

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

```

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      IGListSectionControllerPushThread(self.viewController, self);
8
9      [objects enumerateObjectsUsingBlock:^(id object, NSUInteger idx, BOOL
        *stop) {
10         IGListSectionController *sectionController = [map
            sectionControllerForObject:object];
11
12         if (sectionController == nil) {
13             sectionController = [dataSource listAdapter:self
                sectionControllerForObject:object];
14         }
15
16         // in case the section controller was created outside of -
listAdapter:sectionControllerForObject:
17         sectionController.collectionContext = self;
18         sectionController.viewController = self.viewController;
19
20         [sectionControllers addObject:sectionController];
21         [validObjects addObject:object];
22     }];
23
24     // clear the view controller and collection context
25     IGListSectionControllerPopThread();
26
27     return [[IGListTransitionData alloc] initWithObjects:map.objects
28         toObjects:validObjects
29         toSectionControllers:sectionControllers];
30 }

```

- 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

```

1  - (void)_updateWithData:(IGListTransitionData *)data {
2      _isInObjectUpdateTransaction = YES;
3
4      IGListSectionMap *map = self.sectionMap;
5
6      // Note: We use an array, instead of a set, because the updater should
have dealt with duplicates already.
7      NSMutableArray *updatedObjects = [NSMutableArray new];
8
9      for (id object in data.toObjects) {
10         // check if the item has changed instances or is new
11         NSInteger oldSection = [map sectionForObject:object];
12         if (oldSection == NSNotFound || [map objectForKey:oldSection] !=
object) {
13             [updatedObjects addObject:object];
14         }
15     }
16
17     [map updateWithObjects:data.toObjects
sectionControllers:data.toSectionControllers];
18
19     for (id object in updatedObjects) {
20         [[map sectionControllerForObject:object] didUpdateToObject:object];
21     }
22
23     [self _updateBackgroundView];
24
25     _isInObjectUpdateTransaction = NO;
26 }

```

- 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

```
1  typedef NSInteger (NSInteger, IGListUpdateTransactionBuilderMode) {
2      /// The lowest priority is a batch-update, because a reload or dataSource
take care of any changes.
3      IGListUpdateTransactionBuilderModeBatchUpdate,
4      /// The second priority is reloading all data.
5      IGListUpdateTransactionBuilderModeReload,
6      /// The highest priority is changing the 'UICollectionView' dataSource.
7      IGListUpdateTransactionBuilderModeDataSourceChange,
8  };
9
10 - (void)addSectionBatchUpdateAnimated:(BOOL)animated
11         collectionViewBlock:
12         (IGListCollectionViewBlock)collectionViewBlock
13         sectionDataBlock:
14         (IGListTransitionDataBlock)sectionDataBlock
15         applySectionDataBlock:
16         (IGListTransitionDataApplyBlock)applySectionDataBlock
17         completion:(IGListUpdatingCompletion)completion {
18     self.mode = MAX(self.mode, IGListUpdateTransactionBuilderModeBatchUpdate);
19 }
```

