IGListKit源码分析（二）

前言

我个人对IGListKit的实战并不多，碰到的业务场景较少，学习IGListKit也是出于个人兴趣。所以对于业务场景覆盖的可能不够到位，这两篇文章主要还是分析源码，以此展现对于源码阅读、问题定位、工程思考和学习的能力

这篇文章，我们从sectionController的生命周期出发，来聊聊adpater、updater、transaction、transactionBuilder、coalescer之间的关系，它们是如何联动的，细节是什么。下面的文章，我会尽量体现个人的思考过程，和阅读源码的思路

最后我们会分析一下IGListDiff具体做了什么

1. SectionController

1.1 生命周期

生命周期总结一下就是：当数据源发生变更的时候大致有三种不同的更新场景，下面会介绍，下面统一的流程：

1. 会先过滤objects中的重复项（setDataSource/setCollectionView不过滤）
2. 然后「创建」或者「复用」之前的sectionController，根据objects的顺序调整sectionController的位置
3. 更新sectionMap的映射，同时给sectionController的变量赋值（isLastSection、isFirstSection、section等）

具体代码如下

代码块

```
1  - (void)_updateObjects:(NSArray *)objects dataSource:
   (id<IGListAdapterDataSource>)dataSource {
2      [self _updateWithData:[self _generateTransitionDataWithObjects:objects
   dataSource:dataSource]];
3  }
```

- 这里体现了顺序，先更新sectionController，再更新sectionMap
- 因为sectionMap只是映射，所以要先根据新的数据源创建/调整sectionController，这个时候数据源才算完全准备好了，然后再更新sectionMap

```objc
- (IGListTransitionData *)_generateTransitionDataWithObjects:(NSArray *)objects dataSource:(id<IGListAdapterDataSource>)dataSource {
    IGListSectionMap *map = self.sectionMap;

    NSMutableArray<IGListSectionController *> *sectionControllers = [[NSMutableArray alloc] initWithCapacity:objects.count];
    NSMutableArray *validObjects = [[NSMutableArray alloc] initWithCapacity:objects.count];

    IGListSectionControllerPushThread(self.viewController, self);

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

        if (sectionController == nil) {
            sectionController = [dataSource listAdapter:self sectionControllerForObject:object];
        }

        // 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];
}
```

- 这个方法IGListKit源码分析（一）已经讲过作用了，这里主要分析生命周期相关的事
- 根据旧的sectionMap判断sectionController有没有被创建过，如果没有就创建，有就复用，更新sectionController的context和controller
- addObject就是调整位置，保证和object to section顺序一致

代码块

```objc
- (void)_updateWithData:(IGListTransitionData *)data {
    _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];

    for (id object in updatedObjects) {
        [[map sectionControllerForObject:object] didUpdateToObject:object];
    }

    [self _updateBackgroundView];

    _isInObjectUpdateTransaction = NO;
}
```

- 先检查object有没有改变实例，IGListKit在这里发生了不一致的行为，用toObjects更新map，但是用updatedObjects更新sectionController，这是为什么?
- 按照正常理解，只有当[object isEqualToDiffableObject:newObject] == NO的时候，才会认为这个object变化了
- 这里用了地址做比较就会出现这样的情况
  - diffIdentifier没变，地址变了，会调用didUpdateToObject
  - diffIdentifier变了，地址没变，不会调用didUpdateToObject

1.1.1 怪异的行为?

因为这里比较有意思，也有助于理解IGListKit和sectionController的生命周期，所以我们先来看看，上面这个更新行为的原因是什么?

_updateWithData是一个非常常见的方法，每一次更新数据都会调用这个，IGListKit更新数据只有三个方法（其实有四个，在下面会讲解，在这里先把performUpdate当作一个方法）

在这里我们用了序号代表它们的优先级

1. setDataSource/setCollectionView
2. reloadDataWithCompletion
3. performUpdate

源码如下

代码块

```
typedef NS_ENUM (NSInteger, IGListUpdateTransactionBuilderMode) {
    /// The lowest priority is a batch-update, because a reload or dataSource take care of any changes.
    IGListUpdateTransactionBuilderModeBatchUpdate,
    /// The second priority is reloading all data.
    IGListUpdateTransactionBuilderModeReload,
    /// The highest priority is changing the `UICollectionView` dataSource.
    IGListUpdateTransactionBuilderModeDataSourceChange,
};

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

而它们对应的是三种不同的更新，也对应三种不同的transaction，这三个不同的transaction也有不同的行为（build add之后会给mode赋值，所以这实际上会产生三种不同的transaction）

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];
}

- (void)reloadDataWithCollectionViewBlock:
(IGListCollectionViewBlock)collectionViewBlock
                          reloadUpdateBlock:
(IGListReloadUpdateBlock)reloadUpdateBlock
                                 completion:(nullable
IGListUpdatingCompletion)completion {
    [self.transactionBuilder
addReloadDataWithCollectionViewBlock:collectionViewBlock

reloadBlock:reloadUpdateBlock
                                                       completion:completion];

    [self _queueUpdateIfNeeded];
}

- (void)performDataSourceChange:(IGListDataSourceChangeBlock)block {
    if (!self.transaction
        && ![self.transactionBuilder hasChanges]
        && !IGListExperimentEnabled(self.experiments,
IGListExperimentRemoveDataSourceChangeEarlyExit)) {
        // If nothing is going on, lets take a shortcut.
        block();
        return;
    }

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

    // Lets try to cancel any current transactions.
    if ([self.transaction cancel] && self.lastTransactionBuilder) {
        // We still need to apply the item-updates and completion-blocks, so
lets merge the builders.
        [builder addChangesFromBuilder:(IGListUpdateTransactionBuilder
*)self.lastTransactionBuilder];
```

```
41      }
42
43      // Lets merge pending changes
44      [builder addChangesFromBuilder:self.transactionBuilder];
45
46      // Clear the current state
47      self.transaction = nil;
48      self.lastTransactionBuilder = nil;
49      self.transactionBuilder = builder;
50
51      // Update synchronously
52      [self update];
53  }
```

