iOS View Controller プログラミングガイド

Developer

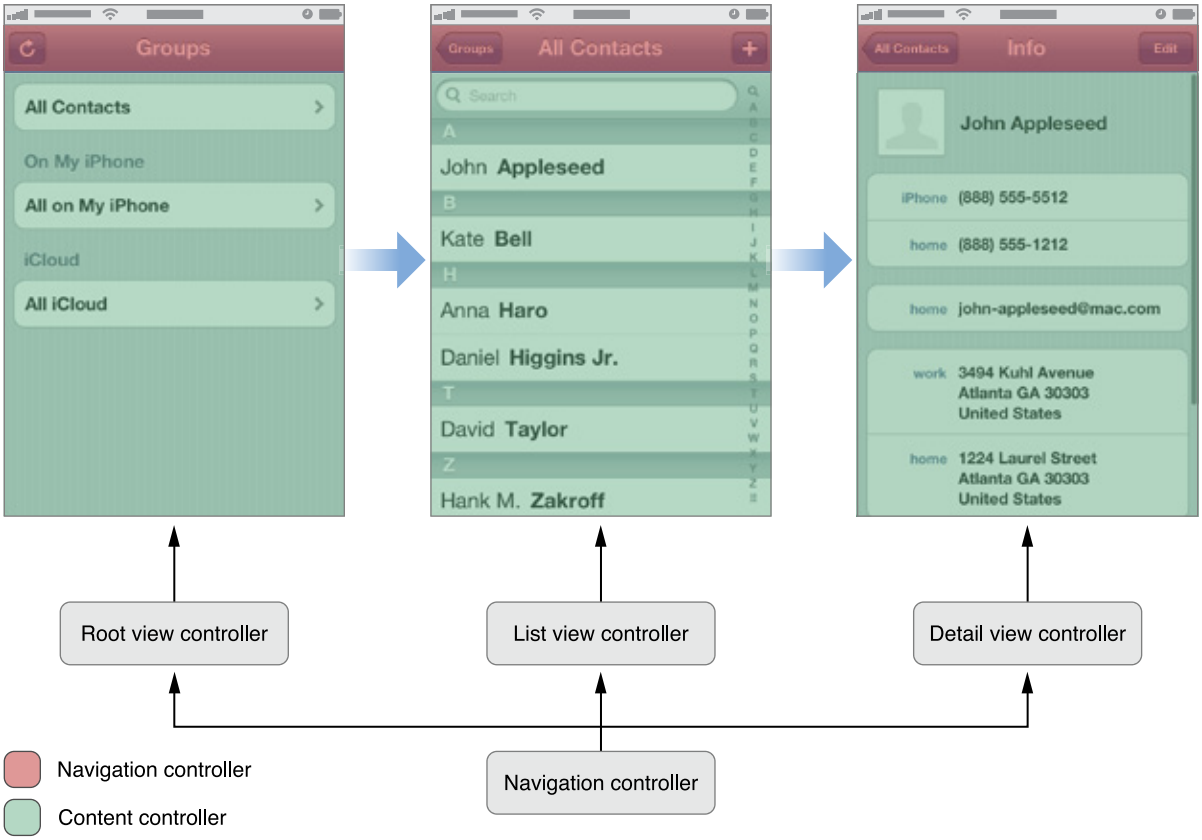
目次

図、表、リスト

View Controllerの概要

View Controllerはアプリケーションのデータとその外観を結びつける重要な役割を果たします。iOSアプリケーションがユーザインターフェイスを表示する際、その内容はView Controllerが単独で、またはそのグループが連携して管理します。したがってView Controllerは、アプリケーションを構成する骨組みと言ってよいでしょう。

iOSには、ナビゲーションバー、タブバーなど、標準的なユーザインターフェイス要素を実装する、View Controllerのさまざまなクラスが組み込まれています。実際の開発においては、アプリケーション特有のコンテンツを表示する、カスタムコントローラもいくつか実装することになるでしょう。



初めに

View Controllerは、Model-View-Controller（MVC）設計パターンにおける従来のControllerオブジェクトに相当しますが、それ以上の機能があります。View Controllerは、あらゆるiOSアプリケーションに見られる、さまざまな動作を実現します。その多くは基底クラスに実装されています。基底クラスでは一部のみ実装されており、派生クラスに残りを実装して補わなければならない動作もあります。たとえばユーザがデバイスを回転したとき、標準の実装ではユーザインターフェイスを回転しようとします。しかし派生クラスでは、回転という動作が適切かどうか、向きに応じてどのように各ビューの並べ方を変えるか、などを判断する処理が加わる場合があります。そこで、基底クラスを適切に構造化しておき、フックの形で派生クラスに固有の処理を実装する、という仕組みで、プラットフォーム設計ガイドラインに従いながら、アプリケーションの動作を容易にカスタマイズできるようにしてあります。

View Controllerはビュー群を管理する

View Controllerはそれぞれ、アプリケーションのユーザインターフェイスを分担して管理します。要求に応じて、画面に表示したり、ユーザと対話したりできる、「ビュー」を生成します。多くの場合これは、さまざまなビュー（ボタン、スイッチなど、iOSに組み込みのユーザインターフェイス要素）を複雑に階層化して組み合わせたルートビューです。View Controllerは、ビュー階層に代わって中心的な調整役として機能し、ビューとコントローラまたはデータオブジェクトの間のやり取りに対処します。

関連する章: “View Controllerの基礎” （14 ページ）

コンテンツはContent View Controllerで管理する

アプリケーション特有のコンテンツを表示するためには、Content View Controllerを実装する必要があります。このView Controllerクラスは、`UIViewController`または`UITableViewController`のサブクラスとして実装し、コンテンツの表示や管理に必要なメソッドを追加します。

関連する章: “カスタムContent View Controllerの作成” （46 ページ）

Container View Controllerはほかの View Controller群を管理する

Container View Controllerは、ほかのView Controllerが所有するコンテンツを表示します。ここで言うほかのView Controllerは、Containerと明示的に親子関係を結びます。Container View ControllerとContent View Contollerが組み合わされてView Contollerオブジェクトの階層を成し、その頂上に唯一あるのがルートView Controllerです。

Containerの型によって、子を管理するインターフェイスは異なります。たとえば、子と子の間に特別な「ナビゲーション」関係を定義するメソッドがあるかも知れません。あるいは、特定の型のView Controllerしか子にできないよう、何らかの制限を設けていることもあります。さらに、子にしたView Controllerに対し、Containerの設定に用いる追加のコンテンツを要求することもありえます。

iOSにはさまざまな型のContainer View Controllerが組み込まれているので、独自のユーザインターフェイスを構築する際に利用してください。

関連する章: "View Controllerの基礎" （14 ページ）

View Controllerを表示すると、画面には一時的にそのビューが現れる

View Controllerがユーザに対し、追加の情報を表示する必要が生じることがあります。あるいは、追加情報を入力したり、あるタスクを実行したりするよう、ユーザに求めることもあるでしょう。しかしiOSデバイスは、画面の大きさに制約があり、ユーザインターフェイス要素を同時にすべて表示することはできません。そこでiOSアプリケーションは、一時的に別のビューを表示して、ユーザと対話できるようにします。要求されたアクションをユーザが実行し終えると、このビューは消えてしまいます。

このようなインターフェイスの実装に要する手間を省くため、iOSでは、View ContollerがほかのView Controllerのコンテンツを表示できるようになっています。被表示（表示された側）View Controllerは、画面のある部分（多くの場合画面全体）にビューを描画します。その後、ユーザがタスクを終えると、被表示View Controllerは、表示した側のView Controllerに対し、タスクが終了した旨を通知します。表示した側のView Controllerが被表示View Controllerを消去し、画面は元の状態に戻ります。

一般にView Controllerは、ほかのView Controllerから「表示され」うることを考慮して設計しなければなりません。

関連する章: "View ControllerをほかのView Controllerから「表示」" （95 ページ）

ストーリーボードはユーザインターフェイス要素をアプリケーションインターフェイスと関連付ける

ユーザインターフェイスの設計は、非常に複雑になりえます。各View Controllerは、さまざまなのビューやGesture Recognizer、その他のユーザインターフェイスオブジェクトを参照します。逆にこういったオブジェクトも、View Controllerを参照し、あるいはユーザのアクションに応答して適当なコード片を実行します。View Controllerが単独で動作することはめったにありません。多数のView Controllerの連携も、アプリケーションに定義される「関係」のひとつです。ユーザインターフェイスの作成とは、簡単に言うと、数多くのオブジェクトを生成、設定し、相互の関係を確立する作業であって、非常に時間がかかり、誤りも容易に入りってしまいます。

代わりに、Interface Builder上で**ストーリーボード**を作る、という方法があります。ストーリーボードには、View Controllerやその関連オブジェクトのインスタンスを、設定済みの状態で保存します。各オブジェクトの属性や、オブジェクト間の関係は、Interface Builderで設定できます。

アプリケーションは実行時にストーリーボードをロードし、これに従ってアプリケーションのインターフェイスを作り出します。ストーリーボードからロードされたオブジェクトは、あらかじめ設定しておいた通りの状態になります。Interface Builderで直接設定できない動作については、UIKitを使って実装することも可能です。

ストーリーボードを使えば、アプリケーションのユーザインターフェイスに現れるオブジェクト同士が、適切に配置されているかどうかを容易に確認できます。また、ユーザインターフェイスオブジェクトを生成、設定するためのコードが少なくて済む、という利点もあります。

関連する章: "View Controllerの基礎" （14 ページ）、"アプリケーションにおけるView Controllerの使い方" （37 ページ）

この文書の使い方

まず"View Controllerの基礎" （14 ページ）を読んでください。ここでは、View Controllerがどのようにしてアプリケーションのインターフェイスを実現するか、を解説しています。次に"アプリケーションにおけるView Controllerの使い方" （37 ページ）を読んで、View Controller（iOSに組み込みのもの、独自に実装したものの両方）の使い方を理解してください。

アプリケーション独自のControllerを実装する必要が生じたら、View Controllerが実行するタスクの概要について"カスタムContent View Controllerの作成" （46 ページ）を読んだ後、それ以降の各章で、具体的な実装方法を調べるとよいでしょう。

必要事項

この文書の前に、『*Start Developing iOS Apps Today*』および『*Your Second iOS App: Storyboards*』の内容を充分に把握しておいてください。ストーリーボードのチュートリアルには、次に挙げるCocoaの考え方をはじめ、本書で説明するさまざまな技法の使用例が載っています。

- 新規Objective-Cクラスの定義の仕方
- アプリケーションの動作を管理する上でのデリゲートオブジェクトの役割
- Model-View-Controller/パラダイム

関連項目

UIKitに付属する標準的なContainer View Controllerについては、『*View Controller Catalog for iOS*』を参照してください。

View Controllerでビューを操作する方法については、『*View Programming Guide for iOS*』を参照してください。

View Controllerでイベントを処理する方法については、『*Event Handling Guide for iOS*』を参照してください。

iOSアプリケーションの全体的な構成については、『*iOS App Programming Guide*』を参照してください。

プロジェクトでストーリーボードを設定する方法については、『*Xcode User Guide*』を参照してください。

View Controllerの基礎

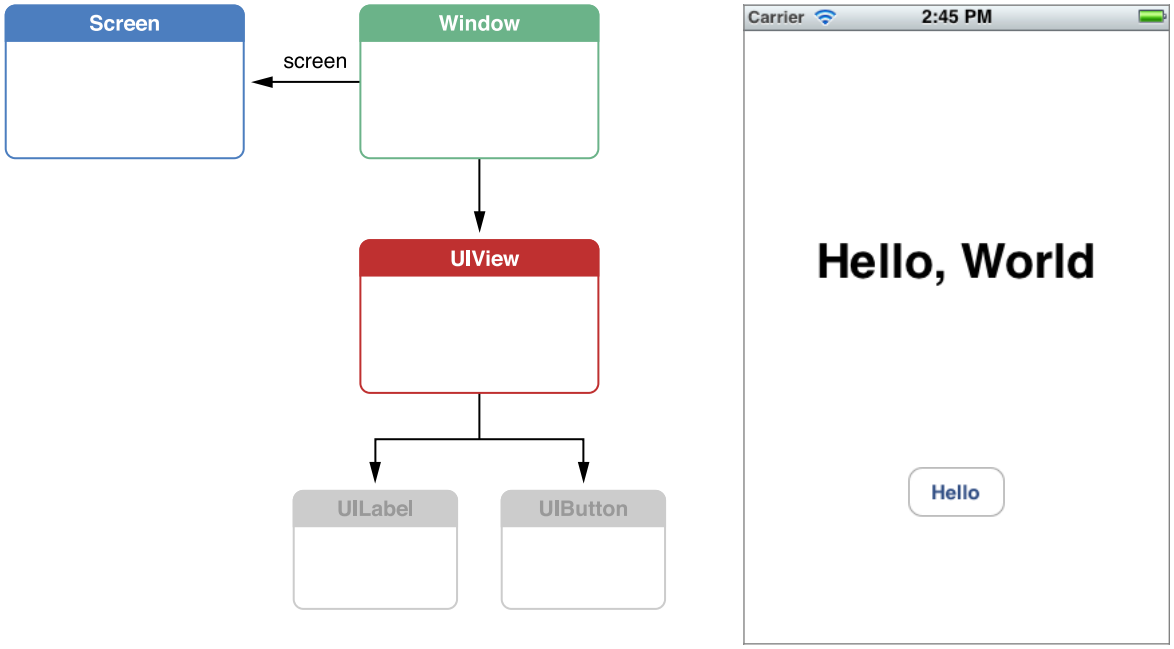
iOSベースのデバイスで動作するアプリケーションがコンテンツを表示するために使用できる画面領域は限られているため、ユーザに情報を提示する方法を工夫する必要があります。多数の情報を表示するアプリケーションは、当初は一部だけを表示し、ユーザの操作に応じて残り部分の表示/非表示を切り替えるようにしなければなりません。View Controllerオブジェクトは、コンテンツを管理したり、その表示および非表示を調整したりする基盤機能を提供します。ある独立した部分のユーザインターフェイスの制御機能を別のView Controllerクラスに移すことにより、実装に関する複雑な問題を、小さくて管理しやすい単位に分割できます。

アプリケーションでView Controllerを利用するためには、iOSアプリケーション上にコンテンツを表示するために用いる、ウインドウやビューなど主なクラスについて理解しておく必要があります。View Controllerの実装において鍵になるのは、コンテンツの表示に用いるビューの管理です。しかし、ビューの管理だけがView Controllerの仕事ではありません。View Controllerの多くは、画面遷移の際に、ほかのView Controllerと通信し、必要な調整を行います。View Controllerは多数の「接続」（内部的にはビューや関連オブジェクトとの接続、外部的にはほかのコントローラとの接続）を管理するので、オブジェクト間の接続は、なかなかきちんと把握できないかも知れません。そこで、Interface Builderを使ってストーリーボードを作成する方法をお勧めします。この方法によれば、アプリケーションに内在する「関係」を視覚的に把握できるとともに、実行時にオブジェクトを初期化するための処理を大幅に簡略化できます。

画面、ウインドウ、ビューが視覚的なインターフェイスを構成する

図1-1にごく単純なインターフェイスを示します。左側は、このインターフェイスを構成するオブジェクトと、その接続の様子です。

図 1-1 ウインドウとその描画対象である画面、およびContent View

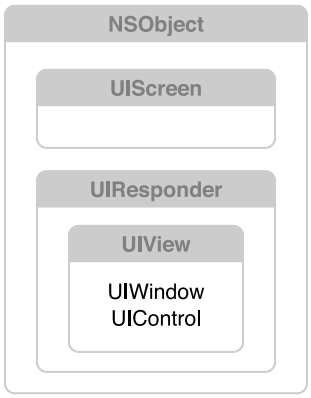


ここには大きく分けて3種類のオブジェクトがあります。

- `UIScreen`オブジェクト：デバイスに接続された物理画面を表します。
- `UIWindow`オブジェクト：この画面に対する描画機能を提供します。
- `UIView`オブジェクト群：実際に描画を行います。ウインドウに関連付けられており、ウインドウからの要求に応じてコンテンツを描画します。

図 1-2に、上記のクラス（および関連する主なクラス）が、UIKitではどのように定義されているかを示します。

図 1-2 ビューシステムに属するクラス群



ビューについて詳しく知らなくてもView Controllerを理解する上で支障はありませんが、ここではビューのもっとも重要な機能を簡単に説明しましょう。

- ビューはユーザインターフェイス要素を表します。各ビューは画面上のある領域を占めます。この領域内にコンテンツを表示したり、ユーザイベントに応答したりします。
- ビューを入れ子にして「ビュー階層」を構成できます。サブビューに描画する際の位置は、スーパービューを基準として決まります。すなわち、スーパービューが移動すれば、サブビューもこれと一緒に移動します。ビュー階層には、一群のビューを共通のスーパービューの下に関連付けて、管理しやすくする働きがあります。
- ビューはプロパティ値をアニメーション化できます。「アニメーション化」とは、ある時間をかけて徐々にプロパティ値を変化させることです。複数のビューにまたがり、複数のプロパティをうまく調整しながら変化させれば、画面上には統一された動きのアニメーション表示が現れます。

  iOSアプリケーションの開発において、アニメーションは重要な役割を果たします。多くのアプリケーションは、コンテンツをすべて一度に表示することはできません。そこで、いつ遷移が発生したか、新しいコンテンツがどこから来たか、がユーザにわかるよう、アニメーションを使います。一瞬で画面が切り替わってしまうと、ユーザを混乱させることになりかねません。
- ビューそのものは、アプリケーションにおいて自身が果たす役割を知りません。たとえば図 1-1には、「*コントロール*」とも呼ばれる特殊なビューである、「ボタン」（表示は「Hello」）が現れています。コントロールは、ユーザが当該領域で何か操作をしたとき、どのように応答すればよいかを知っていますが、何を制御するかは知りません。代わりに、ユーザがコントロールを操作したとき、アプリケーション内のほかのオブジェクトにメッセージを送るようになっていま

す。このような構成は適応性に優れ、単一のクラス（この場合は`UIButton`）でさまざまなボタンを実装できます。個々のボタンは、それぞれに応じたアクションを起動するよう設定すればよいのです。

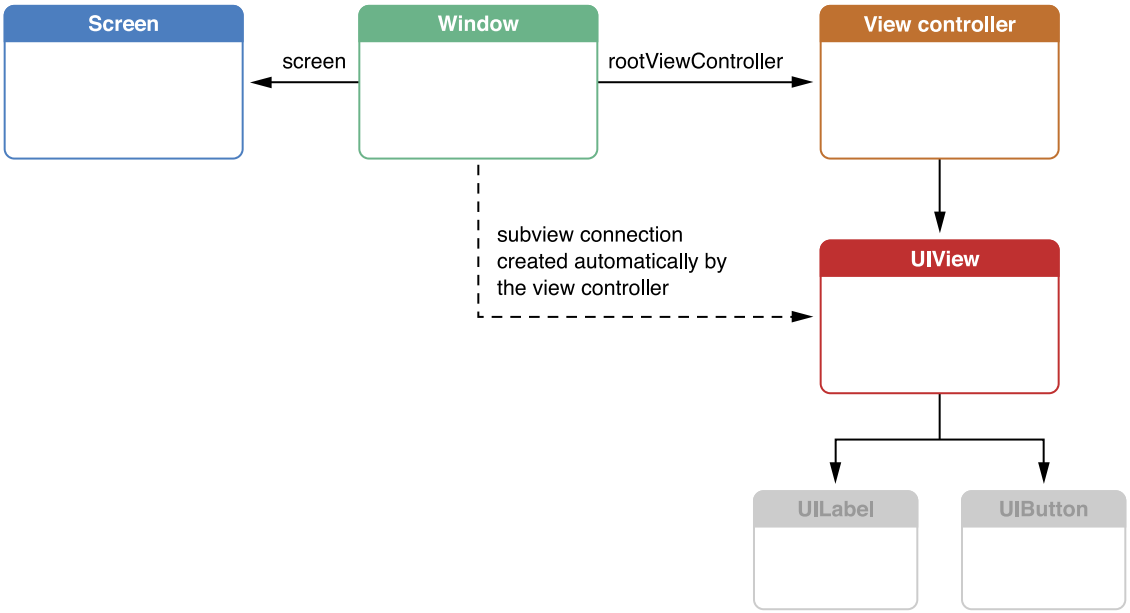
複雑なアプリケーションでは多くのビューが必要になり、多くの場合、これをビュー階層の形に編成します。その一部をアニメーション化して、画面の中へ、あるいは外に向かって動かすとともに、全体がひとつの大きなインターフェイスであるように見せかけます。さらに、ビュークラスの再利用性を高めるため、アプリケーションでそれぞれが果たす具体的な役割を、ビュークラス自身は知らないでよいようにしています。したがって、アプリケーションロジック（頭脳部分）はどこか別の場所に置かなければなりません。View Controllerが「頭脳」となって、アプリケーションのビューを相互に結びつけます。

View Controllerはビューを管理する

それぞれのView Controllerは、あるビューを編成、制御します。多くの場合、これはビュー階層のルートにあたるビューです。View ControllerはMVCパターンにおけるコントローラオブジェクトに当たりますが、iOSの場合、ほかにも特別なタスクを担当します。このタスクは`UIViewController`クラスに定義されており、View Controllerすべてがこれを継承します。View Controllerはすべて、ビューやリソースの管理タスクを担います。それ以外の機能は、View Controllerの用途によって違います。

図 1-3に、図 1-1と同様の、しかしView Controllerを使ったインターフェイスを示します。ビューをウインドウに直接割り当てることはしません。代わりにView Controllerをウインドウに割り当てます。View Controllerは自動的に、ビューをウインドウに追加します。

図 1-3 ウインドウに関連付けられたView Controllerは、自動的にそのビューを、ウインドウのサブビューとして追加する



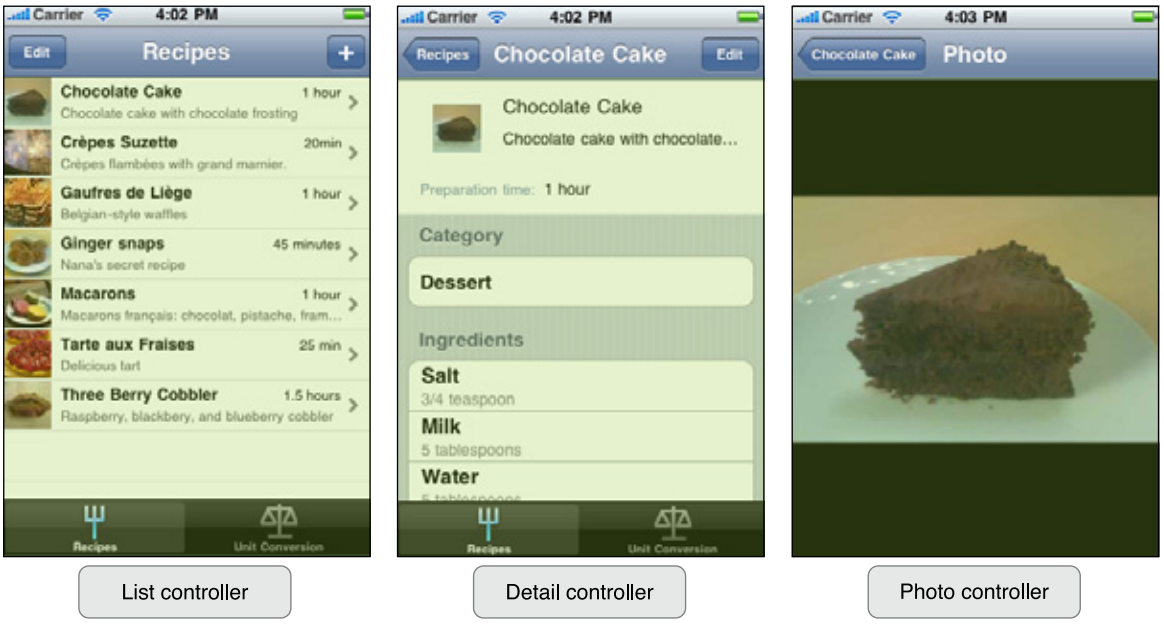
View Controllerはビューを、必要になった時点でロードするようになっています。また、ある条件下でビューを解放することも可能です。したがってView Controllerは、アプリケーションのリソース管理においても重要な役割を果たしているのです。

View Controllerは、接続されたビューのアクションを調整する司令塔としても理にかなっています。たとえばボタンが押されると、ボタンはメッセージをView Controllerに送信します。ビューそのものはどのようなタスクを実行するか知りませんが、View Controllerは、ボタンの押下がどのような意味を持ち、どのように応答するべきかを知っていなければなりません。たとえば、データオブジェクトを変更する、ビューに格納されたプロパティ値を変更する（徐々に変化させてアニメーション化することも可）、あるいはほかのView Controllerのコンテンツ画面に切り替える、といった応答が考えられます。

通常、View Controllerのインスタンスそれぞれは、アプリケーションデータの一部しか扱いません。所定のデータ群についてはどのように表示するかを知っていますが、それ以外のデータについて考慮する必要はないのです。このように、アプリケーションのデータモデル、ユーザインターフェイス設計、View Controllerはすべて、相互に関連し合っています。

図 1-4に、レシピを管理するアプリケーションの例を示します。このアプリケーションは、互いに関連し合っているけれども別々の、3つのビューを表示します。最初のビューには、このアプリケーションが管理するレシピのリストが表示されています。レシピをタップすると、その詳細を記述した第2のビューが現れます。詳細ビューの写真をタップすると、写真を拡大表示する第3のビューに切り替わります。各ビューを管理するのは、別々のView Controllerです。それぞれが、対応するビューを表示したり、サブビューにデータを取り込んだり、ビュー階層をたどってユーザの操作に応答したりします。

図 1-4 個々のView Controllerが管理する3つのビュー



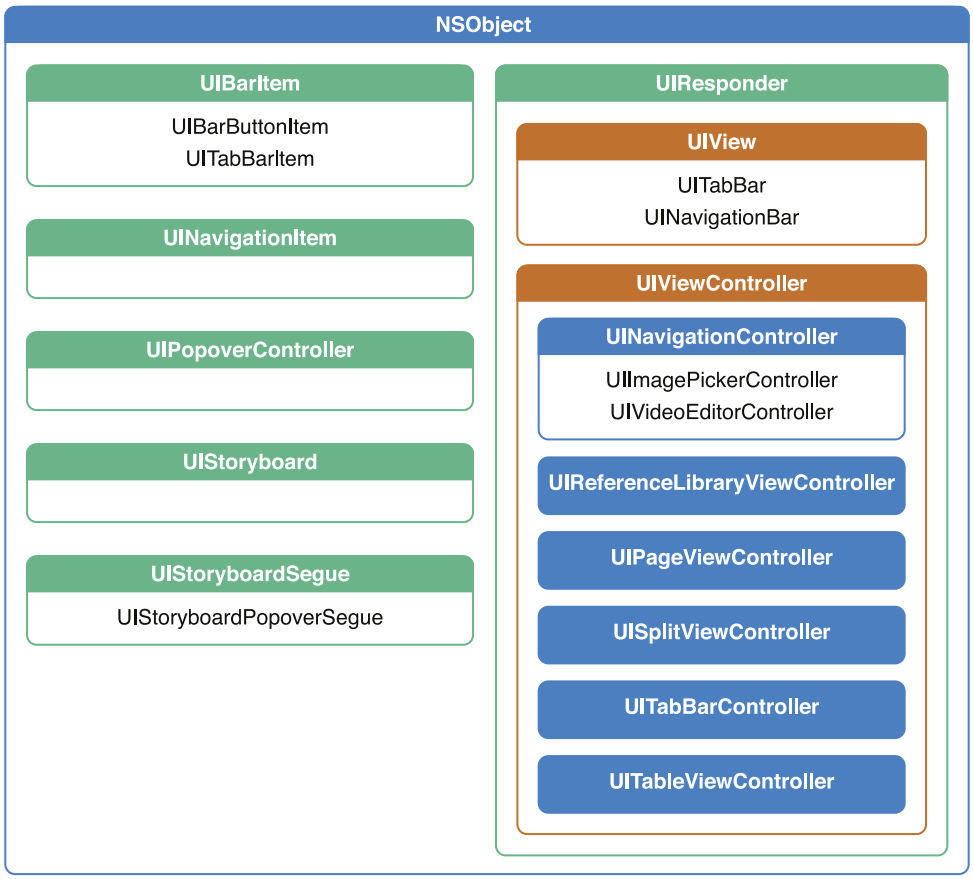
この例にはいくつか、あらゆるView Controllerに共通の性質が現れています。

- ビューはいずれも、単一のView Controllerに管理されます。View Controllerの`view`プロパティにビューを割り当てれば、このView Controllerがビューを所有することになります。ビューが別のビューの子であるときも、親と同じView Controllerが制御する場合と、別に子を制御するView Controllerがある場合があります。複数のView Controllerを組み合わせて単一のビュー階層にする方法については、Container View Controllerに関する節で説明します。
- 各View Controllerは、アプリケーションデータの一部のみ扱います。たとえば「Photo」コントローラは、どの写真を表示するか、を知っているだけで十分です。
- View Controllerがそれぞれ提供するのはユーザの目に映るものの一部分に過ぎないので、相互に通信し、全体としてシームレスにつながるようにしなければなりません。また、データコントローラやドキュメントオブジェクトなど、ほかのコントローラとも通信することがあります。

View Controllerの分類

図1-5に、UIKitフレームワークで利用できるView Controllerクラスと、View Controllerと組み合わせて使う主なクラスを示します。たとえば、`UITabBarController`オブジェクトは、Tab Barインターフェイスに関連付けられたタブを実際に表示する`UITabBar`オブジェクトを管理します。ほかのフレームワークには、この図に載っている以外にもView Controllerクラスが定義されています。

図 1-5 UIKitのView Controllerクラス



View Controllerは、iOSに組み込みのもの、独自に定義するものともに、アプリケーションにおいて果たす役割によって、Content View ControllerとContainer View Controllerの2種類に大きく分類できます。

Content View Controllerはコンテンツを表示する

Content View Controllerは、ビュー、あるいはビュー階層として編成されたビュー群を使って、画面上にコンテンツを表示します。これまでに説明してきたコントローラも、このContent View Controllerに属します。アプリケーションにおいてコントローラが果たす役割に応じ、データのある一部について扱い方を知っているのが普通です。

アプリケーションでは一般に、次のような目的でContent View Controllerを使います。

- ユーザに向けてデータを表示するため
- ユーザに入力させてデータを収集するため
- 所定のタスクを実行するため
- 実行/選択可能なコマンド/オプションを示してナビゲートするため（ゲームの起動画面など）

Content View Controllerは、アプリケーション処理の主たる司令塔オブジェクトと言えます。アプリケーションが提供するデータやタスクのある部分について、詳細を知っているからです。

Content View Controllerオブジェクトは、ビューのすべてを1つのビュー階層で管理する責任を負います。View Controllerとそのビュー階層のビューを1対1で対応させることは、設計上重要な検討項目です。同じビュー階層を管理するために、複数のContent View Controllerを使用するべきではありません。同様に、複数の画面に相当するコンテンツを管理するために、単独のContent View Controllerを使用するべきではありません。

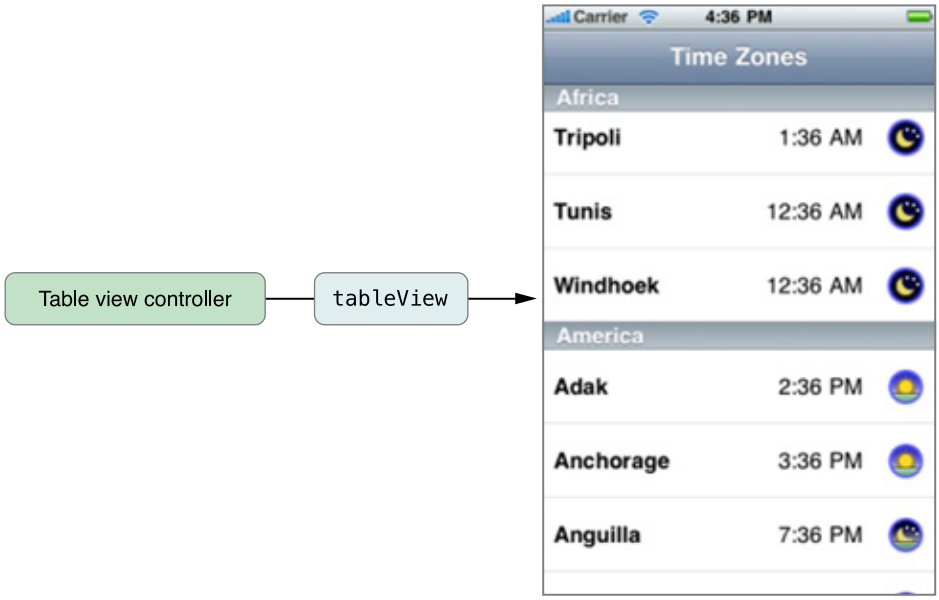
Content View Controllerを定義し、その動作を実装する方法については、“カスタムContent View Controllerの作成”（46 ページ）を参照してください。

Table View Controllerの概要

表形式でデータを表示するアプリケーションは少なくありません。そこでiOSでは、表データの管理に特化した、`UIViewController`の組み込みサブクラスを用意しています。この`UITableViewController`クラスはテーブルビューを管理し、選択の管理、行の編集、テーブルの設定などの標準的なテーブル関連動作に対するサポート機能を追加します。この追加サポート機能によって、テーブルベースのインターフェイスを作成して初期化するために記述しなければならないコードの量を最小限に抑えることができます。また、`UITableViewController`のサブクラスを作成して、ほかのカスタム動作を追加することもできます。