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

```
1  - (void)performUpdateWithCollectionViewBlock:
2      (IGListCollectionViewBlock)collectionViewBlock
3      animated:(BOOL)animated
4      sectionDataBlock:
5      (IGListTransitionDataBlock)sectionDataBlock
6      applySectionDataBlock:
7      (IGListTransitionDataApplyBlock)applySectionDataBlock
```

```

5             completion:(nullable
IGListUpdatingCompletion)completion {
6         [self.transactionBuilder addSectionBatchUpdateAnimated:animated
7             collectionViewBlock:collectionViewBlock
8             sectionDataBlock:sectionDataBlock
9
        applySectionDataBlock:applySectionDataBlock
10             completion:completion];
11
12         [self _queueUpdateIfNeeded];
13     }
14
15     - (void)reloadDataWithCollectionViewBlock:
(IGListCollectionViewBlock)collectionViewBlock
16         reloadUpdateBlock:
(IGListReloadUpdateBlock)reloadUpdateBlock
17         completion:(nullable
IGListUpdatingCompletion)completion {
18         [self.transactionBuilder
addReloadDataWithCollectionViewBlock:collectionViewBlock
19
        reloadBlock:reloadUpdateBlock
20             completion:completion];
21
22         [self _queueUpdateIfNeeded];
23     }
24
25     - (void)performDataSourceChange:(IGListDataSourceChangeBlock)block {
26         // Unlike the other "performs", we need the dataSource change to be
synchronous.
27         // Which means we need to cancel the current transaction, flatten the
changes from
28         // both the current transtion and builder, and execute that new
transaction.
29
30         if (!self.transaction
31             && ![self.transactionBuilder hasChanges]
32             && !IGListExperimentEnabled(self.experiments,
IGListExperimentRemoveDataSourceChangeEarlyExit)) {
33             // If nothing is going on, lets take a shortcut.
34             block();
35             return;
36         }
37
38         IGListUpdateTransactionBuilder *builder = [IGListUpdateTransactionBuilder
new];
39         [builder addDataSourceChange:block];

```

```

40
41     // Lets try to cancel any current transactions.
42     if ([self.transaction cancel] && self.lastTransactionBuilder) {
43         // We still need to apply the item-updates and completion-blocks, so
         lets merge the builders.
44         [builder addChangesFromBuilder:(IGListUpdateTransactionBuilder
         *)self.lastTransactionBuilder];
45     }
46
47     // Lets merge pending changes
48     [builder addChangesFromBuilder:self.transactionBuilder];
49
50     // Clear the current state
51     self.transaction = nil;
52     self.lastTransactionBuilder = nil;
53     self.transactionBuilder = builder;
54
55     // Update synchronously
56     [self update];
57 }

```

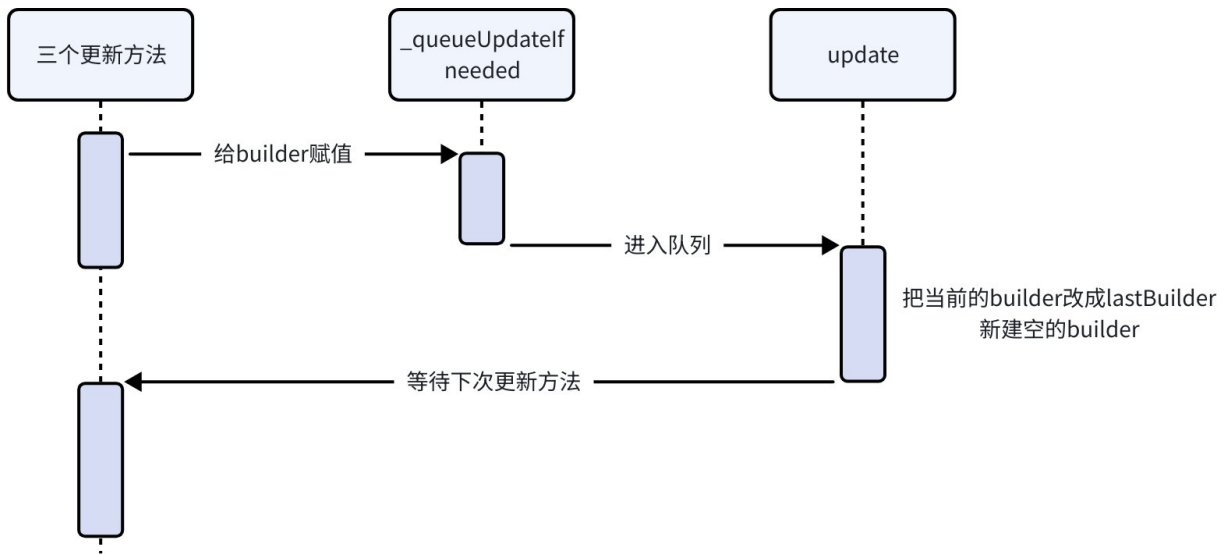
- 这三个方法，当不定义coalescer的行为的时候它们最终调用的更新方法是一致的，就是update
- _queueUpdateIfNeeded是在需要的时候把transaction按照业务定义的方式进行延迟更新，最后也会调用update

