iPhone OS Table Viewプログラミングガイド

iPhone > User Experience

2009-08-19

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/

目次

図、リスト

はじめに

Table Viewは、iPhoneアプリケーション、特に生産性型アプリケーションで、よく目にします。これらは、さまざまなニーズに適用できる多様なユーザインターフェイスオブジェクトです。テーブルは、スクロール可能な項目（行）のリストを表示します。このリストは、いくつかのセクションに分割されている場合もあります。各行には、表現するデータ項目の文字列、画像、その他の識別子を表示できます。ユーザが、あるテーブルの行を選択すると、別のTable Viewが表示されて、最初のTable Viewで選択した項目に関連する項目の一覧が表示されるようにできます。項目の詳細を表示するTable Viewと組み合わせて、項目を表示することもできます。このように、Table Viewは、階層的なデータをたどるための理想的なメカニズムです。また、Table Viewをインデックス付きリストとして使用すると、インデックス指定によってすばやく項目にアクセスできます。選択肢リストとして使用すると、デスクトップシステムにおけるポップアップリストやラジオボタンをシミュレートできます。

『*iPhone OS Table View プログラミングガイド*』では、Table Viewプログラミングの基礎となる概念について説明し、プロジェクト内でTable Viewを作成したり管理したりする方法を示します。まずテーブルのスタイルと特徴のほか、Table Viewに関連するプログラミングインターフェイスの概要を示すことから始めます。また、階層的なデータモデルのさまざまなレベルにあるデータの集合を、一連のTable Viewにマップする方法と、そのためにデータ階層を渡り歩く操作をサポートする特定のUIKitクラスの使いかたについて、一般的な言葉で説明します。概念的な背景を説明した後は、Table Viewプログラミングのさまざまな側面について解説します。これには、Table Viewの作成と設定、Table View Cellのカスタマイズ、選択の管理、Table Viewでの行の挿入、削除、および移動などが含まれます。

この書類の構成

この文書は次の章で構成されています。

1. 「Table Viewとは」（11 ページ）では、Table Viewの特徴と、それを使用する状況について概説します。
2. 「Table View APIの概要」（21 ページ）では、アプリケーションにTable Viewを表示するために使用するクラスを紹介します。
3. 「Table Viewを利用したデータ階層のナビゲーション」（23 ページ）では、ナビゲーションベースのアプリケーション内でデータの階層をTable Viewにマップする方法と、そのようなアプリケーションを実装する際のNavigation ControllerアーキテクチャとTable View Controllerの使いかたを説明します。
4. 「Table Viewの作成と設定」（33 ページ）では、ユーザインターフェイスにTable Viewを作成して設定するためのアプローチについて説明します。
5. 「Table View Cellの詳細」（49 ページ）では、セルオブジェクトについて説明し、Table View用にセルオブジェクトのコンテンツを準備する方法と、アプリケーションに合わせてセルオブジェクトをカスタマイズする方法を説明します。

6. 「選択の管理」（69 ページ）では、Table Viewで選択を管理する方法を説明します。
7. 「行やセクションの挿入と削除」（73 ページ）では、ユーザによる行の挿入と削除を管理する方法を説明します。
8. 「行の並べ替えの管理」（81 ページ）では、ユーザによる行の並べ替えがどのように行われるかを説明し、これらの変更を追跡する方法を示します。

関連項目

この文書を読む前に、『*iPhone Application Programming Guide*』を読んで、iPhoneアプリケーション開発の基本プロセスを理解しておきましょう。また、View Controllerの全般的な情報、およびTable Viewと頻繁に組み合わせて使用されるNavigation Controllerの詳細情報を知りたい場合は、『*View Controller Programming Guide for iPhone OS*』を読むことも検討してください。

次のサンプルコードプロジェクトは、自分でTable Viewを実装する際の参考になります。

- *TableViewSuite*プロジェクト
- *TheElements*プロジェクト
- *SimpleDrillDown*プロジェクト

Table Viewとは

Table Viewは、iPhone OSアプリケーションでよく使われるユーザインターフェイスオブジェクトです。多目的に使用できるため、アプリケーションは、さまざまな用途にTable Viewを採用できます。この章では、Table Viewの部品、特徴、およびスタイルについて、機能とプログラムの両面から解説します。また、その中で、Table Viewを使用する状況をいくつか示します。

注： この章で紹介する情報は、『*iPhone Human Interface Guidelines*』のTable Viewに関する情報を要約したものです。ここでは、Table Viewのユーザレベルの機能、特にTable View APIに関連する機能について説明します。また、Table Viewに固有の用語、概念、および使用パターンについても紹介します。Table Viewのスタイルと特徴の完全な説明と、お勧めの使い方については、『*iPhone Human Interface Guidelines*』の「Table Views, Text Views, and Web Views」を参照してください。

Table Viewの特徴

Table Viewは、スクロール可能な項目リストを表示します。Table Viewには列が1つだけあります。したがって、本質的にはリストです。また、垂直方向のスクロールのみが可能です。アプリケーションは、以下のように、さまざまな用途でTable Viewを使用できます。

- 階層構造のデータ内をユーザが移動できるようにするため
- 選択可能な選択肢リストを表示するため
- インデックス付きの項目リストを表示するため
- 視覚的にグループ分けされた詳細情報とコントロールを表示するため

Table Viewは、いくつかのセクションに分けられた行で構成されます。各セクションには、テキストまたは画像を表示するヘッダとフッタがあります。ただし、セクションを1つしか持たず、ヘッダやフッタを表示しないTable Viewもたくさんあります。プログラムでは、UIkitは、インデックス番号によって行とセクションを識別します。セクション番号は、Table Viewの一番上から下に向かって、0から*n*-1まで付けられます。行番号は、1つのセクション内で0から*n*-1まで付けられます。Table Viewは、セクションのヘッダとフッタとは別に、それ自身のヘッダとフッタを持つことができます。テーブルヘッダは、最初のセクションの先頭行の前に表示されます。テーブルフッタは、最後のセクションの最終行の後に表示されます。

Table Viewに表示される行は、セルで構成されます。セルには、テキスト、画像、その他の種類のコンテンツが表示されます。セルには、コントロールとしての機能を持つことが多いアクセサリビューを付加することもできます。次の3つの標準アクセサリビューがあります（accessory-type定数も示します）。

標準アクセサリビュー	説明
	**ディスクロージャインジケータ**—`UITableViewCellAccessoryDisclosureIndicator`。あるセルを選択すると、データモデル階層の次のレベルを反映した別のTable Viewが表示されるようにする場合に、ディスクロージャインジケータを使用します。
	**詳細ディスクロージャボタン**—`UITableViewCellAccessoryDetailDisclosureButton`。あるセルを選択すると、その項目の詳細ビュー（Table Viewであるとは限らない）が表示されるようにする場合に、詳細ディスクロージャボタンを使用します。
	**チェックマーク**—`UITableViewCellAccessoryCheckmark`。ある行をクリックすると、その項目が選択されるようにする場合に、チェックマークを使用します。この種のTable Viewは、選択リストと呼ばれ、ポップアップリストに似ています。選択リストでは、選択できる行の数を1行に制限したり、複数の行にチェックマークを付加できるようにしたりできます。

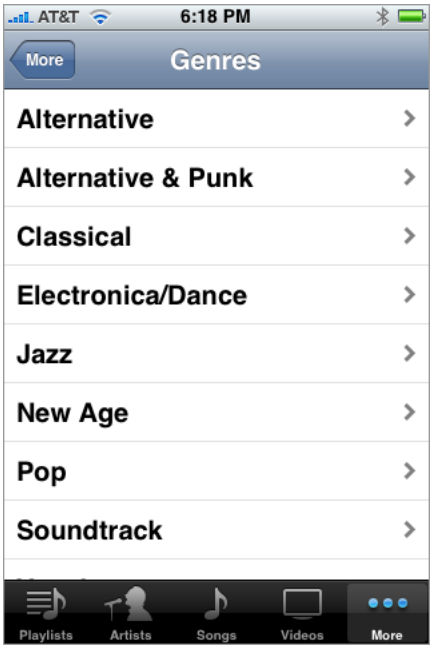
標準のアクセサリビュー(accessory-view)タイプの代わりに、コントロールオブジェクト（スイッチなど）やカスタムビューを、アクセサリビューとして指定することもできます。

Table Viewは、編集モードに入ることができます。編集モードでは、ユーザが行を挿入または削除したり、テーブル内で行を移動したりできます。編集モードのとき、挿入または削除の対象としてマークされた行の左端には、緑のプラス記号（挿入の場合）、または赤のマイナス記号（削除の場合）が表示されます。ユーザが削除コントロールにタッチしたり、Table Viewによっては、行を横切ってスワイプしたりすると、ユーザにその行の削除を確認する赤い「削除(Delete)」ボタンが表示されます。移動可能な行には、その右端に数本の水平線で構成された画像が表示されます。Table Viewが編集モードを終了すると、挿入、削除、並べ替えのコントロールは消えます。

Table Viewのスタイル

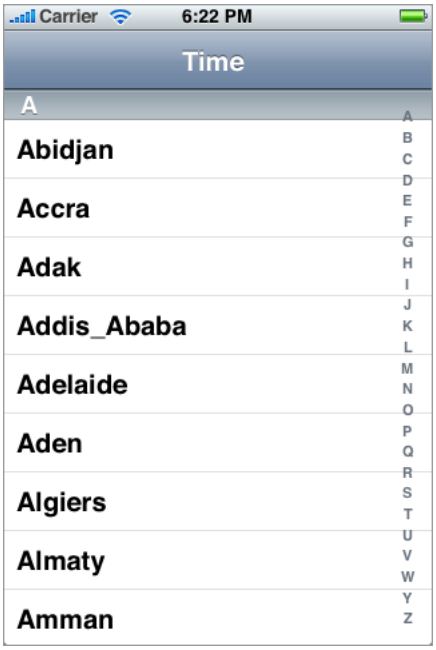
Table Viewには、プレーンとグループの2つの主要なスタイルがあります。プレーン（標準の）スタイルのTable Viewには、画面の横幅いっぱいに広がった、白い背景を持つ行が表示されます（図 1-1を参照）。Table Viewは、1つ以上のセクションを持ち、セクションは1つ以上の行を持ちます。また、各セクションは、それぞれ固有のヘッダタイトルやフッタタイトルを持つことができます（ヘッダやフッタは、カスタムビュー（たとえば、画像を含むビュー）を持つこともできます）。何行もあるセクションをスクロールする場合、セクションのヘッダは、Table Viewの上端に固定され、セクションのフッタは下端に固定されます。

図 1-1 プレーンスタイルのTable View（セクションのヘッダとフッタがない場合）



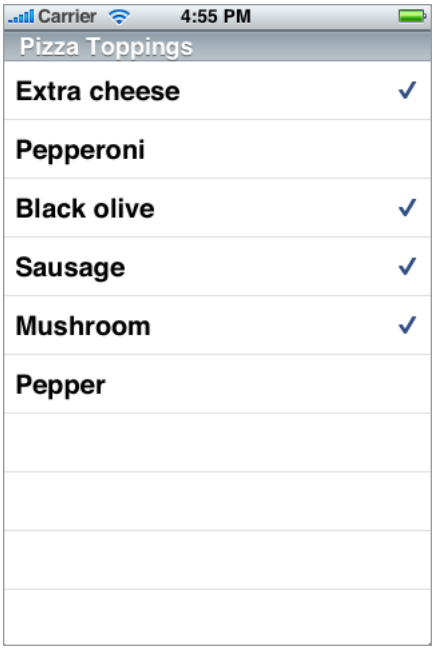
プレーンスタイルのTable Viewのバリエーションの1つに、すばやく検索ができるように、セクションにインデックスを関連付けたものがあります。図 1-2に、インデックス付きリストと呼ばれるこの種のテーブルの一例を示します。インデックスは、Table Viewの右端に上から下に向かって表示され、このインデックスの各エントリは、セクションヘッダのタイトルに対応しています。インデックス内の項目をタッチすると、Table Viewは、その項目に対応するセクションまでスクロールします。たとえば、セクションヘッダを州を表す2文字の略語に設定し、セクション内の各行がその州にある都市を示すとします。インデックス内の任意の場所をタッチすると、選択した州の都市が表示されます。インデックス付きセクションリスト内の各行には、インデックスの妨げになるため、ディスクロージャインジケータや詳細ディスクロージャボタンを付けないようにします。

図 1-2 セクションインデックスとして設定されたTable View



最も単純なTable Viewは、選択（ラジオ）リストです（図 1-3を参照）。選択リストは、ユーザが選択可能な選択肢のメニューを表示するプレーンスタイルのTable Viewです。選択できる行数を1行に制限したり、複数行の選択を許可したりできます。選択リストでは、選択された行にチェックマークが付きます（図 1-3を参照）。

図 1-3 選択リストとして設定されたTable View



グループ化されたTable Viewも、情報のリストを表示します。ただし、関連する行が視覚的に区別されたセクションにグループ化されています。図 1-4に示すように、各セクションは、丸みを帯びた角を持ち、青みがかったグレイの背景に表示されます。各セクションに、ヘッダまたはフッタとしてテキストや画像を持たせて、そのセクションの文脈や要約を提供することもできます。グループ化されたテーブルは、特に、データ階層内の最も詳細な情報を表示する場合に、威力を発揮します。これを利用して、詳細情報を概念的なグループにまとめて、文脈情報を提供すれば、ユーザがその情報をすばやく理解するのに役立ちます。たとえば、連絡先リスト内の連絡先情報は、電話番号、電子メールアドレス、住所、その他のセクションにグループ分けされています。

図 1-4 グループスタイルのTable View

グループ化されたTable View内のセクションのヘッダとフッタは、図 1-5に示すような相対位置とサイズになっています。

図 1-5 セクションのヘッダとフッタ

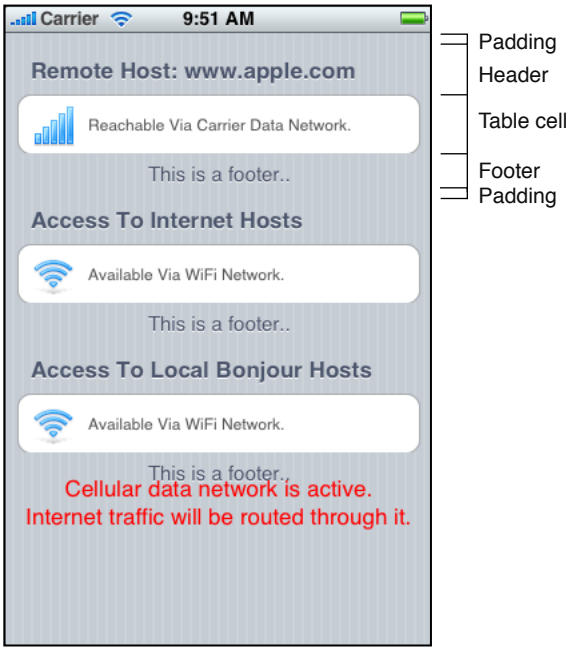


Table View Cellの標準スタイル

Table Viewの2つのスタイルのほかに、UIKitでは、Table Viewが行を表示するために使用するセルについても4つのスタイルが定義されています。必要であれば、外観が異なるカスタムTable View Cellを作成することもできますが、これらの4つの定義済みセルスタイルはほとんどの目的にかないます。定義済みのスタイルでTable View Cellを作成する手法と、カスタムセルを作成する手法については、「Table View Cellの詳細」（49 ページ）で説明します。

注： Table View Cellのスタイルは**iPhone OS 3.0**で導入されました。

Table Viewの行のデフォルトのスタイルは、1つのタイトルを持ち、画像を1つ置ける単純なセルスタイルです（図 1-6を参照）。このスタイルは`UITableViewCellStyleDefault`定数に対応します。

図 1-6 デフォルトのテーブル行スタイル（サブタイトルがない場合）

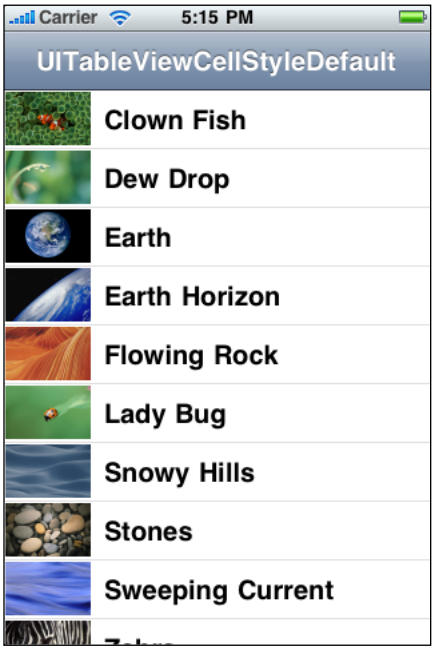


図 1-7に示す行のセルスタイルでは、メインタイトルを左揃えにし、その直下にグレイのサブタイトルを配置しています。また、デフォルト画像の位置に画像を1つ置けます。このスタイルは、「iPod」アプリケーションで使われており、`UITableViewCellStyleSubtitle`定数に対応します。

図 1-7 タイトルの下にサブタイトルを持つテーブル行スタイル

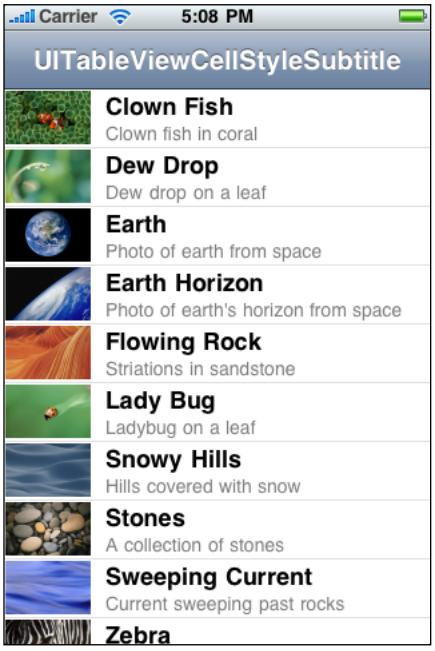


図 1-8に示す行のセルスタイルでは、メインタイトルを左揃えにし、青字のサブタイトルを行の右端に右揃えで配置しています。画像は置けません。このスタイルは、「設定(Settings)」アプリケーションで使われており、サブタイトルは環境設定に対する現在の設定値を表します。このスタイルは`UITableViewCellStyleValue1`定数に対応します。

図 1-8　右揃えのサブタイトルを持つテーブル行スタイル

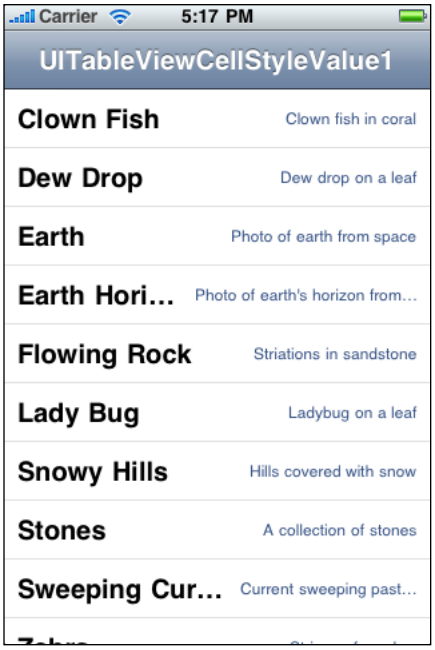


図 1-9に示す行のセルスタイルでは、青字のメインタイトルを行の左端からインデントされた位置に、右揃えで配置しています。サブタイトルは、この位置から少し間隔をあけて、左揃えで表示されています。このスタイルでは、画像は置けません。このスタイルは「電話(Phone)」アプリケーションの「連絡先(Contacts)」部分で使われており、`UITableViewCellStyleValue2`定数に対応します。

図 1-9 「連絡先(Contacts)」形式のテーブル行スタイル

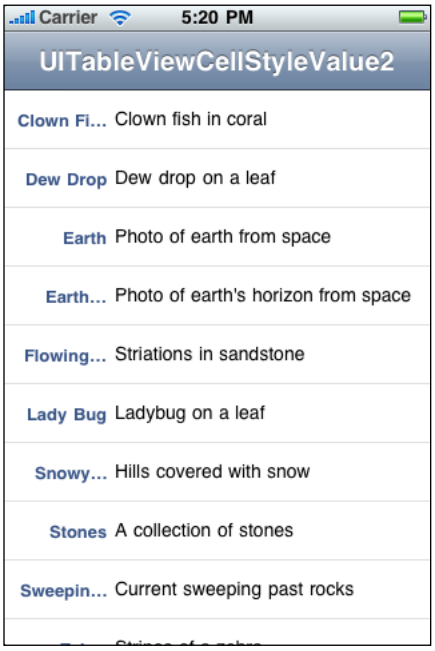


Table View APIの概要

Table View用のプログラムインターフェイスは、いくつかのクラス、2つの形式プロトコル、およびFoundationフレームワークのクラスに追加された1つのカテゴリで表現されます。Table Viewを実装する場合は、これらのすべてのAPIコンポーネントを扱わなければなりません。

Table View

Table View自体は、`UITableView`クラスのインスタンスです。このクラスには、Table Viewの外観を設定する（たとえば、行のデフォルトの高さを指定したり、テーブルのヘッダとして使用するビューを提供したりする）メソッドが定義されています。そのほかに、現在選択されている行にアクセスしたり、特定の行またはセルにアクセスしたりするためのメソッドもあります。`UITableView`のその他のメソッドを呼び出して、選択を管理したり、Table Viewをスクロールしたり、行やセクションを追加または削除したりもできます。

`UITableView`は、ウインドウサイズより大きなコンテンツを持つビューのためのスクロール動作を定義する`UIScrollView`を継承しています。`UITableView`では、垂直スクロールのみを許可するように、スクロール動作を再定義しています。

データソースとデリゲート

`UITableView`オブジェクトは、1つのデリゲートと1つのデータソースを持っていなければなりません。Model-View-Controllerデザインパターンに従って、データソースは、アプリケーションのデータモデル（Modelオブジェクト）とTable Viewの仲介役をします。一方、デリゲートは、Table Viewの外観と動作を管理します。データソースとデリゲートは、通常は同じオブジェクトです（ただし、必ずしも同じとは限りません）。また、そのオブジェクトが`UITableViewController`のカスタムサブクラスであることもよくあります（詳細については、「Table Viewを利用したデータ階層のナビゲーション」（23 ページ）を参照）。

データソースは、`UITableViewDataSource`プロトコルを採用し、デリゲートは、`UITableViewDelegate`プロトコルを採用します。`UITableViewDataSource`には、保有するセクション数（デフォルトは1）をTable Viewに伝えるオプションメソッドが1つあります。また、各セクションの行数をTable Viewに伝える必須メソッドも1つあります。`UITableViewDelegate`プロトコルには、デリゲートが実装しなければならないメソッドが1つ宣言されています。このメソッドは、特定の行を描画するときにTable Viewが使用するセルオブジェクトを返すメソッドです（このデリゲートのタスクの詳細については、「Table Viewのセル」（22 ページ）を参照してください）。また、このメソッドを利用すると、デリゲートはTable Viewの外観を変更できます。どちらのプロトコルにも、選択やアクセサリビューのタップに応答したり、セルの挿入、削除、および並び替えを管理したりするためのオプションメソッドが含まれています。