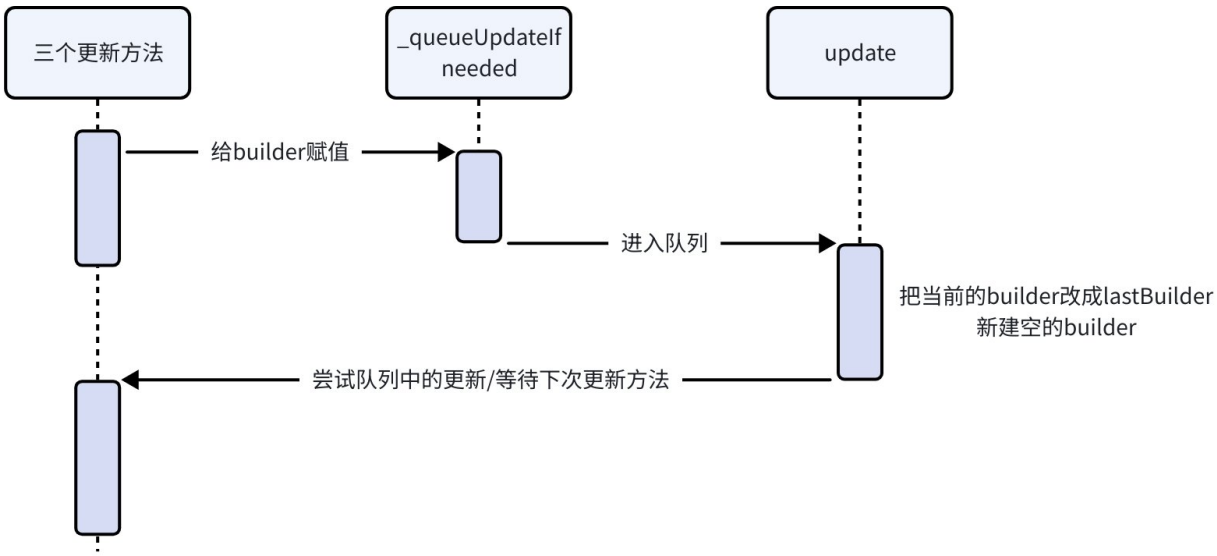
- 这三个方法，当不定义coalescer的行为的时候它们最终调用的更新方法是一致的，就是update
- _queueUpdateIfNeeded是在需要的时候把transaction按照业务定义的方式进行延迟更新，最后也会调用update

别的都是根据不同的方法更新builder，过一会让它返回不同类型的transaction，而DataSourceChange最特别

- 如果没有在进行的transaction更新，直接调用block更新变化，也就是调用之前的_updateWithData方法，set一些状态的block
- 如果有，就取消，而这个取消也是有条件的，在_didDiff方法前，也就是diff算法执行完成后，将要执行reload的时候会判断要不要取消，如果已经reload，就无法取消了
- 如果可以取消，合并之前的builder；接着合并当前的builder，重置所有状态，无视coalescer的配置，直接update

什么是lastTransactionBuilder?

- 在reloadData/reloadDataSource/performUpdate的时候会给这个builder赋值，在真正update的时候会把当前的builder改成lastBuilder，然后新建一个builder
- 时序图如下



代码块

```
        [self.transactionBuilder addItemBatchUpdateAnimated:animated
                                         collectionViewBlock:collectionViewBlock
                                                 itemUpdates:itemUpdates
                                                  completion:completion];
```

- batchUpdate最终会调用[collectionView performUpdates:completion:]，这里的completion是一个异步调用，它的“行为”有点像dispatch_after，当collectionView执行完更新，真正的动画会在之后的某一帧中被异步调用。所以每次都会给当前的builder里，添加新的上下文
- Mode：多次添加，优先级最高的是最终的值
- animated：多次添加，必须每次都是YES才展示动画
- block需要更新的更新，需要进队列的进队列

在上面的时序图中，我们省略了coalescer，但是可以把这个进入队列看作是coalescer，所以我们现在大致明白了updater、builder、transaction之间的关系，知道了更新方法和_queueUpdateIfNeeded、update之间的关系

不同的transaction

代码块

```
- (void)update {
    id<IGListUpdateTransactable> transaction = [self.transactionBuilder buildWithConfig:config delegate:_delegate updater:self];
    self.transaction = transaction;
    self.lastTransactionBuilder = self.transactionBuilder;
    self.transactionBuilder = [IGListUpdateTransactionBuilder new];
}

```

```
- (nullable id<IGListUpdateTransactable>)buildWithConfig:(IGListUpdateTransactationConfig)config
                                                 delegate:(nullable id<IGListAdapterUpdaterDelegate>)delegate
                                                  updater:(IGListAdapterUpdater *)updater {
    switch (self.mode) {
        case IGListUpdateTransactionBuilderModeBatchUpdate: {
            IGListCollectionViewBlock collectionViewBlock = self.collectionViewBlock;
            return [[IGListBatchUpdateTransaction alloc] initWithCollectionViewBlock:collectionViewBlock

updater:updater

delegate:delegate

config:config

animated:self.animated

sectionDataBlock:self.sectionDataBlock

applySectionDataBlock:self.applySectionDataBlock

itemUpdateBlocks:self.itemUpdateBlocks

completionBlocks:self.completionBlocks];
        }
        case IGListUpdateTransactionBuilderModeReload: {
            IGListReloadUpdateBlock reloadBlock = self.reloadBlock;
            IGListCollectionViewBlock collectionViewBlock = self.collectionViewBlock;
            return [[IGListReloadTransaction alloc] initWithCollectionViewBlock:collectionViewBlock

updater:updater

delegate:delegate

reloadBlock:reloadBlock

itemUpdateBlocks:self.itemUpdateBlocks

completionBlocks:self.completionBlocks];
        }
        case IGListUpdateTransactionBuilderModeDataSourceChange: {
```

```
35              IGListDataSourceChangeBlock dataSourceChangeBlock =
    self.dataSourceChangeBlock;
36              return [[IGListDataSourceChangeTransaction alloc]
    initWithChangeBlock:dataSourceChangeBlock
37
    itemUpdateBlocks:self.itemUpdateBlocks
38
    completionBlocks:self.completionBlocks];
39          }
40      }
41  }
```

- update省略了部分代码
- 可以看到transaction实际上会根据不同的更新行为，创建不同的对象，这几个不同的对象

这几个不同对象可以见名知意的，下面的reloadData都相当于[collectionView reloadData]