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，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？

- 其实还是因为coalescer，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

```

1      [self.transactionBuilder addItemBatchUpdateAnimated:animated
2                                     collectionViewBlock:collectionViewBlock
3                                     itemUpdates:itemUpdates
4                                     completion:completion];
  
```

- 因为coalescer可能会产生不同的时序问题，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

```

1  - (void)update {
2      id<IGListUpdateTransactable> transaction = [self.transactionBuilder
3      buildWithConfig:config delegate:_delegate updater:self];
4      self.transaction = transaction;
5      self.lastTransactionBuilder = self.transactionBuilder;
6      self.transactionBuilder = [IGListUpdateTransactionBuilder new];
7  }
8  - (nullable id<IGListUpdateTransactable>)buildWithConfig:
    (IGListUpdateTransactationConfig)config
  
```

```

9                                     delegate:(nullable
id<IGListAdapterUpdaterDelegate>)delegate
10                                     updater:(IGListAdapterUpdater
*)updater {
11     switch (self.mode) {
12         case IGListUpdateTransactionBuilderModeBatchUpdate: {
13             IGListCollectionViewBlock collectionViewBlock =
self.collectionViewBlock;
14             return [[IGListBatchUpdateTransaction alloc]
initWithCollectionViewBlock:collectionViewBlock
15             updater:updater
16             delegate:delegate
17             config:config
18             animated:self.animated
19             sectionDataBlock:self.sectionDataBlock
20             applySectionDataBlock:self.applySectionDataBlock
21             itemUpdateBlocks:self.itemUpdateBlocks
22             completionBlocks:self.completionBlocks];
23         }
24         case IGListUpdateTransactionBuilderModeReload: {
25             IGListReloadUpdateBlock reloadBlock = self.reloadBlock;
26             IGListCollectionViewBlock collectionViewBlock =
self.collectionViewBlock;
27             return [[IGListReloadTransaction alloc]
initWithCollectionViewBlock:collectionViewBlock
28             updater:updater
29             delegate:delegate
30             reloadBlock:reloadBlock
31             itemUpdateBlocks:self.itemUpdateBlocks
32             completionBlocks:self.completionBlocks];
33         }
34         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[viewController didUpdateToObjects]



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didUpdateToObjects

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 objectForKey: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

```
1  - (void)performUpdateWithCollectionViewBlock:
    (IGListCollectionViewBlock)collectionViewBlock
2
3      animated:(BOOL)animated
4      itemUpdates:(void (^)(void))itemUpdates
5      completion:(void (^)(BOOL))completion {
6      // if already inside the execution of the update block, immediately unload
    the itemUpdates block.
7      // the completion blocks are executed later in the lifecycle, so that
    still needs to be added to the batch
8      if ([self isInDataUpdateBlock]) {
9          if (completion != nil) {
10             [self.transaction addCompletionBlock:completion];
11         }
12         itemUpdates();
13     } else {
14         [self.transactionBuilder addItemBatchUpdateAnimated:animated
15             collectionViewBlock:collectionViewBlock
16             itemUpdates:itemUpdates
17             completion:completion];
18         [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

```
1 [adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx){
2     // 在block里又触发了item的增删改
3     [adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx2){
4         // 递归/嵌套
5     } completion:nil];
6 } completion:nil];
7
```