アプリケーションは、UILocalizedIndexedCollationという簡易クラスを利用して、インデックス付きリスト用のデータをデータソースで編成したり、インデックス内の項目をユーザがタップしたときに適切なセクションを表示したりできます。UILocalizedIndexedCollationクラスはセクションタイトルのローカライズにも使用します。UILocalizedIndexedCollationクラスはiPhone OS 3.0で導入されました。

NSIndexPathクラスへの拡張

UITableView、UITableViewDataSource、およびUITableViewDelegateのメソッドの多くは、パラメータや戻り値として、インデックスパスを表すオブジェクトを持ちます。インデックスパスは、ネストされた配列のツリー内の特定のノードへのパスを表し、Foundationフレームワークでは、NSIndexPathオブジェクトで表現されます。UIKitでは、NSIndexPathに対してカテゴリが1つ宣言されています（『*NSIndexPath UIKit Additions*』を参照）。このカテゴリには、主要なパスを返したり、セクション内の行を見つけたり、行とセクションのインデックスからNSIndexPathオブジェクトを生成したりするメソッドが含まれています。

Table Viewのセル

「データソースとデリゲート」（21 ページ）で説明したように、データソースは、Table Viewに表示される各行に対応するセルオブジェクトを返さなければなりません。これらのセルオブジェクトは、UITableViewCellクラスを継承する必要があります。このクラスには、セルの選択や編集を管理したり、アクセサリビューを管理したり、セルを設定するためのメソッドが含まれています。UITableViewCellクラスで定義されている標準スタイルのセルを直接インスタンス化して、これらのセルに、1つまたは2つの文字列と、スタイルによっては、画像とテキストの両方で構成されたコンテンツを入れることができます。または、標準スタイルのセルを使用する代わりに、“既成の”セルオブジェクトのコンテンツビューに独自のカスタムサブビューを配置することもできます。UITableViewCellのサブクラスを定義して、Table View Cellの外観や動作を徹底的にカスタマイズすることもできます。これらの方法についてはすべて、「Table View Cellの詳細」（49 ページ）で解説します。

Table Viewを利用したデータ階層のナビゲーション

Table Viewの一般的な用途（Table Viewに最も適した用途）は、階層的なデータのナビゲーションです。階層の最上位レベルにあるTable Viewには、最も概要的なレベルのデータカテゴリのリストが表示されます。ユーザは、ある行を選択して、階層の次のレベルに“掘り下げ”ます（ドリルダウンします）。階層の一番下には、特定の項目の詳細を表示するビュー（通常はTable View）があります（たとえば、住所録の1つのレコード）。ユーザが、この項目を編集できるようにすることもできます。このセクションでは、データモデル階層のレベルを一連のTable Viewにマップする方法と、このようなナビゲーションベースのアプリケーションの実装に役立つUIKitフレームワークの機能の使いかたを説明します。

階層的なデータモデルとTable View

ナビゲーションベースのアプリケーションでは、通常、アプリケーションデータをModelオブジェクト（アプリケーションのデータモデルと呼ばれる場合もある）のグラフとして設計します。アプリケーションのモデルレイヤは、Core Data、プロパティリスト、カスタムオブジェクトのアーカイブなど、さまざまなメカニズムやテクノロジーを使用して実装できます。どのようなアプローチをとるかに関係なく、アプリケーションのデータモデルを渡り歩くには、すべてのナビゲーションベースのアプリケーションに共通するパターンに従います。データモデルは階層構造になっていて深さがあり、この階層構造のさまざまなレベルにあるオブジェクトが、Table Viewの行を埋めるソースになります。

注： Core DataはiPhone OS 3.0でサポート対象テクノロジーとして導入されました。Core Dataのテクノロジーとフレームワークについては、『*Core Data Overview*』を参照してください。

Modelオブジェクトの階層としてのデータモデル

優れた設計のアプリケーションでは、Model-View-Controller (MVC)デザインパターンに従って、クラスとオブジェクトを分けています。アプリケーションのデータモデルは、このパターンのModelオブジェクトで構成されます。Modelオブジェクトは、（オブジェクトモデリングパターンで定義されている用語を使用すれば）プロパティによって記述できます。これらのプロパティは、属性と関係の大きく2種類に分けられます。

注： ここでの"プロパティ"という概念は、Objective-Cの宣言済みプロパティ機能と、理論的には関連していますが同一ではありません。通常、クラス定義は、インスタンス変数と宣言済みプロパティを通じた、プログラミング上のプロパティを表現します。宣言済みプロパティの詳細については、『*The Objective-C 2.0 Programming Language*』を参照してください。MVCとオブジェクトモデリングの詳細については、『*Cocoa Fundamentals Guide*』の「Cocoa Design Patterns」をお読みください。

属性は、Modelオブジェクトのデータの要素を表します。属性には、プリミティブクラスのインスタンス（`NSString`、`NSDate`、`UIColor`などのオブジェクト）から、C言語の構造体や単純なスカラ値まであります。属性は一般に、データ階層の"リーフノード"を表す、その項目の詳細ビューを表示するTable Viewを埋めるために使用されます。

また、Modelオブジェクトがほかの Modelオブジェクトと関係を持っている場合もあります。データモデルは、そうした関係を通じてオブジェクトグラフを構成することで階層が深くなります。関係は、カーディナリティ（濃度）の点からみると、1対1と1対多の大きく2種類に分類されます。1対1関係は、1つのオブジェクトともう1つのオブジェクトとの関係を表します（たとえば、親との関係）。一方、1対多の関係は、1つのオブジェクトと、同じ種類の複数のオブジェクトとの関係を表します。1対多の関係は、包含関係によって特徴付けられ、プログラムでは、`NSArray`オブジェクト（または、単なる配列）などのコレクションで表現されます。1つの配列にほかの配列が複数含まれる場合や、複数の辞書（内部に保持している値をキーによって識別するコレクション）が含まれる場合もあります。同様に、辞書の中にほかのコレクション（配列、集合、その他の辞書など）が1つ以上含まれる場合もあります。このようにほかのコレクションをネストしたコレクションによって、データモデルは階層が深くなります。

Table Viewとデータモデル

標準（プレーン）スタイルのTable Viewの行は、通常、アプリケーションのデータモデルのコレクションオブジェクトに基づいており、これらのオブジェクトは一般に配列です。この配列には、文字列その他の要素が含まれます。Table Viewは、行の内容を表示する際にこの要素を使用します。Table Viewを作成すると（「Table Viewの作成と設定」（33 ページ）を参照）、Table Viewはまず、データソースにディメンション（セクション数とセクションごとの行数）を問い合わせます。次に、各行のコンテンツを要求します。このコンテンツは、データモデル階層の適切なレベルの配列から取得されます。

Table Viewのデータソースとデリゲートに定義されているメソッドの多くにおいて、Table Viewは、現在の操作（たとえば、行のコンテンツを取得したり、ユーザがタップした行を示すなど）の対象となるセクションと行を識別するインデックスパスを渡します。インデックスパスはFoundationフレームワークの`NSIndexPath`クラスのインスタンスであり、ネストした配列のツリー内の項目を識別するために使用できます。UIKitフレームワークでは、`NSIndexPath`を拡張して、`section`と`row`というプロパティをこのクラスに追加しています。データソースはこれらのプロパティを使用して、Table Viewのセクションと行を、そのTable Viewのデータのソースとして使われている配列の、対応するインデックス位置にある値にマップしなければなりません。

注： UIKitフレームワークでの`NSIndexPath`クラスの拡張については、『*NSIndexPath UIKit Additions*』を参照してください。

図 3-1に示すTable Viewシーケンスを考えます。この例のデータ階層の最上位レベルは4つの配列からなる配列です。内側の各配列には特定の地域の道路を表すオブジェクトが含まれています。ユーザがこれらの地域の1つを選択すると、次のTable Viewには、選択された配列内の道路を表す名前の一覧が表示されます。ユーザが特定の道路を選択すると、次のTable Viewにはその道路についての詳細がグループスタイルのTable Viewで表示されます。

図 3-1 データモデルのレベルのTable Viewへのマッピング

説明のために上の図では、データ階層の3つのレベルを移動している3つのTable Viewのシーケンスを示しています。このアプリケーションは簡単に設計を変更して、使用するTable Viewを2つだけにすることができます。第1のTable Viewを（プレーンスタイルの）リストにして、各地域をそのTable Viewのセクションにし、各セクションの行をその地域の道路名にします。このデータモデルはこの配置を、複数の配列がネストした配列として表現できます。Table Viewが特定の行のコンテンツをデータソースに要求するとき、データソースに`NSIndexPath`オブジェクトを渡します。データソースはこのオブジェクトを使用して、まず内側の配列（`section`プロパティ）を見つけて、次にその配列内のオブジェクト（`row`プロパティ）を見つけます。

View Controllerとナビゲーションベースアプリケーション

UIKitフレームワークには、iPhone OSでよく使われるユーザインターフェイスパターンを管理するためのView Controllerクラスがいくつかあります。View Controllerは、`UIViewController`クラスを継承したコントローラオブジェクトです。これらはビュー管理に不可欠なツールです。特にアプリケーションがこれらのビューを使用して、データ階層の連続するレベルを表示する場合は非常に重要です。Table Viewを管理するために、UIKitでは`UIViewController`のサブクラスである`UITableViewController`クラス（「Table View Controller」（28 ページ）で説明）を提供しています。このセクションでは、View Controllerとは何かについて説明し、Table Viewを含む連続するビューを表示したり管理したりする方法について説明します。

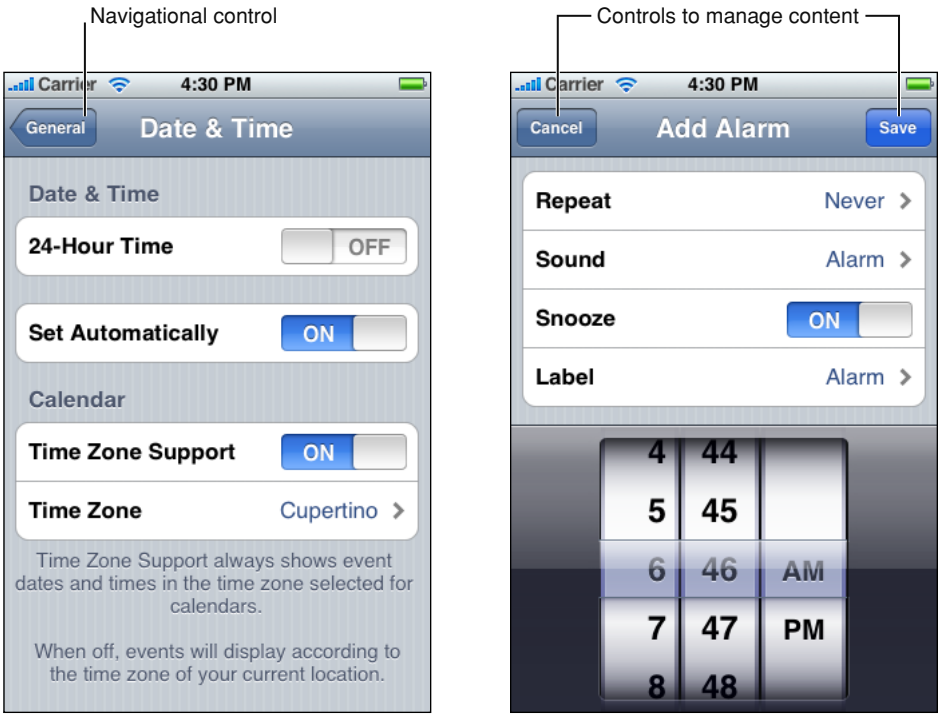
View ControllerとNavigation Controllerの概要

Navigation Barは、ユーザがデータ階層をたどることができるようにするためのユーザインターフェイスの仕組みです。ユーザは概要的な最上位レベルの項目からスタートして、リーフノードの項目の具体的なプロパティを表示する詳細ビューまで階層をドリルダウンします（掘り下げます）。Navigation Barの下のビューには、現在のレベルのデータが表示されます。Navigation Barには、現在のビューのタイトルが表示されます。また、そのビューが最上位レベルよりも下の階層のビューの場合は、Navibation Barの左端に戻るボタンが表示されます。戻るボタンは、ユーザがタップすると前のレベルに戻ることができるナビゲーションコントロールです（デフォルトでは、戻るボタンには、前のビューのタイトルが表示されます）。Navigation Barに「編集(Edit)」ボタン（現在のビューで編集モードに入るために使用する）や、コンテンツを管理する機能に対応したカスタムボタンを持たせることもできます（図 3-2 （27 ページ）を参照）。

注： このセクションでは、この文書全体を通して解説するコーディング作業についての予備知識を提供するために、View ControllerとNavigation Controllerの概要を説明します。View ControllerとNavigation Controllerについて詳しく学習するには、『*View Controller Programming Guide for iPhone OS*』を参照してください。

Navigation Barは`UINavigationBar`オブジェクトです。このオブジェクトが管理するビューごとのコンポーネント（バーのタイトル、戻るボタン、「編集(Edit)」ボタン、カスタムボタンなど）は、`UINavigationItem`クラスのインスタンスです。これらのコンポーネントがボタンの場合、その値として`UIBarButtonItem`オブジェクトを取ります。Navigation Controller（`UINavigationController`のインスタンス）を使用して、Navigation Itemや表示するビューを管理する場合は、`UINavigationBar`や`UINavigationItem`を直接扱う必要はありません。ただし、UINavigationItemの場合は、ナビゲーション要素の値を設定しなければなりません。

図 3-2 Navigation Barとよく使われるコントロール要素

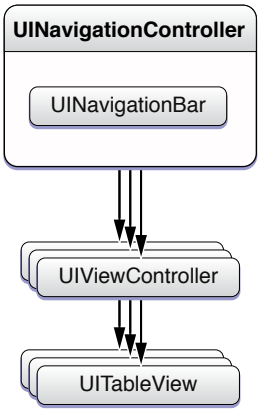


`UINavigationController`クラスは、iPhone OSでビューを管理するControllerオブジェクトに共通のプログラムインターフェイスと動作を定義した基底クラス、`UIViewController`を継承しています。この基底クラスからの継承によって、View Controllerは一般的なビュー管理用のインターフェイスを取得します。このインターフェイス部分を実装すると、View Controllerは、ビューの自動回転、メモリ不足通知への応答、"モーダル"ビューのオーバーレイ、「編集(Edit)」ボタンのタップへの応答、その他のビュー管理を実行できます。

`UINavigationController`は、Navigation Bar（バーの下のビューに対応してバーに表示される要素も含む）を管理します。`UIViewController`オブジェクトは、Navigation Barの下に表示されるビューを管理します。このView Controllerに対しては、`UIViewController`のサブクラス、または特定のタイプのビューを管理するためにUIKitフレームワークが提供しているView Controllerクラスのサブクラスを作成します。Table Viewの場合は、このView Controllerクラスは`UITableViewController`です。データ階層内のレベルを反映した一連のTable Viewを表示するNavigation Controllerの場合は、Table Viewごとに別々のカスタムTable View Controllerを作成する必要があります。

Navigation Controllerは、（表示されるTable Viewごとに1つある）View Controllerのスタックを管理することによって、一連のTable Viewをたどるナビゲーションを操作します（図 3-3を参照）。この動作は、Root View Controllerから始まります。ユーザがTable Viewの行（通常は、ディスクロージャインジケータまたは詳細ディスクロージャボタン）をタップすると、Root View Controlerは、次のView Controllerをスタックにプッシュします。その結果、この新しいView ControllerのTable Viewが右からスライドしてきて画面に表示され、Navigation Barの要素が適切に更新されます。ユーザがNavigation Barの戻るボタンをタップすると、現在のView Controllerがスタックからポップされます。その結果、Navigation Controllerは、現在スタックの一番上にあるView Controllerによって管理されるTable Viewを表示します。

図 3-3 ナビゲーションベースアプリケーションのNavigation ControllerとView Controller



`UIViewController`には、View Controllerが、現在表示されているTable Viewに対応してNavigation Barに表示されるナビゲーション要素（`UINavigationItem`オブジェクト）にアクセスしたり、値を設定するためのメソッドが含まれています。また、現在のTable Viewに対応するNavigation Barのタイトルを設定するための`title`プロパティも宣言されています。

必ずしもすべてのアプリケーションが、Navigation ControllerとView Controllerのアーキテクチャを使用する必要はありません。アプリケーションが、一連のTable Viewではなく、単独のTable Viewを表示する場合は、「編集(Edit)」ボタンやその他の特殊なボタンをNabigation Barに表示する必要がなければ、Navigation Controllerを使用する必要はありません。また、アプリケーションがTable Viewを1つしか表示しない場合は、それを管理するためのカスタムView Controllerも必要ありません。たとえば、1つのTable Viewを使用して、選択可能な選択肢のリストを表示するアプリケーションの場合、そのTable View用のデータソースとデリゲートの役割を果たせば、View Controller以外のオブジェクトを使用することもできます。

Table View Controller

`UIViewController`の直接のサブクラスを使用してNavigation Controllerアーキテクチャ内のTable Viewを管理することもできますが、`UITableViewController`をサブクラス化したほうが手間を省くことができます。`UITableViewController`クラスは、`UIViewController`の直接のサブクラスを作成してTable Viewを管理する場合に実装しなければならないさまざまな処理を行ってくれます。

Table View Controllerを作成するには、`initWithStyle:`メソッドを利用してそのためのメモリの割り当てと初期化を行います。その際、必要なTable Viewタイプに応じて`UITableViewStylePlain`または`UITableViewStyleGrouped`のいずれかを渡します。Table View Controllerを作成すると、Table View ControllerはTable Viewを作成するか、またはnibファイルからロードします。どちらのケースになるかによって、動作が少し異なります。

- nibファイルを指定した場合、`UITableViewController`オブジェクトはそのnibファイルにアーカイブされている`UITableView`オブジェクトをロードします（通常、nibファイルは`UITableViewController`の属性として指定されます。UITableViewControllerはこのnibファイルのFile's Ownerでなければなりません）。Table Viewの属性、サイズ、および自動サイズ変更特性は通常nibファイルで設定します。Table Viewのデータソースとデリゲートは、nibファイルで定義されているオブジェクトになります（定義されている場合）。

- nibファイルが存在しない場合は、`UITableViewController`オブジェクトが、未設定の`UITableView`オブジェクトを割り当てて、適切なディメンションと自動サイズ変更マスクで初期化します。UITableViewControllerは、自身をTable Viewのデータソースおよびデリゲートとして設定します。nibファイルにデータソースまたはデリゲートが定義されていない場合も同様です。

Table Viewが初めて表示されるときに、Table View ControllerはそのTable Viewに`reloadData`を送信し、データソースからデータを要求するように促します。データソースは要求するセクション数とセクションごとの行数をTable Viewに伝えてから、各行に表示するデータをTable Viewに提供します。このプロセスについては、「Table Viewの作成と設定」（33 ページ）で説明します。

`UITableViewController`クラスは、その他の一般的なタスクも実行します。Table Viewが表示される直前に選択範囲をクリアしたり、テーブルの表示が完了したときにスクロールインジケータを点滅させたりする処理も行います。さらに、ユーザが「編集(Edit)」ボタンをタップしたときには、それに応答してTable Viewを編集モードに変更します（ユーザが「完了(Done)」をタップした場合は、編集モードを解除します）。このクラスは、管理下にあるTable Viewにアクセスするための`tableView`プロパティを公開しています。

注： `UITableViewController`は、iPhone OS 3.0では新機能が加わりました。Table View ControllerはTable Viewの行のインライン編集をサポートします。たとえば、行に埋め込まれているテキストフィールドが編集モードのとき、表示されている仮想キーボードの上にある編集中の行をスクロールします。さらに、Core Dataのフェッチ要求によって返される結果を管理するための`NSFetchedResultsController`クラスをサポートしています。

`UITableViewController`クラスは、`loadView`、`viewWillAppear:`、および`UIViewController`から継承したその他のメソッドをオーバーライドすることによって、これらすべての処理を実装しています。`UITableViewController`のサブクラスで、これらのメソッドをオーバーライドして、特殊な動作を実現することもできます。これらのメソッドをオーバライドする場合は、デフォルトの動作を実行するために、そのメソッドのスーパークラスの実装を（通常は、最初のメソッド呼び出しとして）必ず呼び出してください。

注： 管理対象のビューが複数のサブビューで構成されており、その中の1つがTable Viewの場合は、Table Viewを管理するためには`UITableViewController`のサブクラスではなく、`UIViewController`のサブクラスを使用すべきです。`UITableViewController`クラスのデフォルトの動作では、Navigation BarとTab Barの間の（これら両方が存在する場合）画面一杯にTable Viewを表示します。

`UITableViewController`のサブクラスではなく`UIViewController`のサブクラスを使用してTable Viewを管理することにした場合は、ヒューマンインターフェイスガイドラインに適合するように、前述のいくつかのタスクを実行する必要があります。Table Viewを表示する前にTable View内の選択をクリアするために、`deselectRowAtIndexPath:animated:`を呼び出して選択中の行（存在する場合）をクリアする`viewWillAppear:`メソッドを実装します。Table Viewの表示が完了したら、Table View に`flashScrollIndicators`メッセージを送信することでScroll Viewのスクロールインジケータを点滅させなければなりません。それには、`UIViewController`の`viewDidAppear:`メソッドをオーバーライドします。

ナビゲーションベースアプリケーションでのTable Viewの管理

`UITableViewController`オブジェクト（または、Table Viewのデータソースとデリゲートの役割を果たすその他のオブジェクト）は、行をデータで埋めたり、オブジェクトの設定を行ったり、選択操作に応答したり、編集セッションを管理したりするために、Table Viewから送信されたメッセージ

に応答する必要があります。この文書の以降の各章では、これらの処理を行う方法について説明します。ただし、ナビゲーションベースアプリケーションで一連のTable Viewが適切に表示されることを保証するには、それ以外にいくつか実行しなければならないことがあります。

注： このセクションでは、Table Viewに重点を置きながら、View ControllerとNavigation Controllerに関するタスクについて簡単に説明します。View controllerとNavigation Controllerの完全な解説（実装の詳細も含む）については、『*View Controller Programming Guide for iPhone OS*』を参照してください。

通常、このシーケンスは、`applicationDidFinishLaunching:`（または`application:didFinishLaunchingWithOptions:`）メソッドの実装内のアプリケーションデリゲートから始まります。このデリゲートは、Root View Controllerになる`UITableViewController`サブクラスのインスタンスを作成します。次に、`UINavigationController`のインスタンスを割り当て、`initWithRootViewController:`メソッドを利用して、作成したばかりのTable View Controllerでそれを初期化します。このイニシャライザには、Root View Controllerを、Navigation Controllerが管理するView Controllerスタックの先頭のオブジェクトにするという効果があります。Navigation Controllerを作成した後で、デリゲートはNavigation Controllerのビューをウインドウに追加し、そのウインドウを表示します。リスト 3-1にこの一連の呼び出しを示します。