- IGListBatchUpdateTransaction：部分更新，会用到diffKit，大致如下
  - 数量 > 100，直接reloadData
  - 新的数据源数量和之前的不一样，直接reloadData
  - 执行diff更新
- IGListReloadTransaction：全量更新，直接reloadData
- IGListDataSourceChangeTransaction：还是它最特别，它只执行对应的block，不会操作collectionView

所以我们知道：

在setDataSource/setCollectionView的时候实际上不会调用reloadData方法，只会调用[sectionController didUpdateToObject]

> 😅 所以到这里还是没有解决，到底为什么要比较指针更新，而不是diffable，而且要区别map映射和didUpdateToObject

1.1.2 一个乌龙

代码块

```
1   IGListSectionMap *map = self.sectionMap;
2
3   // Note: We use an array, instead of a set, because the updater should have
    dealt with duplicates already.
4   NSMutableArray *updatedObjects = [NSMutableArray new];
5
```

```
6   for (id object in data.toObjects) {
7       // check if the item has changed instances or is new
8       const NSInteger oldSection = [map sectionForObject:object];
9       if (oldSection == NSNotFound || [map objectForSection:oldSection] !=
    object) {
10          [updatedObjects addObject:object];
11      }
12  }
13
14  [map updateWithObjects:data.toObjects
    sectionControllers:data.toSectionControllers];
15
16  // now that the maps have been created and contexts are assigned, we consider
    the section controller "fully loaded"
17  for (id object in updatedObjects) {
18      [[map sectionControllerForObject:object] didUpdateToObject:object];
19  }
```

我们再仔细看看这个方法，就会发现这个方法只是挑出之前没有的section，地址变化的object，更新sectionController

1.1.3 为什么乌龙?

为什么IGListKit源码（一）中在这里没有发现这个问题，看的多了反而出错了?

出错原因是没搞明白细节，复制之前的疑问如下：

- 这里用了地址做比较就会出现这样的情况
  - diffIdentifier没变，地址变了，会调用didUpdateToObject
  - diffIdentifier变了，地址没变，不会调用didUpdateToObject

第一种情况：地址变了调用didUpdateToObject是正常的，diffIdentifier没变，就相当于数据源没发生变化，既然没有发生变化，调用didUpdateToObject也不会出问题

第二种情况：diffIdentifier变了相当于数据源变了，但是地址没变，所以不用调用didUpdateToObject，因为sectionController已经持有它的引用了

所以弄来弄去，居然是个内存问题🙃

1.1.4 值得的错误

虽然弄错了，但是有的时候并不重要，通过这个错误，我们更加完整的了解了IGListKit：

1. 不同的更新方法具体做了什么？谁没有[collectionView reloadData]
2. 不同的transaction，优先级如何
3. coalescer的作用

4. coalescer、transaction、transactionBuilder之间的协作
5. 更新数据源的时序问题
6. 谁会diff
7. 也给分析生命周期引出了一个思路

1.1.5 sectionController的生命周期

_generateTransitionDataWithObjects创建/更新了sectionController，_updateWithData更新了sectionMap，同时更新了sectionController关于section的属性，这两个方法总是成对出现（虽然不同方法调用它们的形式不同）

所以我们知道下面这些方法会创建/更新/赋值sectionController，并且会更新sectionMap

- reloadData
- setDataSource/setCollectionView
- performUpdate系列方法

2. 合并更新

不是所有情况都会合并更新

2.1 不同的performUpdates

代码块

```
- (void)performUpdateWithCollectionViewBlock:(IGListCollectionViewBlock)collectionViewBlock
                                    animated:(BOOL)animated
                                  itemUpdates:(void (^)(void))itemUpdates
                                   completion:(void (^)(BOOL))completion {
    // if already inside the execution of the update block, immediately unload the itemUpdates block.
    // the completion blocks are executed later in the lifecycle, so that still needs to be added to the batch
    if ([self isInDataUpdateBlock]) {
        if (completion != nil) {
            [self.transaction addCompletionBlock:completion];
        }
        itemUpdates();
    } else {
        [self.transactionBuilder addItemBatchUpdateAnimated:animated
                                    collectionViewBlock:collectionViewBlock
                                            itemUpdates:itemUpdates
                                             completion:completion];

        [self _queueUpdateIfNeeded];
```

```
19      }
20  }
```

这个方法用于sectionController内部的变化，相当于局部更新；下面的内容我们称sectionController内部变化的方法performUpdate为局部更新；而之前分析过的用于更新所有section的方法performUpdate为全量更新

这里非常容易搞混，因为它们的方法名很像🤣

在局部更新中（isInDataUpdateBlock）为什么直接执行了itemUpdates？不会有什么问题吗？

保证原子性和一致性

- 在批量更新执行中时（_applyDiff和reload执行中），如果业务或组件代码又想做一些item级别变更，比如：

代码块

```
[adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx){
    // 在block里又触发了item的增删改
    [adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx2){
        // 递归/嵌套
    } completion:nil];
} completion:nil];

```

- updates方法的本质是_applyDiff/reload，updates最后会在这两个方法的其中之一被调用
- 在这里，多次updates会被合并，因为嵌套的updates被直接当作block调用了
- 把completion添加到队列里，在transaction执行完之后再执行
- 具体原因在下面的内容中

2.2 coalescer与合并更新

实际上，coalescer和合并更新并没有任何关系，它只是决定了什么时候更新，前提是业务进行了配置

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) {
            // It's been long enough, so lets reset interval and perform update right away
            _coalescenceInterval = config.minInterval;
            [self _performUpdate];
            return;
        } else {
            // If we keep hitting the delay, lets increase it.
            _coalescenceInterval = MIN(currentCoalescenceInterval + config.intervalIncrement, config.maxInterval);
        }
    }

    // Delay by the time remaining in the interval
    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];
    });
}
```

- 从这个方法可以看到，在处理了一系列配置逻辑之后，coalescer还是会调用_performUpdate，这里影响的只是调用时间而已

所以什么是合并更新呢?

在执行「全量更新」之后进来的transaction有可能会被合并，因为[collectionView performUpdates:completion:]它的completion会是一次主线程的异步操作，completion会在之后的某一帧被主线程异步调用，所以这期间有可能会有transaction被添加进builder

代码块

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

在这之后进来的transaction，包括reloadData、dataSourceChange都有可能被合并，因为[transaction cancel]方法会在_didDiff后返回NO

代码块

