Collection View プログラミングガイド

Developer

目次

図、表、リスト

iOSのコレクションビューについて

コレクションビューは、データ項目の順序つき集合を表示する手段のひとつで、レイアウトを柔軟に変更できる、という特徴があります。よくあるのは格子状に項目を配置する方法ですが、iOSのコレクションビューは、単に行と列の形に並べる以外にもさまざまな配置が可能です。視覚的要素のレイアウトを細かく定義することも、サブクラスを定義して必要な処理を実装することにより可能で、しかもそのレイアウトは動的に変更できます。格子、スタック、円状レイアウト、動的に変化するレイアウトその他、およそ想像できる配置は何でも実装できると言ってよいでしょう。

コレクションビューは、表示されるデータと、その表示に用いる視覚的要素を、厳密に区別して扱います。多くの場合、データ管理はもっぱらアプリケーション側が行います。また、データの表示に用いるビューオブジェクトを用意するのもアプリケーション側の役割です。以上の準備が整えば、コレクションビューは上記のビューオブジェクトを取得し、画面上の配置を決めるために必要なあらゆる処理を行います。その際には、ビューの配置や視覚属性を指定する、レイアウトオブジェクトと連携することになります。レイアウトオブジェクトは、アプリケーションの要件に合わせ、サブクラスを定義して使うことも可能です。以上のように、アプリケーション側がデータを用意し、レイアウトオブジェクトが配置情報を管理し、コレクションビューはこの2つを組み合わせて最終的な外観を決める、という役割分担ができあがっています。

はじめに

iOS標準のコレクションビュークラスには、簡単な格子配置を実装するために必要な機能が、すべて組み込まれています。さらに、これを拡張して、独自のレイアウトやそれに基づく動作を実装することも可能です。

コレクションビューはデータ駆動型ビューの視覚的表現を管理する

コレクションビューには、アプリケーション側が用意する、データ駆動型ビューの表示を支援する働きがあります。とは言っても、実際の関与は、該当するビューを取り込み、所定の方法で配置することだけです。ビューの表示と配置を行うだけであって、その中身に関わることはありません。コレクションビュー、そのデータ源、レイアウトオブジェクト、アプリケーション側が用意するカスタムオブジェクトの、4者間のやり取りを把握することが、コレクションビューを利用して、手際よく効率的にアプリケーションを実装する上で重要です。

関連する章: “コレクションビューの基本事項” （9 ページ）、“データ源およびデリゲートの設計” （15 ページ）

フローレイアウトで、格子その他、行指向の表示を実現できる

「フローレイアウト」オブジェクトは、UIKitに付属する具象レイアウトオブジェクトです。各項目が行と列を成して並ぶ、格子状の配置を実装するには、一般にこの「フローレイアウト」オブジェクトを使いますが、このオブジェクトは、実際には行指向のあらゆる配置を実現できます。格子専用ではないので、柔軟性が高く興味深いさまざまな配置を実現可能で、サブクラスを定義して実装する場合と、そのまま使う場合があります。さまざまな寸法の項目が混在している、あるいは項目間の間隔が可変である状況に対応できるほか、独自のヘッダやフッタをつける、余白を独自に制御する、などの処理も、このオブジェクトを直接使って実装できます。一方、サブクラスを定義して独自の処理を実装することにより、動作をさまざまに変更することも可能です。

関連する章: “フローレイアウトの使い方” （28 ページ）

ジェスチャリコグナイザを使ってセルやレイアウトを操作できる

他のビューと同様、コレクションビューにもジェスチャリコグナイザを関与させることにより、ビューの内容を操作できるようになります。コレクションビューでは多数のビューが連携しているので、ここにジェスチャリコグナイザを組み込む、基本的な技法をいくつか理解しておけば、設計や実装に役立つでしょう。ジェスチャリコグナイザを使って、レイアウト属性を調整する、あるいはコレクションビュー内の項目を操作することができます。

関連する章: “ジェスチャ認識機能の組み込み” （34 ページ）

カスタムレイアウトを使えば格子以外の配置も可能である

基本となるレイアウトオブジェクトのサブクラスを定義し、アプリケーションの特性に応じた、カスタムレイアウトを実装できます。カスタムレイアウトを設計するといっても、通常、それほど多くのコードを記述することはありません。しかし、その作業を効率よく進めるためにも、レイアウト処理の流れを把握しておくとよいでしょう。最後の章では、完全なカスタムレイアウトを実装するプロジェクト例を取り上げて説明します。

関連する章: “カスタムレイアウトの作成”（37 ページ）、“カスタムレイアウト：実例”（48 ページ）

必要事項

この資料は、iOSアプリケーションにおいてビューが果たす役割について、充分に理解している方を対象としています。iOSプログラミングが初めてで、iOSのビューアーキテクチャに慣れていない場合は、あらかじめ『*View Programming Guide for iOS*』を読んでおいてください。

関連項目

コレクションビューは、順序つきのデータを表示するという意味で、テーブルビューとも若干関係します。テーブルビューの実装は、（フローレイアウトを用いる）標準的なコレクションビューと同様に、インデックスパス、セル、ビューの回収機構を利用しています。しかし、テーブルビューの視覚的表現は単一列レイアウトを基本としているのに対し、コレクションビューはそれ以外のレイアウトも可能になっています。テーブルビューについて詳しくは、『*Table View Programming Guide for iOS*』を参照してください。

コレクションビューの基本事項

コレクションビューはその中身を画面に表示するために、さまざまなオブジェクトと連携します。その中には、アプリケーション側で独自に用意するべきオブジェクトもあります。たとえばデータ源オブジェクトがそのひとつで、表示する項目数などをコレクションビューに通知する役割があります。また、UIKitに付属する各種オブジェクトも、コレクションビューの基本設計の一翼を担っています。

コレクションビューはテーブルと同様のデータ指向オブジェクトで、実装のためには、アプリケーション側のオブジェクトとの連携が欠かせません。ここに実装するべき処理を把握するためには、コレクションビューがどのように処理しているか、背景となる情報を少し知っておく必要があるでしょう。

コレクションビューは多くのオブジェクトが連携している

コレクションビューは、表示されるデータと、データを画面上に配置、表示する方法とを、明確に分けて管理する設計になっています。アプリケーション側では、表示内容データを適切に管理する責任を負いますが、その視覚的表現の管理には他の多くのオブジェクトが関与します。表 1-1に、UIKitに付属するコレクションビューのクラス群を示します。そのインターフェイス実装における役割によって分類してあります。多くのクラスは、サブクラスを定義することなくそのまま使えるように設計してあるので、通常、コードをほとんど記述しなくてもコレクションビューを実装できます。用意されている動作だけでは足りない場合に限り、サブクラスを定義し、独自の動作を実装してください。

表 1-1 コレクションビューの実装に用いるクラスとプロトコル

目的	クラス/プロトコル	解説
最上位のコンテナ機能と管理機能	`UICollectionView` `UICollectionView-Controller`	`UICollectionView`オブジェクトは、コレクションビューの中身を表示する、可視領域を定義しています。`UIScrollView`を継承することにより、いくらでも広い領域を確保し、スクロールして使うようになっています。また、レイアウトオブジェクトから必要な情報を取得し、データの表示を支援する働きもあります。 `UICollectionViewController`オブジェクトには、コレクションビューに対する、ビューコントローラとしての管理機能があります。必要に応じて利用してください。

目的	クラス/プロトコル	解説
コンテンツ管理	`UICollectionView-DataSource` プロトコル `UICollectionView-Delegate` プロトコル	**データ源オブジェクト**は、コレクションビューに関連する最も重要なオブジェクトで、アプリケーション側に必ず実装しなければなりません。コレクションビューに表示する内容を管理し、その表示に必要なビューを生成する、という役割があります。データ源オブジェクトを実装するためには、`UICollectionView-DataSource`プロトコルに従うオブジェクトが必要です。 コレクションビューのデリゲートオブジェクトは、コレクションビューから送られてくるメッセージを横取りし、ビューの動作をカスタマイズします。たとえば、コレクションビューにおける選択や強調表示の動作を追跡できるのです。データ源オブジェクトと違って必須ではありません。 データ源オブジェクトやデリゲートオブジェクトの実装方法については、“データ源およびデリゲートの設計”（15 ページ）を参照してください。
プレゼンテーション	`UICollectionReusable-View` `UICollectionViewCell`	コレクションビュー内に表示するビューはすべて、`UICollectionReusableView`クラスのインスタンスでなければなりません。このクラスには、コレクションビューと連携して、不要になったビューを回収する仕組みが組み込まれています。ビューが必要になったとき、（新たに生成する代わりに）回収して使うことにより、全体的に処理性能を改善でき、特にスクロール処理において効果が顕著です。 この再利用可能なビューの中でも、`UICollectionViewCell`は特に、主たるデータ項目を表します。

目的	クラス/プロトコル	解説

配置	`UICollectionViewLayout`
	`UICollectionViewLayoutAttributes`
	`UICollectionViewUpdateItem`
		`UICollectionViewLayout`のサブクラスを**レイアウトオブジェクト**と呼びます。コレクションビューに収容される、セルをはじめとする再利用可能なビューの、位置、大きさ、視覚属性を定義する役割があります。
		レイアウト処理に際して、レイアウトオブジェクトはレイアウト属性オブジェクト（`UICollectionViewLayoutAttributes`のインスタンス）を生成して、コレクションビューに対し、セルその他のビューを表示する位置や方法を指示します。
		データ項目をコレクションビューに追加し、ここから削除し、この中で移動すると、レイアウトオブジェクトは`UICollectionViewUpdateItem`のインスタンスを受け取ることになります。このクラスのインスタンスをアプリケーション側で生成する必要はありません。
		レイアウトオブジェクトについて詳しくは、“レイアウトオブジェクトは視覚表現を制御する”（13 ページ）を参照してください。

フローレイアウト	`UICollectionViewFlowLayout` `UICollectionViewDelegateFlowLayout` プロトコル	`UICollectionViewFlowLayout`クラスは具象レイアウトオブジェクトで、格子その他、行指向のレイアウトを実装するために使います。そのまま使う方法と、フローデリゲートオブジェクトと組み合わせて、レイアウト情報を動的にカスタマイズする方法があります。

図1-1に、コレクションビューに関連する主なオブジェクト間の関係を示します。コレクションビューは、表示するセルに関する情報を、データ源から取得します。データ源やデリゲートはアプリケーション側に実装するカスタムオブジェクトで、コンテンツの管理（セルの選択/強調表示状態の管理を含む）に使います。レイアウトオブジェクトは、セルがどこに属するかを判断し、その情報をコレクションビューに、レイアウト属性オブジェクトの形で送ります。コレクションビューは、実際のセルその他のビューに関するレイアウト情報を組み合わせて、最終的な表示画面を生成します。

図 1-1　コンテンツとレイアウト情報を組み合わせて最終的な表示画面を生成する過程

コレクションビューのインターフェイスを作成する際には、まず`UICollectionView`オブジェクトをストーリーボードまたはnibファイルに追加してください。この意味でコレクションビューを、他のオブジェクトすべてを結びつける「中央ハブ」と考えてもよいでしょう。その後、データ源やデリゲートなど、他の関連オブジェクトを設定します。設定はすべて、コレクションビュー自身に集中することになります。たとえば、レイアウトオブジェクトだけあっても、コレクションビューオブジェクトがなければ役に立ちません。

ビューを再利用することにより処理性能を改善する

コレクションビューには、処理効率を改善するために、不要になったビューを回収する仕組みが組み込まれています。画面の外側に移動したビューはコレクションビューから取り除きますが、削除はせず、再利用キューに登録しておきます。スクロールの結果、新しいコンテンツが画面に現れる時点で、キューからビューを取り出して再利用するのです。この回収と再利用を効率的に行えるよう、コレクションビュー内に表示するビューはすべて、`UICollectionReusableView`クラスを継承することになっています。

コレクションビューは3種類の再利用可能なビューを管理できるようになっており、目的に応じて使い分けます。

- **セル**はコレクションビューの主たるコンテンツを表します。データ源オブジェクトから、ある単一の項目に当たるコンテンツを取得し、画面に表示する役割があります。各セルは`UICollectionViewCell`クラスのインスタンスとします。ただし、このサブクラスを定義し、コンテンツの表示に必要な処理を実装しても構いません。セルオブジェクトには、選択や強調表示の状態を管理する機能が実装されています。実際にセルを強調表示するためには、若干のコードを記述する必要があります。セルの強調表示や選択の実装について詳しくは、“選択や強調の表示状態を管理する”（22 ページ）を参照してください。
- **補助ビュー**はセクションに関する情報を表示します。セルと同様、データ駆動方式で動作します。セルと違ってこれは必須ではなく、使い方や配置はレイアウトオブジェクトが制御します。たとえばフローレイアウトの場合、ヘッダやフッタは補助ビューを使って実装するようになっています。
- **装飾ビュー**は、見た目のみに関係する要素で、レイアウトオブジェクトが所有し、データ源オブジェクトのデータとは関連づけません。たとえば独自の背景を表示するために使うことがあります。

コレクションビューは、テーブルビューと違い、データ源から取得したセルや補助ビューを、特定のスタイルで表示することはありません。基本となる、再利用可能なビューのクラスはいわば真っ白いキャンバスであって、自由に修正できるようになっています。これを使って、小規模なビュー階層を構築し、画像を表示し、さらには動的にコンテンツを描画することも可能です。

データ源オブジェクトは、コレクションビューからの要求に応じて、ビューオブジェクト（セルや補助ビュー）を供給します。しかしこれを直接生成するのではありません。コレクションビュー側に備わっている、ビューの種類に応じたメソッドを使って、キューから取得するようになっています。このメソッドは常に有効なビューを返します。再利用可能なビューを管理するキューから取り出しますが、ない場合は、クラス、nibファイル、ストーリーボードなどを参照して新たに生成します。

データ源からビューを生成、設定する方法については、“セルや補助ビューを設定する”（18 ページ）を参照してください。

レイアウトオブジェクトは視覚表現を制御する

コレクションビューに表示する項目の配置や視覚的スタイルを決めるのは、もっぱらレイアウトオブジェクトの役割です。ビューやその中身を用意するのはデータ源オブジェクトですが、大きさや位置、その他の外観属性は、レイアウトオブジェクトが制御するようになっているのです。このように役割を分担しているため、アプリケーション側が管理しているデータオブジェクトはそのままで、レイアウトを大幅に変えることも可能です。