図 1-6に、Table View Controllerの使用例を示します。このTable View ControllerはUIViewControllerのサブクラスであるため、（viewプロパティに）このインターフェイスのルートビューへのポインタを持っていますが、このインターフェイスに表示されるTable Viewを指す別のポインタも持っています。

図 1-6 テーブル形式のデータの管理



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。

Container View ControllerはほかのView Controllerが提供するコンテンツの配置を調整する

Container View ControllerにはほかのView Controllerのコンテンツが包含されます。ここに言うほかのView Controllerは、Container View Controllerの子として、明示的に対応づけなければなりません。Container Controllerは、ほかのコントローラの親になること、ほかのコンテナの子になることの両方が可能です。こうしてコントローラ同士が組み合わされ、最終的にView Controller階層を形成します。

Container View Controllerの種類によって、その子を取り囲むユーザインターフェイスは異なります。実際、その外観も、子に対する設計上の制約も、コンテナの種類によってまったく違うのです。各種のContainer View Controllerは、たとえば次のような点に違いがあります。

- コンテナは子を管理するための独自のAPIを備えています。
- コンテナは、子同士が関係を持てるか、持てる場合はどのような関係か、を決めることができます。

- コンテナはビュー階層を管理します（コンテナではないView Controllerと同様）。また、どの子のビューでも、ビュー階層に付け加えることができます。ビューをいつ追加できるか、コンテナのビュー階層に合わせてどのように大きさを調整するか、を決めるのはコンテナ自身ですが、ビューやサブビューについて責任を負うのは、子側のビューコントローラです。
- コンテナは子の設計に何らかの制約を負わせることがあります。たとえば、特定のView Controllerクラスしか子にしないよう制限する、コンテナのビューを設定するために必要な追加のコンテンツを提供するよう、子のコントローラに要求する、などといった制約です。

組み込みコンテナクラスはいずれも、ユーザインターフェイスに関する重要な原則に基づいて実装されています。こういったコンテナで管理されるユーザインターフェイスを組み合わせて、複雑なアプリケーションを構成することになります。

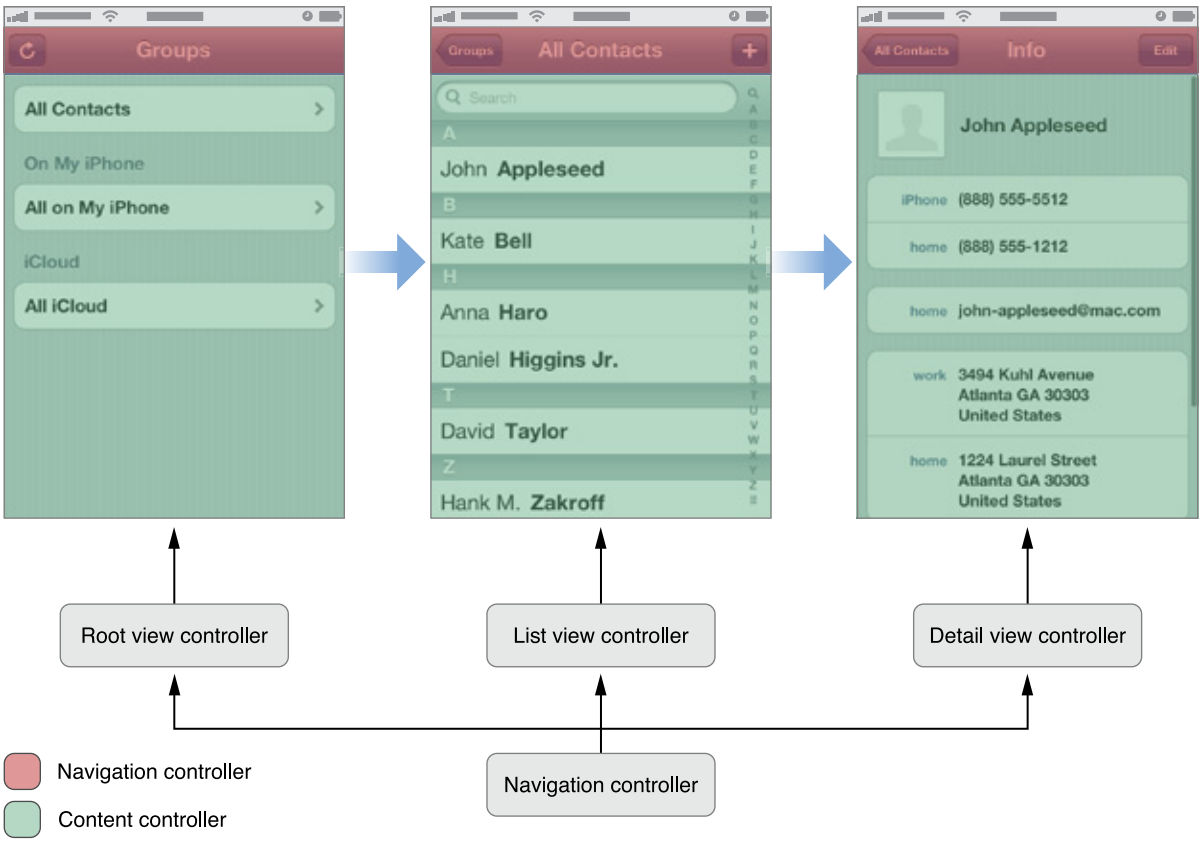
Navigation Controllerの概要

Navigation Controllerは階層的に編成されたデータを表し、`UINavigationController`クラスのインスタンスです。このクラスのメソッドは、Content View Controllerのコレクションをスタックベースで管理する機能をサポートしています。このスタックは、階層的なデータの中をユーザが通った経路を反映します。スタックの一番下は出発点を表し、スタックの一番上はデータ内のユーザの現在位置を表します。

図 1-7に、「連絡先(Contacts)」アプリケーションの画面を示します。このアプリケーションでは、Navigation Controllerを使用して連絡先情報をユーザに表示します。各ページの上部にあるナビゲーションバーは、Navigation Controllerが所有します。ユーザに対して表示される、画面の残り部分は、Content View Controllerによって管理されています。このContent View Controllerはデータ階層の特定の

レベルにある情報を表示します。ユーザがインターフェイス内のコントロールを操作すると、これらのコントロールは、Navigation Controllerに、次のView Controllerを表示したり、現在のView Controllerを閉じたりする指示を伝えます。

図 1-7 階層データのナビゲーション



Navigation Controllerの主な仕事は、子であるView Controllerを管理することですが、いくつかのビューの管理も行います。具体的に言うと、Navigation Controllerは、Navigation Bar（データ階層におけるユーザの現在位置情報を表示）、ボタン（前の画面に戻る）、その他現在のView Controllerに必要なカスタムコントロールを管理します。View Controllerが所有するビューを直接修正することはできません。代わりに、Navigation Controllerが表示するコントロールを、子View Controllerそれぞれのプロパティで設定します。

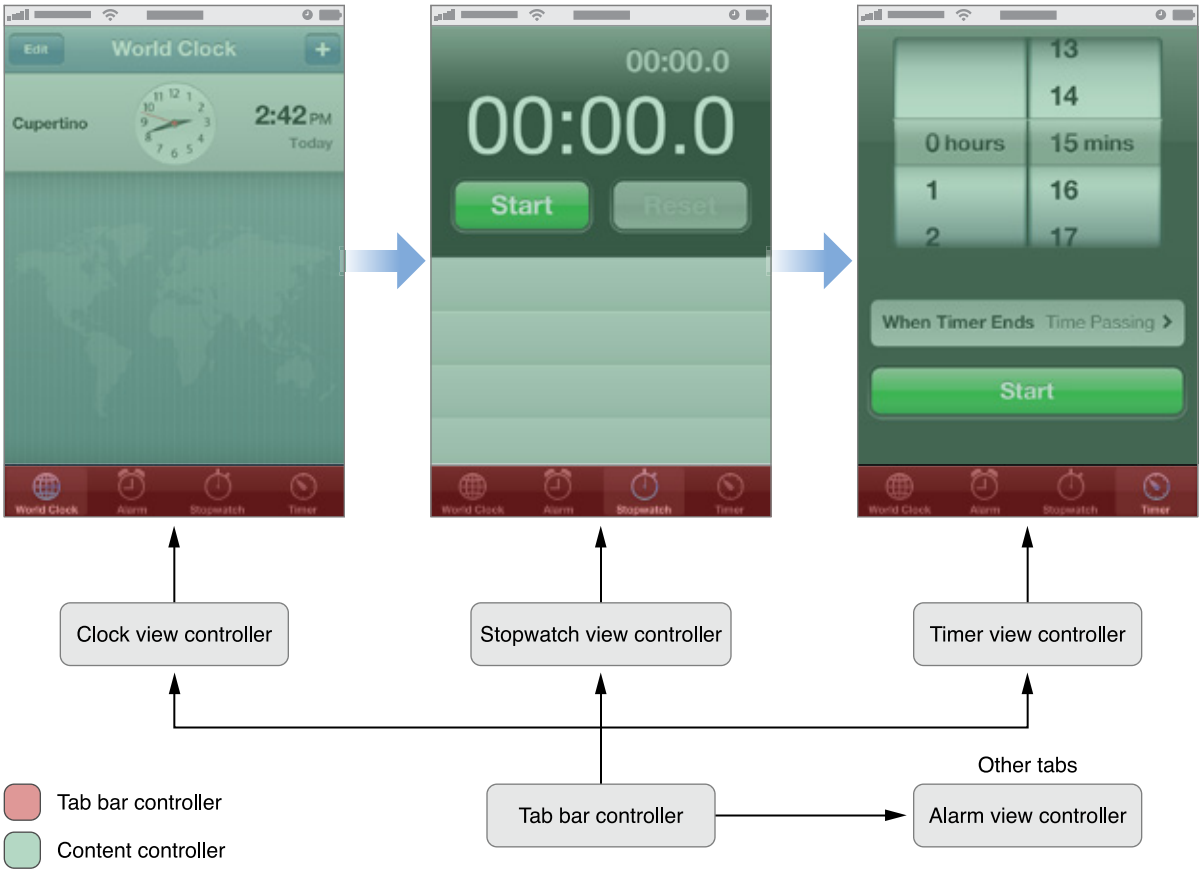
Navigation Controllerオブジェクトを設定したり使用したりする方法については、「Navigation Controller」を参照してください。

Tab Bar Controllerの概要

Tab Bar Controllerは、アプリケーションを2つ以上の操作モードに分割するために使用するコンテナView Controllerです。Tab Bar Controllerは`UITabBarController`クラスのインスタンスです。Tab Barはいくつかのタブから成り、それぞれの中身は子View Controllerが表示します。タブを選択すると、Tab Bar Controllerはそれに対応するView Controllerのビューを画面に表示します。

図 1-8に、「時計(Clock)」アプリケーションのいくつかのモードと、それに対応するView Controllerとの関係を示します。各モードには、メインコンテンツ領域を管理するContent View Controllerがあります。「時計(Clock)」アプリケーションの場合、「Clock」View Controllerと「Alarm」View Controllerは共に、画面の上端に追加コントロールを持つナビゲーションスタイルのインターフェイスを表示します。その他のモードでは、Content View Controllerを使用して、単独の画面を表示します。

図 1-8　「時計(Clock)」アプリケーションのさまざまなモード



アプリケーションで異なる種類のデータを表示するときや、同じデータを異なる方法で表示するときに、Tab Bar Controllerを使用します。

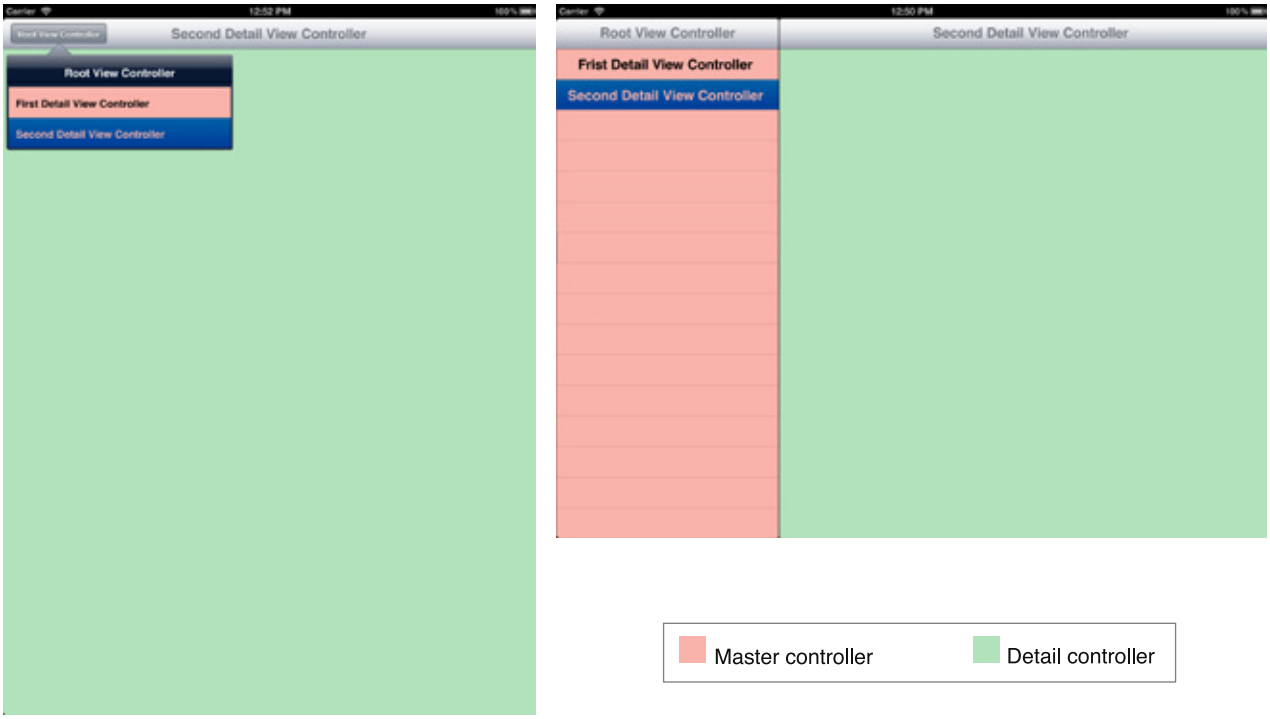
Tab Bar Controllerを設定したり使用したりする方法については、「Tab Bar Controller」を参照してください。

Split View Controllerについて

Split View Controllerは画面をいくつかの部分に分割し、それぞれ独立に更新できるようにします。外観は向きによっても変わることがあります。Split View Controllerは`UISplitViewController`クラスのインスタンスです。Split Viewインターフェイスのコンテンツは、2つの子View Controllerから取得されます。

図 1-9に、『*MultipleDetailViews*』のサンプルアプリケーションで使われているSplit Viewインターフェイスを示します。縦長モードの場合、詳細ビューしか表示されません。リストビューはPopoverを使って表示します。一方、横長モードの場合は、両方の子を横に並べて表示します。

図 1-9　縦長モードと横長モードにおける、マスタと詳細が対になったインターフェイス



Split View ControllerはiPadでのみサポートされており、より大きなiPad画面での利用を目的としています。iPadアプリケーションでは、マスタと詳細が対になったインターフェイスを実装する方法として推奨されています。

Split View Controllerを設定したり使用したりする方法については、「Popover」を参照してください。

Popover Controllerの概要

図 1-9をもう一度見てみましょう。Split View Controllerが縦長モードで表示されている場合、マスタビューは*Popover*という特別なコントロール内に表示されます。iPadアプリケーションでは、Popover Controller（`UIPopoverController`）を使ってPopoverを実装できます。

Popover Controllerは、実際にはコンテナではありません。`UIViewController`を継承してはいないからです。しかし実践上はコンテナによく似ているので、使う際には同じプログラミング原理を適用できます。

Popover Controllerを設定、使用する方法については、「Popover」を参照してください。

Page View Controllerの概要

Page View Controllerは、ページレイアウトを実装するためのContainer View Controllerです。書籍のようにページをめくって表示を切り替えることができます。Page View Controllerは`UIPageViewController`クラスのインスタンスです。各ページのコンテンツはContent View Controllerが提供します。Page View Controllerはページ間の遷移を管理します。新しいページが必要になると、Page View Controllerは関連するデータソースを呼び出して、次のページを表示するView Controllerを取得します。

Page View Controllerを設定したり使用したりする方法については、「Page View Controllers」を参照してください。

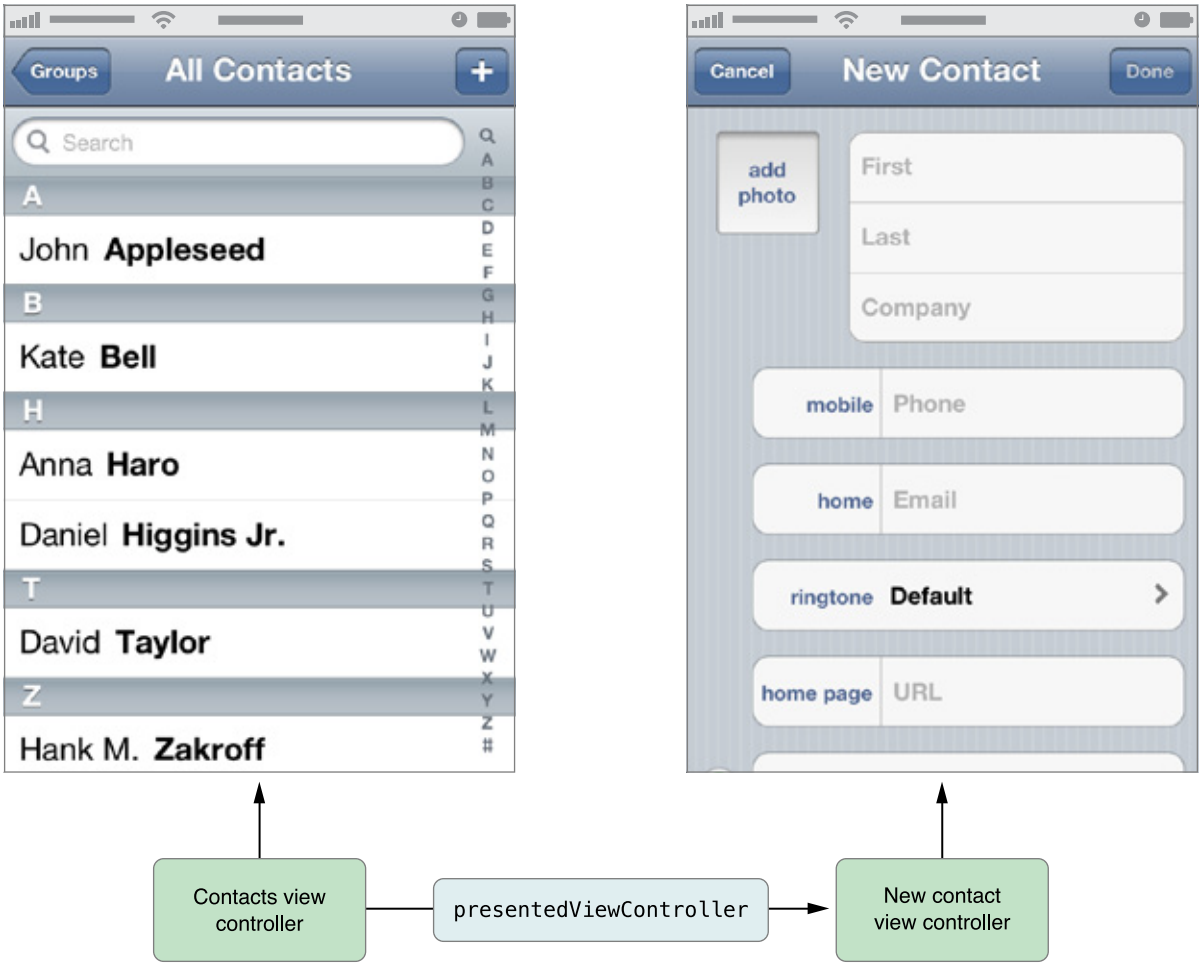
View Controllerのコンテンツはさまざまな方法で表示できる

View Controllerのコンテンツをユーザに向けて表示するためには、これをウインドウに関連づける必要があります。その方法は次のようにいくつかあります。

- View Controllerを、ウインドウのルートView Controllerにする方法。
- View Controllerをコンテナの子にする方法。
- Popoverコントロール内にView Controllerを表示する方法。
- ほかのView Controllerからの「プレゼンテーション」操作により表示する方法。

図1-10に「連絡先(Contacts)」アプリケーションの例を示します。ユーザが新規の連絡先を追加するために「+」ボタンをクリックすると、「Contacts」View Controllerは、「New Contact」View Controllerを表示します。ユーザがこの操作をキャンセルするか、連絡先データベースに保存するのに十分な連絡先情報を入力するまでは、「新規連絡先(New Contact)」画面が表示されたままになります。操作が完了した時点で、「Contacts」View Controllerは子ビューを閉じ、したがって画面から消えます。

図 1-10 View Controllerの表示



表示される側のView Controllerに、特に変わったところはありません。Content View Controllerであっても、別のContent View Controllerを包含するContainer View Controllerであってもよいのです。実際上、Content View Controllerはほかの Controllerから表示されることを想定した設計になっているので、ある種のContent View Controllerであると考えれば分かりやすいでしょう。コンテナView Controllerは、管理下にあるView Controller間の特定の関係を定義しますが、「プレゼンテーション」の仕組みを用いると、（表示される側の）被表示View Controllerと、（これを表示する側の）表示View Controllerとの関係を定義できます。

ほとんどの場合は、ユーザから情報を収集したり、特定の目的でユーザの注意を引くためにView Controllerを表示します。その目的が達成されると、表示した側のView Controllerは被表示View Controllerを画面から消去し、標準アプリケーションインターフェイスに戻ります。

被表示View Controller自身が、立場を変えて、別のView Controllerを「表示する」側になりうることにも注意してください。View Controllerを連鎖させることができる機能は、複数のモーダルアクションを順番に実行する必要があるときに役立ちます。たとえば、図 1-10の「新規連絡先(New Contact)」画面で、ユーザが「写真を追加（Add Photo）」ボタンをタップして既存の画像を選ぶ場合、「New Contact」View Controllerは画像ピッカーインターフェイスを表示します。連絡先のリストに戻るには、この画像ピッカー画面を閉じてから、「新規連絡先(New Contact)」画面を個別に閉じなければなりません。

View Controllerの表示に当たっては、一方の（表示する側の）View Controllerが、そのために画面上のどの位の領域を占めるか判断します。画面上のこの領域のことを*表示コンテキスト*と言います。デフォルトでは、表示コンテキストはウインドウ全体を占めると定義されます。

アプリケーションでView Controllerを表示する方法について詳しくは、“View ControllerをほかのView Controllerから「表示」”（95 ページ）を参照してください。

View Controller同士が連携してアプリケーションのインターフェイスを形成する

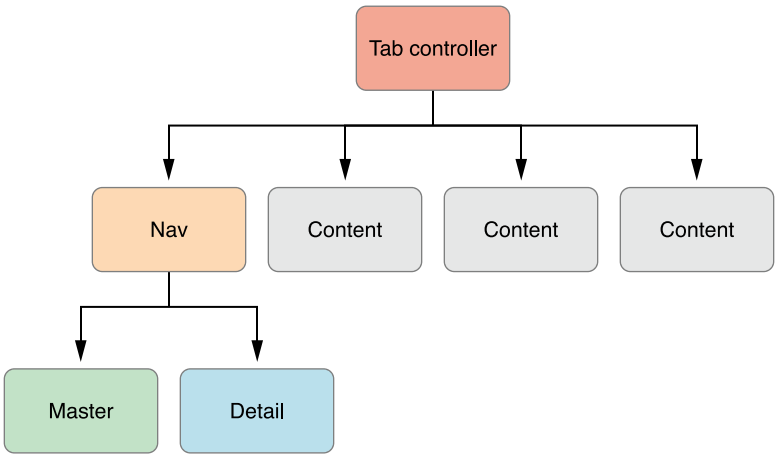
個々のView Controllerは、ビューおよび関連するオブジェクトを管理するだけでなく、ほかのView Controllerと連携して、シームレスなユーザインターフェイスを実現します。アプリケーションのView Controller間で処理を分担し、相互に通信する仕組みは、View Controllerを利用する上で重要な部分です。複雑なアプリケーションを構築する上でView Controller間の関係は重要なので、次の節では、ひと通りおさらいした後、さらに詳しく説明することにしましょう。

親子関係は包含関係を表す

View Controller階層の基点となるのは、ウインドウのルートView Controllerという、単一の親です。このView Controllerがコンテナであれば、そのコンテンツを提供するいくつかの子が属します。子もそれ自身がコンテナであって、さらにいくつかの子が属しているかも知れません。図 1-11にView Controller

階層の例を示します。ルートView Controllerは、4つのタブを持つTab View Controllerです。第1のタブはNavigation Contollerが管理し、それ自身にも子が属しますが、残り3つのタブはContent View Controllerが管理し、子はありません。

図 1-11 親子関係



各View Controllerが描画する領域は親が決めます。ルートView Controllerの描画領域を決めるのはウインドウです。図 1-11で、Tab View Controllerはそのサイズをウインドウから取得します。その中にTab Barの領域を確保し、残りを子の領域に充当します。Navigation Controllerのタブに切り替わると、Navigation Bar用に領域の一部を確保し、残りをContent Controllerに渡します。以上の各ステップで、親は子View Controllerのビューの大きさを変更し、親のビュー階層内に位置付けます。

さらに、ビューとView Controllerは連携して、アプリケーションが扱うイベントのレスポンダチェーンを形作ります。

兄弟関係はコンテナ内に並ぶ関係を表す

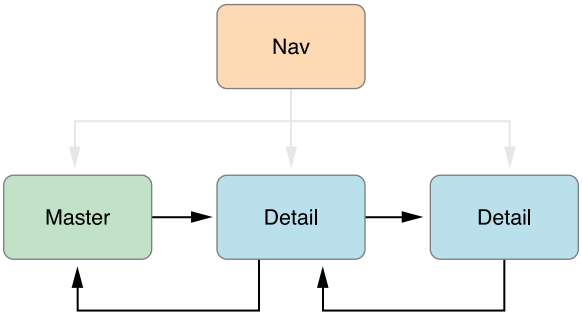
コンテナの種類によって、その子同士の関係（もしあれば）が決まります。たとえば、Tab View ControllerとNavigation Controllerを比べてみましょう。

- Tab View Controllerの場合、（兄弟関係にある）各タブは完全に独立したコンテンツ画面を表します。Tab Bar Controllerは子同士の関係を定義しません（不可能というわけではありませんが）。
- Navigation Controllerの場合、兄弟は、積み重なるように配置される、相互に関連するビューを表します。すぐ上の兄、すぐ下の弟との間には、「接続」があるのが普通です。

図 1-12に、Navigation Controllerに結びつけられたView Controllerの一般的な構成を示します。第1子である「Master」には、詳細を省いた目次部分だけが表示されます。目次のある項目が選択されると、新たに弟（第2子）をNavigation Controllerにプッシュして、詳細が表示されるようにします。さらにその中のある項目について、ユーザがさらに詳しく知ろうとすると、第2子がさらに別のView Controller

（第3子）をプッシュして、もっと詳しい内容を表示するようになっています。この例のように適切な関係が定義された兄弟は、直接、またはContainer Controllerを介して、相互に連携し合います。図 1-15 （33 ページ）を参照してください。

図 1-12 Navigation Controllerにおける兄弟関係

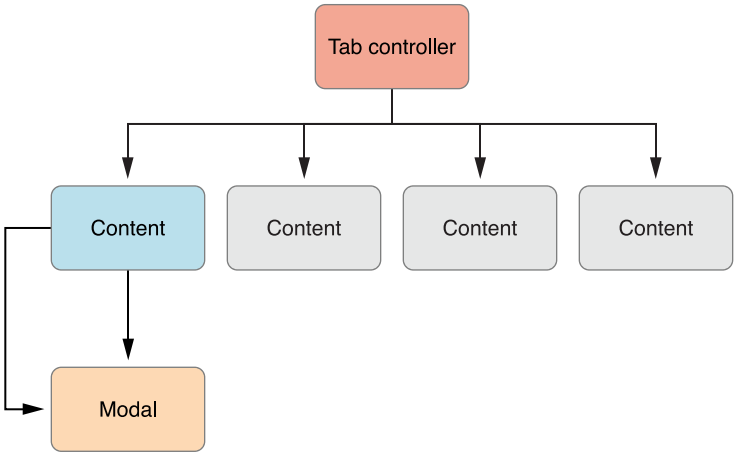


表示/被表示の関係は、ほかのインターフェイスを一時的に表示する、という関係を表す

View Controllerは、ほかのView Controllerを「表示」することにより、あるタスクの実行を委ねます。表示する側のView Controllerがこの動作を管理します。すなわち、被表示View Controllerに対して必要な設定を施し、（タスク実行後）被表示View Controllerから情報を受け取り、その後画面から消去します。しかし、被表示View Controllerのビューは、表示されている間一時的に、ウインドウのビュー階層に組み込まれます。

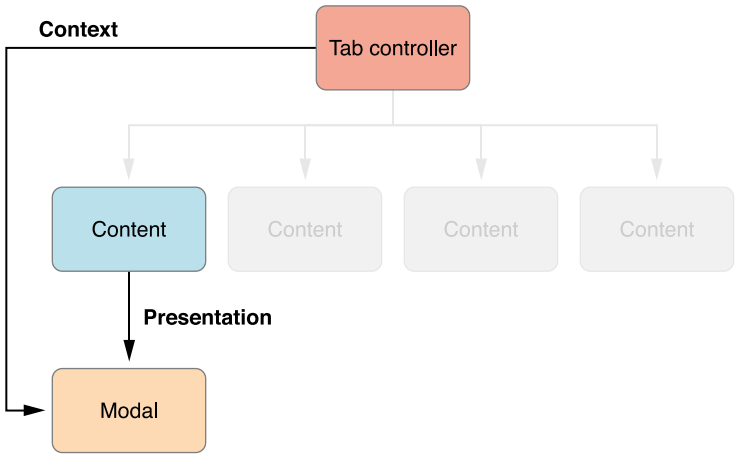
図 1-13に、Tab Viewにアタッチされたあ るContent View Controllerが、View Controllerを表示してタスクを実行する様子を示します。この図のContent Controllerが表示する側のView Controller、モーダルView Controllerが被表示View Controllerに当たります。

図 1-13 Content Viewによるモーダルプレゼンテーション



View Controllerが表示される際、画面上のこれが占める部分は、別のView Controllerが提供する表示コンテキストで定義されます。表示コンテキストを提供するView Controllerは、表示する側のView Controllerと同じでなくても構いません。図 1-14に、図 1-13に示されているのと同じView Controller階層を示します。この場合、Content ViewはView Controllerを表示していますが、表示コンテキストは提供しません。提供しているのはTab Controllerです。表示する側のView Controllerが占めるのは、Tab View Controllerが配分した画面領域だけですが、上記のような構成になっているので、被表示View ControllerはTab View Controllerが所有する領域全体を使えることになります。

図 1-14 実際のプレゼンテーションは、ルートView Controllerが実行します。



制御フローはContent Controller同士が連携して動作することを表す

View Controllerが複数あるアプリケーションの場合、各View Controllerは通常、アプリケーションの起動から終了までの間に生成と破棄を繰り返します。View Controllerは生存している間、相互に通信して、一貫したユーザインターフェイスを実現します。この関係はアプリケーションの制御フローを表します。

このような制御フローが出現する例として、View Controllerのインスタンス生成を考えてみましょう。通常、View Controllerのインスタンスが生成されるのは、ほかのView Controllerのアクションによってです。第1のView Controller（*ソースView Controller*）は、第2のView Controller（*デスティネーションView Controller*）を管理します。デスティネーションView Controllerがユーザに対してデータを表示する場合、そのデータを提供するのは一般にソースView Controllerです。同様に、ソースView Controllerがデスティネーション View Controllerから情報を取得する必要がある場合、両者間に接続を確立するのはソース側の責任です。

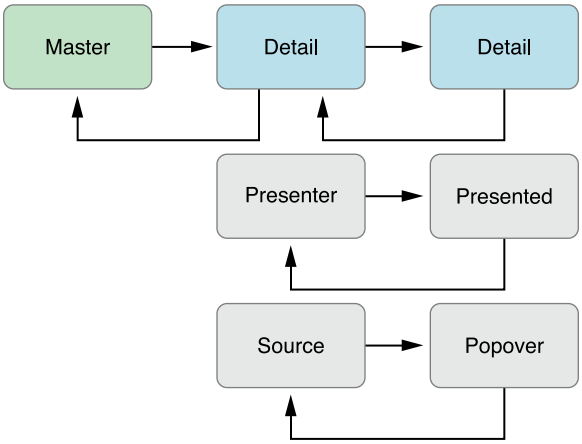
図 1-15に、この関係のもっとも一般的な例を示します。

通信の手順は次のようになっています（図も参照）。

- Navigation Controllerの子は、ほかの子をNavigationスタックにプッシュします。

- View ControllerはほかのView Controllerを表示します。
- View ControllerはほかのView ControllerをPopover内に表示します。

図 1-15 ソースView ControllerとデスティネーションView Controllerの通信



各コントローラを設定するのは、先行する（図の左側の）コントローラです。複数のコントローラが連携する場合、アプリケーション全体を貫く通信チェーンを確立することになります。

このチェーンの各リンクにおける制御フローは、デスティネーションView Controllerが定義します。ソースView Controllerは、デスティネーションView Controllerが定める規約に従います。