リスト 3-1　Root View Controllerのセットアップ（nibファイルのウインドウの場合）

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

    RootViewController *rootViewController = [[RootViewController alloc]
initWithStyle:UITableViewStylePlain];
    NSArray *timeZones = [NSTimeZone knownTimeZoneNames];
    rootViewController.timeZoneNames = [timeZones
sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
    UINavigationController *aNavigationController = [[UINavigationController
alloc] initWithRootViewController:rootViewController];
    self.navigationController = aNavigationController;
    [aNavigationController release];
    [rootViewController release];

    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

この例のウインドウは、nibファイルから展開されて、アプリケーションデリゲートのアウトレットに割り当てられます。このウインドウをプログラムで作成したとすると、コードはリスト 3-2のようになります（この例では、`window`はアプリケーションデリゲートのプロパティです）。

リスト 3-2　Root View Controllerのセットアップ（プログラムでウインドウを作成する場合）

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

    // ウインドウを作成する
    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    // Navigation ControllerとView Controllerを作成する
    RootViewController *rootViewController = [[RootViewController alloc]
initWithStyle:UITableViewStylePlain];
    navigationController = [[UINavigationController alloc]
initWithRootViewController:rootViewController];
    [rootViewController release];
```

```
    // ウインドウを設定して表示する
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

initWithStyle:を利用してTable View Controllerを初期化する場合、この初期化メソッドのほかに、スーパークラスから継承した以下のイニシャライザも一緒に呼び出されます。

- initWithNibName:bundle:—UIViewControllerから継承

  Table Viewがnibファイルで定義されている場合は、Interface BuilderでTable View ControllerのNib Name属性を（File's Ownerに）設定する代わりに、このメソッドをオーバーライドして、superを呼び出してスーパークラスにnibファイル名を提供するようにできます。

- init—NSObjectから継承

これらのイニシャライザをオーバーライドして、View Controllerに共通のセットアップタスクを実行できます（たとえば、リスト 3-3 （31 ページ）に示すようなNavigation Barのタイトルの設定）。

リスト 3-3 initでのNavigation Barタイトルの設定

```
- init {
    if (self = [super init]) {
        self.title = NSLocalizedString(@"List", @"List title");
    }
    return self;
}
```

同じセットアップタスクを、初期化の直後に呼び出されるloadViewメソッドやviewDidLoadメソッドで実行することもできます。また、これらのメソッド内で、スーパークラスから継承したnavigationItemプロパティを利用して、さまざまなUINavigationItemプロパティを設定することもできます。一般に、これらのプロパティはボタン要素です。リスト 3-4に、Table View Controllerクラスで Navigation Barに「編集(Edit)」、「完了(Done)」、および追加（「＋」）の各ボタンを配置する様子を示します。

リスト 3-4 loadViewでのNavigation Barのボタンの設定

```
- (void)loadView {
    [super viewDidLoad];
    self.tableView.allowsSelectionDuringEditing = YES;
    self.navigationItem.leftBarButtonItem = self.editButtonItem;
    self.addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(addItem:)];
    self.navigationItem.rightBarButtonItem = self.addButton;
}
```

ここで、ルートTable View Controllerが管理するTable Viewをユーザに表示する場合を考えます。これは、リスト（標準スタイルの）Table Viewです。アプリケーションは、シーケンス内の次のTable Viewをどのようにして表示するのでしょうか？

ユーザがTable Viewの行をタップすると、そのTable Viewはデリゲートで実装されている tableView:didSelectRowAtIndexPath:メソッドまたは tableView:accessoryButtonTappedForRowWithIndexPath:メソッドを呼び出します（後者のメソッドは、ユーザが詳細ディスクロージャインジケータをタップした場合に呼び出されます）。リ

スト 3-5に示す例のデリゲートは、シーケンス内の次のTable Viewを管理するTable View Controllerを作成し、Table Viewを構成するために必要なデータを設定します。そして、この新しいView ControllerをNavigation ControllerのView Controllerスタックにプッシュします。

リスト 3-5　次のTable View Controllerを作成してスタックにプッシュする

```
- (void)tableView:(UITableView *)tableView
accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath {
    BATDetailViewController *trailDetailController = [[BATDetailViewController
 alloc] initWithStyle:UITableViewStyleGrouped];
    trailDetailController.curTrail = [trails objectAtIndex:indexPath.row];
    [[self navigationController] pushViewController:trailDetailController
animated:YES];
    [trailDetailController release];
}
```

最後のこのコード例は、ナビゲーションベースアプリケーションが一連のTable Viewを表示する方法の概略を示すためのものです。Table Viewでの選択の処理の詳細については、「選択の管理」（69 ページ）を参照してください。

ナビゲーションベースアプリケーション用のデザインパターン

Table Viewを利用したナビゲーションベースアプリケーションを優れた設計にするには、次のデザインパターンに従うべきです。

- View Controllerオブジェクト（通常は`UITableViewController`オブジェクト）は、データソースとしての役割を果たし、データ階層の1つのレベルを表すオブジェクトのデータによってTable Viewを埋めます。

  Table Viewが項目リストを表示する場合、このオブジェクトは通常は1つの配列です。Table Viewが項目の詳細（つまり、データ階層のリーフノード）を表示する場合、このオブジェクトは、カスタムModelオブジェクト、Core Dataの管理下のオブジェクト、辞書、またはそれに類するオブジェクトになります。

- View ControllerはTable Viewを埋めるために必要なデータを格納します。

  View Controllerは、Table Viewを埋めるために直接このデータを使用できます。または、このデータを使用して、必要なデータを取得できます。View Controllerサブクラスを設計するときには、このデータを保持するインスタンス変数（宣言済みプロパティまたはアクセサメソッドを介してアクセスできる）を定義しなければなりません。

  View Controllerは、グローバル変数やシングルトンオブジェクト（アプリケーションデリゲートなど）を介して、Table View用のデータを取得するべきでは*ありません*。このような直接的な依存関係は、コードの再利用性を下げるだけでなく、テストやデバッグを困難にします。

- Navigation Controllerスタックの最上位にある現在のView Controllerは、シーケンス内の次のView Controllerを作成し、それをスタックにプッシュする前に、データソースとしての役割を果たすこのView Controllerが、Table Viewを埋めるために必要なデータを設定します。

Table Viewの作成と設定

行のタップやその他のアクションに応答してTable Viewを操作する前に、Table Viewをユーザに提示しなければなりません。この章では、Table Viewの作成と設定、およびデータの取り込みに必要な作業について説明します。

この章で紹介するコード例のほとんどは、サンプルプロジェクト*TableViewSuite*と*TheElements*から引用しています。

Table View作成の基礎

Table Viewの作成には、アプリケーション内のいくつかのエンティティ（クライアント、Table Viewそのもの、そのTable Viewのデータソースとデリゲート）とのやり取りが必要になります。通常は、クライアント、デリゲート、およびデータソースは同じオブジェクトですが、別々のオブジェクトであってもかまいません。図 4-1 （34 ページ）に図示するように、クライアントが呼び出しシーケンスを開始します。

1. クライアントは、特定のフレームとスタイルの`UITableView`インスタンスを作成します。これは、プログラムまたはInterface Builderのいずれかで行えます。通常、フレームは画面フレームからステータスバーの高さを差し引いた大きさに設定されます。または、ナビゲーションベースアプリケーションの場合は画面フレームからステータスバーとナビゲーションバーの高さを差し引いた大きさに設定されます。この時、クライアントは、Table Viewのグローバルプロパティ（自動サイズ変更動作、行のグローバルな高さなど）も設定します。

   Interface Builderを使用してTable Viewを作成する方法については、「Table Viewアプリケーションの簡単な作成方法」 （35 ページ）を、プログラムによってTable Viewを作成する方法については、「プログラムによるTable Viewの作成」 （40 ページ）をそれぞれ参照してください。

2. クライアントは、Table Viewのデータソースとデリゲートを設定して、`reloadData`を送信します。データソースは、`UITableViewDataSource`プロトコルを採用する必要があります。また、デリゲートは、`UITableViewDelegate`プロトコルを採用しなければなりません。

3. データソースは、`UITableView`オブジェクトから`numberOfSectionsInTableView:`メッセージを受信すると、Table Viewのセクション数を返します。これは、実装が任意のプロトコルメソッドですが、Table Viewが複数のセクションを持つ場合、データソースはこれを実装しなければなりません。

4. 各セクションに関して、データソースは`tableView:numberOfRowsInSection:`メッセージを受信すると、そのセクションの行数を返します。

5. データソースは、Table Viewに表示する各行に関して、`tableView:cellForRowAtIndexPath:`を受け取ります。データソースは、それに対し、各行の`UITableViewCell`オブジェクトを設定して返します。`UITableView`オブジェクトは、このセルを使用して、行を描画します。

図 4-1 Table Viewを作成して設定するための呼び出しシーケンス

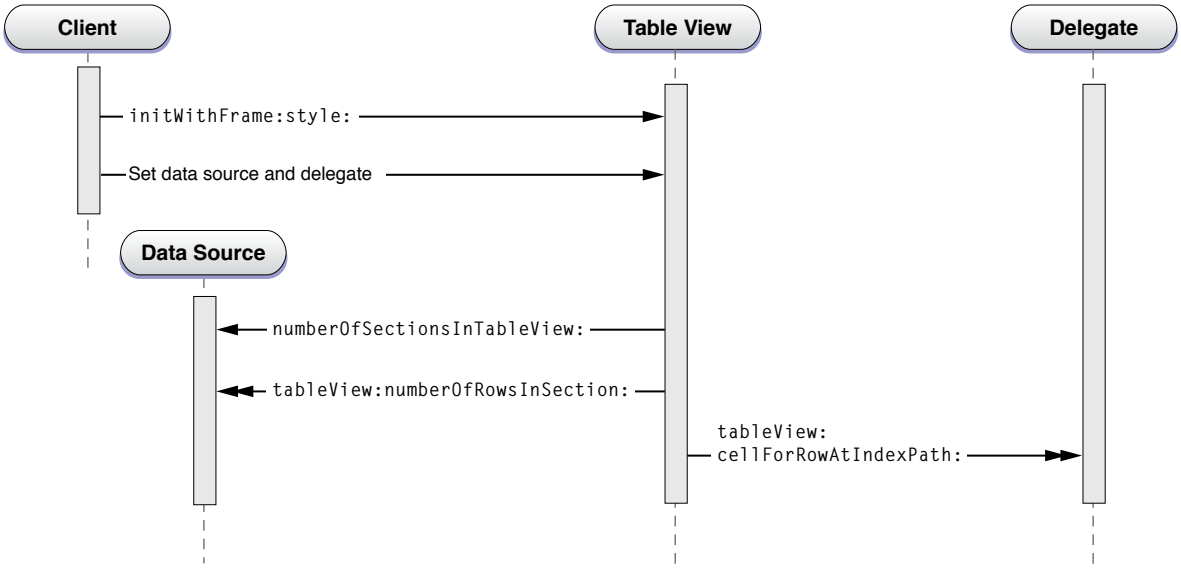


図 4-1の図には、`numberOfSectionsInTableView:`メソッドだけでなく、必須のプロトコルメソッドも示されています（セクション数を返す場合、0は有効な値です）。手順3から5では、Table Viewにデータを取り込んでいます。これらの手順で説明したメソッドの実装方法については、「Table Viewへのデータの取り込み」（41 ページ）で説明します。

データソースとデリゲートには、Table Viewを詳細に設定するために、プロトコルの実装が任意のその他のメソッドを実装する場合もあります。たとえば、データソースに`tableView:titleForHeaderInSection:`を実装することによって、Table Viewの各セクションのタイトルを設定できます。「Table Viewのオプション設定」（46 ページ）では、このようなTable Viewのカスタマイズオプションをいくつか紹介します。

Table Viewは、プレーンスタイル(`UITableViewStylePlain`)またはグループスタイル(`UITableViewStyleGrouped`)のいずれかのスタイルで作成します（Table Viewを初期化するときにスタイルを指定します。それには、`initWithFrame:style:`メソッドを直接的または間接的に呼び出します。どちらのスタイルの場合もTable Viewを作成する手順は同じですが、さまざまな種類の設定を行いたい場合があります。たとえば、グループスタイルのTable Viewは一般に項目の詳細を表示するため、デリゲートの`tableView:cellForRowAtIndexPath:`メソッド内で、カスタムアクセサリビュー（スイッチ、スライダなど）やカスタムコンテンツ（テキストフィールドなど）を追加したい場合もあります。この例は、「Table View Cellの詳細」（49 ページ）で紹介します。

Table Viewを作成したり設定したりする際の推奨事項

Table Viewアプリケーションを構成するには、さまざまな方法があります。たとえば、`NSObject`のカスタムサブクラスのインスタンスを使用して、Table Viewを作成、設定、および管理することができます。ただし、こうした目的のためにUIKitが提供しているクラス、手法、およびデザインパターンを採用した方が作業ははるかに簡単です。次のようなアプローチが推奨されます。

- Table Viewを作成したり管理したりするには、`UITableViewController`のサブクラスのインスタンスを使用します。

ほとんどのアプリケーションでは、Table Viewを管理するためにUITableViewControllerカスタムオブジェクトを使用します。「Table Viewを利用したデータ階層のナビゲーション」（23ページ）で説明したように、UITableViewControllerは自動的にTable Viewを作成し、自身をデリゲートとデータソースの両方として割り当て（対応するプロトコルを採用し）、Table Viewにデータを取り込むための手順を開始します。このオブジェクトは、その他の詳細な「ハウスキーピング（管理処理）」動作も担当します。Navigation Controllerアーキテクチャ内でのUITableViewController（UIViewControllerのサブクラス）の動作については、「Table View Controller」（28 ページ）で説明します。

- アプリケーションが主にTable Viewをベースとしている場合（"リスト"アプリケーション）は、Xcodeで定義されているNavigation-based Applicationテンプレートをプロジェクト作成時に選択します。

  「Table Viewアプリケーションの簡単な作成方法」（35 ページ）で説明するように、このテンプレートには、アプリケーションデリゲート、Navigation Controller、およびRoot View Controller（UITableViewControllerのカスタムサブクラスのインスタンス）を定義するnibファイルとスタブコードが含まれています。

- 連続するTable Viewの場合には、カスタムUITableViewControllerオブジェクトを実装する必要があります。対応するTable Viewは、プログラムで作成したり、個別のnibファイルからロードすることもできます。

  どちらの選択も可能ですが、純粋にプログラムで実行する方が簡単かもしれません。これらの選択肢については、「アプリケーションへのTable Viewの追加」（38 ページ）で説明します。

管理対象のビューが合成ビューで、その中の複数のサブビューの1つがTable Viewの場合は、そのTable View（およびその他のビュー）を管理するために、UIViewControllerのカスタムサブクラスを使用する必要があります。このコントローラクラスは、Table Viewをナビゲーションバーとタブバー（どちらかが存在する場合）の間の画面一杯に表示するため、UITableViewControllerオブジェクトを使用してはいけません。

Table Viewアプリケーションの簡単な作成方法

XcodeでTable Viewアプリケーションを作成するのは非常に簡単です。プロジェクトの作成時に、スタブコードと、Table Viewのセットアップと管理の構造を提供するnibファイルを含むテンプレートを選択します。Table Viewに基づいて構成されたアプリケーションを作成するには、次の手順を実行します。

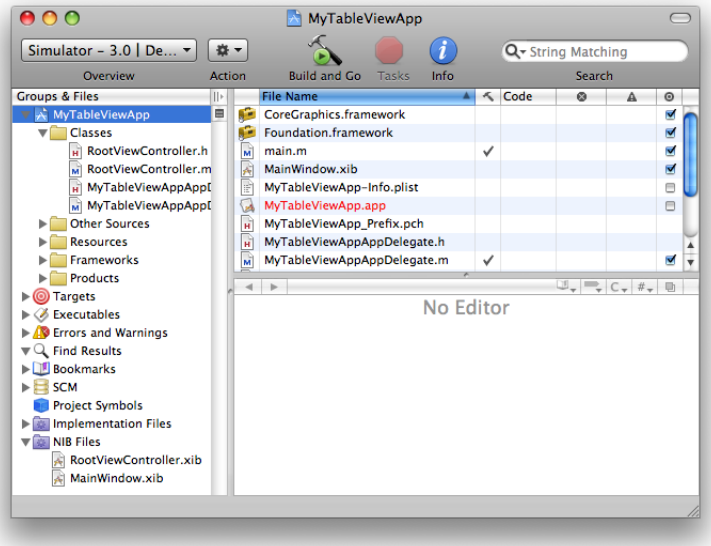
1. Xcodeで、「ファイル(File)」メニューから「新規プロジェクト(New Project)」を選びます。
2. Navigation-Based Applicationのテンプレートプロジェクトを選択し、「選択(Choose)」をクリックします。
3. プロジェクトの名前と場所を指定し、「保存(Save)」をクリックします。

Xcodeは、nibファイルにTable Viewを作成し、UITableViewControllerのカスタムサブクラスのインスタンスを作成します。このインスタンスは実行時に、nibファイルからTable Viewをロードしてデータを格納し、それを管理します。

プロジェクトの考察：組み立ての方法

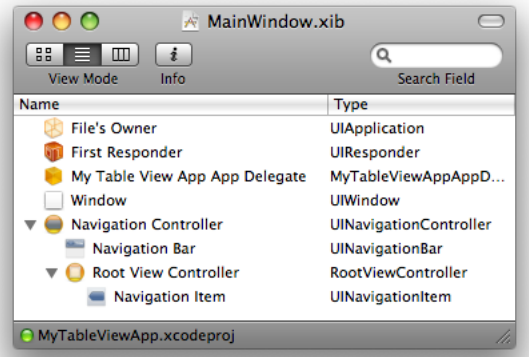
コーディングやnibの作業を開始する前に、作成したばかりのプロジェクトの主要なコンポーネント（アプリケーションデリゲート、Root View Controllerクラス、およびnibファイルの`MainWindow.xib`と`RootViewController.xib`）について考察しておくと役立ちます。図4-2は、これらの項目がXcodeのプロジェクトウインドウの「グループとファイル(Groups & Files)」ビューのどこにあるかを示しています。

図 4-2　作成したばかりのTable Viewアプリケーションプロジェクト



`MainWindow.xib`をダブルクリックして、Interface Builderでこのnibファイルを開きます。このnibファイルのドキュメントウインドウに、ファイルに含まれるオブジェクトが表示されます（図 4-3を参照）。最上位レベルの主なオブジェクトは、File's Owner（アプリケーションオブジェクト自身のプロキシ）、アプリケーションデリゲート、アプリケーションのウインドウ、および`UINavigationController`オブジェクトです。最後のオブジェクトは、Table Viewにとって重要なオブジェクトグラフのルートです。Navigatoin Controllerの直接の子は、画面を横切ってTable Viewの上に表示されるナビゲーションバーと、プロジェクトの`RootViewController`クラスのインスタンスのプレースホルダです。これはNavigation Controllerの子であるため、このView ControllerはRoot View Controller（このNavigation Controllerが管理するView Contorllerスタックの先頭のView Controller）になります（「Table Viewを利用したデータ階層のナビゲート」（23 ページ）を参照）。

図 4-3 メインウインドウのnibファイルの内容



アプリケーションが起動されると、`MainWindow.xib`nibファイルがメモリにロードされます。アプリケーションデリゲートは、リスト 4-1に示す2行のコードで、最初のユーザインターフェイスを表示します。Navigation Controllerにビューを要求することによって、アプリケーションデリゲートは、ナビゲーションバー、ナビゲーションバーのタイトル（ナビゲーション項目）、およびRoot View Controllerに対応するTable Viewを取得します。