- 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

```
1 - (void)_adaptiveDispatchUpdateForView:(nullable UIView *)view {
2     const IGListAdaptiveCoalescingExperimentConfig config =
        _adaptiveCoalescingExperimentConfig;
3     const NSTimeInterval timeSinceLastUpdate = -[_lastUpdateStartDate
        timeIntervalSinceNow];
4     const BOOL isVisible = _isViewVisible(view, config);
5     const NSTimeInterval currentCoalescenceInterval = _coalescenceInterval;
6
7     if (isVisible) {
8         if (!_lastUpdateStartDate || timeSinceLastUpdate >
            currentCoalescenceInterval) {
```

```

9          // It's been long enough, so lets reset interval and perform
update right away
10         _coalescenceInterval = config.minInterval;
11         [self _performUpdate];
12         return;
13     } else {
14         // If we keep hitting the delay, lets increase it.
15         _coalescenceInterval = MIN(currentCoalescenceInterval +
config.intervalIncrement, config.maxInterval);
16     }
17 }
18
19 // Delay by the time remaining in the interval
20 const NSTimeInterval remainingTime = isVisible ?
(currentCoalescenceInterval - timeSinceLastUpdate) : config.maxInterval;
21 const NSTimeInterval remainingTimeCapped = MAX(remainingTime, 0);
22
23 _hasQueuedUpdate = YES;
24 __weak __typeof__(self) weakSelf = self;
25 dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)
(remainingTimeCapped * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
26     [weakSelf _performUpdate];
27 });
28 }

```

- 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？

合并更新的意思就是，多次「全量更新」，下面这个方法，只会调用最后一次更新（是不是和RunLoop定义NSTimer的行为有点像？）

代码块

```

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

```

而sectionController内部的更新方法，「局部更新」会被合并调用

代码块

```

1  - (void)addItemBatchUpdateAnimated:(BOOL)animated
2          collectionViewBlock:
(IGListCollectionViewBlock)collectionViewBlock
3          itemUpdates:(IGListItemUpdateBlock)itemUpdates

```

```

4             completion:(nullable
IGListUpdatingCompletion)completion {
5         self.mode = MAX(self.mode, IGListUpdateTransactionBuilderModeBatchUpdate);
6
7         // disabled animations will always take priority
8         // reset to YES in -cleanupState
9         self.animated = self.animated && animated;
10        self.collectionViewBlock = collectionViewBlock;
11
12        [self.itemUpdateBlocks addObject:itemUpdates];
13
14        IGListUpdatingCompletion localCompletion = completion;
15        if (localCompletion) {
16            [self.completionBlocks addObject:localCompletion];
17        }
18    }

```

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

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,
9                               self.config.adaptiveDiffingExperimentConfig,
10                              ^(IGListIndexSetResult * _Nonnull result, BOOL
onBackground) {
11          [weakSelf _didDiff:result onBackground:onBackground];
12      });
13  }

```

- 这里可以看到sectionController局部更新数据的performUpdate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

```

1  - (void)_applyDiff:(IGListIndexSetResult *)diffResult {
2      [self.delegate listAdapterUpdater:self.updater
3      willPerformBatchUpdatesWithCollectionView:self.collectionView
4          fromObjects:self.sectionData.fromObjects
5          toObjects:self.sectionData.toObjects
6          listIndexSetResult:diffResult
7          animated:self.animated];
8      void (^updates)(void) = ^ {
9          [self _applyDataUpdates];
10         [self _applyCollectionViewUpdates:diffResult];
11     };
12     @try {
13         if (self.animated) {
14             [self.collectionView performBatchUpdates:updates
15             completion:completion];
16         } else {
17             [UIView performWithoutAnimation:^(
18                 [self.collectionView performBatchUpdates:updates
19                 completion:completion];
20             )];
21     }
22 }

```

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

代码块