- デスティネーションView Controllerは、自分自身データやプレゼンテーションの設定に用いるプロパティを提供します。
- デスティネーションView Controllerが、チェーン上の先行するView Controllerと通信する必要がある場合は、デリゲーションを使います。ソースView Controllerがデスティネーション View Controllerのほかのプロパティを設定する際には、デリゲートのプロトコルを実装するオブジェクトも提供しなければなりません。

この制御フローの規約を用いると、ソース/デスティネーションView Controllerの組で、明確に責任を分担できるという利点があります。ソースView Controllerがデスティネーション View Controllerにタスクを実行するよう指示すると、この経路に沿ってデータが流れます。このプロセスを駆動するのはソースView Controllerです。たとえば、デスティネーションControllerが表示するデータオブジェクトを提供する、という流れです。一方、View Controllerが、自分自身を表示したソースControllerに情報を返す必要が生じると、この経路に沿って逆向きにデータが流れます。たとえばタスク完了時の通信（終了通知）がこれに当たります。

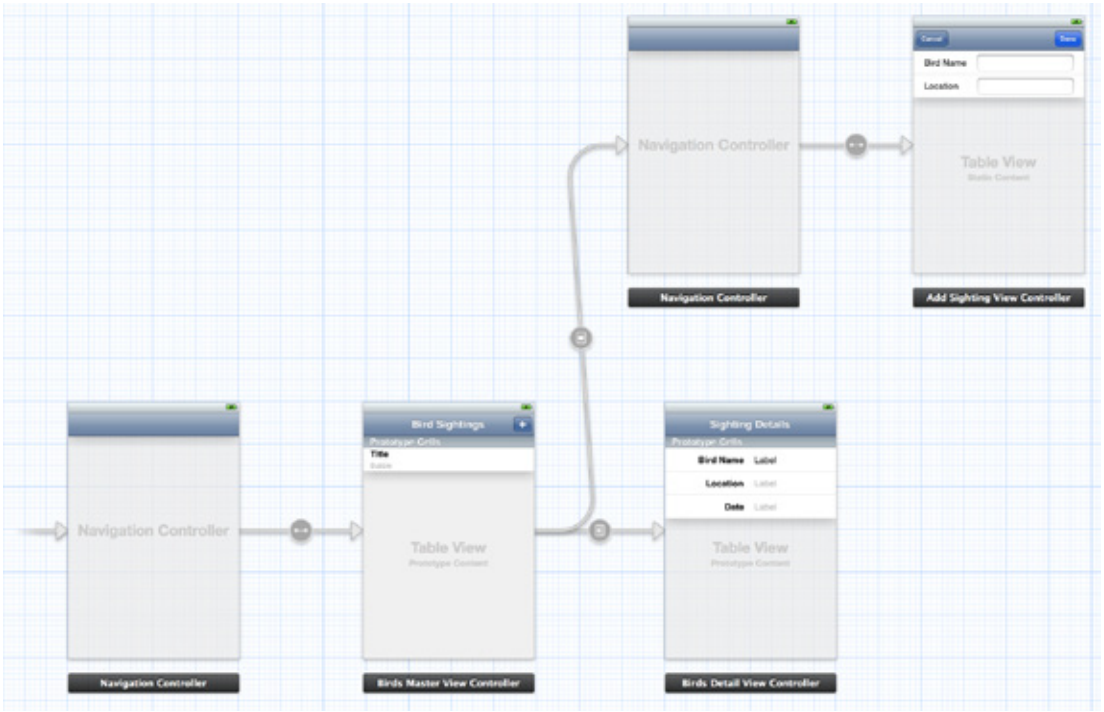
さらに、この制御フローモデルを首尾一貫して実装すれば、デスティネーションView Controllerが、自分自身を設定したソースView Controllerについて、実装詳細を知る必要が決して生じないようになります。チェーンの前段にあるView Controllerについては、そのクラスにデリゲートプロトコルが実

装されていることを認識していれば、クラスの詳細は知らなくても構いません。View Controllerが相互に依存しすぎないようにすることにより、個々のコントローラの再利用性も高まります。さらに、一貫した制御フローモデルに従って実装すると、コントローラ間の通信経路がコードを読む者にとってわかりやすくなる、という利点もあります。

ストーリーボードを使えばユーザインターフェイスを容易に設計できる

アプリケーションの実装にストーリーボードを使う場合、Interface BuilderでView Controllerやこれに関連するビューを編成することになります。図 1-16に、Interface Builderに表示されたインターフェイスレイアウトの例を示します。Interface Builderには、アプリケーション全体の流れが一目でわかるよう、フローが図で表示されます。アプリケーション上で、どのようなView Controllerのインスタンスがどの順序で生成されるか、を把握できます。そればかりでなく、ビューその他のオブジェクトの複雑なコレクションも、ストーリーボード上で設定できます。結果として得られたストーリーボードはプロジェクトにファイルとして格納されます。プロジェクトを構築する際、ストーリーボードは必要な処理を施した上で、アプリケーションバンドルにコピーされます。その結果、アプリケーションは実行時にこれをロードできるようになります。

図 1-16 Interface Builderに表示されるストーリーボード図



iOSは多くの場合、必要が生じた時点で自動的に、ストーリーボードに定義されたView Controllerのインスタンスを生成します。同様に、各コントローラに対応するビュー階層も、表示が必要になった時点で自動的にロードします。View Controllerもビューも、Interface Builderで設定したのと同じ属性に基づいてインスタンスが生成されます。以上の処理の大部分は自動化されているので、アプリケーションでView Controllerを使うための作業は、非常に簡潔になっています。

ストーリーボードの作成について詳しくは、『*Xcode User Guide*』を参照してください。当面は、アプリケーションでストーリーボードを実装する際に用いる、重要な用語をいくつか知っておけばよいでしょう。

シーンは、View Controllerが管理する画面上のコンテンツ領域を表します。View Controllerおよびこれに対応するビュー階層のことと考えてもよいでしょう。

このストーリーボード上で、シーン間の**関係**も定義します。ストーリーボードでは、あるシーンから別のシーンへ向かう矢印として視覚的に表現されます。Interface Builderは通常、オブジェクト間に接続を定義すると、その詳細を自動的に推論するようになっています。中でも重要な、2種類の関係を説明しておきましょう。

- **包含関係**とは、2つのシーン間の親子関係を表します。ほかのView Controllerに含まれているView Controllerは、親コントローラのインスタンス生成時に、同時に生成されます。たとえば、Navigation Controllerからほかのシーンへの接続の1本めは、Navigationスタックに最初にプッシュされるView Controllerを定義します。このView Controllerのインスタンスは、Navigation Controllerのインスタンス生成時に、自動的に生成されます。

  ストーリーボードで包含関係を使うと、子View Controllerの外観を、親（や祖先）を考慮してInterface Builderが自動的に調整できる、という利点があります。したがってInterface Builderは、最終的なアプリケーションと同じような外観で、Content View Controllerを表示することができるのです。

- **セグエ**とは、あるシーンから別のシーンへの視覚的な遷移を表します。実行時、セグエはさまざまなアクションによりトリガされます。セグエにトリガがかかると、新しいView Controllerのインスタンスが生成され、画面遷移が起こります。

  セグエは常に、あるView Controllerから別のView Controllerへの遷移です。しかしこのプロセスには、第3のオブジェクトが関与する場合があります。このオブジェクトが実際にはセグエをトリガするのです。たとえば、ソースView Controllerのビュー階層にあるボタンから、デスティネーションView Controllerへの接続を作ると、ユーザがボタンをタップしたとき、セグエにトリガがかかるようになります。一方、ソースView Controllerからデスティネーション View Controllerに直接つながるセグエを作ると、これは通常、トリガをかける処理をプログラムで実装することを表します。

  遷移元/先のView Controllerの種類に応じて、各種のセグエがあります。

  - **プッシュセグエ**は、デスティネーションView Controllerを、Navigation Controllerのスタックにプッシュします。

- **モーダルセグエ**は、デスティネーションView Controllerを表示します。
- **ポップオーバセグエ**は、デスティネーションView ControllerをPopover内に表示します。
- **カスタムセグエ**を使えば、デスティネーションView Controllerを表示する、独自の遷移を設計できます。

セグエのプログラミングモデルはいずれも共通です。このモデルでは、iOSがデスティネーションView Controllerのインスタンスを自動的に生成し、設定を施すことができるよう、ソースView Controllerを呼び出します。この動作は"制御フローはContent Controller同士が連携して動作することを表す"（32 ページ）に示した制御フローモデルにもかないます。

また、View Controllerと、同じシーンに格納されたオブジェクトとの間に接続を定義することも可能です。アウトレットやアクションと呼ばれるこの接続を使えば、View Controllerとこれに関連するオブジェクトとの間に、綿密な関係を定義できます。接続は通常、ストーリーボードには表示されませんが、Interface BuilderのConnections Inspectorを使えば確認できます。

アプリケーションにおけるView Controllerの使い方

iOS組み込みのView Controllerでも、アプリケーションのコンテンツを表示するために独自に作成したカスタムView Controllerでも、実際の開発で用いる技法はそれほど変わりません。

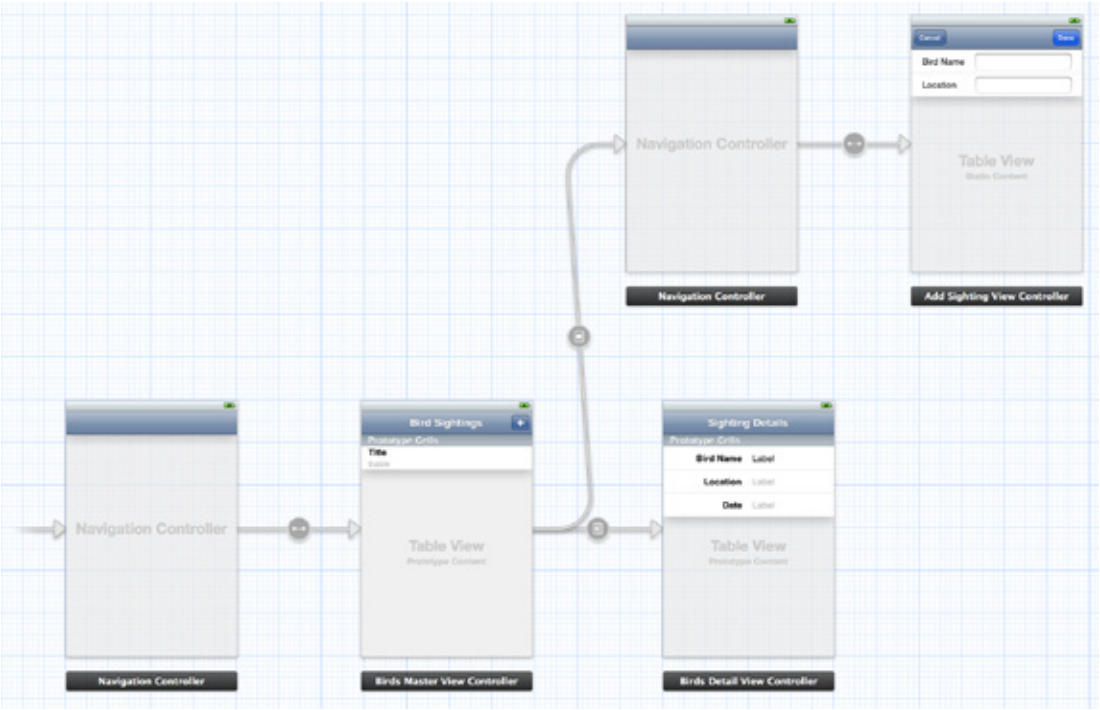
View Controllerにまつわるさまざまな技法の中でも特に有用なのは、これをストーリーボードに置くことでしょう。この方法を使えば、アプリケーションにおけるView Controller間の関係を、コードを記述することなく直接定義できます。アプリケーション起動時にまず生成されるコントローラから、ユーザのアクションに応答して生成されるコントローラに至る、制御フローをここで見ることができます。iOSはこのプロセスの大部分を、必要が生じた時点でView Controllerのインスタンスを生成する、というやり方で管理します。

一方、View Controllerの領域を割り当て、初期化するプログラムにより、そのインスタンスを生成しなければならないこともあります。View Controllerを直接操作するためには、インスタンスを生成し、必要な設定を施し、表示するコードを記述しなければなりません。

ストーリーボードにおけるView Controllerの操作

図2-1にストーリーボードの例を示します。ここには、View Controllerと対応するビュー、そしてView Controller間の関係を表す接続矢印があります。このストーリーボードは、あるシーンから始まり、ユーザのアクションに応じて別のシーンに切り替わっていく、一連のストーリーを語っています。

図 2-1 ストーリーボードには一連のView Controllerと対応するオブジェクトがある



ストーリーボードでは、あるView Controllerを、初期View Controllerと指定することができます。これがUIの流れのある範囲における具体的なワークフローを表すとすれば、初期View Controllerはその先頭のシーンに相当します。

ストーリーボードでは、初期View Controllerからほかの View Controllerへ向かう関係を定義します。さらに、そこから別のView Controllerへの関係も順に定義していくと、最終的に、ほとんどすべてのシーンが接続された、ひとまとまりのグラフになります。定義した関係の種類によって、接続先のView Controllerのインスタンスが、どのような場合に生成されるかが決まります。

- 関係がセグエであれば、デスティネーションView Controllerのインスタンスが生成されるのは、セグエにトリガがかかったときです。
- 包含関係を表すものであれば、子View Controllerのインスタンスが生成されるのは、親のインスタンスが生成されるときです。

- ほかのコントローラのデスティネーションでも子でもないコントローラの場合、自動的にインスタンスが生成されることはありません。ストーリーボードに基づいて生成する、プログラムを記述する必要があります。

ストーリーボードにある特定のView Controllerやセグエを識別したい場合は、Interface Builder上で、これを一意に識別する文字列を指定します。ストーリーボードからView Controllerをロードするプログラムを記述する場合、この識別文字列は必須です。同様に、セグエにトリガをかけるプログラムを記述する場合も、識別文字列を与える必要があります。セグエにトリガがかかると、ソースView Controllerにはその識別文字列が通知されます。ソースView Controllerはこれを手がかりに、どのセグエにトリガがかかったのか判断できるのです。したがって、どのセグエにも識別文字列を与えておくとよいでしょう。

アプリケーションの構築にストーリーボードを使う場合、単一のストーリーボードに必要なView Controllerをすべて定義する方法と、複数のストーリーボードを作成し、ユーザインターフェイスをそれぞれに分割して定義する方法があります。ほとんどの場合、いずれかのストーリーボードをメインとして指定することになります。メインストーリーボードがあれば、iOSはこれを自動的にロードします。他のストーリーボードはアプリケーションが明示的にロードしなければなりません。

メインストーリーボードはアプリケーションのユーザインターフェイスを初期化する

メインストーリーボードは、アプリケーションの情報プロパティリストファイルに定義します。ここにメインストーリーボードが宣言されていれば、アプリケーションの起動時に、iOSが次の処理を行います。

1. ウインドウのインスタンスを生成する。
2. メインストーリーボードをロードし、初期View Controllerのインスタンスを生成する。
3. このView Controllerをウインドウの`rootViewController`プロパティに設定し、ウインドウを画面上に表示する。

初期View Controllerを実際に表示する前に、アプリケーションデリゲートを呼び出すようになっているので、ここで必要な設定を施すことができます。iOSがメインストーリーボードをロードする詳しい手順については、“View Controller間の連携”（105 ページ）を参照してください。

セグエは自動的にデスティネーションView Controllerのインスタンスを生成する

セグエはトリガによる遷移を表し、トリガがかかると、新しいView Controllerに従ってユーザインターフェイスが切り替わります。

セグエには、次のような、遷移に関するさまざまな情報が格納されています。

- セグエにトリガをかけたオブジェクト（センダ）。
- セグエを開始する側であるソースView Controller
- （トリガを受けて）インスタンスが生成されるデスティネーションView Controller
- 遷移の種類（デスティネーションView Controllerを画面に表示する方法を表す）
- ストーリーボード内で特定のセグエを識別するための文字列（オプション）

セグエにトリガがかかると、iOSは次のようなアクションを実行します。

1. ストーリーボードで設定した属性値に基づき、デスティネーションView Controllerのインスタンスを生成する。
2. ソースView Controller側に、新しいコントローラに設定を施す機会を与える。
3. セグエの設定に従って遷移を実行する。

> **注意:** カスタムView Controllerを実装する場合、デスティネーションView Controller側には、公開のプロパティやメソッドを宣言して、ソースView Controllerが必要な設定を施せるようにします。一方、ソースView Controller側では、基底クラスのストーリーボード操作メソッドをオーバーライドして、デスティネーションView Controllerに必要な設定を施す処理を実装します。詳しくは"View Controller間の連携"（105 ページ）を参照してください。

プログラムでセグエにトリガをかける

通常、セグエにトリガをかけるのは、ソースView Controllerに関連するオブジェクト（コントロール、Gesture Recognizerなど）です。しかし、セグエに識別文字列が割り当てられていれば、トリガをかけるプログラムを記述することも可能です。たとえばゲームの場合、対戦終了時にトリガをかける必要があるかも知れません。これを受けて、デスティネーションView Controllerでは最終成績を表示することになります。

セグエにトリガをかけるためには、当該セグエの識別文字列を指定して、ソースView Controllerの`performSegueWithIdentifier:sender:`メソッドを呼び出します。このメソッドには、センダとして動作するオブジェクトも指定します。デスティネーションView Controllerに必要な設定を施せるよう、（遷移処理の途中で）ソースView Controllerが呼び出されますが、その引数として、センダオブジェクトと、セグエの識別文字列が渡されます。

リスト2-1に、セグエにトリガをかける簡単なメソッドの例を示します。これは"代替の横長インターフェイスの作成"（87 ページ）から一部を抜粋したものです。この断片から、View Controllerが「向き変更」通知を受け取っている状況が分かるでしょう。View Controllerが縦長モードのとき、デバイ

スが回転して横長の向きになると、このメソッドはセグエを使って、画面を描画するView Controllerを切り替えます。この場合、通知オブジェクトはセグエのコマンドに有用な情報を渡さないので、View Controllerは自分自身をsenderとします。

リスト 2-1　プログラムでセグエにトリガをかける

```
- (void)orientationChanged:(NSNotification *)notification
{
    UIDeviceOrientation deviceOrientation = [UIDevice currentDevice].orientation;
    if (UIDeviceOrientationIsLandscape(deviceOrientation) &&
        !isShowingLandscapeView)
    {
        [self performSegueWithIdentifier:@"DisplayAlternateView" sender:self];
        isShowingLandscapeView = YES;
    }
// これ以降の部分は省略。
}
```

プログラムでセグエにトリガをかけることしかできない場合は、通常、ソースView Controllerからデスティネーションview Controllerに直接、接続矢印を描画します。

ストーリーボードのView Controllerのインスタンスをプログラムで生成する

View Controllerのインスタンスを、セグエを使わずに、プログラムで生成したい場合もあります。ストーリーボードが有用であることに違いはありません。View Controllerの属性やビュー階層の設定に利用できるからです。しかし当然ながら、セグエの機能を利用することはできません。View Controllerを表示するためには、さらにコードを実装する必要があります。したがって、可能であればセグエを活用し、どうしても必要な場合にのみこの技法を用いるとよいでしょう。

コードに記述するべき処理を以下に示します。

1. ストーリーボードオブジェクト（`UIStoryboard`クラスのインスタンス）を取得する。

   同じストーリーボードからインスタンスを生成した既存のView Controllerがあるならば、その`storyboard`プロパティから取得できます。別のストーリーボードをロードする場合は、`UIStoryboard`のクラスメソッドである`storyboardWithName:bundle:`を、ストーリーボードファイル名と、（必要な場合は）バンドルを指定して呼び出します。

2. ストーリーボードオブジェクトの`instantiateViewControllerWithIdentifier:`メソッドを、Interface BuilderでView Controllerを生成する際に定義した、識別文字列を渡して呼び出す。

ストーリーボードの初期View Controllerであれば、`instantiateInitialViewController`メソッドでインスタンスを生成できます。この場合、識別文字列は必要ありません。

3. このView Controllerのプロパティを設定する。
4. このView Controllerを表示する。詳しくは"View Controllerのコンテンツをプログラムで表示する"（44 ページ）を参照してください。

リスト 2-2にこの技法を使った例を示します。既存のView Controllerからストーリーボードを取得し、これを使って新しいView Controllerのインスタンスを生成しています。

リスト 2-2　同じストーリーボードにあるほかのView Controllerのインスタンスを生成

```
- (IBAction)presentSpecialViewController:(id)sender {
    UIStoryboard *storyboard = self.storyboard;
    SpecialViewController *svc = [storyboard
instantiateViewControllerWithIdentifier:@"SpecialViewController"];

    // 新しいView Controllerに対して設定を施す

    [self presentViewController:svc animated:YES completion:nil];
}
```

リスト 2-3に、もうひとつよく使われる技法を示します。この例では、新しいストーリーボードをロードし、その初期View Controllerのインスタンスを生成しています。このView Controllerを、外部画面に配置する新しいウインドウの、ルートView Controllerとして使うのです。返されたウインドウを表示するためには、ウインドウの`makeKeyAndVisible`メソッドを使います。

リスト 2-3　新しいストーリーボードからView Controllerのインスタンスを生成する

```
- (UIWindow*) windowFromStoryboard: (NSString*) storyboardName
                                onScreen: (UIScreen*) screen
{
    UIWindow *window = [[UIWindow alloc] initWithFrame:[screen bounds]];
    window.screen = screen;

    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:storyboardName
bundle:nil];
    MainViewController *mainViewController = [storyboard
instantiateInitialViewController];
```

```
    window.rootViewController = mainViewController;

    // 新しいView Controllerに対して設定を施す

    return window;
}
```

別のストーリーボードに遷移するためにはプログラムの記述が必要である

セグエで接続できるのは、同じストーリーボード内のシーンに限ります。ほかのストーリーボードのView Controllerを表示するためには、ストーリーボードファイルを明示的にロードし、その中のView Controllerのインスタンスを生成しなければなりません。

ストーリーボードがひとつだけでも、特に困ることはありません。しかし次のような場合には、複数のストーリーボードを使うとよいでしょう。

- 大規模なプログラミングチームがいくつかのサブチームに分かれ、ユーザインターフェイス部分を明確に分担しているとき。この場合、各サブチームのメンバーは、それぞれのストーリーボードの内容に応じた範囲の開発を担当します。
- 各種のView Controllerのコレクションをあらかじめ定義した、ライブラリを購入または作成する場合。各View Controllerの内容は、ライブラリが提供するストーリーボードで定義されています。
- 外部画面に表示するコンテンツがあるとき。この場合、外部画面に対応するView Controllerはすべて、独立したストーリーボードにまとめる必要があります。同じ状況で、独自のセグエを作成することも考えられます。

コンテナは自動的に子のインスタンスを生成する

ストーリーボード内のコンテナのインスタンスを生成すると、その子のインスタンスも自動的に生成されます。同時に生成する必要があるのは、Container Controller側に、子のコンテンツの一部を表示するためです。

同様に、生成された子がやはりコンテナであれば、孫のインスタンスも生成することになり、包含関係をたどれる限り同じ処理が続きます。Tab Bar Controllerの内部にNavigation Controllerを置き、さらにその内部にContent Controllerを置くと、Tab Barのインスタンスを生成したとき、同時に3つのControllerのインスタンスが生成されることになります。

コンテナやその子孫のインスタンスが生成された後で、これに設定を施すため、View Controllerが呼び出されます。このとき、ソースView Controller（またはアプリケーションデリゲート）は、子のインスタンスがすべて生成済みである、という前提で処理できます。このようなインスタンス生成の動作が重要なのは、カスタム設定コードがコンテナの設定をすることはほとんどないからです。代わりに、コンテナにアタッチされたContent Controllerに設定を施します。

ストーリーボードを使わずにView Controllerのインスタンスを生成する

ストーリーボードを使わず、プログラムでView Controllerを生成するためには、これを割り当て、初期化するという、Objective-Cのコードを使います。ストーリーボードの恩恵はないので、新しいView Controllerに必要な設定を施し、表示するコードも実装しなければなりません。

View Controllerのコンテンツをプログラムで表示する

View Controllerのコンテンツは、画面に表示しなければ役に立ちません。View Controllerのコンテンツを表示する方法は、いくつか選択肢があります。

- View Controllerを、ウインドウのルートView Controllerにする方法。
- 可視のContainer View Controllerの子にする方法。
- ほかの可視のView Controllerから「プレゼンテーション」操作により表示する方法。
- Popoverを使って表示する方法（iPadのみ）。

いずれの場合も、View Controllerをほかのオブジェクト（この場合、ウインドウ、View Controller、Popover Controller）に割り当てることになります。割り当てられたオブジェクトは、View Controllerのビューの大きさを変え、自身のビュー階層に追加して、表示されるようにします。

リスト2-4に、View Controllerをウインドウに割り当てる、もっとも一般的なコード例を示します。このコードでは、ストーリーボードは使わないと想定しているので、通常であればオペレーティングシステムが代行してくれるのと同じ処理を行います。すなわち、ウインドウを生成し、新しいControllerをルートView Controllerとして設定しています。次いで、そのウインドウを可視にします。

リスト 2-4　View Controllerをウインドウのルート View Controllerとしてインストール

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    UIWindow *window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];
```