リスト 4-1 最初のユーザインターフェイスを表示するアプリケーションデリゲート

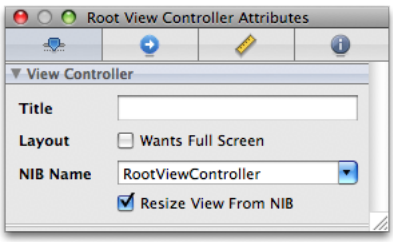
```
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

ただし（すでにお気付きかもしれませんが）、図 4-3 （37 ページ）のnibドキュメントウインドウのRoot View Controllerには、下位のTable Viewは表示されていません。このTable Viewはどこから来るのでしょうか？

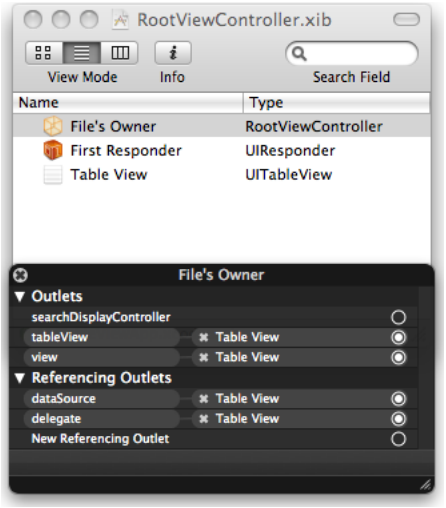
ドキュメントウインドウでRoot View Controllerを選択し、そのオブジェクトの属性インスペクタを表示します（Commandキーを押しながら1を押す）。ここで、「NIB Name」フィールドがRootViewControllerに設定されているのが分かります（図 4-4を参照）。デリゲートがNavigation Controllerにビューを要求すると、Navigation ControllerはそのRoot View Controllerにビューを要求します。そしてRoot View Controllerは、この属性で指定されたnibファイルをロードします。

図 4-4 Table View Controllerのnib-nameプロパティの設定



RootViewController.xib nibファイルにはTable Viewが含まれており、RootViewControllerオブジェクトがFile's Ownerとして設定されています。RootViewControllerとそのTable Viewとの接続を表示するには、File's Ownerを右クリック（またはControlを押しながらクリック）します。この操作によって、図 4-5に示すようなヘッドアップディスプレイが表示されます。

図 4-5 Root View Controllerのnibファイル内の接続



RootViewControllerオブジェクトは、Table Viewへの参照を2つ保持しています（1つは、UIViewControllerから継承したviewプロパティを介します）。また、このオブジェクトは、Table Viewのデータソースとデリゲートとして設定されています。

このTable Viewの特性は、nibドキュメントウインドウでこのTable Viewを選択し、Interface Builderインスペクタの「Attributes」ペインに移動して変更することもできます。たとえば、Table Viewのスタイルをプレーン（デフォルト）からグループに変更できます。

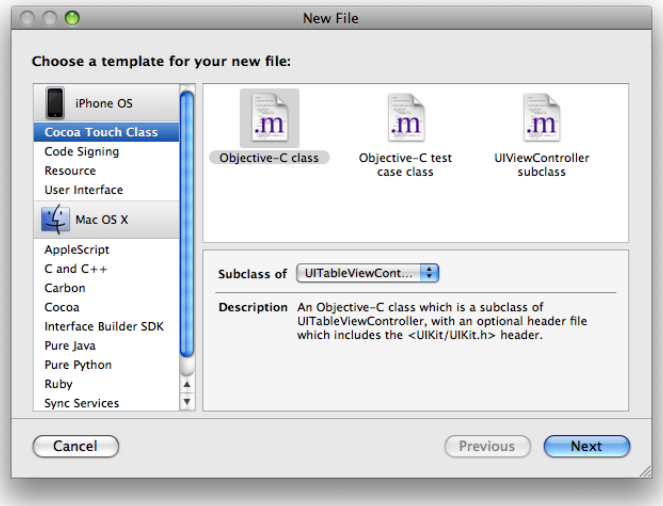
RootViewControllerが管理するTable Viewは、実行時には、Table View ControllerがTable Viewにデータを埋めるまでは空です。この手順については、「Table Viewへのデータの取り込み」（41 ページ）を参照してください。Navigation Barのタイトルや Table Viewのタイトルを指定するなど、Table ViewやNavigation Barをプログラムで設定することもできます（この設定コードの一般的な場所は、initWithStyle:メソッドまたはviewDidLoadメソッドです）。「Table Viewのオプション設定」（46 ページ）で、Table Viewの外観や動作をカスタマイズするためにできることをいくつか解説します。

アプリケーションへのTable Viewの追加

アプリケーションで1つ以上のTable Viewを表示したり管理したりする場合は、これらのTable Viewをプロジェクトに追加する必要があります。通常は、UITableViewControllerオブジェクトを追加することによってTable Viewを追加します。このオブジェクトが、管理対象のTable Viewを作成します。プロジェクトにカスタムのTable View Controllerクラスを追加した後で、Table Viewをプログラムで設定できます。または、実行時にロードされるnibファイルでTable Viewとそれに関連するビューを設定できます。Table Viewが単独のビューの場合は、プログラムによるアプローチの方が簡単です。nibファイルによるアプローチは、管理対象のビューがより複雑な場合（たとえば、サブビューのなかの1つだけがTable Viewであるようなビュー）に適しています。

カスタムのTable View Controllerクラスを追加するには、Xcodeプロジェクトを開き、「ファイル(File)」メニューから「新規ファイル(New File)」を選びます。このコマンドによって、図 4-6に示すような「新規ファイル(New File)」ウインドウが表示されます。左側の列から「Cocoa Touch Class」を選択し、「Objective-C class」の画像を選択します。最後に、「Subclass of」というラベルのポップアップメニューからUITableViewControllerを選び、「次へ(Next)」をクリックします。次のウインドウで、カスタムクラスのヘッダファイルと実装ファイルに適切な名前を付けて、「完了(Finish)」をクリックします。

図 4-6 `UITableViewController`のカスタムサブクラスの指定



Navigation-based Applicationテンプレートに含まれている`RootViewController`クラスと同様に、このカスタムのTable View Controllerクラスにも、Table Viewにデータを格納するためのメソッドを実装しなければなりません。また、Table ViewやNavigation Barのプロパティをプログラム（通常は、`initWithStyle:`メソッドまたは`viewDidLoad`メソッド内）で設定することもできます。ユーザによって、`RootViewController`クラスが管理するTable View内の項目が選択されたら、次のTable View Controllerのインスタンスの割り当てを行って初期化し、アプリケーションのNavigation Controllerが管理するスタックにプッシュします。この処理の詳細については、「選択の管理」（69 ページ）で説明します。

注： Table Viewへのデータの取り込みと、Table Viewの設定については、「Table Viewへのデータの取り込み」（41 ページ）および「Table Viewのオプション設定」（46 ページ）でそれぞれ解説します。

カスタムTable View Controllerが管理するTable Viewをnibファイルからロードする場合は、次の手順を実行する必要があります。

1. Interface Builderで、空のCocoa Touch nibファイルを作成します（「File」＞「New」）。
2. Interface BuilderのライブラリからUITableViewControllerオブジェクトをnibドキュメントウインドウにドラッグします。
3. このnibファイルを適切な名前でプロジェクトディレクトリに保存し、指定を求めたらプロジェクトにnibファイルを追加することを選択します。

4. nibドキュメントウインドウでTable View Controllerを選択し、インスペクタの「Identity」ペインを開きます。このクラスをTable View Controllerカスタムクラスに設定します。
5. nibドキュメントウインドウでFile's Ownerを選択し、そのクラス識別情報をTable View Controllerカスタムクラスに設定します。
6. Interface BuilderでTable Viewをカスタマイズします。
7. nibドキュメントウインドウでTable View Controllerを選択し、インスペクタの「Attributes」ペインを開き、「NIB Name」フィールドでnibファイル名を入力（または選択）します。

実行時に現在のTable View Controllerが新しいTable View Controllerのインスタンスを割り当てて初期化すると、それに対応するnibファイルがロードされます。

プログラムによるTable Viewの作成

Table Viewの管理に`UITableViewController`を使用しない場合は、このクラスが"自動的に"行う処理を自分で実装しなければなりません。Table Viewを作成するクラス（下の例では`UIViewController`のサブクラス）は、通常、`UITableViewDataSource`と`UITableViewDelegate`の各プロトコルを採用することによって、自身をデータソースとデリゲートに設定します。プロトコル採用の構文は、リスト 4-2に示すように、`@interface`ディレクティブ内のスーパークラスの直後に記述します。

リスト 4-2　データソースプロトコルとデリゲートプロトコルの採用

```
@interface RootViewController :UIViewController  <UITableViewDelegate,
UITableViewDataSource> {
    NSArray *timeZoneNames;
}

@property (nonatomic, retain) NSArray *timeZoneNames;
@end
```

次のステップでは、クライアントが、`UITableView`クラスのインスタンスを割り当てて初期化します。リスト 4-3に、プレーンスタイルの`UITableView`オブジェクトを作成するクライアントの例を示します。ここでは、自動サイズ変更の特性を指定し、自身をデータソースおよびデリゲートとして設定します。繰り返しになりますが、`UITableViewController`は、これらすべての処理を自動的に行ってくれることを忘れないでください。

リスト 4-3　Table Viewの作成

```
- (void)loadView
{
    UITableView *tableView = [[UITableView alloc] initWithFrame:[[UIScreen
mainScreen] applicationFrame]
                                style:UITableViewStylePlain];
    tableView.autoresizingMask =
UIViewAutoresizingFlexibleHeight|UIViewAutoresizingFlexibleWidth;
    tableView.delegate = self;
    tableView.dataSource = self;
    [tableView reloadData];

    self.view = tableView;
```

```
    [tableView release];
}
```

この例では、Table Viewを作成するクラスがUIViewControllerのサブクラスであるため、作成したTable Viewを、UIViewControllerクラスを継承するviewプロパティに代入しています。また、このTable ViewにreloadDataを送信します。これにより、Table Viewはセクションと行にデータを取り込む処理が開始されます。

Table Viewへのデータの取り込み

作成されるとすぐに、Table ViewオブジェクトはreloadDataメッセージを受け取ります。これは、セクションと行を表示するために必要な情報をデータソースとデリゲートに要求することを開始する指示です。Table Viewは、ただちにデータソースに論理的なディメンション（セクション数と各セクションの行数）を問い合わせます。次に、tableView:cellForRowAtIndexPath:を繰り返し呼び出して、表示する各行のセルオブジェクトを取得します。Table Viewは、このUITableViewCellオブジェクトを使用して、その行のコンテンツを描画します（Table Viewがスクロールされた場合にも、新たに表示する行に対してtableView:cellForRowAtIndexPath:が呼び出されます）。

「Table Viewへのデータの取り込み」は、データソースとデリゲートで、Table Viewを設定するために必要なプロトコルメソッドを実装する方法を示しています。

リスト 4-4　Table Viewへのデータの取り込み

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [regions count];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    // 行数は、指定されたセクションに対応する地域のタイムゾーンの数
    Region *region = [regions objectAtIndex:section];
    return [region.timeZoneWrappers count];
}


- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    // セクションのヘッダは地域名 -- このセクションインデックスの地域から取得する
    Region *region = [regions objectAtIndex:section];
    return [region name];
}


- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *MyIdentifier = @"MyIdentifier";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:MyIdentifier];
    if (cell == nil) {
       cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
 reuseIdentifier:MyIdentifier] autorelease];
    }
    Region *region = [regions objectAtIndex:indexPath.section];
```

```
    TimeZoneWrapper *timeZoneWrapper = [region.timeZoneWrappers
objectAtIndex:indexPath.row];
    cell.textLabel.text = timeZoneWrapper.localeName;
    return cell;
}
```

データソースは、そのtableView:cellForRowAtIndexPath:メソッドの実装で、設定済みのセルオブジェクトを返します。Table Viewはこれを使用して行を描画します。パフォーマンスの理由から、データソースは、できる限りセルを再利用します。データソースは、まずTable Viewに dequeueReusableCellWithIdentifier:メッセージを送信して、再利用可能な特定のセルオブジェクトを取得します。そのようなオブジェクトが存在しない場合は、データソースがオブジェクトを作成して、再利用識別子を割り当てます。データソースはセルのコンテンツ（この例ではテキスト）を設定して、それを返します。「Table View Cellの詳細」（49 ページ）では、このデータソースメソッドとUITableViewCellオブジェクトについてさらに詳しく説明します。

リスト 4-4のtableView:cellForRowAtIndexPath:の実装は、頻繁に見かけるTable View APIの使いかたを示しています。このメソッドには、データソースが提供することになっているセルに対応するTable Viewセクションと行を識別するNSIndexPath引数が含まれています（この例では、Table Viewが1つのセクションしか持たないため、セクションインデックスは不要です）。UIKitでは、NSIndexPathクラス（Foundationフレームワークで定義されている）のカテゴリが1つ宣言されています。このカテゴリは、セクションと行のインデックス番号によって、Table Viewの行を識別できるように、このクラスを拡張したものです。このカテゴリの詳細については、『*NSIndexPath UIKit Additions*』を参照してください。

インデックス付きリストへの取り込み

インデックス付きリスト（セクションインデックスTable Viewとも呼ばれます）は、アルファベットなどの慣習的な順序付け方式によって編成された大量のデータをナビゲートする場合に最適です。インデックス付きリストは、UITableViewDataSourceの3つのメソッド（sectionIndexTitlesForTableView:、tableView:titleForHeaderInSection:、およびtableView:sectionForSectionIndexTitle:atIndex:）を使用して特別に設定されたプレーンスタイルのTable Viewです。1番目のメソッドは、インデックスのエントリとして（順番に従って）使用する文字列の配列を返します。2番目のメソッドは、これらのインデックス文字列をTable Viewのセクションのタイトルと関連付けます（これらを、同一にする必要はありません）。3番目のメソッドは、ユーザがタップしたインデックスのエントリに関連するセクションインデックスを返します。

インデックス付きリストに埋めるために使用するデータは、このインデックスモデルを反映して編成されている必要があります。具体的には、配列の配列を作成する必要があります。内側の各配列はテーブル内の1つのセクションに対応します。セクションの配列は、慣習的な順序付け方法（通常は、AからZなどのアルファベット順）に従って、親の配列内でソートされ（順番に並べられ）ます。さらに、各セクション配列内の項目もソートされます。この配列の配列を自分で作成してソートすることもできますが、UILocalizedIndexedCollationクラスを利用すると、ごく簡単にこれらのデータ構造を作成してソートしたり、Table Viewにデータを提供したりできます。また、このクラスは、現在のローカライズ設定に従って配列内の項目を順番に並べます。

注： UILocalizedIndexedCollationクラスはiPhone OS 3.0でUIKitフレームワークに追加されました。

ただし、この配列の配列の構造を内部的に管理するのはデベロッパの責任です。順番に並べるオブジェクトは、UILocalizedIndexedCollationクラスが順に並べるために使用する文字列値を返すプロパティまたはメソッドを持っていなければなりません。メソッドの場合は、パラメータはありません。Table View内の行を表すインスタンスを生成するカスタムModelクラスを定義すると便利です。このようなModelオブジェクトは、文字列値を返すだけでなく、そのオブジェクトの割り当て先となるセクション配列のインデックスを保持するプロパティも定義します。リスト 4-5にこのようなクラスの定義を示します。このクラスでは、前者の目的でnameプロパティを、後者の目的でsectionNumberプロパティを宣言しています。

リスト 4-5 Modelオブジェクトのインターフェイスの定義

```
@interface State :NSObject {
    NSString *name;
    NSString *capitol;
    NSString *population;
    NSInteger sectionNumber;
}

@property(nonatomic,copy) NSString *name;
@property(nonatomic,copy) NSString *capitol;
@property(nonatomic,copy) NSString *population;
@property NSInteger sectionNumber;
@end
```

Table View ControllerにTable Viewへのデータの取り込みを要求する前に、使用するデータを何らかのソースからロードして、このデータからModelクラスのインスタンスを作成します。リスト 4-6の例では、プロパティリストで定義されているデータをロードして、それを基にModelオブジェクトを作成しています。また、UILocalizedIndexedCollationの共有インスタンスを取得して、セクション配列を含む可変の配列（states）を初期化しています。

リスト 4-6 Table Viewデータのロードと Modelオブジェクトの初期化

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UILocalizedIndexedCollation *theCollation = [UILocalizedIndexedCollation
currentCollation];
    self.states = [NSMutableArray arrayWithCapacity:1];

    NSString *thePath = [[NSBundle mainBundle] pathForResource:@"States"
ofType:@"plist"];
    NSArray *tempArray;
    NSMutableArray *statesTemp;
    if (thePath && (tempArray = [NSArray arrayWithContentsOfFile:thePath]) ) {
        statesTemp = [NSMutableArray arrayWithCapacity:1];
        for (NSDictionary *stateDict in tempArray) {
            State *aState = [[State alloc] init];
            aState.name = [stateDict objectForKey:@"Name"];
            aState.population = [stateDict objectForKey:@"Population"];
            aState.capitol = [stateDict objectForKey:@"Capitol"];
            [statesTemp addObject:aState];
            [aState release];
        }
```

```
    } else  {
        return;
    }
```

データソースに、この"未加工"のModelオブジェクト配列を持たせたら、UILocalizedIndexedCollationクラスの機能を利用してその配列を処理できます。リスト 4-7に示すコードでは、重要な部分に番号を付けています。

リスト 4-7　インデックス付きリスト用のデータの準備

```
    // viewDidLoadの続き…
    // (1)
    for (State *theState in statesTemp) {
        NSInteger sect = [theCollation sectionForObject:theState
collationStringSelector:@selector(name)];
        theState.sectionNumber = sect;
    }
    // (2)
    NSInteger highSection = [[theCollation sectionTitles] count];
    NSMutableArray *sectionArrays = [NSMutableArray
arrayWithCapacity:highSection];
    for (int i=0; i<=highSection; i++) {
        NSMutableArray *sectionArray = [NSMutableArray arrayWithCapacity:1];
        [sectionArrays addObject:sectionArray];
    }
    // (3)
    for (State *theState in statesTemp) {
       [(NSMutableArray *)[sectionArrays objectAtIndex:theState.sectionNumber]
 addObject:theState];
    }
    // (4)
    for (NSMutableArray *sectionArray in sectionArrays) {
       NSArray *sortedSection = [theCollation sortedArrayFromArray:sectionArray
            collationStringSelector:@selector(name)];
        [self.states addObject:sortedSection];
    }
} // viewDidLoadの終わり
```

1. データソースはModelオブジェクトの配列を列挙して、反復のたびに順序付けマネージャにsectionForObject:collationStringSelector:を送信します。このメソッドは、引数としてModelオブジェクトと、順序付けに使用するオブジェクトのプロパティまたはメソッドを取ります。呼び出しのたびに、そのModelオブジェクトを含むセクション配列のインデックスを返します。そして、その値をsectionNumberプロパティに代入します。

2. 次に、データソースは、（一時的な）親の可変配列と、各セクションに対応する可変配列を作成し、作成した各セクション配列を親の配列に追加します。

3. その後、Modelオブジェクトの配列を列挙して、各オブジェクトを割り当て済みのセクション配列に追加します。

4. データソースは、セクション配列の配列を列挙し、順序付けマネージャのsortedArrayFromArray:collationStringSelector:を呼び出して、各配列内の項目をソートします。その際、セクション配列と、その配列内の項目のソートに使用するプロパティまたはメソッドを渡します。ソートされたセクション配列は、最終的に親の配列に追加されます。

これで、データソースがTable Viewにデータを格納する準備ができました。データソースでは、リスト 4-8に示すようなインデックス付きリスト固有のメソッドを実装します。この処理の中で、`sectionIndexTitles`と`sectionForSectionIndexTitleAtIndex:`の2つの`UILocalizedIndexedCollation`メソッドを呼び出しています。また、`tableView:titleForHeaderInSection:`では、対応するセクションに項目がない場合は、Table Viewにヘッダが表示されないようにする点に注意してください。

リスト 4-8　セクションインデックスデータのTable Viewへの提供

```
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
    return [[UILocalizedIndexedCollation currentCollation] sectionIndexTitles];
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if ([[self.states objectAtIndex:section] count] > 0) {
        return [[[UILocalizedIndexedCollation currentCollation] sectionTitles]
 objectAtIndex:section];
    }
    return nil;
}

- (NSInteger)tableView:(UITableView *)tableView
sectionForSectionIndexTitle:(NSString *)title atIndex:(NSInteger)index
{
    return [[UILocalizedIndexedCollation currentCollation]
sectionForSectionIndexTitleAtIndex:index];
}
```

最後に、データソースでは、すべてのTable Viewに共通の`UITableViewDataSource`メソッドを実装しなければなりません。リスト 4-9にこれらの実装例と、Table Viewの`section`プロパティと`row`プロパティ（『*NSIndexPath UIKit Additions*』で説明されている`NSIndexPath`クラスの特有のカテゴリ）の使いかたを示します。

リスト 4-9　インデックス付きリストの行への取り込み

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [self.states count];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [[self.states objectAtIndex:section] count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"StateCell";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
       cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:CellIdentifier] autorelease];
    }
    State *stateObj = [[self.states objectAtIndex:indexPath.section]
objectAtIndex:indexPath.row];
    cell.textLabel.text = stateObj.name;
```

```
    return cell;
}
```

前述の手順に従ってTable Viewを初期データで埋めたら、`reloadSectionIndexTitles`（iPhone OS 3.0で導入されたメソッド）を呼び出して、そのインデックスのコンテンツを再ロードできます。

インデックス付きのリストのTable Viewの場合は、デリゲートが`tableView:cellForRowAtIndexPath:`で行にセルを割り当てるときに、そのセルの`accessoryType`プロパティを`UITableViewCellAccessoryNone`に設定しなければなりません。`reloadSectionIndexTitles`メソッド（iPhone OS 3.0で利用できる）を呼び出すことによって、インデックス付きリストのセクションタイトルを強制的に再表示できます。

Table Viewのオプション設定

Table View APIを利用すると、Table View（特定の行やセクションも含む）の外観や動作のさまざまな側面を設定できます。次の例は、オプションの使用例を示します。

Table Viewの作成と同じコードブロックで、`UITableView`クラスの特定のメソッドを使用して、グローバル設定を適用できます。リスト 4-10のコード例では、（`UILabel`オブジェクトを使用して）Table Viewにカスタムタイトルを追加します。

リスト 4-10　Table Viewにタイトルを追加する

```
- (void)loadView
{
    [super loadView];

    CGRect titleRect = CGRectMake(0, 0, 300, 40);
    UILabel *tableTitle = [[UILabel alloc] initWithFrame:titleRect];
    tableTitle.textColor = [UIColor blueColor];
    tableTitle.backgroundColor = [self.tableView backgroundColor];
    tableTitle.opaque = YES;
    tableTitle.font = [UIFont boldSystemFontOfSize:18];
    tableTitle.text = [curTrail objectForKey:@"Name"];
    self.tableView.tableHeaderView = tableTitle;
    [self.tableView reloadData];
    [tableTitle release];
}
```

リスト 4-11の例では、セクションヘッダのタイトル文字列を返します。

リスト 4-11　特定のセクションのヘッダタイトルを返す

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    // 次の物理的状態に基づいてセクションタイトルを返す：[solid, liquid, gas, artificial]
    return [[[PeriodicElements sharedPeriodicElements] elementPhysicalStatesArray]
 objectAtIndex:section];
}
```

リスト 4-12のコードでは、特定の行を次のインデントレベルまで移動します。

リスト 4-12　行のカスタムインデント

```
- (NSInteger)tableView:(UITableView *)tableView
indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath {
    if ( indexPath.section==TRAIL_MAP_SECTION && indexPath.row==0 ) {
        return 2;
    }
    return 1;
}
```

リスト 4-13の例では、インデックス値に基づいて特定の行の高さを変更します。

リスト 4-13　行の高さを変更する

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath
 *)indexPath
{
    CGFloat result;

    switch ([indexPath row])
    {
        case 0:
        {
            result = kUIRowHeight;
            break;
        }
        case 1:
        {
            result = kUIRowLabelHeight;
            break;
        }
    }
    return result;
}
```

`tableView:cellForRowAtIndexPath:`で、コンテンツ用の特殊な形式のサブビューを持つ`UITableViewCell`カスタムオブジェクトを返すことによって、行の外観を変更することもできます。セルのカスタマイズについては、「Table View Cellの詳細」（49 ページ）で説明します。

Table View Cellの詳細

TableViewは、セルオブジェクトを使用して、表示する行を描画します。また、行が表示されている間、これらのセルオブジェクトをキャッシュします。これらのセルオブジェクトは、`UITableViewCell`クラスを継承しています。Table Viewのデータソースは、`tableView:cellForRowAtIndexPath:`メソッドを実装することによって、セルオブジェクトをTable Viewに提供します。このメソッドは、`UITableViewDataSource`プロトコルの必須メソッドです。以降の各セクションでは、Table View Cellオブジェクトの特徴について述べ、`UITableViewCell`のデフォルトの機能を使用してセルのコンテンツを設定する方法を説明し、`UITableViewCell`カスタムオブジェクトを作成する方法を示します。

セルオブジェクトの特徴

セルオブジェクトには、TableViewのモードに応じて変更可能なさまざまな構成要素があります。通常、セルオブジェクトの大部分は、コンテンツ（テキスト、画像、またはその他の種類の識別子）用に確保されています。図5-1に示すように、セルの右側の小さな領域は、アクセサリビュー（ディスクロージャインジケータ、詳細ディスクロージャコントロール、スライダやスイッチなどのコントロールオブジェクト、カスタムビュー）のために確保されています。図 5-1は、セルの主な構成要素を示します。

図 5-1 Table View Cellの構成要素

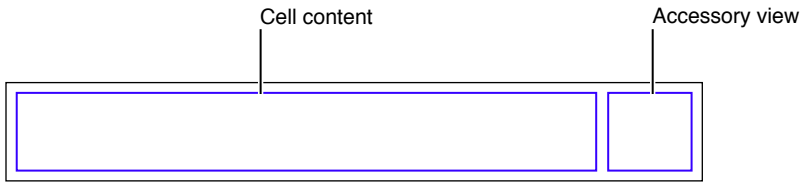
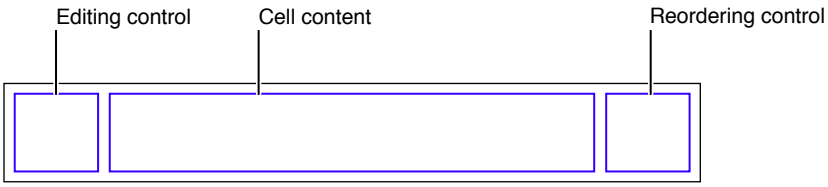