```
- (void)addItemBatchUpdateAnimated:(BOOL)animated
                collectionViewBlock:
(IGListCollectionViewBlock)collectionViewBlock
                      itemUpdates:(IGListItemUpdateBlock)itemUpdates
                       completion:(nullable
IGListUpdatingCompletion)completion {
    self.mode = MAX(self.mode, IGListUpdateTransactionBuilderModeBatchUpdate);

    // disabled animations will always take priority
    // reset to YES in -cleanupState
    self.animated = self.animated && animated;
    self.collectionViewBlock = collectionViewBlock;

    [self.itemUpdateBlocks addObject:itemUpdates];

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

- 这里收集的是itemUpdatesBlocks
- 而全量调用的performUpdates方法只有一个applySectionDataBlock

代码块

```
- (void)_diff {
    IGListTransitionData *data = self.sectionData;
    [self.delegate listAdapterUpdater:self.updater
willDiffFromObjects:data.fromObjects toObjects:data.toObjects];

    __weak __typeof__(self) weakSelf = self;
    IGListPerformDiffWithData(data,
                              self.collectionView,
                              self.config.allowsBackgroundDiffing,
                              self.config.adaptiveDiffingExperimentConfig,
                              ^(IGListIndexSetResult * _Nonnull result, BOOL
onBackground) {
        [weakSelf _didDiff:result onBackground:onBackground];
    });
}
```

- 这里可以看到sectionController局部更新数据的perfromUpdate方法是不会进行diff的，因为它的sectionData是空的，没有传入这个数据

- 只有全量更新会进行diff

代码块

```
- (void)_applyDiff:(IGListIndexSetResult *)diffResult {
    [self.delegate listAdapterUpdater:self.updater
willPerformBatchUpdatesWithCollectionView:self.collectionView
                            fromObjects:self.sectionData.fromObjects
                              toObjects:self.sectionData.toObjects
                    listIndexSetResult:diffResult
                               animated:self.animated];
    void (^updates)(void) = ^ {
        [self _applyDataUpdates];
        [self _applyCollectioViewUpdates:diffResult];
    };
    @try {
        if (self.animated) {
            [self.collectionView performBatchUpdates:updates
completion:completion];
        } else {
            [UIView performWithoutAnimation:^{
                [self.collectionView performBatchUpdates:updates
completion:completion];
            }];
        }
    }
}
```

- _applyDiff/reload：都会在调用_applyDataUpdates、itemUpdates更新数据

代码块

```
- (void)_applyDataUpdates {
    self.state = IGListBatchUpdateStateExecutingBatchUpdateBlock;

    if (self.applySectionDataBlock != nil && self.sectionData != nil) {
        self.applySectionDataBlock((IGListTransitionData *)self.sectionData);
    }

    for (IGListItemUpdateBlock block in self.itemUpdateBlocks) {
        block();
    }

    self.state = IGListBatchUpdateStateExecutedBatchUpdateBlock;
}
```

- 这里可见，多个全量更新只取最后一个；多个局部更新全部都执行
- 这样也能理解，最后执行的全量更新就是最新的值；而局部更新才需要合并
- 在这里可以看到，在这个方法里更新的state变化了，也就是局部更新方法入口正在更新中的判断（isInDataUpdateBlock）就是这里

2.3 不可打断的主线程和RunLoop

主线程不可打断，RunLoop事件也不会打断当前正在执行的事件，所以UI操作不会产生竞态，为什么这里会出现_applyDataUpdates调用了之后，又进入了_applyDataUpdates?

有一种可能的情况，业务方法进行了嵌套，所以performBatchAnimated这个方法中的判断，应该就是防止这样的情况，直接调用itemUpdates()就能理解了

代码块

```
[adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx){
    // 在block里又触发了item的增删改
    [adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx2){
        // 递归/嵌套
    } completion:nil];
} completion:nil];

```

这样的情况下，updates会在这个block里，也就是上层的itemUpdates里被直接调用，相当于被合并，而completion如果有值，会在update执行完之后被统一调用

3. 生命周期总结

- sectionController在_generateTransitionDataWithObjects中被创建或者复用
- 在_updateWithData中和sectionMap关联，并且获得了section相关属性的赋值
- 相关dataSource方法在adapter+UICollectionView中和dataSource相关的方法被调用，因为adapter代理了相关的方法
- delegate方法同理，因为adapter也代理了部分delegate方法

4. 对于diff的性能优化

- 只有调用performUpdate全量刷新会支持diff
- 其它的方法最多只是会根据diffable协议进行去重
- perfromUpdate也是需要满足条件才部分刷新
  - 变化总量小于等于100
  - section总数没变化

变化总量的定义如下

代码块

```
- (NSInteger)changeCount {
    return self.inserts.count + self.deletes.count + self.updates.count + self.moves.count;
}
```

5. diffKit做了什么

IGListBatchUpdateTransaction，可以看到transaction会根据sectionData进行diff

代码块

```
- (void)_diff {
    IGListTransitionData *data = self.sectionData;
    [self.delegate listAdapterUpdater:self.updater willDiffFromObjects:data.fromObjects toObjects:data.toObjects];

    __weak __typeof__(self) weakSelf = self;
    IGListPerformDiffWithData(data,
                              self.collectionView,
                              self.config.allowsBackgroundDiffing,
                              self.config.adaptiveDiffingExperimentConfig,
                              ^(IGListIndexSetResult * _Nonnull result, BOOL onBackground) {
        [weakSelf _didDiff:result onBackground:onBackground];
    });
}
```

全局搜索，sectionData的赋值只有一处

代码块

```
- (instancetype)initWithCollectionViewBlock:(IGListCollectionViewBlock)collectionViewBlock
                                    updater:(IGListAdapterUpdater *)updater
                                   delegate:(id<IGListAdapterUpdaterDelegate>)delegate
                                     config:(IGListUpdateTransactationConfig)config
                                   animated:(BOOL)animated
                             sectionDataBlock:(IGListTransitionDataBlock)sectionDataBlock
```

```
                        applySectionDataBlock:(IGListTransitionDataApplyBlock)applySectionDataBlock
                            itemUpdateBlocks:(NSArray<IGListItemUpdateBlock> *)itemUpdateBlocks
                            completionBlocks:(NSArray<IGListUpdatingCompletion> *)completionBlocks {
    if (self = [super init]) {
        _sectionData = sectionDataBlock ? sectionDataBlock() : nil;
    }
    return self;
}
```