```
    levelViewController = [[LevelViewController alloc] init];
    window.rootViewController = levelViewController;
    [window makeKeyAndVisible];
}
```

Important: View Controllerのビューを、直接ビュー階層に組み込まないでください。ビューを適切に表示したり管理するために、システムは、表示するそれぞれのビュー（およびそれに対応するView Controller）を記録します。システムは、後でこの情報を使用してView Controller関連のイベントをアプリケーションに報告します。たとえば、デバイスの向きが変化した場合、ウインドウはこの情報を使用して、最前面のView Controllerを識別し、それに変化を通知します。これ以外の方法でView Controllerのビューをビュー階層に追加すると、システムがこのようなイベントを正しく扱えなくなるおそれがあります。

カスタムContainer Controllerを実装する場合、他のView Controllerのビューを独自のビュー階層に追加しますが、すると親子関係も作成されることになります。イベントはこの関係に基づいて正しく配送されます。詳しくは“カスタムContainer View Controllerの作成”（119 ページ）を参照してください。

カスタムContent View Controllerの作成

カスタムContent View Controllerはアプリケーションの心臓部です。これを使って、アプリケーション特有のコンテンツを表示します。アプリケーションはすべて、少なくとも1つ、カスタムContent View Controllerが必要です。複雑なアプリケーションでは、Content Controllerを複数用意し、作業を分担します。

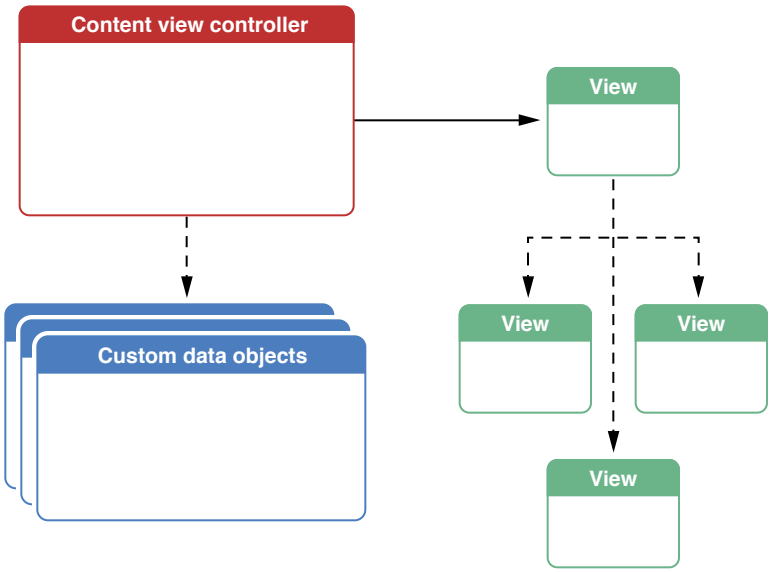
View Controllerはさまざまな責務を担います。iOSが要求する責務のほか、アプリケーションの設計時に、View Controllerの役割として定義するものもあります。

Content View Controllerの構造

`UIViewController`クラスは、すべてのカスタムView Controllerを実装するための基本的なインフラストラクチャを提供します。`UIViewController`のカスタムサブクラスを定義します。このサブクラスには、データをビューに表示し、ユーザのアクションに応答するためのコードを実装します。View Controllerのデフォルトの動作に変更を加えたい場合は、`UIViewController`クラスのメソッドをオーバーライドします。View ControllerがほかのUIKitクラスと連携して、必要な動作を実装することもあります。

図 3-1に、Content View Controllerに直接関連付けられる主要なオブジェクトを示します。これらのオブジェクトは、実質的にはView Controller自身が所有し管理します。ビュー（`view`プロパティを介してアクセス可能）は、View Controllerに提供しなければならない唯一のオブジェクトです。ただし、ほとんどのView Controllerは、これにアタッチされた追加のサブビューや、表示する必要があるデータを含むカスタムオブジェクトも持っています。

図 3-1 Content View Controllerの構造



新たに設計するView Controllerは、さまざまな責務を担う可能性があります。その中には、制御対象であるビューその他のオブジェクトの、内部を意識しなければならないものもあります。ほかのコントローラが外部に公開しているものを意識するだけで十分な責務もあります。以下、View Controllerが担うべき主な責務について説明します。

View Controllerはリソースを管理する

View Controllerを初期化する際、オブジェクトのインスタンスがいくつか生成され、View Controllerの解放時に破棄されます。ビューのように、View Controllerのコンテンツを画面に表示するときだけ必要になるオブジェクトもあります。したがって、View Contollerはリソースを効率的に利用しなければならないので、メモリが不足しそうになったらいつでも解放できるよう備えておく必要があります。アプリケーションのView Controllerに、このような動作を正しく実装すれば、メモリその他のリソース（CPU、GPU、電池など）を効率よく使えるようになります。

詳しくは"View Controllerにおけるリソース管理"（60 ページ）を参照してください。

View Controllerはビューを管理する

View Controllerはビューとそのサブビューを管理しますが、ビューの枠（親ビューを基準とした位置と大きさ）は多くの場合、デバイスの向き、ステータスバーの表示の有無、View Controllerのビューがウインドウ内にどのように表示されるかなど、他の因子によって決まります。View Controllerは、与えられた枠に合わせてビューをレイアウトするよう設計しなければなりません。

ビューの管理については、ほかにも考えるべき事項があります。ビューが画面に表示されようとしている、あるいは消去されようとしているとき、View Controllerには通知が届きます。この通知を利用して、表示や消去の処理に必要な、ほかのアクションを実行できるようになっているのです。

“View Controllerのビューの大きさ変更”（74 ページ）、“デバイスの向きに応じた調整”（80 ページ）、“表示関連の通知への応答”（71 ページ）を参照してください。

View Controllerはイベントに応答する

View Controllerは多くの場合、ビューやコントロールの司令塔オブジェクトとして振る舞います。典型的には、ユーザがコントロールを操作したとき、当該コントロールがコントローラにメッセージを送信する、という形でユーザインターフェイスを設計します。View Controllerは、メッセージを処理し、ビュー、あるいは自分自身に格納されたデータに対して、必要な変更を施します。

View Controllerはまた、イベントをアプリケーションに配送するために用いる、レスポンダチェーンにも関与します。View Controllerクラスのメソッドをオーバーライドして、イベント処理に直接参加することもできます。さらに、View Controllerは、システム通知、タイマー、その他アプリケーションに特有のイベントの処理を実装するためにも適したオブジェクトです。

詳しくは“レスポンダチェーンに組み込まれたView Controllerの使用”（78 ページ）を参照してください。

View Controllersはほかのコントローラと連携して動作する

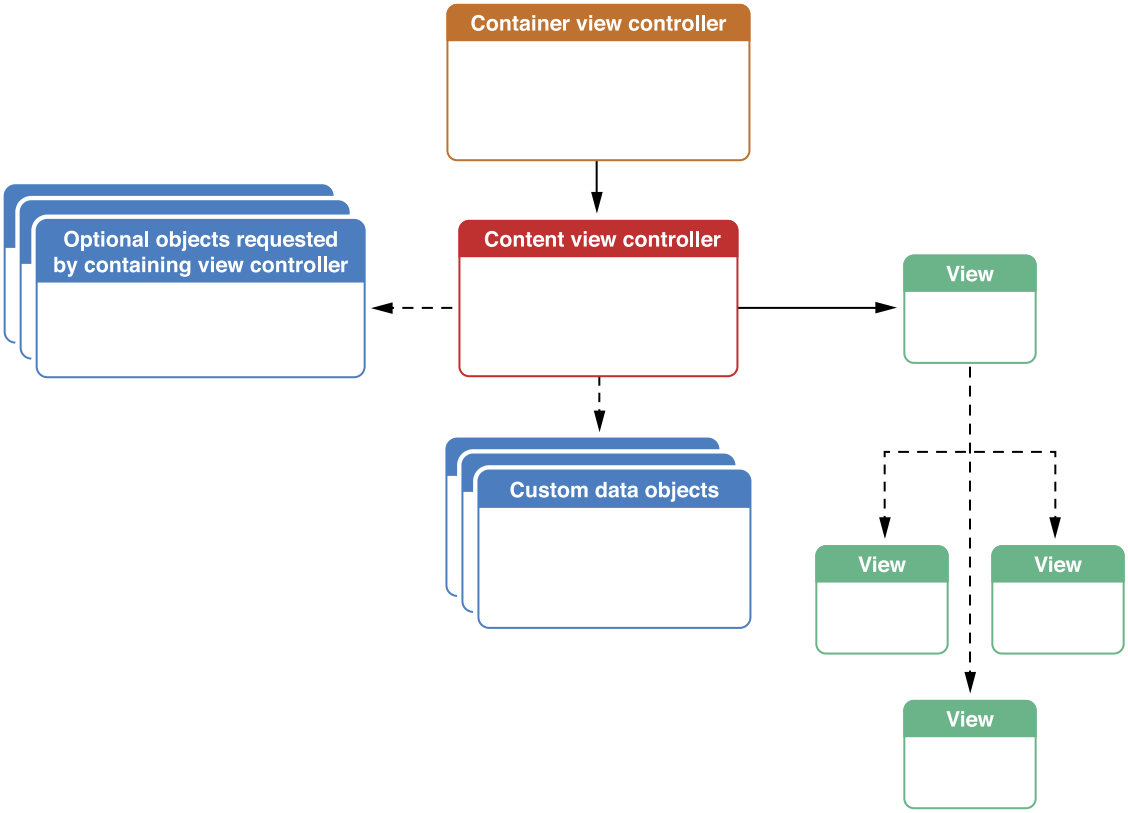
View Controllerはほかのオブジェクトを数多く生成、管理しますが、通常、こういったオブジェクトを公開して、外部から情報を得たり修正を施したりできるようにする必要はありません。ほかのオブジェクト（特にほかのView Controller）と連携することはありますが、その相手と通信するために必要な、ごく少数のプロパティやメソッドを公開するだけです。View Controllerクラスの実装詳細を必要以上に公開すると、後でその実装を変更するのが難しくなります。連携の相手も、View Controllerクラスの実装詳細に依存していると、それが変更されるごとに、自身も修正を余儀なくされます。

詳しくは“View Controller間の連携”（105 ページ）を参照してください。

View Controllerはしばしばコンテナと連携する

View ControllerをContainer View Controller内に配置すると、コンテナはView COntrollerに対して、新たに制約を課すことになります（図 3-2を参照）。すなわち、コンテナ自身のユーザインターフェイスを設定するために用いる、何らかのオブジェクトを渡すよう要求することがあるのです。たとえばContent View Controllerは、Tab View Controller内に配置された場合、Tab Bar項目を渡し、タブとして表示してもらうことができます。

図 3-2 Container View Controllerは子に対してほかにも要求を行う



UIKitに付属するコンテナの設定に使うプロパティは、`UIViewController`クラスで定義されています。各種のコンテナ、およびその設定に用いるプロパティについては、『*View Controller Catalog for iOS*』を参照してください。

View ControllerはほかのView Controllerから表示される可能性がある

View Controllerの中には、ほかのView Controllerから表示されることを前提としたものがあります。当該View Controllerを直接表示するだけでなく、これをContainer View Controllerの子として設定し、Containerの方を表示するというやり方もあります。表示されたView Controllerは、画面上の位置を変え、破棄されるまでの間その位置にとどまります。

View Controllerを表示する目的はいくつかあります。

- ユーザから直ちに情報を収集するため。
- 何らかのコンテンツを一時的に表示するため。
- 作業モードを一時的に変更するため。
- デバイスの向きに応じて代替のインターフェイスを実装するため。
- 特殊なアニメーショントランジションを使用する（またはトランジションなしの）新しいビュー階層を表示するため。

これらの理由のほとんどでは、何らかの情報を収集したり表示するために、アプリケーションのワークフローに一時的に割り込みを行うことを意味しています。ほとんどの場合、被表示View Controller側にデリゲートを実装します。被表示View Controllerはデリゲートを使って、表示した側のView Controllerと通信します。アプリケーションが必要な情報を得る（あるいは表示された情報をユーザが見終わる）と、被表示View Controllerはその旨の通知を表示した側のView Controllerに返します。表示した側のView Controllerは、被表示View Controllerを破棄して、アプリケーションを直前の状態に戻します。

詳しくは"View ControllerをほかのView Controllerから「表示」"（95 ページ）を参照してください。

Content View Controllerの設計

View Controllerのコードを記述する前に、意図する使い方について、基本的な問いに答えられるようにしておかなければなりません。以下に挙げる事項を検討することにより、設計上の選択肢を絞り込み、アプリケーションにおいて果たす役割を明確にすることができるでしょう。特に、タスクを実行するために、どのような接続（一般にほかのView Controllerとの接続）が必要かがわかります。

- View Controllerの実装にストーリーボードを使うか？
- どの時点でインスタンスを生成するか？
- 何のデータを表示するか？、
- どのようなタスクを実行するか？
- そのビューが画面にどのように表示されるか？
- ほかのView Controllerとどのように連携するか？

View Controllerが果たす役割を検討している段階であれば、きちんと答えられなくても構いません。しかしこのように整理して考えれば、View Controllerが何をし、ほかのオブジェクトがこれとどのようにやり取りするか、大まかに把握するために役立つでしょう。

上記の問いは、View Controllerの外観を定義するよう求めているのでも、これに割り当てるタスクの実行方法について、実装詳細を厳密に決めるよう求めているのでもありません。もちろんこれも重要な問いではありますが、いずれもView Controllerの公開インターフェイスに影響を与えるべきではないのです。View Controllerの外観の設計は、クラス宣言、すなわちほかのコントローラがこれと連携する方法を変えることなく、柔軟に変更できるようでなければなりません。

ストーリーボードを使ってView Controllerを実装

ストーリーボードを使うか否かは、実装詳細に属する事項と考えられるかも知れませんが、View Controllerの実装方法や、ほかのオブジェクトがこれと連携する方法に、大きな影響を与えます。どうしても使えない理由がない限り、ストーリーボードを使ってください。

ストーリーボードには次のような利点があります。

- View Controllerのインスタンスは、通常、iOSが自動的に生成します。
- インスタンスが生成された直後に何らかの処理が必要な場合は、`awakeFromNib`メソッドをオーバーライドしてください。
- ほかのオブジェクトは、プロパティを介して必要な設定を施します。
- ビュー階層やほかの関連オブジェクトをInterface Builder上で作成できます。このオブジェクトは、必要になった時点で自動的にロードされます。
- ほかのView Controllerとの関係をストーリーボード上で作成できます。

View Controllerをプログラムで生成する場合は次のようになります。

- View Controllerのインスタンスを生成するためには、割り当てと初期化の処理が必要です。
- View Controllerを初期化するために、独自の初期化メソッドを作成することになります。
- ほかのオブジェクトがView Controllerに設定を施す場合、初期化メソッドやプロパティを使います。
- `loadView`メソッドをオーバーライドすることにより、プログラムでビュー階層を生成、設定します。
- ほかのView Controllerとの関係も、コードを記述することにより作成します。

Controllerのインスタンスがいつ生成されるかの把握

どの時点でView Controllerのインスタンスを生成するか、が決まれば、アプリケーションの処理方法など、詳細も自然に決まります。たとえば、View Controllerのインスタンスが、常に同じオブジェクトに生成されることが分かっているとしましょう。このオブジェクトは多くの場合、それ自身がView

Controllerです。ストーリーボードを使うアプリケーションの多くがこれに該当します。いずれにしても、どの時点で、どんな理由で、どのオブジェクトによって生成されるか、が決まれば、当該View Controllerと、その生成元オブジェクトとの間でやり取りする情報の詳細が分かってきます。

View Controllerが表示し、返すデータの把握

この2つを検討する過程で、アプリケーションのデータモデルを把握しながら、View Controller間でこのデータをやり取りする必要があるかどうか、も考えることになります。

View Controllerにどのような想定をするか、よく見られるパターンをいくつか示します。

- View Controllerはほかのコントローラからデータを受け取って表示するけれども、編集する手段は与えないパターン。データは何も返しません。
- View Controllerがユーザに新しいデータを入力させるパターン。データの編集終了後、新しいデータを別のコントローラに送ります。
- View Controllerが別のコントローラからデータを受け取り、ユーザが編集できるようにするパターン。データの編集終了後、新しいデータを別のコントローラに送ります。
- View Controllerがデータの送信も受信もしないパターン。代わりに静的なビューを表示します。
- View Controllerがデータの送信も受信もしないパターン。代わりに、必要なデータをロードしますが、ほかのView Controllerにこの機構を公開しません。たとえば`GKAchievementViewController`クラスには、どのプレイヤにデバイスを使う権限があるかを判断する機能が組み込まれています。また、プレイヤのデータをGame Centerからロードする方法も知っています。表示する側のView Controllerが、何のデータがどのようにロードされるか、を知っている必要はありません。

実際には、上記の設計パターンにとらわれる必要はありません。

データがView Controllerを出入りする場合、新しいコントローラに配送されるデータを保持する、データモデルのクラスを定義することも検討してください。たとえば『*Your Second iOS App: Storyboards*』では、鳥観アプリケーションのマスタコントローラが`BirdSighting`オブジェクトを使って、目撃データを詳細コントローラに送信します。データ保持用のオブジェクトを使うと、データを更新して追加のプロパティを入れるのが容易になります。コントローラクラスのメソッドのシグニチャを変更することもありません。

Controllerがユーザに許可する作業の把握

View Controllerの中には、ユーザがデータを表示、作成、編集することを許可するものがあります。ほかのコンテンツ画面にナビゲートできるようにしているものもあります。さらに、View Controllerが提供するタスクを実行できるようになっている場合もあります。たとえば

MFMailComposeViewControllerクラスは、ユーザがほかのユーザに、電子メールを作って*送信*できるようになっています。このクラスには電子メールメッセージの送信に関する低レベルの機能が組み込まれているのです。

View Controllerが実行するタスクを決めながら、当該タスクの制御機能を、ほかのコントローラにどの程度公開するか、も決めなければなりません。View Controllerの多くは、設定データをほかのコントローラに公開することなく、そのタスクを実行できます。たとえばGKAchievementViewControllerクラスは、成績をユーザに向けて表示しますが、データをロードし表示する方法を設定するためのプロパティは何も公開しません。一方、MFMailComposeViewControllerクラスは若干異なります。最初に表示するコンテンツをほかのコントローラが設定できるよう、プロパティをいくつか公開しているのです。その後、ユーザはコンテンツを編集し、電子メールメッセージを送信できます。ほかのコントローラオブジェクトはこの手順に干渉できません。

View Controllerが画面上にどのように表示されるかの把握

View Controllerの中には、ルートView Controllerとして設計されているものがあります。それ以外のView Controllerは、ほかのView Controllerに表示されるか、またはContainer Controller内に置かれるものと想定しています。場合によっては、いくつもの方法で表示できるコントローラを設計することもあるでしょう。たとえばSplit View Controllerのマスタビューは、Split Viewには横長モード、Popoverコントロールには縦長モードで表示されます。

View Controllerはどのように表示されるか、が分かれば、画面上の大きさや位置をどうするかも決まります。さらに、どのView Controllerと連携するかなど、さまざまな点に影響します。

Controllerが他のControllerとどのように連携するかの把握

ここまで進んでくれば、どのように連携するか、についてもすでに分かっているでしょう。たとえば、View Controllerのインスタンスをセグエから生成するのであれば、おそらく自分自身に設定を施すソースView Controllerと連携するはずです。コンテナの子であれば、コンテナ（親）と連携するはずです。しかしこの場合、逆向きの関係もあります。たとえばView Controllerは、処理の一部を保留にして、ほかのView Controllerに委ねるかも知れません。あるいは、既存のView Controllerとデータを交換することもありえます。

View Controllerは、以上のような接続すべてを用いて、ほかのコントローラが使うインターフェイスを提供し、あるいは、ほかのコントローラを知っていてそのインターフェイスを使います。これらの接続は、シームレスな動作を実現するために重要である一方、アプリケーションのクラス間に依存関係が生じる、という意味では設計における難題でもあります。依存関係が問題なのは、アプリケーションを構成するほかのクラスと切り離して、あるクラスを変更するのが難しくなるからです。そのため、後でいつでも変更できるよう適応性の高いアプリケーション設計を保ちたい、という要請とうまく両立させる必要があります。

一般的なView Controllerの設計例

新しいView Controllerの設計にはさまざまな困難が待ち構えています。既存の設計を調べ、何をしているか、それはなぜかを理解すれば、今後の設計にも役立つでしょう。以下、iOSアプリケーションで広く使われているスタイルのView Controllerをいくつか取り上げます。それぞれ、View Controllerが担う役割、その大雑把な動作原理の説明、そして先に挙げた問いに対する答えを示します。

例：ゲームのタイトル画面

目標

ゲームの遊び方をユーザが選択できるView Controller。

Description

ゲームを起動しても、すぐにゲーム本体が始まることはほとんどありません。通常はタイトル画面が現れ、何種類かの遊び方から選択するようになっています。たとえば、1人モードと対戦モードの選択ボタンがあるかも知れません。ユーザがいずれかを選択すると、アプリケーションは自分自身を適切に設定した後、ゲーム画面を起動します。

タイトル画面は、コンテンツが静的である、すなわちほかのコントローラからデータを取得する必要がない、という点で興味深いものです。このような性質のため、ほぼ自己完結しています。それでもほかのView Controllerについてまったく知らずに済むわけではありません。ゲーム画面を起動するためには、ほかのView Controllerのインスタンスを生成する必要があるのです。

デザイン/設計

- **View Controllerの実装にストーリーボードを使うか？**はい。
- **どの時点でインスタンスを生成するか？**このView Controllerは、メインストーリーボードに属する初期シーンです。
- **何のデータを表示するか？**設定済みのコントロールと画像を表示します。ユーザデータはありません。また、設定可能なプロパティもありません。
- **どのようなタスクを実行するか？**ユーザがボタンをタップすると、セグエにトリガをかけて、ほかのView Controllerのインスタンスを生成します。各セグエを識別して、該当するゲーム画面に遷移できるようにする必要があります。
- **そのビューが画面にどのように表示されるか？**ウインドウのルートView Controllerなので、自動的にインストールされます。

- **ほかのView Controllerとどのように連携するか？** ほかのView Controllerのインスタンスを生成して、ゲーム画面を表示します。ゲームが終了すると、ゲーム画面に対応するView Controllerは、タイトル画面のコントローラにその旨のメッセージを送信します。タイトル画面コントローラはこれを受けて、ゲーム画面に対応するView Controllerを破棄します。

その他考えられる設計方針

以上の答えは、ユーザデータを表示しないと想定したものです。場合によっては、プレイヤのデータをもとに、ビューやコントロールを設定するかも知れません。以下に例を挙げます。

- View Controllerに、Game Centerに登録されている、ユーザのニックネームを表示させる。
- デバイスがGame Centerに接続されているかどうかに応じて、ボタンの有効/無効を切り替える。
- ユーザが購入したIn-App Purchaseの商品に応じて、ボタンの有効/無効を切り替える。

こういった事項を設計に加味すると、View Controllerはよりオーソドックスな役割を担うことになります。データオブジェクトやデータコントローラをアプリケーションデリゲートから受け取り、必要に応じてその状態を問い合わせたり、更新したりするかも知れません。あるいは、これはウインドウのルートView Controllerなので、タイトル画面コントローラに直接この動作を実装することも考えられます。実際の設計は、コードにどの程度の適応性が必要か、にも依存するでしょう。

例：マスタView Controller

目標

Navigation Controllerの初期View Controller。ここに、アプリケーションで利用できるデータオブジェクトのリストを表示します。

Description

マスタView Controllerは、ナビゲーションベースのアプリケーションでは非常によく使われます。たとえば『*Your Second iOS App: Storyboards*』では、鳥観アプリケーションのマスタビューに目撃データをリスト表示しています。ユーザがリストから目撃データを選択すると、マスタView Controllerは新しい詳細コントローラを画面にプッシュします。

このView Controllerは項目のリスト表示に使うので、`UIViewController`ではなく`UITableViewController`のサブクラスを定義して使います。

デザイン/設計

- **View Controllerの実装にストーリーボードを使うか？** はい。

- **どの時点でインスタンスを生成するか？**Navigation ControllerのルートView Controllerなので、親と同時にインスタンスが生成されます。
- **何のデータを表示するか？**アプリケーションデータの高レベルビュー。アプリケーションデリゲートがデータを渡すために使うプロパティを実装します。たとえば鳥観アプリケーションの場合、カスタムデータコントローラオブジェクトをマスタView Controllerに渡します。
- **どのようなタスクを実行するか？**「Add」ボタンを実装して、ユーザが新規レコードを作成できるようにします。
- **そのビューが画面にどのように表示されるか？**Navigation Controllerの子です。
- **ほかのView Controllerとどのように連携するか？**リスト上のある項目をユーザがタップしたとき、プッシュセグエを使って詳細コントローラを表示します。また、「Add」ボタンをユーザがタップしたとき、モーダルセグエを使って新しいView Controllerを表示し、新規レコードを編集できるようにします。このモーダルView Controllerからデータを受け取り、これをデータコントローラに送って、新しい鳥観レコードを作成します。

その他考えられる設計方針

iPhoneアプリケーションを構築する際には、Navigation Controllerと初期View Controllerを使います。同じアプリケーションをiPad用に設計する場合、マスタView ControllerはSplit View Controllerの子にしなければなりません。それ以外の設計はほぼ同じです。

例：詳細View Controller

目標

Navigationスタックにプッシュされるコントローラで、マスタView Controllerで選択されたリスト項目の詳細を表示します。

Description

詳細View Controllerは、マスタView Controllerが表示したリスト項目の、より詳しいビューを表示します。マスタView Controllerと同様、リストはNavigation Barインターフェイス内に現れます。ユーザが詳細ビューを見終わってNavigation Bar上のボタンをクリックすると、マスタビューに戻ります。

『*Your Second iOS App: Storyboards*』では`UITableViewController`クラスを使って詳細ビューを実装しています。静的なテーブルセルを使い、それぞれが1件の鳥観データにアクセスします。この設計を実装するためには静的なテーブルビューが適しています。

デザイン/設計

- **View Controllerの実装にストーリーボードを使うか？**はい。

- **どの時点でインスタンスを生成するか？**マスタView Controllerからのプッシュセグエがインスタンスを生成します。
- **何のデータを表示するか？**カスタムデータオブジェクトに格納されたデータを表示します。ソースView Controllerがデータを渡すためのプロパティが宣言されています。
- **どのようなタスクを実行するか？**ユーザがビューを破棄できるようにします。
- **そのビューが画面にどのように表示されるか？**Navigation Controllerの子です。
- **ほかのView Controllerとどのように連携するか？**データをマスタView Controllerから受け取ります。

その他考えられる設計方針

Navigation Controllerは、iPhoneアプリケーションを構築する際、もっともよく使われます。同じアプリケーションをiPad用に設計する場合、詳細View ControllerはSplit View Controllerの子にしなければなりません。それ以外の実装詳細はほぼ同じです。

ほかにもカスタムビューの動作を実装する必要がある場合は、`UIViewController`のサブクラスを定義し、独自のカスタムビュー階層を実装してください。

例：メール作成View Controller

目標

ユーザが電子メールを作成、送信できるView Controller。

Description

MessageUIフレームワークに`MFMailComposeViewController`クラスがあります。このクラスを使うと、ユーザが電子メールを作成、送信できるようになります。このView Controllerが興味深いのは、単なるデータの表示や編集を越えて、電子メールの送信という処理を行うことです。

このクラスに関してもうひとつ興味深い設計上の選択は、アプリケーションが電子メールメッセージの初期値を設定できるようになっていることです。初期設定を示した後、ユーザはこれを必要に応じて書き換えた上で送信できます。

デザイン/設計

- **View Controllerの実装にストーリーボードを使うか？** No.
- **どの時点でインスタンスを生成するか？**プログラムでインスタンスを生成します。

- **何のデータを表示するか？**電子メールメッセージを構成する各部分（送信先リスト、題名、添付書類、メッセージ本体）を表示します。ほかのView Controllerが事前に電子メールメッセージを設定できるよう、クラスにはいくつかプロパティがあります。
- **どのようなタスクを実行するか？**電子メールを送信します。
- **そのビューが画面にどのように表示されるか？**View ControllerはほかのView Controllerにより表示されます。
- **ほかのView Controllerとどのように連携するか？**ステータス情報をデリゲートに返します。これにより表示View Controllerは、電子メールが正常に送信されたか否かを知ることができます。

カスタムContent View Controllerの実装チェックリスト

カスタムContent View Controllerを作成する場合、View Controllerで処理できるようにしなければならないタスクがいくつかあります。

- ビューはView Controllerがロードするように設定しなければなりません。

  カスタムクラスでは、何らかのメソッドをオーバーライドして、ビュー階層をロード/アンロードする方法を管理しなければならないかも知れません。同じメソッドで、同時に作成されたほかのリソースも管理します。詳しくは“View Controllerにおけるリソース管理”（60 ページ）を参照してください。
- View Controllerがサポートするデバイスの向きと、向きが変わったときの反応を決める必要があります（“デバイスの向きに応じた調整”（80 ページ）を参照）。

View Controllerを実装するときには、アクションメソッドや、そのビューで使用するアウトレットを定義する必要があるかも知れません。たとえば、ビュー階層にテーブルが含まれている場合は、そのテーブルへのポインタをアウトレットに格納して、後でそれにアクセスできるようにします。同様に、ビュー階層にボタンやその他のコントロールが含まれている場合は、それらのコントロールから、View Controllerの対応するアクションメソッドを呼び出すようにします。View Controllerクラスの定義の繰り返しになりますが、次のような項目をView Controllerクラスに追加する必要があることに気付いたかも知れません。

- 対応するビューによって表示されるデータを含むオブジェクトを指す宣言済みプロパティ
- View Controller独自の振る舞いをほかのView Controllerに公開するための、公開メソッドや公開プロパティ
- View Controllerがやり取りしなければならない、ビュー階層内のビューを指すアウトレット
- ビュー階層内のボタンやその他のコントロールに関連付けられたタスクを実行するアクションメソッド

Important: View Controllerクラスのクライアントが、どのビューを当該View Controllerが表示するか、あるいはビューがどのようなアクションのトリガをかけるか、を知っている必要はありません。アウトレットやアクションは、可能な限りクラスの実装ファイルカテゴリ内に宣言するべきです。たとえばクラス名が`MyViewController`であれば、次のような宣言を`MyViewController.m`に追加することにより、カテゴリを実装します。

```
@interface MyViewController()
// アウトレットやアクションをここに記述。
@end
@implementation MyViewController
// 非公開で宣言したカテゴリの実装をここに記述。
@end
```

名前なしのカテゴリを宣言する場合、プロパティやアクションは、公開インターフェイスに宣言したメソッドやプロパティと同じ実装ブロックに実装しなければなりません。非公開カテゴリに定義したアウトレットやアクションは、Interface Builderからは見えますが、アプリケーションのほかのクラスからはアクセスできません。この方法を採ると、Interface Builderの長所を活かしつつ、クラス内に隠蔽するべき事項を公開しないでおくことができます。

View Controllerの機能をほかのクラスが利用する必要があれば、そのための公開メソッドやプロパティを追加してください。

View Controllerにおけるリソース管理

View Controllerは、アプリケーションのリソースを管理する上で重要な部分です。View Controllerを活用すれば、アプリケーションをいくつかに分割し、必要な部分のみインスタンス化する、という方式が取れます。しかしそれだけではなく、View Controller自身がさまざまなリソースを管理し、それぞれ適当な時点でインスタンス化するようになっています。たとえば、View Controllerのビュー階層がインスタンス化されるのは、そこに含まれるビューがアクセスされた時点です。一般に、これが起こるのは、ビューが画面に表示されているときに限ります。複数のView Controllerを同時にナビゲーションスタックにプッシュしても、画面上に見えるのは最上位にあるView Controllerのコンテンツだけであり、したがってそのビューしかアクセスされないことになります。同様に、あるView ControllerをNavigation Controllerが表示していない場合も、該当するナビゲーション項目をインスタンス化する必要はありません。View Controllerは、リソースの割り当て処理の多くを、実際に必要になった時点で行うようにして、リソースの消費量を抑制しています。

アプリケーションの実行に必要なメモリが枯渇しそうになると、View Controllerすべてに、自動的に通知が届きます。View Controllerはこの通知に応じて、キャッシュその他のオブジェクト（後でメモリを確保できれば簡単に生成し直せるもの）を破棄します。細かい動作はアプリケーションが稼働するiOSの版によって異なり、これがView Controllerの設計にも影響を及ぼします。

アプリケーションを効率良く実行するためには、View Controllerに関連するリソースを慎重に管理することが重要です。また、できるだけ遅延割り当てすることも重要です。生成や維持の負荷が高いオブジェクトは、必要になった時点で生成するようにしてください。したがって、View Controllerの生存期間を通して必要なオブジェクトと、一時的に必要なだけのオブジェクトを区別してください。View Controllerは、メモリ不足の警告を受け取ったとき、画面に表示していなければメモリ消費量を抑制できるよう準備しておかなければなりません。

View Controllerを初期化する

View Controllerのインスタンスを最初に生成する際には、その生存期間を通して必要になるオブジェクトを生成またはロードします。表示するコンテンツに関連するビュー階層やオブジェクトは生成しません。この段階では、データオブジェクトや、重要な振る舞いを実装するために必要なオブジェクトを用意することに集中してください。

ストーリーボードからロードしたView Controllerを初期化する

ストーリーボードにView Controllerを作成する際、Interface Builderで設定した属性は直列化してアーカイブに格納されます。後でView Controllerのインスタンスを生成する際に、このアーカイブをメモリ上にロードして処理することになります。その結果、Interface Builder上で設定したのと同じ属性のオブジェクト群が生成されるのです。アーカイブの読み込みには、View Controllerの`initWithCoder:`メソッドを使います。次いで、このメソッドを実装しているオブジェクトの、`awakeFromNib`メソッドを実行します。関係するほかのオブジェクトのインスタンスが生成済みであることを前提とする設定処理は、このメソッド内で実行してください。

アーカイブについて詳しくは、『*Archives and Serializations Programming Guide*』を参照してください。

View Controllerをプログラムで初期化する

View Controllerがリソースをプログラムで確保する場合、当該View Controller用のカスタム初期化メソッドを作成する必要があります。このメソッドは、スーパークラスの`init`メソッドを呼び出した後、当該クラスに特有の初期化処理を実行します。

一般に、初期化メソッドはあまり複雑なものにしないでください。単純な処理のみを実装しておき、それとは別に適当なプロパティを用意して、View Controllerを利用する側が動作を調整できるようにするとよいでしょう。

View Controllerは対応するビューがアクセスされた時点でビュー階層をインスタンス化する

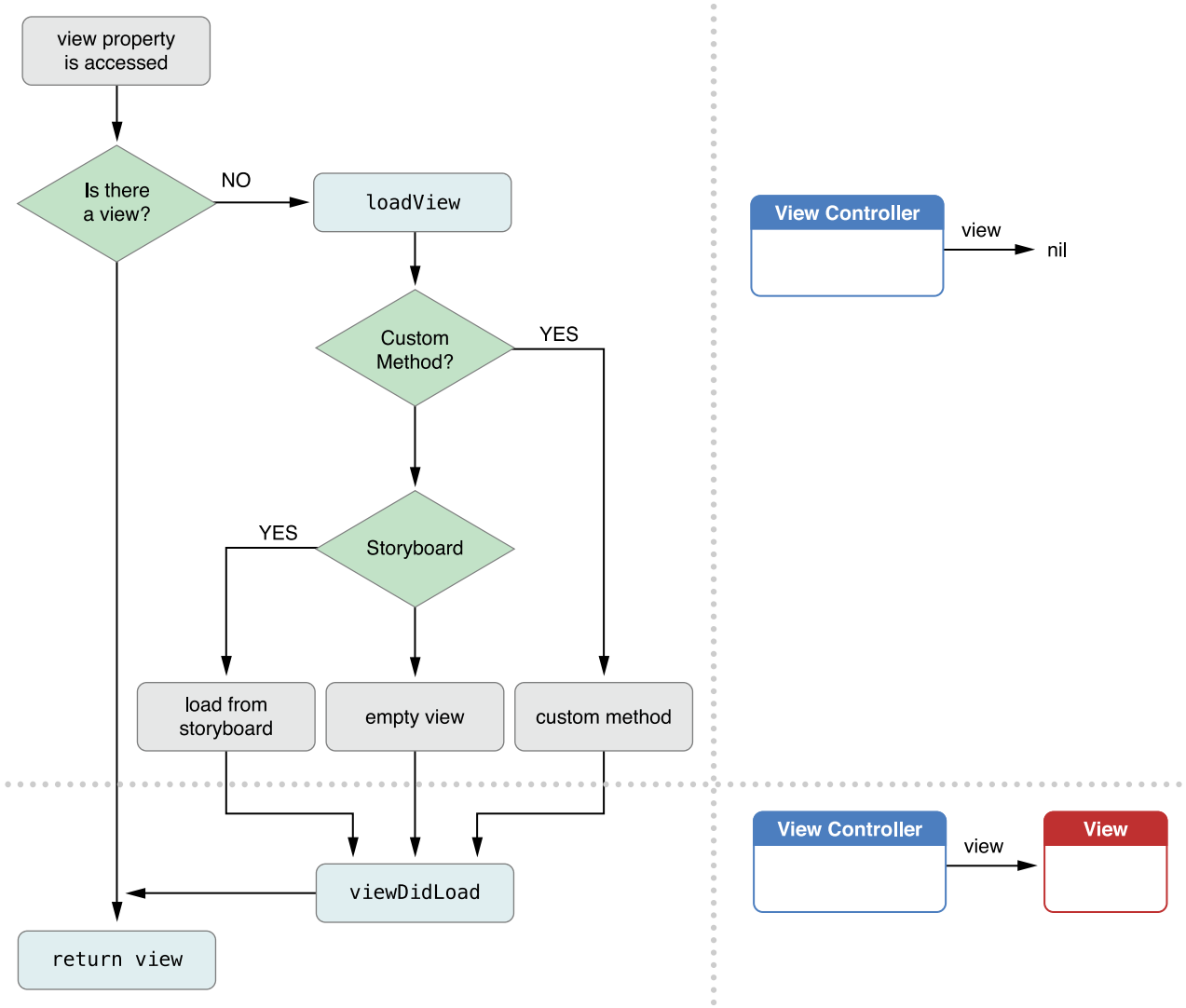
View Controllerは、アプリケーションのある処理のためにビューオブジェクトを要求され、それがメモリ上になかった場合、ビュー階層をロードしてメモリ上に展開するとともに、今後いつでも参照できるよう`view`プロパティに格納します。ロードサイクルでは、次のようなステップが発生します。

1. View Controllerは自身の`loadView`メソッドを呼び出します。`loadView`メソッドのデフォルト実装では、次のいずれかの処理を行います。
    - View Controllerがストーリーボードに関連付けられていれば、ストーリーボードからビューをロードします。
    - そうでなければ、空の`UIView`オブジェクトを生成し、`view`プロパティに代入します。
2. View Controllerは`viewDidLoad`メソッドを呼び出して、ロード時に実行すべきほかの処理を、サブクラスで実行できるようにします。

図 4-1は、ロードサイクルを図示したものです。この図には、ロードサイクルで呼び出されるいくつかのメソッドも含まれています。必要であれば、アプリケーションで`loadView`メソッドと`viewDidLoad`メソッドの両方をオーバーライドして、View Controllerに希望の動作を追加することもできます。た

とえば、ストーリーボードを使っていないアプリケーションで、ビュー階層にビューを追加したい場合、`loadView`メソッドをオーバーライドし、プログラムで当該ビューのインスタンスを生成することになります。

図 4-1 ビューをメモリにロードする



ストーリーボードからView Controllerのビューをロードする

ほとんどのView Controllerは、それに対応するストーリーボードからビューをロードします。ストーリーボードを使用する利点は、ビューを視覚的に配置したり設定したりできるため、レイアウトの調整がすばやく簡単にできることです。いくつかの候補を次々と切り替えながら見比べて、最も洗練されたユーザインターフェイスを選択することも可能です。

Interface Builderでのビューの作成

Interface BuilderはXcodeの一部であり、View Controller用のビューを作成したり設定したりするための直感的な方法を提供します。Interface Builderを使用して、ビューとコントロールを直接操作してワークスペースにドラッグし、位置を決めたり、サイズを変更したり、インスペクタウインドウを使用してそれらの属性を変更したりすることによって、ビューとコントロールを組み立てることができます。そして、その結果はストーリーボードファイルに保存されます。ストーリーボードファイルには、デベロッパが組み立てたオブジェクトの集合が、それに対するすべてのカスタマイズ情報と一緒に保存されます。

Interface Builderでビューの表示属性を設定する

ビューのコンテンツを適切にレイアウトできるように、Interface Builderが提供するコントロールの中には、ビューがNavigation Bar、ToolBar、その他、カスタムコンテンツの位置に影響を与えるオブジェクトを持つかどうかを指定できるものがあります。ストーリーボードで、あるControllerがContainer Controllerに接続されている場合、その設定をコンテナから推論することにより、実行時にどのように表示されるべきか、容易に知ることができます。

View Controllerのアクションとアウトレットを設定する

Interface Builderを使用して、インターフェイス内のビューとView Controllerを接続します。

リスト4-1に、2つのカスタムアウトレット（`IBOutlet`キーワードで指定されている）と、1つのアクションメソッド（`IBAction`という戻り値型で指定されている）を定義するカスタムの`MyViewController`クラスの宣言を示します。宣言は実装ファイルのカテゴリ内に記述します。これらのアウトレットは、ストーリーボードファイル内の1つのボタンと1つのテキストフィールドへの参照を格納します。一方、アクションメソッドはそのボタンのタップに応答します。

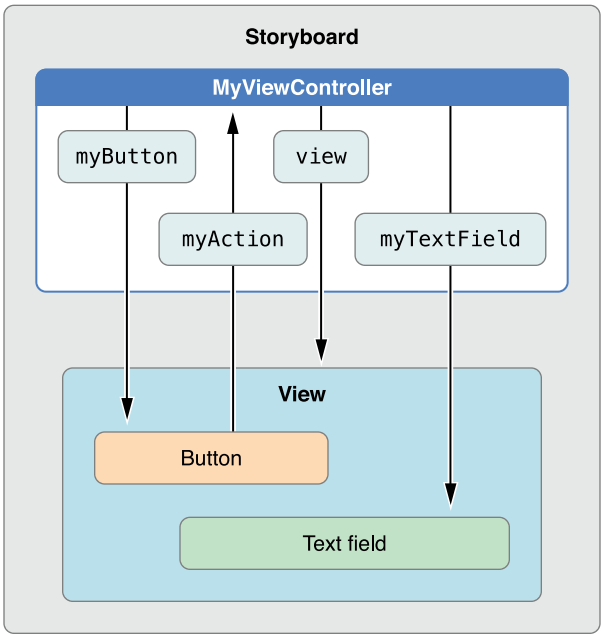
リスト 4-1　カスタムView Controllerクラスの宣言