Table Viewが編集モードに入ると、各セルオブジェクトには編集用のコントロールが表示されます（編集用のコントロールを持つように設定されている場合）。編集用のコントロールは、図 5-2の左端に示されています。編集用のコントロールは、削除コントロール（丸の中に赤いマイナス記号）か、挿入コントロール（丸の中に緑のプラス記号）のいずれかです。セルのコンテンツは、編集用のコントロールのスペースを確保するために、右に移動します。セルオブジェクトが並べ替えられるように（Table View内で位置を移動できるように）設定されている場合は、並べ替えコントロールがセルの右側に（編集モード用に指定されたアクセサリビューの横に）表示されます。並べ替えコントロールは、水平方向の線が重なった記号です。TableView内で行の位置を移動するには、ユーザは並べ替えコントロールを押してセルをドラッグします。

図 5-2 Table View Cellの構成要素-編集モード



セルオブジェクトが再利用可能な場合（通常は再利用可能）、データソースは、そのセルを作成するときに、そのセルオブジェクトに再利用識別子（任意の文字列）を割り当てます。Table Viewは、このセルオブジェクトを内部キューに格納します。Table Viewが次に別のセルオブジェクトを要求すると、データソースは、Table Viewに`dequeueReusableCellWithIdentifier:`メッセージを送信して再利用識別子を渡し、キューに格納されているオブジェクトにアクセスします。データソースは、単に、そのセルのコンテンツと特殊なプロパティを設定して、セルオブジェクトを返します。このようにセルオブジェクトを再利用すると、セルを作成するためのオーバーヘッドがなくなるので、パフォーマンスが向上します。複数のセルオブジェクトを1つのキューに格納し、それぞれに固有の識別子を付けておけば、異なるタイプのセルオブジェクトで構成されたTable Viewを持つこともできます。たとえば、Table Viewのいくつかの行には、定義済みのスタイルの`UITableViewCell`の画像プロパティやテキストプロパティに基づくコンテンツを持たせ、別の行には、特殊な形式を定義するようにカスタマイズした`UITableViewCell`に基づくコンテンツを持たせることができます。

Table Viewにセルを提供する場合は、一般的な方法が3つあります。いくつかのスタイルの既成のセルオブジェクトを使用する方法、セルオブジェクトのコンテンツビューに独自のサブビューを追加する方法（Interface Builderアプリケーションで実行可能）、または`UITableViewCell`のカスタムサブクラスから作成したセルオブジェクトを使用する方法です。コンテンツビューはほかのビューの単なるコンテナであり、コンテンツビュー自身はコンテンツそのものを表示しません。

定義済みのスタイルのセルオブジェクトの使用

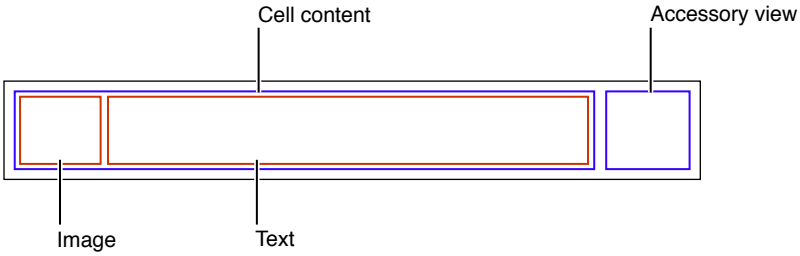
`UITableViewCell`クラスを直接使用すると、いくつかの定義済みのスタイルの"既成の"セルオブジェクトを作成できます。「Table View Cellの標準スタイル」（16 ページ）では、これらの標準的なセルについて説明し、Table Viewでの表示例を示しています。これらのセルは、`UITableViewCell.h`:で宣言されている次の`enum`定数に対応しています。

```
typedef enum {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
} UITableViewCellStyle;
```

注： 定義済みのセルスタイルとそれに対応するコンテンツプロパティは、iPhone OS 3.0で導入されました。

これらのセルオブジェクトは、1つ以上のタイトル（テキスト文字列）と、場合によっては画像の2種類のコンテンツを持ちます。図 5-3は、画像とテキストの大体の領域を示しています。画像が右に広がると、それに応じてテキストは右に移動します。

図 5-3 UITableViewCellオブジェクトのデフォルトのセルコンテンツ



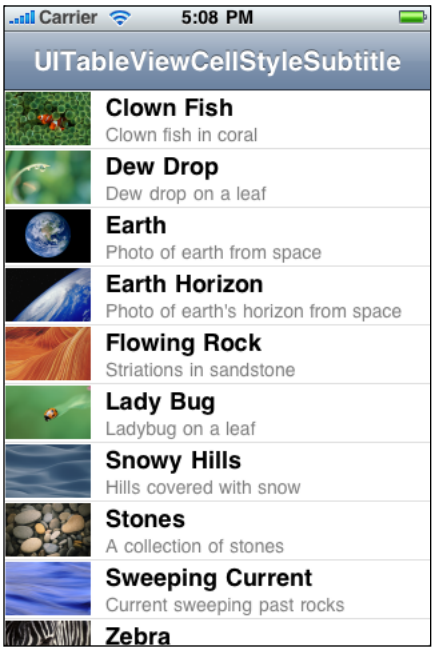
UITableViewCellクラスには、これらの定義済みセルスタイルのセルコンテンツ用に次のプロパティが定義されています。

- textLabel—セル内のテキストのメインラベル（UILabelオブジェクト）
- detailTextLabel—セル内のテキストのサブラベル（追加の詳細情報が存在する場合）（UILabelオブジェクト）。
- imageView—画像を保持する画像ビュー（UIImageViewオブジェクト）。

最初の2つのプロパティはラベルであるため、UILabelクラスで定義されているプロパティを介して、対応するテキストのフォント、配置、改行モード、および色を設定できます（行がハイライトされているときのテキストの色も含む）。画像ビューのプロパティに対しては、UIImageViewクラスのhighlightedImageプロパティを使用して、セルがハイライトされた場合の代替画像を設定することもできます。

図 5-4に、UITableViewCellStyleSubtitleスタイルのUITableViewCellオブジェクトを使用して描画された行を持つTable Viewの例を示します。

図 5-4 画像とテキストの両方を表示する行を持つTable View



リスト 5-1は、図 5-4 （51 ページ）に示したTable Viewを作成するデータソースの`tableView:cellForRowAtIndexPath:`の実装です。通常、データソースが最初に行うべきことは、Table View に`dequeueReusableCellWithIdentifier:`を送信して、再利用識別子の文字列を渡すことです。Table Viewが再利用可能なセルオブジェクトを返さない場合、データソースはセルオブジェクトを作成して、`initWithStyle:reuseIdentifier:`の最後のパラメータに再利用識別子を割り当てます。この時に、Table Viewのセルオブジェクトの全般的なプロパティ（この例では、選択スタイル）も設定します。次に、セルオブジェクトのコンテンツ（テキストと画像の両方）を設定します。

リスト 5-1　画像とテキストの両方を持つ`UITableViewCell`オブジェクトの設定

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"MyIdentifier"];
    if (cell == nil) {
       cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
 reuseIdentifier:@"MyIdentifier"];
        cell.selectionStyle = UITableViewCellSelectionStyleNone;
    }
    NSDictionary *item = (NSDictionary *)[self.content
objectAtIndex:indexPath.row];
    cell.textLabel.text = [item objectForKey:@"mainTitleKey"];
    cell.detailTextLabel.text = [item objectForKey:@"secondaryTitleKey"];
    NSString *path = [[NSBundle mainBundle] pathForResource:[item
objectForKey:@"imageKey"] ofType:@"png"];
    UIImage *theImage = [UIImage imageWithContentsOfFile:path];
    cell.imageView.image = theImage;
    return cell;
}
```

Table Viewのデータソースの`tableView:cellForRowAtIndexPath:`の実装で、セルを再利用する場合は、すべてのコンテンツを*必ず*リセットする必要があります。

`UITableViewCell`オブジェクトを設定するときには、ほかにも次に示す各種のプロパティを設定できます（ただし、これだけではありません）。

- `selectionStyle`—セルが選択されたときのセルの外観を制御します。
- `accessoryType`と`accessoryView`—標準アクセサリビュー（ディスクロージャインジケータ、または詳細ディスクロージャコントロール）のいずれか、またはカスタムアクセサリビューを通常（非編集）モードのセルに設定できます。カスタムビューの場合は、スライダ、スイッチ、カスタムビューなど任意の`UIView`オブジェクトを提供できます。
- `editingAccessoryType`と`editingAccessoryView`—標準アクセサリビュー（ディスクロージャインジケータ、または詳細ディスクロージャコントロール）のいずれか、またはカスタムアクセサリビューを編集モードのセルに設定できます。カスタムビューの場合は、スライダ、スイッチ、カスタムビューなど任意の`UIView`オブジェクトを提供できます（これらのプロパティはiPhone OS 3.0で導入されました）。
- `showsReorderControl`—編集モードに入ったときに、並べ替えコントロールを表示するかどうかを指定します。これに関連する読み取り専用の`editingStyle`プロパティは、セルの編集用コントロールのタイプを表します（セルに編集用コントロールがある場合）。デリゲートは、`tableView:editingStyleForRowAtIndexPath:`メソッドの実装で、`editingStyle`プロパティの値を返します。

- backgroundViewとselectedBackgroundView—セルのすべてのビューの背後に表示される（セルが選択されているときと、選択されていないときの）背景ビューを指定します。
- indentationLevelとindentationWidth—セルコンテンツのインデントレベルと、各インデントレベルの幅を指定します。

Table View CellはUIViewを継承しているため、当該スーパークラスが定義しているプロパティを設定することで、その外観と動作を変更することもできます。たとえば、セルの背景色を変更するには、スーパークラスのbackgroundColorプロパティを設定します。リスト 5-2に、Table Viewの行の背景色を（基になるセルを介して）交互に変更する方法を示します。

リスト 5-2　tableView:willDisplayCell:forRowAtIndexPath:でのセルの背景色の変更

```
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell
*)cell forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (indexPath.row == 0 || indexPath.row%2 == 0) {
        UIColor *altCellColor = [UIColor colorWithWhite:0.7 alpha:0.1];
        cell.backgroundColor = altCellColor;
    }
}
```

リスト 5-2は、Table View APIの重要な特性も示しています。Table Viewは、行を描画する直前に、デリゲートにtableView:willDisplayCell:forRowAtIndexPath:メッセージを送信します。デリゲートがこのメソッドを実装する場合は、セルが表示される前に、セルオブジェクトに最終の変更を加えることができます。デリゲートがこのメソッドで変更を加える場合には、それ以前にTable Viewによって設定された状態ベースのプロパティ（選択や背景色など、コンテンツ以外）を対象とするべきです。

セルのカスタマイズ

さまざまな定義済みスタイルのUITableViewCellオブジェクトを使用することで、Table Viewに表示するほとんどの行に十分対応できます。これらの既成のセルオブジェクトを利用すると、行に1つまたは2つのスタイルのテキスト、画像、および何らかの種類のアクセサリビューを含めることができます。アプリケーションは、テキストのフォント、色、その他の特性を変更できます。また、選択状態と通常状態の行に、それぞれ1つの画像を割り当てることもできます。

しかし、このセルコンテンツがいかに柔軟性が高く有用でも、必ずしもすべてのアプリケーションの要求を満たすことはできません。たとえば、ネイティブのUITableViewCellオブジェクトで使用可能なラベルは、行内の特定の位置に固定されます。また、画像は行の左端に表示されなければなりません。セルにさまざまなコンテンツ要素を持たせて、それらをさまざまな位置に配置したい場合や、セルにさまざまな動作をさせたい場合は、2つの選択肢があります。セルオブジェクトのcontentViewプロパティにサブビューを追加する方法、およびUITableViewCellのカスタムサブクラスを作成する方法の2つです。

- コンテンツのレイアウトは適切な自動サイズ変更設定によって完全に指定できるとしても、セルのデフォルトの動作は変更する必要がない場合は、セルのコンテンツビューにサブビューを追加するべきです。
- コンテンツにカスタムレイアウトコードが必要な場合や、セルのデフォルトの動作（編集モードへの応答など）を変更する必要がある場合は、カスタムサブクラスを作成するべきです。

以降のセクションでは、この両方のアプローチを説明します。

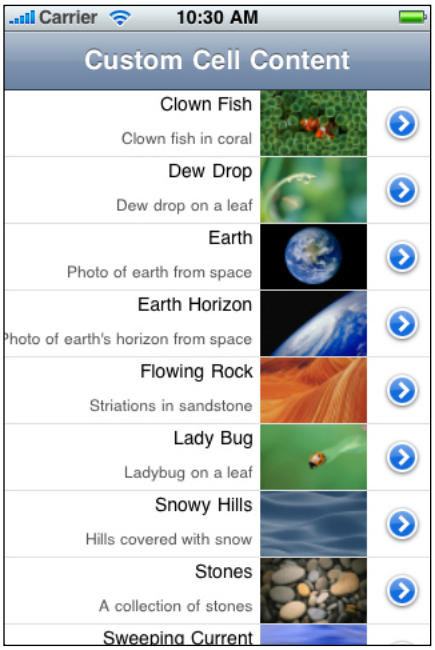
セルのコンテンツビューにプログラムでサブビューを追加する

Table Viewが行を表示するために使用するセルは、ビューの一種です（`UITableViewCell`は`UIView`を継承しています）。ビューとして、セルはコンテンツビュー（セルコンテンツ用のスーパービュー）を持っています。コンテンツビューは、プロパティとして公開されています。Table Viewの行の外観をカスタマイズするには、セルのコンテンツビュー（`contentView`プロパティを介してアクセス可能）にサブビューを追加し、スーパービューの座標系の中でそのサブビューを希望の位置に配置できます。サブビューの設定や配置は、プログラムの中で、あるいはInterface Builderを使用して行えます（Interface Builderを使用して行う方法については「カスタムTable View CellをnibファイルからロードするJ （56 ページ）で説明します）。

このアプローチの利点の1つは、比較的簡単に実現できる点です。`UITableViewCell`のカスタムサブクラスを作成して、カスタムビューに必要な実装の詳細をすべて扱う必要がありません。ただし、このアプローチをとる場合は、可能であれば、ビューを透明にするのは避けます。透明なサブビューは、画像合成のための処理が増加するため、スクロールのパフォーマンスに影響を与えます。サブビューは不透明にし、通常はセルと同じ背景色にすべきです。セルが選択可能な場合は、選択時に必ずセルコンテンツが適切にハイライト状態になるようにします。サブビューで`highlighted`プロパティのアクセサメソッドを（必要に応じて）実装していれば自動的にこの動作になります。

セルの中でテキストと画像のコンテンツを、標準のセルスタイルでの位置とはまったく異なる位置に配置したい場合を考えます。たとえば、画像をセルの右端に配置し、セルのタイトルとサブタイトルをその画像の左側に右寄せで表示するとします。図 5-5に、このようなセルで描画された行を持つTable Viewの外観を示します（この例は、実例を示すためだけのもので、ヒューマンインターフェイスのモデルを示すことが目的ではありません）。

図 5-5　カスタムコンテンツをサブビューとして持つセル



リスト 5-3のコード例は、このTable Viewが行を描画するために使用するセルを、プログラムの中でデータソースから作成する方法を示しています。tableView:cellForRowAtIndexPath:では、まず、特定の再利用識別子を持つセルオブジェクトをTable Viewがすでに持っているかどうかをチェックします。そのようなオブジェクトが存在しない場合、データソースは、2つのラベルオブジェクトと、固有のフレームを持つ1つの画像ビューを、スーパービュー（コンテンツビュー）の座標系内に作成します。また、これらのオブジェクトの属性も設定します。次に（または、セルがTable Viewのキューにすでに存在する場合）、データソースは、セルのコンテンツを設定して、そのセルを返します。

リスト 5-3　セルのコンテンツビューにサブビューを追加する

```
#define MAINLABEL_TAG 1
#define SECONDLABEL_TAG 2
#define PHOTO_TAG 3

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"ImageOnRightCell";

    UILabel *mainLabel, *secondLabel;
    UIImageView *photo;
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
       cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
 reuseIdentifier:CellIdentifier] autorelease];
        cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;

        mainLabel = [[[UILabel alloc] initWithFrame:CGRectMake(0.0, 0.0, 220.0,
 15.0)] autorelease];
        mainLabel.tag = MAINLABEL_TAG;
        mainLabel.font = [UIFont systemFontOfSize:14.0];
        mainLabel.textAlignment = UITextAlignmentRight;
        mainLabel.textColor = [UIColor blackColor];
        mainLabel.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin |
UIViewAutoresizingFlexibleHeight;
        [cell.contentView addSubview:mainLabel];

        secondLabel = [[[UILabel alloc] initWithFrame:CGRectMake(0.0, 20.0,
220.0, 25.0)] autorelease];
        secondLabel.tag = SECONDLABEL_TAG;
        secondLabel.font = [UIFont systemFontOfSize:12.0];
        secondLabel.textAlignment = UITextAlignmentRight;
        secondLabel.textColor = [UIColor darkGrayColor];
        secondLabel.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin |
UIViewAutoresizingFlexibleHeight;
        [cell.contentView addSubview:secondLabel];

       photo = [[[UIImageView alloc] initWithFrame:CGRectMake(225.0, 0.0, 80.0,
 45.0)] autorelease];
        photo.tag = PHOTO_TAG;
        photo.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin |
UIViewAutoresizingFlexibleHeight;
        [cell.contentView addSubview:photo];
    } else {
        mainLabel = (UILabel *)[cell.contentView viewWithTag:MAINLABEL_TAG];
```

```
        secondLabel = (UILabel *)[cell.contentView viewWithTag:SECONDLABEL_TAG];
         photo = (UIImageView *)[cell.contentView viewWithTag:PHOTO_TAG];
    }
    NSDictionary *aDict = [self.list objectAtIndex:indexPath.row];
    mainLabel.text = [aDict objectForKey:@"mainTitleKey"];
    secondLabel.text = [aDict objectForKey:@"secondaryTitleKey"];
    NSString *imagePath = [[NSBundle mainBundle] pathForResource:[aDict
objectForKey:@"imageKey"] ofType:@"png"];
    UIImage *theImage = [UIImage imageWithContentsOfFile:imagePath];
    photo.image = theImage;

    return cell;
}
```

データソースは、セルを作成するときにサブビューごとに1つのタグを割り当てます。タグはビューの識別子です。これによって、`viewWithTag:`メソッドを呼び出すことで、ビュー階層内のビューを見つけることができます。後で、デリゲートが指名されたセルをTable Viewのキューから取得する場合は、このタグを使用して3つのサブビューへの参照を取得し、それらにコンテンツを割り当てます。

また、このコードでは`UITableViewCell`オブジェクトを定義済みのデフォルトスタイル(`UITableViewCellStyleDefault`)で作成します。ただし、標準セルのコンテンツプロパティ（`textLabel`、`detailTextLabel`、および`imageView`）は、コンテンツが割り当てられるまでは`nil`であるため、任意の定義済みセルをカスタマイズ用のテンプレートとして使用できます。

注： 図 5-5 （54 ページ）に示したものと同じセルをプログラムで作成するための、おそらくもっと簡単なもう1つの方法は、`UITableViewCell`のサブクラスを定義し、`UITableViewCellStyleSubtitle`スタイルのインスタンスを作成することです。そして、（`super`を呼び出した後に）`layoutSubviews`メソッドをオーバーライドして、`textLabel`、`detailTextLabel`および`imageView`の各サブビューの配置を変えるようにします。

テキストコンテンツに“文字列属性”を付加するために、よく使うテクニックは、`UITableViewCell`のコンテンツビューに`UILabel`のサブビューを配置する方法です。各ラベルのテキストには、固有のフォント、色、サイズ、配置、その他の特性を持たせることができます。1つのラベルオブジェクト内でテキスト属性にバリエーションを持たせることはできなので、複数のラベルを作成して、望みのバリエーションになるように、それらを並べて配置する必要があります。

カスタムTable View Cellをnibファイルからロードする

Table View Cellのサブビューについて、プログラミングによる方法と同様のカスタマイズをInterface Builderでも簡単に行うことができます。nibファイル内のセルに対しては、そのセルに対応する行コンテンツが静的か動的かに応じて、次のいずれかの方法が必要になります。動的なコンテンツの場合、Table Viewは多数の（無限にある可能性もある）行を持つリストです。静的なコンテンツの場合、行数は有限で既知の数です。項目の詳細ビューを表示するTable Viewは、通常、静的なコンテンツを持ちます。この2つのコンテンツタイプは、nibファイルでは扱いが異なります。動的なコンテンツの場合は、各`UITableViewCell`オブジェクトはそれぞれ個別のnibファイルに入っていなければなりません。静的なコンテンツの場合は、Table Viewで使われる複数の`UITableViewCell`オブジェクトを、そのTable Viewと同じnibファイルに入れることができます。

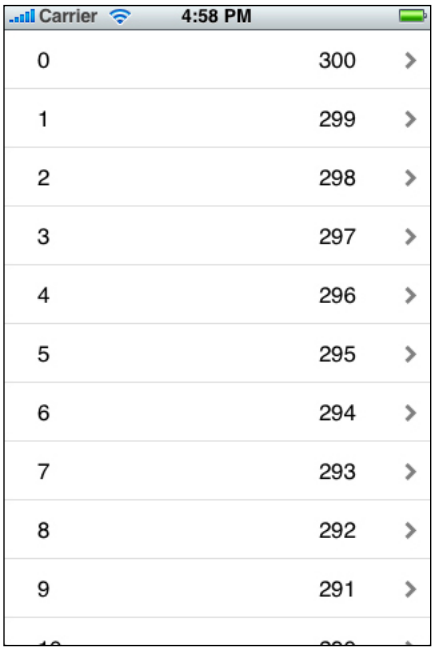
以降の各セクションでは、カスタム構成されたTable View Cellを含むnibファイルをロードするさまざまな方法を示します。また、セルのサブビューにアクセスするためのいくつかの手法も示します。1つは、タグ（`UIView`で定義されている`tag`プロパティ）を使用する手法、もう1つは、Table

View Controllerカスタムクラスで宣言されているアウトレットプロパティを使用する手法です。前者の手法は、タグ付きビューの検索に伴っていくらか負担が生じます。後者の手法では、アウトレット接続を介してサブビューに直接アクセスできます。

動的な行コンテンツのための手法

このセクションでは、Interface Builderで単純なTable View Cellを作成して、nibファイルに保存します。データソースは、実行時にこのnibファイルをロードしてセルを準備し、図 5-6に示す行を描画するために、それをTable Viewに渡します。

図 5-6 nibファイルからのセルを使用したTable Viewの行の描画



このTable Viewを管理するView Controller（通常は`UITableViewController`オブジェクト）には、nibファイルからロードする予定のカスタマイズされたTable Viewセルのoutletプロパティを定義します。その定義をリスト 5-4に示します。実装ファイルには、このプロパティ用のアクセサメソッドを必ず作成します。

リスト 5-4 セルのアウトレットの定義

```
@interface TVController :UITableViewController {
    UITableViewCell *tvCell;
}