コレクションビューのレイアウト処理は、ビュー側が行うレイアウト処理と関係しますが、完全に切り離されているのです。レイアウトオブジェクトが行うことと、親ビュー内にある子ビューの位置を制御する`layoutSubviews`メソッドの処理を、混同しないようにしてください。レイアウトオブジェクトは、管理対象であるビューを所有はしていないので、直接操作することもありません。代わりに、コレクションビューに収容されるセルや補助ビュー、装飾ビューの、位置、大きさ、外観を記述した「属性」オブジェクトを生成するようになっています。これをビューオブジェクトに適用するのはコレクションビューの役割です。

レイアウトオブジェクトは、コレクションビュー内のビューの配置を、どのようにでも決めることができます。レイアウトオブジェクトが移動できるビューと、そうでないビューがあります。もちろん、少ししか動かせないもの、画面上のどこにでも動かせるものがありえます。ビュー自身を取り囲んでいるビューとは関係ない位置に持って行くことさえ可能です。たとえば、あるビューの上に別のビューを積み重ねるような「スタック」配置も、状況によっては有用でしょう。ただし、ビューそのものの視覚スタイルはアプリケーション側が制御するので、レイアウトオブジェクトが変更することはできません。

図 1-2に、縦スクロール型のフローレイアウトにより、セルや補助ビューを配置している様子を示します。この場合、コンテンツ領域の幅は固定であり、高さが中身に応じて変化します。この領域の幅や高さを計算するため、レイアウトオブジェクトはセルやビューをひとつずつ、最も適切と思われる位置を選んで配置していきます。フローレイアウトの場合、セルや補助ビューの大きさは、レイアウトオブジェクトのプロパティとして、またはデリゲートを使って指定されます。レイアウトの計算とは単に、このプロパティを使って、コレクションビュー内に各ビューを配置する処理のことなのです。

図 1-2　　レイアウトオブジェクトが配置を決める様子

レイアウトオブジェクトが制御するのは、ビュー（セル、補助ビューなど）の大きさと位置だけではありません。透明度、3D空間における座標変換、他のビューとの前後関係（z-order）など、他の属性も指定することにより、さまざまな面白いレイアウトを実現できます。たとえば、セルを積み上げたような表示は、ビューの上に他のビューを積み重ねていき、z-orderを変更することにより可能です。あるいは、ある軸の周りの回転変換を利用して、変わったレイアウトが作れるかもしれません。

レイアウトオブジェクトがコレクションビューに対して果たす役割について詳しくは、“カスタムレイアウトの作成”（37 ページ）を参照してください。

コレクションビューは自動的にアニメーション効果を作り出す

コレクションビューにはアニメーション効果を作り出す能力が基盤機能として組み込まれています。項目やセクションを追加（または削除）する際、自動的に、対応するビューが追加/削除される様子をアニメーションで表示するのです。たとえば挿入の場合、それ以降の項目が移動して、新しい項目を入れる空間を作ります。このアニメーションを生成できるのはコレクションビューです。各項目の現在位置を検出でき、挿入後の最終的な位置も計算可能なので、現在位置から最終位置まで動く様子をアニメーション表示できるのです。

挿入、削除、移動操作に伴うアニメーション表示とは別に、アプリケーションはいつでも、現在のレイアウトを無効にし、レイアウト属性を計算し直すよう指示できます。この場合、直接的なアニメーション効果は生じません。単に、新しい位置を計算し、表示していくだけです。カスタムレイアウトではこの動作を利用して、セルを一定間隔に並べ、アニメーション効果を生み出すことができます。

データ源およびデリゲートの設計

コレクションビューにはデータ源オブジェクトが必要です。**データ源オブジェクト**とは、アプリケーションの表示内容を管理するオブジェクトのことです。アプリケーションのデータモデルに基づくオブジェクトでも、コレクションビューを管理するビューコントローラでも構いません。唯一必要なのは、項目の総数や、各項目を表示するビューなど、コレクションビューが必要とする情報を、要求に応じて返せることです。

デリゲートオブジェクトは、コンテンツの表示やその情報のやり取りを管理するオブジェクトで、必要な場合のみ用意します（が、できるだけ用意するよう推奨します）。主な役割はセルの強調表示や選択を管理することですが、これを拡張し、他の情報を渡すことも可能です。たとえばフローレイアウトの場合、基本的なデリゲートの動作を拡張して、セルの大きさ、セル間の間隔などといった寸法をカスタマイズできるようになっています。

データ源はコンテンツを管理する

データ源オブジェクトは、コレクションビューに表示しようとするコンテンツを管理します。`UICollectionViewDataSource`プロトコルに従い、所定の基本的な動作やメソッドを実装しなければなりません。データ源の役割は、コレクションビューが必要とする、次のような情報を提供することです。

- コレクションビューにいくつのセクションを置くか。
- 各セクションにはそれぞれいくつの項目を置くか。
- それぞれのセクションや項目の中身を、どのビューを使って表示するか。

セクションや**項目**は、コレクションビューにコンテンツを表示するための基本的な単位です。セクションは少なくとも1つ必要で、複数あっても構いません。一方、各セクションには0個以上の項目があります。項目は表示しようとする主たるコンテンツを表し、セクションには項目を論理的なグループに分ける働きがあります。たとえば写真アプリケーションの場合、1冊のアルバムを表すため、あるいは同じ日に撮影した写真をまとめるためにセクションを使います。

コレクションビューは収容する各データを、`NSIndexPath`オブジェクトを使って参照します。ある項目を特定したい場合、レイアウトオブジェクトが当該項目に割り当てた、インデックスパス情報を使います。項目の場合、これはセクション番号と項目番号から成ります。補助ビューや装飾ビューであれば、レイアウトオブジェクトが割り当てた値から成ります。補助ビューや装飾ビューに付与するイ

ンデックスパスの意味はアプリケーション次第です。もっとも、その第1インデックスは、データ源のあるセクションに対応します。この場合のインデックスパスには、何らかの意味を持たせるというよりも、識別の働きが強いと考える方がよいでしょう。着目しているのがどの種類のビューか、特定する役割です。したがって、たとえば（フローレイアウトに見られるような）あるセクションのヘッダ/フッタを生成する補助ビューの場合、インデックスパスは、参照しているセクションを示す、という役割を担います。

> **注意:** 標準のインデックスパスは多段の階層づけも可能ですが、コレクションビューのセルには、「セクション」および「項目」という2階層のパラメータから成るインデックスパスしか付与できません（`UITableView`クラスのインデックスパスと同様）。補助ビューや装飾ビューには、必要に応じて、より複雑なインデックスパスを設定することも可能です。パスの第2インデックス以降の要素は、第1インデックスで指定されるセクションに対応すると考えます。従来は第2インデックスのみ必要とされてきましたが、補助ビューや装飾ビューの場合、この制約にとらわれる必要はありません。データ源を設計する際には、これを頭に入れておくとよいでしょう。

データオブジェクトのセクションや項目をどのように配置するにしても、その見かけ上の表現は、レイアウトオブジェクトが決めるようになっています。レイアウトオブジェクトが違えば、セクションや項目の表示方法は大きく異なります（図 2-1を参照）。フローレイアウトオブジェクト（左）は、セクションを番号順に縦に並べて表します。カスタムレイアウト（右）の場合、セクションは一直線上に並んでいません。このようなことができるのも、レイアウトと実データとを分けて管理している効果です。

図 2-1 レイアウトオブジェクトの配置に従ってセクションを配置した様子

データオブジェクトを設計する

データ源は、セクションおよび項目という概念を使って、データオブジェクトを編成するとよいでしょう。こうしておけば、データ源に関するメソッドの実装が容易になります。また、このメソッドは頻繁に呼び出されるので、できるだけ迅速に必要なデータを渡せるようにしなければなりません。

入れ子になった配列を使う実装方法がひとつ考えられます（図 2-2を参照）。もちろんこれが唯一の方法ではありません。この編成の場合、最上位の配列は、セクションを表す配列から成ります。セクション配列には、当該セクションに属するデータ項目が並びます。あるセクションに属するある項目

を検索するためには、該当するセクション配列を見つけ、その中から項目を探せばよいことになります。この方法は、中程度の項目数から成るコレクションを管理し、要求に応じて個々の項目を検索する、という用途に向いています。

図 2-2　入れ子になった配列を使ってデータオブジェクトを管理する様子

データ構造を設計する際には、配列を組み合わせた単純な構造をまず検討し、必要であればより効率的な構成を考えるとよいでしょう。一般に、このデータオブジェクトが処理性能を律する隘路となってはなりません。コレクションビューがデータ源にアクセスするのは、通常、オブジェクトの総数を計算するときと、現時点で画面に表示するべき要素のビューを取得するときに限ります。レイアウトオブジェクトの動作が、データオブジェクトから取得するデータのみに依存している場合、データ源に数千ものオブジェクトがあれば、処理性能を大きく損なう虞があります。

コンテンツに関する情報をコレクションビューに通知する

コレクションビューはデータ源に対して、セクション数や、各セクションの項目数を、次のようなときに問い合わせます。

- コレクションビューを最初に表示するとき。
- 他のデータ源オブジェクトをコレクションビューに割り当てたとき。
- コレクションビューの`reloadData`メソッドを明示的に呼び出したとき。
- コレクションビューのデリゲートは、`performBatchUpdates:completion:`でブロックを実行し、あるいは移動/挿入/削除メソッドを実行します。

セクション数は`numberOfSectionsInCollectionView:`メソッド、各セクションの項目数は`collectionView:numberOfItemsInSection:`メソッドで返します。`collectionView:numberOfItemsInSection:`メソッドは必ず実装しなければなりませんが、セクションが1つだけであれば、`numberOfSectionsInCollectionView:`メソッドは実装しなくても構いません。いずれも整数型の値を返します。

データ源を図 2-2 （17 ページ）のように実装した場合、データ源メソッドの実装は比較的単純です（リスト 2-1）。ここで変数`_data`は、データ源が管理するメンバー変数で、最上位のセクション配列を格納しています。この配列の要素数が、そのままセクション数になります。また、その要素は再び配列になっており、要素数を数えればそれがあるセクションに属する項目数になります（実際には、無効な値を返すことのないよう、エラーチェックその他の処理を組み込まなければなりません）。

リスト 2-1　セクション数および項目数を返すメソッドの実装例

```
- (NSInteger)numberOfSectionsInCollectionView:(UICollectionView*)collectionView {
    // _data is a class member variable that contains one array per section.
```

```
    return [_data count];
}


- (NSInteger)collectionView:(UICollectionView*)collectionView
numberOfItemsInSection:(NSInteger)section {
    NSArray* sectionArray = [_data objectAtIndex:section];
    return [sectionArray count];
}
```

セルや補助ビューを設定する

データ源にはほかに、コレクションビューがコンテンツの表示に用いる、ビューを供給する処理を実装する必要があります。コレクションビューはコンテンツに関与しません。ビューを取得し、これにレイアウト情報を適用するだけです。したがって、ビューの表示内容を管理するのはアプリケーション側の役割です。

コレクションビューは、データ源からセクション数や項目数を取得した後、レイアウトオブジェクトに、コレクションビューのコンテンツ表示に必要なレイアウト属性を渡すよう要求します。その過程でコレクションビューは、レイアウトオブジェクトに対し、ある矩形領域（通常は可視領域を表す矩形）内に配置される要素のリストを要求します。これをもとに、データ源から、該当するセルや補助ビューを取得するのです。データ源が要求されたセルや補助ビューを返せるよう、次のような処理を実装しなければなりません。

1. セルや補助ビューのテンプレートをストーリーボードファイルに埋め込む（または、セルやビューの種類に応じ、クラスまたはnibファイルを登録する）。
2. 適当なセルやビューをキューから取り出し、必要な設定を施す（データ源に実装）。

セルや補助ビューを効率よく使用できるよう、コレクションビュー側に、当該オブジェクトの生成を支援する仕組みが用意されています。各コレクションビューは、不要になったセルや補助ビューを、内部キューに入れて管理しています。したがって、アプリケーション側で生成する代わりに、コレクションビューに対して必要なセルや補助ビューを渡すよう要求すればよいのです。未使用のセルや補助ビューがキューにあれば、それを取り出して即座に返します。ない場合は、登録済みクラスまたはnibファイルを使い、生成して返すようになっています。したがって、セルや補助ビューをキューから取り出せば、すぐに使えるオブジェクトが必ず得られます。

再利用識別子を使えば、いくつもの型のセルや補助ビューを区別して登録できます。**再利用識別子**は単なる文字列で、登録したセルや補助ビューの型を識別するために使います。文字列の中身に関与するのはデータ源オブジェクトだけです。しかし、セルやビューを渡すよう要求されたときには、指定されたインデックスパスに基づき、どの型のセルやビューが必要かを判断して、適切な再利用識別子を、キューから取り出すメソッドに渡さなければなりません。

セルや補助ビューを登録する

コレクションビューのセルや補助ビューは、プログラムで、またはストーリーボードファイルで設定します。

セルや補助ビューの設定をストーリーボードで行う。ストーリーボード上で、該当する項目をコレクションビューにドラッグし、そこで必要な設定を施します。その結果、コレクションビューと、対応するセルや補助ビューとの間に、関係が生成されます。

- セルの場合、「Collection View Cell」をオブジェクトライブラリからドラッグし、コレクションビュー上にドロップしてください。セルのカスタムクラスおよび再利用識別子を、適当な値に設定します。
- 補助ビューの場合、「Collection Reusable View」をオブジェクトライブラリからドラッグし、コレクションビュー上にドロップしてください。補助ビューのカスタムクラスおよび再利用識別子を、適当な値に設定します。

セルの設定をプログラムで行う。`registerClass:forCellWithReuseIdentifier:`メソッドまたは`registerNib:forCellWithReuseIdentifier:`メソッドで、セルに再利用識別子を対応づけます。親であるビューコントローラの初期設定の際に、このメソッドを呼び出すようにしてもよいでしょう。

補助ビューの設定をプログラムで行う。
`registerClass:forSupplementaryViewOfKind:withReuseIdentifier:`メソッドまたは`registerNib:forSupplementaryViewOfKind:withReuseIdentifier:`メソッドで、それぞれの種類のビューと再利用識別子を対応づけます。親であるビューコントローラの初期設定の際に、このメソッドを呼び出すようにしてもよいでしょう。

