(condition)) { \
            NSLog(@"[XMAssert][%s:%d] " fmt, __FILE__, __LINE__, ##__VA_ARGS__); \
            /* 也可以加上自动埋点上报到自己的后台，比如: XMAssertReporterUpload(...) */ \
        } \
    } while (0)
#endif
```

- 使用这样的方式，简单区分debug和release的行为也许是一种解决方案，让debug模式下crash，让release情况下上报问题
- 最好不要配置xcode来定义NSAssert的行为，这样会修改所有业务的配置
- 实际上这里的命名有点问题，没有突出这是一个区分环境的断言，真实开发的时候可以更加规范

7. 局限性

1. IGListKit不支持不同section之间cell的移动，但如果整个section只有一个cell是支持移动的

代码块

```
- (void)moveInSectionController:(IGListSectionController *)sectionController fromIndex:(NSInteger)fromIndex toIndex:(NSInteger)toIndex {
    UICollectionView *collectionView = self.collectionView;

    NSIndexPath *fromIndexPath = [self indexPathForSectionController:sectionController index:fromIndex usePreviousIfInUpdateBlock:YES];
    NSIndexPath *toIndexPath = [self indexPathForSectionController:sectionController index:toIndex usePreviousIfInUpdateBlock:NO];

    if (fromIndexPath == nil || toIndexPath == nil) {
        return;
    }
```

```
10
11      [self.updater moveItemInCollectionView:collectionView
    fromIndexPath:fromIndexPath toIndexPath:toIndexPath];
12  }
```

- 移动发生在同一个sectionController之内

代码块

```
1   - (void)moveSectionControllerInteractive:(IGListSectionController
    *)sectionController
2                                   fromIndex:(NSInteger)fromIndex
3                                     toIndex:(NSInteger)toIndex
    NS_AVAILABLE_IOS(9_0) {
4       UICollectionView *collectionView = self.collectionView;
5
6       if (fromIndex != toIndex) {
7           id<IGListAdapterDataSource> dataSource = self.dataSource;
8
9           NSArray *previousObjects = [self.sectionMap objects];
10
11          if (self.isLastInteractiveMoveToLastSectionIndex) {
12              self.isLastInteractiveMoveToLastSectionIndex = NO;
13          }
14          else if (fromIndex < toIndex) {
15              toIndex -= 1;
16          }
17
18          NSMutableArray *mutObjects = [previousObjects mutableCopy];
19          id object = [previousObjects objectAtIndex:fromIndex];
20          [mutObjects removeObjectAtIndex:fromIndex];
21          [mutObjects insertObject:object atIndex:toIndex];
22
23          NSArray *objects = [mutObjects copy];
24
25          [self.moveDelegate listAdapter:self moveObject:object
    from:previousObjects to:objects];
26
27          // update our model based on that provided by the data source
28          NSArray<id<IGListDiffable>> *updatedObjects = [dataSource
    objectsForListAdapter:self];
29          [self _updateObjects:updatedObjects dataSource:dataSource];
30      }
31
32      [self.updater moveSectionInCollectionView:collectionView
    fromIndex:fromIndex toIndex:toIndex];
```

```
33    }
```

- 之前讲过，sectionMap就是两个map + 一个array，里面存储着所有的objects对象，每个object对应一个sectionController，而不是cell，如果业务有cell，这个cell对应的cellObject应该存储在这个object里，而不是object本身，除非这个sectionController只有一个cell
- 在这里IGListKit只是对objects数据进行更新，并不涉及cellObject的变化，而这里也是唯一调用moveDelegate方法的地方