@property (nonatomic, assign) IBOutlet UITableViewCell *tvCell;
@end
```

Interface Builderで、次の手順を実行します。

1. nibファイルを作成します。

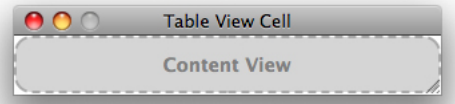
   「File」メニューから「New」を選び、Emptyテンプレートを選択して、「Choose」をクリックします。

2. このnibファイルを適切な名前で保存し、プロンプトが表示されたらプロジェクトに追加します。

   この名前は、アプリケーションのメインバンドルからこのnibファイルをロードするための`loadNibNamed:owner:options:`メソッド呼び出しの第1引数として指定する名前です。コード例については、リスト 5-5 （59 ページ）を参照してください。

3. Table View Cellオブジェクトをライブラリからnibドキュメントウインドウにドラッグします。

   このセルオブジェクトはコンテンツビューの位置を表します。



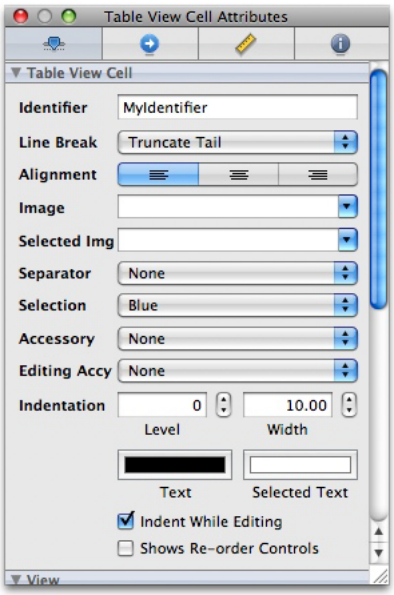
4. オブジェクトをライブラリからこのコンテンツビューにドラッグします。

   この例では、2つのラベルオブジェクトをドラッグして、各セルの端の近くに（アクセサリビュー用のスペースを残して）配置します。

5. コンテンツビュー上のオブジェクトを選択して、属性、サイズ、自動サイズ調整特性を設定します。

   この手順のプログラム部分用に設定すべき重要な属性は、各オブジェクトの`tag`プロパティです。「Attributes」ペインの「View」セクションでこのプロパティを見つけて、各オブジェクトに一意の整数を割り当てます。

6. セル自体を選択し、セルに持たせたい一般的な属性（配置、フォントサイズ、色、改行モードなど）を設定します。



セルの識別子としての文字列を必ず設定します。Table Viewは、キャッシュからセルを取得するためにこの識別子を必要とします（キャッシュが存在する場合）。識別子を設定することは、Table Viewに複数の種類のセルを持たせる場合には、特に重要です。ここで、セルの高さや自動サイズ調整特性を設定することもできます。

7. nibドキュメントウインドウでFile's Ownerを選択して、インスペクタの「Identity」ペインを開き、File's OwnerのクラスとしてView Controllerカスタムクラスを設定します。

8. File's Ownerのセルアウトレット（この時点ではカスタムサブクラスのプレースホルダインスタンス）をnibファイルドキュメント内のTable View Cellオブジェクトに接続します。

ここで、nibファイルを保存し、Xcodeプロジェクトに戻ります。Table Viewのデータを取得してTable Viewをセットアップするためのすべてのコードを通常どおり記述します。`tableView:cellForRowAtIndexPath:`以外のデータソースメソッドを通常どおり実装します。リスト 5-5の例と同様にこれを実装します。

リスト 5-5　nibファイルからセルをロードしてコンテンツを割り当てる

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *MyIdentifier = @"MyIdentifier";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:MyIdentifier];
    if (cell == nil) {
        [[NSBundle mainBundle] loadNibNamed:@"TVCell" owner:self options:nil];
        cell = tvCell;
        self.tvCell = nil;
    }
    UILabel *label;
    label = (UILabel *)[cell viewWithTag:1];
    label.text = [NSString stringWithFormat:@"%d", indexPath.row];

    label = (UILabel *)[cell viewWithTag:2];
    label.text = [NSString stringWithFormat:@"%d", NUMBER_OF_ROWS -
indexPath.row];

    return cell;
}
```

このコードには、注目すべき優れた点がいくつかあります。

- Interface Builderでセルに割り当てた文字列識別子は、`dequeueReusableCellWithIdentifier:`でTable Viewに渡される文字列と同じです。このセルは、Table Viewを埋めるために必要に応じてnibファイルからロードされます。行がビューの範囲外へスクロールされると、対応するセルはTable Viewのキャッシュ内で利用可能になります。

  Table Viewは、さまざまな理由でいつでも追加の（再利用キューに存在する数よりも多くの）セルを要求する可能性があることを忘れないでください。たとえば、新しい行を100行挿入すると、Table Viewは一度に100個の新しいセルを要求することが考えられます。それらは、挿入操作が完了するまで再利用キューには保持されません。

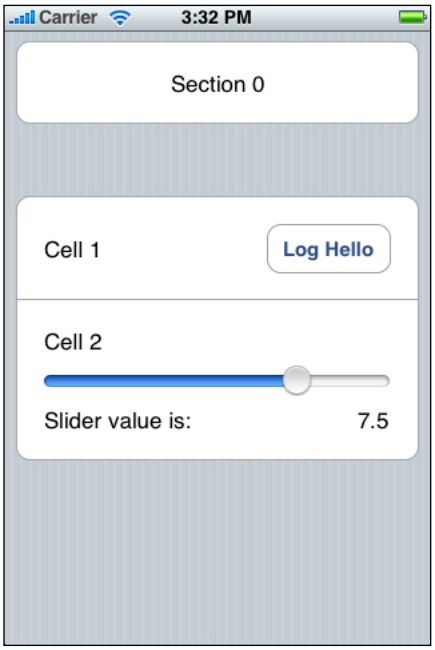
- nibファイルは`NSBundle`の`loadNibNamed:owner:options:`メソッドを使用してロードされます。このメソッドは、所有者として`self`を渡します（File's OwnerはTable View Controllerを指すことを忘れないでください）。

- アウトレット接続を作成したことにより、tvCellアウトレットは、nibファイルからロードされたセルへの参照を持ちます。渡されたcell変数にただちにこのセルを代入し、アウトレットをnilに設定します。
- このコードでは、セルのラベルを取得するためにviewWithTag:を呼び出し、ラベルのタグ整数を渡しています。これで、このラベルのテキストコンテンツを設定できます。

静的な行コンテンツのための手法

このセクションでは、Interface BuilderでTable View Cellをいくつか作成し、Table Viewを含むものと同じnibファイルに保存します。実行時、Table Viewがnibファイルからロードされると、データソースはこれらのセルに直接アクセスし、これらを使用してTable Viewのセクションや行を作成します（図5-7に示します）。

図 5-7 nibファイルからの複数のセルを使用して描画したTable Viewの行



動的コンテンツの手順と同様に、UITableViewControllerのサブクラスをプロジェクトに追加することから始めます（詳細については、「アプリケーションへのTable Viewの追加」（38 ページ）を参照してください）。リスト5-6に示すように、nibファイル内の3つのセルのそれぞれに対して、および最後のセルのスライダ値のラベルに対して、アウトレットプロパティを定義します。

リスト 5-6 nibファイル内のセルのアウトレットプロパティの定義

```
@interface MyTableViewController :UITableViewController {

    UITableViewCell *cell0;
    UITableViewCell *cell1;
    UITableViewCell *cell2;
    UILabel *cell2Label;
}
```

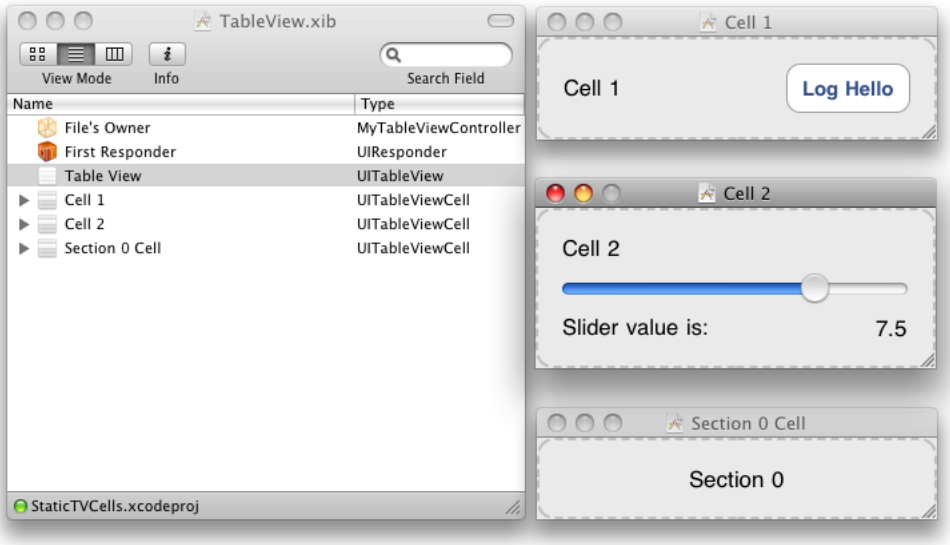
```
@property (nonatomic, retain) IBOutlet UITableViewCell *cell0;
@property (nonatomic, retain) IBOutlet UITableViewCell *cell1;
@property (nonatomic, retain) IBOutlet UITableViewCell *cell2;
@property (nonatomic, retain) IBOutlet UILabel *cell2Label;

- (IBAction)logHello;
- (IBAction)sliderValueChanged:(UISlider *)slider;

@end
```

Table Viewを含むnibファイルを作成したら、カスタムTable View ControllerをこのnibファイルのFile's Ownerに設定します。このコントローラのビューアウトレットをTable Viewに接続し、インスペクタの「Attributes」ペインで、このTable Viewのスタイルを Groupedに変更します（その方法については、「Table Viewアプリケーションの簡単な作成方法」 （35 ページ）を参照してください）。次に、セルに対して次の手順を実行します。

1. 3つのTable View Cellオブジェクトを Interface Builderのライブラリからnibドキュメントウインドウにドラッグします。
2. ライブラリからオブジェクトをドラッグして、次に示すように各セルのサブビューを作成します。

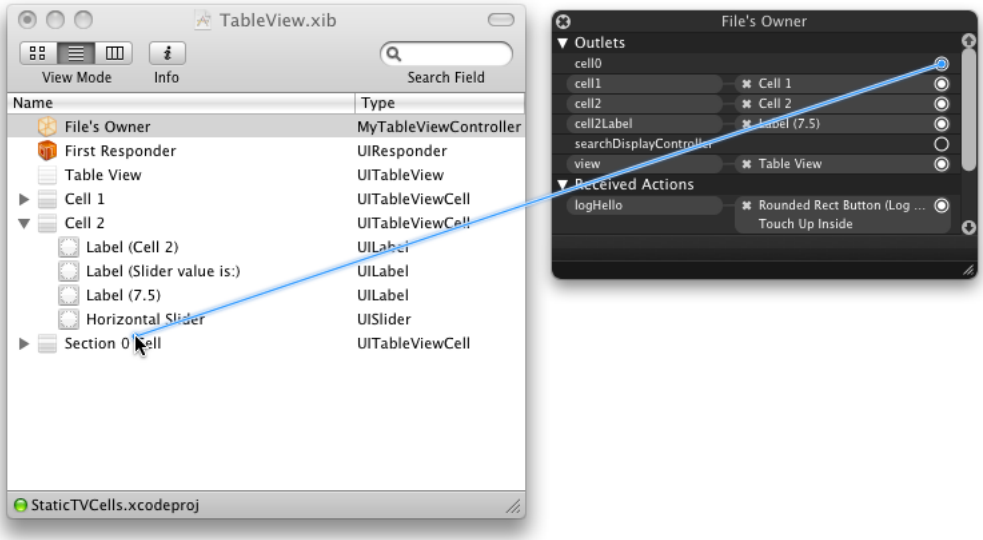


これらのセルは一度だけ使用するセルのため、これらのセルの属性に識別子を割り当てる必要はありません。

3. これらのオブジェクトに希望の属性を設定します。

   たとえば、スライダの値を0から10までの範囲に設定し、初期値を7.5に設定します。

4. nibドキュメントウインドウでFile's Ownerを右クリック（または、Controlキーを押しながらクリック）し、接続ウインドウを表示します。File's Owner (Table View Controller)と各セルオブジェクトとの間にアウトレット接続を作成します。



また、`cell2Label`アウトレットをスライダ値のラベルに接続します。静的なコンテンツの場合は、アウトレット接続を作成できるため、このようなタグをビューに付ける必要はありません。

5. ここで、リスト 5-6 （60 ページ）で宣言した2つのアクションメソッドを実装して、上の図に示すようなターゲット-アクション接続を作成します。

このnibファイルを保存して、Xcodeプロジェクトに戻り、Table View用のデータソースメソッドを実装します。アプリケーションデリゲートまたは直前のTable View Controllerが、現在のTable View Controllerをインスタンス化するときに、Table ViewとTable View Cellを含むこのnibファイルが、アプリケーションのメモリにロードされます。nibファイル内のセルは一度だけ使用するセルのため、それらを（アウトレットを介して）Table Viewに返す必要があるのは、`tableView:cellForRowAtIndexPath:`メソッドでTable Viewが要求したときにのみです（リスト 5-7に示します）。

リスト 5-7　nibファイルのセルをTable Viewに渡す

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    if (indexPath.section == 0) {
        return cell0;
    }
    // セクション1
    if (indexPath.row == 0) {
        return cell1;
    }
    return cell2;
}
```

nibファイル内のセルが静的コンテンツ用で、Table Viewで一度だけ使用されるセルの場合でも、そのコンテンツを動的に変更することができます。たとえば、1番目のセルのラベルのテキスト値を、現在のデータ項目のプロパティに簡単に変更できます。そのコードをリスト 5-8に示します。

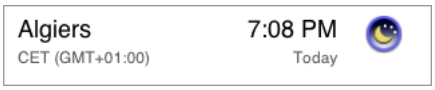
リスト 5-8　nibファイルのセルのコンテンツを動的に変更する

```
// ...
if (indexPath.section == 0) {
        UILabel *theLabel = (UILabel *)[cell0 viewWithTag:1];
        theLabel.text  = [self.currentAddress lastName];
        return cell0;
    }
// ...
```

UITableViewCellのサブクラス化

セルオブジェクトの外観をカスタマイズするもう1つの方法は、独自のコンテンツを描画するUITableViewCellのカスタムサブクラスを作成することです。次にこのクラスのインスタンスを使用してTable Viewの行を埋めます。また、この方法ではセルの動作（たとえば、Table Viewが編集モードに入ったときの動作）の制御範囲が広がります（編集モードでは、コンテンツ領域が縮小します）。図 5-8に、カスタムTable View Cellオブジェクトの例を示します。

図 5-8　カスタムTable View Cell



サブクラス化するコードを書き始める前に、UITableViewCellのサブクラスの設計とパフォーマンスの制約について注意深く検討します。

- **セル全体の描画は必要なときにだけ行う**。UITableViewCellのサブクラスを使用すると、コンテンツのすべてをdrawRect:メソッドで描画することができますが、不都合な点もあり得ることを認識しておく必要があります。カスタム描画は、セルのレイヤに適用されますが、そのレイヤが、その手前に配置されているビューによって覆い隠される場合があります。たとえば、グループスタイルのTable Viewでは、背景ビュー（backgroundViewプロパティ）が、drawRect:で実行された描画をすべて覆い隠します。選択範囲の青い背景も、描画を覆い隠します。さらに、アニメーション中（Table Viewが編集モードに入ったり、編集モードから出るときなど）に発生するカスタム描画は、パフォーマンスを大幅に低下させます。

  代わりの方法としては、サブビューを使用してセルのコンテンツを構成するようにして、それらのビューを望みどおりの配置にレイアウトするサブクラスを用意するという方法があります。これらのビューはキャッシュされるため、（たとえば、セルが編集モードに入った場合は）単にビューを移動させるだけで済みます。「セルのコンテンツビューにプログラムでサブビューを追加する」（54 ページ）では、このようなアプローチと、もう1つ別のアプローチを示しています。

  しかし、1つのセルのコンテンツを3〜4個以上のサブビューで構成すると、スクロールのパフォーマンスが低下する可能性があります。この場合は（特に、セルが編集可能でない場合は）、セルのコンテンツビューの1つのサブビューに直接描画することを検討します。このガイドラインの要点は、カスタムTable View Cellを実装する場合は、最適なスクロールパフォーマンスと最適な編集または並べ替えパフォーマンスとの間のトレードオフを認識する必要があるということです。

- **透明を避ける。**Table View Cellのサブビューには、画像合成のための負荷がかかります。この負荷は、ビューを不透明にすることによって大幅に軽減できます。各セルに透明のサブビューが1つあるだけで、スクロールのパフォーマンスに影響が出ます。可能な場合は、常に不透明のサブビューを使用します。
- **表示可能なプロパティが変更されたときは、セルを表示する必要があることを示すマークを付ける。**再利用可能なカスタムテーブルセルがあり、それがセルコンテンツの一部としてカスタムプロパティを表示する場合、そのプロパティの値が変わったら、そのセルに`setNeedsDisplay`メッセージを送信する必要があります。そうしないと、UIKitは、そのセルが"ダーティ"である（変更されている）ことを認識しないので、そのセルの`drawRect:`メソッドを呼び出して、新しい値でセルを再描画しません。`setNeedsDisplay`を呼び出すのに適した場所は、そのプロパティの（自動生成されたものではない）setterメソッド内です。

このセクションのここから先は、`CustomTableViewCell`プロジェクトの、`UITableViewCell`のカスタムサブクラスを実装した部分について解説していきます（このプロジェクトは、*TableViewSuite*の拡張サンプルに含まれています）。このサブクラスは複雑なコンテンツを持つセルを実装しています。コンテンツが複雑なため、自身を描画する単一のカスタムビューを持ちます。このプロジェクトが、どのようにして図 5-8 （63 ページ）で示したカスタムセルオブジェクトを作成するかを考察することで、`UITableViewCell`のカスタムサブクラスを作成する実践的な方法を理解することができます。

リスト 5-9に示すように、`CustomTableViewCell`プロジェクトでは、`UITableViewCell`の`TimeZoneCell`サブクラスのインターフェイスを宣言しています。このインターフェイスは単純で、カスタムビュークラスへの1つの参照と2つのメソッド（1つは、カスタムビューで描画に使用するコンテンツを設定するメソッド。もう1つは、必要に応じてセルを再描画するメソッド）で構成されています。

リスト 5-9　`TimeZoneCell`クラスのプロパティとメソッドの宣言

```
@class TimeZoneWrapper;
@class TimeZoneView;

@interface TimeZoneCell :UITableViewCell {
    TimeZoneView *timeZoneView;
}

@property (nonatomic, retain) TimeZoneView *timeZoneView;
- (void)setTimeZoneWrapper:(TimeZoneWrapper *)newTimeZoneWrapper;
- (void)redisplay;
@end
```

`setTimeZoneWrapper:`メソッドは、引数として、タイムゾーンを表すカスタムModelオブジェクトを取ります。また、計算コストの高い派生プロパティを遅延作成したり、キャッシュしたりします。`TimeZoneWrapper`クラスは、そのインスタンスが各セルのコンテンツのソースとなるため、重要です（ここでは、このクラスの実装は示しません）。

その実装内で、`TimeZoneCell`クラスは、スーパークラスの`initWithStyle:reuseIdentifier:`イニシャライザをオーバーライドして、最初のステップでスーパークラスの実装を呼び出します。このメソッドで、セルのコンテンツビューの境界に合ったサイズの`TimeZoneView`クラスのインスタンスを作成します。このビューの自動サイズ変更特性を設定し、それをセルのコンテンツビューのサブビューとして追加します。リスト 5-10に、この初期化コードを示します。

リスト 5-10　TimeZoneCellインスタンスの初期化

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString
*)reuseIdentifier {
    if (self = [super initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:reuseIdentifier]) {
       CGRect tzvFrame = CGRectMake(0.0, 0.0, self.contentView.bounds.size.width,
             self.contentView.bounds.size.height);
        timeZoneView = [[TimeZoneView alloc] initWithFrame:tzvFrame];
        timeZoneView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
        [self.contentView addSubview:timeZoneView];
    }
    return self;
}
```

TimeZoneViewクラスは、リスト5-11に示すようなインターフェイスを持ちます。TimeZoneWrapperインスタンスのほかに、このクラスは、データフォーマッタ、略語、および2つのフラグ（セルがハイライトされているかどうかを示すフラグと、編集モードになっているかどうかを示すフラグ）をカプセル化しています。

リスト 5-11　TimeZoneViewクラスのインターフェイスの宣言

```
@interface TimeZoneView :UIView {
    TimeZoneWrapper *timeZoneWrapper;
    NSDateFormatter *dateFormatter;
    NSString *abbreviation;
    BOOL highlighted;
    BOOL editing;
}
@property (nonatomic, retain) TimeZoneWrapper *timeZoneWrapper;
@property (nonatomic, retain) NSDateFormatter *dateFormatter;
@property (nonatomic, retain) NSString *abbreviation;
@property (nonatomic, getter=isHighlighted) BOOL highlighted;
@property (nonatomic, getter=isEditing) BOOL editing;

@end
```

TableViewCellクラスが、TimeZoneWrapperオブジェクトを設定するメソッドを宣言していたことを思い出してください。このメソッドは、TimeZoneCellでカプセル化されているTimeZoneViewクラスの（自動生成されたのではない）同じsetterメソッドを呼び出しているだけです。このsetterメソッドは、リスト5-12に示すように実装されています。このsetterメソッドは、標準的なメモリ管理コードを提供するだけでなく、タイムゾーンとデータフォーマッタを関連づけて、タイムゾーンの略称を作成し、ビューに再表示マークを付加します。

リスト 5-12　タイムゾーンラッパーとそれに関連する値の設定

```
- (void)setTimeZoneWrapper:(TimeZoneWrapper *)newTimeZoneWrapper {

    // タイムゾーンラッパーが変更された場合は、データフォーマッタと略称文字列を更新する
    if (timeZoneWrapper != newTimeZoneWrapper) {
        [timeZoneWrapper release];
        timeZoneWrapper = [newTimeZoneWrapper retain];
        [dateFormatter setTimeZone:timeZoneWrapper.timeZone];
        NSString *string = [[NSString alloc] initWithFormat:@"%@ (%@)",
timeZoneWrapper.abbreviation, timeZoneWrapper.gmtOffset];
        self.abbreviation = string;
```

```
        [string release];
    }
    [self setNeedsDisplay];
}
```

TimeZoneViewクラスに再表示マークが付加されると、そのクラスのdrawRect:メソッドが呼び出されます。リスト 5-13に、TimeZoneViewの実装の代表的なセクションを示します。簡潔にするためにほかの部分は省略しています。省略されている部分の1つに、ビューのフィールドをレイアウトするための定数と、セルが通常状態かハイライト状態かに応じて異なる描画テキストの色を定義する初期コードがあります。この実装では、テキストの描画にはNSStringのdrawAtPoint:forWidth:withFont:fontSize:lineBreakMode:baselineAdjustment:メソッドを使用し、画像の描画にはUIImageのdrawAtPoint:メソッドを使用しています。