セルの登録には再利用識別子のみ使うのに対し、補助ビューの場合は**種類文字列**という識別子も指定する必要があります。各レイアウトオブジェクトは、扱う補助ビューの*種類*を定義しなければなりません。たとえば`UICollectionViewFlowLayout`クラスには、補助ビューの種類として、セクションヘッダとセクションフッタの2つがあります。2種類のビューを区別するため、文字列定数`UICollectionElementKindSectionHeader`および`UICollectionElementKindSectionFooter`が定義されています。レイアウトオブジェクトはその処理に際して、補助ビューのレイアウト属性のひと

つとして種類文字列も考慮しながら配置を決めていきます。コレクションビューは、データ源からビューを取得する際に、種類文字列を指定します。データ源は、種類文字列と再利用識別子をもとに、どのビューオブジェクトをキューから取り出して返すか判断します。

> **注意:** カスタムレイアウトを実装する場合、扱う補助ビューの種類を定義しなければなりません。補助ビューはいくつでも扱えますが、それぞれに種類文字列を指定することになります。カスタムレイアウトの定義については、“カスタムレイアウトの作成”（37 ページ）を参照してください。

登録は、セルや補助ビューをキューから取得する処理の前に、1度だけ実行します。いったん登録してしまえば、必要に応じていくつでも取得できるようになります。キューから取得した後で登録情報を変更することはお勧めできません。いったん登録したら、最後までそのままで通してください。

セルや補助ビューをキューから取り出して設定する

データ源オブジェクトは、コレクションビューからの要求に応じて、セルや補助ビューを返さなければなりません。そのため`UICollectionViewDataSource`プロトコルには、`collectionView:cellForItemAtIndexPath:`および`collectionView:viewForSupplementaryElementOfKind:atIndexPath:`という2つのメソッドが定義されています。コレクションビューにはセルが必須なので、`collectionView:cellForItemAtIndexPath:`メソッドの実装も必須ですが、`collectionView:viewForSupplementaryElementOfKind:atIndexPath:`メソッドが必要かどうかは、レイアウトの種類によります。いずれにしても、次の簡単なパターンに従って実装してください。

1. 適切な型のセルまたは補助ビューを、`dequeueReusableCellWithReuseIdentifier:forIndexPath:`メソッドまたは`dequeueReusableSupplementaryViewOfKind:withReuseIdentifier:forIndexPath:`メソッドで、キューから取り出す。
2. 指定されたインデックスパスのデータを使ってビューを設定する。
3. このビューを返す。

キューからセルや補助ビューを取り出す処理は、（キューから取り出せなくても）アプリケーション側で生成しなくてもよいように設計されています。あらかじめセルや補助ビューをあらかじめ登録しておけば、キューから取り出すメソッドが`nil`を返すことはありません。不要になって再利用キューに入っているセルや補助ビューがなければ、ストーリーボード、または登録済みのクラスやnibファイルを使って、生成するようになっているからです。

キューからセルや補助ビューを取り出して返すメソッドは、そのまま新しいデータを設定できるよう、初期状態にして返すはずです。新規に生成する場合、通常の手順で、すなわち、ストーリーボードやnibファイルからビューをロードし、または新規にインスタンスを生成した後、`initWithFrame:`メソッドで初期化するようになっています。これに対し、再利用キューから取り出した場合は、前回のデータが設定されたままになっています。そこで、当該セルや補助ビューの`prepareForReuse`メソッドを呼び出して、初期状態に戻してから返すようになっています。独自にセルや補助ビューのクラスを実装する場合は、このメソッドをオーバーライドし、プロパティをデフォルト値に戻すなど、初期状態にするための処理を組み込むとよいでしょう。

データ源は、ビューをキューから取り出した後、新しいデータに基づいてビューの設定を行います。データ源のメソッドに渡されたインデックスパスをもとに、該当するデータオブジェクトを見つけ、そのデータを設定することになります。このセルや補助ビューを返せば、データ源側の処理は終了です。リスト 2-2に、セルを設定する簡単なコード例を示します。セルをキューから取り出した後、セルの位置に関する情報をもとにラベルを設定し、このセルを返しています。

リスト 2-2　カスタムセルを設定するコード例

```
- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
                  cellForItemAtIndexPath:(NSIndexPath *)indexPath {
   MyCustomCell* newCell = [self.collectionView
dequeueReusableCellWithReuseIdentifier:MyCellID


forIndexPath:indexPath];


   newCell.cellLabel.text = [NSString stringWithFormat:@"Section:%d, Item:%d",
indexPath.section, indexPath.item];
   return newCell;
}
```

> **注意:** データ源から返すビューは、有効なものでなければなりません。何らかの理由で、ビューが実際に表示されることはないとしても、`nil`を返すと表明（アサート）違反になり、アプリケーションは停止してしまいます。レイアウトオブジェクトは、このメソッドが有効なビューを返すものと想定しているからです。

セクションや項目を挿入、削除、移動する

セクションや項目をひとつ、追加、削除、移動する手順は次の通りです。

1. データ源オブジェクトのデータを更新する。

2. コレクションビューの適切なメソッドを呼び出して、セクションや項目を挿入または削除する。

重要なのは、まずデータ源を更新し、その後でコレクションビューに変更を通知することです。コレクションビュー側のメソッドは、データ源が最新のデータを保持していると想定しているからです。この想定が正しくなければ、データ源から誤った項目を受け取り、あるいは存在しない項目を渡すよう要求する結果、アプリケーションがクラッシュする虞があります。

単一の項目をプログラム上で追加、削除、移動すると、コレクションビュー側のメソッドは自動的に、この操作に応じたアニメーションを表示します。しかし、複数の項目を操作（挿入、削除、移動）する場合、アニメーション表示はまとめて一度で行う必要があるでしょう。これは、操作をすべてブロック内に記述し、このブロックを`performBatchUpdates:completion:`メソッドに渡すことにより実現します。このバッチ更新処理では、複数の操作をまとめてひとつと看做してアニメーション表示を行うほか、同じブロック内で操作の順序を入れ替えるなど工夫することも可能です。

リスト 2-3に、選択状態の項目をまとめて削除する、バッチ更新のコード例を示します。`performBatchUpdates:completion:`メソッドに渡されるブロックでは、まず、データ源を更新するカスタムメソッドを実行します。次に、該当する項目群を削除するよう、コレクションビューに通知します。このメソッドに渡した更新ブロックと完了ブロックは、同期して実行されます。

リスト 2-3　選択状態の項目群を削除するコード例

```
[self.collectionView performBatchUpdates:^{
    NSArray* itemPaths = [self.collectionView indexPathsForSelectedItems];

    // Delete the items from the data source.
    [self deleteItemsFromDataSourceAtIndexPaths:itemPaths];

    // Now delete the items from the collection view.
    [self.collectionView deleteItemsAtIndexPaths:tempArray];
} completion:nil];
```

選択や強調の表示状態を管理する

コレクションビューには単一項目を選択する処理が初めから組み込まれているほか、複数の項目を選択する、あるいはまとめて選択を解除することができるよう設定することも可能です。コレクションビューは、境界内におけるタップ操作を検出し、対応するセルを強調表示、あるいは選択状態にします。多くの場合、セルが選択状態や強調表示になっていても、コレクションビューはセルのプロパ

ティを変更するだけで、外観を変えることはありません。ただし例外として、セルの`selectedBackgroundView`プロパティに有効なビューが設定されていれば、セルが強調表示または選択状態になったとき、このビューを表示するようになっています。

リスト 2-4のようなコードを、コレクションビューのセルを実装するコードに組み込むことにより、強調表示や選択状態を表す外観を変更できます。セルを初めてロードした時点や、強調表示にも選択状態にもなっていない状態では、`backgroundView`プロパティのビューが表示されます。強調表示や選択状態になると、これが`selectedBackgroundView`プロパティのビューに置き換わります。このコード例の場合、強調表示や選択状態になると、セルの背景色が赤から白に変わります。

リスト 2-4　状態の変化を表す背景ビューを設定するコード例

```
UIView* backgroundView = [[UIView alloc] initWithFrame:self.bounds];
backgroundView.backgroundColor = [UIColor redColor];
self.backgroundView = backgroundView;

UIView* selectedBGView = [[UIView alloc] initWithFrame:self.bounds];
selectedBGView.backgroundColor = [UIColor whiteColor];
self.selectedBackgroundView = selectedBGView;
```

コレクションビューのデリゲートには、コレクションビューの強調/選択表示を行う、次のようなメソッドがあります。

- `collectionView:shouldSelectItemAtIndexPath:`
- `collectionView:shouldDeselectItemAtIndexPath:`
- `collectionView:didSelectItemAtIndexPath:`
- `collectionView:didDeselectItemAtIndexPath:`
- `collectionView:shouldHighlightItemAtIndexPath:`
- `collectionView:didHighlightItemAtIndexPath:`
- `collectionView:didUnhighlightItemAtIndexPath:`

以上のメソッドを使って、コレクションビューを強調/選択したときの動作をさまざまに調整できます。

たとえば、セルの選択状態をアプリケーション側で描画したい場合は、`selectedBackgroundView`プロパティを`nil`にし、デリゲートオブジェクトに、外観を変更する処理を組み込んでください。`collectionView:didSelectItemAtIndexPath:`メソッドに外観を変更する処理、`collectionView:didDeselectItemAtIndexPath:`メソッドに元の状態に戻す処理を実装します。

同様に、強調表示状態をアプリケーション側で描画したい場合は、デリゲートの`collectionView:didHighlightItemAtIndexPath:`メソッドおよび`collectionView:didUnhighlightItemAtIndexPath:`メソッドをオーバーライドして実装してください。`selectedBackgroundView`プロパティにビューを指定している場合は、セルのコンテンツビューにも変更を及ぼして、変化が外観に現れるようにしなければなりません。リスト 2-5に、コンテンツビューの背景色を変えて「強調表示」状態を表示する、簡単なコード例を示します。

リスト 2-5　セルを一時的に強調表示するコード例

```
- (void)collectionView:(UICollectionView *)colView
didHighlightItemAtIndexPath:(NSIndexPath *)indexPath {

    UICollectionViewCell* cell = [colView cellForItemAtIndexPath:indexPath];

    cell.contentView.backgroundColor = [UIColor blueColor];

}


- (void)collectionView:(UICollectionView *)colView
didUnhighlightItemAtIndexPath:(NSIndexPath *)indexPath {

    UICollectionViewCell* cell = [colView cellForItemAtIndexPath:indexPath];

    cell.contentView.backgroundColor = nil;

}
```

強調表示状態と選択状態には、微妙な、しかし重要な違いがあります。**「強調表示」**はユーザの指がデバイスに触れている間の一時的な状態です。これが`YES`になるのは、コレクションビューがセル上のタッチイベントを追跡している間だけです。タッチイベントが止まれば`NO`に戻ります。これに対し**「選択」状態**は、一連のタッチイベントが終了した後、すなわち、ユーザがこのセルを選択しようとしていることを表すタッチイベントが終了した後で変化えいます。

図 2-3に、未選択状態のセルにユーザがタッチしたときに発生する、一連の動作を示します。最初の「タッチダウン」イベントを受けて、コレクションビューはセルの「強調表示」状態を`YES`にします。もっとも、自動的にセルの外観に反映されるわけではありません。その後、「タッチアップ」イベントが発生すると、「強調表示」状態は`NO`に戻り、「選択」状態は`YES`に変わります。ユーザが選択状

態を変更すると、コレクションビューはセルのselectedBackgroundViewプロパティに設定されているビューを表示します。コレクションビューが関与してセルの外観を変えるのは、この場合に限ります。ほかの視覚効果はデリゲートオブジェクトが行わなければなりません。

図 2-3 セルに対するタッチ操作を追跡する様子

ユーザがセルを選択する場合も解除する場合も、セルの「選択」状態が変化するのは、一連の操作が終わってからです。セル内をタップすると、まず「強調表示」状態が変化します。一連のタップ操作が終わり、「強調表示」状態が元に戻った後で、「選択」状態が変化します。セルを設計する際には、「強調表示」状態と「選択」状態の外観が、意図せぬ競合を引き起こさないよう注意してください。

セルの編集メニューを表示する

ユーザがセル上で「ロングタップ」ジェスチャをすると、コレクションビューはそのセルの「Edit」メニューを表示しようとします。このメニューには、セルを切り取り、コピーし、貼り付けるコマンドがあります。ただし、実際に表示されるのは、以下の条件を満たす場合に限ります。

- デリゲートに、アクションを処理する次の3つのメソッドが実装されていること。

  `collectionView:shouldShowMenuForItemAtIndexPath:`

  `collectionView:canPerformAction:forItemAtIndexPath:withSender:`

  `collectionView:performAction:forItemAtIndexPath:withSender:`

- `collectionView:shouldShowMenuForItemAtIndexPath:`メソッドが、指定されたセルに対して`YES`を返すこと。

- `collectionView:canPerformAction:forItemAtIndexPath:withSender:`メソッドが、次のうち少なくとも1つのアクションに対して`YES`を返すこと。コレクションビューが実行できるアクションは次の通りです。

  `cut:`

  `copy:`

  `paste:`

以上の条件が満たされ、ユーザがメニューからアクションを選択すると、コレクションビューはデリゲートの`collectionView:performAction:forItemAtIndexPath:withSender:`メソッドを呼び出して、該当するアクションを実行しようとします。

リスト 2-6に、いずれかのメニュー項目が現れないようにするコード例を示します。この`collectionView:canPerformAction:forItemAtIndexPath:withSender:`メソッドは、「Edit」メニューの「Cut」が現れないようにしています。「Copy」と「Paste」はあるので、コンテンツの挿入はできることになります。

リスト 2-6　「Edit」メニューのあるアクションを無効にするコード例

```
- (BOOL)collectionView:(UICollectionView *)collectionView
         canPerformAction:(SEL)action
         forItemAtIndexPath:(NSIndexPath *)indexPath
         withSender:(id)sender {
    // Support only copying and pasting of cells.
    if ([NSStringFromSelector(action) isEqualToString:@"copy:"]
       || [NSStringFromSelector(action) isEqualToString:@"paste:"])
       return YES;

    // Prevent all other actions.
    return NO;
}
```

ペーストボード関係のコマンドについて詳しくは、『*Text Programming Guide for iOS*』を参照してください。

異なるレイアウトへの遷移

異なるレイアウトに遷移するためには、`setCollectionViewLayout:animated:`メソッドを使うのが簡単です。しかし、遷移方法を制御したい、あるいは対話的に行いたい場合は、`UICollectionViewTransitionLayout`オブジェクトを使います。