- 这个初始化方法的调用同样只有一处，就是「全局更新」的方法performUpdates
- 所以可以确定，只有它可以用到diffKit

5.1 IGListPerformDiff

代码块

```
void IGListPerformDiffWithData(IGListTransitionData *_Nullable data,
                               UIView *view,
                               BOOL allowsBackground,
                               IGListAdaptiveDiffingExperimentConfig adaptiveConfig,
                               IGListDiffExecutorCompletion completion) {
    if (!completion) {
        return;
    }

    if (adaptiveConfig.enabled) {
        _adaptivePerformDiffWithData(data, view, allowsBackground, adaptiveConfig, completion);
    } else {
        // Just to be safe, lets keep the original code path intact while adaptive diffing is still an experiment.
        _regularPerformDiffWithData(data, allowsBackground, completion);
    }
}
```

- 这里的adaptiveConfig是updater的一个属性控制的，它可以定制化在后台线程执行diff的行为，可以根据view如果不可见就以较低的优先级执行，也可以指定在后台线程以高优先级执行等等

代码块

```
@property (nonatomic, assign) IGListAdaptiveDiffingExperimentConfig
```

```
    adaptiveDiffingExperimentConfig;
```

- 如果打开它，IGListKit会根据参数allowsBackgroundDiffing，在后台线程执行diff，如果切换了线程最后会在主线程执行completion，在这里就不具体展开了

代码块

```
@property (nonatomic, assign) BOOL allowsBackgroundDiffing;
```

IGListPerformDiff是一个中间层，并不是一个类，它是一些对perform这个行为不同定义的方法的集合，体现了单一性原则，值得学习

5.2 IGListDiff

然后才正式进入IGListDiff

一开始会检查是不是空，空就空返回，newArray空就返回空，oldArray空就返回一个全insert的结果

接下来是一些准备工作，我们也通过准备工作先了解一下这些C++的变量

代码块

```
unordered_map<id<NSObject>, IGListEntry, IGListHashID, IGListEqualID> table;
```

- unordered_map：是C++的哈希表容器，比NSDictionary更高效
- id<NSObject>：是key
- IGListEntry：是value，是 IGListKit 为每个对象在 diff 过程中存储的中间状态/信息
- IGListHashID：自定义的哈希函数，其实就是返回[object hash]
- IGListEqualID：自定义的等价函数，就是先比较指针，再比较isEqual

代码块

```
vector<IGListRecord> newResultsArray(newCount);
```

- C++动态数组，高性能，分配地址，大小newCount，值全空

代码块

```
    vector<IGListRecord> newResultsArray(newCount);
    for (NSInteger i = 0; i < newCount; i++) {
        id<NSObject> key = IGListTableKey(newArray[i]);
        IGListEntry &entry = table[key];
```

```
        entry.newCounter++;

        // add NSNotFound for each occurence of the item in the new array
        entry.oldIndexes.push(NSNotFound);

        // note: the entry is just a pointer to the entry which is stack-allocated in the table
        newResultsArray[i].entry = &entry;
    }
```

这里主要分析C++特性，下一个方法对做了什么做总结

- 主要是table的一个特性，如果table[key]没有值，会自动插入一个空的entry然后返回
- IGListTableKey就是i位置上的对象它的diffIdentifier

代码块

```
    vector<IGListRecord> oldResultsArray(oldCount);
    for (NSInteger i = oldCount - 1; i >= 0; i--) {
        id<NSObject> key = IGListTableKey(oldArray[i]);
        IGListEntry &entry = table[key];
        entry.oldCounter++;

        // push the original indices where the item occurred onto the index stack
        entry.oldIndexes.push(i);

        // note: the entry is just a pointer to the entry which is stack-allocated in the table
        oldResultsArray[i].entry = &entry;
    }
```

这个方法和上面的方法类似，所以不解释这个方法的细节了，主要分析一下这两个方法的作用

- 这两个方法分别遍历新/旧数组
- 都使用对象的diffIdentifier生成key，并且如果没有entry就创建新的
- 方法1：
  - 记录key在新元素数组中出现的次数
  - 每次出现都给oldIndexes压栈一个NSNotFound，表示暂时没有找到旧的数组中匹配的下标
  - 按照顺序，记录该entry，方便后续按顺序进行访问和对比
- 方法2：
  - 标记该key在旧元素数组中出现的次数

    - 把当前旧数组下标i push进entry的oldIndexes栈
    - 按照oldArray记录entry的顺序，方便后续访问和操作
- 试着理解一下oldIndexes的逻辑，后面肯定用得到
    - old存在，new不存在：此时栈里会push一个oldIndex
    - old不存在，new存在：此时栈里会push一个NSNotFound
    - old存在，new存在：此时会有一个NSNotFound，一个oldIndex
- 暂时理解不了oldIndexes的逻辑，先往后看

代码块

```
    for (NSInteger i = 0; i < newCount; i++) {
        IGListEntry *entry = newResultsArray[i].entry;

        // grab and pop the top original index. if the item was inserted this will be NSNotFound
        NSCAssert(!entry->oldIndexes.empty(), @"Old indexes is empty while iterating new item %li. Should have NSNotFound", (long)i);
        const NSInteger originalIndex = entry->oldIndexes.top();
        entry->oldIndexes.pop();

        if (originalIndex < oldCount) {
            const id<IGListDiffable> n = newArray[i];
            const id<IGListDiffable> o = oldArray[originalIndex];
            switch (option) {
                case IGListDiffPointerPersonality:
                    // flag the entry as updated if the pointers are not the same
                    if (n != o) {
                        entry->updated = YES;
                    }
                    break;
                case IGListDiffEquality:
                    // use -[IGListDiffable isEqualToDiffableObject:] between both version of data to see if anything has changed
                    // skip the equality check if both indexes point to the same object
                    if (n != o && ![n isEqualToDiffableObject:o]) {
                        entry->updated = YES;
                    }
                    break;
            }
        }
        if (originalIndex != NSNotFound
            && entry->newCounter > 0
```

```
            && entry->oldCounter > 0) {
            // if an item occurs in the new and old array, it is unique
            // assign the index of new and old records to the opposite index (reverse lookup)
            newResultsArray[i].index = originalIndex;
            oldResultsArray[originalIndex].index = i;
        }
    }
```