```
@interface MyViewController()
@property (nonatomic) IBOutlet id myButton;
@property (nonatomic) IBOutlet id myTextField;

- (IBAction)myAction:(id)sender;
@end
```

図 4-2に、このようなMyViewControllerクラス内のオブジェクト間に作成する接続を示します。

図 4-2 ストーリーボードにおける接続



前述のように設定したMyViewControllerクラスが作成されて表示されると、View Controllerのインフラストラクチャがストーリーボードからビューを自動的にロードし、アウトレットやアクションを再構成します。このように、ビューがユーザに表示されるまでには、View Controllerのアウトレットとアクションが設定されて使用できる状態になります。実行時のコードと設計時のリソースファイルをこのように連動させられることが、ストーリーボードの強みの1つです。

プログラムによるビューの作成

ストーリーボードを使用せず、プログラムによってビューを作成したい場合は、View Controllerの`loadView`メソッドをオーバーライドします。このメソッドの実装では、次の処理を実行しなければなりません。

1. ルートビューオブジェクトを作成します。

   このルートビューは、View Controllerに関連付けられたその他のすべてのビューを含みます。通常は、このビューに対して、アプリケーションウインドウ（画面一杯に表示される）のサイズに一致するフレームを定義します。しかしフレームは、View Controllerがどのように表示されるか、に応じて調整されます。詳しくは“View Controllerのビューの大きさ変更”（74 ページ）を参照してください。

   汎用の`UIView`オブジェクト、独自に定義したカスタムビュー、その他、画面いっぱいに拡張できる任意のビューを使用できます。

2. 追加のサブビューを作成し、それらをルートビューに追加します。

   ビューごとに、次の処理を実行しなければなりません。

   a. ビューを作成して初期化します。

   b. `addSubview:`メソッドを使用して、そのビューを親ビューに追加します。

3. 自動レイアウト機能を使うのであれば、生成した各ビューの位置や大きさを適切に制御できるよう、充分な制約条件を与えてください。そうでなければ、`viewWillLayoutSubviews`および`viewDidLayoutSubviews`メソッドに、ビュー階層に属するサブビューの枠を調整する処理を実装します。詳しくは"View Controllerのビューの大きさ変更"（74 ページ）を参照してください。

4. ルートビューをView Controllerの`view`プロパティに代入します。

リスト4-2に`loadView`メソッドの実装例を示します。このメソッドは、ビュー階層にカスタムビューの組を生成し、View Controllerに割り当てます。

リスト 4-2　ビューをプログラムによって作成

```
- (void)loadView
{
    CGRect applicationFrame = [[UIScreen mainScreen] applicationFrame];
    UIView *contentView = [[UIView alloc] initWithFrame:applicationFrame];
    contentView.backgroundColor = [UIColor blackColor];
    self.view = contentView;

    levelView = [[LevelView alloc] initWithFrame:applicationFrame
viewController:self];
    [self.view addSubview:levelView];
}
```

注意: `loadView`メソッドをオーバーライドして、プログラムによってビューを作成するときは、`super`を呼び出してはいけません。これを行うと、ビューをロードするデフォルトの動作が開始されます。通常、これはCPUサイクルを浪費するだけです。`loadView`メソッドの独自実装では、View Controller用のルートビューとサブビューの作成に必要なすべての処理を実行するべきです。ビューのロード処理の詳細については、"View Controllerは対応するビューがアクセスされた時点でビュー階層をインスタンス化する"（61 ページ）を参照してください。

効率的なメモリ管理

View Controllerとメモリ管理と言えば、検討しなければならない問題は次の2つです。

- どのようにしてメモリを効率的に割り当てるか？
- いつ、どのようにメモリを解放するか？

メモリ割り当てについては、完全にデベロッパが判断すべき部分もありますが、`UIViewController`クラスにはメモリ管理タスクに関連するメソッドがいくつかあります。表4-1は、View Controllerオブジェクト内でメモリの割り当てや解放を行う可能性のある場所と、それぞれの場所で実行しなければならない処理のリストです。

表 4-1 メモリの割り当てや解放を行う場所

タスク	メソッド	説明
View Controllerで必要な重要なデータ構造を割り当てる	初期化メソッド	カスタム初期化メソッドには（`init`という名前であるか、それ以外の名前であるかに関わらず）、常にView Controllerオブジェクトを既知の適切な状態に設定する責任があります。これには、適切な操作を保証するために必要なあらゆるデータ構造の割り当てが含まれます。
ビューオブジェクトを作成する	`loadView`	プログラムによってビューを作成する場合にのみ、`loadView`メソッドのオーバーライドが必要になります。ストーリーボードを使っている場合、ビューは自動的に、ストーリーボードファイルからロードされます。

タスク	メソッド	説明
カスタムオブジェクトを生成する	カスタムのプロパティとメソッド	どうしてもと言うわけではありませんが、`loadView`メソッドのような設計パターンを検討してください。すなわち、メソッドを保持するプロパティと、実際にオブジェクトを初期化するメソッドを用意します。プロパティ値を調べ、`nil`であった場合は、これに対応するロードメソッドを実行するのです。
ビューに表示するデータの割り当てやロードを行う	`viewDidLoad`	通常、データオブジェクトは、View Controllerのプロパティを設定することにより与えます。View Controllerが追加のデータオブジェクトを生成する場合は、`viewDidLoad`メソッドをオーバーライドします。このメソッドが呼び出された時点で、ビューオブジェクトの存在と、それが既知の適切な状態にあることが保証されます。
メモリ不足通知に応答する	`didReceiveMemory-Warning`	View Controllerに関連付けられている、重要でないオブジェクトをすべて解放するために、このメソッドを使用します。iOS6の場合、このメソッドで、ビューオブジェクトの参照を解放することもできます。
View Controllerで必要な重要なデータ構造を解放する	`dealloc`	このメソッドをオーバーライドするのは、View Controllerクラスで、最後のクリーンアップ処理を実行するためだけにしてください。インスタンス変数やプロパティに格納されたオブジェクトは、自動的に解放されます。明示的に行う必要はありません。

iOS 6以降、View Controllerは必要な場合、自身のビューをアンロードする

View Controllerは、`view`プロパティが最初にアクセスされた時点でビュー階層をロードし、View Controllerの破棄までメモリ上に保持しておく、という動作がデフォルトになっています。ビューが自分自身を描画するために必要なメモリは、非常に大量になる可能性があります。しかし、ビューがウインドウから切り離されていれば、システムは自動的に、この高価なリソースを解放してしまいます。大多数のビューが用いるそれ以外のメモリは小容量なので、システムが自動的に破棄し、必要に応じてビュー階層を再構築する、という方式を採るには値しません。

ビュー階層は、そのために確保していたメモリが必要になれば、明示的に解放して構いません。これはリスト 4-3のように、`didReceiveMemoryWarning`メソッドをオーバーライドして実装します。まず、スーパークラスの実装を呼び出して、必要なデフォルトの処理を行います。次にView Controllerのリソースを解放します。最後に、View Controllerのビューが画面上に現れているかどうか調べます。

ビューにウインドウが結びついていれば、View Contollerの、ビューやそのサブビューに対する強い参照をすべて解除します。再生成が必要なデータをビューに格納している場合は、あらかじめそのデータを保存してから、当該ビューに対する参照をすべて解除するように実装してください。

リスト 4-3　可視になっていないView Controllerのビューを解放するコード例

```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 不要になった独自リソースを解放するコードをここに追加。
    if ([self.view window] == nil)
    {
        // ビューに格納していたデータのうち、後で必要に
        // なるものを別の場所に保存するコードを追加。

        // ビュー階層に属するビューを指す、他の強い参照を
        // 解除するコードを追加。
        self.view = nil;
    }
```

以後、`view`プロパティがアクセスされれば、初回と同じ手順でビューを再ロードします。

メモリが不足しそうなとき、システムがビューをアンロードすることがある（iOS 5以前）

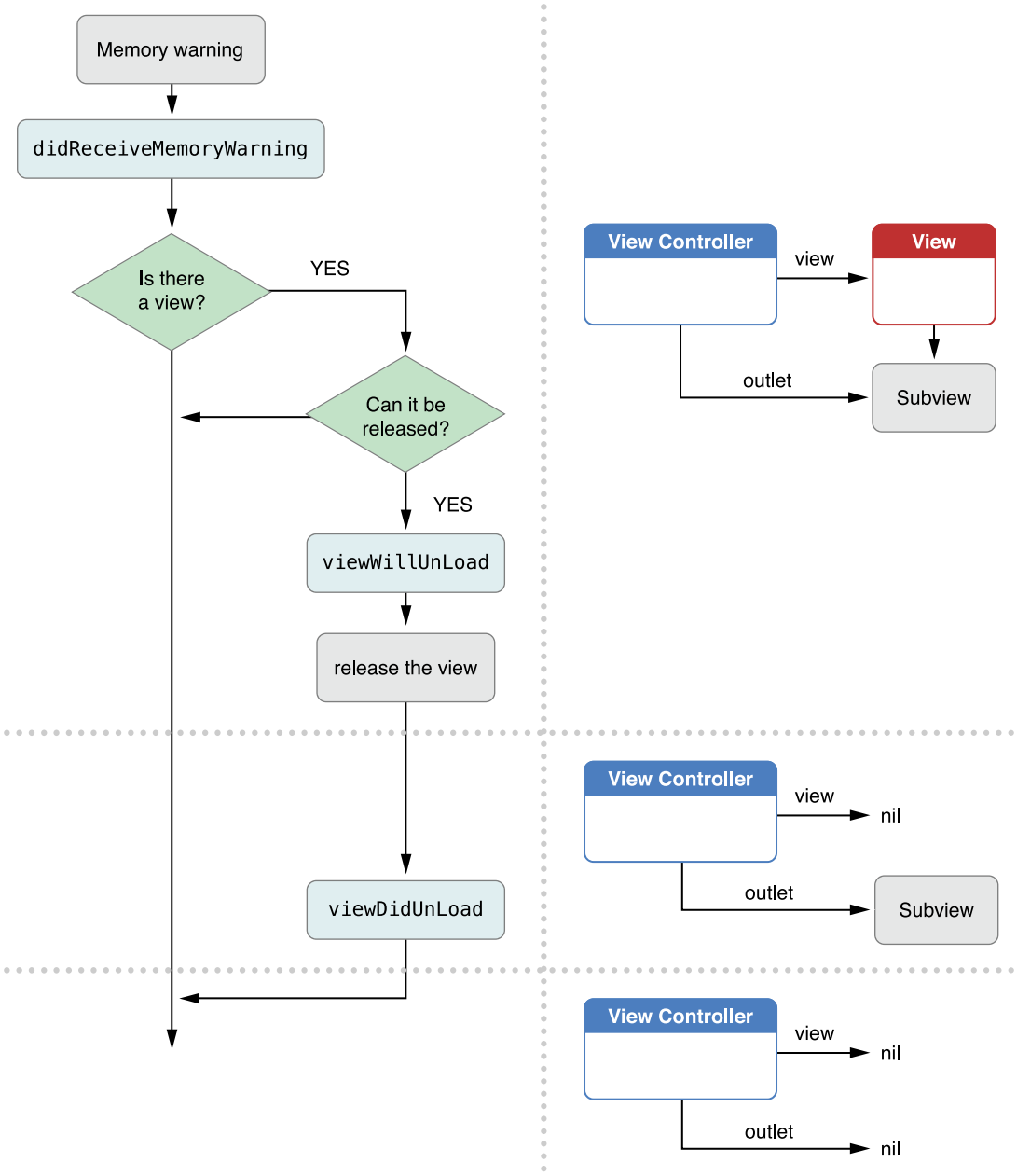
iOSの旧版では、メモリが不足しそうなとき、システムが自動的に、View Controllerのビューをアンロードすることがあります。アンロードサイクルでは、次のようなステップが発生します。

1. アプリケーションがシステムからメモリ不足の警告を受け取ります。
2. 各View Controllerは自身の`didReceiveMemoryWarning`メソッドを呼び出します。このメソッドをオーバーライドする場合は、その中で、View Controllerオブジェクトで不要になったすべてのメモリやオブジェクトを解放します。このメソッドの実装のどこかで`super`を呼び出して、デフォルト実装を実行しなければなりません。iOS 5以前のデフォルトの実装は、ビューを解放するようになっていました。iOS 6以降、そのまま終了する、という実装になりました。
3. ビューを安全に解放できない（たとえばまだ画面上に見えている状態など）場合、デフォルト実装は、何もせずに終了するようになっています。

4. View Controllerは自身の`viewWillUnload`メソッドを呼び出します。サブクラスは一般に、ビューが破棄される前にそのプロパティを保存しておく必要がある場合、このメソッドをオーバーライドします。
5. `view`プロパティを`nil`と設定します。
6. View Controllerは自身の`viewDidUnload`メソッドを呼び出します。通常、サブクラスではこのメソッドをオーバーライドして、当該ビューに対する強い参照を解放します。

図 4-3は、View Controllerのアンロードサイクルを図示したものです。

図 4-3 ビューをメモリからのアンロードする



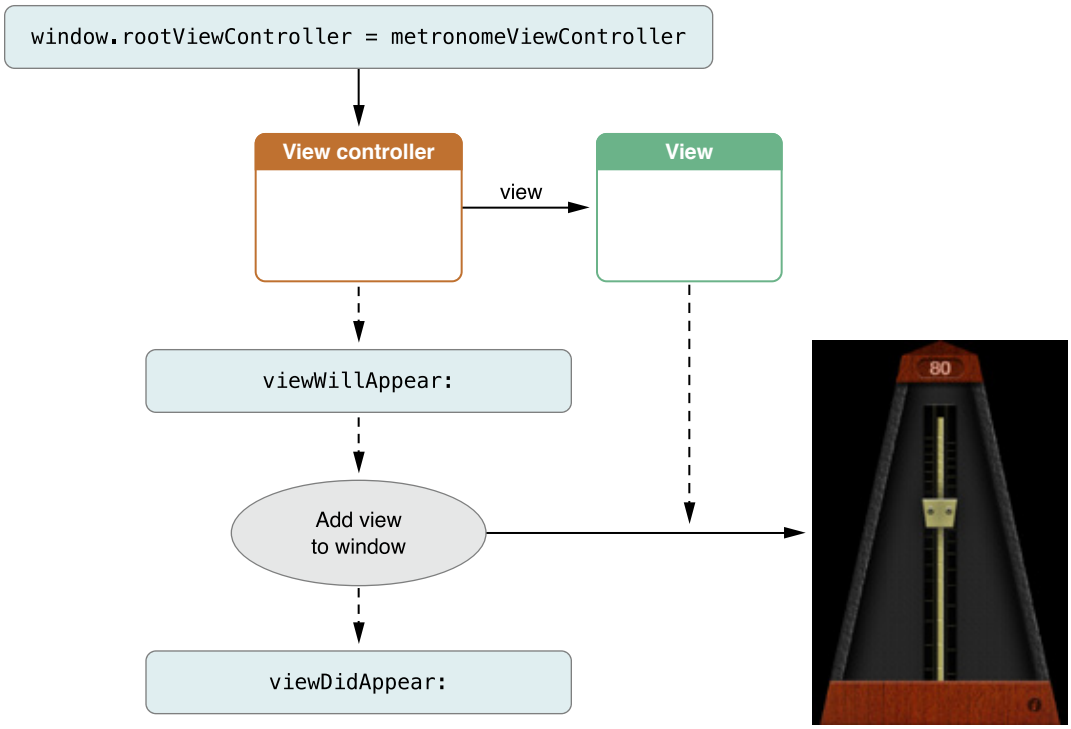
表示関連の通知への応答

View Controllerのビューの外観が変化すると、View Controllerはいくつかの組み込みメソッドを呼び出して、その変化をサブクラスに通知します。これらのメソッドを使用して、変化に対してサブクラスがどのように反応するか、をオーバーライドできます。たとえば、これらの通知を使用して、これから表示されるビューの表示スタイルに合うようにステータスバーの色と向きを変更できます。

ビューの表示に応答する

図 5-1に、View Controllerのビューがウインドウのビュー階層に追加されるときに起きる一連のことの順番を示しますviewWillAppear:メソッドとviewDidAppear:メソッドは、サブクラスに、ビューの表示に関連する追加アクションを実行する機会を与えます。

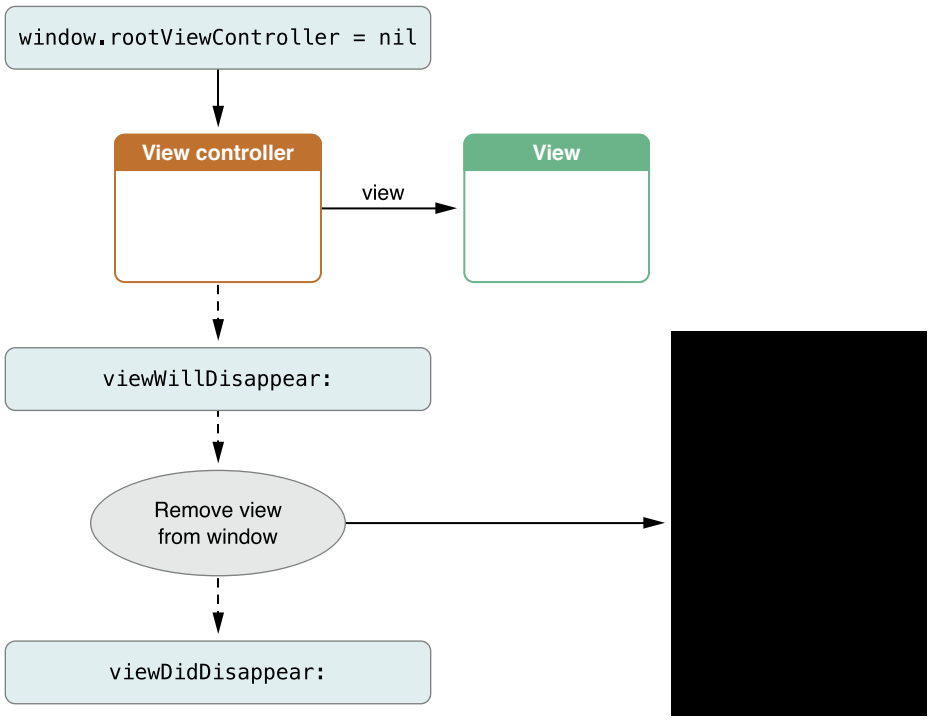
図 5-1 ビューの表示に応答する



ビューの消去に応答する

図5-2に、ビューがウインドウから消去されるときに起きる一連のことの順序を示しますView Controllerは、ビューが消去または隠されようとしていることを検出すると、`viewWillDisappear:`メソッドと`viewDidDisappear:`メソッドを呼び出して、サブクラスに、関連するタスクを実行する機会を与えます。

図 5-2　ビューの消去に応答する



ビューの表示が変化した理由を判断する

ビューが現れた、あるいは消えた理由がわかると役に立つ場合があります。たとえば、ビューが現れたのが、コンテナに追加されたためなのか、これを隠していたほかのビューが削除されたためなのか、知ることができれば有用かも知れません。このような状況は、Navigation Controllerを使っている場合によく起こります。Content Controllerのビューが現れたのは、View Controllerがナビゲーションスタックにプッシュされたからとも、その上に積まれていたコントローラがポップされたからとも考えられるのです。

`UIViewController`クラスには、View Controllerから呼び出して、表示が変化した理由を判断するためのメソッド群があります。表 5-1にその使い方を示します。いずれも、`viewWillAppear:`、`viewDidAppear:`、`viewWillDisappear:`、`viewDidDisappear:`の各メソッドの実装コードから呼び出せます。

表 5-1 ビューの表示が変化した理由を判断するメソッド

メソッド名	使用方法
`isMovingFromParentViewController`	`viewWillDisappear:`メソッドや`viewDidDisappear:`メソッド内から呼び出して、View Controllerのビューが消えようとしている理由が、View Controllerが（その親である）Container View Controllerから削除されたためであるか否か、を判断するために使います。
`isMovingToParentViewController`	`viewWillAppear:`メソッドや`viewDidAppear:`メソッド内から呼び出して、View Controllerのビューが現れようとしている理由が、View ControllerがContainer View Controllerに（子として）追加されたためであるか否か、を判断するために使います。
`isBeingPresented`	`viewWillAppear:`メソッドや`viewDidAppear:`メソッド内から呼び出して、View Controllerのビューが現れようとしている理由が、別のView Controllerにより当該View Controllerが表示されたためであるか否か、を判断するために使います。
`isBeingDismissed`	`viewWillDisappear:`メソッドや`viewDidDisappear:`メソッド内から呼び出して、View Controllerのビューが消えようとしている理由が、View Controllerが削除されたためであるか否か、を判断するために使います。

View Controllerのビューの大きさ変更

View Controllerは自身のビューを持ち、そのコンテンツを管理しています。その一環としてサブビューも管理するのが普通です。しかし多くの場合、ビューの枠をView Controllerが直接設定することはありません。View Controllerのビューがどのように表示されるか、によって決まります。具体的にいうと、ビューを表示するために使われるオブジェクトが設定するのです。ステータスバーの有無など、アプリケーションのほかの条件によっても、枠の表示は変わります。そのためView Controllerは、枠の変化に応じてビューの内容を調整できるようにしておかなければなりません。

ウインドウはそのルートView Controllerのビューの枠を判断する

ウインドウのルートView Controllerに対応するビューの枠は、ウインドウの特性に応じて決まります。ウインドウが設定した枠は、さまざまな要因に基づいて変化します。

- ウインドウ自体の枠
- ステータスバーの表示の有無
- ステータスバーに、他に一時的な情報（電話呼び出し中など）が表示されているか否か
- ユーザインターフェイスの向き（横長/縦長）
- ルートView Controllerの`wantsFullScreenLayout`プロパティの値

ステータスバーがある場合、その奥にビューの一部が隠れてしまわないよう、ビュー自体が小さくなります。つまり、ステータスバーが不透明な場合は、その下に重なるようにコンテンツを表示したり、そのようなコンテンツとやり取りをする方法はありません。ただし、アプリケーションが半透明なステータスバーを表示する場合は、View Controllerの`wantsFullScreenLayout`プロパティを`YES`に設定すれば、画面全体に表示できます。ステータスバーはビューの手前に描画されます。

フルスクリーン表示は、コンテンツを表示するために利用できるスペースを最大にしたい場合に役立ちます。ステータスバーの背後にコンテンツを表示する場合は、ユーザがスクロールバーの下にあるコンテンツをスクロールして見ることができるよう、必ずそのコンテンツをスクロールビュー内に置いてください。ユーザはステータスバーやその他の半透明なビューの背後にあるコンテンツとやり取りをすることはできないため、コンテンツをスクロールできるようにすることは重要です。Navigation

Barは、Navigation Barの高さを考慮して、自動的にスクロールビュー（View Controllerのルートビューと仮定する）にスクロールコンテンツ挿入枠を追加します。それ以外の場合は、スクロールビューの`contentInset`プロパティを手動で変更しなければなりません。

コンテナは子のビューの枠を設定する

あるView ControllerがContainer View Controllerの子である場合、どの子が可視になるかを決めるのは親の役割です。親は表示しようとするビューを、自身のビュー階層にサブビューとして追加し、そのユーザインターフェイスに合わせて枠を設定します。以下に例を挙げます。

- Tab View Controllerは、ビューの下部に領域を確保してTab Barを描画します。現在可視になっている子のビューは、残りの領域を使うよう設定されます。
- Navigation View Controllerは、Navigation Bar用として上部に領域を確保します。現在可視になっている子がNavigation Barを表示しようとすれば、画面下部にもビューを配置します。ビューの残り部分は子が使えます。

子はその枠を親から受け取りますが、親もさらにその親から枠を受け取ります。このように祖先に向かって順次たどっていくとルートView Controllerに至りますが、これはウインドウから枠を受け取ります。

被表示View Controllerは、表示コンテキストを使う

View ControllerがほかのView Controllerに表示される場合、これが受け取る枠は、その表示に用いる表示コンテキストに基づいて決まります。“表示コンテキストは、表示される側のView Controllerが占める領域を表す” （102 ページ）を参照してください。

Popover Controllerは表示するView Controllerのサイズを設定する

Popover Controllerによって表示されるView Controllerは、そのビュー領域の大きさを、自身の`contentSizeForViewInPopover`プロパティ値を設定することにより決めることができます。ただし、Popover Controllerも自身の`popoverContentSize`プロパティ値を設定している場合は、こちらが優先します。ほかのView Controllerが使っているモデルと合わせたい場合は、Popover Controller側のプロパティを使って大きさや位置を制御してください。

View Controllerはビューのレイアウト処理に、どのように関与するか

View Controllerのビューの大きさが変わると、そのサブビューの位置も、空いている空間に合わせて変わります。Controllerのビュー階層にあるビューが、レイアウト制約や自動大きさ調整マスクを参照しながら、自分自身で大部分の処理を行います。しかしView Controllerも、処理の過程で随時呼び出されるので、大きさ調整処理に関与できるようになっています。実際に起こることを以下に示します。

1. View Controllerのビューが新しい大きさに変わります。
2. 自動レイアウト処理が無効であれば、自動大きさ調整マスクに従ってビューの大きさが変わります。
3. View Controllerの`viewWillLayoutSubviews`メソッドが呼び出されます。
4. ビューの`layoutSubviews`メソッドが呼び出されます。ビュー階層を構築するために自動レイアウト機能が使われていれば、次の手順でレイアウト制約を更新できます。
    a. View Controllerの`updateViewConstraints`メソッドが呼び出されます。
    b. `UIViewController`クラスの`updateViewConstraints`メソッドは、処理の過程でビューの`updateConstraints`メソッドを呼び出します。
    c. レイアウト制約の更新後、新しいレイアウトを計算し、ビューの位置を調整し直します。
5. View Controllerの`viewDidLayoutSubviews`メソッドが呼び出されます。

ビュー自身が位置調整に必要な処理をすべて実行し、View Controllerはまったく関与しなくてよいのが理想です。レイアウト全体をInterface Builderで設定できることも少なくありません。しかし、View Controllerが動的にビューを追加/削除する場合、Interface Builderによる静的なレイアウトでは対応できないでしょう。この場合、処理を制御する場所としてはView Controllerが適しています。シーン内にある他の各ビューの状況を把握しているのは、多くの場合、それぞれのビュー自身だけだからです。View Controllerにこの処理を実装する、最も適切な方針を以下に示します。

- レイアウト制約を使ってビューの位置を自動調整する（iOS 6以降）。`updateViewConstraints`をオーバーライドして、必要なレイアウト制約のうち、ビューがまだ設定していないものがあれば追加します。なお、このメソッドの実装コードでは、`[super updateViewConstraints]`を呼び出す必要があります。

  レイアウト制約について詳しくは、『*Cocoa Auto Layout Guide*』を参照してください。
- 自動大きさ調整マスクとコードを併用してビューの位置を調整する（iOS 5.x）。`layoutSubviews`をオーバーライドして、大きさ調整マスクだけでは自動設定できない位置を、適切に調整するコードを実装します。

ビューの自動サイズ変更プロパティと、それらがビューに与える影響の詳細については、『*View Programming Guide for iOS*』を参照してください。

レスポンダチェーンに組み込まれたView Controllerの使用

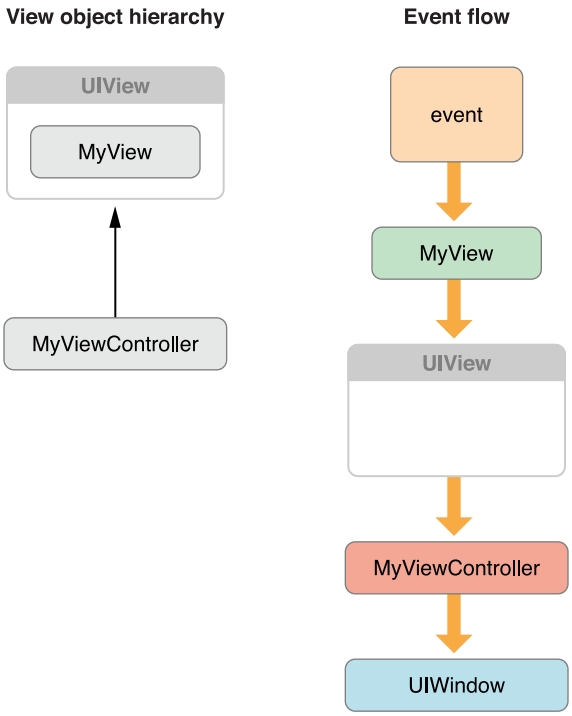
View Controllerは、`UIResponder`クラスの下位クラスです。したがって、すべての種類のイベントを処理することができます。ビューは通常、受け取ったイベントを自分自身で処理しない場合、スーパービューに渡して処理を委ねます。イベントはビュー階層をたどって順次伝搬し、最終的にルートビューに至ります。一方、チェーン内のビューがView Controllerの管理下にあれば、スーパービューに渡す前に、まずView Controllerオブジェクトに渡します。View Controllerはこのように、ビューで処理されないイベントに応答できます。View Controllerがイベントを処理しない場合は、通常どおり、そのイベントはそのビューのスーパービューに渡されます。

レスポンダチェーンは、イベントがどのようにアプリケーションに伝搬するかを定義する

図7-1に、ビュー階層内のイベントの流れを示します。この図では、画面と同じサイズの汎用のビューオブジェクトの中にカスタムビューが埋め込まれています。この汎用ビューはカスタムView Controllerで管理されています。このカスタムビューのフレームに届いたタッチイベントは、処理のためにそのビューに配信されます。そのビューがイベントを処理しない場合、そのイベントは親ビューに渡され

ます。この汎用ビューはイベントを処理しないため、これらのイベントは、まずView Controllerに渡されます。このView Controllerがイベントを処理しない場合は、そのイベントはさらに、汎用の`UIView`のスーパービュー（この場合は、ウインドウオブジェクト）に渡されます。

図 7-1 View Controllerのレスポンダチェーン

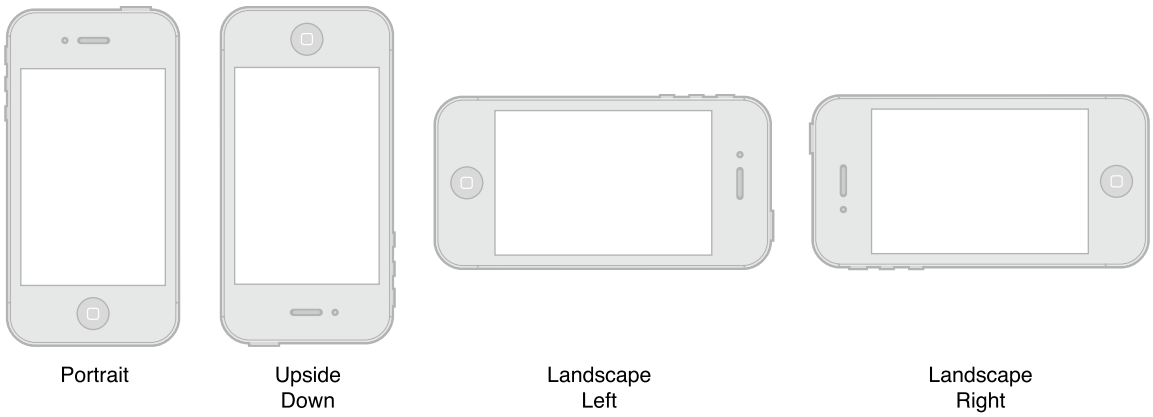


注意: View Controllerとそのビューの間のメッセージ送信関係は、View Controllerによってプライベートに管理されているため、アプリケーションのプログラムで変更することはできません。

カスタムView Controllerでわざわざタッチイベントを処理しようとは思わないかも知れませんが、それを使用して動きに基づくイベントを処理することもできます。また、カスタムView Controllerを使用して、ファーストレスポンダの設定や変更を行うこともできます。iOSアプリケーションでイベントを配信したり処理したりする方法の詳細については、『*Event Handling Guide for iOS*』を参照してください。

デバイスの向きに応じた調整

iOSベースのデバイスでは、加速度センサーを利用してデバイスの現在の向きを判別できます。アプリケーションは通常、縦向きと横向きのどちらでも動作します。iOSベースのデバイスの向きが変わると、システムは変化が生じたことを関連部分に知らせるために`UIDeviceOrientationDidChangeNotification`通知を送信します。デフォルトでは、UIKitフレームワークがこの通知を受け取り、インターフェイスの向きを自動的に更新するようになっています。つまり、わずかな例外を除けば、この通知を処理する必要はまったくありません。



画面を回転すれば、向きに応じてウインドウの大きさが変わります。ウインドウは大きさに合わせてルートView Controllerの枠を調整し、この大きさがビュー階層をたどって他のビューにも伝わります。したがって、View Controllerの向きの変化に対応する方法として最も簡単なのは、ルートビューの枠が変わればサブビューの位置も更新されるよう、ビュー階層を適切に構築することで多くの場合、ほかの条件によってView Controllerの可視領域が変わることもあるので、向きの変化を考えないとしてもこれは必要です。ビューのレイアウト設定について詳しくは、“View Controllerのビューの大きさ変更”（74 ページ）を参照してください。

デフォルトの動作が望ましくなければ、次の事項を制御できます。

- アプリケーションが対応するべき向き。
- 向きが変わる（回転する）際のアニメーション効果。

画面全体を占めないView Controllerは一般に、向きを考慮しなくても構いません。親View Controllerから描画領域を指定されるので、その中に描画すればよいのです。一方、ルートView Controller（あるいは画面全体を表すView Controller）は、デバイスの向きを意識することが多いでしょう。

アプリケーションが対応するべき向きの制御（iOS 6）

UIKitは、向きに関する通知を受け取ると、`UIApplication`オブジェクトおよびルートView Controllerに対し、新しい向きにしてよいかどうか問い合わせます。どちらも可と応答した場合に限り、実際に画面表示（ユーザインターフェイス）を回転します。それ以外の場合は、デバイスの向きを無視します。

ルートView Controller上にView Controllerを表示すると、システムの動作が2つ変わります。まず、その向きに対応しているかどうか判断する際、ルートView Controllerではなく、実際に表に出ているViewControllerに問い合わせるようになります。また、問い合わせを受けたView Controllerが、**望ましい向き**を指定できるようになります。View Controllerが全画面表示のとき、画面はこの望ましい向きになります。ユーザは、それがデバイスの向きとは違うことに気がつけば、デバイスの方を回転して向きを合わせるでしょう。望ましい向きを指定する状況としてよくあるのは、（アプリケーション側の都合で）ある決まった向きで表示したい、という場合です。

View Controllerが対応するインターフェイスの向きを宣言する

主ウインドウのルートView Controllerとして動作し、あるいは主ウインドウ上に全画面表示されるView Controllerは、どの向きに対応する（画面表示ができる）かを宣言できます。これは`supportedInterfaceOrientations`メソッドをオーバーライドして行います。デフォルトでは、iPadとしての特性を持つデバイス上のView Controllerは、4つの向きすべてに対応します。iPhoneとしての特性を持つデバイスの場合は、上下逆の縦向きを除く、3つの向きに対応します。

設計時には、ビューでサポートする向きを必ず選択して、それらの向きを考慮してコードを実装しなければなりません。実行時の情報に基づいて、サポートする向きを動的に選択する方法にはメリットはありません。たとえ、そのような選択をしても、可能性のあるすべての向きをサポートするために必要なコードを実装しなければなりません。そうであれば、それらの向きをサポートするかどうかを事前に選択しておいても同じです。

リスト8-1に、縦長と横長左向きをサポートするView Controllerの`supportedInterfaceOrientations`メソッドの典型的な実装を示します。このメソッドを独自に実装する場合も、これと同じくらい単純にするべきです。

リスト 8-1　`supportedInterfaceOrientations`メソッドの実装