`UICollectionViewTransitionLayout`クラスは特殊なレイアウトで、新しいレイアウトに遷移する際、コレクションビューのレイアウトオブジェクトとしてインストールされます。遷移レイアウトオブジェクトを使えば、非線形のパスに沿って遷移する、通常とは異なるタイミングアルゴリズムを適用する、タッチイベントに応じて動かす、などの制御が可能です。`UICollectionViewTransitionLayout`クラスをそのまま使っても線形のパスに沿って遷移することは可能ですが、`UICollectionViewLayout`クラスと同様に、サブクラスを定義すればさまざまな効果を実装できます。このサブクラスに、カスタムレイアウトを生成する場合と同じメソッドを実装し、

ユーザ入力（多くの場合ジェスチャリコグナイザ）に合わせて動作するようにしてください。カスタムレイアウトオブジェクトの生成について詳しくは、“カスタムレイアウトの作成” （37 ページ）を参照してください。

`UICollectionViewLayout`クラスには、レイアウト間の遷移を追跡するためのメソッドがいくつかあります。`UICollectionViewTransitionLayout`オブジェクトは遷移の進行状況を、`transitionProgress`プロパティを通して認識します。遷移中は定期的に、遷移終了までの比率を表す値でこのプロパティを更新するように実装してください。`UICollectionViewTransitionLayout`クラスを、たとえばジェスチャリコグナイザのような、レイアウト間の遷移に使えるオブジェクトと連携させることにより、対話的な遷移を実現することも可能です。同様に、カスタム遷移レイアウトオブジェクトを実装すれば、`UICollectionViewTransitionLayout`の`updateValue:forAnimatedKey:`メソッドおよび`valueForAnimatedKey:`メソッドを使って、レイアウトに関与する値を追跡できます。このメソッドで追跡するのは特別な浮動小数点数値であり、遷移中に随時、この値を更新することにより、レイアウト上重要な情報を伝えることができます。たとえばピンチジェスチャでレイアウト間の遷移を行う場合、この仕組みを利用して、遷移の状態（オフセット）を遷移レイアウトオブジェクトに伝える、という使い方が考えられます。

`UICollectionViewTransitionLayout`オブジェクトをアプリケーションに組み込む手順を以下に示します。

1. 標準クラス、あるいは独自に実装したカスタムクラスのインスタンスを、`initWithCurrentLayout:nextLayout:`メソッドで生成します。
2. 遷移の進行状況は、定期的に`transitionProgress`プロパティを更新することにより伝えます。更新の都度、忘れずにレイアウトを「無効化」してください（コレクションビューの`invalidateLayout`メソッド）。
3. コレクションビューのデリゲートに、遷移レイアウトオブジェクトを返す、`collectionView:transitionLayoutForOldLayout:newLayout:`メソッドを実装します。
4. 必要に応じ、レイアウトに関与する値を`updateValue:forAnimatedKey:`メソッドで更新することにより、レイアウトオブジェクトにその旨を通知します。この場合、値を0とすれば安定状態であることになります。

フローレイアウトの使い方

コレクションビューの各項目を、具象レイアウトオブジェクトである`UICollectionViewFlowLayout`クラスを使って配置できます。行指向のレイアウトで、セルを一直線（行）に沿ってできるだけ多く並べます。現在の行に空きがなくなったら、新しい行を作って続行します。図 3-1に、縦スクロール型のフローレイアウトの外観を示します。行を水平に配置し、新しい行が必要になったら、現在行のすぐ下に作るようになっています。同じセクションに属するセル群を、セクションヘッダとセクションフッタで囲むことも可能です。

図 3-1　フローレイアウトによりセクションとセルを配置した様子

フローレイアウトは項目を格子状に配列するために使えますが、それだけではありません。行指向のレイアウトは、ほかにもさまざまな配置に適用できます。たとえば、格子状に並べるのではなく、スクロールと同じ方向に（一列に）項目を並べ、その間隔を調整する、という配置が考えられます。各項目の大きさは違っていてもよく、その場合、格子とは言い難い対称性の崩れた配置になりますが、それでも行指向のフローの一種です。このように、可能性は多岐にわたります。

フローレイアウトの設定は、プログラムで記述しても、XcodeのInterface Builderを使っても構いません。具体的な手順は次の通りです。

1. 「フローレイアウト」オブジェクトを生成し、コレクションビューに対応づける。
2. セルの幅と高さを設定する。
3. （必要に応じ）行間および項目間の間隔を設定する。
4. セクションヘッダやセクションフッタが必要ならば、その大きさを指定する。
5. スクロール方向を設定する。

Important: 少なくともセルの幅と高さは指定する必要があります。そうでないと、幅や高さが0であるとして配置されるため、画面には何も現れません。

フローレイアウトの属性をカスタマイズする

「フローレイアウト」オブジェクトには、外観を制御するプロパティがいくつかあります。この設定は、配置する項目すべてに、一斉に適用されます。たとえばセルの大きさを`itemSize`プロパティで設定すると、セルはすべて同じ大きさになります。

大きさや間隔を個別に変えたい場合は、`UICollectionViewDelegateFlowLayout`プロトコルに定義されたメソッドを使います。このメソッドは、コレクションビューに対応づけた、同じデリゲートオブジェクトに実装してください。所定のメソッドが存在すれば、指定された固定値を使う代わりに、そのメソッドを呼び出して大きさや間隔を取得するようになります。メソッドは、コレクションビューに収容する項目すべてについて、適切な値を返すように実装しなければなりません。

フローレイアウトの各項目の大きさを指定する

コレクションビューに収容される項目の大きさがすべて同じであれば、その幅と高さを、「フローレイアウト」オブジェクトの`itemSize`プロパティに設定してください（ポイント単位）。大きさが項目によって変わらない場合、これが最も簡単な方法です。

セルごとに異なる大きさを指定したい場合は、コレクションビューのデリゲートに、`collectionView:layout:sizeForItemAtIndexPath:`メソッドを実装する必要があります。引数として渡されるインデックスパス情報をもとに、対応する項目の大きさを返します。「フローレイアウト」オブジェクトは、同じ行に並ぶ項目を、高さ方向には中央揃えにして配置します（図 3-2を参照）。行全体の高さや幅は、その方向における最大の項目によって決まります。

図 3-2 大きさが異なる項目をフローレイアウトで配置した様子

注意: セルの大きさが違えば、1行当たりの項目数も行によって異なる可能性があります。

項目間および行間の間隔を指定する

フローレイアウトでは、同じ行に並ぶ項目間、あるいは隣り合う行間の、最小間隔を指定できます。ここで与える間隔は、あくまで「最小」値であることに注意してください。配置方法の性質上、項目間の実際の間隔は、指定より大きくなることもありえます。行間の間隔も同様に、さまざまな大きさの項目を配置すれば、指定より大きくなるかもしれません。

フローレイアウトオブジェクトは、現在行に項目を追加していく処理を、新しい項目を完全に収容できるだけの余裕がなくなるまで続けます。その結果、幅が余ることなくぴったりと配置し終えた場合、項目間の間隔は、指定された最小間隔と等しくなります。行の末尾に余りができた場合、項目間の間隔を均等に拡げて、最後の項目がちょうど行の末尾に来るよう調整します（図 3-3を参照）。これにより全体の見た目を改善し、各行の末尾に大きな隙間ができるのを防ぐことができます。

図 3-3　項目間の間隔が指定された最小間隔よりも大きくなっている様子

行間の間隔についても、項目間の間隔と同様の手法で調整します。項目の大きさがすべて同じであれば、最小行間隔の指定をそのまま適用するので、次の行にある項目との間隔はすべて均等になります。項目ごとに大きさが異なれば、実際の間隔も変わる可能性があります。

図 3-4に、項目ごとに大きさが異なる場合、行間の間隔がどのようになるかを示します。項目によって大きさが異なる場合、「フローレイアウト」オブジェクトは、各行の中で、スクロール方向の寸法が最大の項目を選びます。たとえば縦スクロール型の場合、各行から高さが最大の項目を選ぶことになります。そして、当該項目間の間隔が指定された最小値になるよう配置するのです。図のように、選ばれた項目が同じ並びになければ、実際の行間間隔は、指定された最小値よりも大きくなります。

図 3-4　項目の大きさが異なるために行間の間隔が変わっている様子

間隔についても、フローレイアウトの他の属性と同様、固定値を指定するほか、動的に異なる値を与えることも可能です。行間や項目間の間隔は、セクションごとに指定できます。同じセクション内では同じ間隔ですが、別のセクションでは異なる間隔であっても構わないのです。すべて同じ間隔を指定する場合は`minimumLineSpacing`および`minimumInteritemSpacing`プロパティを使い、そうでない場合はコレクションビューのデリゲートに`collectionView:layout:minimumLineSpacingForSectionAtIndex:`および`collectionView:layout:minimumInteritemSpacingForSectionAtIndex:`メソッドを実装してください。

セクション差し込みを使って余白を調整する

セクション差し込みは、セルの配置領域を調整するための手段となります。セクションのヘッダビューの後、フッタビューの前に、余白を設けるために使います。また、両側にも余白を与えることができます。図 3-5に、縦スクロール型のフローレイアウトで、差し込みがどのように配置に影響するかを示します。

図 3-5　セクション差し込みを使ってセルの配置領域を調整している様子

差し込みを置くとセルの配置領域が狭くなるので、行に配置するセルの個数を制限するためも使えます。スクロール方向以外の差し込みを指定すると、各行の長さを制限できるのです。この情報とセルの大きさ情報を組み合わせると、各行に配置するセルの個数を制御できます。

フローレイアウトのサブクラスを定義するべき状況

「フローレイアウト」オブジェクトはそのままでも有用ですが、サブクラスを定義し、必要な動作を実装するべき状況もいくつかあります。表 3-1に、所定の効果を得るために、`UICollectionViewFlowLayout`のサブクラスを定義するべき状況を示します。

表 3-1 UICollectionViewFlowLayoutのサブクラスを定義するべき状況

シナリオ	サブクラス化のヒント
補助ビューや装飾ビューを新たに追加したい	標準の「フローレイアウト」クラスは、セクションヘッダやセクションフッタを補助ビューとして追加できるだけであり、装飾ビューにも対応していません。補助ビューや装飾ビューを追加したい場合、少なくとも次のメソッドをオーバーライドする必要があります。 • layoutAttributesForElementsInRect:（必須） • layoutAttributesForItemAtIndexPath:（必須） • layoutAttributesForSupplementaryViewOfKind: atIndexPath:（補助ビューの場合） • layoutAttributesForDecorationViewOfKind: atIndexPath:（装飾ビューの場合） layoutAttributesForElementsInRect:メソッドでは、superを呼び出してセルのレイアウト属性を取得し、指定された矩形内に置く、新しい補助ビューや装飾ビューの属性を追加します。他のメソッドには、必要に応じて属性を指定する働きがあります。 レイアウト処理中にビューの属性を与える方法については、“レイアウト属性を生成する”（39 ページ）および“指定された矩形内に表示する項目のレイアウト属性を与える”（41 ページ）を参照してください。
「フローレイアウト」オブジェクトが返すレイアウト属性を調整したい	layoutAttributesForElementsInRect:その他、レイアウト属性を返すメソッドをオーバーライドします。superを呼び出して親クラスから属性を取得し、修正したものを返してください。 上記のメソッドについて詳しくは、“レイアウト属性を生成する”（39 ページ）および“指定された矩形内に表示する項目のレイアウト属性を与える”（41 ページ）を参照してください。
セルやビューのレイアウト属性を追加したい	UICollectionViewLayoutAttributesのカスタムサブクラスを定義し、独自のレイアウト情報を表すプロパティを追加します。 UICollectionViewFlowLayoutのサブクラスを定義し、layoutAttributesClassメソッドをオーバーライドして、先に定義したカスタムサブクラスを返すように実装します。 さらに、layoutAttributesForElementsInRect:、layoutAttributesForItemAtIndexPath:その他、レイアウト属性を返すメソッドをオーバーライドし、追加した独自属性の値を設定して返すように実装します。

シナリオ	サブクラス化のヒント
項目の挿入/削除時に表示されるアニメーションの開始/終了位置を指定したい	標準の状態では、項目を挿入または削除する際、簡単なフェード効果のアニメーションが表示されます。これをカスタマイズするためには、対象に応じて次のメソッドをオーバーライドしなければなりません。 • `initialLayoutAttributesForAppearingItemAtIndexPath:` • `initialLayoutAttributesForAppearingSupplementaryElementOfKind:atIndexPath:` • `initialLayoutAttributesForAppearingDecorationElementOfKind:atIndexPath:` • `finalLayoutAttributesForDisappearingItemAtIndexPath:` • `finalLayoutAttributesForDisappearingSupplementaryElementOfKind:atIndexPath:` • `finalLayoutAttributesForDisappearingDecorationElementOfKind:atIndexPath:` 上記のメソッドを、挿入前または削除後における、各ビューの属性を指定するように実装します。「フローレイアウト」オブジェクトは、この属性に基づき、挿入や削除のアニメーション表示を行います。 上記のメソッドをオーバーライドする場合、`prepareForCollectionViewUpdates:`および`finalizeCollectionViewUpdates`もオーバーライドするとよいでしょう。このメソッドでは、現在のサイクルで挿入または削除される項目を追跡できます。 挿入や削除の処理について詳しくは、“挿入/削除時により面白いアニメーションを表示する”（45 ページ）を参照してください。

カスタムレイアウトを白紙の状態から構築するべき状況もあります。しかしその場合、本当に必要かどうか、時間をかけて検討してください。フローレイアウトは、工夫次第でさまざまなレイアウトに対応でき、使いやすく、各種の最適化も施されているので効率よく動作します。もちろん、カスタムレイアウトを生成するなと言っているのではありません。状況に応じて的確に判断するべきです。フローレイアウトはスクロールが一方向に限定されているので、縦横どちらの方向についても画面内に収まらないような場合、カスタムレイアウトを実装する方がよいでしょう。あるいは、格子その他、上述のような行指向のレイアウトに当てはまらない場合も、カスタムレイアウトを採用するべきです。配置する項目が頻繁に移動する場合も、フローレイアウトのサブクラスを定義して実装しようとすると、無駄に複雑になってしまうかも知れません。