- 按照新数组的顺序遍历每个entry，出栈，根据上面的分析，此时出栈top的结果只有两种
  a. top是oldIndex，且栈中只有这一个元素
  b. 栈底还有一个NSNotFound
- 这对应着两种情况：1. 旧元素被删除了 2. 旧元素还在，可能位移，可能没有
- option在这里是IGListDiffEquality：判断指针不相等，并且diff协议判断不相等，同时满足就是更新了
- 如果一个元素在新旧数组中同时出现，就在新数组的record中记录旧数组中这个元素的下标，在旧数组的record中记录新数组中这个元素的下标
- 这样一一建立好“配对”后，后面就能非常高效地判断某个元素是insert、delete、move还是update：
  - 如果新旧index互指，就是move/update；
  - 如果只有新有，旧index是NSNotFound，就是insert；
  - 如果只有旧有，新index是NSNotFound，就是delete

代码块

```
    id mInserts, mMoves, mUpdates, mDeletes;
    if (returnIndexPaths) {
        mInserts = [NSMutableArray<NSIndexPath *> new];
        mMoves = [NSMutableArray<IGListMoveIndexPath *> new];
        mUpdates = [NSMutableArray<NSIndexPath *> new];
        mDeletes = [NSMutableArray<NSIndexPath *> new];
    } else {
        mInserts = [NSMutableIndexSet new];
        mMoves = [NSMutableArray<IGListMoveIndex *> new];
        mUpdates = [NSMutableIndexSet new];
        mDeletes = [NSMutableIndexSet new];
    }
```

- 根据传入参数，区分构建IndexPath数组还是IndexSet，这里是IndexSet，因为transaction进入的方法调用的时候传入了固定的参数

代码块

```
    vector<NSInteger> deleteOffsets(oldCount), insertOffsets(newCount);
    NSInteger runningOffset = 0;
    // iterate old array records checking for deletes
    // incremement offset for each delete
    for (NSInteger i = 0; i < oldCount; i++) {
        deleteOffsets[i] = runningOffset;
        const IGListRecord record = oldResultsArray[i];
        // if the record index in the new array doesn't exist, its a delete
        if (record.index == NSNotFound) {
            addIndexToCollection(returnIndexPaths, mDeletes, fromSection, i);
            runningOffset++;
        }

        addIndexToMap(returnIndexPaths, fromSection, i, oldArray[i], oldMap);
    }
```

- 记录offset，每次发现了delete就记录+1
- 上面解释过，如果只有新record有index，旧没有index，就是删除（因为旧的指向新的，新的指向旧的，新record有index说明旧数组里有，旧record没有index说明新数组里没有）
- 如果有有删除的record，添加到mDeletes集合中
- 建立i和oldArray[i]这个对象的diffIdentifier的映射

代码块

```
    runningOffset = 0;

    for (NSInteger i = 0; i < newCount; i++) {
        insertOffsets[i] = runningOffset;
        const IGListRecord record = newResultsArray[i];
        const NSInteger oldIndex = record.index;
        // add to inserts if the opposing index is NSNotFound
        if (record.index == NSNotFound) {
            addIndexToCollection(returnIndexPaths, mInserts, toSection, i);
            runningOffset++;
        } else {
            // note that an entry can be updated /and/ moved
            if (record.entry->updated) {
                addIndexToCollection(returnIndexPaths, mUpdates, fromSection,
oldIndex);
            }

            // calculate the offset and determine if there was a move
            // if the indexes match, ignore the index
```

```
            const NSInteger insertOffset = insertOffsets[i];
            const NSInteger deleteOffset = deleteOffsets[oldIndex];
            if ((oldIndex - deleteOffset + insertOffset) != i) {
                id move;
                if (returnIndexPaths) {
                    NSIndexPath *from = [NSIndexPath indexPathForItem:oldIndex
inSection:fromSection];
                    NSIndexPath *to = [NSIndexPath indexPathForItem:i
inSection:toSection];
                    move = [[IGListMoveIndexPath alloc] initWithFrom:from
to:to];
                } else {
                    move = [[IGListMoveIndex alloc] initWithFrom:oldIndex
to:i];
                }
                [mMoves addObject:move];
            }
        }

        addIndexToMap(returnIndexPaths, toSection, i, newArray[i], newMap);
    }
```

- 重置runningOffset，建立insetOffsets到每个newRecord的映射，即这个数组下标和newRecord一一对应
- 根据record.index的参数判断它是不是insert，如果是就添加到mInserts里，现在相当于newRecord.index没有值，说明它没有old，所以是一个insert
- 如果不是insert，说明是一个update/delete
- 把这个update/mode记录添加进mUpdates集合
- 最后做一个判断，把符合要求的对象添加进mMoves

5.2.1 如何判断Move?

这就要好好分析一下deleteOffset和insertOffset了

这里的i好理解，就是该元素在新数组的下标，oldIndex也好理解，就是之前record记录的它在old数组中的下标

代码块

```
    NSInteger runningOffset = 0;
    for (NSInteger i = 0; i < oldCount; i++) {
        deleteOffsets[i] = runningOffset;
        const IGListRecord record = oldResultsArray[i];
        if (record.index == NSNotFound) {
            runningOffset++;
```

```
        }
    }
```

- 删除其它的逻辑代码来看，这里我们可以把原数组看作[1, 0, 0, 0, 0, 1]，0代表没有发生删除，1代表发生了删除
- 所以runningOffset的值表示着这个下标之前，发生了几次删除，所以deleteOffsets数组实际上是一个前缀和数组

代码块

```
    runningOffset = 0;

    for (NSInteger i = 0; i < newCount; i++) {
        insertOffsets[i] = runningOffset;
        const IGListRecord record = newResultsArray[i];
        const NSInteger oldIndex = record.index;

        if (record.index == NSNotFound) {
            addIndexToCollection(returnIndexPaths, mInserts, toSection, i);
            runningOffset++;
        }
    }
}
```

- 这里关于insert记录也是一样，insertOffsets每个下标记录着当前位置之前发生了多少次insert

代码块

```
if ((oldIndex - deleteOffset + insertOffset) != i)
```

- 尝试理解这个判断
- 元素在旧数组中的位置 - 它之前发生的删除次数 + 它之前发生的插入次数
- oldIndex在这里相当于是一个确定的变量，它一定有值，如果没有值会是一个insert，也不会进入这个分支

为什么不直接对比新旧index?