```

1  - (void)_applyDataUpdates {
2      self.state = IGListBatchUpdateStateExecutingBatchUpdateBlock;
3
4      // run the update block so that the adapter can set its items. this makes
5      sure that just before the update is
6      // committed that the data source is updated to the /latest/ "toObjects".
7      this makes the data source in sync
8      // with the items that the updater is transitioning to
9      if (self.applySectionDataBlock != nil && self.sectionData != nil) {
10         self.applySectionDataBlock((IGListTransitionData *)self.sectionData);
11     }
12
13     // execute each item update block which should make calls like insert,
14     delete, and reload for index paths
15     // we collect all mutations in corresponding sets on self, then filter
16     based on UICollectionView shortcomings
17     // call after the objectTransitionBlock so section level mutations happen
18     before any items

```

```

14     for (IGListItemUpdateBlock block in self.itemUpdateBlocks) {
15         block();
16     }
17
18     self.state = IGListBatchUpdateStateExecutedBatchUpdateBlock;
19 }

```

- 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

```

1  [adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx){
2      // 在block里又触发了item的增删改
3      [adapter performBatchAnimated:YES updates:^(id<IGListBatchContext> ctx2){
4          // 递归/嵌套
5          } completion:nil];
6  } completion:nil];
7

```

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
- performUpdate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

```
1  - (NSInteger)changeCount {
2      return self.inserts.count + self.deletes.count + self.updates.count +
      self.moves.count;
3  }
```

5. diffKit做了什么

IGListBatchUpdateTransaction，可以看到transaction会根据sectionData进行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,
9                              self.config.adaptiveDiffingExperimentConfig,
10                             ^(IGListIndexSetResult * _Nonnull result, BOOL
      onBackground) {
11         [weakSelf _didDiff:result onBackground:onBackground];
12     });
13 }
```

全局搜索，sectionData的赋值只有一处

代码块

```
1  - (instancetype)initWithCollectionViewBlock:
      (IGListCollectionViewBlock)collectionViewBlock
2      updater:(IGListAdapterUpdater *)updater
```

```

3         delegate:
        (id<IGListAdapterUpdaterDelegate>)delegate
4         config:
        (IGListUpdateTransactationConfig)config
5         animated:(BOOL)animated
6         sectionDataBlock:
        (IGListTransitionDataBlock)sectionDataBlock
7         applySectionDataBlock:
        (IGListTransitionDataApplyBlock)applySectionDataBlock
8         itemUpdateBlocks:(NSArray<IGListItemUpdateBlock>
*)itemUpdateBlocks
9         completionBlocks:(NSArray<IGListUpdatingCompletion>
*)completionBlocks {
10     if (self = [super init]) {
11         _sectionData = sectionDataBlock ? sectionDataBlock() : nil;
12     }
13     return self;
14 }

```

- 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

```

1 void IGListPerformDiffWithData(IGListTransitionData *Nullable data,
2                               UIView *view,
3                               BOOL allowsBackground,
4                               IGListAdaptiveDiffingExperimentConfig
5                               adaptiveConfig,
6                               IGListDiffExecutorCompletion completion) {
7     if (!completion) {
8         return;
9     }
10    if (adaptiveConfig.enabled) {
11        _adaptivePerformDiffWithData(data, view, allowsBackground,
12        adaptiveConfig, completion);
13    } else {
14        // Just to be safe, lets keep the original code path intact while
15        // adaptive diffing is still an experiment.
16        _regularPerformDiffWithData(data, allowsBackground, completion);
17    }
18 }

```

- 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

```
1  @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

```
1  @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

```
1  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

```
1  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

```
1      vector<IGListRecord> newResultsArray(newCount);
2      for (NSInteger i = 0; i < newCount; i++) {
3          id<NSObject> key = IGListTableKey(newArray[i]);
4          IGListEntry &entry = table[key];
5          entry.newCounter++;
6
7          // add NSNotFound for each occurrence of the item in the new array
8          entry.oldIndexes.push(NSNotFound);
9
10         // note: the entry is just a pointer to the entry which is stack-
            allocated in the table
11         newResultsArray[i].entry = &entry;
12     }
```