カスタムレイアウトの生成について詳しくは、“カスタムレイアウトの作成”（37 ページ）を参照してください。

ジェスチャ認識機能の組み込み

コレクションビューにジェスチャリコグナイザを組み込めば、対話性を大幅に高めることができます。所定のジェスチャを認識したとき、あるアクションを起動するよう設定してください。コレクションビューに実装することが多いアクションとしては、次の2種類があります。

- コレクションビューのレイアウト情報を変更するアクション。
- セルやビューを直接操作するアクション。

ジェスチャリコグナイザは、個々のセルやビューではなく、コレクションビュー自体に組み込まなければなりません。`UICollectionView`は`UIScrollView`の子孫クラスなので、ジェスチャリコグナイザを組み込んでも、追跡するべき他のジェスチャと干渉する虞はあまりありません。さらに、コレクションビューはデータ源オブジェクトやレイアウトオブジェクトにアクセスできるので、セルやビューを適切に操作するために必要な情報はすべて取得できます。

ジェスチャリコグナイザを使ってレイアウト情報を変更する

ジェスチャリコグナイザは、実行時に、レイアウトパラメータを手軽に変更する手段となります。たとえばピンチジェスチャにより、項目間の間隔を変更できます。このように設定する手順も、次のように比較的簡単です。

1. ジェスチャリコグナイザを生成する。
2. これをコレクションビューに組み込む。
3. ジェスチャリコグナイザのハンドラメソッドとして、レイアウトパラメータを更新し、レイアウトオブジェクトを無効化する処理を実装する。

ジェスチャリコグナイザを生成するためには、一般のオブジェクトと同様に、`alloc/init`の処理が必要です。初期化の際に、ターゲットオブジェクトと、ジェスチャを認識したときに呼び出すアクションメソッドを指定します。次にコレクションビューの`addGestureRecognizer:`メソッドで、ジェスチャリコグナイザを組み込みます。実際の処理の大部分は、初期化の際に指定したアクションメソッドの中で行うことになります。

リスト 4-1に、「ピンチ」ジェスチャを認識したときに呼び出される、アクションメソッドの実装例を示します。この例ではピンチデータを使って、セル間の間隔を変更しています。レイアウトオブジェクトには`updateSpreadDistance`メソッドを実装します。間隔の値が適切な範囲内かどうか確認

し、後でレイアウト処理の際に使えるよう保存する、という処理になります。アクションメソッドでは次に、現在のレイアウトを無効化することにより、新しい値に基づいて強制的に位置を更新されるようにします。

リスト 4-1　ジェスチャリコグナイザを使ってレイアウト値を変更するコード例

```
- (void)handlePinchGesture:(UIPinchGestureRecognizer *)sender {
    if ([sender numberOfTouches] != 2)
        return;

   // Get the pinch points.
   CGPoint p1 = [sender locationOfTouch:0 inView:[self collectionView]];
   CGPoint p2 = [sender locationOfTouch:1 inView:[self collectionView]];

   // Compute the new spread distance.
    CGFloat xd = p1.x - p2.x;
    CGFloat yd = p1.y - p2.y;
    CGFloat distance = sqrt(xd*xd + yd*yd);

   // Update the custom layout parameter and invalidate.
   MyCustomLayout* myLayout = (MyCustomLayout*)[[self collectionView]
collectionViewLayout];
   [myLayout updateSpreadDistance:distance];
   [myLayout invalidateLayout];
}
```

ジェスチャリコグナイザを生成し、ビューに組み込む方法については、『*EventHandlingGuideforiOS*』を参照してください。

デフォルトのジェスチャ動作に優先する動作を組み込む

`UICollectionView`クラスはシングルタップを検知すると、強調表示/選択の処理を行うデリゲートメソッドを起動します。「タップ」や「長押し」の範疇に属する独自のジェスチャを追加する場合、何らかのパラメータを既に使われているのとは異なる値にして、識別できるようにします。たとえば、「タップ」の中でも「ダブルタップ」のみを認識する、独自のジェスチャリコグナイザを組み込みたいとしましょう。

リスト4-2に、コレクションビューがジェスチャに応じて、選択/強調表示以外の処理をするようにしたコード例を示します。コレクションビューがジェスチャリコグナイザを使ってデリゲートメソッドを起動することはありません。したがって、カスタムジェスチャリコグナイザが、デフォルトの選択検知処理よりも優先的に扱われるようにすることができます。これは、他のタッチイベントの登録を遅らせる（ジェスチャリコグナイザの`delaysTouchesBegan`プロパティを`YES`とする）か、またはタッチイベントをキャンセルする（`cancelsTouchesInView`プロパティを`YES`とする）ことにより行います。するとコレクションビューは、タップを検知する都度、カスタムジェスチャリコグナイザに優先的に処理を委ねるかどうか判断するようになります。入力を調べた結果、カスタムジェスチャリコグナイザで処理するべきものでないと判断すれば、通常通りデリゲートメソッドを呼び出します。

リスト 4-2　カスタムジェスチャリコグナイザが優先的に処理するよう設定するコード例

```
UITapGestureRecognizer* tapGesture = [[UITapGestureRecognizer alloc]
initWithTarget:self action:@selector(handleTapGesture:)];

tapGesture.delaysTouchesBegan = YES;

tapGesture.numberOfTapsRequired = 2;

[self.collectionView addGestureRecognizer:tapGesture];
```

セルやビューを操作する

ジェスチャリコグナイザを使ってセルやビューを操作する方法は、その操作の内容によって異なります。単純な挿入や削除は、標準ジェスチャリコグナイザのアクションメソッド内で実行できます。しかし、より複雑な操作の場合、独自のジェスチャリコグナイザを定義し、タッチイベントを自分で追跡する必要があるかもしれません。

独自のジェスチャリコグナイザが必要な操作としては、コレクションビュー内のセルを、ある位置から別の位置に移動する、というものがあります。すぐに思いつく方法として、（一時的に）セルを削除し、ジェスチャリコグナイザを使ってドラッグ操作に応じたセルのアニメーション表示を行い、タッチイベントが終了した時点で新しい位置にセルを挿入する、というやり方があります。これを実現するためには、タッチイベントを自分で管理し、レイアウトオブジェクトと密に連携して新しい挿入位置を判断し、データ源の変更操作を行い、新しい位置に項目を挿入する必要があります。

独自のジェスチャリコグナイザを作成する方法については、『*Event Handling Guide for iOS*』を参照してください。

カスタムレイアウトの作成

カスタムレイアウトを構築するに当たっては、まず、本当に必要かどうかよく検討してください。`UICollectionViewFlowLayout`クラスには多くの機能が実装されています。処理効率を考慮して最適化済みであり、工夫次第で、一般的な多くのレイアウトに対応できます。次のような場合に限って、カスタムレイアウトの実装を検討するとよいでしょう。

- 格子や行を基本とした（行に沿って並べ、端に達したら次の行に移る）レイアウトではない場合、あるいは複数の方向にスクロールしなければならない場合。
- セルの位置が頻繁に変わるため、既存のフローレイアウトを利用すると手を加える箇所が多数にわたる（カスタムレイアウトの方が手間が省ける）場合。

とは言え、APIの観点からすると、カスタムレイアウトの実装はそれほど難しくありません。最も難しいのは、項目の位置を決める計算です。各項目の位置が決まってしまえば、その情報をコレクションビューに伝える処理は簡単です。

UICollectionViewLayoutのサブクラスを定義する

カスタムレイアウトを実装するためには、まず`UICollectionViewLayout`のサブクラスを定義し、設計の出発点とします。このクラスのメソッドのうち、レイアウト処理の中心的な役割を担い、したがって実装する必要があるのは、ごく少数に過ぎません。それ以外のメソッドは、レイアウト調整のために必要ならばオーバーライドしてください。中核となるメソッドは、次に挙げる重要なタスクを実行します。

- スクロール可能なコンテンツ領域の大きさを指定する。
- レイアウト対象であるセルやビューの属性オブジェクトを管理して、コレクションビューがその位置を計算できるようにする。

中核となるメソッドを実装するだけでも構いませんが、それ以外のメソッドも実装すれば、より高度なレイアウトが可能になります。

レイアウトオブジェクトは、データ源から渡される情報をもとに、コレクションビューのレイアウトを決めていきます。データ源とのやり取りには、`collectionView`プロパティで示されるオブジェクト（レイアウトオブジェクトのどのメソッドからもアクセス可）のメソッドを使います。レイアウト処理に際して、コレクションビューが何を認識し、何を認識しないか、把握しておくことが重要で

す。レイアウト処理の進行中、コレクションビューがその進捗状況や各ビューの位置を追跡することはできません。レイアウトオブジェクトはコレクションビューのメソッドを自由に呼び出せますが、レイアウトの計算に必要でないデータを取得することは避けてください。

中核となるレイアウト処理を把握する

コレクションビューは、カスタムレイアウトオブジェクトに直接働きかけ、レイアウト処理全般を管理します。また、レイアウト情報が必要と判断すると、レイアウトオブジェクトに対して問い合わせます。具体的には、最初に表示する時点や、大きさが変わったときです。また、コレクションビューに対して明示的に、レイアウトを更新するよう要求することも可能です。これは、レイアウトオブジェクトの`invalidateLayout`メソッドを呼び出して行います。このメソッドには、既存のレイアウト情報を破棄し、新たにレイアウト情報を生成するよう強制する、という働きがあるのです。

> **注意:** レイアウトオブジェクトの`invalidateLayout`メソッドを、コレクションビューの`reloadData`メソッドと混同しないでください。`invalidateLayout`メソッドを実行しても、コレクションビューが既存のセルやサブビューを破棄することにはなりません。レイアウトオブジェクトが処理する項目を移動/追加/削除したとき、必要なレイアウト属性をすべて計算し直すよう強制するだけです。データ源のデータが変化した場合は、`reloadData`メソッドを実行してください。どのような方法でレイアウトの更新を始めたとしても、実際のレイアウト処理は同じです。

レイアウト処理に際してコレクションビューは、レイアウトオブジェクトの中核メソッドを呼び出します。このメソッドの中で、項目の位置を計算し、必要な情報をコレクションビューに返すことになります。ほかのメソッドも呼び出されることがありますが、少なくとも上記の中核メソッドは、レイアウト処理中、必ず次の順序で呼び出されます。

1. `prepareLayout`メソッド。事前にレイアウト情報を計算します。
2. `collectionViewContentSize`メソッド。先に計算した結果に基づき、コンテンツ全体を収容する領域の大きさを返します。
3. `layoutAttributesForElementsInRect:`メソッド。指定された矩形内にあるセルやビューの属性を返します。

図 5-1に、上記のメソッドでレイアウト情報を計算する手順を示します。

図 5-1　　カスタムコンテンツのレイアウト処理

prepareLayoutメソッドでは、セルやビューの位置を決めるために必要な計算を何でも実行できます。少なくとも、手順（2）でコンテンツ領域全体の大きさを返すために必要な情報は、ここで計算しておくとよいでしょう。

コレクションビューはこの大きさに基づき、スクロールビューを適切に設定します。たとえば、幅、高さとも、デバイス画面に入りきらない大きさであれば、どちらの方向にもスクロールできるようにするのです。UICollectionViewFlowLayoutと違って、一方向のみのスクロール動作がデフォルトにはなっていません。

次にコレクションビューは、現在のスクロール位置に基づき、ある矩形を指定してlayoutAttributesForElementsInRect:メソッドを呼び出し、その内部にあるセルやビューの属性を問い合わせます。この矩形は、現在可視になっている領域と同じかもしれませんし、そうでないかもしれません。レイアウト処理では、この情報をもとに、実際に配置を計算します。

レイアウト処理が終了しても、セルやビューの属性は、そのレイアウトが無効になるまで変わりません。レイアウトオブジェクトのinvalidateLayoutメソッドを実行すると、再びレイアウト処理がprepareLayoutメソッドを呼び出すところから始まります。スクロール処理の一環として、コレクションビューの側が自動的に現在のレイアウトを無効にすることもあります。コンテンツをスクロールすると、コレクションビューはレイアウトオブジェクトのshouldInvalidateLayoutForBoundsChange:メソッドを呼び出し、その戻り値がYESであれば現在のレイアウトを無効にするようになっています。

> **注意:** invalidateLayoutメソッドを呼び出しても、直ちにレイアウトの更新処理が始まるわけではないことを、頭に入れておくとよいでしょう。データに合ったレイアウトになっていない（更新が必要である）旨の印をつけるだけです。コレクションビューは次のビュー更新サイクルでこの印を確認し、実際に更新するようになっています。したがって、invalidateLayoutメソッドを短時間のうちに複数回呼び出しても、その都度レイアウトが更新されるとは限りません。

レイアウト属性を生成する

レイアウトに関係する属性オブジェクトは、UICollectionViewLayoutAttributesクラスに属します。アプリケーションは状況に応じ、適切なメソッドでこのインスタンスを生成します。項目数が数千個以上に及ぶのでなければ、レイアウトの準備処理中に生成するとよいでしょう。レイアウト情報

をその都度計算するのではなく、事前に求めてキャッシュしておき、必要に応じて参照できるようにするのです。一方、事前に属性値を求めるコストが、キャッシュによる利益に見合わないようであれば、必要になった時点で属性オブジェクトを生成するようにしてください。

いずれにしても、`UICollectionViewLayoutAttributes`クラスのインスタンス生成には、次のいずれかのクラスメソッドを使います。

- `layoutAttributesForCellWithIndexPath:`
- `layoutAttributesForSupplementaryViewOfKind:withIndexPath:`
- `layoutAttributesForDecorationViewOfKind:withIndexPath:`

表示するビューの種類に応じて、適切なクラスメソッドを使わなければなりません。コレクションビューはこの情報に基づき、データ源オブジェクトから適切な種類のビューを要求します。不適切なメソッドを使うと、コレクションビューは不正な位置に誤ったビューを生成することになり、意図どおりの表示にならない虞があります。

生成した属性オブジェクトに、対応するビューの属性を設定します。少なくともビューの大きさおよび位置は設定しなければなりません。ビューどうしが重なり合うレイアウトの場合、重なり合いの順序が不定にならないよう`zIndex`プロパティも設定してください。ほかにもセルやビューの表示や外観を制御するプロパティがあり、必要に応じて変更も可能です。標準の属性クラスがアプリケーションの要件に合わなければ、そのサブクラスを定義し、各ビューに関する他の情報も格納できるよう拡張してください。その際、属性どうしを比較する`isEqual:`メソッドも実装する必要があります。コレクションビューはいくつかの処理のためにこのメソッドを呼び出すからです。