```
- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait |
UIInterfaceOrientationMaskLandscapeLeft;
}
```

回転が起こるか否かを動的に制御する

自動回転の機能を動的に解除したい場合があります。たとえば、しばらくの間、まったく回転しないようにしたい、という状況です。ステータスバーの位置を手動で制御している（たとえば`setStatusBarOrientation:animated:`メソッドを実行している）間は、向きの変更を一時的に無効にしなければなりません。

対応する（画面表示ができる）向きを表すマスクを操作する、という方法で自動回転を一時的に無効にすることはできません。そうではなく、最上位View Controllerの`shouldAutorotate`メソッドをオーバーライドしてください。このメソッドは自動回転の実行前に呼び出されます。`NO`を返すようにすれば、回転は起こりません。

望ましい向きを宣言する

View Controllerを全画面表示する際、特定の向きであれば良好に表示される、ということがあります。常に同じ向きで表示されるようにしたい場合は、`supportedInterfaceOrientations`メソッドの戻り値として、その向きだけを返してください。向きを変えて表示しても構わないが、特に望ましい向きがあるという場合は、`preferredInterfaceOrientationForPresentation`メソッドをオーバーライドし、望ましい向きを返すようにします。リスト 8-2に、横向きが望ましいView Controllerの場合のコード例を示します。もちろん、この望ましい向きは、対応する（画面表示ができる）向きのうちのいずれかでなければなりません。

リスト 8-2　`preferredInterfaceOrientationForPresentation`メソッドの実装

```
- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation
{
    return UIInterfaceOrientationLandscapeLeft;
}
```

画面の表示について詳しくは、“View ControllerをほかのView Controllerから「表示」”（95 ページ）を参照してください。

アプリケーションが対応する画面の向きを宣言する

アプリケーションが対応する画面の向きを設定する最も簡単な方法は、プロジェクトの`Info.plist`ファイルを編集することです。View Controllerの場合と同様、4通りの中から表示できる向きを選んで定義します。詳細については、『*Information Property List Key Reference*』を参照してください。

表示可能な向きを制限する場合、アプリケーションのView Controllerすべてについて同じように制限しなければなりません。これはSystem View Controllerを使う場合も同様です。最上位View Controllerとアプリケーションのマスクの論理ANDを取って、その時点で表示可能な向きを判断します。この計算結果が0になってはなりません。0になった場合は`UIApplicationInvalidInterfaceOrientationException`例外が発生します。

マスクはアプリケーション全体に適用されるので、慎重に使ってください。

Important: アプリケーションとView Controllerの、どちらのマスクでも表示可能とされている向きが、少なくとも1つ必要です。ない場合は例外が発生します。

回転処理について（iOS 5以前）

iOS 5以前では、最上位の全画面View Controller以外も回転処理に関与することがありました。これは一般に、Container View Controllerが子に対して、対応する（画面表示できる）向きを問い合わせる場合に起こります。しかし実際上、子の側が親とは異なる答えを返すようオーバーライドしたとしても、役に立つことはほとんどありません。したがって、アプリケーションをiOS 5でも動作するようにしたい場合は、できるだけiOS 6の動作をエミュレートし、次のように実装してください。

- ルートView Controllerや全画面表示するView Controllerについては、ユーザインターフェイスの観点で意味のある向きの中から選択します。
- 子コントローラは、適応性の高いレイアウトを工夫して、デフォルトの解像度すべてに対応します。

サポートするインターフェイスの向きの宣言

サポートする（画面表示可能な）インターフェイスの向きを宣言するためには、`shouldAutorotateToInterfaceOrientation:`メソッドをオーバーライドし、その向きを返すようにします。設計時には、ビューでサポートする向きを必ず選択して、それらの向きを考慮してコードを実装しなければなりません。実行時の情報に基づいて、サポートする向きを動的に選択する方法にはメリットはありません。たとえ、そのような選択をしても、可能性のあるすべての向きをサポートするために必要なコードを実装しなければなりません。そうであれば、それらの向きをサポートするかどうかを事前に選択しておいても同じです。

リスト 8-3に、デフォルトの縦長と横長左向きをサポートするView Controllerの`shouldAutorotateToInterfaceOrientation:`メソッドの典型的な実装を示します。このメソッドを独自に実装する場合も、これと同じくらい単純にするべきです。

リスト 8-3 `shouldAutorotateToInterfaceOrientation:`メソッドの実装

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation
{
   if ((orientation == UIInterfaceOrientationPortrait) ||
       (orientation == UIInterfaceOrientationLandscapeLeft))
      return YES;

   return NO;
}
```

Important: 必ず少なくとも1つのインターフェイスの向きに対してYESを返す必要があります。

アプリケーションで両方向の横長をサポートする場合は、`orientation`パラメータを両方の横長定数と明示的に比較する代わりに、`UIInterfaceOrientationIsLandscape`マクロを近道として使用できます。同様に、UIKitフレームワークには、両方の縦長変数を識別する`UIInterfaceOrientationIsPortrait`マクロが定義されています。

可視のView Controllerの向き変更に応答する

回転が起こると、View Controllerは重要な処理を担います。可視のView Controllerは、回転処理のさまざまな段階で通知されるので、その機会に独自の処理を実行できるのです。これらのメソッドを使用して、ビューの表示や非表示、ビューの位置やサイズの変更、アプリケーションのほかの部分への向きの変化の通知が行えます。これらのカスタムメソッドは回転動作中に呼び出されるため、そこで時間のかかる操作を実行することは避けるべきです。また、ビュー階層全体を新しいビューセットに置き換える操作も避けるべきです。向きの違いに応じて特有のビューを提供する場合は、新規のView Controllerを表示するなどの、効率的な方法もあります（"代替の横長インターフェイスの作成"（87ページ）を参照）。

回転イベントがルートView Controllerに送られます。ルートView Controllerはこのイベントを、必要に応じて子に渡します。イベントは順次、View Controller階層に沿って伝わっていきます。回転の際に発生する一連のイベントを以下に示します。

1. ウインドウはルートView Controllerの`willRotateToInterfaceOrientation:duration:`メソッドを呼び出します。

コンテナView Controllerは、このメッセージを現在表示中のContent View Controllerに転送します。カスタムContent View Controllerでこのメソッドをオーバーライドして、インターフェイスが回転する前に、ビューを非表示にしたり、ビューのレイアウトにその他の変更を加えることもできます。

2. ウインドウは、View Controllerのビューの境界を調整します。その結果、ビューはView Controllerの`viewWillLayoutSubviews`メソッドを起動して、サブビューのレイアウトを調整します。その際、アプリケーションオブジェクトの`statusBarOrientation`プロパティを参照して、現在のユーザインターフェイスレイアウトを調べることができます。

   “View Controllerはビューのレイアウト処理に、どのように関与するか” （76 ページ）を参照してください。

3. View Controllerの`willAnimateRotationToInterfaceOrientation:duration:`メソッドが呼び出されます。このメソッドはアニメーションブロック内から呼び出されるので、プロパティ値を変更すれば、回転に伴う他の効果と併せて、アニメーションに反映されます。

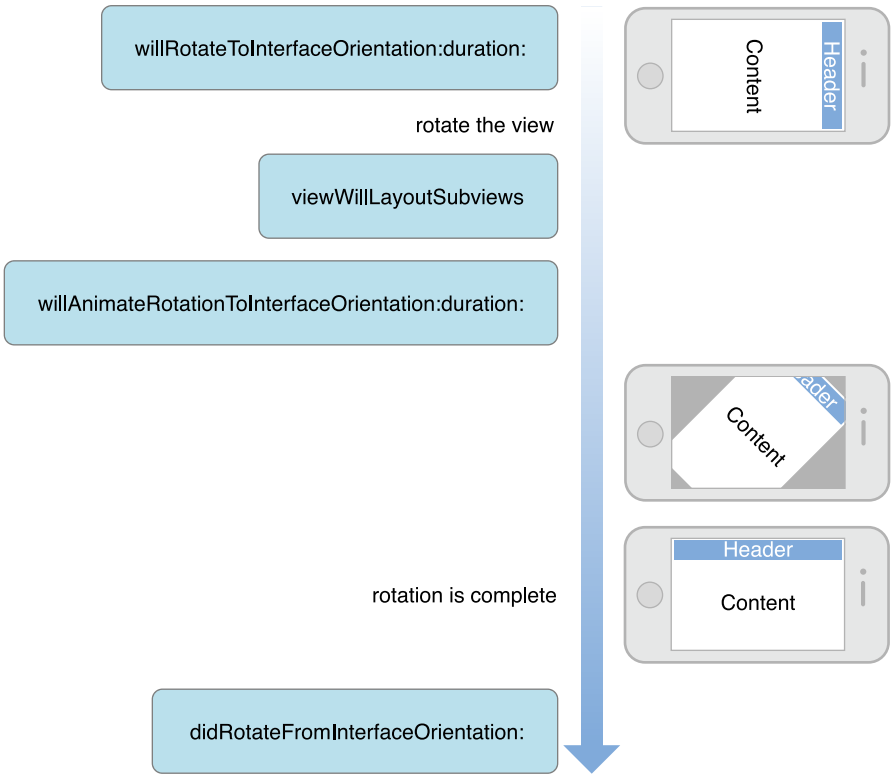
4. アニメーションが実行されます。

5. ウインドウはView Controllerの`didRotateFromInterfaceOrientation:`メソッドを呼び出します。

   コンテナView Controllerは、このメッセージを現在表示中のContent View Controllerに転送します。このアクションは、回転処理の終了を表します。このメソッドを使用して、ビューを表示したり、ビューのレイアウトを変更したり、アプリケーションにその他の変更を加えたりできます。

図 8-1は、上記のステップを図示したものです。また、この図には、回転処理のさまざまな段階でのインターフェイスの外観も示されています。

図 8-1 インターフェイス回転の処理



View Controllerが不可視であっても回転は起こりうる

回転が起こった時点で内容が画面上に現れていなかったView Controllerは、回転メッセージの送信先リストに入りません。たとえば次のような一連のイベントを考えてみましょう。

1. View Controller（A）が他のView Controller（B）の内容を、（自分自身を覆う形で）全画面に表示する。
2. ユーザがデバイスを回転し、その結果、画面（ユーザインターフェイス）の向きが変わる。
3. アプリケーションが、画面に現れているView Controller（B）を消去する。

この場合、回転中、View Controller（A）は不可視だったので、回転イベントを受け取っていません。次に画面に現れる時点で、通常のビューレイアウト処理の一環として、大きさや位置が変わるだけです。レイアウト処理のコードが、デバイスの現在の向きを知る必要がある場合は、アプリケーションオブジェクトの`statusBarOrientation`プロパティを参照してください。

代替の横長インターフェイスの作成

デバイスが縦長か横長かによって、同じデータを違う形式で表示したい場合、それを実現するには、2つの別々のView Controllerを使用します。一方のView Controllerは主たる向き（通常は縦長）でのデータの表示を管理し、もう一方のView Controllerは、別の向きでのデータの表示を管理します。2つのView Controllerを使用する方が、向きが変わるたびにビュー構成を大幅に変更するよりも簡単かつ効率的です。これによって、それぞれのView Controllerは、1つの向きでのデータの表示に集中できて、それに応じて要素を管理できます。また、現在の向きを確認するための条件判断によって、View Controllerのコードが煩雑になるのを避けることもできます。

代替として横長のインターフェイスが必要であれば、以下のことを行う必要があります。

- 次の2つのView Controllerオブジェクトを実装します。1つは縦長のみのインターフェイス、もう1つは横長のみのインターフェイスを表示します。
- `UIDeviceOrientationDidChangeNotification`通知の受け取りを登録します。このハンドラメソッド内で、デバイスの現在の向きに応じて、代替View Controllerを表示または非表示にします。

View Controllerは、通常は向きの変化を内部的に管理するため、1つの向きの場合にだけ表示されるように各View Controllerに指示しなければなりません。さらに、主たるView Controllerの実装では、デバイスの向きの変化を検出して、適切な向きに変化したときに代替View Controllerを表示する必要があります。主たる向きに戻ったときは、主たるView Controllerが代替View Controllerを消去します。

リスト8-4に、縦長をサポートする主たるView Controllerに実装しなければならない主要なメソッドを示します。主たるView Controllerは、ストーリーボードからロードされる際、向きの変化通知を、共用の`UIDevice`オブジェクトから受け取るよう登録します。このような通知を受信すると、`orientationChanged:`メソッドは、現在の向きに応じて横長用のView Controllerを表示または消去します。

リスト 8-4　横長用のView Controllerの表示

```
@implementation PortraitViewController
- (void)awakeFromNib
{
    isShowingLandscapeView = NO;
    [[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                 selector:@selector(orientationChanged:)
                                 name:UIDeviceOrientationDidChangeNotification
                                 object:nil];
}
```

```
- (void)orientationChanged:(NSNotification *)notification
{
    UIDeviceOrientation deviceOrientation = [UIDevice currentDevice].orientation;
    if (UIDeviceOrientationIsLandscape(deviceOrientation) &&
        !isShowingLandscapeView)
    {
        [self performSegueWithIdentifier:@"DisplayAlternateView" sender:self];
        isShowingLandscapeView = YES;
    }
    else if (UIDeviceOrientationIsPortrait(deviceOrientation) &&
             isShowingLandscapeView)
    {
        [self dismissViewControllerAnimated:YES completion:nil];
        isShowingLandscapeView = NO;
    }
}
```

回転コードの実装に関するヒント

ビューの複雑さの度合いによって、回転をサポートするコードを大量に書かなければならない場合もあれば、まったく書く必要のない場合もあります。必要な作業を見積もる際に、次のヒントをコード記述の目安にできます。

- **回転中はイベントの送付を一時的に無効にする。**ビューに対するイベントの送付を無効にすることで、向きの変更中に不要なコードが実行されるのを防ぎます。
- **表示されている地図の領域を保存する。**アプリケーションにMap Viewが含まれている場合は、回転を開始する前に、表示されている地図の領域の値を保存します。回転が完了したら、必要に応じて保存済みの値を使用して、表示される領域が回転前とほぼ同じになるようにします。
- **複雑なビュー階層の場合は、ビューをスナップショットイメージに置き換える。**大量のビューをアニメーション化することでパフォーマンスの問題が生じる場合は、これらのビューをビューのイメージが含まれているイメージビューに一時的に置き換えます。回転が完了したら、元のビューに入れ替えてイメージビューを削除します。
- **表示されているテーブルの中身を回転後に再度ロードする。**回転が終了したら、新しく表示されたテーブル行に適切にデータが埋められるように、強制的に再ロード操作を行います。

- **回転通知を使用して、アプリケーションの状態情報を更新する。**アプリケーションが現在の向きを使用してコンテンツの表示方法を決めている場合は、View Controllerの回転メソッド（またはデバイス向きの通知）を使用して向きの変更を知らせ、必要な調整を行います。

View Controllerの観点から見たアクセシビリティ

View Controllerは、ビューの動作を管理する以外にも、アプリケーションのアクセシビリティを高める支援ができます。アクセシビリティに配慮したアプリケーションとは、障碍者などであっても機能性や使い勝手を損なうことなく、有用なツールとして利用できるものを言います。

iOSアプリケーションは、アクセシビリティを高めるため、そのユーザインターフェイス要素に関する情報が、VoiceOverユーザにも伝わるようにしなければなりません。VoiceOverとは画面読み上げ技術のことで、画面に表示されているテキストや画像、コントロール部品などの情報を声に出して読み上げてくれるので、視覚障碍者でもアプリケーションを使うことができます。UIKitオブジェクトはアクセシビリティを備えていますが、View Controllerの側にも、アクセシビリティを向上するために行うべきことがあります。高いレベルでいうと、次の点を確実にする必要があるということです。

- ユーザがやり取りを行うすべてのユーザインターフェイス要素がアクセシブルである。これには、静的テキストなどの単に情報を提供する要素と同様に、アクションを実行するコントロールも含まれます。
- アクセシブルな要素がすべて、正確で有用な情報を提供している。

View Controllerは、以上の基本事項に加え、VoiceOverのカーソル位置をプログラムで設定する、VoiceOverのジェスチャに応答する、アクセシビリティに関する通知を監視する、などといった処理を組み込むことにより、アクセシビリティをさらに改善することができます。

VoiceOverのカーソルを所定の要素の位置に移動する

画面レイアウトが変わると、*VoiceOverのカーソル*位置は、画面上の要素を左から右、上から下の順に並べた、先頭の要素に戻ります。この「先頭」要素を変更して、ビューが画面に現れたとき、VoiceOverのカーソルがその位置に行くようにすることができます。

たとえば、Navigation ControllerがView Controllerをナビゲーションスタックにプッシュすると、VoiceOverのカーソルは、ナビゲーションバーの「Back」ボタンに置かれます。アプリケーションの内容にもよりますが、ナビゲーションバーの見出しなど、他の要素に置く方が便利かも知れません。

これを実現するには`UIAccessibilityPostNotification`メソッドを使います。引数として、通知の種類を表す定数`UIAccessibilityScreenChangedNotification`（画面の内容が変化した旨をVoiceOverに通知）と、最初にフォーカスを与える要素を指定します（リスト 9-1を参照）。

リスト 9-1　アクセシビリティ通知を送って最初に読み上げる要素を変更するコード例

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

    UIAccessibilityPostNotification(UIAccessibilityScreenChangedNotification,
                                    self.myFirstElement);
}
@end
```

縦長から横長への切り替えなど、画面の内容は同じでレイアウトだけが変わった場合は、通知の種類として`UIAccessibilityScreenChangedNotification`の代わりに`UIAccessibilityLayoutChangedNotification`指定します。

> **注意:** デバイスを回転するとレイアウトが変化し、VoiceOverのカーソル位置がリセットされます。

VoiceOverの特別なジェスチャに応答する

VoiceOverのユーザが実行して独自のアクションを起動できる、特別なジェスチャがあります。VoiceOverの標準的なジェスチャと違い、動作を独自に定義できる点が特別です。このジェスチャが行われた旨を検出するためには、ビューまたはView Controllerの、所定のメソッドをオーバーライドする必要があります。

ジェスチャが為されると、VoiceOverのフォーカスがあるビューが、これに応答できるかどうかをまず調べます。次いで、ここからレスポンダチェーンを順次たどって、該当するVoiceOverジェスチャメソッドが実装されていないか確認します。実装が見つからなければ、最終的にシステムのデフォルトアクションが実行されます。たとえば「マジックタップ」ジェスチャの場合、現在のビューからアプリケーションデリゲートまで順にたどっても実装が見つからなければ、Musicアプリケーションによる音楽再生を始める/一時停止する、という動作になります。

アクションとしてどのような処理でも実装できますが、VoiceOverのユーザは通常、一定のガイドラインに基づいて実装されているものと期待します。他のジェスチャと同様、以下に示すパターンに従い、直感的に予測できるようなアクションを実装してください。

VoiceOverの特別なジェスチャは5種類あります。

- **エスケープ。**2本の指をZ字型に動かすジェスチャで、モーダルダイアログを閉じる、あるいはナビゲーション階層を1段階戻る、というアクションが対応します。
- **マジックタップ。**2本の指によるダブルタップで、その状況で最も実行したいであろうアクションが対応します。
- **3本指スクロール。**3本の指によるスワイプ操作で、上下方向または左右方向にスクロールする、というアクションが対応します。
- **増加と減少。**1本の指による上下のスワイプ操作で、ある要素の値を、所定の大きさだけ増やす/減らす、というアクションが対応します。Adjustableアクセシビリティ特性を備えた要素には、このメソッドを実装しなければなりません。

> **注意:** VoiceOverの特別なジェスチャに対応するメソッドは、レスポンダチェーンをたどって次に進むかどうか、を表すブール値を返します。ここでやめる場合は`YES`、そうでなければ`NO`を返してください。

エスケープ

コンテンツを覆うビュー（モーダルダイアログ、警告など）が現れている場合に、これを消去する処理を、`accessibilityPerformEscape`メソッドをオーバーライドして実装します。「エスケープ」ジェスチャに対応する機能は、キーボードの`Esc`キーを押したときと同じにしてください。一時ダイアローグやシートを消してメインのコンテンツを表に出す、という機能です。

もうひとつの使い方として、ナビゲーション階層に沿って1レベル戻る、という機能も考えられます。`UINavigationController`にはデフォルトでこの機能が実装されています。Navigation Controllerに類したものを設計するのであれば、「エスケープ」ジェスチャに対応して、ナビゲーションスタックを1レベル戻る、というアクションを実装してください。これが、VoiceOverユーザが想定するであろう機能です。

マジックタップ

「マジックタップ」ジェスチャの目的は、現在の状況に最もふさわしい、あるいは最もよく使われるであろうアクションを、迅速に実行することです。たとえばPhoneアプリケーションの場合、電話をかける、あるいはかかってきた電話に出る、というアクションです。Clockアプリケーションであれば、ストップウォッチを開始/停止します。VoiceOverのカーソルが置かれているビューに関係なく、同じアクションを実行するようにしたい場合は、View Controllerの`accessibilityPerformMagicTap`メソッドを実装してください。

> **注意:** 「マジックタップ」ジェスチャに対して、アプリケーションのどこからでも同じアクションを実行したい場合は、アプリケーションデリゲートの`accessibilityPerformMagicTap`メソッドを実装する方が適しています。

3本指スクロール

VoiceOverのユーザが3本指スクロールをすると、`accessibilityScroll:`メソッドが起動されます。スクロール方向を表す値が、引数`UIAccessibilityScrollDirection`として渡されます。独自のスクロールビューがある場合は、そのビュー自身に実装する方が適切かも知れません。

増加と減少

Adjustable特性を備えた要素には、`accessibilityIncrement`メソッド、`accessibilityDecrement`メソッドが必要です。該当するビュー自身に実装してください。

アクセシビリティ通知の監視

アクセシビリティ通知を監視し、コールバックメソッドを起動することができます。状況によっては、UIKitがアクセシビリティ通知を発し、アプリケーションがこれを監視して機能を拡張する、ということができます。

たとえば通知`UIAccessibilityAnnouncementDidFinishNotification`を監視していれば、VoiceOverの読み上げが終了したときに、所定のメソッドを起動できます。Appleが提供するiBooksアプリケーションにもこの仕組みが使われています。VoiceOverが1行分を読み上げ終わると通知が送出され、これを受けて、次の行を読み上げる処理を起動するようになっています。あるページの最終行であった場合は、コールバック内で必要な処理を行い、iBooksに対し、ページをめくって読み上げを続行するよう指示します。この仕組みにより、行を単位としてテキストをナビゲートし、中断することなく読み上げが続く、という動きを実現しているのです。

アクセシビリティ通知のオブザーバとして登録するには、デフォルトの通知センターを使います。次に、`selector`引数に指定したのと同じ名前のメソッドを実装します（リスト 9-2を参照）。

リスト 9-2　アクセシビリティ通知のオブザーバとして登録するコード例

```
@implementation MyViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
    [[NSNotificationCenter defaultCenter]
        addObserver:self
           selector:@selector(didFinishAnnouncement:)
               name:UIAccessibilityAnnouncementDidFinishNotification
             object:nil];
}