リスト 5-13　カスタムTable View Cellの描画

```
- (void)drawRect:(CGRect)rect {

    // #define定数とフォントをここでセットアップする...
   // 通常状態とハイライト状態のセル内のメインテキストとサブテキストの色をセットアップする...

    CGRect contentRect = self.bounds;
    if (!self.editing) {
        CGFloat boundsX = contentRect.origin.x;
        CGPoint point;
        CGFloat actualFontSize;
        CGSize size;

        // メインテキストを描画する
        [mainTextColor set];

        // タイムゾーンのロケール文字列を描画する
        point = CGPointMake(boundsX + LEFT_COLUMN_OFFSET, UPPER_ROW_TOP);
        [timeZoneWrapper.timeZoneLocaleName drawAtPoint:point
forWidth:LEFT_COLUMN_WIDTH withFont:mainFont minFontSize:MIN_MAIN_FONT_SIZE
actualFontSize:NULL lineBreakMode:UILineBreakModeTailTruncation
baselineAdjustment:UIBaselineAdjustmentAlignBaselines];

        // ... その他の文字列をここで描画する...

        // サブキストを描画する
        [secondaryTextColor set];

        // タイムゾーンの略称を描画する
        point = CGPointMake(boundsX + LEFT_COLUMN_OFFSET, LOWER_ROW_TOP);
        [abbreviation drawAtPoint:point forWidth:LEFT_COLUMN_WIDTH
withFont:secondaryFont minFontSize:MIN_SECONDARY_FONT_SIZE actualFontSize:NULL
 lineBreakMode:UILineBreakModeTailTruncation
baselineAdjustment:UIBaselineAdjustmentAlignBaselines];

        // ... その他の文字列をここで描画する...

        // 月の弦の画像を描画する
        CGFloat imageY = (contentRect.size.height -
timeZoneWrapper.image.size.height) / 2;
        point = CGPointMake(boundsX + RIGHT_COLUMN_OFFSET, imageY);
        [timeZoneWrapper.image drawAtPoint:point];
    }
```

```
}
```

このdrawRect:の実装の重要な部分は、セルのTable Viewが編集モードかをチェックするifステートメントです。ビューがセルのコンテンツを描画するのは、そのセルが編集モードではない場合のみです。必要であれば、編集モードの場合にセルを描画するelse句を、このステートメントに追加することもできます。編集モードでは、セルのコンテンツ領域が狭いため、フィールドを移動したり、フォントサイズを小さくしたり、重要度の低いフィールドを省略したりするなどの処理を行う必要があります。ただし、すでに述べたように、セルが編集モードに入るときや編集モードから出るときのカスタム描画は、パフォーマンスに重大な影響を与えるため、編集モードでの描画はお勧めできません。

最後に、データソースは、リスト5-14に示すように、tableView:cellForRowAtIndexPath:メソッドで、カスタムセルを持つTable Viewを提供します。セルコンテンツに対して、Table Viewの該当するセクションと行に対応するTimeZoneWrapperオブジェクトを見つけて設定します。

リスト 5-14　カスタムTable View Cellの初期化済みインスタンスを返す

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath  {
    static NSString *CellIdentifier = @"TimeZoneCell";

    TimeZoneCell *timeZoneCell = (TimeZoneCell *)[tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (timeZoneCell == nil) {
        timeZoneCell = [[[TimeZoneCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]
autorelease];
        timeZoneCell.frame = CGRectMake(0.0, 0.0, 320.0, ROW_HEIGHT);
    }
    Region *region = [displayList objectAtIndex:indexPath.section];
    NSArray *regionTimeZones = region.timeZoneWrappers;
    [timeZoneCell setTimeZoneWrapper:[regionTimeZones
objectAtIndex:indexPath.row]];
    return timeZoneCell;
}
```

UITableViewCellのサブクラスでprepareForReuseメソッドをオーバーライドして、セルオブジェクトの属性をリセットすることもできます。Table Viewは、dequeueReusableCellWithIdentifier:で、データソースにセルオブジェクトを返す直前にこのメソッドを呼び出します。パフォーマンスのためには、コンテンツに関係のないセル属性（alpha、editing、selection stateなど）だけをリセットするべきです。

セルとTable Viewのパフォーマンス

既成のセルオブジェクトでも、カスタムセルオブジェクトでも、Table View Cellを適切に使用することは、Table Viewのパフォーマンスにおいて重要な要素です。アプリケーションでは、次の3つのことを必ず実行してください。

- **セルを再利用する。**オブジェクトの割り当ては、パフォーマンスに影響します。特に、短期間に繰り返して割り当てを行わなければならない場合（ユーザがTable Viewをスクロールする場合など）は影響が大きくなります。セルを新規に割り当てる代わりに、セルを再利用すると、Table Viewのパフォーマンスを大幅に向上させることができます。

- **コンテンツをレイアウトし直すのは避ける。**カスタムサブビューを持つセルを再利用する場合は、Table Viewがセルを要求するたびに、これらのサブビューをレイアウトし直すのは避けてください。サブビューのレイアウトは、セルを作成した時に一度だけ行います。
- **不透明なサブビューを使用する。**Table View Cellをカスタマイズする場合は、セルのサブビューを透明ではなく不透明にします。

選択の管理

ユーザがTable Viewの行をタップすると、通常は、その結果として何らかのアクションが発生します。別のTable Viewがスライドしてきて画面に表示されたり、その行にチェックマークが表示されたり、その他のアクションが実行されます。以降の各セクションでは、選択への応答方法とプログラムで選択を行う方法について説明します。

Table Viewでの選択

Table Viewでのセルの選択を扱う場合は、次のようなヒューマンインターフェイスガイドラインに留意する必要があります。

- 状態を表すために選択を使用してはいけません。代わりに、状態を表すためにチェックマークやアクセサリビューを使用します。
- ユーザがセルを選択したときは、それに応答して、すでに選択されているセルを選択解除する（`deselectRowAtIndexPath:animated:`メソッドを呼び出す）とともに、任意の適切なアクション（詳細ビューの表示など）を実行する必要があります。
- セルの選択に応答して、新規のView ControllerをNavigation Controllerのスタックにプッシュした場合は、そのView Controllerがスタックからポップされたときに、セルを（アニメーションを伴って）選択解除します。

`UITableView`の`allowsSelectionDuringEditing`プロパティを設定することによって、Table Viewが編集モードになっているときに、行を選択可能にするかどうかを制御できます。さらに、iPhone OS 3.0からは、`allowsSelection`プロパティを設定することによって、編集モードでないときに、セルを選択可能にするかどうかも制御できます。

選択への応答

ユーザは、Table View内の行をタップして、その行が示す内容をもっと詳細に知りたいということをアプリケーションに伝えたり、その行が示す内容を選択したりします。ユーザが行をタップすると、それに応答して、アプリケーションは以下のいずれかを実行します。

- データモデル階層の次のレベルを表示する。
- 項目の詳細ビュー（データモデル階層のリーフノード）を表示する。
- 行にチェックマークを表示して、その項目が選択されたことを示す。
- 行に埋め込まれたコントロール内でタッチが発生した場合は、そのコントロールによって送信されたアクションメッセージに応答する。

行の選択のほとんどのケースに対処するために、Table Viewのデリゲートは、`tableView:didSelectRowAtIndexPath:`メソッドを実装しなければなりません。リスト 6-1で示すメソッドの実装例では、デリゲートは、まず選択中の行の選択を解除します。その後、シーケンス内の次のTable View Controllerのインスタンスを割り当てて初期化します。メソッドはTable Viewを埋めるためにこのView Controllerに必要なデータを設定し、このオブジェクトをアプリケーションの`UINavigationController`オブジェクトによって管理されているスタックにプッシュします。

リスト 6-1　行の選択に応答する

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
 *)indexPath
 {
     [tableView deselectRowAtIndexPath:indexPath animated:NO];
    BATTrailsViewController *trailsController = [[BATTrailsViewController alloc]
 initWithStyle:UITableViewStylePlain];
     trailsController.selectedRegion = [regions objectAtIndex:indexPath.row];
     [[self navigationController] pushViewController:trailsController
animated:YES];
     [trailsController release];
}
```

行がアクセサリビューとしてディスクロージャコントロール（青い丸の中に白い山形記号）を備えている場合、そのコントロールがクリックされると、デリゲートは（`tableView:didSelectRowAtIndexPath:`ではなく）`tableView:accessoryButtonTappedForRowWithIndexPath:`メッセージを受け取ります。デリゲートは、その他の種類の選択に応答する場合と同様に、このメッセージに応答します。

行が、アクセサリビューにコントロールオブジェクト（スイッチ、スライダなど）を持つ場合もあります。このコントロールオブジェクトは、ほかのコンテキストにおける場合と同様に動作します。つまり、このオブジェクトを適切に操作すると、アクションメッセージがターゲットオブジェクトに送信されます。リスト 6-2は、セルのアクセサリビューとして`UISwitch`オブジェクトを追加し、そのスイッチが"切り替えられた"ときに送信されたアクションメッセージに応答するデータソースオブジェクトを示しています。

リスト 6-2　スイッチオブジェクトをアクセサリビューに設定してアクションメッセージに応答する

```
- (UITableViewCell *)tableView:(UITableView *)theTableView
}
       UITableViewCell *cell = [tv
dequeueReusableCellWithIdentifier:@"CellWithSwitch"];
        if (cell == nil) {
            cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"CellWithSwitch"]
autorelease];
            cell.selectionStyle = UITableViewCellSelectionStyleNone;
            cell.textLabel.font = [UIFont systemFontOfSize:14];
        }
        UISwitch *switchObj = [[UISwitch alloc] initWithFrame:CGRectMake(1.0,
1.0, 20.0, 20.0)];
        switchObj.on = YES;
        [switchObj addTarget:self
action:@selector(toggleSoundEffects:)forControlEvents:(UIControlEventValueChanged
 | UIControlEventTouchDragInside)];
        cell.accessoryView = switchObj;
        [switchObj release];
```

```
        cell.textLabel.text = @"Sound Effects";
        return cell;

- (void)toggleSoundEffects:(id)sender {
    [self.soundEffectsOn = [(UISwitch *)sender isOn];
    [self reset];
}
```

Interface Builderで作成したTable View Cellのアクセサリビューにコントロールを定義することもできます。コントロールオブジェクト（スイッチ、スライダなど）をTable View Cellを含むnibドキュメントウインドウにドラッグします。次に、接続ウインドウを使用して、そのコントロールをセルのアクセサリビューに設定します。「カスタムTable View Cellをnibファイルからロードする」（56 ページ）では、nibファイルでTable View Cellオブジェクトを作成したり設定したりする手順について説明しています。

選択リストの場合は、選択の管理も重要です。次の2種類の選択リストがあります。

- 1つの行にのみチェックマークを付けることができる排他リスト
- 複数の行にチェックマークを付けることができる包含リスト

リスト 6-3に、排他的な選択リストを管理するためのアプローチの1つを示します。最初に、現在選択されている行を選択解除し、その行と同じ行が選択された場合には戻ります。それ以外の場合には、新たに選択された行にチェックマークアクセサリタイプを設定し、以前選択されていた行のチェックマークを削除します。

リスト 6-3　選択リストの管理—排他リスト

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
 *)indexPath {

    [tableView deselectRowAtIndexPath:indexPath animated:NO];
    NSInteger catIndex = [taskCategories indexOfObject:self.currentCategory];
    if (catIndex == indexPath.row) {
        return;
    }
    NSIndexPath *oldIndexPath = [NSIndexPath indexPathForRow:catIndex
inSection:0];

    UITableViewCell *newCell = [tableView cellForRowAtIndexPath:indexPath];
    if (newCell.accessoryType == UITableViewCellAccessoryNone) {
        newCell.accessoryType = UITableViewCellAccessoryCheckmark;
        self.currentCategory = [taskCategories objectAtIndex:indexPath.row];
    }

    UITableViewCell *oldCell = [tableView cellForRowAtIndexPath:oldIndexPath];
    if (oldCell.accessoryType == UITableViewCellAccessoryCheckmark) {
        oldCell.accessoryType = UITableViewCellAccessoryNone;
    }
}
```

リスト 6-4は、包含的な選択リストを管理する方法を示しています。この例のコメントにあるように、デリゲートが行に対してチェックマークの追加や削除を行うと、通常は、それに関連するModelオブジェクトの属性も設定または設定解除します。

リスト 6-4　選択リストの管理—包含リスト

```
- (void)tableView:(UITableView *)theTableView
          didSelectRowAtIndexPath:(NSIndexPath *)newIndexPath {

    [theTableView deselectRowAtIndexPath:[theTableView indexPathForSelectedRow]
 animated:NO];
    UITableViewCell *cell = [theTableView cellForRowAtIndexPath:newIndexPath];
    if (cell.accessoryType == UITableViewCellAccessoryNone) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
        // データモデルに選択を反映する
    } else if (cell.accessoryType == UITableViewCellAccessoryCheckmark) {
        cell.accessoryType = UITableViewCellAccessoryNone;
        // データモデルに選択解除を反映する
    }
}
```

`tableView:didSelectRowAtIndexPath:`では、現在選択されている行を必ず選択解除する必要があります。

プログラムによる選択とスクロール

TableView内のタップではなく、アプリケーションの内部から行の選択が生じる場合もあります。外部要因によってデータモデルに変更が生じることもあります。たとえば、ユーザがアドレスブックに新しい人を追加してから、連絡先リストに戻った場合に、アプリケーションは、最後に追加された人の位置まで連絡先リストをスクロールするのが理想です。このような場合は、`UITableView`の`selectRowAtIndexPath:animated:scrollPosition:`メソッドと（その行が既に選択されている場合は）`scrollToNearestSelectedRowAtScrollPosition:animated:`メソッドを使用します。また、選択されていない特定の行までスクロールする場合は、`scrollToRowAtIndexPath:atScrollPosition:animated:`を呼び出します。

リスト 6-5のコードは、プログラムによって適当な行を選択し、`selectRowAtIndexPath:animated:scrollPosition:`メソッドを使用して、今選択した行から20行離れた行までスクロールします。

リスト 6-5　プログラムによる行の選択

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
 *)newIndexPath {
    NSIndexPath *scrollIndexPath;
    if (newIndexPath.row + 20 < [timeZoneNames count]) {
        scrollIndexPath = [NSIndexPath indexPathForRow:newIndexPath.row+20
inSection:newIndexPath.section];
    } else {
        scrollIndexPath = [NSIndexPath indexPathForRow:newIndexPath.row-20
inSection:newIndexPath.section];
    }
    [theTableView selectRowAtIndexPath:scrollIndexPath animated:YES
                        scrollPosition:UITableViewScrollPositionMiddle];
}
```

行やセクションの挿入と削除

Table Viewには、通常（選択）モードのほかに、編集モードがあります。Table Viewが編集モードに入ると、行に関連付けられた編集用コントロールや並べ替えコントロールが表示されます。編集用コントロールは、行の左端に表示されます。ユーザは、これを利用して、Table Viewの行を挿入したり削除したりできます。編集用コントロールは、次のような特徴的な外観を持っています。

	削除コントロール
	挿入コントロール

Table Viewが編集モードに入って、ユーザが編集用コントロールをクリックすると、Table Viewは、データソースとデリゲートに一連のメッセージを送信します（ただし、これらのメソッドが実装されている場合のみ）。これらのメソッドによって、データソースとデリゲートは、Table Viewの行の外観と動作を改良できます。また、このメッセージによって、削除操作と挿入操作が可能になります。

Table Viewが編集モードでない場合も、複数の行やセクションをまとめて挿入したり削除したりすることができます。また、これらの操作をアニメーション化できます。

最初のセクションでは、テーブルが編集モードのときに、ユーザのアクションに応答してTable Viewに新しい行を挿入したり、Table View内の既存の行を削除したりする方法を示します。2つ目のセクション「行やセクションの一括挿入と一括削除」（78 ページ）では、複数のセクションや行をまとめてアニメーション化して挿入したり削除したりする方法について説明します。

注： 編集モードでの行の並べ替えの手順については、「行の並べ替えの管理」（81 ページ）で説明します。

編集モードでの行の挿入と削除

Table Viewが編集モードになるタイミング

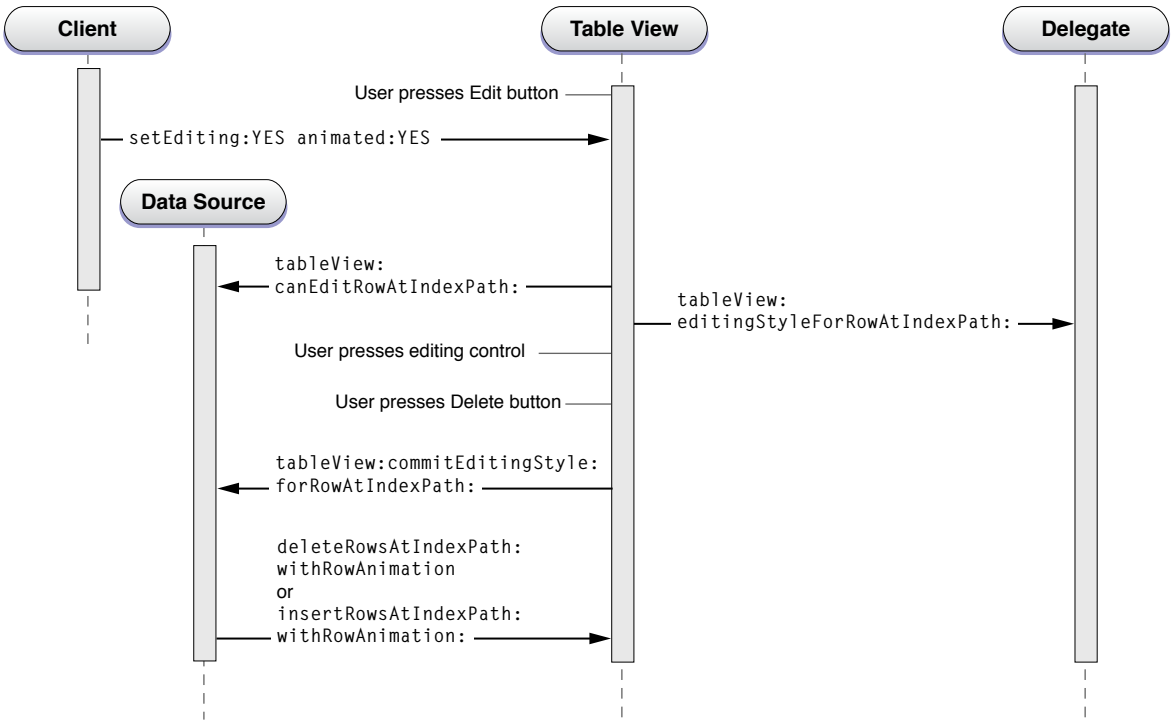
Table Viewは、`setEditing:animated:`メッセージを受信すると、編集モードに入ります。通常（必ずしもそうでない場合もある）、このメッセージは、ユーザがNavigation Barの「編集(Edit)」ボタンをタップしたときに送信されるアクションメッセージとして発生します。編集モードでは、Table

Viewには、デリゲートによって各行に割り当てられた編集用（および並べ替え）コントロールが表示されます。デリゲートは、`tableView:editingStyleForRowAtIndexPath:`メソッドで行の編集スタイルを返すことによって、このコントロールを割り当てます。

注： `UIViewController`オブジェクトがTable Viewを管理している場合は、「編集(Edit)」ボタンがタップされると、Table Viewは自動的に`setEditing:animated:`メッセージを受信します。このメッセージの実装内で、Table Viewバージョンのメソッドを呼び出す前に、ボタンの状態を更新したり、その他のタスクを実行できます。

Table Viewは、`setEditing:animated:`を受信すると、表示中の各行の`UITableViewCell`オブジェクトにそれと同じメッセージを送信します。次に、データソースとデリゲートに一連のメッセージを送信します（ただし、これらのメソッドが実装されている場合）。図 7-1は、その様子を示しています。

図 7-1 Table Viewでの行の挿入または削除の呼び出しシーケンス



表示中の行に対応するセルに`setEditing:animated:`が再送信された後は、次のような順序でメッセージのやり取りが行われます。

1. Table Viewは、`tableView:canEditRowAtIndexPath:`メソッドを呼び出します（データソースがこのメソッドを実装している場合）。このメソッドを利用して、アプリケーションは、セルの`editingStyle`プロパティに関係なく、このTable View内の行を編集できないようにすることができます。ほとんどのアプリケーションでは、このメソッドを実装する必要はありません。

2. Table Viewは、`tableView:editingStyleForRowAtIndexPath:`メソッドを呼び出します（デリゲートがこのメソッドを実装している場合）。このメソッドを利用して、アプリケーションは、行の編集スタイルとその行に表示する編集用コントロールを指定できます。

この時点で、このTable Viewは完全に編集モードになります。該当する各行には、挿入コントロールまたは削除コントロールが表示されます。

3. ユーザが編集用コントロール（挿入コントロールまたは削除コントロールのいずれか）をタップします。ユーザが削除コントロールをタップした場合は、その行に「削除(Delete)」ボタンが表示されます。ユーザはこのボタンをタップして、削除を確定します。

4. Table Viewは、`tableView:commitEditingStyle:forRowAtIndexPath:`メッセージをデータソースに送信します。このプロトコルメソッドの実装は任意となっていますが、行の挿入または削除を行う場合、データソースはこれを実装しなければなりません。その際、次の2つの処理を行う必要があります。

    - Table Viewに`deleteRowsAtIndexPaths:withRowAnimation:`または`insertRowsAtIndexPaths:withRowAnimation:`を送信して、表示を変更するように指示します。
    - 参照先の項目を配列から削除するか、または項目を配列に追加することによって、対応するデータモデル配列を更新します。

ユーザが行をスワイプして、その行に「削除(Delete)」ボタンを表示させた場合は、図 7-1 （74 ページ）に示した呼び出しシーケンスが一部変更になります。ユーザが行をスワイプして削除する場合は、Table Viewはまずデータソースが`tableView:commitEditingStyle:forRowAtIndexPath:`メソッドを実装しているかどうかを確認します。実装している場合は、Table View自身に`setEditing:animated:`を送信し、編集モードに入ります。この“スワイプによる削除”モードでは、Table Viewには編集用コントロールや並べ替えコントロールは表示されません。これはユーザ駆動のイベントなので、`tableView:willBeginEditingRowAtIndexPath:`と`tableView:didEndEditingRowAtIndexPath:`の2つのメッセージの間にデリゲートへのメッセージを挟み込みます。これらのメソッドを実装することによって、デリゲートは、Table Viewの外観を適切に更新できます。

注： データソースは、`tableView:commitEditingStyle:forRowAtIndexPath:`の実装内から`setEditing:animated:`を呼び出すべきではありません。何らかの理由でそれを行わなければならない場合は、`performSelector:withObject:afterDelay:`メソッドを使用して、一定の遅延の後にそれを呼び出します。

Table Viewに新規の行を挿入するために、挿入コントロールをトリガとして使用することもできますが、別の方法として、Navigation Barに「追加(Add)」（プラス記号）ボタンを持たせることもできます。このボタンをタップすると、View Cotrollerにアクションメッセージが送信されて、新規の項目を入力するためのモーダルビューがTable Viewの手前に表示されます。項目の入力が完了したら、コントローラはそれをデータモデル配列に追加して、テーブルを再ロードします。このアプローチについては、「Table Viewの行を追加する例」 （77 ページ）で説明します。

Table Viewの行を削除する例

このセクションでは、Table Viewを編集モード用に準備し、そこから行を削除するためのプロジェクトの一部を紹介します。このプロジェクトは、Navigation ControllerとView Controllerのアーキテクチャを使用して、Table Viewを管理します。`loadView`メソッドで、カスタムView Controllerは、Table Viewを作成して、自分自身をデータソースとデリゲートとして設定します。また、Navigation Barの右端の項目を標準の「編集(Edit)」ボタンとして設定します。

```
self.navigationItem.rightBarButtonItem = self.editButtonItem;
```