レイアウト属性について詳しくは、『*UICollectionViewLayoutAttributesClassReference*』を参照してください。

レイアウト処理の準備をする

レイアウトサイクルの先頭で、レイアウトオブジェクトは実際のレイアウト処理に先立ち、`prepareLayout`を呼び出します。このメソッド内で、あらかじめレイアウト情報を計算しておくとよいでしょう。`prepareLayout`メソッドは、必須ではありませんが、事前に何らかの計算をしておく必要がある場合、その処理を記述する場所として適しています。このメソッドの実行を終えた時点で、次にコレクションビューの中身の大きさを求める際に必要となる情報が揃っていなければなりません。ごく最小限の情報で足りる場合もありますが、状況によっては、レイアウト処理に用いる属性オブジェクトをすべて生成、保存しておかなければならないでしょう。`prepareLayout`メソッドを実装するかどうか、アプリケーションの基盤機能や、事前に計算する方法と必要に応じて計算する方法の得失を考慮して判断してください。`prepareLayout`メソッドの実装例が“レイアウト処理の準備をする” （51 ページ）に載っています。

指定された矩形内に表示する項目のレイアウト属性を与える

レイアウト処理の最終段階で、コレクションビューはレイアウトオブジェクトの`layoutAttributesForElementsInRect:`メソッドを呼び出します。このメソッドの目的は、指定された矩形内に一部でも含まれるセルや補助ビュー、装飾ビューの、レイアウト属性を与えることです。スクロール可能な広いコンテンツ領域のレイアウトをする場合、コレクションビューは、現在可視になっている領域にある項目についてのみ属性を問い合わせます。たとえば図 5-2のような状況では、セル6〜20と2番目のヘッダビューの属性オブジェクトのみ生成すればよいことになります。それ以外の領域にある項目についても、問い合わせがあればレイアウト属性を返せるよう準備しておかなければなりません。項目を挿入/削除したときのアニメーション効果を生み出すために必要になることもありうるからです。

図 5-2　可視になっているビューのみレイアウトする様子

`layoutAttributesForElementsInRect:`メソッドは、レイアウトオブジェクトの`prepareLayout`メソッドよりも後で呼び出されるので、要求された属性を返すために必要な情報は、既にほとんど揃っているはずです。したがって`layoutAttributesForElementsInRect:`メソッドの実装は次のようになります。

1. `prepareLayout`メソッドで生成した各データについて、キャッシュ済みの属性を取得するか、または新たに生成する。
2. 各項目の位置と大きさを調べて、`layoutAttributesForElementsInRect:`メソッドに渡された矩形と共通部分があるかどうか判定する。
3. 共通部分がある場合、対応する`UICollectionViewLayoutAttributes`オブジェクトを配列に追加する。
4. 最後に、レイアウト属性の配列をコレクションビューに返す。

`UICollectionViewLayoutAttributes`オブジェクトは、レイアウト情報の管理方法に応じ、`prepareLayout`メソッド内で事前に生成しても、`layoutAttributesForElementsInRect:`メソッドが呼び出されてから生成しても構いません。実装はアプリケーションの要件に応じて行いますが、レイアウト情報をキャッシュ保存する方式の得失にも注意してください。同じセルについて、必要の都度新たにレイアウト属性を計算する方式は、相応の負荷がかかります。キャッシュを活用すれば、処理性能の面で顕著な効果がえられるでしょう。とは言え、項目が相当数に及ぶならば、要求に応じて生成する方が逆に効率的かも知れません。どちらの方針を採用するか、アプリケーションごとに検討してください。

> **注意:** レイアウトオブジェクトは、要求があれば個々の項目に関するレイアウト属性も返せるよう、準備しておかなければなりません。コレクションビューは、アニメーション表示その他のため、通常のレイアウト処理とは別に、こういった情報を問い合わせることがあるのです。要求に応じてレイアウト属性を提供することについて詳しくは、“要求に応じてレイアウト属性を返す” （42 ページ）を参照してください。

`layoutAttributesForElementsInRect:`の実装例が“レイアウト属性を返す” （55 ページ）に載っています。

要求に応じてレイアウト属性を返す

コレクションビューはレイアウトオブジェクトに対して、通常のレイアウト処理とは別に、個々の項目の属性を問い合わせることがあります。たとえば、挿入や削除の操作の際、アニメーション表示のためにこの情報が必要になることがあるのです。レイアウトオブジェクトは、セル、補助ビュー、装飾ビューのそれぞれについて、いつでもレイアウト属性を返せるよう備えておかなければなりません。これは次のメソッドをオーバーライドすることにより行います。

- `layoutAttributesForItemAtIndexPath:`
- `layoutAttributesForSupplementaryViewOfKind:atIndexPath:`
- `layoutAttributesForDecorationViewOfKind:atIndexPath:`

このメソッドに、指定されたセルやビューの、現在のレイアウト属性を検索して返す処理を実装します。カスタムレイアウトオブジェクトには、`layoutAttributesForItemAtIndexPath:`メソッドを実装する必要があります。一方、`layoutAttributesForSupplementaryViewOfKind:atIndexPath:`メソッドは、補助ビューがないのであれば、実装しなくても構いません。同様に`layoutAttributesForDecorationViewOfKind:atIndexPath:`メソッドも、装飾ビューがなければ不要です。属性を返す際に、レイアウト属性を更新してはなりません。レイアウト情報を変更する必要があれば、レイアウトオブジェクトを無効化し、次のレイアウトサイクルでデータが更新されるようにしてください。

カスタムレイアウトをコレクションビューに結びつける

カスタムレイアウトをコレクションビューに関連づける方法として、プログラムで記述する方法と、ストーリーボードを使う方法があります。いずれにしてもこの関連づけは、コレクションビューの`collectionViewLayout`プロパティに、該当するカスタムレイアウトオブジェクトを設定することにより行います。リスト 5-1にその記述例を示します。

リスト 5-1　カスタムレイアウトを関連づけるコード例

```
self.collectionView.collectionViewLayout = [[MyCustomLayout alloc] init];
```

ストーリーボードを使う場合は、「Document Outline」パネルを開き、該当するコレクションビューを（コントローラのドロップダウンメニューから）選択してください。この状態で「Utilities」ペインに「Attributes」インスペクタを開き、「Collection View」セクション以下にある「Layout」を、「Flow」から「Custom」に変更します。すると、すぐ下の選択肢が「Scroll Direction」から「Class」に変わるので、ここで該当するカスタムレイアウトのクラスを選択します。

より魅力的なレイアウトにする

レイアウト処理の際には各セルやビューのレイアウト属性が必要ですが、ほかにも使いやすいカスタムレイアウトにするための機能があります。どうしても必要というわけではありませんが、できるだけ実装するようお勧めします。

補助ビューを使って表示を改善する

補助ビューはコレクションビューのセルとは独立しており、レイアウト属性も独自に設定できます。セルと同様にデータ源オブジェクトから必要な情報を受け取りますが、その目的はアプリケーションの表示を見やすくすることです。たとえば`UICollectionViewFlowLayout`は、セクションのヘッダ/フッタとして補助ビューを利用します。あるいは、セルごとにテキストラベルを用意し、当該セルについての情報を表示するために、補助ビューを用いるアプリケーションもあります。セルと同様、コレクションビューのリソース消費量を抑制するため、リソースを回収する仕組みの対象になります。したがって、補助ビューはすべて、`UICollectionReusableView`のサブクラスとして実装しなければなりません。

補助ビューをレイアウトに組み込む手順を以下に示します。

1. `registerClass:forSupplementaryViewOfKind:withReuseIdentifier:`メソッドまたは`registerNib:forSupplementaryViewOfKind:withReuseIdentifier:`メソッドで、コレクションビューのレイアウトオブジェクトに補助ビューを登録します。
2. データ源に`collectionView:viewForSupplementaryElementOfKind:atIndexPath:`メソッドを実装します。使用済みのビューは再利用可能であり、`dequeueReusableSupplementaryViewOfKind:withReuseIdentifier:forIndexPath:`メソッドでキューから取得できます。キューにない場合は新規に生成し、必要なデータを設定して返します。
3. 補助ビューのレイアウト属性オブジェクトを、セルの場合と同様に生成します。

4. `layoutAttributesForElementsInRect:`メソッドで取得したレイアウト属性配列に、補助ビューのレイアウト属性オブジェクト（上記）を追加します。
5. 指定された補助ビューの属性オブジェクトを返す、`layoutAttributesForSupplementaryViewOfKind:atIndexPath:`メソッドを実装します。

カスタムレイアウトに用いる補助ビューの属性オブジェクトを生成する手順は、セルの場合とほぼ同じです。ただし、補助ビューはいくつかの型を混在させることができるのに対し、セルはすべて同じ型である点が異なります。これは、補助ビューの目的が表示を見やすくすることであり、したがって主たる表示内容とは独立しているからです。補助ビューの使い方はさまざまなので、その各メソッドは引数として型を受け取り、これに応じて適切に属性を計算できるようになっています。レイアウトオブジェクトは、補助ビューの登録時に渡される文字列を使って、他の補助ビューと区別します。補助ビューをカスタムレイアウトに組み込む例を"補助ビューを組み込む"（57 ページ）に示します。

装飾ビューをカスタムレイアウトに取り入れる

装飾ビューは、コレクションビューのレイアウトを見やすくする、見た目の効果を提供します。セルや補助ビューと違い、外観を変えるだけの要素であって、データ源から情報を得ることもありません。独自の背景を与える、セルの周囲に埋める、セルを覆い隠す、などの用途が考えられます。もっぱらレイアウトオブジェクトが定義するものであり、コレクションビューのデータ源オブジェクトは関与しません。

装飾ビューをレイアウトに追加する手順は次の通りです。

1. 装飾ビューをレイアウトオブジェクトに、`registerClass:forDecorationViewOfKind:`メソッドまたは`registerNib:forDecorationViewOfKind:`メソッドで追加する。セルや補助ビューと同様ですが、装飾ビューの場合、データ源ではなくレイアウトオブジェクトに登録することに注意してください。
2. レイアウトオブジェクトの`layoutAttributesForElementsInRect:`メソッドで、装飾ビューの属性を生成する（手順はセルや補助ビューと同様）。
3. レイアウトオブジェクトの`layoutAttributesForDecorationViewOfKind:atIndexPath:`メソッドを実装し、要求に応じて装飾ビューの属性を返すようにする。
4. 必要ならば`initialLayoutAttributesForAppearingDecorationElementOfKind:atIndexPath:`メソッドや`finalLayoutAttributesForDisappearingDecorationElementOfKind:atIndexPath:`メソッドに、装飾ビューの表示/消去時に行うアニメーション処理を実装する。詳しくは"挿入/削除時により面白いアニメーションを表示する"（45 ページ）を参照してください。

装飾ビューの生成処理は、セルや補助ビューの場合とは違います。クラスまたはnibファイルを登録するだけで、装飾ビューは必要に応じて生成されるようになります。純粋に見た目を変えるだけのビューなので、nibファイルや`initWithFrame:`メソッドで実行した以上の設定処理は必要ないので

す。したがって、装飾ビューが必要になると、コレクションビューが生成し、レイアウトオブジェクトから取得した属性を適用します。装飾ビューもリソース回収機構の対象になるので、`UICollectionReusableView`のサブクラスでなければなりません。

> **注意:** 装飾ビューの属性を生成する際には、`zIndex`プロパティを考慮することも忘れないでください。この属性を適切に指定することにより、セルや補助ビューの奥に（あるいは前面に）配置することができます。

挿入/削除時により面白いアニメーションを表示する

レイアウトを設計する上で、セルやビューを挿入/削除する際の動作は検討に値します。セルを挿入すると、他のセルやビューのレイアウトも変わることがあります。既存のセルやビューの位置を動かす場合、そのアニメーション表示のやり方はレイアウトオブジェクトが知っていますが、新たに挿入する場合、そのセルには（アニメーションの起点となる）現在の位置というものがありません。だからといってアニメーションなしで挿入するわけにもいかないので、コレクションビューはレイアウトオブジェクトに対し、アニメーション表示に使う起点属性を問い合わせます。セルを削除する場合も同様に、アニメーションの最終位置を表す終点属性を問い合わせるようになっています。

起点属性の働きは、例を示して説明すると分かりやすいでしょう。当初はコレクションビューに、セルが3つしか収容されていないとします（図 5-3を参照）。新しいセルを挿入すると、コレクションビューはレイアウトオブジェクトに対し、このセルの起点属性を要求します。この例の場合、レイアウトオブジェクトは、セルの起点位置としてコレクションビューの中央を設定し、アルファ値を0として不可視にしています。アニメーション効果により、この新しいセルは、コレクションビューの中央から最終位置である右下隅に移動しながら、徐々にその姿を現します。

図 5-3　起点属性を指定して徐々に姿を現すようにしている様子

リスト5-2に、新たに挿入したセルが図5-3のように徐々に姿を現すよう、起点属性を指定するコード例を示します。このメソッドは、セルの位置としてコレクションビューの中央を設定し、さらに「透明」属性も与えています。レイアウトオブジェクトは次に、通常のレイアウト処理の一環として、このセルの最終位置およびアルファ値を与えます。

リスト 5-2　挿入するセルの起点属性を指定するコード例

```
- (UICollectionViewLayoutAttributes
*)initialLayoutAttributesForAppearingItemAtIndexPath:(NSIndexPath *)itemIndexPath
 {

   UICollectionViewLayoutAttributes* attributes = [self
layoutAttributesForItemAtIndexPath:itemIndexPath];

   attributes.alpha = 0.0;
```

```
    CGSize size = [self collectionView].frame.size;
    attributes.center = CGPointMake(size.width / 2.0, size.height / 2.0);
    return attributes;
}
```

> **注意:** リスト 5-2に、セルの挿入時にアニメーション表示を行うコード例を示します。第4のセルを挿入する際、既存の3つのセルも、コレクションビューの中央からポップアウトします。挿入するセルのみアニメーション表示したい場合は、そのインデックスパスを、`prepareForCollectionViewUpdates:`メソッドに渡すセルのインデックスパスと比較し、一致した場合のみアニメーション表示するようにしてください。一致しなかった場合は、`initialLayoutAttributesForAppearingItemAtIndexPath:`のsuperメソッドの戻り値である属性を返します。