- (void)didFinishAnnouncement:(NSNotification *)dict
{
    NSString *valueSpoken = [[dict userInfo]
objectForKey:UIAccessibilityAnnouncementKeyStringValue];
    NSString *wasSuccessful = [[dict userInfo]
objectForKey:UIAccessibilityAnnouncementKeyWasSuccessful];
    // ...
}
@end
```

UIAccessibilityAnnouncementDidFinishNotificationには引数としてNSNotification辞書が渡されます。これを参照して、実際に読み上げられた値（テキスト）を調べ、中断することなく読み上げが終了したかどうかを判断します。ユーザが「中止」のジェスチャをしたり、他の要素にスワイプしたりすると、読み上げは中断します。

ほかにも有用な通知としてUIAccessibilityVoiceOverStatusChangedがあります。これはVoiceOverのオン/オフ切り替えを検出します。アプリケーション外で切り替えられた場合、アプリケーションがフォアグラウンドに戻ってきた時点で通知を受け取ることになります。
UIAccessibilityVoiceOverStatusChangedは引数を取らないので、セレクタに指定するメソッド名の末尾にコロン（:）をつける必要はありません。

監視可能な通知の一覧が、『*UIAccessibility Protocol Reference*』の「Notifications」に載っています。なお、監視できるのはUIKitが送出する通知（その実体はNSStringオブジェクト）だけです。アプリケーションが送出するint型の通知は監視できません。

View ControllerをほかのView Controllerから「表示」

ほかのView Controllerの表示を働きかける機能は、現在のワークフローに割り込んで一連の新しいビューを表示するときに自由に使えるツールです。もっとも一般的なのは、ユーザから重要な情報を得るために、View Controllerを一時的な割り込みとして表示するケースです。一方、特定のタイミングでアプリケーションに代替のインターフェイスを実装するために、View Controllerを表示して使用することもできます。

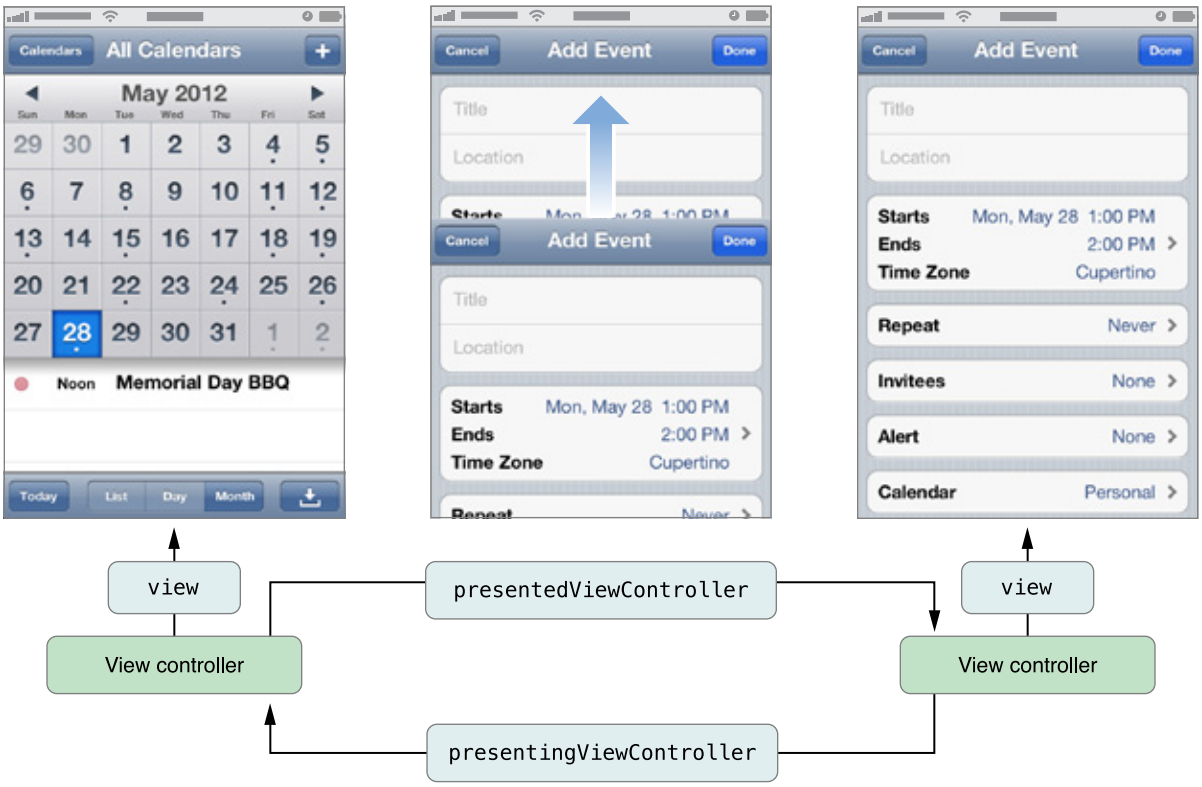
View ControllerがほかのView Controllerを表示する方法

被表示（表示される側）View Controllerは、（`UITabBarController`や`UINavigationController`のように、）`UIViewController`の特定のサブクラスではありません。むしろ、任意のView Controllerをアプリケーションから表示できます。しかし、Tab Bar ControllerやNavigation Controllerのように、前のビュー階層と新たに表示するビュー階層の関係に特定の意味を持たせる場合に、View Controllerを表示します。

モーダルView Controllerを表示すると、システムによって表示元のView Controllerと表示されたView Controllerの間に関係が作成されます。具体的に言うと、表示元のView Controllerの`presentedViewController`プロパティが、表示したView Controllerを指すように更新されます。同様に、表示されたView Controllerの`presentingViewController`プロパティが、表示元のView Controller

を指すように更新されます。図 10-1に、「カレンダー(Calendar)」アプリケーションのメイン画面を管理するView Controllerと、新規イベントを作成するために使われる、被表示View Controllerとの関係を示します。

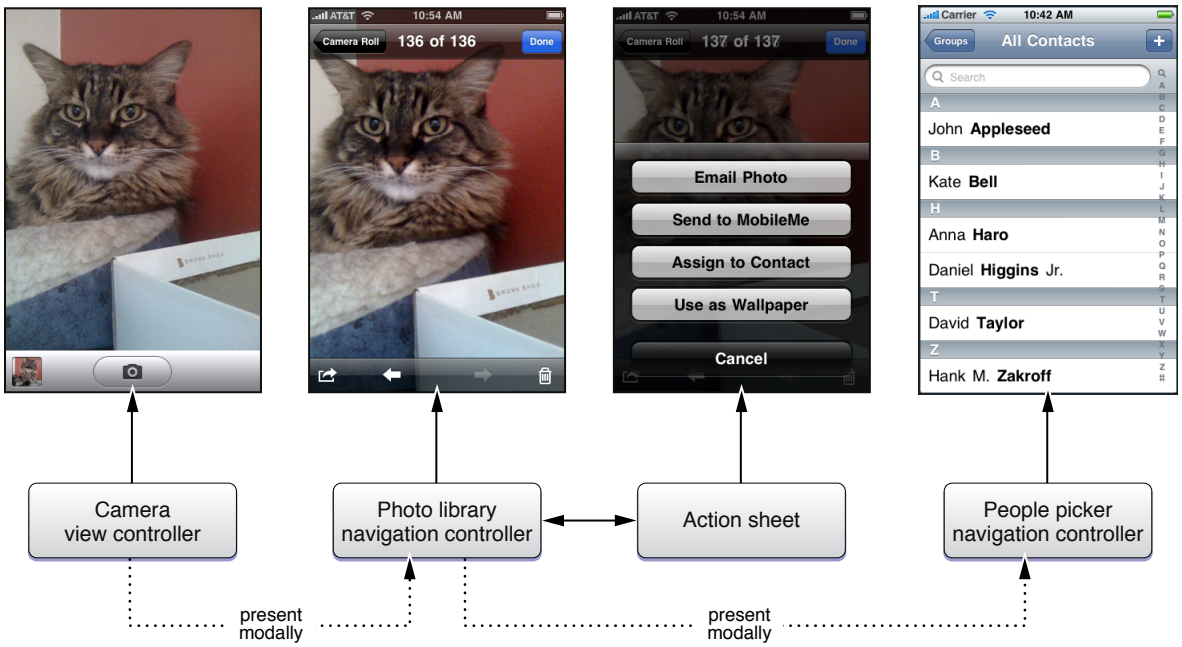
図 10-1　「カレンダー(Calendar)」アプリケーションの被表示ビュー



View Controllerオブジェクトは、同時に1つだけView Controllerを表示できます。これは、ほかのView Controllerによって表示されたView Controller自身にも当てはまります。つまり、被表示View Controllerの手前に新規のView Controllerを表示して、View Controllerを連鎖させることができます。図 10-2に、この連鎖のプロセスとそれを開始するアクションを図示します。この例では、ユーザがカメラビューのアイコンをタップすると、アプリケーションは、ユーザの写真を含むView Controllerを表示します。フォトライブラリのツールバーのアクションボタンをタップすると、適切なアクションを選択する画面がユーザに表示されます。次に、ユーザが選択したアクションに対応する別のView Controller（People

ピッカー）が表示されます。連絡先を選択するか、またはPeopleピッカーをキャンセルすると、そのインターフェイスが閉じてフォトライブラリに戻ります。「完了(Done)」ボタンをタップすると、フォトライブラリが閉じてカメラインターフェイスに戻ります。

図 10-2　モーダルView Controllerチェーンの作成



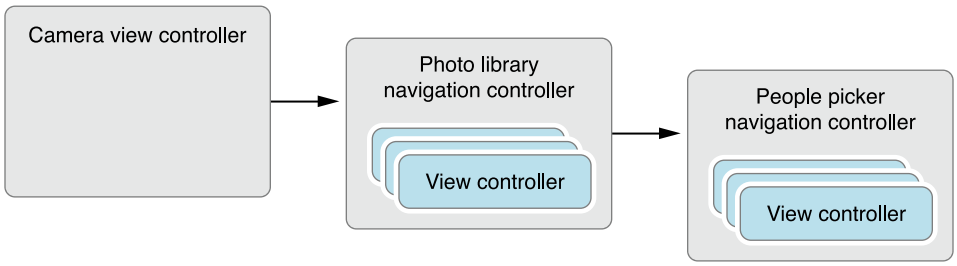
表示されたView Controllerのチェーン内の各View Controllerは、同じチェーン内の周囲のほかのオブジェクトへのポインタを持っています。つまり、被表示View Controllerが立場を変えてほかのView Controllerを表示する場合、`presentingViewController`プロパティと`presentedViewController`プロパティの両方に有効なオブジェクトを持っています。これらの関係を使用して、必要に応じてView Controllerのチェーンをたどることができます。たとえば、ユーザが現在の操作をキャンセルした場合は、最初に表示されたView Controllerを閉じることによって、チェーン内のすべてのオブジェクトを閉じることができます。1つのView Controllerを閉じると、そのView Controllerだけでなく、そのView Controllerが表示したすべてのView Controllerを閉じることができます。

図 10-2 （97 ページ）では、表示したView Controllerが両方ともNavigation Controllerであることに注目してください。`UINavigationController`オブジェクトも、Content View Controllerと同じ手順で表示できます。

Navigation Controllerを表示するときは、ナビゲーションスタック上の任意のView Controllerを表示するのではなく、必ず`UINavigationController`オブジェクトそのものを表示します。ただし、ナビゲーションスタック上の個々のView Controllerが、ほかのView Controller（ほかのNavigation Controllerも含

む）を表示することはできます。図10-3に、上記の例に関連するオブジェクトの詳細を示します。この図から分かる通り、Peopleピッカーは「Photo Library」Navigation Controllerによって表示されるのではなく、そのナビゲーションスタック上のContent View Controllerの1つによって表示されます。

図 10-3 Navigation Controllerのモーダルモードでの表示



モーダルビューの表示スタイル

iPadアプリケーションの場合、コンテンツをいくつかの異なるスタイルを使用して表示できます。iPhoneアプリケーションでは、表示されるビューは常にウインドウの可視部分を覆いますが、iPad上で実行しているときは、View Controllerはビューの`modalPresentationStyle`プロパティの値を使用して、表示されているときのビューの外観を決定します。このプロパティに異なるオプションを指定して、View Controllerの表示が画面全体を占有するのか、あるいは画面の一部だけを占有するのかを調整できます。

図 10-4に、利用可能な主要な表示スタイルを示します（`UIModalPresentationCurrentContext`スタイルを利用すると、View Controllerは親の表示スタイルを採用します）。各表示スタイルでは、薄暗くなっている領域に背後のコンテンツが表示されますが、そのコンテンツ内をタップすることはできません。したがって、Popoverとは違って、表示されるビューには、従来通り、ユーザがビューを消去するためのコントロールが含まれていなければなりません。

図 10-4　iPadの表示スタイル

各種の表示スタイルをどの局面で使用するかのガイダンスについては、「Modal View」 in *iOS Human Interface Guidelines* を参照してください。

View Controllerの表示とトランジションスタイルの選択

ストーリーボードのセグエを使ってView Controllerの表示を働きかけると、自動的に該当するインスタンスが生成、表示されます。表示する側のView Controllerは、実際に画面に現れる前に、デスティネーションView Controllerに必要な設定を施すことができます。詳しくは“セグエにトリガがかかったときの、デスティネーションControllerの設定”（109 ページ）を参照してください。

View Controllerをプログラムで表示する必要がある場合は、次の手順を実行しなければなりません。

1. 表示するView Controllerを作成します。
2. そのView Controllerの`modalTransitionStyle`プロパティを希望の値に設定します。
3. View Controllerにデリゲートオブジェクトを割り当てます。通常、デリゲートは表示する側のView Controllerです。デリゲートは、表示された側のView Controllerが、自分自身が破棄されてもよい状態になったとき、表示した側のView Controllerにその旨を通知するために使います。これが典型的な使い方ですが、ほかにも情報をデリゲートに返すことがあります。
4. 現在のView Controllerの`presentViewController:animated:completion:`メソッドを、表示するView Controllerを渡して呼び出します。

`presentViewController:animated:completion:`メソッドは、指定されたView Controllerオブジェクトに対応するビューを表示して、新しいView Controllerと現在のView Controllerとの間に「表示する/表示される」の関係を作成します。アプリケーションを以前の状態に復元する場合を除いて、通常は、新しいView Controllerの表示をアニメーション化します。使用するトランジションスタイルは、表示するのView Controllerの使い方によって異なります。表 10-1に、表示するView Controllerの`modalTransitionStyle`プロパティに割り当てることができるトランジションスタイルとその使用法を示します。

表 10-1　モーダルView Controllerのトランジションスタイル

トランジションスタイル	使用方法
`UIModalTransition-StyleCoverVertical`	このスタイルは、ユーザから情報を収集するために現在のワークフローに割り込みを行う場合に使用します。ユーザが変更を加える可能性のあるコンテンツの表示にも使用できます。 このトランジションスタイルでは、Content View Controllerが、そのView Controllerを明示的に閉じるためのボタンを提供しなければなりません。通常、これは「完了(Done)」ボタンとオプションの「キャンセル(Cancel)」ボタンになります。 トランジションスタイルを明示的に設定しない場合は、デフォルトでこのスタイルが使われます。

トランジションスタイル	使用方法
`UIModalTransitionStyleFlipHorizontal`	このスタイルは、アプリケーションの作業モードを一時的に変更するために使用します。このスタイルがもっともよく使われるのは、「株価(Stocks)」アプリケーションや「天気(Weather)」アプリケーションなどで、頻繁に変更される可能性がある設定を表示する場合です。これらの設定はアプリケーション全体に影響を与えるものもあれば、現在の画面に特有のものもあります。 このトランジションスタイルでは、一般に、アプリケーションの通常の動作モードに戻るための何らかのボタンを提供します。
`UIModalTransitionStyleCrossDissolve`	このスタイルは、デバイスの向きが変化したときに代替のインターフェイスを表示するために使用します。このような場合に、向きの変化の通知に応答して代替のインターフェイスを表示したり閉じたりするのはアプリケーションの仕事です。 メディアベースのアプリケーションでは、このスタイルを使用してメディアコンテンツを表示する画面をフェードインすることもできます。 デバイスの向きの変化に応答して代替のインターフェイスを実装する方法の例は、“代替の横長インターフェイスの作成”（87 ページ）を参照してください。

リスト 10-1に、View Controllerをプログラムで表示する方法を示します。ユーザが新しいレシピを追加すると、アプリケーションは、Navigation Controllerを表示して、そのレシピに関する基本情報の入力をユーザに要求します。「キャンセル(Cancel)」ボタンと「完了(Done)」ボタンを標準で置く場所を確保するために、Navigation Controllerを選択しました。また、Navigation Controllerを使用すると、新規レシピのインターフェイスを拡張することも簡単になります。デベロッパがしなければならないのは、新しいView Controllerをナビゲーションスタックにプッシュすることだけです。

リスト 10-1　View Controllerをプログラムで表示する

```
- (void)add:(id)sender {
    // Navigation Controller用のルートView Controllerを作成する。
    // この新しいView Controllerに、ナビゲーションバー用の
    // 「キャンセル(Cancel)」ボタンと「完了(Done)」ボタンを設定する。
    RecipeAddViewController *addController = [[RecipeAddViewController alloc]
                        init];

    // RecipeAddViewControllerを設定する。この場合コントローラは、
    // 変更点をカスタムデリゲートオブジェクトに報告する。
```

```
    addController.delegate = self;

    // Navigation Controllerを作成し表示する。
    UINavigationController *navigationController = [[UINavigationController alloc]
                                initWithRootViewController:addController];
    [self presentViewController:navigationController animated:YES completion: nil];
}
```

新規レシピ入力用のインターフェイスでユーザが「完了(Done)」ボタンまたは「キャンセル(Cancel)」のいずれかをタップすると、アプリケーションはこのView Controllerを閉じてメインビューに戻ります。詳しくは"表示されたView Controllerを閉じる"（103 ページ）を参照してください。

表示コンテキストは、表示される側のView Controllerが占める領域を表す

プレゼンテーション領域を定義するために使われる画面上の領域は、表示コンテキストで決まります。デフォルトでは、表示コンテキストを提供するのはルートView Controllerであり、したがってそのビューの枠に基づいて、表示コンテキストの枠を定義することになります。しかし、表示する側のView Controller、あるいはView Controller階層上のほかの祖先が、代わりに表示コンテキストを提供することもあります。この場合、他のView Controllerが表示コンテキストを提供するならば、代わりにその枠を使って、表示される側のビューの枠を決めます。このように適応性の高い機構を利用して、画面の一部を区切ってモーダルプレゼンテーションを行い、それ以外の領域はほかのコンテンツが見える状態のままにすることも可能です。

View Controllerが表示される際、iOSは表示コンテキストを検索します。検索の最初の対象は、表示する側のView Controllerで、その`definesPresentationContext`プロパティを参照します。このプロパティ値が`YES`であれば、表示する側のView Controllerに表示コンテキストが定義されています。そうでなければ、View Controller階層を上に向かって順次たどり、`YES`を返すView Controllerが見つかるか、ウインドウのルートView Controllerに到達するまで続けます。

あるView Controllerが表示コンテキストを定義する場合、これは表示スタイルも定義します。通常、表示される側のView Controllerは、`modalTransitionStyle`プロパティで表示形態を判断します。`definesPresentationContext`が`YES`であるView Controllerは、`providesPresentationContextTransitionStyle`も`YES`とすることができます。`providesPresentationContextTransitionStyle`が`YES`であれば、iOSは表示コンテキストの`modalPresentationStyle`を調べて、新しいView Controllerをどのように表示するか判断します。

表示されたView Controllerを閉じる

表示されたView Controllerを閉じるときによく使われるアプローチは、表示した側のView Controllerに閉じさせる方法です。つまり、できる限り、そのView Controllerを表示したのと同じView Controllerが閉じるべきなのです。表示された側のView Controllerが、表示した側に対して、自分自身を閉じるように通知する方法はいくつかありますが、よく使われるのはデリゲーションです。詳細については、“デリゲーションを使ってほかのコントローラと通信する” （111 ページ）を参照してください。

標準のシステムView Controllerの表示

アプリケーションから表示できる、標準のシステムView Controllerがいくつも設計されており、これらのView Controllerを表示する際の基本規則は、カスタムContent View Controllerの場合と同じです。ただし、アプリケーションは、システムView Controllerが管理するビュー階層にはアクセスできないため、そのビュー内のコントロールに対応するアクションを単純に実装することはできません。システムView Controllerとのやり取りは通常、デリゲートオブジェクトを介して行われます。

各システムView Controllerには、それに対応するプロトコルが定義されています。それらのメソッドをデリゲートオブジェクトに実装します。通常、各デリゲートには、選択された項目を受け付けたり、操作をキャンセルしたりするためのメソッドを実装します。デリゲートオブジェクトでは、必ず両方のケースを処理できるようにしておくべきです。デリゲートオブジェクトで実行しなければならないもっとも重要な処理の1つは、表示元のView Controller（表示された側のView Controllerにとっては親）の`dismissModalViewControllerAnimated:`メソッドを呼び出して、表示したView Controllerを閉じることです。

表 10-2に、iOSで利用できる標準のシステムView Controllerをいくつか示します。これらのクラスの詳細（その機能も含む）については、それぞれのクラスのリファレンスドキュメントを参照してください。

表 10-2　標準のシステムView Controller

フレームワーク	View Controller

Address Book UI	`ABNewPersonViewController`
	`ABPeoplePickerNavigationController`
	`ABPersonViewController`
	`ABUnknownPersonViewController`

Event Kit UI	`EKEventEditViewController` `EKEventViewController`

フレームワーク	View Controller

Game Kit	`GKAchievementViewController`
	`GKLeaderboardViewController`
	`GKMatchmakerViewController`
	`GKPeerPickerController`
	`GKTurnBasedMatchmakerViewController`

Message UI	`MFMailComposeViewController` `MFMessageComposeViewController`
Media Player	`MPMediaPickerController` `MPMoviePlayerViewController`
UIKit	`UIImagePickerController` `UIVideoEditorController`

注意: Media Playerフレームワークの`MPMoviePlayerController`クラスは、技術的にはモーダルView Controllerと見なされますが、それを使用する際のセマンティクスは少し異なります。このView Controllerを表示するのではなく、View Controllerを初期化したら、そのメディアファイルを再生するように指示します。その後は、このView Controllerが、ビューの表示と消去のすべての側面を処理します（ムービーの再生用には`MPMoviePlayerController`の代わりに`MPMoviePlayerViewController`クラスを標準View Controllerとして使用できます）。

View Controller間の連携

画面に表示するコンテンツが1つだけのiOSアプリケーションはほとんどありません。最初に起動された時点でいくつかのコンテンツを表示した後、ユーザのアクションに応じて、ほかのコンテンツを表示したり消去したりします。この「画面遷移」の仕組みにより、同時ではないけれども多数のコンテンツを表示する、統一されたユーザインターフェイスを実現できます。

一般に、コンテンツは細かく分割し、さまざまなView Controllerクラスで管理します。このコーディング方針によると、実装しやすい、小さくて単純なコントローラクラスを作成すればよいことになります。しかし複数のクラスで処理を分担すれば、クラスの設計に新たな要件が生じます。すなわち、全体が単一のインターフェイスに見えるよう、View Controller同士が相互にメッセージやデータを交換し、あるコントローラから別のコントローラにうまく遷移できるようにする必要があるのです。そのため、View Controllerクラスの内部でビューを制御し、割り当てられたタスクを実行するだけでなく、その外部にあるView Controllerと相互に通信しています。

View Controller間の連携動作が発生するとき

View Controller間の通信は、アプリケーションにおいて当該View Controllerが果たす役割と不可分に結びついています。View Controller間に起こりうるやり取りをすべて記述することはできません。相互の関係の数や性質は、アプリケーションの設計によってそれぞれ異なるからです。しかし、相互のやり取りがどの時点で起こるか、を列挙し、実際のアプリケーションでどのような調整が必要になるか、若干の例を挙げることは可能です。

View Controllerの生存期間は3つに分かれ、それぞれの段階でほかのオブジェクトと連携します。

View Controllerのインストール時。View Controllerが生成される際、その生成処理を起こすのは、既存のView Controllerその他のオブジェクトです。通常、View Controllerを生成した理由や、実行するべきタスクは、この生成元オブジェクトが知っています。したがって、View Controllerのインスタンスを生成した後、これを伝えなければなりません。

この初期設定の具体的な内容は、場合によってさまざまです。ある場合には、データオブジェクトを新しいView Controllerに渡します。あるいは、表示スタイルを設定したり、2つのView Controller間を接続するリンクを設けたりする場合もあります。このリンクは、View Controllerの生存期間を通じて、さまざまな通信に使えます。

View Controllerの生存期間を通じて。View Controllerの中には、生存期間を通じて、ほかのView Controllerと通信するものがあります。通信相手は、自分自身を生成した元のView Controllerであったり、生存期間が重なるピアであったり、あるいは自分自身が生成した新しいView Controllerであったりします。よく見られる設計例を示しましょう。

- あるView Controllerは、ユーザがあるアクションをした旨の通知を送ります。これは一方的な通知なので、受け取り手のオブジェクトは、単に何かが起こった旨を知らされるだけです。
- あるView Controllerは、別のView Controllerにデータを送ります。たとえば、Tab Bar Controllerは普通、その子同士の間に何の関係も結ばないのが普通ですが、アプリケーションによっては、各タブが同じデータオブジェクトを異なる方法で表示していることがあります。ユーザがタブを切り替えると、切り替え前のタブに対応するView Controllerは、選択情報を、切り替え後の次に表示されようとしているView Controllerに送ります。一方、新しいView Controllerはこのデータを使ってビューを設定し、遷移がシームレスに起こったように見せます。この場合、アクションの結果、View Controllerのインスタンスが新たに生成されることはありません。2つのView Controllerは対等の関係にあり、生存期間も同じであって、ユーザがタブを切り替える都度、互いに調整し合いながら動作します。
- あるView Controllerは、別のView Controllerに、あるアクションの実行権限を与えるメッセージを送ります。たとえば、当該View Controllerはユーザのデータ入力を受け付け、別のコントローラにそのデータを送って妥当性の判断を委ねる、という形の連携です。妥当でなければ、当該データを受理しない、あるいはエラー表示用のインターフェイスに変更する、などの処理をすることになります。

View Controllerの破棄。View Controllerは多くの場合、タスク終了時にメッセージを送ります。自分自身を生成したコントローラが解放（破棄）も行う、という規約が貫かれているので、このメッセージは頻繁に現れます。単にユーザがタスクを終了した旨を伝えるだけのメッセージかも知れません。しかし場合によっては、当該タスクにより生成された新しいデータオブジェクトを、自分自身を生成したコントローラに返すこともあります。

View Controllerの生存期間を通じ、ほかのView Controllerと情報を交換するのはありふれたことです。何かが起こった旨を伝える、データを送る、といった形の情報交換ばかりでなく、相手のアクティビティに何らかの制御を及ぼすことすらあります。

View Controllerのインスタンス生成時には、ストーリーボードを使って必要な設定を施す

ストーリーボードには、新たに生成したコントローラに対して、表示に先立ち、必要な設定を行う機能があります。ストーリーボードは、必要に応じて自動的にView Controllerのインスタンスを生成しますが、その際、アプリケーション上のあるオブジェクトを呼び出して、当該View Controllerに設定

を施したり、これとの間にリンクを生成したりすることができるようにします。アプリケーションの最初の起動時に、アプリケーションデリゲートは初期View Controllerを設定します。セグエにトリガがかかった場合、ソースView Controllerはデスティネーション View Controllerの設定を行います。

デスティネーションView Controllerの実装に関していくつか規約があります。

- デスティネーションView Controllerは、自分自身に設定を施すためのプロパティやメソッドを公開します。
- デスティネーションView Controllerは、自分自身の生成元であるView Controller以外とは、できるだけ通信をしないようにします。どうしても必要な場合、その通信経路にはデリゲーションを使わなければなりません。デスティネーションView Controllerのデリゲートは、プロパティのひとつとして設定します。

以上の規約に従えば、アプリケーションを構成するView Controllerクラス間に、過度の依存関係が生じません。依存関係をできるだけ少なくすれば、コードの再利用性が高まります。また、アプリケーションのほかの部分と切り離して、独立にテストすることも容易になります。

起動時における初期View Controllerの設定

プロジェクトでメインストーリーボードを定義していれば、iOSは自動的に、アプリケーションのセットアップ処理の大部分を実行します。アプリケーションが`UIApplicationMain`関数を呼び出すと、iOSは次のような処理を行います。

1. アプリケーションデリゲートのインスタンスを、`UIApplicationMain`関数に渡されたクラス名に基づいて生成します。
2. ウインドウを生成し、メイン画面にアタッチします。
3. アプリケーションデリゲートに`window`プロパティが実装されていれば、このプロパティに新しいウインドウを設定します。
4. アプリケーションの情報プロパティリストファイルで参照されている、メインストーリーボードをロードします。
5. メインストーリーボードの初期View Controllerのインスタンスを生成します。
6. ウインドウの`rootViewController`プロパティに、新しいView Controllerを設定します。
7. アプリケーションデリゲートの`application:didFinishLaunchingWithOptions:`メソッドを呼び出します。アプリケーションデリゲートは初期View Controller（Container View Controllerであればその子を含む）の設定をする、という想定です。
8. ウインドウの`makeKeyAndVisible`メソッドを呼び出して、ウインドウを表示します。

リスト 11-1に`application:didFinishLaunchingWithOptions:`メソッドの実装例を示します（『*Your Second iOS App: Storyboards*』のチュートリアルから引用）。この例で、ストーリーボードの初期View ControllerはNavigation Controllerであり、マスタビューを表示するカスタムContent Controllerが付随しています。コードはまず、該当するView Controllerの参照を検索します。次に、Interface Builderではできなかった設定を施します。この例では、カスタムデータコントローラオブジェクトをマスタView Controllerに渡しています。

リスト 11-1　アプリケーションデリゲートがコントローラの設定をする

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

    UINavigationController *navigationController = (UINavigationController*)
self.window.rootViewController;

    BirdsMasterViewController * firstViewController = [[navigationController
viewControllers] objectAtIndex:0];


   BirdSightingDataController *dataController = [[BirdSightingDataController alloc]
 init];

    firstViewController.dataController = dataController;


    return YES;

}
```

プロジェクトでメインストーリーボードを指定していなかった場合、`UIApplicationMain`関数はアプリケーションデリゲートを生成して呼び出しますが、上記のそれ以外の処理はしません。そのためのコードを別途記述する必要があります。リスト 11-2に、上記の処理をプログラムで実行する実装例を示します。

リスト 11-2　メインストーリーボードを使わずにウインドウを生成する

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"MyStoryboard"
bundle:nil];

    MainViewController *mainViewController = [storyboard
instantiateInitialViewController];
```

```
    self.window.rootViewController = mainViewController;

    // View Controllerに必要な設定を施すコード

    [self.window makeKeyAndVisible];
    return YES;
}
```

セグエにトリガがかかったときの、デスティネーションControllerの設定

セグエにトリガがかかると、iOSは次のような処理を行います。

1. デスティネーションView Controllerのインスタンスを生成します。
2. トリガがかかったセグエに関する情報をすべて保持する、セグエオブジェクトのインスタンスを生成します。

> **注意:** Popoverセグエには、デスティネーションView Controllerの制御に用いる、Popover Controllerを識別するためのプロパティがあります。

3. ソースView Controllerの`prepareForSegue:sender:`を、新しいセグエオブジェクトと、トリガをかけたオブジェクトを引数として呼び出します。
4. セグエの`perform`メソッドを呼び出して、デスティネーションControllerを画面に表示します。実際の振る舞いは、実行されるセグエの種類に依存します。たとえばモーダルセグエは、ソースView Controllerに対し、デスティネーションView Controllerを表示するよう指示します。
5. セグエオブジェクトを解放し、セグエは終了します。

ソースView Controllerの`prepareForSegue:sender:`メソッドは、デスティネーションView Controllerのプロパティ（デリゲートが実装されていればこれも含む）を設定するために必要な処理をすべて実行します。

リスト 11-3に`prepareForSegue:sender:`メソッドの実装例を示します（『*Your Second iOS App: Storyboards*』のチュートリアルから引用）。

リスト 11-3　セグエにおけるデスティネーションControllerの設定

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
```

```
    if ([[segue identifier] isEqualToString:@"ShowSightingsDetails"])
    {
        DetailViewController *detailViewController = [segue
destinationViewController];
        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];
    }

    if ([[segue identifier] isEqualToString:@"ShowAddSightingView"])
    {
        AddSightingViewController *addSightingViewController = [[[segue
destinationViewController] viewControllers] objectAtIndex:0];
        addSightingViewController.delegate = self;
    }
}
```

この実装はアプリケーションのマスタView Controllerから引用したものであり、ストーリーボードで設定された2種類のセグエを扱います。2つのセグエは`identifier`プロパティで識別します。いずれの場合も、先に示したコーディング規約に従って、まずView Controllerを検索し、次いで設定を施します。

セグエが詳細View Controllerを表示するものであれば、セグエが発生したのは、ユーザがテーブルビューの該当する行を選択したからです。この場合コードは、鳥観データを表示できるよう、必要なデータをデスティネーションView Controllerに送信します。ユーザの選択に応じて、鳥観オブジェクトをマスタView Controllerのデータコントローラから検索します。次いで、この鳥観データをデスティネーションControllerに割り当てます。

もう一方のセグエの場合に現れる「追加」View Controllerでは、新しい鳥観データをユーザが入力できるようにします。このView Controllerにデータを送る必要はありません。しかし逆に、ユーザがデータ入力を終えたとき、マスタView Controllerが鳥観データを受け取れるようにする必要があります。そのため、「追加」View Controllerが定義したデリゲートプロトコルをソースView Controllerに実装し（実装例は省略）、自分自身をデスティネーションView Controllerのデリゲートにします。

デリゲーションを使ってほかのコントローラと通信する

デリゲーションを使用するモデルでは、View Controllerに、デリゲートで実装するプロトコルを定義します。このプロトコルには、特定のアクション（「完了(Done)」ボタンのタップなど）に応じてView Controllerから呼び出されるメソッドを定義します。このメソッドを実装するのはデリゲートですたとえば、被表示View Controllerは、タスクが終了すると表示した側のView Controllerにメッセージを送ります。当該View Controllerはこれを受けて、被表示View Controllerを消去します。

デリゲーションを使って、ほかのアプリケーションオブジェクトとのやり取りを管理すれば、ほかの方法に比べて次のような利点があります。

- デリゲートオブジェクトは、View Controllerでの変更を検証したり反映したりすることができます。
- View Controllerは、デリゲートのクラスについて何も知る必要がないため、デリゲートを使用することによってカプセル化が促進されます。これによって、そのView Controllerをアプリケーションの別の部分で再利用できます。

デリゲートプロトコルの実装を示すために、“View Controllerの表示とトランジションスタイルの選択”（100 ページ）で使用したレシピ用のView Controllerの例を考えます。この例では、レシピアプリケーションが、ユーザからの新規レシピ追加要求に応答して、View Controllerを表示しました。このView Controllerを表示する前に、現在のView Controllerが自身を`RecipeAddViewController`オブジェクトのデリゲートにします。リスト 11-4に、`RecipeAddViewController`オブジェクトのデリゲートプロトコルの定義を示します。

リスト 11-4　被表示View Controllerを閉じるためのデリゲートプロトコル

```
@protocol RecipeAddDelegate <NSObject>
// キャンセルの場合はrecipe == nil
- (void)recipeAddViewController:(RecipeAddViewController *)recipeAddViewController
                    didAddRecipe:(MyRecipe *)recipe;
@end
```

新規レシピ用のインターフェイスで、ユーザが「キャンセル(Cancel)」ボタンまたは「完了(Done)」ボタンをタップすると、`RecipeAddViewController`オブジェクトは、デリゲートオブジェクトの上記のメソッドを呼び出します。次に、デリゲートはどのようなアクションを取るべきかを決定します。

リスト 11-5に、新規レシピの追加を処理するデリゲートメソッドの実装を示します。このメソッドは、`RecipeAddViewController`オブジェクトを表示したView Controllerで実装します。ユーザが新規レシピを受け入れた場合（レシピオブジェクトが`nil`でない場合）、このメソッドはそのレシピを内

部のデータ構造に追加し、テーブルビューの更新を指示します（それを受けて、テーブルビューは、ここに示したものと同じrecipesControllerオブジェクトからレシピデータを再ロードします）。続いて、デリゲートメソッドは被表示View Controllerを閉じます。

リスト 11-5　デリゲートを使用して被表示View Controllerを閉じる

```
- (void)recipeAddViewController:(RecipeAddViewController *)recipeAddViewController
                    didAddRecipe:(Recipe *)recipe {
    if (recipe) {
        // レシピをレシピコントローラに追加する。
        int recipeCount = [recipesController countOfRecipes];
        UITableView *tableView = [self tableView];
        [recipesController insertObject:recipe inRecipesAtIndex:recipeCount];

        [tableView reloadData];
    }
    [self dismissViewControllerAnimated:YES completion: nil];
}
```

View Controllerのデータを管理する上でのガイドライン

View Controller間のデータフロや制御フローを慎重に管理することが、アプリケーションの動作を理解し、複雑なエラーを避ける上で重要です。View Controllerの設計に当たっては、以下のガイドラインを念頭に置いてください。

- デスティネーションView Controllerがアプリケーションデータを参照する場合、ソースView Controllerが当該データを設定しなければなりません。ただしデスティネーションView Controllerが自己完結している（したがって独立に設定できる）場合を除きます。
- コントローラの設定はできるだけ、プログラムで行うのではなく、Interface Builderを使ってください。
- ほかのコントローラに情報を返す場合は必ずデリゲートを使ってください。Content View Controllerが、ソースView Controllerや、自分自身が生成したのではないコントローラのクラスを知っている必要はありません。
- View Controllerからほかのオブジェクトに対して、必要以上に接続を作らないでください。過度に依存関係が生じ、アプリケーションの設計を変更しにくくなります。

たとえばNavigation Controllerの子は、親であるNavigation Controllerと、スタック上で直前と直後にある兄弟について知っていれば十分です。ほかの兄弟との通信が必要になることはほとんどありません。

View Controllerの編集モードの有効化

同じView Controllerを、コンテンツの表示と編集の両方に使うことができます。編集モードに切り替えると、カスタムView Controllerは、ビューを表示モードから編集モードに遷移させるために必要な処理をすべて実行します（逆に切り替えた場合も同様）。

表示モードと編集モードの切り替え

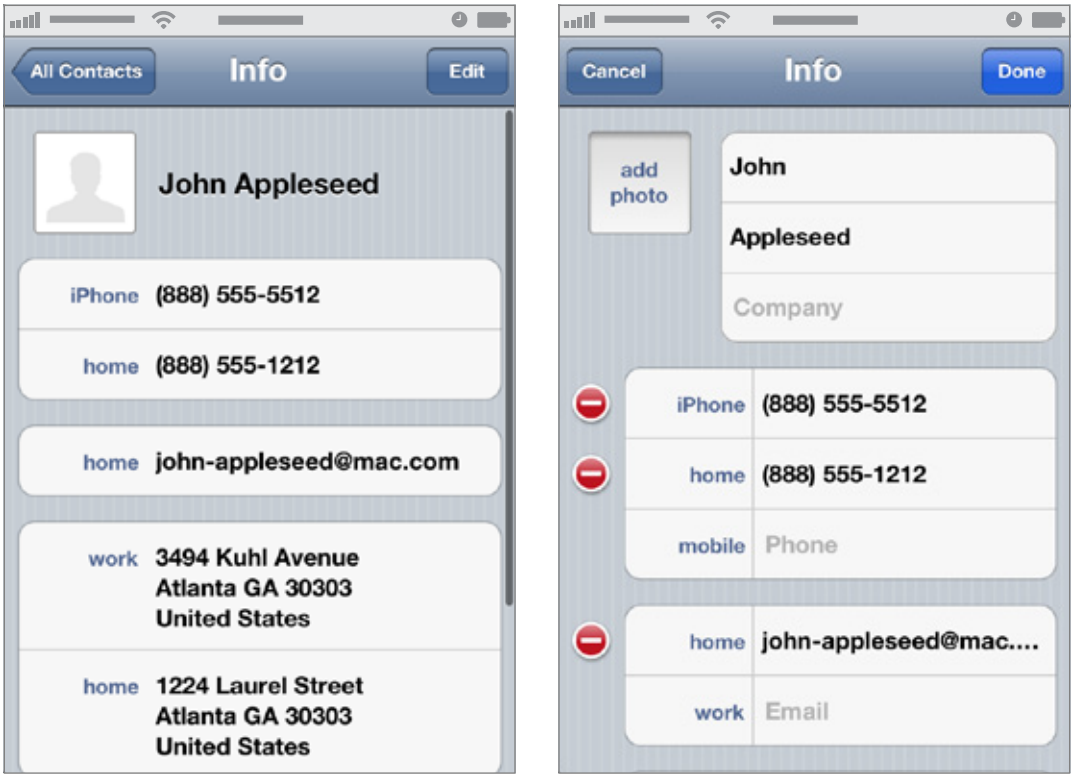
カスタムView Controllerクラスを表示モードと編集モードの両方に使うためには、`setEditing:animated:`メソッドをオーバーライドします。このメソッドの実装では、メソッドが呼び出されたときに、指定のモードに合うようにView Controllerのビューを追加、削除、および調整しなければなりません。たとえば、ビューが現在編集可能であることを伝えるために、コンテンツを変更したり、ビューの外観を変更することができます。View Controllerがテーブルを管理している場合は、そのテーブル自身の`setEditing:animated:`メソッドを呼び出して、そのテーブルを適切なモードに設定することもできます。