このボタンは、タップされたときにView Controllerに`setEditing:animated:`を送信するように、あらかじめ設定されています。ボタンをタップするたびに、ボタンのタイトルが「編集(Edit)」と「完了(Done)」の間で交互に切り替わります。また、Booleanの*`editing`*パラメータも切り替わります。このメソッドの実装では、リスト 7-1に示すように、View Controllerは、このメソッドのスーパークラスの実装を呼び出して、Table Viewに同じメッセージを送信します。次に、Navigation Barのもう一方のボタン（項目を追加するためのプラス記号のボタン）のenabledステータスを更新します。

リスト 7-1　`setEditing:animated:`に応答するView Controller

```
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {
    [super setEditing:editing animated:animated];
    [tableView setEditing:editing animated:YES];
    if (editing) {
        addButton.enabled = NO;
    } else {
        addButton.enabled = YES;
    }
}
```

Table Viewが編集モードに入ると、View Controllerは、最後の行以外のすべての行に削除コントロールを設定します。最後の行は挿入コントロールを持ちます。この処理は、`tableView:editingStyleForRowAtIndexPath:`メソッド（リスト 7-2）の実装内で行います。

リスト 7-2　行の編集スタイルのカスタマイズ

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
    SimpleEditableListAppDelegate *controller = (SimpleEditableListAppDelegate
 *)[[UIApplication sharedApplication] delegate];
    if (indexPath.row == [controller countOfList]-1) {
        return UITableViewCellEditingStyleInsert;
    } else {
        return UITableViewCellEditingStyleDelete;
    }
}
```

ユーザが行内の削除コントロールをタップすると、View Controllerは、Table Viewから`tableView:commitEditingStyle:forRowAtIndexPath:`メッセージを受信します。リスト 7-3に示すように、View Controllerは、その行に対応する項目をモデル配列から削除して、Table View に`deleteRowsAtIndexPaths:withRowAnimation:`を送信することによって、このメッセージを処理します。

リスト 7-3　データモデル配列の更新と行の削除

```
- (void)tableView:(UITableView *)tv
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    // 行が削除された場合は、それをリストから取り除く
    if (editingStyle == UITableViewCellEditingStyleDelete) {
       SimpleEditableListAppDelegate *controller = (SimpleEditableListAppDelegate
 *)[[UIApplication sharedApplication] delegate];
        [controller removeObjectFromListAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

Table Viewの行を追加する例

このセクションでは、Table Viewに行を挿入するプロジェクトのコードを示します。行を挿入するためのトリガとして挿入コントロールを使用する代わりに、Table Viewの上に表示されるNavigation Barの「追加(Add)」ボタン（プラス記号として表示される）を使用します。このコードも、Navigation ControllerとView Controllerのアーキテクチャに基づいています。View Controllerは、`loadView`メソッドの実装で、リスト 7-4に示すコードを使用して、Navigation Barの右端の項目に「追加(Add)」ボタンを割り当てます。

リスト 7-4　Navigation Barに「追加(Add)」ボタンを追加する

```
    addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(addItem:)];
    self.navigationItem.rightBarButtonItem = addButton;
```

View Controllerは、タイトルの制御状態のほか、アクションセレクタとターゲットオブジェクトの制御状態も設定します。ユーザが「追加(Add)」ボタンをタップすると、`addItem:`メッセージがターゲット(View Controller)に送信されます。View Controllerは、リスト 7-5に示すように応答します。View Controllerは、1つのNavigation Controllerを作成します。このコントローラは、画面上にモーダルモードで表示される（上方向に移動するようにアニメーション化されTable Viewの手前に表示される）ビューを持つView Controllerを含んでいます。`presentModalViewController:animated:`メソッドがこれを実行します。

リスト 7-5　「追加(Add)」ボタンのタップに応答する

```
- (void)addItem:sender {
    if (itemInputController == nil) {
        itemInputController = [[ItemInputController alloc] init];
    }
    UINavigationController *navigationController = [[UINavigationController
alloc] initWithRootViewController:itemInputController];
   [[self navigationController] presentModalViewController:navigationController
 animated:YES];
    [navigationController release];
}
```

モーダル（オーバーレイ）ビューは、1つのカスタムテキストフィールドで構成されています。ユーザは、新規のTable View項目用のテキストを入力して、「保存(Save)」ボタンをタップします。このボタンは、`save:`アクションメッセージをターゲット（このモーダルビューのView Controller）に送信します。リスト 7-6に示すように、View Controllerはテキストフィールドから文字列値を取得し、それを使用してアプリケーションのデータモデル配列を更新します。

リスト 7-6　データモデル配列に新規項目を追加する

```
- (void)save:sender {

    UITextField *textField = [(EditableTableViewTextField *)[tableView
cellForRowAtIndexPath:[NSIndexPath indexPathForRow:0 inSection:0]] textField];

    SimpleEditableListAppDelegate *controller = (SimpleEditableListAppDelegate
 *)[[UIApplication sharedApplication] delegate];
    NSString *newItem = textField.text;
    if (newItem != nil) {
```

```
            [controller insertObject:newItem inListAtIndex:[controller
countOfList]];
    }
    [self dismissModalViewControllerAnimated:YES];
}
```

モーダルビューが閉じられると、Table Viewが再ロードされ、追加された項目がTable Viewに反映されます。

行やセクションの一括挿入と一括削除

UITableViewクラスを利用すると、行やセクションを一度にまとめて挿入したり削除したりできます。同時に、指定した方法でこの操作をアニメーション化できます。リスト 7-7に示す6つのメソッドは、一括挿入と一括削除に関係するものです。これらの挿入メソッドと削除メソッドは、（「編集モードでの行の挿入と削除」（73 ページ）で解説したデータソースメソッドtableView:commitEditingStyle:forRowAtIndexPath:と同様に）アニメーションブロックの外側で呼び出すことができます。

リスト 7-7　一括挿入と一括削除のメソッド

```
- (void)beginUpdates;
- (void)endUpdates;

- (void)insertSections:(NSIndexSet *)sections
withRowAnimation:(UITableViewRowAnimation)animation;
- (void)deleteSections:(NSIndexSet *)sections
withRowAnimation:(UITableViewRowAnimation)animation;

- (void)insertRowsAtIndexPaths:(NSArray *)indexPaths
withRowAnimation:(UITableViewRowAnimation)animation;
- (void)deleteRowsAtIndexPaths:(NSArray *)indexPaths
withRowAnimation:(UITableViewRowAnimation)animation;
```

行やセクションの一括挿入と一括削除をアニメーション化するには、連続するbeginUpdates呼び出しとendUpdates呼び出しで定義されるアニメーションブロック内で、挿入メソッドや削除メソッドを呼び出します。このブロック内で挿入メソッドや削除メソッドを呼び出さないと、行やセクションのインデックスが無効になる場合があります。beginUpdates…endUpdatesまでのブロックはネストできません。

ブロックの最後（つまり、endUpdatesが戻った後）で、Table Viewは通常どおりデータソースとデリゲートに行とセクションのデータを要求します。したがって、Table Viewの基になるコレクションオブジェクトは、挿入または削除された行やセクションを反映するように更新されなければなりません。

iPhone OS 3.0で導入されたreloadSections:withRowAnimation:メソッドとreloadRowsAtIndexPaths:withRowAnimation:メソッドは、上で説明したメソッドに関連しています。これらを利用すれば、表示するTable View全体をロードするreloadDataを呼び出すのではなく、代わりに、特定のセクションや行のデータを再ロードするようにTable Viewに要求できます。

一括挿入と一括削除の操作例

TableView内の一まとまりの行やセクションを挿入したり削除したりするには、まず、そのセクションや行のデータソースとなる配列（複数の場合もある）を用意します。行やセクションを削除したり挿入したりすると、変更後の行やセクションがこのデータソースから読み込まれます。

次に、`beginUpdates`メソッドを呼び出してから、`insertRowsAtIndexPaths:withRowAnimation:`、`deleteRowsAtIndexPaths:withRowAnimation:`、`insertSections:withRowAnimation:`、または`deleteSections:withRowAnimation:`を呼び出します。`endUpdates`を呼び出して、アニメーションブロックを終了します。リスト 7-8に、この処理を示します。

リスト 7-8　Table Viewでの1ブロックの行の挿入と削除

```
- (IBAction)insertAndDeleteRows:(id)sender {
    // 元の行：Arizona、California、Delaware、New Jersey、Washington

    [states removeObjectAtIndex:4]; // Washington
    [states removeObjectAtIndex:2]; // Delaware
    [states insertObject:@"Alaska" atIndex:0];
    [states insertObject:@"Georgia" atIndex:3];
    [states insertObject:@"Virginia" atIndex:5];

    NSArray *deleteIndexPaths = [NSArray arrayWithObjects:
                                 [NSIndexPath indexPathForRow:2 inSection:0],
                                 [NSIndexPath indexPathForRow:4 inSection:0],
                                 nil];
    NSArray *insertIndexPaths = [NSArray arrayWithObjects:
                                 [NSIndexPath indexPathForRow:0 inSection:0],
                                 [NSIndexPath indexPathForRow:3 inSection:0],
                                 [NSIndexPath indexPathForRow:5 inSection:0],
                                 nil];
    UITableView *tv = (UITableView *)self.view;

    [tv beginUpdates];
    [tv insertRowsAtIndexPaths:insertIndexPaths
withRowAnimation:UITableViewRowAnimationRight];
    [tv deleteRowsAtIndexPaths:deleteIndexPaths
withRowAnimation:UITableViewRowAnimationFade];
    [tv endUpdates];

    // 最終的な行：Alaska、Arizona、California、Georgia、New Jersey、Virginia
}
```

この例では、配列（およびそれに対応する行）から2つの文字列を削除し、3つの文字列を配列（およびそれに対応する行）に挿入しています。次のセクションの「操作の順序とインデックスパス」では、行（またはセクション）の挿入動作と削除動作の特異な点について説明します。

操作の順序とインデックスパス

リスト7-8に示したコードには、奇妙に思われる点があることに気づいたかもしれません。このコードでは、`insertRowsAtIndexPaths:withRowAnimation:`を呼び出した後に`deleteRowsAtIndexPaths:withRowAnimation:`メソッドを呼び出しています。しかし、これは、

UITableViewがこの操作を完了する順序ではありません。行やセクションの挿入は、行やセクションの削除処理が完了するまで延期されます。これは、挿入メソッドと削除メソッドの呼び出し順序に関係なく起こります。

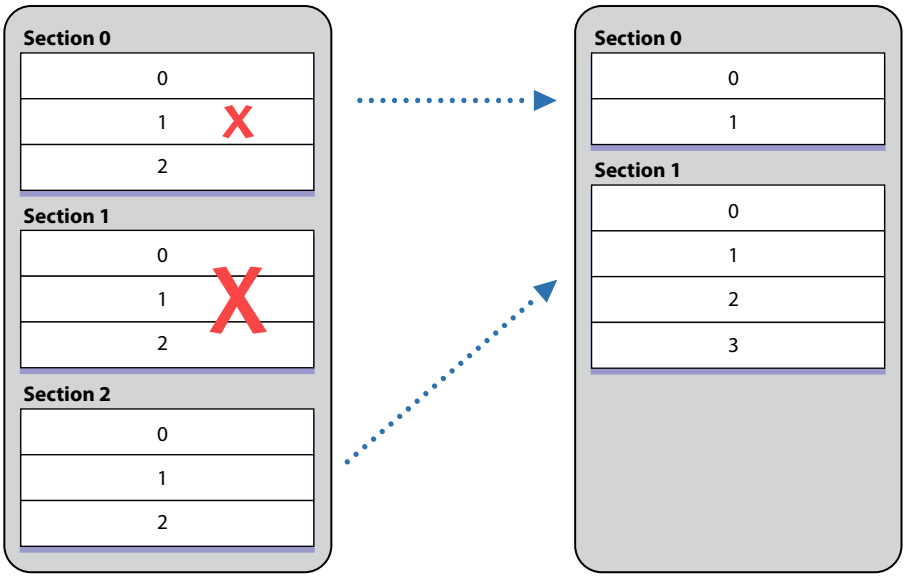
アニメーションブロック内での削除は、元のテーブルのどの行やセクションを削除すべきかを指定します。挿入は、削除後のテーブルにどの行やセクションを追加すべきかを指定します。セクションや行を指定するために使われるインデックスパスは、このモデルに従います。一方、可変配列で1つの項目を挿入または削除すると、その後の挿入操作や削除操作に使われる配列インデックスに影響します。たとえば、あるインデックス位置に項目を1つ挿入すると、配列内のそれ以降のすべての項目のインデックスがインクリメントされます。

分かりやすいように、例を示します。3つのセクションを持つTable Viewを考えます。各セクションには3つの行が含まれています。ここで、次のようなアニメーションブロックを実装します。

1. 更新を開始する。
2. インデックス0のセクションのインデックス1の行を削除する。
3. インデックス1のセクションを削除する。
4. インデックス1のセクションのインデックス1に行を挿入する。
5. 更新を終了する。

図 7-2に、このアニメーションブロックが終了した後にどうなるかを示します。

図 7-2 セクションと行の削除と行の挿入



行の並べ替えの管理

Table Viewには、通常（選択）モードのほかに、編集モードがあります。Table Viewが編集モードに入ると、行に関連付けられた編集用コントロールや並べ替えコントロールが表示されます。並べ替えコントロールを利用すると、ユーザは、1つの行をテーブル内の別の場所に移動できます。図 8-1に示すように、並べ替えコントロールは行の右端に表示されます。

図 8-1　行の並べ替え

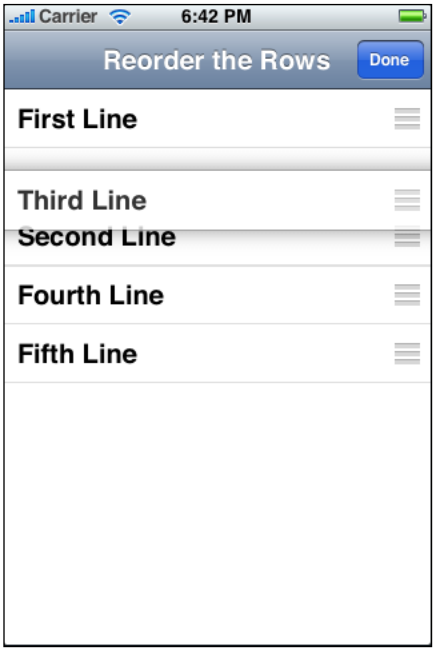


Table Viewが編集モードに入って、ユーザが並べ替えコントロールをドラッグすると、Table Viewは、データソースとデリゲートに一連のメッセージを送信します（ただし、これらのメソッドが実装されている場合のみ）。これらのメソッドを利用して、データソースとデリゲートは、実際の移動操作を実行するだけでなく、行を移動できるかどうかや、行を移動できる場所を制限することができます。以降の各セクションでは、Table View内で行を移動する方法を示します。

行を移動すると何が起きるか

Table Viewは、`setEditing:animated:`メッセージを受信すると、編集モードに入ります。通常（必ずしもそうでない場合もある）、このメッセージは、ユーザがNavigation Barの「編集(Edit)」ボタンをタップしたときに送信されるアクションメッセージとして発生します。編集モードのTable Viewは、デリゲートによって各行に割り当てられた並べ替え（および編集用）コントロールを表示しま

す。デリゲートは、tableView:cellForRowAtIndexPath:内で、UITableViewCellオブジェクトのshowsReorderControlプロパティをYESに設定することによって、これらのコントロールを割り当てます。

注： UIViewControllerオブジェクトがTable Viewを管理している場合は、「編集(Edit)」ボタンがタップされると、Table Viewは自動的にsetEditing:animated:メッセージを受信します。UITableViewController（UIViewControllerのサブクラス）は、このメソッドを実装して、ボタンの状態を更新したり、このメソッドのTable Viewにおける実装を呼び出します。UIViewControllerを使用してTable Viewを管理している場合は、これと同じ動作を実装する必要があります。

Table Viewは、setEditing:animated:を受信すると、それと同じメッセージを表示中の各行のUITableViewCellオブジェクトに送信します。次に、データソースとデリゲートに一連のメッセージを送信します（ただし、これらのメソッドが実装されている場合）。図 8-2は、その様子を示しています。

図 8-2 Table Viewでの行の並べ替えの呼び出しシーケンス

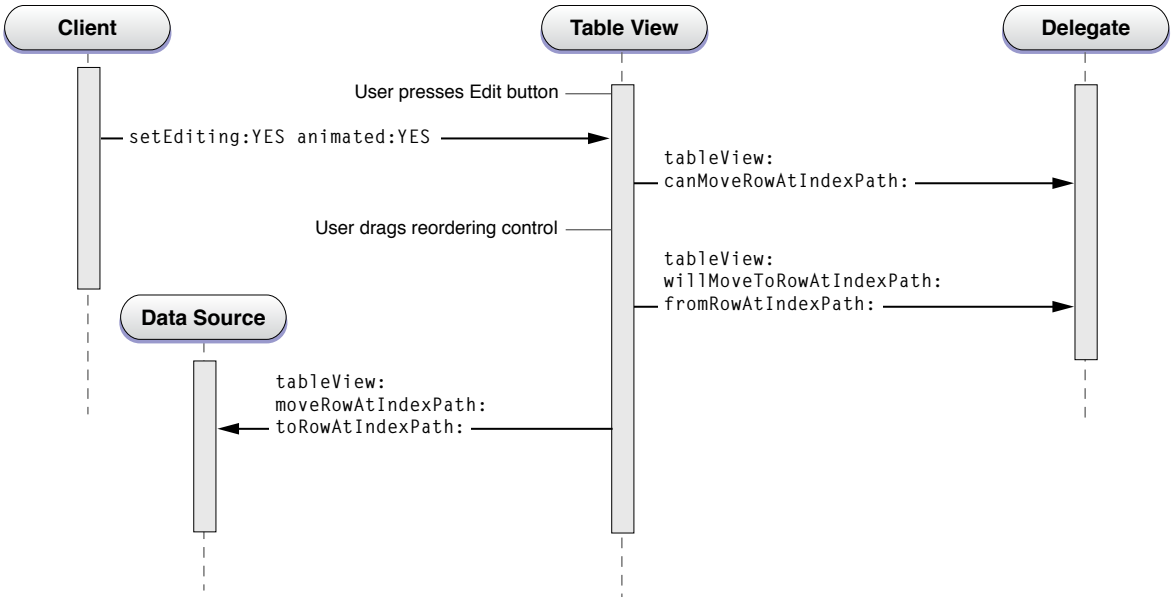


Table Viewは、setEditing:animated:メッセージを受信すると、表示中の行に対応するセルオブジェクトにそれと同じメッセージを再送信します。その後、次のような順序でメッセージのやり取りが行われます。

1. Table Viewは、データソースにtableView:canMoveRowAtIndexPath:メッセージを送信します（データソースがこのメソッドを実装している場合）。このメソッド内で、デリゲートは、特定の行に並べ替えコントロールを表示しないようにできます。

2. ユーザが、Table Viewの中で並べ替えコントロールを上または下に移動して、行をドラッグします。ドラッグ中の行がTable Viewの一部の上に覆いかぶさると、ドラッグ先を示すために、下に隠れた行が下方へスライドします。

3. ドラッグ中の行が移動先の上に覆いかぶさるたびに、Table Viewはデリゲートに`tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:`を送信します（デリゲートがこのメソッドを実装している場合）。このメソッド内で、デリゲートは、ドラッグ中の行の現在の移動先を拒否して、別の場所を指定することもできます。

4. Table Viewは、データソースに`tableView:moveRowAtIndexPath:toIndexPath:`メッセージを送信します（データソースがこのメソッドを実装している場合）。このメソッド内で、データソースは、Table Viewの項目の情報源となっているデータモデル配列を更新し、その項目を配列の別の場所に移動します。

行を移動する例

このセクションでは、「行を移動すると何が起こるか」（81 ページ）で列挙した並べ替えのステップを示すサンプルコードについて説明します。リスト 8-1に、Table Viewの最初の行を移動できないようにする`tableView:canMoveRowAtIndexPath:`の実装を示します（この行には、並べ替えコントロールが表示されません）。

リスト 8-1　行を移動できないようにする

```
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath
*)indexPath {
    if (indexPath.row == 0) // 最初の行は移動できない
      return NO;

   return YES;
}
```

ユーザが行のドラッグを終了すると、その行は、Table Viewの移動先に挿入されます。Table Viewは、データソースに`tableView:moveRowAtIndexPath:toIndexPath:`を送信します。リスト 8-2は、このメソッドの実装を示します。Table Viewは、配列から取得したデータ項目（移動した項目）を保持します。これは、データ項目を配列から削除すると、それが自動的に解放されるからです。

リスト 8-2　移動した行に対応するデータモデル配列の更新

```
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:(NSIndexPath
*)sourceIndexPath toIndexPath:(NSIndexPath *)destinationIndexPath {
    NSString *stringToMove = [[self.reorderingRows
objectAtIndex:sourceIndexPath.row] retain];
    [self.reorderingRows removeObjectAtIndex:sourceIndexPath.row];
    [self.reorderingRows insertObject:stringToMove
atIndex:destinationIndexPath.row];
    [stringToMove release];
}
```

デリゲートは、`tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:`メソッドを実装することによって、提案された移動先を、別の行に変更することもできます。次の例では、行の移動をグループ内に制限し、グループの最後の行（追加項目用のプレースホルダとして予約されている）への移動を禁止しています。

リスト 8-3　移動操作の移動先の行の変更

```
- (NSIndexPath *)tableView:(UITableView *)tableView
```

```
        targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath *)sourceIndexPath
         toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath {
    NSDictionary *section = [data objectAtIndex:sourceIndexPath.section];
    NSUInteger sectionCount = [[section valueForKey:@"content"] count];
    if (sourceIndexPath.section != proposedDestinationIndexPath.section) {
        NSUInteger rowInSourceSection =
             (sourceIndexPath.section > proposedDestinationIndexPath.section)
?
               0 : sectionCount - 1;
        return [NSIndexPath indexPathForRow:rowInSourceSection
inSection:sourceIndexPath.section];
    } else if (proposedDestinationIndexPath.row >= sectionCount) {
        return [NSIndexPath indexPathForRow:sectionCount - 1
inSection:sourceIndexPath.section];
    }
    // 提案された移動先を許可する
    return proposedDestinationIndexPath;
}
```

書類の改訂履歴

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

日付	メモ
2009-08-19	セルの背景色の設定方法。tableView:willDisplayCell:forRowAtIndexPath:の目的を強調。若干の訂正。
2009-05-28	3.0 API、特に、定義済みのセルスタイルとそれに関連するプロパティについて説明するための更新。Table ViewとTable View Cellに対してnibファイルを使用する方法も説明し、View Controllerとデザインパターン手法に関する章を更新。
2008-10-15	tableView:commitEditingStyle:forRowAtIndexPath:でのsetEditing:animated:の呼び出しに関する注意。図の更新。
2008-09-09	一括挿入と一括削除に関するセクションを追加。TOCフレームに関連するクラスを追加。選択のクリアに関するガイドラインを追加。若干の訂正。
2008-06-25	本書の初版。

書類の改訂履歴