削除の場合も挿入と同様ですが、起点属性の代わりに終点属性を指定します。先の例に示した起点属性と同じものを終点属性として指定すれば、セルを削除したとき、コレクションビューの中央に向かって移動しながら徐々に消えていくようになります。`UICollectionViewLayout`クラスには6つのメソッドがあります。アイテム、補助ビュー、装飾ビューのそれぞれについて、起点属性または終点属性を指定するメソッドです。

スクロール時の動作を改善する

カスタムレイアウトオブジェクトを実装すれば、スクロール時のコレクションビューの動作を、使いやすく改善することも可能です。スクロール操作のタッチイベントが終わると、スクロールビューは、その時点の実効速度と実効減速率をもとに、最終的にどの位置でスクロールを止めるか、を判断します。コレクションビューは、レイアウトオブジェクトの`targetContentOffsetForProposedContentOffset:withScrollingVelocity:`メソッドを呼び出して、この位置を補正するかどうか問い合わせます。コンテンツがまだ動いている間にこのメソッドを呼び出すので、カスタムレイアウトが、最終的にスクロールが止まる位置に影響を与えることもありえます。

図 5-4に、レイアウトオブジェクトを使ってスクロール時の動作を改善する一例を示します。コレクションビューのオフセットが当初(0,0)から始まるとし、ユーザが左にスワイプしたとしましょう。コレクションビューは、スクロールが減速して自然に止まる位置を計算し、これをオフセット値として「提案」します。レイアウトオブジェクトは、スクロールが止まったとき、項目の中央がちょうど、

コレクションビューの可視領域の境界に来るように、この値を補正します。`targetContentOffsetForProposedContentOffset:withScrollingVelocity:`メソッドが返すのは、この補正済みのオフセット値です。

図 5-4　提案されたオフセット値を補正する様子

カスタムレイアウトを実装する上でのヒント

カスタムレイアウトオブジェクトを実装する上でのヒントや提案をいくつか示します。

- レイアウト属性オブジェクト（`UICollectionViewLayoutAttributes`）は、`prepareLayout`メソッドで事前に生成し、保存しておくとよいかもしれません。コレクションビューはいつでもレイアウト属性オブジェクトを要求する可能性があるので、この方法には一考の価値があります。特に、項目が比較的少ない（数百程度）、あるいは項目のレイアウト属性がそれほど変化しないのであれば、ぜひ検討してください。

  一方、数千以上の項目を配置する必要がある場合は、このように事前に生成してキャッシュする方法と、その都度再計算する方法を、比較してみるべきでしょう。大きさが可変の項目でも、レイアウトがまれにしか変わらない場合、キャッシュを活用すれば、複雑なレイアウト計算を頻繁に実行しなくてもよくなります。これに対し、大きさが固定の項目が多数ある場合は、必要が生じてから属性を計算する方が簡単かもしれません。また、属性が頻繁に変わる項目があれば、いずれにしてもその都度計算し直すことになるので、キャッシュは単にメモリ空間の無駄になってしまう可能性もあります。

- `UICollectionView`のサブクラスを定義することは避けてください。コレクションビューに独自の外観というものはほとんどありません。ビューはすべてデータ源オブジェクトから、レイアウトに必要な情報はすべてレイアウトオブジェクトから取得します。3次元空間に項目をレイアウトする場合は、カスタムレイアウトを実装し、セルやビューに、3次元空間上の位置を平面に射影する座標変換を適切に設定するとよいでしょう。

- `UICollectionView`の`visibleCells`メソッドを、カスタムレイアウトオブジェクトの`layoutAttributesForElementsInRect:`メソッドから呼び出すことは避けてください。コレクションビューは、項目を配置する位置については何も知らず、レイアウトオブジェクトに問い合わせることしかできません。したがって、どのセルが可視かを問い合わせても、その問い合わせはレイアウトオブジェクトに戻ってくるだけです。

  レイアウトオブジェクトは常に、コンテンツ領域における各項目の位置を把握し、いつでも当該項目の属性を返せるようにしておかなければなりません。多くの場合、これはレイアウトオブジェクト内部だけで実行できます。ごく稀に、データ源から何らかの情報を得ないと、項目の位置が決められない場合があります。たとえば、地図上に項目を表示するレイアウトの場合、各項目の地図上の位置は、データ源から取得しなければなりません。

カスタムレイアウト：実例

コレクションビューのカスタムレイアウトを行う手順は、要件が明確になっていれば決して難しくありませんが、細かく見ていけば、状況によってさまざまです。いずれにしても、コレクションビューに収容するビューそれぞれについて、レイアウト属性オブジェクトを生成する必要があります。どの段階でレイアウト属性を生成するかは、アプリケーションの性質に応じて判断しなければなりません。コレクションビューに収容する項目が数千個に及ぶ場合、事前にレイアウト属性を計算してキャッシュ保存するためには相当の処理時間を要するので、必要が生じた時点で、該当する項目についてのみ属性を生成するとよいでしょう。一方、項目数が少なければ、全項目について一度だけ計算し、キャッシュしておくことにより、無駄に同じ計算を繰り返す必要がなくなります。この章で取り上げる実例は後者に当たります。

もっとも、以下に示す例が、カスタムレイアウトを生成する唯一の方法というわけではありません。開発に着手する前に少し時間を取って、アプリケーションの実行性能が向上するよう、実装の構成を検討してみましょう。レイアウトをカスタマイズする大まかな流れについては、“カスタムレイアウトの作成”（37 ページ）を参照してください。

この章ではカスタムレイアウトの実装手順に沿って説明しているので、最終的な目標を念頭に、実装例を先頭から順に読み進めるとよいでしょう。ただし、ここに示すのはカスタムレイアウトの作成部分だけであり、完全なアプリケーションの実装にはなっていません。最終的な製品にするために必要な、ビューやコントローラの実装については省略します。コレクションビューに収容するカスタムセルを、レイアウト上の「セル」として使います。また、セル間をつなぐ線を、カスタム「補助ビュー」として描画します。こういったセルや補助ビューを生成する手順、コレクションビューを使うための要件については、既に解説しました。“コレクションビューの基本事項”（9 ページ）、“データ源およびデリゲートの設計”（15 ページ）を参照してください。

レイアウトの考え方

以下、実例として、情報の階層木を図6-1のような形で表示する、カスタムレイアウトを実装します。コードを断片ごとに示すとともに、その説明と、カスタマイズする上でのポイントを述べていきます。コレクションビューの各セクションが、階層木のある深さに対応します。第0セクション（一番左の列）に入るのは、ルートに当たる「NSObject」セルだけです。第1セクション（左から2番目の列）には、NSObjectの直下に当たる子セルが並びます。さらに右の第2セクションにあるのは、その子（NSObjectから見れば孫）に当たるセルです。セルはいずれもカスタムセルで、クラス名を表すラ

ベルがついています。セル間を結ぶ線は補助ビューを使って描画します。この「コネクタ」ビュークラスは、線の本数を判断するために、データ源から情報を取得しなければなりません。したがって、その実装には装飾ビューでなく補助ビューを使うことになります。

図 6-1　クラス階層

初期化

カスタムレイアウトを作成するためには、まず、`UICollectionViewLayout`のサブクラスを定義します。これにより、カスタムレイアウトの構築に必要な基盤が用意されます。

今回の例では、項目間の間隔に関する情報を受け渡しするために、カスタムプロトコルが必要です。ある項目の属性値を決めるために、データ源から情報を取得する必要がある場合、このデータ源に直接接続して問い合わせるのではなく、カスタムレイアウト用のプロトコルを実装するとよいでしょう。こうすればレイアウトは、再利用性が高い堅実なものになります。特定のデータ源に縛られることがなく、このプロトコルさえ実装すれば、あらゆるデータ源オブジェクトに対応できるようになるのです。

リスト 6-1に、カスタムレイアウトのヘッダファイルに記述するべきコードを示します。その結果、*MyCustomProtocol* プロトコルを実装したクラスは、カスタムレイアウトによる描画が可能になります。レイアウト処理コードからは、当該クラスに対し、このプロトコルを用いて必要な情報を問い合わせることになります。

リスト 6-1　カスタムプロコトルに接続するコード例

```
@interface MyCustomLayout : UICollectionViewLayout
@property (nonatomic, weak) id<MyCustomProtocol> customDataSource;
@end
```

コレクションビューが管理する項目数は比較的少ないので、次に、カスタムレイアウトの処理コードにキャッシュ機構を組み込みます。準備処理中に生成したレイアウト属性を保存しておき、コレクションビューからの要求に応じて返すようにするのです。リスト 6-2に、レイアウト処理のために管理するべき3つの非公開プロパティと、`init`メソッドの実装を示します。`layoutInformation`辞書には、コレクションビューに収容するあらゆる型のビューについて、レイアウト属性をすべて格納します。`maxNumRows`プロパティは、階層木の描画に必要な、（最も多くの行を要する列の）行数を求めるために使います。`insets`オブジェクトはセル間の間隔を制御し、ビューを配置する枠や、その中身の大きさを設定するために使います。上記のうち2つのプロパティはレイアウトの準備処理中に設定し、`insets`オブジェクトは`init`メソッドで設定します。この例で、`INSET_TOP`、`INSET_LEFT`、`INSET_BOTTOM`、`INSET_RIGHT`は、各パラメータの値を表す定数です。

リスト 6-2　変数を初期化するコード例

```
@interface MyCustomLayout()


@property (nonatomic) NSDictionary *layoutInformation;
@property (nonatomic) NSInteger maxNumRows;
@property (nonatomic) UIEdgeInsets insets;


@end


-(id)init {
    if(self = [super init]) {
        self.insets = UIEdgeInsetsMake(INSET_TOP, INSET_LEFT, INSET_BOTTOM,
INSET_RIGHT);
    }
    return self;
}
```

最後に、カスタムレイアウト属性を生成します。必須の手順ではありませんが、今回の例では、セルの配置を決める際に、着目しているセルの子のインデックスパスを参照し、その枠を親に合わせて調整しなければなりません。そこで、`UICollectionViewLayoutAttributes`のサブクラスを定義し、セルの子の配列を格納して、必要な情報が得られるようにします。
`UICollectionViewLayoutAttributes`のサブクラスを定義し、ヘッダファイルに次の行を追加します。

```
@property (nonatomic) NSArray *children;
```

`UICollectionViewLayoutAttributes`のクラスリファレンスに説明があるように、レイアウト属性のサブクラスを定義する際には、`isEqual:`メソッドをオーバーライドする必要があります（iOS 7以降）。その理由については『*UICollectionViewLayoutAttributes Class Reference*』を参照してください。

今回の場合、`isEqual:`メソッドの実装は簡単で、子を列挙した配列を比較するだけです。この配列が一致していれば、子はすべて同じクラスに属するので、レイアウト属性どうしも一致していることになります。リスト 6-3に`isEqual:`メソッドの実装例を示します。

リスト 6-3　レイアウト属性のサブクラスに必要なコードの例

```
-(BOOL)isEqual:(id)object {
```

```
    MyCustomAttributes *otherAttributes = (MyCustomAttributes *)object;
    if ([self.children isEqualToArray:otherAttributes.children]) {
        return [super isEqual:object];
    }
    return NO;
}
```

カスタムレイアウトファイルには、カスタムレイアウト属性のヘッダファイルも忘れずに記述してください。

ここまで進めると、いよいよカスタムレイアウトの主要部の実装に取りかかることになります。

レイアウト処理の準備をする

必要なコンポーネントをすべて初期化したので、次にレイアウトの準備をします。コレクションビューはレイアウト処理に当たって、まずprepareLayoutメソッドを呼び出します。今回の例では、prepareLayoutメソッド内で、レイアウト属性オブジェクトのインスタンスを生成します。コレクションビューに収容する各ビューについて生成し、後で必要に応じて参照できるよう、キャッシュとしてlayoutInformation辞書に格納するのです。prepareLayoutメソッドについて詳しくは、“レイアウト処理の準備をする”（40 ページ）を参照してください。

レイアウト属性を生成する

prepareLayoutメソッドの実装例を、2つに分けて示します。図6-2はメソッドの前半部分の目標を表します。各セルについて、子があればこれを親セルに関連づける、という処理です。図のように、他のセルの子になっているセルを含め、すべてのセルについて実行します。

図 6-2　親と子のインデックスパスを接続した様子

リスト 6-4に、prepareLayoutメソッドの前半部分を示します。先頭で辞書（内容の書き換え可能）を初期化していますが、これがキャッシュ機構の基盤となります。一つ目の辞書であるlayoutInformationは、layoutInformationプロパティに相当する仕組みをローカルに用意したものです。書き換え可能な辞書をローカルに保持しているため、インスタンス変数の方は不変であって構いません。prepareLayoutメソッドの実行後は、レイアウト属性を改変することがないので、このように分けて管理します。続いて、セクションの番号順に、当該セクション内の各項目に対応するセルの属性を生成します。attributesWithChildrenForIndexPath:メソッド（別に実装）は、カスタムレイアウト属性のインスタンスを返します。そのchildrenプロパティには、指定したインデックスパスの項目から見た、子のインデックスパスが設定されています。この属性オブジェクトを、ロー

カルの`cellInformation`辞書に、インデックスパスをキーとして格納します。このように、前半部分のループで、項目すべてについて子を調べた結果、次に各項目を描画する枠を設定する準備が整いました。

リスト 6-4　レイアウト属性を生成するコード例

```
- (void)prepareLayout {
    NSMutableDictionary *layoutInformation = [NSMutableDictionary dictionary];
    NSMutableDictionary *cellInformation = [NSMutableDictionary dictionary];
    NSIndexPath *indexPath;
    NSInteger numSections = [self.collectionView numberOfSections;]
    for(NSInteger section = 0; section < numSections; section++){
        NSInteger numItems = [self.collectionView numberOfItemsInSection:section];
        for(NSInteger item = 0; item < numItems; item++){
            indexPath = [NSIndexPath indexPathForItem:item inSection:section];
            MyCustomAttributes *attributes =
            [self attributesWithChildrenAtIndexPath:indexPath];
            [cellInformation setObject:attributes forKey:indexPath];
        }
    }
    //end of first section
```