- 直接对比新旧index：因为需要遍历oldRecords，所以在这里时间复杂度会是O(N^2)

所以考虑到时间复杂度，这里使用计算的方式来进行比较

在这里，oldIndex - 它前面发生过删除的次数 + 它前面发生过的insert次数，就是判断它还在不在原位置

代码块

```
    return [[IGListIndexSetResult alloc] initWithInserts:mInserts
                                                 deletes:mDeletes
                                                 updates:mUpdates
                                                   moves:mMoves
                                               oldIndexMap:oldMap
                                               newIndexMap:newMap];
```

最后它返回了一个IndexSetResult，在这个对象里，其它方法都是根据存储的moves之类的集合直接返回结果，我们只分析做了处理的结果

代码块

```
- (NSInteger)changeCount {
    return self.inserts.count + self.deletes.count + self.updates.count + self.moves.count;
}

- (IGListIndexSetResult *)resultForBatchUpdates {
    NSMutableIndexSet *deletes = [self.deletes mutableCopy];
    NSMutableIndexSet *inserts = [self.inserts mutableCopy];
    NSMutableIndexSet *filteredUpdates = [self.updates mutableCopy];

    NSArray<IGListMoveIndex *> *moves = self.moves;
    NSMutableArray<IGListMoveIndex *> *filteredMoves = [moves mutableCopy];

    // convert all update+move to delete+insert
    const NSInteger moveCount = moves.count;
    for (NSInteger i = moveCount - 1; i >= 0; i--) {
        IGListMoveIndex *move = moves[i];
        if ([filteredUpdates containsIndex:move.from]) {
            [filteredMoves removeObjectAtIndex:i];
            [filteredUpdates removeIndex:move.from];
            [deletes addIndex:move.from];
            [inserts addIndex:move.to];
        }
    }

    // iterate all new identifiers. if its index is updated, delete from the old index and insert the new index
    for (id<NSObject> key in [_oldIndexMap keyEnumerator]) {
        const NSInteger index = [[_oldIndexMap objectForKey:key] integerValue];
        if ([filteredUpdates containsIndex:index]) {
            [deletes addIndex:index];
            [inserts addIndex:[[_newIndexMap objectForKey:key] integerValue]];
        }
    }
```

```
33
34      return [[IGListIndexSetResult alloc] initWithInserts:inserts
35                                                   deletes:deletes
36                                                   updates:[NSIndexSet new]
37                                                     moves:filteredMoves
38                                                oldIndexMap:_oldIndexMap
39                                                newIndexMap:_newIndexMap];
40  }
```

- 第一个方法很好理解，改变的数量就是这些相加，不过有个问题，它们之间是交集吗？
  - delete、insert、和updates/moves互不相交
  - updates和moves是相交的，因为update的逻辑很简单，只要指针变化并且identifier变化就是更新了，move涉及到了位移，所以它们之间是个交集
- 在这里可以知道，changeCount > 100就全量更新，这个changeCount其实是有水份的，因为moves/updates很可能有相同的值，但是都算进了数量里

第二个方法

把 update+move 统一转为 delete+insert

- 解决“同一个 cell 既要 move 又要 update”在 iOS 动画里的兼容性（UIKit 的 move+reload 动画有坑，通常推荐变成 delete+insert 两步来保证动画和数据一致性）。
- 所以代码第一段遍历所有move，如果 move 的 from 位置也在 updates 里，说明这个元素即将被 move 并且内容还要更新。则
- 移除这个move（不做move动画）
- 把 update 变成 delete+insert（删除老位置，插入新位置）

这样做的目的是保证 UIKit 能原子性地正确执行批量动画，避免因 move+update 产生动画/数据错乱。

再次处理所有update，将其也变成 delete+insert

- 第二个循环遍历 oldIndexMap，找出所有“内容有更新的元素”
- 对于这些元素，也同样用 delete（旧index）+ insert（新index）来代替单纯的 reload，原因类似：让动画表现和数据变更保持原子一致。

构造最终结果

- 用最终处理过的 inserts、deletes、moves 返回新的 IGListIndexSetResult
- updates 直接用空集，因为都被转成 delete+insert 了。

5.3 时间复杂度

虽然DiffKit多次遍历oldArray/newArray，但是没有嵌套的循环，所以时间复杂度是O（N）

使用到了records、map等数据结构，空间复杂度是O（N）

6. 最后

- IGListKit无论是编程规范：单一职责、防御式编程、函数最小化、函数代码自解释、职责拆分、细节和性能都做的非常好
- 它使用sectionController对不同section进行解藕，用object映射sectionController的方式非常好的管理了object to section之间的映射，让业务只需要关心model的变化，而adapter处理它们之间的映射和顺序关系
- 面向协议编程：功能解藕，面向能力编程，接口最小化

7. 关于异步调用

本文关于异步更新和coalescer、合并更新有多处错误，在这里纠正一下

reloadDataTransaction、dataSourceChangeTransaction都是在主线程上「同步」执行的，它们执行期间如果有其它地方调用了更新方法，这个方法会在主线程的队列里排队，直到当前执行的方法执行完，才会按顺序执行队列里的方法

因此不会出现合并transaction的情况

代码块

```
- (BOOL)cancel {
    // This transaction is synchronous
    return NO;
}
```

- 这是dataSourceChangeTransaction，reloadDataTransaction是一样的，它们不能被取消，更新是同步执行的

所以只有BatchUpdateTransaction可能是异步的，在它执行期间可能会出现合并transaction的情况，原因如下：

1. 如果业务配置了updater执行diff在后台线程，diff将会在后台线程执行

代码块

```
static void _regularPerformDiffWithData(IGListTransitionData *_Nullable data,
                                        BOOL allowsBackground,
                                        IGListDiffExecutorCompletion
completion) {
    if (allowsBackground) {
        dispatch_async(dispatch_get_global_queue(QOS_CLASS_USER_INITIATED, 0),
^{
```

```
            IGListIndexSetResult *result = IGListDiff(data.fromObjects, data.toObjects, IGListDiffEquality);
            dispatch_async(dispatch_get_main_queue(), ^{
                completion(result, allowsBackground);
            });
        });
    } else {
        IGListIndexSetResult *result = IGListDiff(data.fromObjects, data.toObjects, IGListDiffEquality);
        completion(result, allowsBackground);
    }
}
```