> **注意:** 通常は、表示モードと編集モードを切り替えるときに、ビュー階層全体を入れ替えるようなことはしません。`setEditing:animated:`メソッドを使用することのねらいは、既存のビューを少しだけ変更できる点にあります。編集用に新たなビューセットを表示する場合は、新規のView Controllerを表示するか、Navigation Controllerを使用して新規のビューを表示するかのいずれかにするべきです。

図 12-1に、 その場での編集をサポートする「連絡先(Contacts)」アプリケーションのビューを示します。右上隅の「編集(Edit)」ボタンをタップすると、View Controllerが編集モードに切り替わります。「完了(Done)」ボタンをタップすると、表示モードに戻ります。テーブルの変更だけでなく、この

ビューでは、画像ビューや、ユーザ名を表示するビューのコンテンツを変更することもできます。また、関連するビューやセルをタップすると、ほかのアクションを実行するのではなく、そのコンテンツの編集ができるように設定されています。

図 12-1 ビューの表示モードと編集モード



Display mode　　Edit mode

独自の`setEditing:animated:`メソッドの実装は比較的簡単です。View Controllerがどちらのモードに入ろうとしているかを確認し、それに応じてビューのコンテンツを調整するだけです。

```
- (void)setEditing:(BOOL)flag animated:(BOOL)animated
{
    [super setEditing:flag animated:animated];
    if (flag == YES){
        // ビューを編集モードに変更する
    }
    else {
        // 必要であれば変更を保存し、ビューを編集不可に変更する
    }
}
```

ユーザに編集オプションを示す

編集可能なビューをよく使用する場所は、ナビゲーションインターフェイス内です。ナビゲーションインターフェイスを実装する場合は、編集可能なView Controllerを表示するときに、Navigation Barに特殊な「編集(Edit)」ボタンを含めることができます（View Controllerの`editButtonItem`メソッドを呼び出すことによって、このボタンを取得できます）。このボタンは、タップされると、自動的に「編集(Edit)」ボタンと「完了(Done)」ボタンの間で切り替わり、View Controllerの`setEditing:animated:`メソッドを適切な値で呼び出します。また、独自のコードからこのメソッドを呼び出して（または、View Controllerの`editing`プロパティの値を変更して）、モードを切り替えることもできます。

Navigation Barに「編集(Edit)」ボタンを追加する方法の詳細については、『*View Controller Catalog for iOS*』を参照してください。

カスタムセグエの作成

Interface Builderでは、あるView Controllerから別のView Controllerに遷移する標準的な方法を組み込んだ、各種のセグエを提供しています。View Controllerを表示するよう働きかける、Popover内にコントローラを表示する、などの遷移が可能です。しかし、目的に合致するセグエがない場合は、カスタムセグエを作成しなければなりません。

セグエのライフサイクル

カスタムセグエの動作を理解するためには、セグエオブジェクトのライフサイクルについて知っておく必要があります。セグエオブジェクトは、`UIStoryboardSegue`またはそのサブクラスのインスタンスです。アプリケーションがセグエオブジェクトを直接生成することはありません。トリガがかかったとき、iOSが代わって生成します。実際に起こることを以下に示します。

1. デスティネーションコントローラを生成、初期化します。
2. セグエオブジェクトを生成し、`initWithIdentifier:source:destination:`メソッドを呼び出します。引数のidentifierは、Interface Builder上でセグエに与えた一意的な識別文字列です。ほかの2つの引数は、遷移元と遷移先のコントローラを表します。
3. ソースView Controllerの`prepareForSegue:sender:`メソッドを呼び出します。“セグエにトリガがかかったときの、デスティネーションControllerの設定”（109 ページ）を参照してください。
4. セグエオブジェクトの`perform`メソッドを呼び出します。このメソッドは、デスティネーションView Controllerを画面に表示するという、遷移の処理を行います。
5. セグエオブジェクトの参照を解放します（その結果、割り当てが解除されます）。

カスタムセグエの実装

カスタムセグエを実装するためには、`UIStoryboardSegue`のサブクラスを定義し、先に挙げた2つのメソッドを実装します。

- `initWithIdentifier:source:destination:`メソッドをオーバーライドする場合、まずスーパークラスの実装を呼び出した後に、サブクラス側の初期化を行います。

- `perform`メソッドでは、必要に応じてView Controllerのメソッドを呼び出し、実際に画面を遷移します。一般に、標準的な方法で新しいView Controllerを表示しますが、アニメーションその他、装飾効果を加えても構いません。

> **注意:** セグエの設定に用いるプロパティを追加しても、Interface Builderでその属性を設定することはできません。セグエにトリガをかけるソースView Controllerの`prepareForSegue:sender:`メソッドで、当該プロパティに値を設定する必要があります。

"カスタムセグエの作成"に、非常に簡単なカスタムセグエを示します。この例は、デスティネーションView Controllerを単に表示するだけであり、アニメーション効果はありません。必要に応じてアニメーションを組み込むなど拡張するとよいでしょう。

リスト 13-1 カスタムセグエ

```
- (void)perform
{
// 独自のアニメーション効果のコードをここに追加


    [[self sourceViewController] presentModalViewController:[self
destinationViewController] animated:NO];
}
```

カスタムContainer View Controllerの作成

Container View ControllerはiOSアプリケーションの設計において重要な役割を果たします。これを使って、アプリケーションをより小さく単純な部品に分割し、個別に専用のView Controllerで制御することができます。コンテナには、それぞれのView Controllerが協調して動作し、シームレスなインターフェイスを与えるようにする働きがあります。

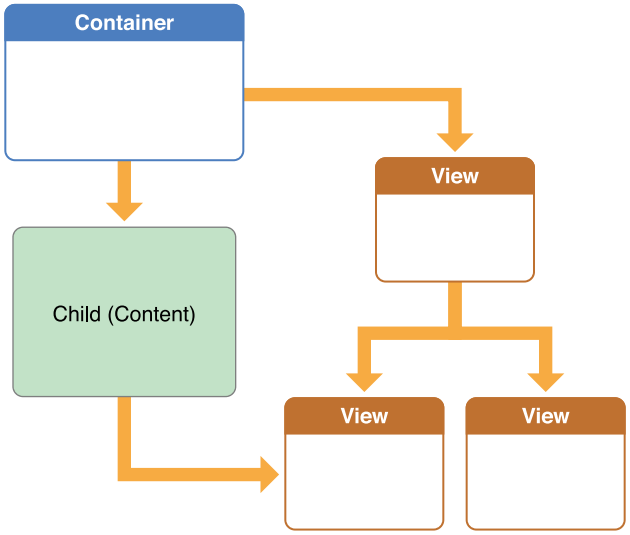
iOSには多くの標準コンテナがあり、アプリケーションの編成に活用できます。しかし、どの標準コンテナにも適合しない、独自のワークフローを作成しなければならないことがあります。子に当たるView Controller群を特殊な方法で編成し、特別なナビゲーションジェスチャや、アニメーションによる遷移効果を組み込みたい、というような場合です。そのためにはカスタムコンテナを実装しなければなりません。

Container View Controllerの設計

Container View Controllerはさまざまな面で、Content View Controllerに似ています。ビューとコンテンツを管理し、他のオブジェクトと連携し、レスポンダチェーンに沿ってイベントに応答します。したがって、Container View Controllerを設計するためには、Content View Controllerの設計方法も知っておかなければなりません。“カスタムContent View Controllerの作成”（46 ページ）で説明した設計上の問題は、Container View Controllerにも適用されます。

コンテナを設計する際には、コンテナを親、他のView Controllerを子とする、親子関係を明示的に設定しなければなりません。より具体的に、ビュー間も明示的に接続されている様子を図14-1に示します。コンテナは、自身のビュー階層に、他のView Controllerのコンテンツビューを追加します。コンテナのビュー階層に含まれている子のビューを表示する際には、子View Controllerへの接続も確立して、View Controllerが処理するべきイベントがすべて子に渡るようにしなければなりません。

図 14-1 Container View Controllerのビュー階層に他のコントローラのビューが入っている様子



Yourコンテナ（親）には、子が従うべき規則を決めなければなりません。ビュー階層上にある子のコンテンツをいつ可視にするか、もっぱら親が決めることになります。階層上のどこにビューを置くか、大きさや位置をどうするか、を決めるのもコンテナ側です。この設計原則は、Content View Controllerの場合とまったく同じです。View Controllerは自身のビュー階層を管理し、他のクラスがそのコンテンツを操作することはありません。コンテナクラスは、必要ならばメソッドやプロパティを公開して、外部から動作を制御できるようにすることも可能です。たとえば、他のオブジェクトがコンテナに対して新しいビューを表示するよう指示する必要がある場合、コンテナクラスはこの遷移を起こさせるための公開メソッドを用意しなければなりません。ビュー階層を変更する実際の実装は、コンテナクラス内に行います。この原則に従えば、コンテナとその子の責任を明確に分離できます。それぞれのView Controllerが常に、自身のビュー階層に責任を負うことになるからです。

コンテナクラスに関しては、次の事項を理解しておかなければなりません。

- コンテナの役割、その子の役割は何か。
- 兄弟間には関係があるか。
- 子であるView Controllerをコンテナに追加し、あるいはコンテナから削除するにはどうすればよいか。コンテナクラスには、子を表示するための公開プロパティや公開メソッドが必要です。
- コンテナは子をいくつ表示できるか。

- コンテナの内容は静的か動的か。静的設計の場合、子は多かれ少なかれ固定されているのに対し、動的設計の場合は兄弟間の遷移が発生します。したがって、どのようなきっかけで新しい兄弟への遷移が発生するか、定義する必要があります。プログラム的に発生するのでも、ユーザがコンテナを操作することにより発生するのでも構いません。
- コンテナ自身が独自のビューを持つか。たとえば、コンテナ側のユーザインターフェイスとして、子であるView Controllerに関する情報や、ナビゲーション用のコントロール部品を表示することが考えられます。
- 子の側に、`UIViewController`クラスに定義されている以外のメソッドやプロパティを実装し、コンテナ側がこれを呼び出す/アクセスする必要があるか。実際にその必要が生じる状況はさまざま考えられます。子の側のある情報に基づいて、コンテナ側の表示を制御し、あるいはコンテナの動作を変えるという状況です。さらに、コンテナ側で何らかのイベントが発生したときに、子の側のメソッドを呼び出すということもありえます。
- コンテナの動作を外部から変更できるようにするか。
- 子をすべて同等に扱うか、それともいくつか種類を設けて扱い方を変えるか。たとえば2つの子を表示し、相互のアクションをコンテナがうまく連携させる、ということが考えられます。子にはそれぞれの役割に応じたメソッドを実装して、コンテナがその動作を調整できるようにしなければなりません。

以上のように、Container View Controllerは多くの場合、Content View Controllerに比べて、他のオブジェクトとさまざまな関係を持ちます。したがって、コンテナの動作を理解するためには、より多くの時間がかかるかも知れません。Content Controllerのときと同様に、公開のクラスAPIによって隠蔽されている実装詳細は、とりあえず無視して考えるとよいでしょう。

よく使われるコンテナの設計例

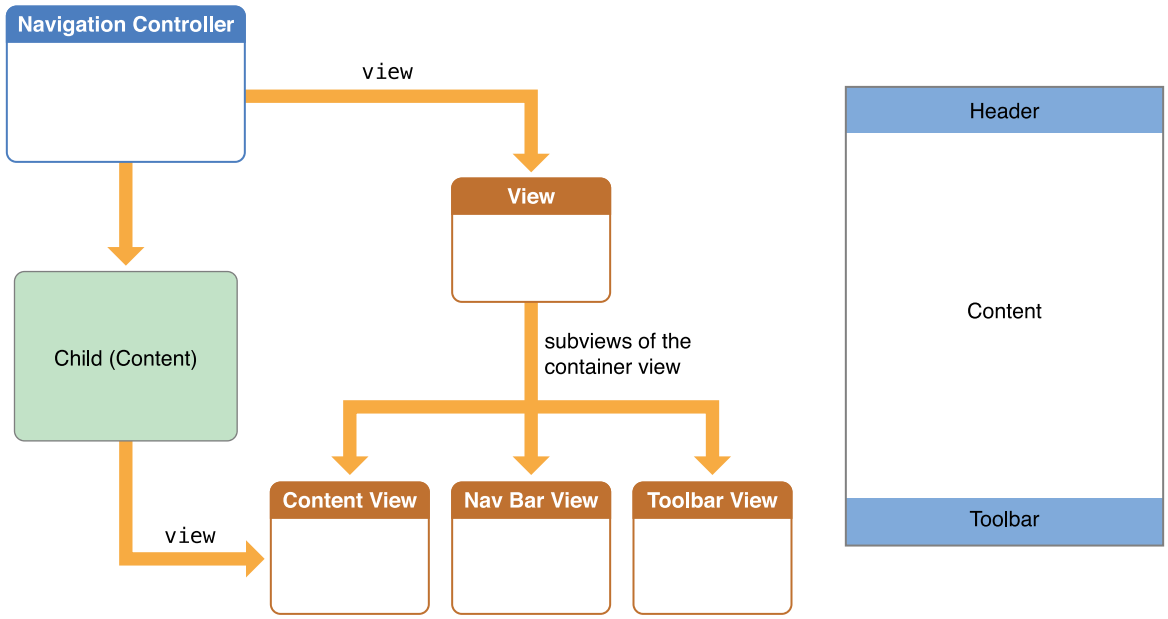
新しいコンテナクラスの設計方法を理解する近道は、既存のシステムコンテナクラスの動作や公開APIを試してみることです。それぞれが、独自のナビゲーションメタファや、設定に用いるプログラミングインターフェイスを定義しています。この節では、コンテナの設計という観点から、いくつかのクラスを見ていきます。各クラスのプログラミングインターフェイスをすべて説明することはしません。重要な考え方を紹介するだけです。ここで取り上げるシステムコンテナについて詳しくは、『*View Controller Catalog for iOS*』を参照してください。

Navigation Controllerは子（View Controller）をスタックで管理する

Navigation Controllerには、ひと続きのユーザインターフェイス画面を、ユーザに向けて順に表示する、という働きがあります。ここに使われているメタファは、子（View Controller）を積み上げたスタックです。最上位に当たるView Controllerのビューが、Navigation Controllerのビュー階層に配置されます。新しいView Controllerを表示したければ、これをスタックにプッシュすることになります。終了後は当該View Controllerをスタックから削除します。

図 14-2に、あるひとつの子のビューが可視になっている状況を示します。子のビューは、NavigarionControllerが管理する、より複雑なビュー階層の一部になっています。

図 14-2 Navigation ControllerのビューとView Controller階層



View Controllerをスタックにプッシュし、あるいはスタックからポップすると、アニメーション効果を伴う画面遷移が発生します。すなわち、少しの間、2つの子のビューがどちらも表示された状態になります。Navigation Controllerには、子のビューのほかに、ナビゲーションバーを表示するために用いる独自のコンテンツビューもあります。ナビゲーションバーの内容は、現在画面に現れている子に応じて更新されます。

`UINavigationController`クラスの動作を定義する、主なメソッドやプロパティを以下に示します。

- `topViewController`プロパティは、スタックの最上位にあるView Controllerを表します。
- `viewControllers`プロパティは、スタックに積み上げられた子（View Controller）すべてのリストです。
- `pushViewController:animated:`メソッドは、新しいView Controllerをスタックにプッシュします。新しい子のビューが表示されるよう、ビュー階層を更新する処理をすべて行います。

- `popViewControllerAnimated:`メソッドは、スタックの最上位にあるView Controllerを削除します。
- `delegate`プロパティを使えば、状態遷移が起こったとき、コンテナのクライアントに通知が届くようにすることができます。

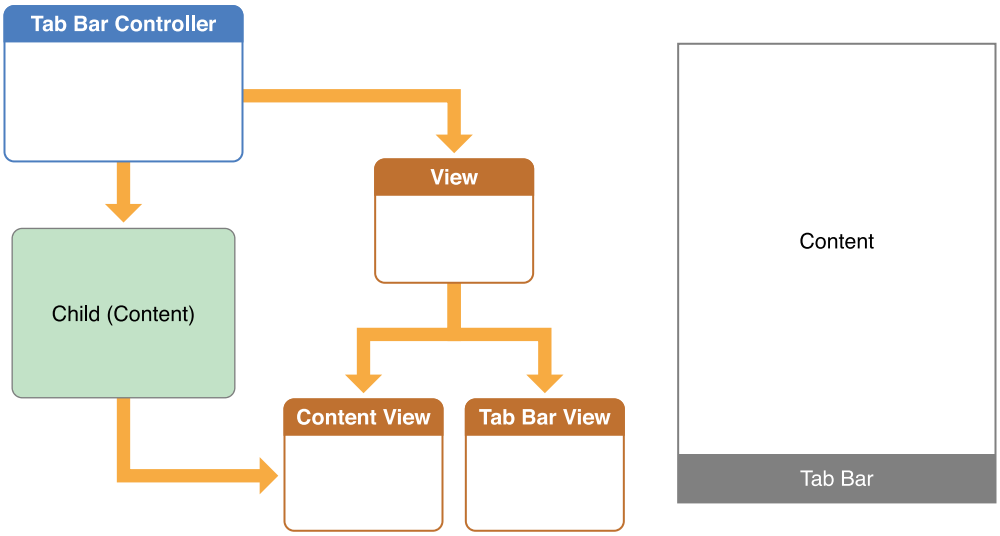
Navigation Controllerは表示内容を判断するために、子であるView Controllerのプロパティを参照します。これはあらゆるView Controllerの基底クラスである`UIViewController`に定義されているプロパティであり、デフォルトの動作が決まっています。したがって、View Controllerであれば何でも、Navigation Controllerの子にすることができます。具体的には、たとえば次のようなプロパティです。

- `navigationItem`プロパティは、ナビゲーションツールバーの内容を表します。
- `toolbarItems`プロパティは、ボトムバの内容を表します。
- `editButtonItem`プロパティは、これを介してナビゲーション項目のビューにアクセスできるようになっています。Navigation Controllerはこれを使って、子（ビュー）の編集モードを切り替えます。

Tab Bar Controllerは子（View Controller）をコレクションで管理する

Tab Bar Controllerには、独立したいくつかのユーザインターフェイス画面を切り替えながら、ユーザに向けて表示する、という働きがあります。先の例と違い、子（View Controller）を、スタックではなく単純な配列を使って管理します。図 14-3も、先に示した図と同じく、あるひとつの子（ViewController）のビューが可視になっている状況を示します。しかし今回は、順序通りに表示するとは限りません。また、画面遷移は通常、アニメーション効果を伴いません。

図 14-3　Tab Bar ControllerのビューとView Controller階層



`UITabBarController`クラスの主なメソッドやプロパティを以下に示します。アプリケーションはこれを使って、Tab Bar Controllerが何を表示するかを制御します。

- `viewControllers`プロパティは、コンテンツのタブとして動作する、子（View Controller）のリストを保持します。
- `selectedViewController`プロパティは、現在可視になっている子を調べ、あるいは変更するために使います。
- `delegate`プロパティを使えば、状態遷移が起こったとき、コンテナのクライアントに通知が届くようにすることができます。

Tab Bar Controllerは子の`tabBarItem`プロパティを使って、対応するタブに、どのように表示するかを判断します。

Page View Controllerはデータ源を使って新しい子（View Controller）に切り替える

Page View Controllerはコンテンツのページを、メタファとして書籍のページのように扱います。コンテナが表示する各ページの内容は、子であるView Controllerが供給します。

書籍は何ページにもわたる（Navigation Controllerで実装したとすれば目次の画面数をはるかに超える）ことがありうるので、全体がメモリ内に納まるとは限りません。そこでPage View Controllerは、可視になっているページに当たるView Controllerを子として保持し、他のページは必要になってから取得するようになっています。ユーザが新しいページを開こうとすると、コンテナは`dataSource`プロパティに保持しているオブジェクトを呼び出して、新しいView Controllerを取得します。すなわち、Page View Controllerは、データ源を使って新しいページを「引っ張ってくる（プル）」というモデルを採用しています。アプリケーションが直接、Page View Controllerに新しいページをプッシュするのではありません。

Page View Controllerは、書籍のさまざまなレイアウトに合わせてカスタマイズできます。ページ数や各ページの大きさも違っていて構いません。Page View Controllerの動作を制御する主なプロパティとして、次の2つがあります。

- `spineLocation`プロパティはページの編成方法を表します。同時に1ページしか表示しないレイアウトがある一方で、複数ページを表示するレイアウトも可能です。
- `transitionStyle`プロパティはページをめくる際のアニメーションを表します。

カスタムContainer View Controllerを実装する

Container View Controllerの動作を設計し、公開APIの詳細を決めれば、いよいよ実装作業を開始できます。実装の目標は、他のView Controllerのビュー（および対応するビュー階層）を、コンテナ側に、ビュー階層の部分木として追加できるようにすることです。子の側のビュー階層は、コンテナに追加してもやはり子が管理します。ただし、画面上の描画位置はコンテナ側が決めます。子のビューを追加する際には、イベントが親子どちらにも伝わるようにする必要があります。これは、新しいView Controllerを、コンテナの子として明示的に関連づけることにより行います。

`UIViewController`クラスには、Container View Controllerが自分自身とその子との関係を管理するために使うメソッドがあります。具体的なメソッドやプロパティの一覧は、「Managing Child View Controllers in a Custom Container」 in *UIViewController Class Reference*を参照してください。

Important: 上に触れた`UIViewController`のメソッドは、もっぱらContainer View Controllerを実装するために使うことを意図したものです。Content View Controllerでは使わないでください。

子の追加と削除

リスト 14-1に、あるView ControllerをContainer View Controllerの子として追加する、典型的な実装例を示します。コード例を示した後、番号をつけた行について詳しく解説します。

リスト 14-1 他のView Controllerのビューをコンテナのビュー階層に追加するコード例

```
- (void) displayContentController: (UIViewController*) content;
{
   [self addChildViewController:content];                 // 1
   content.view.frame = [self frameForContentController]; // 2
   [self.view addSubview:self.currentClientView];
   [content didMoveToParentViewController:self];          // 3
}
```

このコードが実行することを以下に示します。

1. コンテナの`addChildViewController:`メソッドを呼び出して、子を追加します。`addChildViewController:`メソッドを呼び出せば、自動的に子の`willMoveToParentViewController:`メソッドも呼び出されます。
2. 子の`view`にアクセスしてビューを取得し、自身のビュー階層に追加します。コンテナは、子の大きさと位置を設定した後で、ビューを追加するようになっています。子のコンテンツの表示位置を選ぶのは、常にコンテナ側の役割です。このコード例では明示的にフレームを設定していますが、レイアウト制約を使ってビューの位置を決めることも可能です。

3. 子の`didMoveToParentViewController:`メソッドを明示的に呼び出して、処理が終了した旨のシグナルを送出します。

子のビューをビュー階層から削除できるようにしたい場合もあります。これは、リスト 14-2のように、追加とは逆の順で処理します。

リスト 14-2 他のView Controllerのビューをコンテナのビュー階層から削除するコード例

```
- (void) hideContentController: (UIViewController*) content
{
   [content willMoveToParentViewController:nil];  // 1
   [content.view removeFromSuperview];            // 2
   [content removeFromParentViewController];      // 3
}
```

このコードが実行することを以下に示します。

1. 子の`willMoveToParentViewController:`メソッドを、引数として`nil`を渡して呼び出し、削除されようとしている旨を子に通知します。
2. ビュー階層から削除します。
3. 子の`removeFromParentViewController`メソッドを呼び出して、コンテナから削除します。`removeFromParentViewController`メソッドを呼び出すと、子の`didMoveToParentViewController:`メソッドが自動的に呼び出されます。

静的な内容のみから成るコンテナであれば、View Controllerの追加や削除は単純です。追加する場合は、まずView Controller、その後でビューを追加します。逆に削除の場合は、ビューの後でView Controllerという順序にすればよいのです。しかし、それまで表示されていた子が消えていくのと同時に、新しい子が画面に現れるよう、アニメーション表示したいこともあるでしょう。リスト 14-3にその実装例を示します。

リスト 14-3 2つのView Controller間を遷移するコード例

```
- (void) cycleFromViewController: (UIViewController*) oldC
            toViewController: (UIViewController*) newC
{
    [oldC willMoveToParentViewController:nil];                          // 1
    [self addChildViewController:newC];
```

```
    newC.view.frame = [self newViewStartFrame];                          // 2
    CGRect endFrame = [self oldViewEndFrame];

    [self transitionFromViewController: oldC toViewController: newC    // 3
          duration: 0.25 options:0
          animations:^{
             newC.view.frame = oldC.view.frame;                          // 4
             oldC.view.frame = endFrame;
           }
           completion:^(BOOL finished) {
             [oldC removeFromParentViewController];                      // 5
             [newC didMoveToParentViewController:self];
            }];
}
```

このコードが実行することを以下に示します。

1. 両方のView Controllerについて、遷移を開始します。
2. 遷移アニメーションに使う、2つの新しいフレーム位置を計算します。
3. `transitionFromViewController:toViewController:duration:options:animations:completion:` メソッドを呼び出して、入れ替えを行います。このメソッドは自動的に、新しいビューを追加し、アニメーション表示を行い、古いビューを削除します。
4. ビューが入れ替わる様子をアニメーション表示する、1フレーム分の処理に当たります。
5. 遷移が終了すると、ビュー階層は最終的な状態になるので、最後に通知を2つ送信して処理を終えます。

外観や回転に関するコールバックの動作をカスタマイズする

子（View Controller）をコンテナに追加すると、回転や外観に関するイベントが発生したとき、コンテナは必要に応じて自動的に、子にメッセージを転送するようになります。一般には、イベントがすべて適切に送信されるようになるので、これは望ましい動作と言えるでしょう。しかし場合によっては、コンテナの特性上、不適切な順序でイベントが配送されてしまうことがあります。たとえば、複数の子のビュー状態が同時に変化する場合、これを統合して、外観に関するコールバックが論理的に適切な順序で呼び出されるようにしたいことがあります。これを実現するには、外観や回転に関するコールバックの呼び出しを、コンテナクラス側が制御するよう修正すればよいことになります。

外観に関するコールバックの制御権を取得するには、`shouldAutomaticallyForwardAppearanceMethods`メソッドをオーバーライドして、`NO`を返すようにします。リスト 14-4にそのコード例を示します。

リスト 14-4 外観に関するコールバックの自動転送を無効にするコード例

```
- (BOOL) shouldAutomaticallyForwardAppearanceMethods
{
    return NO;
}
```

子（View Controller）に対して実際に、外観の遷移が発生している旨を通知するためには、子の`beginAppearanceTransition:animated:`および`endAppearanceTransition`メソッドを使います。

コールバックの制御権を取得して上記のメッセージを送信することにした場合、Container View Controllerが現れたり消えたりした場合にも、メッセージを子に転送する責任を負います。たとえば、コンテナに子が1つあり、それが`child`プロパティとして参照されている場合、コンテナがメッセージを子に転送するコードはリスト 14-5のようになります。

リスト 14-5 コンテナの出現/消滅時に、外観に関するメッセージを転送するコード例

```
-(void) viewWillAppear:(BOOL)animated
{
    [self.child beginAppearanceTransition: YES animated: animated];
}

-(void) viewDidAppear:(BOOL)animated
{
    [self.child endAppearanceTransition];
}

-(void) viewWillDisappear:(BOOL)animated
{
    [self.child beginAppearanceTransition: NO animated: animated];
}

-(void) viewDidDisappear:(BOOL)animated
```

```
{
    [self.child endAppearanceTransition];
}
```

回転イベントの転送もほぼ同様であり、外観メッセージの転送とは独立に行うことができます。まず、`shouldAutomaticallyForwardRotationMethods`メソッドをオーバーライドして、`NO`を返すようにします。次に、コンテナ側の判断により、適切な時点で次のメソッドを呼び出してください。

- `willRotateToInterfaceOrientation:duration:`
- `willAnimateRotationToInterfaceOrientation:duration:`
- `didRotateFromInterfaceOrientation:`

Container View Controllerの構築に関する実践的なヒント

Container View Controllerを新たに設計、開発、テストするには、相応の時間がかかります。個々の動作は単純でも、全体としてみれば非常に複雑なものになるからです。独自のコンテナクラスを実装する際には、次の事項を検討してください。

- 最初はContent View Controllerとして使うことを想定し、コンテナが所有する通常のビューを使って設計してください。親子関係を管理する必要がないので、レイアウトやアニメーション効果を正しく実装することに集中できます。
- 子（View Controller）の最上位のビュー以外にはアクセスしないでください。子も同様に、親がビューに対して何をするか、最小限の事項しか意識しないようにします。親は必要以上の実装詳細を子に見せないでください。
- コンテナの動作の都合上、子がメソッドやプロパティを宣言しなければならない場合は、次のように、プロトコルを定義して子に強制します。

```
@protocol MyContentContainerProtocol <NSObject>
...
@end
- (void) displayContentController:
(UIViewController<MyContentContainerProtocol>*) content;
```

書類の改訂履歴

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

日付	メモ
2012-12-13	表示関連の通知に関する図（2枚）を修正しました。
2012-09-19	カスタムContainer View Controllerや、ビュー階層におけるアクセシビリティについて、設計ガイドラインを追加しました。iOS 6に合わせて、ビューの回転、ビューのレイアウト、リソース管理に関する記述を更新しました。
2012-02-16	わかりやすくするために表現を編集しました。用語解説の項目を追加しました。View Controllerのビューが現れた/消えた理由の判断方法に関する節を追加しました。
2012-01-09	ストーリーボードやARCを使って新規iOSアプリケーションを構築する手順を新たに書き起こしました。
2011-01-07	誤字をいくつか訂正しました。
2010-11-12	iPad専用のコントローラオブジェクトについての情報を追加しました。
2010-07-08	『iPhone OS View Controllerプログラミングガイド』からドキュメント名を変更しました。
2010-05-03	誤字をいくつか訂正しました。
2010-02-24	誤字をいくつか訂正し、2ステップの回転処理の図を更新しました。

日付	メモ
2009-10-19	iOS 3.0の変更に対応するため、内容の書き換えと拡充を行いました。
2009-05-28	iOS 3.0でサポートされていないことについての注記を追加しました。
2008-10-15	『iOS Programming Guide』への古い参照を更新しました。
2008-09-09	誤字を訂正しました。
2008-06-23	View Controllerを使用してラジオインターフェイス、ナビゲーションインターフェイス、およびモーダルインターフェイスを実装する方法について説明した新規ドキュメント。

用語解説

コンテナView Controller（container view controller） 特定のタイプのユーザインターフェイスを表示するために、ほかのView Controllerとのやり取りを調整するView Controller。

Content View Controller 何らかのコンテンツを画面に表示するView Controller。

カスタムセグエ（custom segue） セグエのうち、遷移効果をカスタムサブクラスで定義したもの。

モーダルセグエ（modal segue） セグエのうち、既存のView Controllerを使って新しいView Controllerを表すよう、遷移効果を実装したもの。

Navigation Controller 階層コンテンツを表すために用いるContainer View Controller。

ナビゲーションインターフェイス（navigation interface） Navigation Controllerのビューによって表示されるインターフェイスのスタイル。ナビゲーションインターフェイスには、異なる画面間の移動をサポートするナビゲーションバーが画面の上端に沿って表示されます。

ナビゲーションスタック（navigation stack） Navigation Controllerによって現在管理されているView Controllerのリスト。スタック上のView Controllerは、ナビゲーションインターフェイスに現在表示されているコンテンツを表します。

Page View Controller Container View Controllerのうち、物理的な書籍に似た様式で、コンテンツの各ページを表示するもの。

Popover Controller 他のView Controllerのビューをpopover Control内に表示するために用いるコントローラクラス。

Popoverセグエ（popover segue） セグエのうち、新しいView Controllerのコンテンツをpopover Controlに表示する形で遷移効果を実装したもの。

プッシュセグエ（push segue） セグエのうち、新しいView ControllerをNavigation Controllerのスタックにプッシュすることにより遷移効果を実装したもの。

ルートView Controller（root view controller） View Controller階層の最上位に位置するView Controller。

シーン（scene） View Controllerおよびこれに関連するオブジェクト（表示の際にロードされるビューを含む）を、Interface Builder上で可視的に表したもの。

セグエ（segue） 2つのシーン間の遷移（Interface Builder上で設定）。

Split View Controller iPadアプリケーションで使われるContainer View Controllerで、マスタと詳細が対になったインターフェイスを表すもの。

Tab Bar Controller Container View Controllerのうち、独立したインターフェイス画面をそれぞれタブの形で表し、個々のContent View Controllerを切り替えるようにしたもの。

Tab Barインターフェイス（tab bar interface） Tab Bar Controllerのビューによって表示されるインターフェイスのスタイル。Tab Barインター

フェイスには、画面の下端に1つ以上のタブが表示されます。タブをタップすると、現在表示されている画面のコンテンツが変化します。

View Controller `UIViewController`クラスから派生したオブジェクト。View Controllerは、一連のビューとそれらのビューによって表示されるカスタムデータとのやり取りを調整します。

View Controller階層（view controller hierarchy） 一連のContainer View ControllerやContent View Controllerを、包含関係に基づいて木構造に配置したもの。葉でないノードは常にContainer View Controllerを表します。

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

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/