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

```
1      vector<IGListRecord> oldResultsArray(oldCount);
2      for (NSInteger i = oldCount - 1; i >= 0; i--) {
3          id<NSObject> key = IGListTableKey(oldArray[i]);
4          IGListEntry &entry = table[key];
5          entry.oldCounter++;
6
7          // push the original indices where the item occurred onto the index
            stack
8          entry.oldIndexes.push(i);
9
10         // note: the entry is just a pointer to the entry which is stack-
            allocated in the table
11         oldResultsArray[i].entry = &entry;
12     }
```

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

```

1      for (NSInteger i = 0; i < newCount; i++) {
2          IGListEntry *entry = newResultsArray[i].entry;
3
4          // grab and pop the top original index. if the item was inserted this
will be NSNotFound
5          NSAssert(!entry->oldIndexes.empty(), @"Old indexes is empty while
iterating new item %li. Should have NSNotFound", (long)i);
6          const NSInteger originalIndex = entry->oldIndexes.top();
7          entry->oldIndexes.pop();
8
9          if (originalIndex < oldCount) {
10             const id<IGListDiffable> n = newArray[i];
11             const id<IGListDiffable> o = oldArray[originalIndex];
12             switch (option) {
13                 case IGListDiffPointerPersonality:
14                     // flag the entry as updated if the pointers are not the
same
15                     if (n != o) {
16                         entry->updated = YES;
17                     }
18                     break;
19                 case IGListDiffEquality:
20                     // use -[IGListDiffable isEqualToDiffableObject:] between
both version of data to see if anything has changed
21                     // skip the equality check if both indexes point to the
same object
22                     if (n != o && ![n isEqualToDiffableObject:o]) {
23                         entry->updated = YES;

```

```

24         }
25         break;
26     }
27 }
28 if (originalIndex != NSNotFound
29     && entry->newCounter > 0
30     && entry->oldCounter > 0) {
31     // if an item occurs in the new and old array, it is unique
32     // assign the index of new and old records to the opposite index
    (reverse lookup)
33     newResultsArray[i].index = originalIndex;
34     oldResultsArray[originalIndex].index = i;
35 }
36 }

```

- 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
 - top是oldIndex，且栈中只有这一个元素
 - 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

```

1     id mInserts, mMoves, mUpdates, mDeletes;
2     if (returnIndexPaths) {
3         mInserts = [NSMutableArray<NSIndexPath *> new];
4         mMoves = [NSMutableArray<IGListMoveIndexPath *> new];
5         mUpdates = [NSMutableArray<NSIndexPath *> new];
6         mDeletes = [NSMutableArray<NSIndexPath *> new];
7     } else {
8         mInserts = [NSMutableIndexSet new];
9         mMoves = [NSMutableArray<IGListMoveIndex *> new];
10        mUpdates = [NSMutableIndexSet new];

```

```

11         mDeletes = [NSMutableIndexSet new];
12     }

```

- 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

```

1     vector<NSInteger> deleteOffsets(oldCount), insertOffsets(newCount);
2     NSInteger runningOffset = 0;
3     // iterate old array records checking for deletes
4     // increment offset for each delete
5     for (NSInteger i = 0; i < oldCount; i++) {
6         deleteOffsets[i] = runningOffset;
7         const IGListRecord record = oldResultsArray[i];
8         // if the record index in the new array doesn't exist, its a delete
9         if (record.index == NSNotFound) {
10             addIndexToCollection(returnIndexPaths, mDeletes, fromSection, i);
11             runningOffset++;
12         }
13
14         addIndexToMap(returnIndexPaths, fromSection, i, oldArray[i], oldMap);
15     }

```

- 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）
- 如果有有删除的记录，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

```

1     runningOffset = 0;
2
3     for (NSInteger i = 0; i < newCount; i++) {
4         insertOffsets[i] = runningOffset;
5         const IGListRecord record = newResultsArray[i];
6         const NSInteger oldIndex = record.index;
7         // add to inserts if the opposing index is NSNotFound
8         if (record.index == NSNotFound) {
9             addIndexToCollection(returnIndexPaths, mInserts, toSection, i);
10            runningOffset++;
11        } else {
12            // note that an entry can be updated /and/ moved