在这期间，如果有其它更新发生了，将会被放进builder中，相当于更新被合并

而且如果在diff期间有dataSourceChange进来了，这一次的transaction更新将会被打断（准确的说是打断以后，被合并进新的transaction里执行）

代码块

```
- (BOOL)cancel {
    if (_mode != IGListBatchUpdateTransactionModeCancellable) {
        return NO;
    }
    _mode = IGListBatchUpdateTransactionModeCancelled;
    return YES;
}

- (void)_didDiff:(IGListIndexSetResult *)diffResult onBackground:(BOOL)onBackground {
    if (self.mode == IGListBatchUpdateTransactionModeCancelled) {
        return;
    }

    self.mode = IGListBatchUpdateTransactionModeNotCancellable;
}
```

因为它在diff执行完之后，才会在主线程调用

2. 如果BatchUpdateTransaction执行的是_applyDiff(还有可能是_reload、_bail)，在这期间如果有其它更新进来了，还是会被放进builder里进行合并，包括dataSourceChangeTransaction

代码块

```
- (void)_applyDiff:(IGListIndexSetResult *)diffResult {
    @try {
```

```
        if (self.animated) {
            [self.collectionView performBatchUpdates:updates completion:completion];
        } else {
            [UIView performWithoutAnimation:^{
                [self.collectionView performBatchUpdates:updates completion:completion];
            }];
        }
    }
}
```

[collectionView performBatchUpdates:completion:]其中completion会被异步提交到下一帧执行，这期间主线程可以执行其它方法

而其它两个更新方法都是同步执行，也就是_reload、_bail

8. 其它

8.1 为什么ChangeDataSourceTransaction合并了其它transaction但是不调用它的方法?

代码块

```
- (void)addChangesFromBuilder:(IGListUpdateTransactionBuilder *)builder {
    if (!builder) {
        return;
    }

    self.mode = MAX(self.mode, builder.mode);

    // Section update
    self.animated = self.animated && builder.animated;
    self.sectionDataBlock = self.sectionDataBlock ?: builder.sectionDataBlock;
    self.applySectionDataBlock = self.applySectionDataBlock ?: builder.applySectionDataBlock;

    // Item updates
    [self.itemUpdateBlocks addObjectsFromArray:builder.itemUpdateBlocks];

    // Reload
    self.reloadBlock = self.reloadBlock ?: builder.reloadBlock;

    // All
    self.collectionViewBlock = self.collectionViewBlock ?: builder.collectionViewBlock;
```

```
    [self.completionBlocks addObjectsFromArray:builder.completionBlocks];
}
```

在这段代码里，可以看到，如果changeDataSource取消了当前执行的transaction，它会合并当前正在执行的builder，包括它的applySectionDataBlock、itemUpdateBlock

代码块

```
        case IGListUpdateTransactionBuilderModeDataSourceChange: {
            IGListDataSourceChangeBlock dataSourceChangeBlock = self.dataSourceChangeBlock;
            if (!dataSourceChangeBlock) {
                return nil;
            }
            return [[IGListDataSourceChangeTransaction alloc] initWithChangeBlock:dataSourceChangeBlock

itemUpdateBlocks:self.itemUpdateBlocks

completionBlocks:self.completionBlocks];
        }
```

在创建的时候，它的确加入了itemUpdateBlocks、completionBlocks，但是没有applySectionDataBlock

代码块

```
- (void)begin {
    // Item updates must not send mutations to the collection view while we are reloading
    _state = IGListBatchUpdateStateExecutingBatchUpdateBlock;

    // Execute all stored item update blocks even if all cells will get reloaded. the actual collection view
    // mutations will be discarded, but clients are encouraged to put their actual /data/ mutations inside the
    // update block as well, so if we don't execute the block the changes will never happen
    for (IGListItemUpdateBlock itemUpdateBlock in _itemUpdateBlocks) {
        itemUpdateBlock();
    }

    _state = IGListBatchUpdateStateExecutedBatchUpdateBlock;

    // Apply dataSource change
```

```
    if (_block) {
        _block();
    }

    for (IGListUpdatingCompletion completion in _completionBlocks) {
        completion(YES);
    }

    // Execute any completion blocks from item updates. Added after item blocks are executed in order to capture any
    // re-entrant updates.
    NSArray *inUpdateCompletionBlocks = [_inUpdateCompletionBlocks copy];
    for (IGListUpdatingCompletion completion in inUpdateCompletionBlocks) {
        completion(YES);
    }

    _state = IGListBatchUpdateStateIdle;
}
```

为什么调用itemUpdateBlock这里已经解释了，注解翻译如下：

> 即便这次我们会 reload 所有 cell，我们也要把 **itemUpdateBlocks** 全部执行一遍。
> 这些 block 里**应该写“数据变更”**（而不是直接操作 collection view）。
> 如果不执行它们，那么**数据层的变更永远不会发生**。

为什么不调用applySectionDataBlock

猜测：因为当前更换了数据源，applySectionBlock应该算是旧的数据了，数据源都更换了，再调用它就没有意义了

在updater执行完之后，会调用_updateObjects更新数据源，创建或者更新sectionController，然后完成数据源到sectionController的映射

代码块

```
- (void)setDataSource:(id<IGListAdapterDataSource>)dataSource {
    if (_dataSource == dataSource) {
        return;
    }

    [_updater performDataSourceChange:^{
        self->_dataSource = dataSource;

        // Invalidate the collectionView internal section & item counts, as if its dataSource changed.
        self->_collectionView.dataSource = nil;
        self->_collectionView.dataSource = self;

```

```
        // Sync the dataSource <> adapter
        [self _updateObjects];
    }];
}

- (void)_updateObjects {
    if (_collectionView == nil) {
        // If we don't have a collectionView, we can't do much.
        return;
    }
    id<IGListAdapterDataSource> dataSource = _dataSource;
    NSArray *uniqueObjects =
objectsWithDuplicateIdentifiersRemoved([dataSource
objectsForListAdapter:self]);
    [self _updateObjects:uniqueObjects dataSource:dataSource];
}
```