レイアウト属性を格納する

図 6-3に、`prepareLayout`メソッドの後半で行う処理の様子を示します。最終行から降順に、階層構造を構築していく手続きです。一見、奇妙に思われるかも知れませんが、子セルを配置する枠を調整するための複雑な処理を回避できる、巧妙なやり方です。子セルを配置する枠は、親セルの枠と正しく照応していなければなりません。また、セル間の間隔（間にはさむ行数）は、当該セルの子（や、さらにその子孫）の個数に依存して決まります。したがって、子の枠をまず配置し、その後に親の枠を配置する、という順序の方が処理しやすいのです。この方式で進めれば、子セルやその子孫セルすべての配置を、その全体を束ねる親セルに照応するよう調整できます。

ステップ1では、最終列のセルを順に並べて配置します。ステップ2で、末尾から2番目の列について、配置する枠を決めていきます。この列には2つ以上の子を持つセルがないので、各セルを順に並べていくだけで構いません。ただし、緑のセルの枠は、親セルの枠に照応させるため、下に1行ずらします。最後に第1列のセルを配置します。この列の第1セルには子が3つある（第2列の先頭3つ）ので、第2セル以降は下にずらす必要があります。実際には、第2セル以降には子がないのでずらさなくても

描画はできるのですが、レイアウトオブジェクトはそこまで賢くありません。以降のセルにも子があることを想定して、必ず間隔をあけるようになっています。先と同様に、緑のセルをどちらも下にずらして、親セルの枠に照応させます。

図 6-3　　セルを枠に配置する処理の様子

リスト 6-5にprepareLayoutメソッドの後半部分を示します。各項目を枠に配置していく処理です。番号を添えた行について、以下に説明します。

リスト 6-5　レイアウト属性を設定するコード例

```
    //continuation of prepareLayout implementation
    for(NSInteger section = numSections - 1; section >= 0; section--){
        NSInteger numItems = [self.collectionView numberOfItemsInSection:section];
        NSInteger totalHeight = 0;
        for(NSInteger item = 0; item < numItems; item++){
            indexPath = [NSIndexPath indexPathForItem:item inSection:section];
           MyCustomAttributes *attributes = [cellInfo objectForKey:indexPath]; //
1
            attributes.frame = [self frameForCellAtIndexPath:indexPath
                                withHeight:totalHeight];
          [self adjustFramesOfChildrenAndConnectorsForClassAtIndexPath:indexPath];// 2

            cellInfo[indexPath] = attributes;
            totalHeight += [self.customDataSource
                            numRowsForClassAndChildrenAtIndexPath:indexPath];     // 3
        }
        if(section == 0){
            self.maxNumRows = totalHeight;                                        // 4
        }
    }
    [layoutInformation setObject:cellInformation forKey:@"MyCellKind"];           // 5
    self.layoutInformation = layoutInformation
}
```

リスト 6-5では、セクション番号の降順に処理を繰り返し、ボトムアップ方式で階層木を組み立てています。変数totalHeightは、着目している項目を下にずらす行数を表します。間隔調整のアルゴリズムはごく単純で、子を持つセルの下に間隔をあけ、とにかく2つのセルの子が重なり合わなければよい、という方式です。その補助として変数totalHeightを使います。これを次に示す順序で実行しています。

1. 前半部分のループで生成したレイアウト属性を、ローカル辞書から取得します。これをもとに、セルをどの枠に配置するか決めていくことになります。
2. adjustFramesOfChildrenAndConnectorsForClassAtIndexPath:メソッド（別に実装）で再帰的に、セルの子（およびその子孫）をすべて配置調整して、着目しているセルの枠に照応するようにします。
3. 調整が済んだ属性を辞書に書き戻した後、次の項目を配置するべき枠に合わせて、変数totalHeightの値を更新します。ここでカスタムプロコトルを利用していることに着目してください。このプロトコルを実装するオブジェクトには、numRowsForClassAndChildrenAtIndexPath:メソッドが必要です。各クラスの子の数に合わせて、当該クラスが占める行数を求めるメソッドです。
4. maxNumRowsプロパティ（後で木階層全体の描画に必要な大きさを求めるために必要）として、第0セクション全体の高さを設定します。列の中で最も高さがあるのは第0セクションであり、これは木を構成するすべての子について配置を調整した後の高さに当たります（今回の実装では、間隔を詰める処理はしない）。
5. 最後に、各セルの属性を格納した辞書を、ローカルのlayoutInformation辞書に、一意的な文字列識別子をキーとして挿入します。

以降のレイアウト処理では、セルの属性を取得するために、この文字列識別子を使うことになります。今回の例では、後で補助ビューを組み込む際にもこれが重要な役割を果たします。

木構成の表示に要する大きさを求める

レイアウトの準備処理に際して、最も高さがあるセクションの行数を変数maxNumRowsに求めました。次のステップでは、これをもとに、木構成全体の表示に要する大きさを求めます。リスト 6-6にcollectionViewContentSizeの実装例を示します。ここに現れる定数ITEM_WIDTHおよびITEM_HEIGHTは、おそらくアプリケーション全体に影響が及びます（たとえば、カスタムセルの実装において、ラベルの大きさを適切に決めるために必要）。

リスト 6-6　コンテンツ領域の大きさを求めるコード例

```
- (CGSize)collectionViewContentSize {

    CGFloat width = self.collectionView.numberOfSections * (ITEM_WIDTH +
self.insets.left + self.insets.right);
```

```
    CGFloat height = self.maxNumRows * (ITEM_HEIGHT + _insets.top + _insets.bottom);
    return CGSizeMake(width, height);
}
```

レイアウト属性を返す

レイアウト属性オブジェクトをすべて初期化し、キャッシュとして格納すれば、`layoutAttributesForElementsInRect:`メソッドの処理に必要なレイアウト情報はすべて揃ったことになります。これはレイアウト処理の第2のステップに相当し、`prepareLayout`メソッドとは違って必須です。引数として渡された矩形領域に含まれるビューの、レイアウト属性オブジェクトをすべて、配列の形で返すメソッドです。コレクションビューにに収容される項目数が数千個に及ぶけれども、渡された矩形内にはごく少数しかない、という状況であれば、レイアウト属性オブジェクトをあらかじめ初期化せず、実際に必要になってから容易する方がよいかも知れません。しかし今回の実装では、キャッシュを活用することにします。したがって`layoutAttributesForElementsInRect:`メソッドは、保存済みの属性オブジェクトすべてについて（矩形内にあるかどうか）調べ、配列にまとめて返せばよいことになります。

リスト 6-7に`layoutAttributesForElementsInRect:`メソッドの実装例を示します。メインの辞書`_layoutInformation`から、所定の型のビューに対応する辞書を取得し、この辞書に格納されたレイアウト属性オブジェクトについて、判定処理を行います。すなわち、当該属性オブジェクトが指定された矩形内にあるかどうか判定し、あれば配列に追加していって、最後にこの配列を返します。

リスト 6-7　辞書に格納されている属性オブジェクトのうち該当するものを収集するコード例

```
- (NSArray*)layoutAttributesForElementsInRect:(CGRect)rect {
    NSMutableArray *myAttributes [NSMutableArray
arrayWithCapacity:self.layoutInformation.count];
    for(NSString *key in self.layoutInformation){
        NSDictionary *attributesDict = [self.layoutInformation objectForKey:key];
        for(NSIndexPath *key in attributesDict){
            UICollectionViewLayoutAttributes *attributes =
            [attributesDict objectForKey:key];
            if(CGRectIntersectsRect(rect, attributes.frame)){
                [attributes addObject:attributes];
            }
        }
    }
```

```
    return myAttributes;
}
```

注意: `layoutAttributesForElementsInRect:`の実装コードで、あるレイアウト属性に対応するビューが可視かどうか、を参照することはありません。このメソッドに引数として渡される矩形が、可視である領域を表すものとは限らないことに注意してください。実装方法はどうあれ、可視であるビューの属性を返すよう求められている、と想定してはなりません。`layoutAttributesForElementsInRect:`メソッドについて詳しくは、“指定された矩形内に表示する項目のレイアウト属性を与える” （41 ページ）を参照してください。

要求に応じて個々の属性を返す

“要求に応じてレイアウト属性を返す” （42 ページ）で説明したように、レイアウト処理が終わったレイアウトオブジェクトは、コレクションビューに収容されている各項目について、要求に応じてレイアウト属性オブジェクトを返せるようになっていなければなりません。セル、補助ビュー、装飾ビューの3種類に対してメソッドが必要ですが、現時点ではセルしか使っていないので、`layoutAttributesForItemAtIndexPath:`メソッドを実装するだけで動作します。

リスト6-8にこのメソッドの実装例を示します。属性オブジェクトは既に辞書に格納されているので、インデックスパスをキーとして取得し、返すだけです。

リスト 6-8　指定された項目のレイアウト属性を返すコード例

```
- (UICollectionViewLayoutAttributes *)layoutAttributesForItemAtIndexPath:(NSIndexPath
 *)indexPath {
    return self.layoutInfo[@"MyCellKind"][indexPath];
}
```

図 6-4に、これまでに実装したコードを実行した様子を示します。セルをすべて配置し、親に照応するよう正しく調整してありますが、セルどうしを結ぶ線はまだありません。

図 6-4　現時点のレイアウトの様子

補助ビューを組み込む

現状でもクラス階層を反映して適切にセルを表示するようになっていますが、親子を結ぶ線がないため、あまり見やすくありません。子セルに向かう線を描画するために、カスタムビューを実装し、補助ビューとしてレイアウトに組み込みます。補助ビューの設計について詳しくは、“補助ビューを使って表示を改善する”（43 ページ）を参照してください。

リスト 6-9に、`prepareLayout`の実装コード中に追加して、補助ビューを組み込むコード例を示します。補助ビューの属性オブジェクトを生成するコードには、セルの場合と比べてひとつ違いがあります。すなわち、生成しようとする属性オブジェクトに対応する、補助ビューの種類を表す文字列識別子が必要なのです。これは、カスタムレイアウトに使えるセルは1種類に限るのに対し、補助ビューは2種類以上使えるためです。

リスト 6-9　補助ビューの属性オブジェクトを生成するコード例

```
// create another dictionary to specifically house the attributes for the
supplementary view

NSMutableDictionary *supplementaryInfo = [NSMutableDictionary dictionary];

…

// within the initial pass over the data, create a set of attributes for the
supplementary views as well

UICollectionViewLayoutAttributes *supplementaryAttributes =
[UICollectionViewLayoutAttributes
layoutAttributesForSupplementaryViewOfKind:@"ConnectionViewKind"
withIndexPath:indexPath];

[supplementaryInfo setObject: supplementaryAttributes forKey:indexPath];

…

// in the second pass over the data, set the frame for the supplementary views just
 as you did for the cells

UICollectionViewLayoutAttributes *supplementaryAttributes = [supplementaryInfo
objectForKey:indexPath];

supplementaryAttributes.frame = [self
frameForSupplementaryViewOfKind:@"ConnectionViewKind" AtIndexPath:indexPath];

[supplementaryInfo setObject:supplementaryAttributes ForKey:indexPath];

...
```

```
// before setting the instance version of _layoutInformation, insert the local
supplementaryInfo dictionary into the local layoutInformation dictionary

[layoutInformation setObject:supplementaryInfo forKey:@"ConnectionViewKind"];
```

補助ビューの処理コードは、セルに対するものとよく似ているので、`prepareLayout`メソッドに簡単に組み込めます。補助ビューに対しても、セルの場合と同様のキャッシュ機構を実装します。そのため、補助ビュー専用の*ConnectionViewKind*という辞書を用意します。2種類以上の補助ビューを使う場合は、種類ごとに辞書を生成し、同様のコードを実装するとよいでしょう。しかし今回の場合、補助ビューは1種類だけです。セルのレイアウト属性を初期化する場合と同様に、ビューの種類にもとづき補助ビューを配置する枠を求める、`frameForSupplementaryViewOfKind:AtIndexPath:`メソッドを実装します。`prepareLayout`メソッドの実装中に現れる`adjustFramesOfChildrenAndConnectorsForClassAtIndexPath:`は、クラス階層図のレイアウトに関わる、補助ビューの配置も調整します。

もっとも、今回の場合、`layoutAttributesForElementsInRect:`の実装を修正する必要はありません。もともと、メインの辞書に格納された属性すべてについて反復するようになっているからです。補助ビューの属性をメインの辞書に追加しさえすれば、`layoutAttributesForElementsInRect:`はそのままで想定どおりに機能します。

最後に、セルの場合と同様に、コレクションビューはいつでも、補助ビューを指定してその属性を要求することがありえます。したがって`layoutAttributesForSupplementaryElementOfKind:atIndexPath:`の実装は必須です。

リスト6-10にその実装例を示しますが、これは`layoutAttributesForItemAtIndexPath:`とよく似ています。ただし、`kind`文字列は即値として記述する代わりに引数として渡すようになっているので、2種類以上の補助ビューが使えることになります。

リスト 6-10　要求に応じて補助ビューの属性を返すコード例

```
- (UICollectionViewLayoutAttributes *)
layoutAttributesForSupplementaryViewOfKind:(NSString *)kind atIndexPath:(NSIndexPath
 *)indexPath {

    return self.layoutInfo[kind][indexPath];

}
```

まとめ

補助ビューを組み込むことにより、クラス階層図を適切に描画するレイアウトオブジェクトを実装できました。これをもとに、できるだけ空間を有効利用するよう改良してみるのもよいでしょう。ここでは、コレクションビューのカスタムレイアウトをどのように実装するか、現実的な例を使って示しました。コレクションビューは非常に協力で、さまざまな能力を秘めています。セルを移動/挿入/削除したとき、強調や選択表示（あるいはアニメーション表示）するよう改良するのも非常に容易です。さらにレイアウトを改善する際には、“カスタムレイアウトの作成”（37 ページ）の後半の数節も参考になるでしょう。

書類の改訂履歴

この表は「*Collection View プログラミングガイド（iOS用）*」の改訂履歴です。

日付	メモ
2014-07-15	誤記を修正しました。
2013-09-18	カスタムレイアウトに関する説明を大幅に改訂し、技術的な誤りを修正しました。カスタムレイアウトの実装例を説明する章を新たに追加しました。iOS 7の新機能を取り入れて改訂しました。
2012-09-19	iOSアプリケーションにコレクションビューを組み込む方法について説明した新規ドキュメント。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

Apple Japan
〒106-6140 東京都港区六本木
6丁目10番1号 六本木ヒルズ
http://www.apple.com/jp/

Offline copy. Trademarks go here.