```

```

13         if (record.entry->updated) {
14             addIndexToCollection(returnIndexPaths, mUpdates, fromSection,
oldIndex);
15         }
16
17         // calculate the offset and determine if there was a move
18         // if the indexes match, ignore the index
19         const NSInteger insertOffset = insertOffsets[i];
20         const NSInteger deleteOffset = deleteOffsets[oldIndex];
21         if ((oldIndex - deleteOffset + insertOffset) != i) {
22             id move;
23             if (returnIndexPaths) {
24                 NSIndexPath *from = [NSIndexPath indexPathForItem:oldIndex
inSection:fromSection];
25                 NSIndexPath *to = [NSIndexPath indexPathForItem:i
inSection:toSection];
26                 move = [[IGListMoveIndexPath alloc] initWithFrom:from
to:to];
27             } else {
28                 move = [[IGListMoveIndex alloc] initWithFrom:oldIndex
to:i];
29             }
30             [mMoves addObject:move];
31         }
32     }
33
34     addIndexToMap(returnIndexPaths, toSection, i, newArray[i], newMap);
35 }

```

- 重置runningOffset，建立insertOffsets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/delete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


```

1  NSInteger runningOffset = 0;
2  for (NSInteger i = 0; i < oldCount; i++) {
3      deleteOffsets[i] = runningOffset;
4      const IGListRecord record = oldResultsArray[i];
5      if (record.index == NSNotFound) {
6          runningOffset++;
7      }
8  }

```

- 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

```

1      runningOffset = 0;
2
3      for (NSInteger i = 0; i < newCount; i++) {
4          insertOffsets[i] = runningOffset;
5          const IGListRecord record = newResultsArray[i];
6          const NSInteger oldIndex = record.index;
7
8          if (record.index == NSNotFound) {
9              addIndexToCollection(returnIndexPaths, mInserts, toSection, i);
10             runningOffset++;
11         }
12     }
13 }

```

- 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

```

1  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

```
1      return [[IGListIndexSetResult alloc] initWithInserts:mInserts
2                                          deletes:mDeletes
3                                          updates:mUpdates
4                                          moves:mMoves
5                                          oldIndexMap:oldMap
6                                          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

```
1  - (NSInteger)changeCount {
2      return self.inserts.count + self.deletes.count + self.updates.count +
3      self.moves.count;
4  }
5  - (IGListIndexSetResult *)resultForBatchUpdates {
6      NSMutableIndexSet *deletes = [self.deletes mutableCopy];
7      NSMutableIndexSet *inserts = [self.inserts mutableCopy];
8      NSMutableIndexSet *filteredUpdates = [self.updates mutableCopy];
9
10     NSArray<IGListMoveIndex *> *moves = self.moves;
11     NSMutableArray<IGListMoveIndex *> *filteredMoves = [moves mutableCopy];
12
13     // convert all update+move to delete+insert
14     const NSInteger moveCount = moves.count;
15     for (NSInteger i = moveCount - 1; i >= 0; i--) {
16         IGListMoveIndex *move = moves[i];
17         if ([filteredUpdates containsIndex:move.from]) {
18             [filteredMoves removeObjectAtIndex:i];
19             [filteredUpdates removeIndex:move.from];
20             [deletes addIndex:move.from];
21             [inserts addIndex:move.to];
22         }
23     }
24 }
```

```

25     // iterate all new identifiers. if its index is updated, delete from the
    old index and insert the new index
26     for (id<NSObject> key in [_oldIndexMap keyEnumerator]) {
27         const NSInteger index = [[_oldIndexMap objectForKey:key] integerValue];
28         if ([filteredUpdates containsIndex:index]) {
29             [deletes addIndex:index];
30             [inserts addIndex:[_newIndexMap objectForKey:key] integerValue]];
31         }
32     }
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